# Data Structures and Algorithms

Alan Sexton[1]

[1] Thanks to many staff from the School of Computer Science, Uni of Birmingham, UK

# Programs = Algorithms + Data Structures

**Data Structures** efficiently organise data in computer memory.

**Algorithms** manipulate data structures to achieve a given goal.

In order for a program to terminate fast (or in time), it has to use *appropriate* data structures and *efficient* algorithms.

In this module we focus on:

- various data structures
- basic algorithms
- understanding the strengths and weaknesses of those, in terms of their time and space complexities

## Learning Outcomes

After completing this module, you should be able to:

- Design and implement data structures and algorithms
- Argue that algorithms are correct, and derive time and space complexity measures for them
- Explain and apply data structures in solving programming problems
- Make informed choices between alternative data structures, algorithms and implementations, justifying choices on grounds such as computational efficiency

# Abstract Data Types (ADT)

A *type* is

- a set of possible values
- with a set of allowed operations on those values

An *abstract data type (ADT)* is a type whose internal representation is hidden to the user.

Thus users of an abstract data type may have no information about how the ADT is implemented, but depends only on the published information about how it behaves.

This means that the implementation of an abstract type can be changed without having to change the code that uses it.

**Example**

"Integer" is an abstract data type consisting of integer values with operations `+`, `-`, `*`, `mod`, `div`, . . .

- A type is a collection of values, e.g. integers, Boolean values (true and false) with their operations.

- The operations on an ADT might come with mathematically specified constraints, for example on the time complexity of the operations.

- Advantages of ADT's as explained by Aho, Hopcroft and Ullman (1983):

  > *"At first, it may seem tedious writing procedures to govern all accesses to the underlying structures. However, if we discipline ourselves to writing programs in terms of the operations for manipulating abstract data types rather than making use of particular implementations details, then we can modify programs more readily by reimplementing the operations rather than searching all programs for places where we have made accesses to the underlying data structures. This flexibility can be particularly important in large software efforts, and the reader should not judge the concept by the necessarily tiny examples found in this book."*

## List is an ADT

An example of a list of numbers

$$\langle 2, 5, 1, 8, 23, 1 \rangle \qquad \text{(ordered collection of elements)}$$

List is an ADT; list operations may include:

- insert an entry (on a certain position)
- delete an entry
- access data by position
- search
- concatenate two lists
- sort
- ...

# Different Representations of Lists

Depending on what operations are needed for our application, we choose from different data structures (some implement certain operations faster than the others):

- Arrays
- Linked lists
- Dynamic arrays
- Unrolled linked lists
- . . .

We will study some of these in detail later in the module.

# Computer Memory to a Program

Data is stored, managed and manipulated in computer memory.

- The computer architecture (the hardware) and the operating system work together to allow each running program to see an illusion that they have all the memory on the computer to themselves.
- This memory is organised as a long list of memory cells, each 1 byte or 8 bits large. A bit can hold either a zero or a one.
- Every one of these 1 byte cells has an address, a number in the range 0 to the highest possible address for this system combination
- On older, 32 bit systems, this highest address is hexadecimal FFFFFFFF (i.e. 8 'F's), or 4,294,967,295 (i.e. $2^{32} - 1$)
- On newer, 64 bit systems, this is FFFFFFFFFFFFFFFF (i.e. 16 'F's), or approximately $1.844674407 \times 10^{19}$ (i.e. $2^{64} - 1$)
- For technical reasons, the hardware actually manipulates memory a whole word at a time, where is word is either 4 bytes (32 bits) or 8 bytes (64 bits)

# Memory Management

Of course, computers can not hold such gigantic amounts of memory, and if a program tried to access every one of those bytes of memory, it would reach addresses in memory where the illusion breaks down, and an error would be triggered.

To support the illusion of all this memory for a program, programs have to explicitly request the operating system to add large sections of memory to their memory space via *operating system calls*

However, programs themselves need to manage their own memory with much finer and more efficient control than these expensive operating system calls

So the *Runtime System*, a software library that is integral to each programming language, typically provides one of two options

1. Explicit Memory Management, with *allocate* and *free* or
2. Implicit Memory Management with Garbage Collection

# Explicit Memory Management

The programming languages C and C++ are examples of programming languages with explicit memory management.

C's runtime system maintains it's own data structure to organise the memory it has obtained from the operating system. This data structure is usually accessed by two functions:

1. *malloc*: allows the program to request a contiguous amount of memory. If available, this is recorded in the memory managment data structure, the address of the start of the block is returned to the program and the data structure is updated accordingly.

   If not available, a system call requests more memory from the operating system. If successful, it integrates the memory into its data structure and continues as before. If unsuccessful, it reports an error back to the program.
2. *free*: Given the address of a block that was previously allocated, marks it as available for future allocation

# Explicit Memory Management

Explicit Memory Management is a simple model and very efficient. However, it suffers from a number of disadvantages:

- The program needs to keep track of every block requested with *malloc* so that it can be freed later. Otherwise, the program will have *memory leaks* — these are blocks of allocated memory that can no longer be used or freed. This causes the program to use more memory than it needs (which causes performance problems), and may cause the program to crash if it eventually runs out of memory.
- If an error is made in giving an address to *free* that had not previously been returned by *malloc*, or if that address had already been freed, then the memory management data structure can be corrupted, causing unpredictable errors or crashes

# Implicit Memory Management

Implicit Memory Management is much more sophisticated. The programming language itself, through its runtime system, provides mechanisms to allocate data structures (without exposing memory addresses to programmers), and identifies allocated structures that can no longer be accessed by the running program and adds them back into its data structure of available free memory without requiring the program to specifically ask for the memory to be freed.

This mechanism is used by Java, Python, Functional programming languages like Haskell and OCaml, and many more.

It relieves the heavy burden of keeping track of allocated memory from the programmer, and avoids most of the bugs and problems that address manipulation in languages like C are famous for.

# Pseudocode

In this module we will be discussing details of algorithms. This necessitates using a programming language to capture the precise steps of the processing involved. However, standard programming languages require a lot of administrative detail that is necessary to get the program working on the computer but ends up obscuring the important aspects of the algorithm we wish to explain.

Instead we will use a simplified, less rigorous style of a programming language that will not run on any computer but makes clear to a human reader the steps involved. This is called *pseudocode*.

Throughout this module, when you are asked to write some pseudocode, you are free to use pseudocode, Java code or any mix between the two. In such cases any syntactic errors will not be penalised so long as the intent is clear to a human reader.

# Pseudocode

The core elements of a programming language like Java are

- Variables: named memory cells that can hold values of some type, e.g. Integers, Strings, Arrays, etc.
- Expressions: how to calculate new values
- Assignment statements: how to modify variables
- Sequences and Blocks: how to execute a sequence of steps and group them
- Conditionals: How to choose between different steps
- Loops: how to run steps multiple times
- Input/Output: How to write out or read in values
- Function definitions: How to combine steps into a function that can be executed from multiple locations in the code
- Function calls: How to invoke a function

## Pseudocode: Variables, Expressions

- For variables we use simple names with no spaces that start with a letter and can contain letters, digits and the underscore character "_". Examples include
    - "total", "name", "length", "account_holder_2"
- Expressions can be any meaningful (to humans) mathematical, logical or text expression using:
    - Arithmetic operations: ($+$, $-$, $\times, \div, *, /$ etc.)
        - (sum1 + sum2) / 2
    - Logical operations: (AND, OR, NOT, etc)
        - isRaining AND haveUmbrella
        - balance > 100 AND dayOfMonth == 1
    - Strings can be joined together using "+" as a string concatenation operator
        - "hello " + name
    - Extra operators from Java can be used.

# Pseudocode: Assignments

- Assignments put values (possibly calculated by expressions, into variables using a single “=” symbol
  - n = (25 + n) MOD 50
  - isRaining = TRUE
  - message = ”Hello ” + name
- When you first use a variable, you can specify its type if it is not clear from the context:
  - float sum = 0
- You should not use different types in the same variable at different times
- You should never use a variable in an expression whose value has not first been set

# Pseudocode: Sequences and Blocks of Statements

- To specify a sequence of statements, you can put them one per line, indented to the same level. Optionally, you can put a semicolon, “;”, at the end of each line:

  -
    ```
    a = a+10
    b = a*2-4
    ```
- To group them into a block, e.g. to be able to loop over them, you can surround them with braces, “{“, “}”, or with BEGIN, END:

  -
    ```
    BEGIN
          a = a+10
          b = a*2-4
    END
    ```
- **Always** indent the contents of the block correctly

## Pseudocode: Conditional Statements

- Here we use one of the forms (always separated onto different lines and indented:
  - IF condition THEN statement ENDIF
    - 
      ```
      IF balance < 0 THEN
              print("Your balance is overdrawn")
              print("You owe £", -balance )
      ENDIF
      ```
  - IF condition THEN statement1 ELSE statement2 ENDIF
  - IF condition1 THEN statement1 ELIF condition2 THEN statement2 ... ELSE statementN ENDIF
- Alternatively you can use a more Java-like syntax:
  - 
    ```
    if (balance < 0)
    {
        print("your balance is overdrawn")
        print("You owe £", -balance )
    }
    ```

## Pseudocode: Loops

- There are a number of forms for loops
  - WHILE condition DO statement ENDWHILE
  - DO statement WHILE condition
  - REPEAT statement UNTIL condition
  - FOR (initial statement; condition ; step) statement ENDFOR

    -
      ```
      sum = 0
      FOR (i = 0 ; i < 100 ; i=i+1)
          sum = sum + i
      ENDFOR
      ```
- These all have Java-like versions:

  -
    ```
    sum = 0
    for (i = 0 ; i < 100 ; i=i+1)
    {
        sum = sum + i
    }
    ```

## Pseudocode: Input/Output

- To output, use the function “print”, with as many arguments as you need
    - print(”answer is: ”, sum)
- To input, use the function “read”, assigning the return value to a variable. There is no need to specify precisely the details of the input format. The intention should be clear from the surrounding code and the name of the variable.
    - numberOfStudents = read()

# Pseudocode: Functions

- Function definitions specify a sequence of statements to be executed. Values can be passed in to the function by parameters, results can be returned from the function using a RETURN statement. You should specify the type of value that the function returns, or specify the type as VOID if it does not return anything (e.g. if it just prints values out).

  -
    ```
    int max(int a, int b)
    BEGIN
        IF a > b
            RETURN a
        RETURN b
    END
    ```

  -
    ```
    int max(int a, int b)
    {
        if (a > b)
            return a
        return b
    }
    ```

- Such a function can be called as follows:
  - maximum = max(myValue, 10)

# Arrays

# Arrays

An *array* is a data structure that is laid out in memory as a contiguous list of cells, with each cell indexed by the position of the cell in the structure.

Just like in Java, We will always index arrays with the first cell having index 0, and the last cell having index `len - 1`, where `len` is the number of cells in the array (i.e. the length of the array). Some programming languages (e.g. Fortran, R, Matlab) use a different policy of using 1 and `len` for the first and last cell respectively instead.

Cells do not need to be single bytes: the array is declared to contain some underlying type, such as Integers, Floating Point Numbers, Strings, etc. It is even possible to have an array of arrays of integers, where each cell contains a whole array of integers

## Array Operations

The basic array data type has very few operations.

- Array creation

```
int[] nums = new int[4]
```

- Getting values from cells in the array

```
val = nums[0]
```

- Assigning values to cells in the array

```
nums[1] = 23
```

- Getting the length of the array

```
len = length(nums)
```

More sophisticated List ADTs often use basic arrays in their implementation, but add more complex operations such as increasing the size of a List, Sorting a list, concatenating Lists etc.

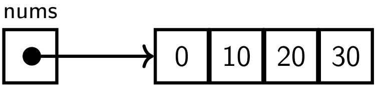
# List as an array of a fixed length

In pseudocode, you can create an array using Java-like syntax:

```
int[] nums = new int[4]
for (i=0, i<length(nums) ; i=i+1)
    nums[i] = i * 10
```

This can be described in a diagram such as the one below, and results in the memory layout shown to the right

| i | Memory | |
|---|---|---|
| ⋮ | ⋮ | |
| 3344 | 23 | |
| 3340 | 30271 | |
| 3336 | 30 | array<br>nums |
| 3332 | 20 | |
| 3328 | 10 | |
| 3324 | 0 | |
| 3320 | 6738 | |
| ⋮ | ⋮ | |
| nums:3100 | 3324 | |



Here we assume that the word size is 32 bits or 4 bytes, and integers (and memory pointers) are also 32 bits.

- `nums` is a variable whose cell in memory is at address 3100. The contents of this cell is a memory address, 3324, which is where the array starts

- Every `int` in the array occupies one word or 4 bytes in memory

- Because we declared our array to be an array of integers, the compiler knows that every entry in the array takes 4 bytes, and if the array starts at location 3324, then it knows that:
  - `nums[0]` is at address 3324,
  - `nums[1]` is at address $3324 + (1 \times 4)$,
  - `nums[2]` is at address $3324 + (2 \times 4)$, etc.

# More complicated arrays in memory

In its simplist form, a Java class collects variables together into a single structure

```
class Point {
    float x;
    float y;
}
Point[] locations = new Point[3];
locations[1].x = 25.2;
locations[1].y = 38.6;
```

| i | Memory |
|---|---|
| ⋮ | ⋮ |
| 4052 | 0.0 |
| 4048 | 0.0 |
| 4044 | 38.6 |
| 4040 | 25.2 |
| 4036 | 0.0 |
| 4032 | 0.0 |
| ⋮ | ⋮ |
| locations:3100 | 4032 |

Given that a float is 4 bytes, the following is what happens in memory:

```
Cell at address locations+1*2*4+0 is set to 25.2;
Cell at address locations+1*2*4+1 is set to 38.6;
```

In the 1*2*4: the 1 is the index into locations, the 2 is the number of words in a Point object, and the 4 is the size of the word.

# Memory Management Reviewed

In Java

- Memory allocation is automatic
- Freeing memory is automatic (by the garbage collector)
- Bounds of arrays are checked

In C or C++

- Allocations are explicit
- Freeing memory is explicit
- Bounds are not checked

Java is slower and safe, C and C++ is fast and dangerous.

A very common mistake is to try to access the last cell in an array incorrectly:

```
int[] a = new int[5];
a[5] = 1000; // Error:  the cells are a[0] to a[4]
```

This leads to an `ArrayIndexOutOfBoundsException` in Java whereas in C (or C++) this goes through without a warning and can lead to a corruption of data in memory!

# Inserting into an Array by Shifting Up

To insert a point at position `pos`, where $0 \leq pos \leq size$:

```
maxsize = 100
Point[] locations = new Point[maxsize];
int size = 0;    // number of points currently stored

void insert(int pos, Point pt) {
  if (size == maxsize) {
      throw new ArrayFullException("locations␣array");
  }
  for (int i=size-1; i >= pos; i--) {
    // Copy entry in pos i one pos towards the end
    locations[i+1] = locations[i];
  }
  locations[pos] = pt;
  size++;
}
```

If we want to insert a value to an array (at a certain position) we can do this in two steps:

1. Create a new array, of size bigger by one.
2. Copy elements of the old array to the new one to the corresponding positions.

However, this requires to copy the whole array every single time. Instead, we can allocate a big array at the beginning (of size `maxsize`) and then always “only” shift elements whenever we are inserting/deleting one.

Exercise: write the corresponding pseudocode to remove an item from an array by shifting down

# Linked Lists

# Linked Lists in Memory

Linked lists hold values of a particular type. They are constructed from structures (Class objects in Java), called *nodes* that have a `value` variable to hold the value and one or more `node` variables to identify the next node in the list.

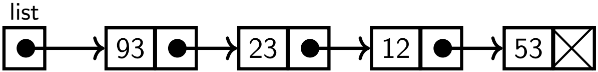
The linked list is then a collection of Node structures each connected to others in a chain.

```
class Node {
  int val;
  Node next;
}
Node list = END;
```

Note that no Node has yet been allocated and therefore the list is empty. `END` is a special value that represents an impossible memory address. Any attempt to access `Node.val` or `Node.next` when `list` has the value `END`, would immediately cause an error.

# Linked Lists in Memory

Linked list representing a list $\langle 93, 23, 12, 53 \rangle$:



Inserting at the beginning of a list:

```
void insert_beg(Node list, int number) {
  newNode = new Node();
  newNode.val = number
  newNode.next = list;
  list = newNode;
}
```

Check that it works, even if `list == END`.

What is the complexity of `insert_beg`, i.e. how many operations does it take to run the function once?

Does it depend on the size of the list?

| i | Memory |
|---|---|
| 6140 | 5312 |
| 6136 | 23 |
| ⋮ | ⋮ |
| 5316 | 1248 |
| 5312 | 12 |
| ⋮ | ⋮ |
| 3828 | 6136 |
| 3824 | 93 |
| ⋮ | ⋮ |
| list:2072 | 3824 |
| ⋮ | ⋮ |
| 1252 | END |
| 1248 | 53 |
| ⋮ | ⋮ |

Similarly to what we had before, each □□ is realised as a block of two consecutive locations in memory. The first location of such a block stores a number and the second location stores the address of the following block.

The `list` variable contains the address pointing to the first node, called the *head pointer*.

`END` indicates the end of the list (graphically as ☒). Its value can be anything that is not a valid address, for example, `-1`. Most languages use `0` for this purpose, which, although theoretically is a valid address, is never so in practice. Java uses a special value called `null`.

A linked list is empty whenever `list` is equal to `END`.

An advantage of linked lists over arrays is that the length of linked lists is not fixed. We can insert and delete items as we want. On the other hand, accessing an entry on a specific position requires traversing the list.

# Deleting at the beginning

```
Boolean is_empty(Node list) {
  return (list == END);
}
```

```
void delete_begin(Node list) {
  if is_empty(list) {
    throw new EmptyListException(”delete_begin”);
  }
  list = list.next;
}
```

# Lookup

```
int value_at(Node list, int index) {
  int i = 0;
  Node nextnode = list;
  while (true) {
    if (nextnode == END) {
      throw new OutOfBoundsException();
    }
    if (i == index) {
      break;
    }
    nextnode = nextnode.next;
    i++;
  }
  return nextnode.val;
}
```

What is the time complexity of these operations?

(How does this compare to arrays?)

How would you implement `insert_end` and `delete_end`?

# Insert at the end

```
void insert_end(Node list, int number) {
  newblock = new Node();
  newblock.val = number;
  newblock.next = END;
  if (list == END) {
    list == newblock;
  }
  else
  {
    cursor = list;
    while (cursor.next != END){
      cursor = cursor.next;
    }
    cursor.next = newblock;
  }
}
```

*Search* is a procedure which finds the position (counting from 0) where a value, given as a parameter, is stored in the array or linked list.

By "cost as a function of the number of elements" we mean: how does the number of items we have to inspect or modify grow as the number of elements in the structure grows? *Constant* means that the number of operations does not depend on the number of elements. *Linear* means that if we increase the number of elements in the structure by a factor of $n$, then the cost of the operation in the worst case will be multiplied by $n$ too.

We compare costs by comparing the number of elements of the list we have to inspect (in the worst case) in order to finish the operation.

In practise, we choose between using an array or linked list depending on both relative frequency of use of the operations and their relative costs.

# Comparison

If we store a list of `n` elements as an array (without spare space) or a linked list, what costs will the basic operations of lists have as a function of the number of elements in the list (when the list is seen as an ADT)?

| | Array | Linked List |
|---|---|---|
| access data by position | | |
| search for an element | | |
| insert an entry at the beginning | | |
| insert an entry at the end | | |
| insert an entry (on a certain position) | | |
| delete first entry | | |
| delete entry `i` | | |
| concatenate two lists | | |

## Comparison (solution)

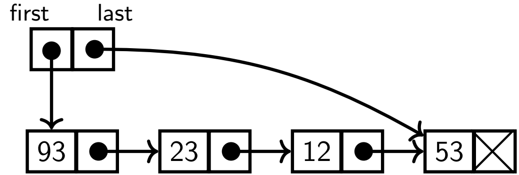
If we store a list of `n` elements as an array (without spare space) or a linked list, what costs will the basic operations of lists have as a function of the number of elements in the list (when the list is seen as an ADT)?

| | Array | Linked List |
|---|---|---|
| access data by position | constant | linear |
| search for an element | linear | linear |
| insert an entry at the beginning | linear | constant |
| insert an entry at the end | linear | linear* |
| insert an entry (on a certain position) | linear | linear |
| delete first entry | linear | constant |
| delete entry `i` | linear | linear |
| concatenate two lists | linear | linear* |

The stars indicate that it could be improved to constant time if we modified the representation slightly. As we'll see very soon...

# Modifications

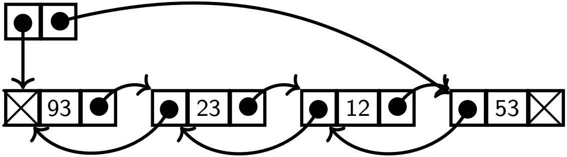
Linked list with a pointer to the last node:



Fast `insert_end`

Slow `delete_end`

Doubly linked list:



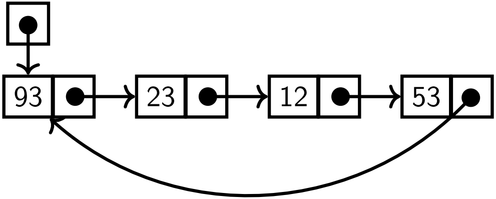
Try yourself: Write `insert_beg`, `insert_end`, `delete_beg` and `delete_end` for those two representations.

If we remember where the first and the last block of a doubly linked list is stored, then inserting at the end and deleting from the end could be implemented in constant time.

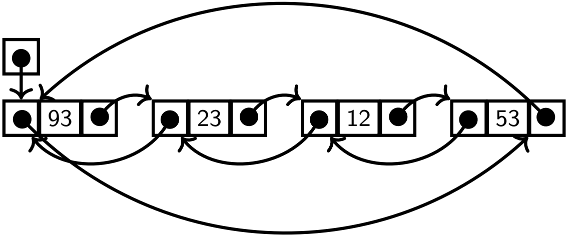
- For a singly linked list node we use:
  - `a.val` for the value contained node `a`
  - `a.next` for the (pointer to the) next node in the list
- For a doubly linked list node we use:
  - `a.prev` for the (pointer to the) previous node in the list
  - `a.val` for the value contained node `a`
  - `a.next` for the (pointer to the) next node in the list

# More Modifications

Circular Singly Linked List:



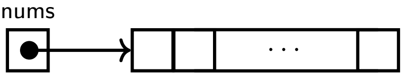
Circular Doubly Linked List:



Try yourself: Write `insert_beg`, `insert_end`, `delete_beg` and `delete_end` for those two representations.

# Time for a quiz!

Let `nums` be the address of an array of integers of length `len`.



Which of the following algorithms creates a **Singly Linked List** faster?

```
list = END
for (int i=0; i < len; i++) {
  insert_end(list, nums[i]);
}
```

```
list = END;
for (int i=len-1; i >= 0; i--) {
  insert_beg(list, nums[i]);
}
```

If we do not keep track of the last node then the second algorithm is faster. This is because every `insert_beg` takes constant time to execute whereas when running `insert_end` we need, every time we insert, to traverse the list to the end before actually adding the new element, and each time the list grows by one element so in total we have to traverse first a list of 0 element, then a list of 1 elements,..., until finally we traverse a list of $\texttt{len} - 1$ elements. That is, in total we do $0 + 1 + 2 + 3 + \ldots (\texttt{len} - 1)$, traversal steps, that is:

$$0 + 1 + 2 + 3 + \cdots + (\texttt{len} - 1) = \frac{\texttt{len}(\texttt{len} - 1)}{2}$$

# Abstract Data Types

# Abstract Data Types in Java

List is an ADT. A specific instance of a list, e.g. a List of integers, would be specified in Java by `List<int>`, a List of Strings as `List<String>` etc.

There are different implementations of the List ADT in the Java library, for example an Array based List (`ArrayList<int>`, `ArrayList<String>`, ...) and a Linked List (`LinkedList<int>`, `LinkedList<String>`, ...)

In Java we can declare and allocate a List, specifying which implemention we want, with the code:

```
    List<int> myArrayList = new ArrayList<int>();
    List<int> myLinkedList = new LinkedList<int>();
```

From this point on, you can use any of the predefined List methods on the `myArrayList` or `myLinkedList` variables

# Abstract Data Types Revisited

Recall that An *abstract data type* is

- a type
- with associated operations
- whose representation is hidden to the user

While a *List of integers* contains the type *integer*, the type of *List of integers* is not *integer*. It is a more complex "*container type*". This is usually specified contructively: that is, we identify every possible value of type *List of integers* by specifying how to create each one. We do this by providing a list of constructor operations that create an empty *List of integers* and construct new values of type *List of integers* out of old ones

We also need to specify all other operations that any user of our ADT can depend on

# List Abstract Data Type

Here is a possible list of operations for a List ADT (many variations are possible)[1]

- Constructors:
  - `EmptyList` : returns an empty List
  - `MakeList(element, list)` , adds an element at the front of a list.
- Accessors
  - `first(list)` : returns the first element of the list[2]
  - `rest(list)` : returns the list excluding the first element[2]
  - `isEmpty(list)` : reports whether the list is empty

From these, all other operations (e.g. find the n$^{th}$ element of the list, append one list onto another) can be implemented without requiring any other access to the List implementation details.

[1] Read chapters 1 and 2 of the module handouts

[2] Triggers error if the list is empty

# List Operations: last element

in Pseudocode:

```
last(lst) {
  if ( isEmpty(lst) )
    error(”Error:␣empty␣list␣in␣last”)
  elseif ( isEmpty(rest(lst) )
    return first(lst)
  else
    return last(rest(lst))
```

# List Operations: getElementByIndex

in Pseudocode:

```
getElementByIndex(index, lst) {
  if ( index < 0 or isEmpty(lst) )
    error(”Error:␣index␣out␣of␣range”)
  elseif ( index == 0 )
    return first(lst)
  else
    return getElementByIndex(index −1, rest(lst))
```

# List Operations: append

in Pseudocode:

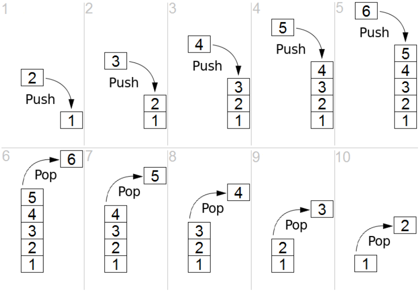
```
append(lst1, lst2) {
  if ( isEmpty(lst1) )
    return lst2
  else
    return MakeList( first(lst1),
                     append(rest(lst1), lst2) )
```

# Stacks

# Stacks = LIFOs (Last-In-First-Out)

*Stack* is an **abstract data type** defined by its three operations:

- `push(x)` puts value `x` on the *top* of the stack
- `pop()` takes out a value from the *top* of the stack
  If there are no values in the stack, it raises
  `EmptyStackException`.
- `is_empty()` says whether the stack is empty



(picture source: wikipedia)

*Stack* is an abstract data type. A stack is a list of values. The two ends of the list are called the *bottom* and *top*. We *push* a value (add it to the top of the stack) and *pop* a value (remove from the top of the stack). Thus a stack behaves in a Last In First Out (LIFO) manner. If we try to pop from an empty stack, we get an `EmptyStackException`. In theory we should be able to push any number of values, but in practice that won't be possible and we will get an exception if we run out of memory.

We can have a stack of any type of value, e.g. a stack of integers, a stack of strings, a stack of Booleans etc.

As a part of the specification of stacks it is usually also said that `push(x)` followed by `is_empty()` gives `false`, and `push(x)` followed by `pop()` gives `x` back.

Since we see a stack as an abstract data type, we do not specify how it is implemented.

### Example

Suppose we create an empty stack and we push 3, push 5, push 2 and pop; we get 2. Suppose we then pop; we get 5. Suppose we push 1, push 8, then pop. Pop again three times, what do we get?

### Usage

For example, when we are solving tasks with dependencies. Imagine that we want to complete a task A, `push(A)`, and

1. A depends on B and C $\Longrightarrow$ `push(B)` and `push(C)` *(ok, we need to solve C first but ...)*
2. C depends on D $\Longrightarrow$ `push(D)`.

Once we complete D (`pop()`), C (`pop()`), and B (`pop()`), we can also complete A (`pop()`).

*(More verbose example of usage:)*

One reason that stacks are useful is that sometimes, in order to complete job A we must first do job B and then job C, but in order to complete job B we must first do job D, and in order to do that we must first do job E and job F. Using a stack, we can push the primary job onto the stack first and each time we push any job onto the stack, we follow it by pushing the jobs that it depends on. So long as you then execute the jobs in the order that pop retrieves them, all the jobs will only be executed when the jobs they depend upon are complete.

Another example: the most natural way of evaluating an expression written in *reverse Polish notation* is by using stacks. This is because sub-expressions have to be evaluated before the higher level expressions that must use them, hence evaluating a sub-expression is a job that must be completed before we can evaluate the higher level expressions

Stacks are heavily used in applications that involve exhaustive searches of some problem space and in constructing and searching tree structures.

# Stack ADT

Here is a possible list of operations for a Stack ADT (many variations are possible)[3]

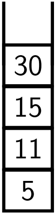
Constructors and Accessors:

- `EmptyStack`: returns an empty Stack
- `push(element, stack)`, pushes an element on top of the given stack.
- `top(stack)`: returns the value at the top of the stack without changing the stack[4]
- `pop(stack)`: returns the stack with the top element removed[4]
- `isEmpty(stack)`: reports whether the stack is empty

[3] Read chapters 1 and 2 of the module handouts

[4] Triggers error if the stack is empty

# Stacks as linked lists



To store a stack as a linked list we pick the faster of the two options:

1. the top is at the beginning, i.e. $\langle 30, 15, 11, 5 \rangle$
2. the top is at the end, i.e. $\langle 5, 11, 15, 30 \rangle$

Question: Which one is better and why?

Since inserting and deleting from the beginning of a linked list is constant, the first option is better. In other words, we take

- `push` = `insert_beg`
- `pop` = `delete_beg`
- `is_empty` (for stacks) = `is_empty` (for linked lists)

This way every operation on stacks takes constant time.

The second option would mean that `push` = `insert_end` and `pop` = `delete_end`. Then, even if we stored the position of the end of the linked list (to make sure `insert_end` is fast), `delete_end` would still be slow (linear time) and so the second option is not reasonable.

## Stacks as arrays

One can implement Stacks using a simple array in Java:

```java
// Initialize an empty stack:
stack = new int[MAXSTACK];
stack_size = 0;
```

Here the bottom of the stack is stored on position `0` of the array and the top of the stack in position `stack_size-1`. We can implement `push` and `pop` for this representation in constant time.

⟹ No matter if we store stacks as linked lists or arrays, in both cases, `push`, `pop`, and `is_empty` finish in *constant time*.

Storing stacks as arrays has the advantage that we avoid calling `allocate_memory` all the time (this takes time, even if it is done automatically for us, like in Java). On the other hand, we need to know the maximum size of the stack in advance.

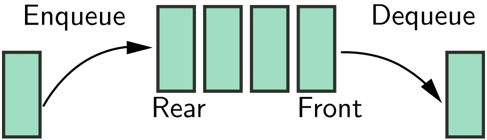
In practice, in Java, we normally use the library class `Deque<...>`, which implements the Stack ADT as well as some others we will discuss and adjusts to grow as the stack increases in size. We will see this later in this lecture.

# Queues

# Queues = FIFOs (First-In-First-Out)

*Queue* is an abstract data type defined by its three operations:

- `enqueue(x)` puts value `x` at the *rear* of the queue
- `dequeue()` takes out a value from the *front* of the queue
  If there are no values in the queue, it raises
  `EmptyQueueException`.
- `is_empty()` says whether the queue is empty



*Queue* is an abstract data type. A queue is a list of values (e.g. integers, Booleans, ...). The two ends of the list are called the *rear* and *front*. We *enqueue* a value (add it to the rear of the queue) and *dequeue* a value (remove it from the front of the queue). Thus a queue behaves in a First In First Out (FIFO) manner. If we try to dequeue from an empty queue, we get an `EmptyQueueException`. In theory we should be able to enqueue any number of values, but in practice that won't be possible and we will get an exception if we run out of memory.

### Example

Starting from an empty queue, enqueue 4 and 3, dequeue, then enqueue 1 and dequeue two times. What is the last value you get? What would happen in the next dequeue?

### Usage

A typical application of queues is a print queue: files are sent to the queue for printing and are printed in the order in which they were sent.

After sending a file, you know that only the jobs currently in the queue will be printed before yours.

Queues are useful whenever we have need to process tasks in the order in which they came. We demonstrate this in the printer queue but there are many more examples (e.g. web server when serving websites).

Notice that since the tasks in a print queue are executed in the order in which they came, there is no priority. Even as a lecturer I have to wait for all student tasks that came before mine to finish before my file gets printed.

# Queue ADT

Here is a possible list of operations for a Queue ADT (many variations are possible)[5]

Constructors and Accessors:

- `EmptyQueue`: returns an empty Queue
- `push(element, queue)`: (also called `enqueue`) pushes an element onto the back of the given queue.
- `top(queue)`: (also called `front`) returns the value at the front of the queue without changing the queue[6]
- `pop(queue)`: (also called `dequeue`) returns the queue with the front element removed[6]
- `isEmpty(queue)`: reports whether the queue is empty

[5] Read chapters 1 and 2 of the module handouts

[6] Triggers error if the queue is empty

## Queue as a linked list

In order to have an efficient implementation we need to store the location of the last element in the linked list.

Remember: **Dequeue** from **front**, **enqueue** to **rear**

We have two options again:

1. Front at beginning of the linked list, rear at end
2. Rear at beginning of the linked list, front at end

Question: Which one is better and why?

How would enqueue and dequeue be implemented if we did (1) or (2)?

For (1) as long as we use 2 pointers in the head node, one to the first node of the linked list, one to the last node, it will take constant time to dequeue (remove from the start of the Linked List) and to enqueue (add a node at the end of the linked list).

For (2), again with 2 pointers in the head node, we can enqueue in constant time (insert a node at the start of the linked list). However, to dequeue, we now need the address of the penultimate node of the linked list in order to remove the last node, and to find that address, we will need to iterate through the whole linked list, costing linear time.

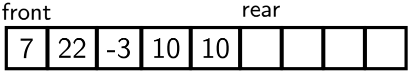
Thus we can ensure constant time enqueues and dequeues in case (1)

- `enqueue` = `insert_end`
- `dequeue` = `delete_beg`

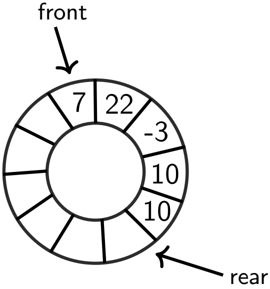
Case (2) gives us constant time enqueues but linear time dequeues.

# Queue stored in an array (1st attempt)

Whenever we `enqueue` (resp. `dequeue`) we increment `rear` (resp. `front`):



We eventually run out of space! To fix this, every time we `dequeue`, move everything to the left. It works but is slow!



Instead we use a circular storage of a queue. We move Front and Rear clockwise.

Works beautifully in theory but how do we implement it in an array?

If we know that the size of queue is limited during the run of our program, we can store the queue as an array. We store the front position and size in variables `front` and `size`, respectively. Then, values stored in the queue are stored on the positions `front`, `front+1`, ..., `front+size-1`. We must maintain the invariant `front+size` $\leqslant$ `MAXQUEUE`.

To `enqueue` we store the new value in position `front+size` and increment `size`. To `dequeue` we read out the value on position `front` and increment `front`.

However, this way, we eventually reach the end of the array and we can't continue enqueueing elements even if there is space in the array to do so, that is, the space in the array where old values were which have since been dequeued.

We can fix this by, everytime we dequeue a value, copying all the elements in the queue down to the start of the array so that `front` is back to 0. But this makes dequeue very slow. There is a better solution using a *Circular Array Queue Implementation*

# Digression: Invariants

An *invariant* is a condition on code. It must always be true during the execution of some section of code, e.g. a loop, or during the execution of a method, or, if it applies to a class, then it can be a condition that must be met by all objects of the class on every entry to and exit from the methods of the class even if it can be temporarily false during the execution of those methods.

Invariants are important because they specify conditions that must be met and maintained in parts of the code. So not only do they communicate information to the reader of the code, but they are tools that can be used both to identify and debug errors in the code (e.g. print an error message if this invariant is broken, stop in the debugger at any point when this invariant becomes false, etc.), and can be used to mathematically prove that the program is correct.

# Digression: `div` and `mod` (maths break)

For numbers $a, b$ with $b > 0$ we write `a div b` to mean the result of dividing $a$ by $b$ and discarding the remainder, and we write `a mod b` to mean the reminder.

For example:

$$123 \text{ div } 10 = 12 \qquad 58.7 \text{ div } 10 = 5$$
$$123 \text{ mod } 10 = 3 \qquad 58.7 \text{ mod } 10 = 8.7$$

**Example**

Racing on a track: Assume that one lap in a race is 600 meters. Then a runner who has run 1550 meters has completed `1550 div 600 = 2` laps and `1550 mod 600 = 350` meters of the third lap.

Note that `a div b` is always an integer and $0 \leq$ `a mod b` $< b$ such that

$$(a \text{ div } b) * b + a \text{ mod } b = a.$$

Consequently: $-7 \text{ div } 10 = -1$ and $-7 \text{ mod } 10 = 3$
$-123 \text{ div } 10 = -13$ and $-123 \text{ mod } 10 = 7$.

`mod` can be used to define the floor function (rounding down to an integer), we have

$$\lfloor x \rfloor = x \text{ div } 1 \quad \text{and} \quad x \text{ div } y = \lfloor x/y \rfloor$$

(very useful when we do complexity)

Note that in Java `Math.floorDiv` and `Math.floorMod` behave better than `%` and `/`, respectively, because the latter interpret the operations on negative numbers incorrectly (Java, and C, allows negative results for `mod`).

More examples:

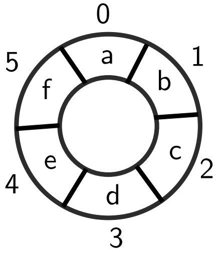
- $100 \bmod 10 = 0$ (100 is divisible by 10)
- $-18 \text{ div } 10 = -2$ (from 0, move backwards 2 * 10)
- $-18 \bmod 10 = 2$ (and then 2 steps forwards)

  (it has to be in the bounds: $0 \leq -18 \bmod 10 < 10$)

- Floor: $\lfloor 14.3 \rfloor = 14, \quad \lfloor 7.0 \rfloor = 7, \quad \lfloor -5.3 \rfloor = -6$
- Ceiling: $\lceil 14.3 \rceil = 15, \quad \lceil 7.0 \rceil = 7, \quad \lceil -5.3 \rceil = -5$

# Representing circles

Positions in the circle, numbered clockwise, correspond to the positions in the array:



$\Longrightarrow$

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| a | b | c | d | e | f |

**Example**

For a position `pos` in the circle (e.g. `pos = 5`), moving clockwise by one positions is computed as `(pos + 1) mod 6`.

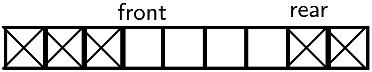
Or in general `(pos + 1) mod circle_length`.

# Circular queue: Array implementation

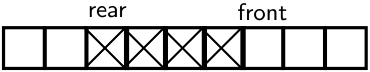
```
// Initialize an empty queue:
queue = new int[MAXQUEUE];
front = 0;
size = 0;
```

We store values in the queue in between positions marked by `rear` and `front`, that is,

- if $front + size \leqslant MAXQUEUE$ then the queue consists of the entries on positions $front$, $front + 1, \ldots, front + size - 1$:



- if $front + size > MAXQUEUE$ then the queue consists of the entries on positions $front$, $front + 1, \ldots, MAXQUEUE - 1$ and $0, 1, \ldots, (front + size - MAXQUEUE - 1)$:



## Circular queue: Array implementation

The invariants maintained in this implementation are:

- $0 \leqslant front < MAXQUEUE$
- $0 \leqslant size \leq MAXQUEUE$

Note that rather than using a *rear* index variable, we will always calculate it when we need it from *front*, *size* and *MAXQUEUE*. Thus `rear = (first + size) mod MAXQUEUE`

The queue is full if `size == MAXQUEUE` and empty when `size == 0`

To enqueue, we first check that the queue is not full, put the new value at index position `(first + size) mod MAXQUEUE`, and increment *size* by adding 1 to it.

To dequeue, we first check that the queue is not empty, get the value at index position `front`, increment *front* by calculating `front = (front + 1) mod MAXQUEUE` and decrement *size* by subtracting 1 from it.

# Circular queue: Array implementation

```
// Initialize empty queue:
queue = new int[MAXQUEUE];
front  = 0;
size = 0;
```

```
boolean is_empty () {
  return size == 0;
}
```

```
boolean is_full () {
  return size == MAXQUEUE;
}
```

```
void enqueue (int val) {
  if (size == MAXQUEUE) { throw QueueFullException; }
  queue[(front+size) mod MAXQUEUE] = val;
  size ++;
}
```

```
int dequeue () {
  int val;
  if (size == 0) { throw QueueEmptyException; }
  val = queue[front];
  front = (front+1) mod MAXQUEUE
  size --;
  return val;
}
```

# Double Ended Queues: Deques

While the Java library does have a `Stack<...>` class, it is an old design that has been kept for backwards compatibilty purposes and should not normally be used. For both `Stack` and `Queue` classes, you should use the `Deque<...>` class, which implements a *double-ended queue* data type. This has implementations `ArrayDeque<...>` and `LinkedList<...>` and supports inserting at and removing from both ends.

Actually, the `Deque<...>` class is really a Java *Interface*, rather than a full *Class*. This will be covered in your Java programming module but the distinction is not important for the purposes of this module.

# Final points

As we will see, stacks and queues are used in many algorithms.

We often just say “make a stack” or “make a queue” and we don’t care how they are implemented.

We know that, whether it is as an array or linked list, it can be done efficiently.

# Maths for Complexity Analysis

# Maths introduction: Exponentials

$$a^n = \underbrace{a \times a \times \ldots \times a}_{n \text{ times}}$$

$$a^m a^n = \underbrace{a \times \ldots \times a}_{m \text{ times}} \times \underbrace{a \times \ldots \times a}_{n \text{ times}} = a^{m+n}$$

$$(a^m)^n = \underbrace{a^m \times \cdots \times a^m}_{n \text{ times}} = a^{mn} = (a^n)^m$$

$$a^{(m^n)} = a^{\overbrace{m \times \cdots \times m}^{n \text{ times}}}$$

NOTE: $a^{(m^n)} \neq (a^m)^n$

$$a^0 = 1 \quad \text{because } a^0 a^1 = a^{0+1} = a^1 \Rightarrow a^0 = 1$$

$$a^{\frac{1}{n}} = \sqrt[n]{a} \quad \text{because } \underbrace{a^{\frac{1}{n}} \times \cdots \times a^{\frac{1}{n}}}_{n \text{ times}} = a^{\frac{n}{n}} = a$$

$$a^{-n} = \frac{1}{a^n} \quad \text{because } a^{-n} a^n = a^0 = 1$$

# Exponentials: Examples

$$\begin{aligned}
2^3 &= 8 \\
10^2 &= 100 \\
10^1 &= 10 \\
10^0 &= 0^0 = 1 \\
9^{1/2} &= 3 \\
2^{-3} &= 1/8 \\
\sqrt{5} \times \sqrt{5} &= 5^{\frac{1}{2}} \times 5^{\frac{1}{2}} = 5^{\frac{1}{2}+\frac{1}{2}} = 5^1 = 5 \\
16^{3/2} &= (16^{1/2})^3 = 4^3 = 64
\end{aligned}$$

# Maths introduction: Logarithms

$\log_a b$ is the number you have to raise $a$ to in order to get $b$:

$$\log_a b = c \text{ means that } a^c = b$$
$$\Rightarrow a^{\log_a b} = b$$

By applying $\log_a$ to both sides, and letting $c = \log_a b$, we get

$$\log_a\left(a^{\log_a b}\right) = \log_a b$$
$$\Rightarrow \log_a\left(a^c\right) = c \tag{1}$$

Thus $\log_a \bullet$ and $a^\bullet$ are inverses of each other and cancel.

$$\log_{10} 1000000 = 6$$
$$\log_{10} 0.0001 = -4$$
$$\log_2 32 = 5$$
$$\log_8 32 = \log_8(2^5) = \log_8\left((\sqrt[3]{8})^5\right) = 5/3$$

# Maths introduction: Rules for Logarithms

$\log_a bc = \log_a b + \log_a c$:

$$\begin{aligned}(bc) &= (b)(c)\\ \Rightarrow a^{\log_a bc} &= a^{\log_a b} a^{\log_a c}\\ &= a^{\log_a b + \log_a c}\\ \Rightarrow \log_a a^{\log_a bc} &= \log_a a^{\log_a b + \log_a c}\\ \Rightarrow \log_a bc &= \log_a b + \log_a c\end{aligned}$$

$\log_a \frac{b}{c} = \log_a b - \log_a c$ (similarly)
$\log_a b^c = \log_a \underbrace{b \times \cdots \times b}_{c \text{ times}} = \underbrace{\log_a b + \cdots + \log_a b}_{c \text{ times}} = c \log_a b$

# Maths introduction: Changing base of Logarithms

$\log_c x = (\log_c b)(\log_b x)$, First (impressive looking) proof:

$$\begin{aligned} x &= x \\ &= b^{\log_b x} \\ &= \left(c^{\log_c b}\right)^{\log_b x} \\ &= c^{(\log_c b)(\log_b x)} \\ \Rightarrow \log_c x &= \log_c c^{(\log_c b)(\log_b x)} \\ &= (\log_c b)(\log_b x) \end{aligned}$$

$\log_c x = (\log_c b)(\log_b x)$, Second, simpler proof:

$$\begin{aligned} x &= b^{\log_b x} \\ \Rightarrow log_c x &= log_c b^{\log_b x} \\ &= (\log_b x)(\log_c b) \end{aligned}$$

# Maths introduction: More facts about Logarithms

$$\begin{aligned} \log_a a &= 1 \\ \log_a 1 &= 0 \\ \log_a x &\text{ when } x \leq 0 \text{ is undefined} \\ \lim_{x \to 0^+} \log_a x &= -\infty \\ \lim_{x \to +\infty} \log_a x &= \infty \end{aligned}$$

# Complexity

# Linear search (worst case complexity)

```
 1 int search (int[] array, int x) {
 2   int n = array.length;
 3   int i = 0;
 4
 5   while (i < n) {            // iterate over the elements
 6     if (array[i] == x) {
 7       return i;              // found it!
 8     } else {
 9       i = i + 1;             // try the next one
10     }
11   }
12
13   return -1;                 // the value not found
14 }
```

Worst case: the value `x` is not in the array.

Number of steps: $2 + n \times (1 + 1 + 1) + 2 = 3n + 4$

(2nd and 3rd lines, then `n` -times 5th, 6th and 9th lines, and finally the 5th and 13th line)

# Linear search (worst case complexity), recursively

```
 1  int search (int[] array, int x) {
 2    search_rec(array, 0, x);
 3  }
 4
 5  int search_rec(int[] array, int i, int x) {
 6    if (i == array.length)
 7      return -1;              // the value not found
 8
 9    if (array[i] == x)
10      return i;               // found it!
11
12    int i_next = i + 1;       // try the next one
13    return search_rec(array, i_next, x);
14  }
```

Worst case: the value `x` is not in the array.

Number of steps: $1 + n \times (1 + 1 + 1 + 1) + 3 = 4n + 4$

(2nd line, then `n` -times 6th, 9th, 12th and 13th lines, and finally 6th and 7th line)

## What is the difference between the two?

From the theoretical perspective we are more interested in how the number of steps *grows with respect to the input size*, rather than in the actual number of steps. This is because the actual speed depends on

- the hardware on which it runs,
- programming language used (or its compiler),
- how well is the implementation optimised, ...

## Performance of Algorithms

A number of timed searches were performed on a sorted list of 10,000 numbers. When searching for a number at the start, at the end, and that was not in the list respectively, the timings were in the following ranges:

- Linear search:
    - Start: 2 $\mu$secs
    - End: 391 $\mu$secs
    - Not in: 356 $\mu$secs
- Binary search:
    - Start: 6 $\mu$secs
    - End: 5 $\mu$secs
    - Not in: 5 $\mu$secs

# Performance of Algorithms

So: Binary search seems to be faster than linear search

- unless searching for the first item in the list
- on the machine that these timings were run on
- when run on sorted lists of numbers
- when the list is of length 10,000
- ... and possibly with other restrictions

We need a better way to be able to think about and compare algorithm performance.

# Time and Space Performance

There are 2 *dimensions* of performance we might be interested in:

- Time: How long it takes to run the algorithm:
  - Measuring this in normal time units makes us dependent on the machine we run it on.
  - Instead measure it in numbers of steps: e.g. the number of additions or multiplications, the number of comparison operations, the number of memory accesses etc.
- Space: How much memory it requires:
  - Different algorithms to accomplish the same result might use different amounts of memory. For example:
    - One takes 1,000,000 steps and require only 2 integer variables
    - Another take 20,000 steps but requires a list of length 1,000

To choose between algorithms, we need to understand both its time and space performance.

# Complexity

Even if we know an algorithm's time performance (in units of steps) and space performance (in units of words of memory), we still do not yet have a way of capturing how that performance changes with different sizes of problems.

Solution: parameterize the performance by the size of the input:

- This algorithm takes $N$ steps on an input of size $N$
- This algorithm takes $2N^2 + N$ steps on an input of size $N$
- This algorithm uses $3N$ words of memory on an input of size $N$

# Average and Worst Case Complexity

Linear search took different times depending on whether the item was at the start of the list or at the end even though the size of the list didn't change:

- *Average case time complexity*: Consider every possible case for an input of size $n$ and calculate the average
  - Linear search on $n$ elements requires 1 comparison if the item is first in the list.
  - It requires 2 comparisons if the item is second in the list. . .
  - It requires $n$ comparisons if the item is last in the list.
  - Average time complexity: $\frac{1+2+\cdots+n}{n} = \frac{n(n+1)}{2n} = \frac{n+1}{2}$ comparisons
- *Worst case time complexity*: Choose the case that will take the largest number of steps.
  - Linear search for the last item in the list: $n$ comparisons

# Worst Case Complexity of Binary Search

How large a list, *n* can we search with *c* comparisons?

$$\begin{aligned}
1&: \quad 1 &&= 1 = 2^1 - 1\\
2&: 1 + 1 + 1 &&= 3 = 2^2 - 1\\
3&: 3 + 1 + 3 &&= 7 = 2^3 - 1\\
4&: 7 + 1 + 7 &&= 7 = 2^4 - 1
\end{aligned}$$

In general: $c$ comparisons lets us search a list of length $2^c - 1$

We want to know the inverse: how many comparisons do we need to search a list of length *n*?

If we ignore the "-1", which only has a small relative effect, the worst time complexity of binary search is $log_2(n)$

# Average Case Complexity of Binary Search

What about the average case? Take a list of length *n*:

- Only 1 case where we find the target item in 1 comparison
- 2 cases where we find the item in 2 comparisons
- 4 cases where we find the item in 3 comparisons
- $\vdots$
- $n/2$ cases where we find the item in $log_2(n)$ comparisons

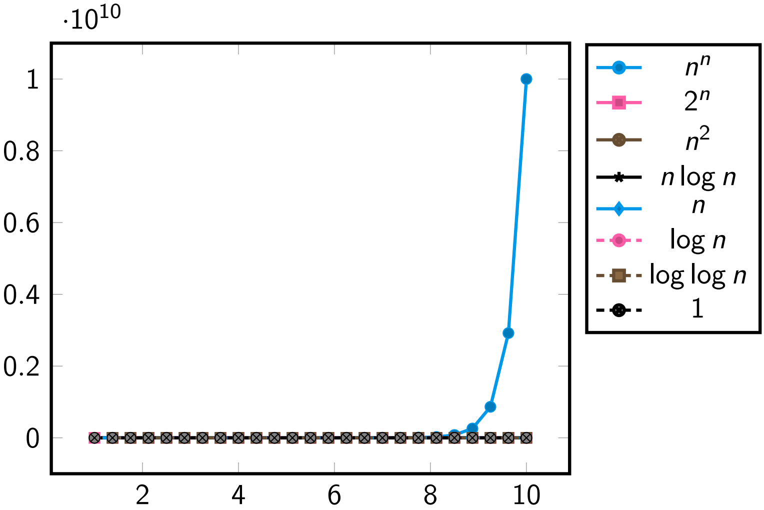
Thus *n* cases in total where:

- Half have the worst case complexity of $log_2(n)$
- One quarter have complexity $log_2(n) - 1$
- One eighth have complexity $log_2(n) - 2$
- $\vdots$

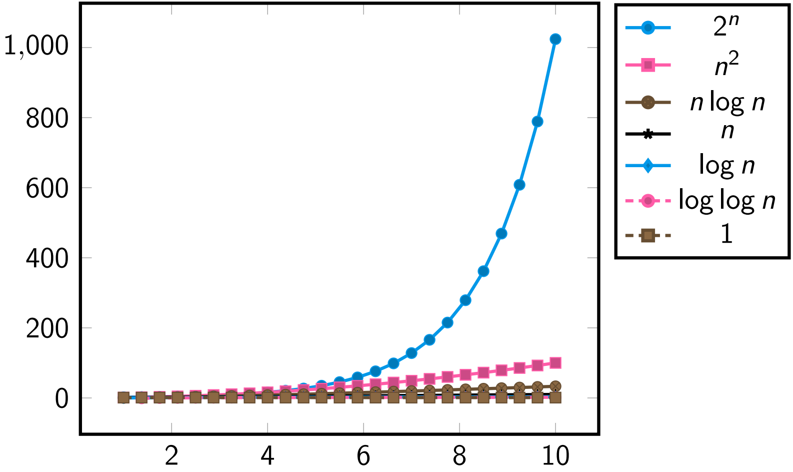
Average case is only slightly less than the worst case:

Approximate it with the worst case.

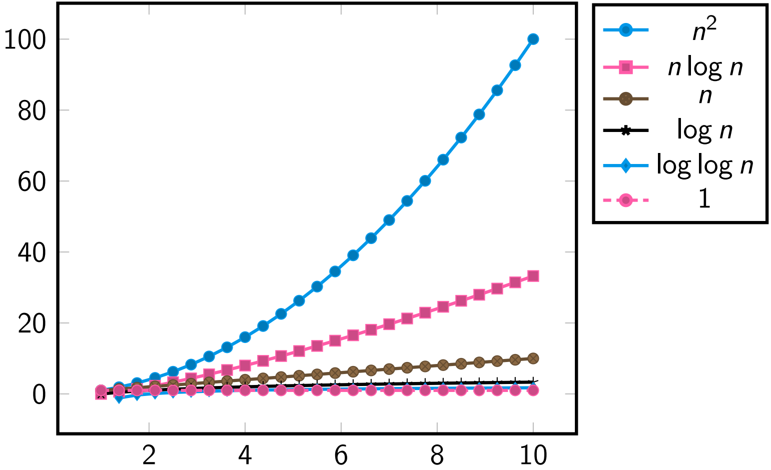
# Comparing Functions



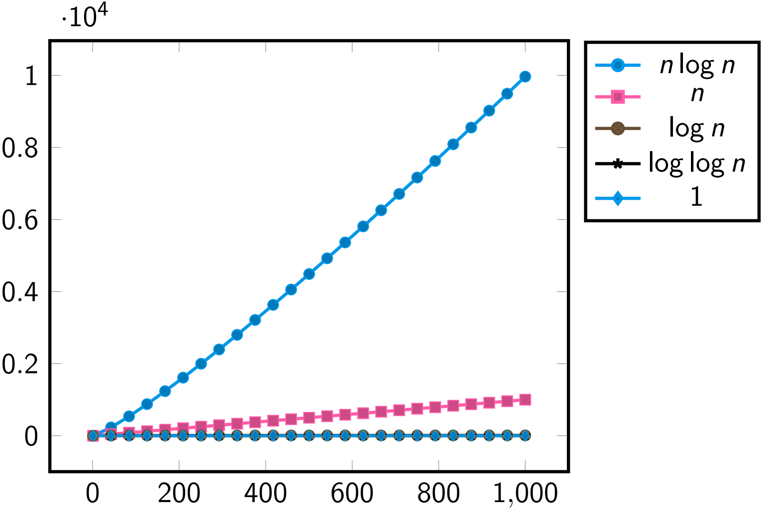
# Comparing Functions



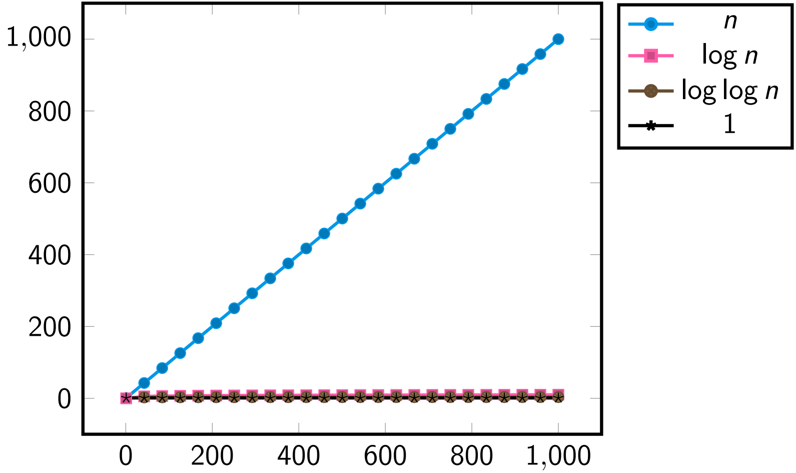
# Comparing Functions



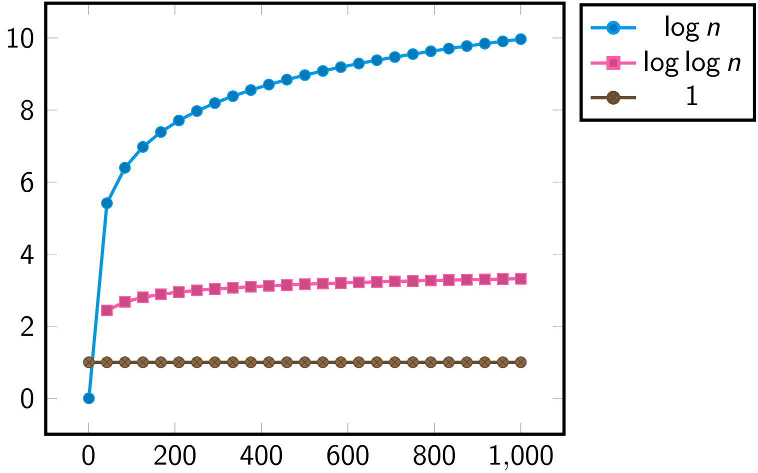
# Comparing Functions



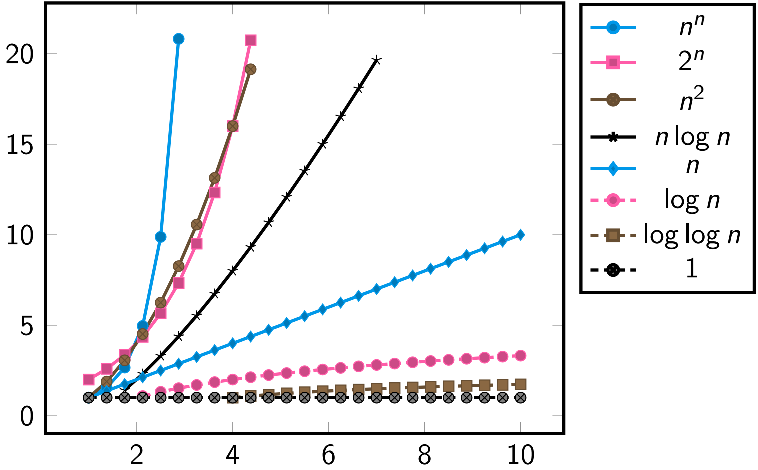
# Comparing Functions



# Comparing Functions



# Comparing Functions



## Big O notation

The precise number of steps is often too detailed to get a clear understanding of the performance of an algorithm:

- What if an algorithm does a comparison and a multiplication on every element of a list:
  - Complexity is $n$ steps, where a step is a comparison **AND** a multiplication
  - Complexity is $2n$ steps, where a step is a comparison **OR** a multiplication

The difference between an algorithm that has time complexity $n$ and one that has $2n$ is small in relation to the difference between two algorithms that have respective complexities of $n$ and $n^2$.

Similarly in an algorithm that has complexity $n^2 + n$, the $n$ part only makes a small contribution.

**Solution: simplify to headline complexity**

# Big O notation

Describe complexity by the most significant overview feature:

$$f(n) = O(g(n)) \iff |f(n)| \leq |Cg(n)|$$

for positive constants $C, n_0$ where $n > n_0$

Idea: $f$ does not grow at a faster rate than $g$ as $n$ increases. It might grow at the same rate or it might grow at a slower rate.

Examples (restrict ourselves to $n \geq 4$):

- $2n = O(n)$?
- $2n + 100 = O(n)$?
- $n = O(1)$?
- $3n^2 + n = O(n^2)$?
- $3n^2 + n = O(n^3)$?
- $3n^2 + n = O(n)$?

# Big O notation

Examples (restrict ourselves to $n \geq 4$):

- $2n = O(n)$?
  - *TRUE*: choose C to be 3
- $2n + 100 = O(n)$?
  - *TRUE*: choose C to be 1000
- $n = O(1)$?
  - *FALSE*: no value of C is large enough so that $n \leq C$
- $3n^2 + n = O(n^2)$?
  - *TRUE*: $3n^2 + n \leq 3n^2 + n^2 = 4n^2$ choose C to be 5
- $3n^2 + n = O(n^3)$?
  - *TRUE*: $3n^2 + n \leq 4n^2 \leq 4n^3$ choose C to be 5
- $3n^2 + n = O(n)$?
  - *FALSE*: no value of $C$ is large enough so that $3n^2 + n \leq Cn$

## More Big O examples

For all values of $n \geq 4$:

$$1 \leq \log\log n \leq \log n \leq n \leq n \log n \leq n^2 \leq 2^n \leq n^n$$

Therefore

$$\begin{aligned} 1 &= O(\log\log n) \\ \log\log n &= O(\log n) \\ \log n &= O(n) \\ n &= O(n \log n) \\ n \log n &= O(n^2) \\ n^2 &= O(2^n) \\ 2^n &= O(n^n) \end{aligned}$$

Thus, for example:

$$3n^n + 42^n + 100n^2 + 454n \log n + 24n + 12 \log n + 52 \log\log n + 43 = O(n^n)$$

# Be careful with Big O

Big O is a *notation*, not a function. Thus $O(f(n))$ is not a function, even if it looks like one:

- "$3n^2 + 4 = O(n^2)$" is just a shorthand way of writing "$|3n^2 + 4| \leq |Cn^2|$ for some constant $C$"
- "$O(n^2) = 3n^2 + 4$" does not have any meaning
  - If it did, we could do nasty things like:

$$3 = O(1) = 2 \text{ hence } 3 = 2 \textbf{ WRONG!!}$$

Big O notation can be made mathematically precise by defining it to be the *class* of functions with complexity $O(f(n))$. Therefore we can say that the complexity of an algorithm is "*in*" $O(f(n))$ or, shortening it, simply say that, for an algorithm $X$, we have that $X \in O(f(n))$

# Linear search (average case complexity)

```java
int search (int[] array, int x) {
  int n = array.length;
  int i = 0;

  while (i < n) {
    if (array[i] == x) {
      return i;
    } else {
      i++;
    }
  }

  return -1; // the value not found
}
```

Average case: the value `x` is on the position $\frac{n}{2}$

(We assume that `x` appears once in the array)

Number of steps: $2 + \frac{n}{2} \times 3 = \frac{3}{2}n + 2 \implies$ it is in $O(n)$

(one iteration of the while loop is 3 steps, no matter if we found `x` or not)

Previously we computed that the **worst case** complexity of linear search is $O(n)$. This happens if the value is not in the array and we need to search through the whole array.

Next, we consider a situation when the value `x` is in the array. (And for simplicity we assume that it is there only once). How many steps does it take **on average** to find `x`?

Because `x` can be on any position, it is on average in the middle of the array. This means that we find it on position $\frac{n}{2}$ and so the while-loop evaluates $\frac{n}{2}$-many times.

# Binary Search (worst case and average case)

Searching `x` in a **sorted** array `arr`:

1. Compare `x` and `arr[arr.length div 2]`.
2. If `x` is bigger, recursively search `arr` on positions `(arr.length div 2) + 1`, ..., `arr.length - 1`.
3. Otherwise, recursively search `arr` on positions `0`, ..., `arr.length div 2`
4. We continue like this until we are left with only one element in the array. Then, return whether this element equals `x`.

The length of the array we search through reduces by one half in every step and we continue until the length is 1.

For simplicity assume that the length of `arr` is $n = 2^k$

$\implies$ the number of steps is $O(k) = O(\log_2 n)$.

# Other Complexity Measures

# Big O and Friends

So far we have looked at Big O as a way to identify the complexity of an algorithm, and that is what we will be most concerned with. But there are others:

- **Big O**: $f(n) = O(g(n))$: $g$ is an upper bound on how fast $f$ grows as $n$ increases.
- **Little o**: $f(n) = o(g(n))$: A stricter upper bound than Big O.
- **Theta**: $f(n) = \Theta(g(n)$: More precise than Big O and Little o, it provides both upper and lower bounds, which are given by the same function, except with different constant factors. That is, $f$ and $g$ grow at the same rate.
- **Asymptotically Equal**: $f(n) \sim g(n)$: stricter upper and lowerbounds
- **Omega**: $f(n) = \Omega(g(n))$: a lower bound on how fast $f$ grows as $n$ increases (the lower bound equivalent of big O)

# Big O revisited

$$f(n) = O(g(n)) \iff |f(n)| \leq |Cg(n)|$$

for some constants $C, n_0$ where $n > n_0$

- $f$ grows at the same rate or slower than $g$.
- But $2n^2 + n = O(n^2)$, so we can have $f(n) > g(n)$ for all n.
- So Big O only refers to relative growth rate, NOT relative speed or memory usage.

# Little o

$$f(n) = o(g(n)) \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} \text{ exists and is equal to 0}$$

This makes $g$ an upperbound on $f$ but a stronger one than Big O:

Note that $2n^2 = O(n^3)$ and $2n^2 = O(n^2)$ (choose C = 3)

$2n^2 = o(n^3)$ because:

$$\begin{aligned}\lim_{n\to\infty} \frac{2n^2}{n^3} &= \lim_{n\to\infty} \frac{2}{n} \\ &= 0\end{aligned}$$

But it is not true that $2n^2 = o(n^2)$ because:

$$\begin{aligned}\lim_{n\to\infty} \frac{2n^2}{n^2} &= \lim_{n\to\infty} 2 \\ &= 2\end{aligned}$$

# Theta

$$f(n) = \Theta(g(n)) \iff c_1 g(n) \leq f(n) \leq c_2 g(n)$$

for positive constants $c_1, c_2, n_0$, and $n > n_0$

This means that $f$ and $g$ have the same rates of growth, with some constant multiple, i.e. that $f$ is bounded above and below by (possibly different) multiples of $g$.

This is only true if $f(n) = O(g(n))$ and $g(n) = O(f(n))$

Example: $x^2 + 2x + 1 = \Theta(x^2)$

But it is not true that $x^2 + 2x + 1 = \Theta(x^3)$

# Asymptotically Equal

$$f(n) \sim g(n) \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} \text{ exists and is equal to 1}$$

This has the same relation to Theta that Little o has to Big O: Asymptotically Equal is a tighter upper and lowerbound than Theta.

$x^2 + x = \Theta(x^2)$ and $x^2 + x \sim x^2$

However, $2x^2 + x = \Theta(x^2)$ and it is **NOT** true that $2x^2 + x \sim x^2$

# Omega

$$f(n) = \Omega(g(n)) \iff |f(n)| \geq |cg(n)|$$

for positive constants $c, n_0$ where $n > n_0$

This provides a lower bound on $f$: As $f$ grows, it will always grow at least at the same rate as $g$ and it could grow faster.

# Amortized Complexity

# Different kinds of complexities

**Average Case complexity**

= average complexity over all *possible inputs/situations*
(we need to know the likelihood of each of the input!)

**Worst Case complexity**

= the worst complexity over all *possible inputs/situations*

**Best Case complexity**

= the best complexity over all *possible inputs/situations*

**Amortized complexity**

= average time taken over a sequence of *consecutive* operations

We will often see algorithms where the worst case complexity is much more than the average case complexity. It depends on the usage, if you're processing a lot of data, occasional bad performance might be OK. However, if we are serving a customer and (in the worst case) she has to wait a long time, that's not good for the company's reputation.

In average-, worst- or best-case complexities, we are concerned with the performance of **one** (independent) operation. The amortized complexity is different: we are thinking about the average time complexity among a number of **successive operations**.

The idea is that, sometimes, you deliberately put extra effort in some operations, in order to speed up subsequent operations. For example, you might spend some time cleaning up or reorganizing of your data structure in order to improve the speed of the coming operations.

Whenever you are reorganizing your data structure, it might slow you down now but you'll benefit from this later (and hopefully many times).

## Example: Linear search

What is the time complexity of linear search, where the searched value is stored in `x`? Assume that the length of the array is $n$.

- **Worst Case:** `x` is at the end of the array

  $\Longrightarrow$ we need to traverse $n$ elements
- **Best Case:** `x` is at the beginning of the array

  $\Longrightarrow$ we only compare with the first one
- **Average Case (assuming that `x` is in the array)** – we consider two scenarios:

  **(1)** The likelihood of `x` being on any position is uniform. In other words, the chance that `x` is on position 0 is the same as for 1 or any other position.

  Then, the average number of traversed elements is equal to

$$\frac{1+2+3+\ldots+n}{n} = \frac{\frac{n(n+1)}{2}}{n} = \frac{n+1}{2}$$

We assume that `x` is in the array and it is there exactly once! Otherwise, we would have to compute the average complexity slightly differently.

Recall that we have a formula for *triangular numbers*:

$$1 + 2 + \ldots + n = \frac{n(n+1)}{2}$$

The *arithmetic progression* $1 + 2 + \ldots + n$ can be also written as $\sum_{i=1}^{n} i$.

Another example of a progression for which we have an exact formula is the following *geometric progression*:

$$1 + a + a^2 + a^3 + \cdots + a^{k-1} = \frac{a^k - 1}{a - 1}$$

This is easily shown by letting $S = 1 + a + a^2 + \cdots + a^{k-1}$

Then $(a-1)S = a^k - 1 \Rightarrow S = \frac{a^k-1}{a-1}$

# Example: Linear search

- **Average Case (assuming that `x` is in the array)**
  The average number of traversed elements is computed as

$$1P(1) + 2P(2) + 3P(3) + \cdots + nP(n)$$

where $P(i)$ denotes the likelihood (or probability) that `x` is stored on the `i` th position. This means that $0 \leq P(i) \leq 1$ and the sum of all likelihoods is equal to 1, i.e.

$$P(1) + P(2) + \ldots + P(n) = 1.$$

(In the uniform case $P(i) = \frac{1}{n}$ for every position.)

Note: The sum $1P(1) + 2P(2) + \ldots + nP(n)$ can be also written as $\sum_{i=1}^{n} iP(i)$.

# Example: Amortized car cost

Oil consumption: 8 litres per 100 miles

Price of 1 litre is £1.20

$\implies$ £9.60 per 100 miles

Extra expenses

- new tyres £320 every 70 000 miles
- new brakes £250 every 30 000 miles
- fix gearbox £300 every 130 000 miles
- fix clutch £406 every 100 000 miles

$\implies$ extra £1.93 per 100 miles

$\implies$ £11.53 is **amortized cost** per 100 miles

*($\star$ those numbers are somewhat made up!)*

# Amortized complexity: Dynamic array (first attempt)

**Naive approach:**

1. initially allocate an array of 1000 entries
2. whenever the array becomes full, increase its size by 100

To insert $n$ entries, starting from empty, how long does it take?

For simplicity assume that $n = 1000 + 100k$ (for some $k$).

1000 insertions +
1000 copies + 100 insertions +
1100 copies + 100 insertions +
1200 copies + 100 insertions +
1300 copies + 100 insertions +
...

In total:

- insertions: $1000 + 100k$
- copies:
  $$1000k$$
  $$+100\times(1 + 2 + \ldots + (k-1))$$
  $$= 1000k + 50k(k-1)$$

At the beginning we have

```
MAXSIZE = 1000;
arr = new int[MAXSIZE];
stored = 0;
```

We add elements to it by storing them at the end and increasing `stored` by one. Then, anytime `stored == MAXSIZE` (i.e. `arr` becomes full), we have to allocate a new array of size `MAXSIZE = MAXSIZE + 100` and copy all elements from `arr` into it.

Recall that $1 + 2 + \ldots + n = \frac{n(n+1)}{2}$ therefore

$$1 + 2 + \ldots + (k-1) = \frac{(k-1)k}{2}$$

The following analysis, however, does not depend on the exact choice of the parameters. If we started with an array of length 10 and increased its size by 5 every time it becomes full, for example, the resulting amortized complexity would still be the same.

Copies : $1000k + 50k(k-1)$

Insertions: $1000 + 100k$

Copies and insertions together: $1000(k+1) + 100k + 50k(k-1)$

Amortized cost of one insertion:

$$\frac{1000(k+1) + 100k + 50k(k-1)}{1000 + 100k}$$

The numerator is in $\theta(k^2)$ and denominator is in $\theta(k)$

$\implies$ the whole fraction is in $\theta(k)$.

But $O(n) = O(k)$ because $n = 1000 + 100k$

$\implies$ the amortized complexity of insertion is $O(n)$.

The amortized cost is computed as the average number of operations needed for one insertion. In our case:

$$\frac{\text{number of copying and inserting}}{\text{number of inserting}}$$

We see that copying is the problem. In the following smarter approach we try to suggest a different strategy which makes sure that copying happens less often.

# Amortized complexity: Dynamic array (= Java's ArrayList)

**Smart approach:**

1. initially allocate an array of 1000 entries
2. whenever the array becomes full, double its size

To insert $n$ entries, starting from empty, how long does it take?
For simplicity assume that $n = 1000 \times 2^k$ (for some $k$).

1000 insertions +
1000 copies + 1000 insertions +
2000 copies + 2000 insertions +
4000 copies + 4000 insertions +
8000 copies + 8000 insertions +
...

In total:

- insertions:
  $1000$
  $+1000 \times (1+2+4+\ldots+2^{k-1})$
- copies:
  $1000 \times (1+2+4+\ldots+2^{k-1})$

Copying and inserting together:

$$1000 + 2{\times}1000{\times}(1 + 2 + 4 + \ldots + 2^{k-1})$$

Because $1 + 2 + 4 + \ldots + 2^{k-1} = 2^k - 1$, this is equal to

$$1000 + 2{\times}1000{\times}(2^k - 1)$$

Amortized cost of one insertion:

$$\frac{2{\times}1000{\times}2^k - 1000}{n}$$

Because $n = 1000 \times 2^k$, the numerator is in $\Theta(n)$ and denominator is in $\Theta(n)$

$\Longrightarrow$ the whole fraction is in $\Theta(1)$

$\Longrightarrow$ amortized complexity of insertion is $\Theta(1)$

# Comparison

## Inserting at the end of an array

| | Average Case | Best Case | Worst Case | Amortized |
|---|---|---|---|---|
| Naive alg. | — | $O(1)$ | $O(n)$ | $O(n)$ |
| Smart alg. | — | $O(1)$ | $O(n)$ | $O(1)$ |

(Average Case complexity doesn't make sense to consider here.)

## Search in a sorted array

| | Average Case | Best Case | Worst Case | Amortized |
|---|---|---|---|---|
| Linear srch | $O(n)$ | $O(1)$ | $O(n)$ | $O(n)$ |
| Binary srch | $O(\log n)$ | $O(\log n)$ | $O(\log n)$ | $O(\log n)$ |

(Amortized complexity is the same as Average Case complexity because the previous searches don't have any effect on the next one.)

We see that the best case and worst case complexities of the Naive algorithm and the Smart algorithm are the same. The only difference is the amortized complexity. The reason why the amortized complexity of the naive algorithm is worse is because the worse case happens too often.

Average Case complexity doesn't make sense to consider in the first table. The time complexity does not depend on the "size" of the value that is being added. It only depends on the current number of elements stored in the array.

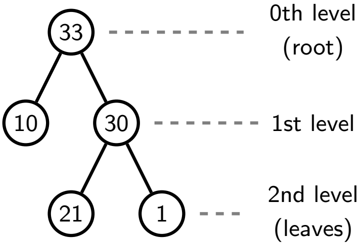
# Trees

# Trees

A tree is a very flexible and powerful data structure that, like a linked list, involves connected nodes, but has a hierarchical structure instead of the linear structure of linked lists.

Depending on the number of child nodes that each node has:

- Unary trees (0–1 children) = Linked Lists,
- Binary trees (0–2),
- Ternary trees (0–3),
- Quad trees (0–4), ...

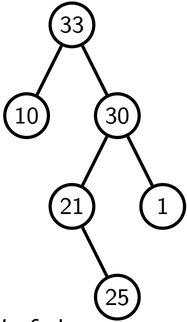


size = 5
height = 2

Size = number of nodes

Height = length of longest path from the root to a leaf

# Tree Terminology



- *Root*: the unique node at the base of the tree.
- Each node is connected by a link, called an *edge*, to each of its *child* nodes.
- Each *child* node has exactly one *parent* node.
- *Siblings* are nodes with the same *parent*.
- A node with no child nodes is called a *leaf*
- An *ancestor/descendent* of a node is the *parent/child* of the node or (inductively) the *ancestor/descendent* of that *parent/child*.
- A path is a sequence of connected edges between two nodes.

# Tree Terminology

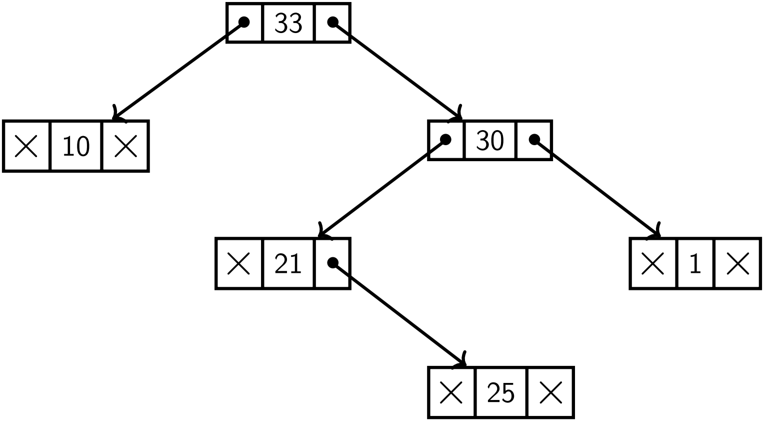
- Trees have the property that there is exactly 1 path between each node and the root
- The *depth* or *level* of a node is the length of the path from the node to the *root* (*root* has *level* 0).
- The *height* of a tree is the length of the longest path from the *root* to a *leaf*.
- The *size* of a tree is the number of nodes in the tree.
- A tree with one node has *size* 1 and *height* 0.
- An empty tree (with no nodes) has *size* 0 and, by convention, a *height* of -1.

# Tree Implementation Options

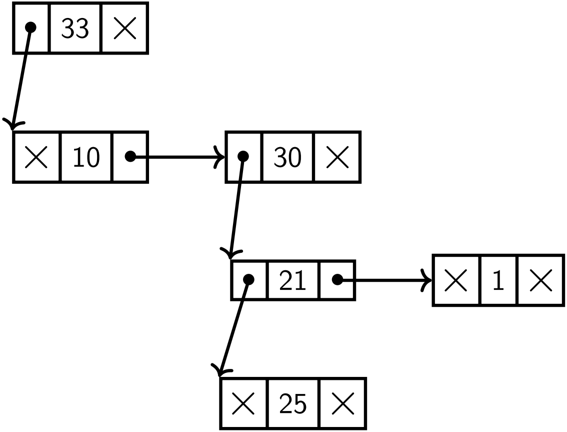
There are 3 common approaches to implementing trees:

1. Basic: Use nodes like doubly linked list nodes with a value field, and left and right child pointers
2. Sibling List: Use nodes with a value field, a single *children* pointer, and a pointer to the next sibling. This is good for trees with a variable number of children in each node.
3. Array: For binary trees, use arrays with a layout based on storing the root at index 1, then the children of the node at index $i$ is stored at index $2 * i$ and $2 * i + 1$.

# Tree Implementation Options: Basic



# Tree Implementation Options: Sibling List



# Tree Implementation Options: Array

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ☒ | 33 | 10 | 30 | ☒ | ☒ | 21 | 1 | ☒ | ☒ | ☒ | ☒ | 25 | ☒ | ☒ | ☒ |

# Binary Tree ADT

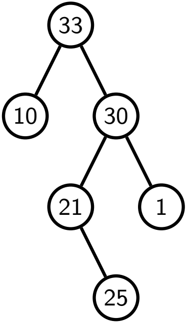
Just as with lists, we can define the *Binary Tree Abstract Data Type* inductively:

- Constructors:
  - `EmptyTree`: returns an empty tree
  - `MakeTree(v, l, r)`: returns a new tree where the root node has value `v`, left subtree `l` and right subtree `r`
- Accessors:
  - `isEmpty(t)`: return true if `t` is the empty tree, otherwise returns false
  - `root(t)`: returns the value of the root node of the tree `t`[1]
  - `left(t)`: returns the left subtree of the tree `t`[2]
  - `right(t)`: returns the right subtree of the tree `t`[2]
- Convenience Constructor:
  - `Leaf(v)` = `MakeTree(v, EmptyTree, EmptyTree)`

[1] Triggers error if the tree is empty

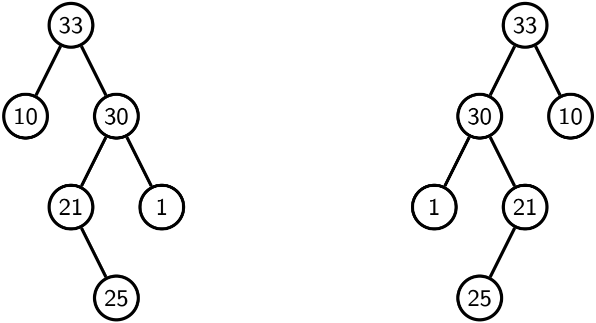
# Example: Construct a Tree

```
    MakeTree(33,
             Leaf(10),
             MakeTree(30,
                      MakeTree(21,
                               EmptyTree,
                               Leaf(25)),
                      Leaf(1)))
```



# Example: Reverse a tree

```
reverseTree(t) {
  if ( isEmpty(t) )
     return (t)
  else
    return (MakeTree(root(t),
            reverseTree(right(t)),
            reverseTree(left(t))))
```



## Example: Flatten a tree into a list

Here we assume that we have the code to append two lists (see the handout *dsa-slides-02-01-intro-ADT.pdf*) and that `isEmpty(...)` can distinguish between the list and the tree version (e.g. by qualifying it with the ADT name):

```
flatten(t) {
  if Tree.isEmpty(t)
    return EmptyList
  else
   return append(flatten(left(t)),
                  MakeList(root(t), flatten(right(t)))))
}
```

# Quad Trees

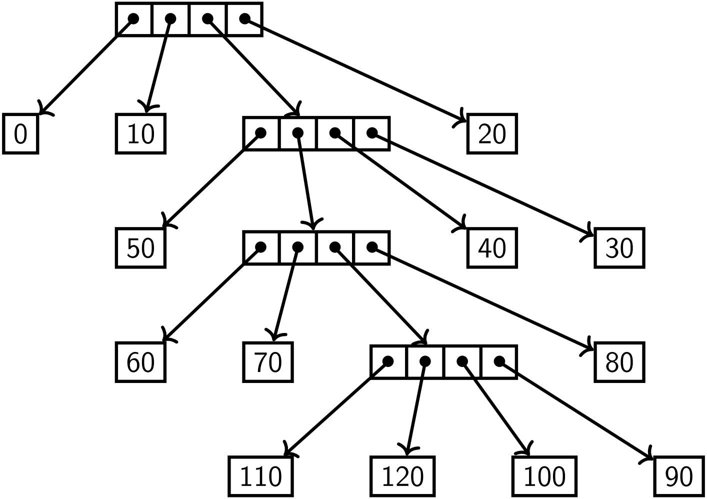
# Quad Trees

A *Quad Tree* is a particular kind of tree that differs from the binary tree that we have looked at so far in two respects:

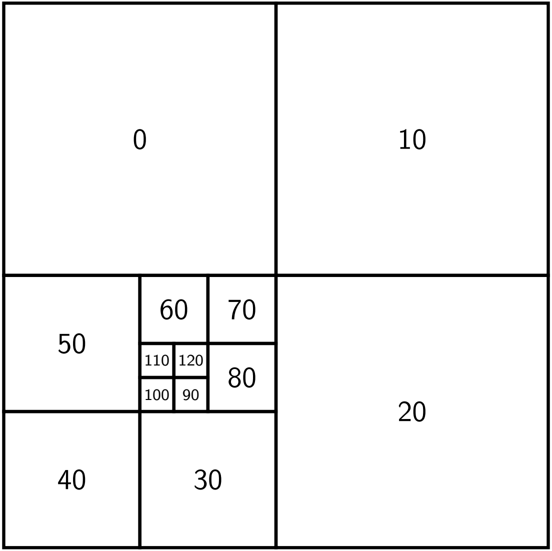
- The values are ONLY at the leaf level: there are no values in internal nodes of the tree
- Internal nodes have 4 children

This data structure is particularly useful in representing and manipulating some kinds of 2-dimensional data such as images. A typical application is in image compression.

# Quad Trees



## Quad Trees



# Quad Tree ADT

- Constructors:
  - `baseQT`: returns a single, leaf node quad tree with a value
  - `MakeQT(luqt, ruqt, llqt, rlqt)`: returns a new quad tree built from four sub-quad trees.
- Accessors:
  - `isValue(qt)`: return true if `qt` is a value node quad tree, otherwise returns false
  - `lu(qt)`: returns the left upper sub-quad tree of `qt` [2]
  - `ru(qt)`: returns the right upper sub-quad tree of `qt` [2]
  - `ll(qt)`: returns the left lower sub-quad tree of `qt` [2]
  - `rl(qt)`: returns the right lower sub-quad tree of `qt` [2]

If `qt` is a value node quad tree, then we conventionally refer to the value stored in the node as `qt`, rather than define another accessor `value(qt)` for that purpose.

[2] Triggers an error if the `qt` is a value node quad tree

# Example

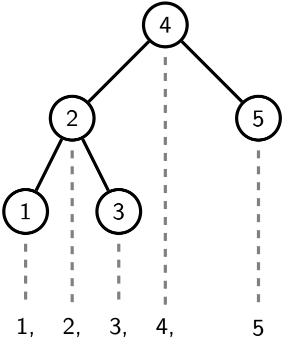
```
rotate(qt) {
  if ( isValue(qt) )
    return qt
  else
    return makeQT( rotate(rl(qt)),
                   rotate(ll(qt)),
                   rotate(ru(qt)),
                   rotate(lu(qt)) )
```

# Binary Search Trees

# Binary Search Trees

A Binary Search Tree is a tree which is either **empty** or

1. values in the left subtree are **smaller** than in the root
2. values in the right subtree are **larger** than in the root
3. root's left and right subtrees are also Binary Search Trees



$\Longrightarrow$ values in the flattened Binary Search Tree are in order!

A Binary Search Tree is a Binary Tree, so the same constructors and accessors apply. It is just that there is an extra constraint that the node value ordering must be maintained during construction and manipulation.

# Binary Search Trees: Insertion

```
insert(v,bst) {
  if ( isEmpty(bst) )
    return MakeTree(v, EmptyTree, EmptyTree)
  elseif ( v < root(bst) )
    return MakeTree(root(bst),
                    insert(v,left(bst)),
                    right(bst))
  elseif ( v > root(bst) )
    return MakeTree(root(bst),
                     left(bst),
                     insert(v,right(bst)))
  else error(”Error:␣value␣already␣in␣tree”)
}
```

This creates a new BST which is a copy of the old one but with the extra value inserted correctly. For many applications, this is inefficient because we usually do not need a new copy; we usually want to update the data structure we have in place.

# Binary Search Trees: Insertion in Java

We can instead insert the value in place, modifying an existing BST if we use pointers. Here is an implementation in Java (full working program is in Canvas):

```java
public class BSTTree {
  private BSTNode tree = null;

  private static class Node {
    private int     val;
    private Node left, right;

    public BSTNode(int val, Node left, Node right){
      this.val=val; this.left=left; this.right=right;
    }

  // insert methods here
}
```

# Binary Search Trees: Insertion in Java

```java
public void insert(int v) {
  if (tree == null) tree = new Node(v, null, null);
  else insert(v, tree);
}

private void insert(int v, Node ptr) {
  if (v < ptr.val) {
    if (ptr.left == null)
      ptr.left = new Node(v, null, null);
    else insert(v, ptr.left);
  }
  else if (v > ptr.val) {
    if (ptr.right == null)
      ptr.right = new Node(v, null, null);
    else insert(v, ptr.right);
  }
  else throw new Error("Value␣already␣in␣tree");
}
```

# Searching Binary Search Trees

Starting from the root node, how do we determine whether a value `x` is in the tree?

If the tree is empty, `x` is not in the tree! Otherwise, compare `x` and the value stored in the root. There are three possibilities:

- They are equal $\implies$ we found it!
- `x` is smaller $\implies$ we search the left subtree.
- `x` is larger $\implies$ we search the right subtree.

# Searching Binary Search Trees Recursively

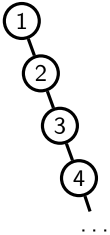
```
isIn(value v, tree t) {
    if ( isEmpty(t) )
        return false
    elseif ( v == root(t) )
        return true
    elseif ( v < root(t) )
        return isIn(v, left(t))
    else
        return isIn(v, right(t))
}
```

# Searching Binary Search Trees Iteratively

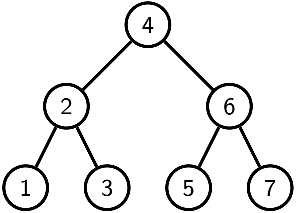
```
isIn(value v, tree t) {
    while ( (not isEmpty(t)) and (v != root(t)) )
        if (v < root(t) )
            t = left(t)
        else
            t = right(t)
    return ( not isEmpty(t) )
}
```

## Searching Binary Search Trees

Compare complexities:



vs.



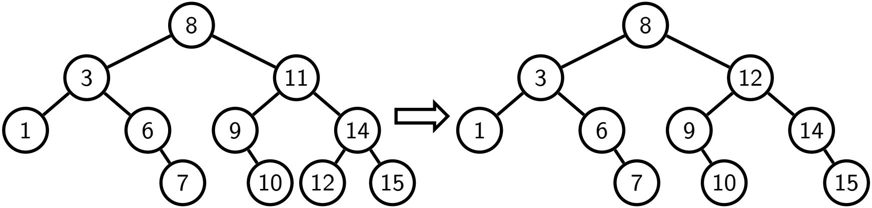
- For the left case, complexity of search is $O(n)$
- For the right case, complexity of search is $O(\log_2 n)$
- For the average case (not proven here), complexity of search is also $O(\log_2 n)$
- On average, complexity of insertion is $O(\log_2 n)$, because it depends on the height of the Binary Search Tree, which is $O(\log_2 n)$
- Average height of a General Binary Tree is $O(\sqrt{n})$

# Deleting from a Binary Search Tree

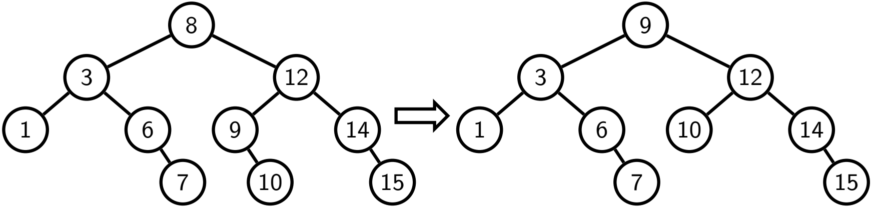
- First Option:
  - Insert all items except the one to be deleted into a new BST
  - $n$ inserts of complexity $O(\log_2 n)$, results in complexity $O(n \log_2 n)$
  - Worse than deleting from an array: $O(n)$
- Second Option:
  1. Find the node containing the element to be deleted:
  2. If it is a leaf, just remove it
  3. Else, if only one of the node's children is not empty, replace the node with the root of the non-empty subtree
  4. Else,
     - 4.1 find the left-most node in the right sub-tree (this contains the smallest value in the right sub-tree)
     - 4.2 replace the value to be deleted with that of the left-most node
     - 4.3 replace the left-most node with its right child (may be empty)
  - Complexity $O(\log_2 n)$

# Deleting from a Binary Search Tree

Delete 11:



Delete 8:



# Deleting from a Binary Search Tree

```
delete(value v, tree t) {
 if ( isEmpty(t) )
   error("Error:␣given␣item␣is␣not␣in␣given␣tree")
 else
   if ( v < root(t) )
     return MakeTree(root(t), delete(v,left(t)), right(t))
   else if ( v > root(t) )
     return MakeTree(root(t), left(t), delete(v,right(t)))
   else
     if ( isEmpty(left(t)) )
       return right(t)
     elseif ( isEmpty(right(t)) )
       return left(t)
     else return
       MakeTree(smallestNode(right(t)), left(t),
                             removeSmallestNode(right(t))
}
```

# Deleting from a Binary Search Tree

```
smallestNode(tree t) {
  // Precondition: t is a non−empty binary search tree
  if ( isEmpty(left(t) )
     return root(t)
  else
     return smallestNode(left(t));
}

removeSmallestNode(tree t) {
  // Precondition: t is a non−empty binary search tree
  if ( isEmpty(left(t) )
     return right(t)
  else
     return MakeTree(root(t),
                     removeSmallestNode(left(t)),
                     right(t))
}
```

# Checking whether a Binary Tree is a BST

Simple algorithm:

1. If the tree is empty, it is a valid BST
2. Else, it is a valid BST if:
   - 2.1 all values in the left branch are less than the root and
   - 2.2 all values in the right branch are greater than the root and
   - 2.3 the left branch is a valid BST and
   - 2.4 the right branch is a valid BST

```
isbst(tree t) {
    if ( isEmpty(t) )
      return true
    else
      return ( allsmaller(left(t),root(t)) and
               isbst(left(t)) and
               allbigger(right(t),root(t)) and
               isbst(right(t)) )
}
```

# Checking whether a Binary Tree is a BST

```
allsmaller(tree t, value v) {
   if ( isEmpty(t) )
      return true
   else
      return ( (root(t) < v) and
               allsmaller(left(t),v) and
               allsmaller(right(t),v) )
}

allbigger(tree t, value v) {
  if ( isEmpty(t) )
    return true
  else
    return ( (root(t) > v) and
             allbigger(left(t),v) and
             allbigger(right(t),v) )
}
```

# Checking whether a Binary Tree is a BST

That was a simple algorithm, but inefficient. Exercise:

1. Count the number of comparisons that occur during the execution of this algorithm on a BST of size 7 and height 2
2. Repeat the exercise on a BST of size 15 and height 3.
3. What conclusions can you draw about the complexity of this algorithm?
4. A perfect Binary Tree is one where every internal node has two children and all the leaf nodes are at the same level (which means that the leaf nodes fill that level). What is the minimum number of comparisons, as a function of the size of the tree, that are truly necessary for any algorithm to check whether a perfect Binary Tree is a BST?
5. Figure out an algorithm that matches the best complexity possible for checking that a Binary Tree is a BST.

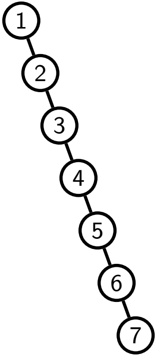
# Sorting using BSTs

```
printInOrder(tree t) {
  if ( not isEmpty(t) ) {
    printInOrder(left(t))
    print(root(t))
    printInOrder(right(t))
  }
}

sort(array a of size n) {
  t = EmptyTree
  for i = 0,1,...,n-1
    t = insert(a[i],t)
  printInOrder(t)
}
```

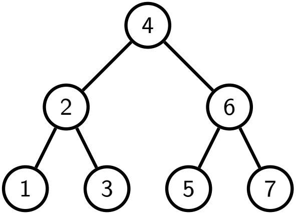
Exercise: Modify the above code to put the values back in the array in order instead of printing them out. What is the complexity of sorting an array this way?

# AVL Trees

# Balancedness of trees matters



vs.



Can we assume extra conditions to make sure that the height of the tree is under control?

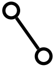
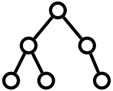
# AVL Tree

The **height** of a node is the length of the longest path from that node to a leaf node (compare to the height of a tree)

The **balance** at a node is

$$\begin{pmatrix}\text{The height of} \\ \text{the left subtree}\end{pmatrix} - \begin{pmatrix}\text{The height of} \\ \text{the right subtree}\end{pmatrix}$$

Examples:

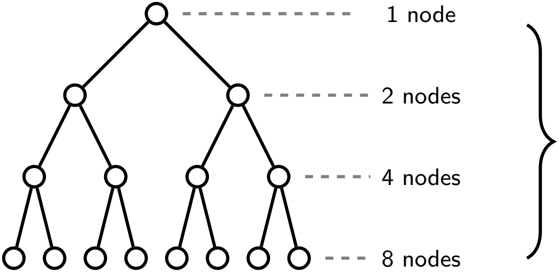
- Note that the height of an empty tree is $-1$
- The balance at a leaf node is $(-1) - (-1) = 0$.
- The balance at the root of  is $(-1) - 0 = -1$.
- The balance of the root of  is $1 - 1 = 0$.

## AVL Tree

**Definition:** A Binary Search Tree is said to be **AVL** when the balance at *every* node is either 1, 0 or $-1$.

# Perfect Binary Tree = Maximal AVL tree of a given height

Assume that the tree is **perfectly balanced**, that is, the balance of each node is 0. How many nodes does the tree have?



each level has **twice** as many nodes as the previous level

If the tree has height $h$, then the number of nodes is

$$1 + 2 + 4 + 8 + \cdots + 2^h = 2^{h+1} - 1$$

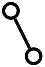
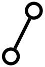
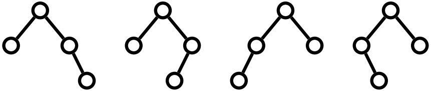
Another way of saying that the tree is perfectly balanced is that

1. every node, except for leaf nodes, has exactly two children and
2. all leaf nodes are on the same level.

# AVL-Tree: Worst Case Imbalance

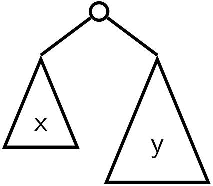
# Fibonacci trees = Minimal AVL trees of a given height

How many nodes does the tree have if the balance of each (non-leaf) node is either 1 or -1?

- If the height is 1 – two options:  or  $\Longrightarrow$ size is 2
- If the height is 2: 

  $\Longrightarrow$ size is *always* 4

- In general, we obtain the **Fibonacci tree of height h+2** (called $T_{h+2}$), from the Fibonacci trees of height $h$ and $h+1$ (called $T_h$ and $T_{h+1}$, respectively) as:



$\Longrightarrow$ the size of $T_{h+2} = 1 +$ size of $T_h +$ size of $T_{h+1}$

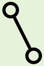
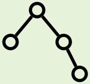
We see that there are two **minimal** AVL trees of height 1 and four **minimal** AVL trees of height 2. However, those minimal trees are all the same, except for the ordering of children. Similarly, the minimal AVL trees of larger heights are also of the same size.

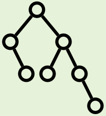
For now, we are only interested in the **size** of a minimal AVL tree of a certain height. Because all minimal AVL trees of a given height have the same size, we can pick just one representative AVL tree for every height.

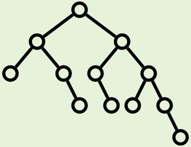
The following procedure describes a construction of **Fibonacci trees** $T_{-1}, T_0, T_1, T_2, T_3, \dots,$ where $T_h$ is the minimal AVL tree of height $h$ (up to ordering of children):

- $T_{-1}$ is the empty tree
- $T_0$ is the one element tree
- $T_{h+2}$ is obtained by making $T_h$ and $T_{h+1}$ children of the root node (as shown in the picture on the previous slide).

For example, to construct $T_3$ we combine $T_1$ and $T_2$. Because

$T_1 =$  $T_2 =$ 

we obtain that $T_3$ is the following tree 

and $T_4$ is the following tree 

and so on.

# Fibonacci trees and Fibonacci numbers

Denote the size of $T_h$ as $|T_h|$:

| h | $\lvert T_h \rvert$ |
|---|---|
| -1 | 0 |
| 0 | 1 |
| 1 | $1 + \lvert T_{-1}\rvert + \lvert T_0\rvert = 2$ |
| 2 | $1 + \lvert T_0\rvert + \lvert T_1\rvert = 4$ |
| 3 | $1 + \lvert T_1\rvert + \lvert T_2\rvert = 7$ |
| 4 | $1 + \lvert T_2\rvert + \lvert T_3\rvert = 12$ |
| 5 | $1 + \lvert T_3\rvert + \lvert T_4\rvert = 20$ |
| $\vdots$ | $\vdots$ |

| k | $F_k$ |
|---|---|
| 0 | 0 |
| 1 | 1 |
| 2 | $F_0 + F_1 = 1$ |
| 3 | $F_1 + F_2 = 2$ |
| 4 | $F_2 + F_3 = 3$ |
| 5 | $F_3 + F_4 = 5$ |
| 6 | $F_4 + F_5 = 8$ |
| $\vdots$ | $\vdots$ |

$$|T_{h+2}| = 1 + |T_h| + |T_{h+1}|$$
$$1 + |T_{h+2}| = (1 + |T_h|) + (1 + |T_{h+1}|)$$

vs $\quad F_{k+2} = F_k + F_{k+1}$

initial values plus equation $\implies |T_h| + 1 = F_{h+3}$

# Computing the bounds

If an AVL tree has height $h$ then its size is

- $\leq$ the size of the perfectly balanced tree of height $h$, and
- $\geq$ the size of Fibonacci tree of height $h$ (that is, $|T_h|$).

Therefore (because $|T_h| = F_{h+3} - 1$)

$$F_{h+3} - 1 \leq \text{the size of the tree} \leq 2^{h+1} - 1$$

Binet's formula: $F_k = \dfrac{\left(\frac{\sqrt{5}+1}{2}\right)^k - \left(\frac{\sqrt{5}-1}{2}\right)^k}{\sqrt{5}} \approx O(1.61^k)$

$\Longrightarrow$ the size of an AVL tree is exponential in its height

$\Longrightarrow$ the height of an AVL tree is logarithmic in its size

$\Longrightarrow$ **an AVL tree of size $n$ has height $\mathcal{O}(\log n)$**

If we have a tree of height $h$ which is AVL, we know that the size of our tree could be as small as $|T_h|$, or as big as the size of the perfectly balanced tree of height $h$. However, in general it is somewhere in between.

Let $n$ be the size of an AVL tree, then we have that

$$F_{h+3} - 1 \leq n \leq 2^{h+1} - 1$$

These are conditions on size, given that we know the height of our AVL tree. Conversely, if we know the size and we know that the tree is AVL, then what implications does this have for the height? Let's express the conditions for height in terms of $n$.

For example $n \leq 2^{h+1} - 1$, gives us that

$$\log_2 n \leq \log_2(2^{h+1} - 1) \leq \log_2(2^{h+1}) = h + 1.$$

In other words, height $h$ is at least $\log_2 n - 1$.

# Consequences for time complexities

For a Binary Search Tree implemented as a height-balanced tree (e.g. AVL tree), where $n =$ the number of nodes of the tree:

- `search(x)` goes through at most $O(\log n)$-many levels
  $\Longrightarrow$ $O(\log n)$ steps
- `insert(x)`:
  1. We first find the leaf where to insert `x` $\Longrightarrow$ $O(\log n)$ steps,
  2. then, insert it there $\Longrightarrow$ $O(1)$ steps,
  3. finally, on the way up, in each node we do balancing
     $\Longrightarrow$ $O(\log n)$-many times we do $O(1)$ steps
     $\Longrightarrow$ $O(\log n)$ steps in total
- `delete(x)` is similar to `insert`, it also takes $O(\log n)$ steps

# AVL-Tree Operations

# AVL tree operations

**AVL Trees Invariant:** The balance of every node is -1, 0, or 1. When inserting an element to an AVL tree we allow breaking the invariant and then, by re-balancing, we fix it again.

- AVL find: Same as BST find
- AVL insert:
  - First BST insert, then check balance and potentially fix the AVL tree
  - Four different balance cases
- AVL Delete: like insert we do the deletion and then have several balance cases

# AVL re-balancing via Rotations

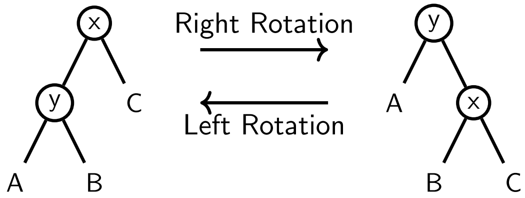
When we insert into an AVL tree, all nodes meet the balance invariant initially.

We find where the value should go, just like in a BST tree, and insert a new leaf there.

However, that may break the balance invariant of the AVL tree.

# AVL re-balancing via Rotations

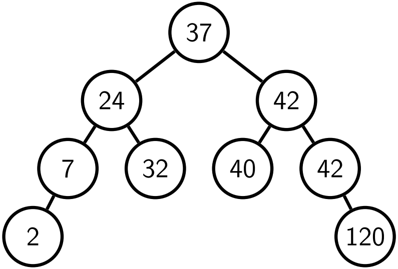
We will fix imbalances by a series of rotations:



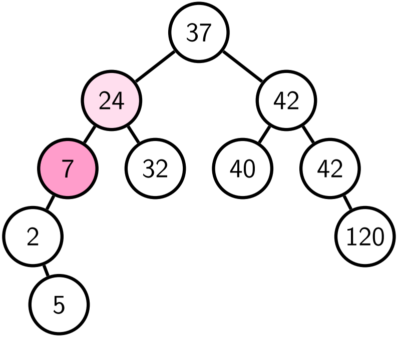
- $A < y < B < x < C$: rotation preserves this order
- $x$'s right child ($C$) remains unchanged
- $y$'s left child ($A$) remains unchanged
- Right rotation:
  - $y$'s right child ($B$) becomes $x$'s left child
  - $x$ becomes $y$'s right child
- Left rotation:
  - $x$'s left child ($B$) becomes $y$'s right child
  - $y$ becomes $x$'s left child

# AVL tree insert example[1]

AVL Tree



After inserting 5,
before rebalance

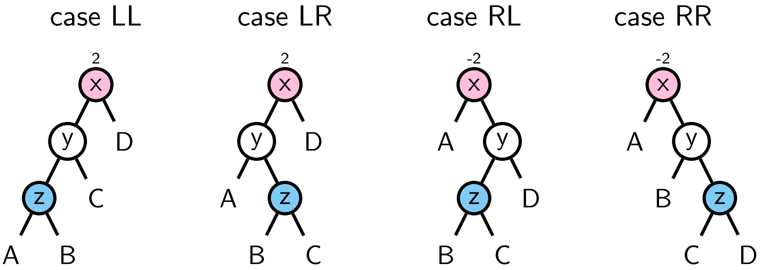


We only find the imbalance in a node on *return* from the insert call to its child node, and fix the *lowest* node with an imbalance first.

[1] Shaffer, *Data Structures and Algorithm Analysis*

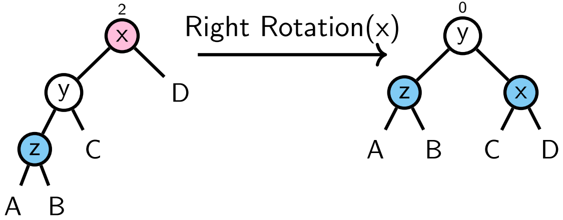
# AVL tree insert

Let x be the lowest node where an imbalance occurs, following an insert into subtree z. This imbalance is found on returning from the nested recursive insert call up to node x. There are 4 different cases possible:



## AVL tree insert: Case LL

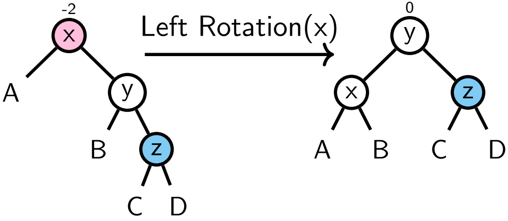
This can be fixed with a *right rotation* at $x$:



- Before insertion, balance at $x$ had to be 1
- Inserting into $z$ caused imbalance at $x$, so, after insertion but before rotation, $h(y) = h(D) + 2$
- After insertion, but before rebalancing, $h(y - z - \dots) = h(D) + 2$ and $h(y - C - \dots) = h(D) + 1$ (otherwise $y$ would be the lowest node with imbalance)
- After rotation, balance at $y$ is 0

# AVL tree insert: Case RR

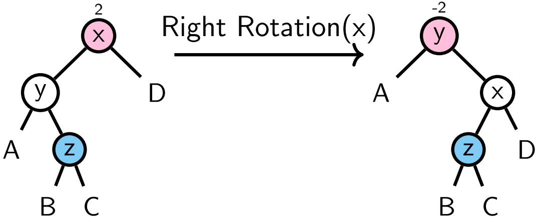
This case is symmetric to LL and can be fixed with a *left rotation* at $x$:



- Before insertion, balance at $x$ had to be $-1$
- After rotation, balance at $y$ is 0

# AVL tree insert: Case LR

This case raises a probem: the necessary right rotation at $x$ alone does not fix the imbalance:



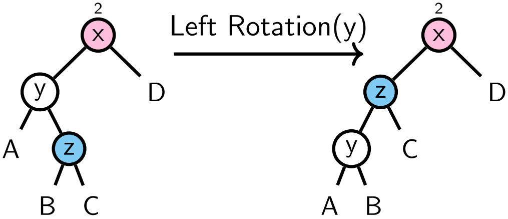
**STILL UNBALANCED . . . just the opposite way!**

Actually turns it into the RL case

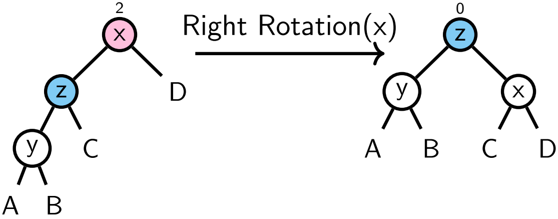
Solution:

- Do a **left rotation** at $y$ first
- Then do a **right rotation** at $x$.

# AVL tree insert: Solution to Case LR

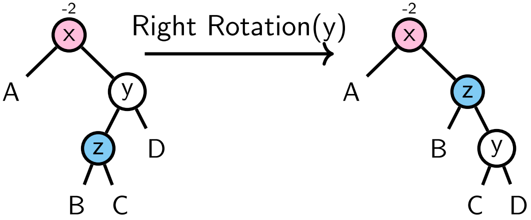


This results in a simple LL case which can be fixed by a right rotation at *x*:

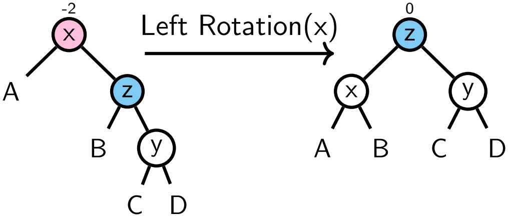


# AVL tree insert: Case RL

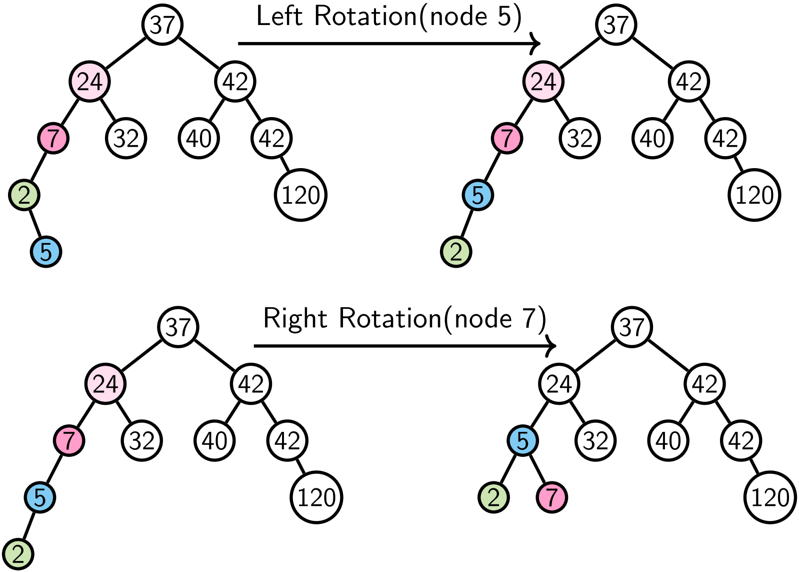
This case is symmetric to the LR case



This results in a simple RR case which can be fixed by a left rotation at $x$:



# AVL tree insert example [Shaffer]



Left Rotation(node 5)

Right Rotation(node 7)

# AVL tree deletion

To delete from an AVL tree, the general approach is to modify the BST delete algorithm (c.f. `dsa-slides-04-03-binary-search-trees.pdf`):

- Delete the node from the tree using the BST algorithm
- On returning up the tree, rebalance as necessary just as for AVL Tree insert

## Further reading on AVL trees

- https://www.programiz.com/dsa/avl-tree has a very nice explanation, explains deletion as well and has full code implementations.
- https://www.tutorialspoint.com/data_structures_algorithms/avl_tree_algorithm.htm also has some nice explanations.
- https://www.cs.usfca.edu/~galles/visualization/AVLtree.html allows you to insert and delete values in an AVL tree and animates the operations.

# B-Trees and B+Trees

# B-Tree

A B-tree of order $m$ is a type of n-ary trees with some particularly nice properties:

- Every node has at most $m$ children
- Every non-leaf node, except the root, has at least $m/2$ children
- The root node, if it is not a leaf node, has at least 2 children
- A non-leaf node with $c$ children, contains $c - 1$ search key values, which act as separators or *discriminators*, to guide seaches down appropriate sub-trees
- All leaf nodes appear in the same level
- B-Trees are always height balanced
- Update and search is $O(\log n)$

# B-Tree

In fact, different authors define the B-Tree in slightly different ways.

This does not really matter, because no-one really uses basic B-Trees:

- When used as an in-memory data structure, they have no significant advantage over AVL-trees or other height balanced binary trees like 2-3 trees or red-black trees, and are a bit more complex to implement.

Instead, a variant of the B-Tree called a B+Tree is the main-stay of databases and the most comment *external* data structure in use today.

We will not consider the basic B-Tree further but concentrate on the B+Tree.

# External Data Structures

# External Data Structures

An external data structure is one which is stored on external or secondary memory (i.e. disks) rather than in internal memory (RAM). This means that the data structure can:

- Persist beyond the end of the program execution without the programmer having to explicitly save or load the structure to/from disk
- Grow to sizes larger than can fit in internal memory, often hugely larger, being limited only by the amount of secondary storage available.
- Be accessed and updated by multiple different programs at the same time so long as suitable coordination protocols are observed by the different programs.

# Properties of Secondary Storage

- Disks store data in block of sizes configured by the operating system, usually 4KBytes but can be up to 64KBytes
- The disk can only transfer whole blocks at a time: to write a single byte to disk, the operating system would have to
    1. read the block, which would contain the byte, into memory
    2. replace the byte in the memory copy of the disk block
    3. write the memory copy of the block back out to the disk
- In spinning hard disks, the time to read (or write) a block includes
    1. time to move the disk head to the correct track (approx 6ms),
    2. time to wait for the block to spin around to the disk head (approx 4ms) and
    3. time for the disk to spin further until all the block has passed under the disk head.
    4. reading or writing consecutive blocks is much faster that reading a random sequence of disk blocks

# B+Trees

# Glossary for B+Trees

There are a few terms we use in B+Trees

- **Data Record**: an element of data information to be managed by the B+Tree, e.g. a student record with ID number, name, email address etc.
- **Key value**: the value by which we identify the record, e.g. the ID number or the student name.
- **Discriminator**: a value used to decide which path to take down the tree in searching for a record. Almost always the key value of some record, but, in principle, it doesn't have to be.
- **Disk Address**: the offset from the start of the disk file to a particular block in the file. A file read/write can be executed by rquesting the operating system to "*seek*" to this address and read/write a block from/to the file. Think of a disk address as a pointer to disk memory.

# Order of a B+Tree

B+Trees nodes are designed to fit in disk blocks so that reading or writing a node corresponds to reading or writing a single disk block (or a sequence of consecutive disk blocks)

Most descriptions of B+Trees start with the *order* of a B+Tree, which is variously defined as the minimum or maximum number of children or the minimum or maximum number of keys in a non-root internal node of the tree (these can be 4 different numbers for the same tree!)

However, in real-world B+Tree implementations, first of all the key values are often variable length strings, and second the the limiting factor on the number of children in each node is not some arbitrary order specification, but instead is decided by how many keys and disk addresses can fit in a single disk block.

- Note that, since keys can be of variable length, the maximum number of children of an internal node of the tree is not a fixed number.

# B+Trees

There are 2 variants of B+Trees in common use which we can call *Record-Embedded B+Trees*[1] and *Index B+Trees*[1]

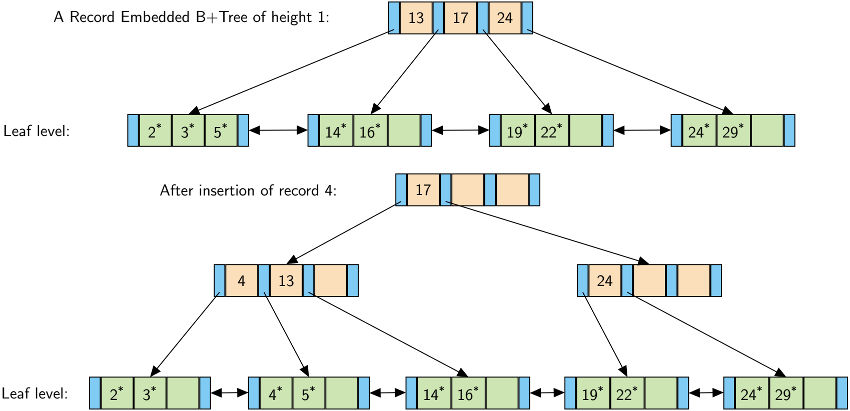
- **Record-Embedded B+Trees** store the data records in the leaf nodes of the B+Tree. This is a suitable data structure when
    - you only need to find records by one search key, e.g. you look up student records by student ID numbers, but not by student name.
    - you want the B+Tree to do all management of the data records, e.g. if you delete the B+Tree, you delete the student records.
- **Index B+Trees** store only key-values and disk addresses in the leaf nodes, which identifies the disk block in the separate file of data records where the record with the corresponding key values reside:
    - The data records are kept in blocks separate from the B+Tree blocks, so dropping the B+Tree does not delete the data records
    - Multiple B+Tree indexes can be created on a collection of data records, e.g. one indexing on student IDs, another on student names.

[1] Not standard terminology, but there is none for these variants

# Record Embedded B+Trees

# Record Embedded B+Tree

| | |
|---|---|
| ▮ | A disk address |
| 24 | A key value |
| $24^*$ | A full record with given key value |



A Record Embedded B+Tree of height 1:

Leaf level:

After insertion of record 4:

Leaf level:

# Search Operations

- **Search:** Start at the root and follows the path down indicated by the discriminator values: go to the node identified by the disk address whose left discriminator is less than the search key and whose right discriminator is greater than or equal to the search key.
- **Range:** Search for the record with a key at one end of the range, then iterate, using the next/prev disk addresses in the leaf level, over all records in the range.

## Insert Operation

- Search with the key of the record to be inserted to find the location to insert the record. If there is space there, insert the record and done.
- If there is not enough space there, split the block in two so that approximately half the number of records go to the left, half to the right (the new record is added to the appropriate half).
- **Post** (i.e. insert to the level above) the key value of the lowest record in the right page as a discriminator to the level above
- If there is room in the block above for this insertion then done.
- Otherwise, continue splitting and posting until either the insertion is complete or the root node of the tree is split, in which case there is guaranteed to be sufficient room because the resulting new root node will only have 1 key and two disk addresses after the split.

# Insert Operation Properties

- The tree grows in height **ONLY** when the root node splits, hence every path from the root to any leaf is of the same height at all times: i.e. the tree is height balanced
- No node is ever less than (approximately) half full except for the root node
- Deletion works as an inverse of insertion: whenever a record is removed from a leaf node, if the node is becomes less than half full, then the records of it and its neighbouring node is distributed between them. If both the nodes become less than half full, then the nodes are merged and an entry removed from the level above. This can cascade up the tree until, possibly, the two children of the root node are merged and become the new root and the tree reduces in height by 1.

# Bulk Loading

Creating a new B+Tree index on a set of records can be done by iterating over the records and inserting them into the B+Tree. However, this is inefficient because of the many searches down the tree to find the location to insert the records.

Instead bulk loading is much more efficient:

- Sort the records and insert them into a leaf level set of records, connecting the leaf blocks as you go.
- As you construct leaf blocks, construct a parent node by inserting leaf disk addresses and discriminators into the parent node.
- When a parent node becomes full, split it as usual for insert

This results in all the splits happening along the rightmost path in the tree, rather than randomly distributed across many different paths, which in turn means that, even with very large trees that do not fit in memory, one can hold the rightmost path in memory and only write blocks when they become full, resulting in much greater performance.

# Index B+Trees

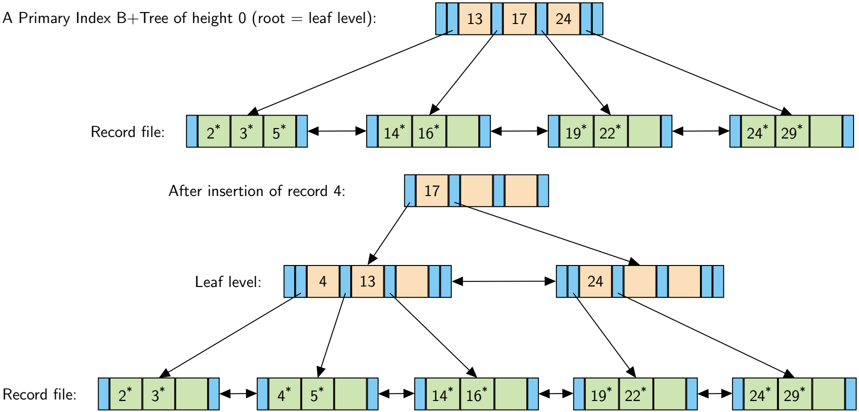
## Index B+Tree

There are two sub-variants of the Index B+ depending on the sort order (if any) of the data records in the record file:

- **Secondary Index B+Tree:** Here the records are **NOT** necessarily sorted in the data file of blocks. Thus each entry in the leaf nodes of the tree contains a pair consisting of a key value and a disk address which identifies the disk block where the record with that key value can be found. Leaf nodes also have forward and reverse pointers.
- **Primary Index B+Tree:** Here the records are kept sorted by the key value used in the B+Tree in the data file containing the blocks of records. Thus the leaf nodes of the B+tree only need to store discriminator values to separate the data file blocks and look much like internal B+Tree nodes except that they also have forward and reverse pointers.

There can be only one primary key index on a file of records as the records can be in only one order, but there can be multiple secondary indexes on the same file.

# Primary Index B+Tree



A Primary Index B+Tree of height 0 (root = leaf level):

Record file:

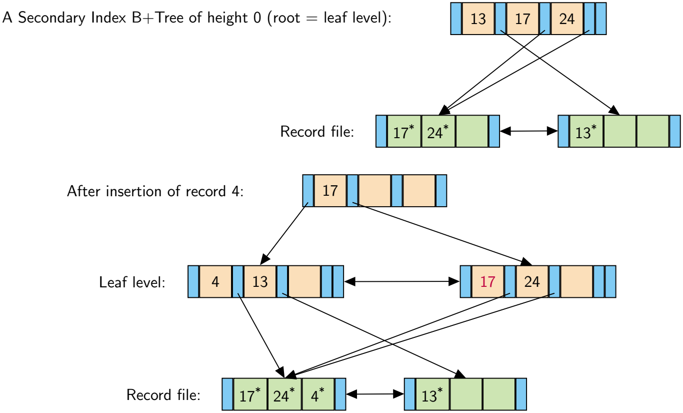
After insertion of record 4:

Leaf level:

Record file:

Note that the leaf level of the Primary B+Tree contains discriminators to separate blocks of records in the record file

# Secondary Index B+Tree


A Secondary Index B+Tree of height 0 (root = leaf level):
13
17
24
Record file:
17*
24*
13*
After insertion of record 4:
17
Leaf level:
4
13
17
24
Record file:
17*
24*
4*
13*

Note that every value in the record file has an individual entry in the leaf level of the secondary B+Tree, hence we see key values in the leaf level that may occur higher in the tree, e.g. 17 in the above case.

# B+Tree Complexity

# B+Tree Complexity

The B+Tree is an n-ary tree which is height balanced, hence search is going to be $O(\log n)$

In practice, search will take $log_m n$ reads of pages where the fanout ratio of the tree is $m$ (The fanout ratio is the number of children in each node). With a 4KByte block, a 4 byte integer key, a 4 byte disk address size, with every block being half full, and even assuming a little space is in each block is taken with extra administrative information, that gives a fanout ratio of at least 250.

- Tree of height 1: 250 records
- Tree of height 2: 62,500 records
- Tree of height 3: 15,625,000 records
- Tree of height 4: 3,906,250,000 records

# B+Tree Complexity

This is actually an underestimate because the nodes will, on average have more entries in them and the block sizes used will usually be larger than 4Kbytes

Thus can find a record from a collection of approx 4 Trillion in 4 disk reads.

In practice, we would normally cache all except the bottom two levels in memory, so really only takes 2 disk reads.

Note that the cost of the disk read is so much larger than the cost of the processing of in-memory aspects of the data structure that we can ignore in-memory processing costs.

Insertion has the same order of complexity as search.

# Priority Queues

## Priority Queue

A Priority Queue is an Abstract Data Type that maintains a collection of items which has an associated *priority* value and can get (and remove) the item with highest priority efficiently. New items with arbitrary priorities can be added at any time.

There are a number of obvious implementation strategies we could use:

- Unsorted Array: Inserting an item is $O(1)$, `get` is $O(n)$
- Sorted Array: Insert is $O(n)$, `get` is $O(1)$
- AVL Tree: Insert and `get` are both $O(\log n)$

We can do better than that.

# Priority Queue applications

In general, priority queues are useful whenever we repeatedly need the maximum of a changing collection.

For example, if do a search on the web, the underlying search algorithm will find potential matches, each with a score that estimates how good a match it is. It may put those matches into a priority queue, and then extract the 10 best matches from the queue. In the background, further matches might be searched for and added to the queue. When the go to the next page of results, then the next 10 best matches are extracted and displayed.

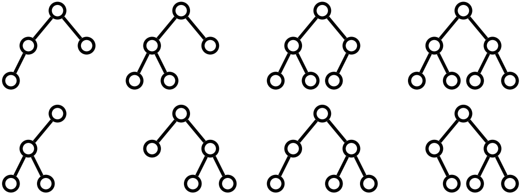
Similar examples can be found in finding the best next move in a computer game, online flight search websites, hotel booking systems, comparative pricing websites

# Complete Binary Tree

Our first implementation of Priority Queues will use a type of **Complete Binary Tree**, called a **Binary Heap Tree**, which, in turn, we will implement using an array.

**Definition**
A binary tree is *complete* if every level, except possibly the last, is completely filled, and all the leaves on the last level are placed as far to the left as possible.

This simply defines, for a binary tree with a given number of nodes, the tidiest, most balanced and compact form possible

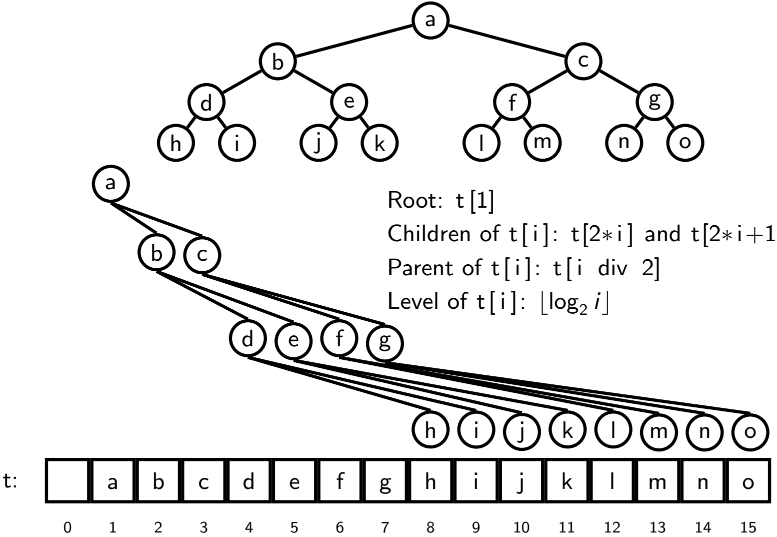


Complete:

Not Complete:

# Array Implementation of Complete Binary Trees

We can store Binary Trees in a simple array structure:



Root: t [1]

Children of t[ i ]: t[2∗i] and t[2∗i+1]

Parent of t[ i ]: t[ i div 2]

Level of t[ i ]: $\lfloor \log_2 i \rfloor$

| t: | | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |

If the binary tree is not complete, then:

- this is a wasteful implementation strategy because space is still reserved for the missing nodes
- when a node is missing, the array cell needs to be marked to indicate that it is not part of the tree
  - use an invalid value for this application (null, -1, etc.), or
  - use a separate data structure (e.g. a bit array — 1 bit per cell of the binary tree array) to indicate whether the corresponding cell contains a real value

If the binary tree is always complete, then:

- A single integer to record number of nodes in the tree is sufficient to identify the end of the tree
- You do not need to mark missing nodes because there are none before the end of the tree

# Binary Heap Trees

# Binary Heap Trees

**Definition**
A *binary heap tree* is a complete binary tree which is either empty or satisfies the following conditions:

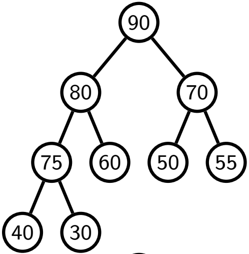
1. The priority of the root is greater than or equal to that of its children.
2. The left and right subtrees of the root are heap trees.

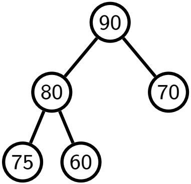
Note that, unlike in Binary Search Trees, there is no restriction on the relationship between the left and right children of any node: Binary Heap Trees do not keep the value stored in left child node less than that stored in the right child node.

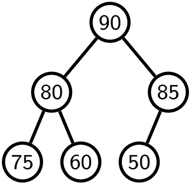
Thus we have no need to keep the Binary Heap Tree sorted, which makes working with the tree somewhat easier.

# Binary Heap Tree Examples

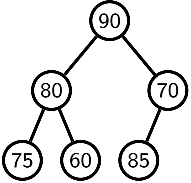
Valid:

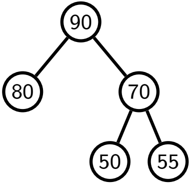


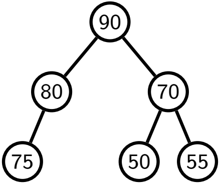




Invalid:







# Priority Heap: Java-like Pseudo-Code

WARNING: Necessary error checking has been omitted!

```
public class PriorityHeap{
  private int MAX = 100;
  private int heap[MAX+1];
  private int n = 0;

  public int value(int i){
    if (i < 1 or i > n)
      throw IndexOutOfBoundsException;
    return heap[i];
  }
  public boolean isRoot(int i) { return i == 1; }
  public int level(int i)      { return log(i); }
  public int parent(int i)     { return i / 2; }
  public int left(int i)       { return 2 * i; }
  public int right(int i)      { return 2 * i + 1; }
  // More methods to be added here
}
```

# Priority Heap: Java-like Pseudo-Code

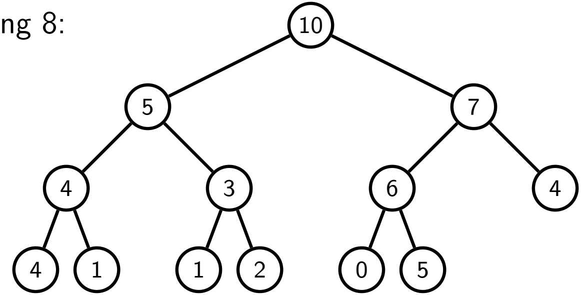
```
public boolean isEmpty() {
  return n == 0;
}

public int root() {
  if ( heapEmpty() )
      throw HeapEmptyException;
   else return heap[1]
}

public int lastLeaf()) {
   if ( heapEmpty() )
      throw HeapEmptyException;
   else return heap[n]
}
```

# Binary Heap Tree Insertion

# Insertion

**Idea:** Insert the value at the end of the last level and then keep bubbling it up as long as it is larger than its parent.

Inserting 8:

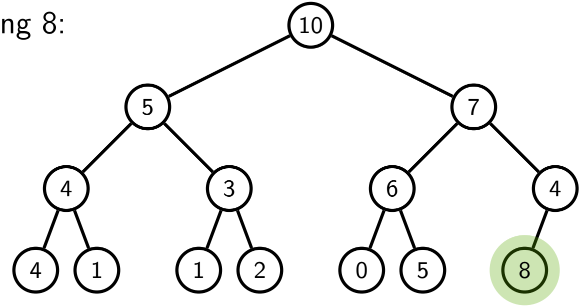


As we bubble up, when we swap the value of a node *i* with that of its parent, we don't have to compare *i* with its sibling, because if the value of *i* is greater than that of its parent, then it must be greater than that of its sibling because of the Binary Heap Tree property.

## Insertion

**Idea:** Insert the value at the end of the last level and then keep bubbling it up as long as it is larger than its parent.

Inserting 8:

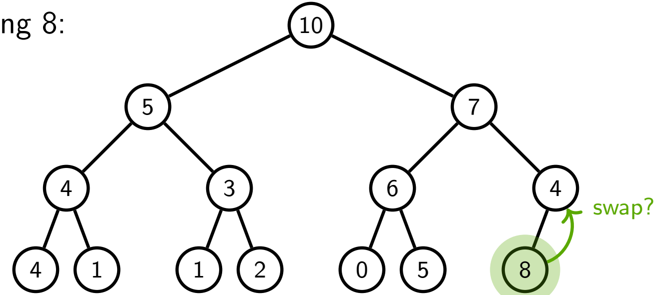


As we bubble up, when we swap the value of a node *i* with that of its parent, we don't have to compare *i* with its sibling, because if the value of *i* is greater than that of its parent, then it must be greater than that of its sibling because of the Binary Heap Tree property.

## Insertion

**Idea:** Insert the value at the end of the last level and then keep bubbling it up as long as it is larger than its parent.

Inserting 8:

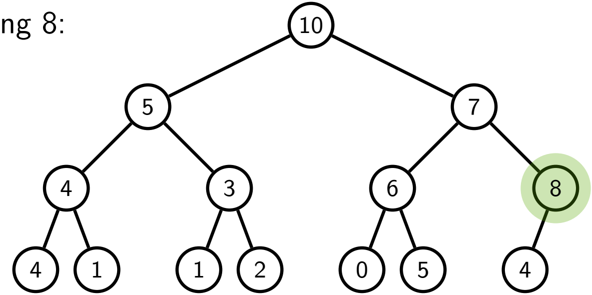


As we bubble up, when we swap the value of a node *i* with that of its parent, we don't have to compare *i* with its sibling, because if the value of *i* is greater than that of its parent, then it must be greater than that of its sibling because of the Binary Heap Tree property.

# Insertion

**Idea:** Insert the value at the end of the last level and then keep bubbling it up as long as it is larger than its parent.

Inserting 8:

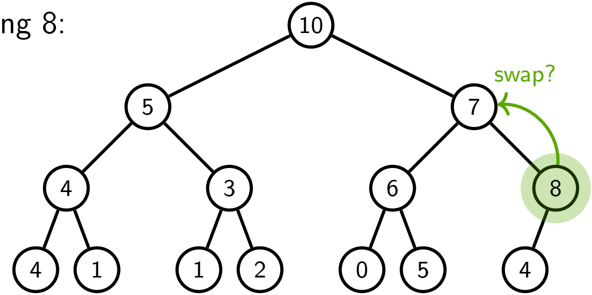


As we bubble up, when we swap the value of a node *i* with that of its parent, we don't have to compare *i* with its sibling, because if the value of *i* is greater than that of its parent, then it must be greater than that of its sibling because of the Binary Heap Tree property.

# Insertion

**Idea:** Insert the value at the end of the last level and then keep bubbling it up as long as it is larger than its parent.

Inserting 8:

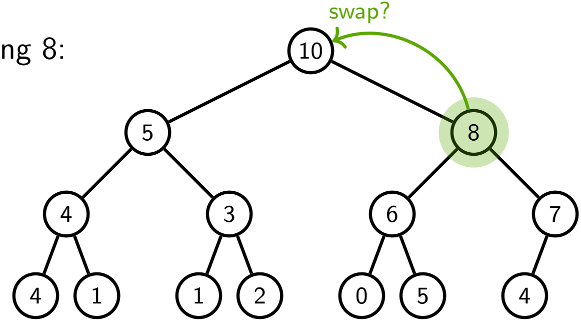


As we bubble up, when we swap the value of a node *i* with that of its parent, we don't have to compare *i* with its sibling, because if the value of *i* is greater than that of its parent, then it must be greater than that of its sibling because of the Binary Heap Tree property.

# Insertion

**Idea:** Insert the value at the end of the last level and then keep bubbling it up as long as it is larger than its parent.

Inserting 8:

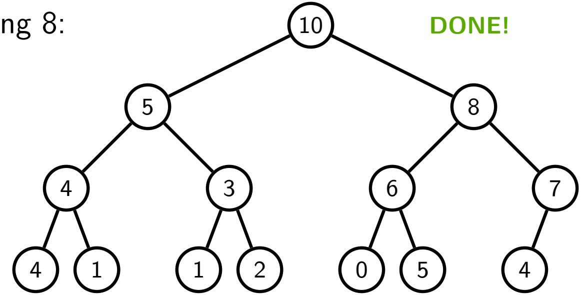


As we bubble up, when we swap the value of a node *i* with that of its parent, we don't have to compare *i* with its sibling, because if the value of *i* is greater than that of its parent, then it must be greater than that of its sibling because of the Binary Heap Tree property.

## Insertion

**Idea:** Insert the value at the end of the last level and then keep bubbling it up as long as it is larger than its parent.

Inserting 8:



DONE!

As we bubble up, when we swap the value of a node *i* with that of its parent, we don't have to compare *i* with its sibling, because if the value of *i* is greater than that of its parent, then it must be greater than that of its sibling because of the Binary Heap Tree property.

# Insert (pseudocode)

```
public void insert(int p) {
  if (n == MAXSIZE)
    throw HeapFullException;
  n = n + 1;
  heap[n] = p;   // insert the new value as the last
                 // node of the last level
  bubbleUp(n);   // and bubble it up
}
```

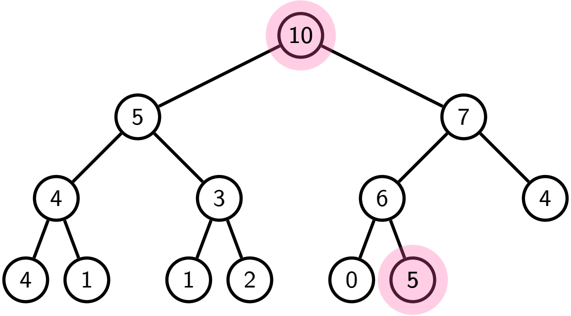
```
private void bubbleUp(int i) {
  if (i == 1) return;  // i is the root

  if (heap[i] > heap[parent(i)]) {
    swap heap[i] and heap[parent(i)];
    bubbleUp(parent(i));
  }
}
```

# Binary Heap Tree Deletion

# Delete Root

**Idea:**

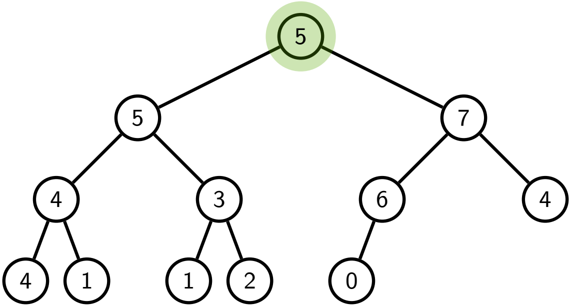
1. Remove the last node and use it to replace the root
2. Then bubble down: keep swapping it with the higher priority child as long as any of its children has a higher priority.



# Delete Root

**Idea:**

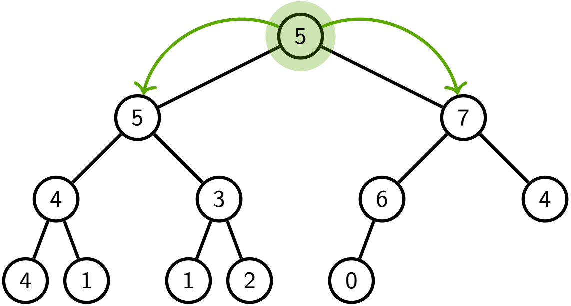
1. Remove the last node and use it to replace the root
2. Then bubble down: keep swapping it with the higher priority child as long as any of its children has a higher priority.



# Delete Root

**Idea:**

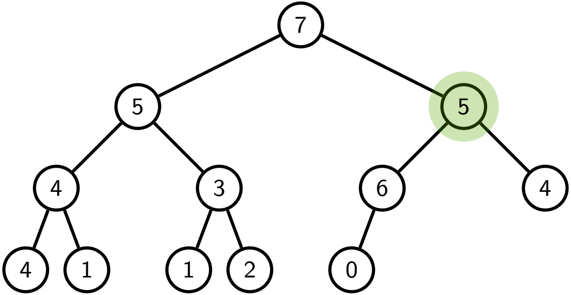
1. Remove the last node and use it to replace the root
2. Then bubble down: keep swapping it with the higher priority child as long as any of its children has a higher priority.



# Delete Root

**Idea:**

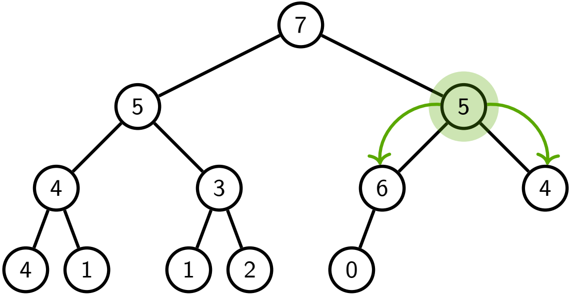
1. Remove the last node and use it to replace the root
2. Then bubble down: keep swapping it with the higher priority child as long as any of its children has a higher priority.



# Delete Root

**Idea:**

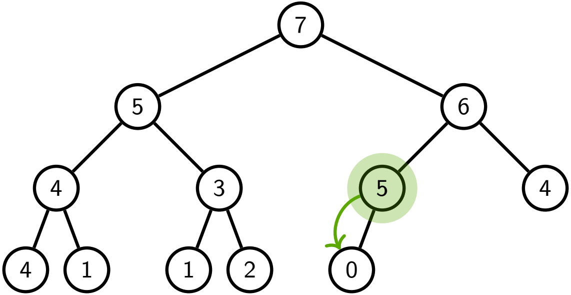
1. Remove the last node and use it to replace the root
2. Then bubble down: keep swapping it with the higher priority child as long as any of its children has a higher priority.



# Delete Root

**Idea:**

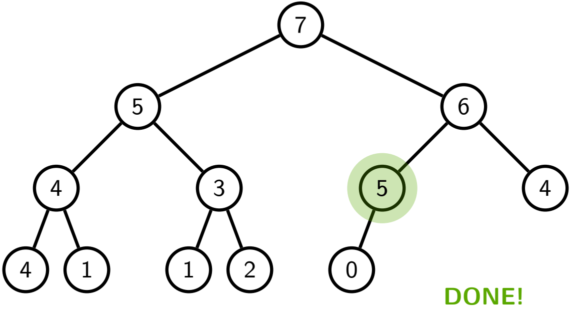
1. Remove the last node and use it to replace the root
2. Then bubble down: keep swapping it with the higher priority child as long as any of its children has a higher priority.



## Delete Root

**Idea:**

1. Remove the last node and use it to replace the root
2. Then bubble down: keep swapping it with the higher priority child as long as any of its children has a higher priority.



# Delete Root (pseudocode)

```
public void deleteRoot() {
  if (n < 1)
    throw EmptyHeapException;

  heap[1] = heap[n];
  n = n-1;
  bubbleDown(1);
}
```

# bubbleDown

The `bubbleDown` method needs to deal with 5 cases, depending on the current node that is being bubbled down:

1. it is a leaf node: nothing to do, `bubbledown` is complete
2. it has a left child, but no right child: if left child is greater than it, swap left child with current and `bubbleDown` is complete (no right child and this is a complete tree means the left child cannot have any children)
3. it has two children, the left child is greater of the two and is greater than the current node: swap left child and current and continue recursively bubbling down the left child
4. as the previous case but the right child is the greater: continue as above but on the right child
5. it has two children and neither is greater than the current node: nothing to do, `bubbledown` is complete

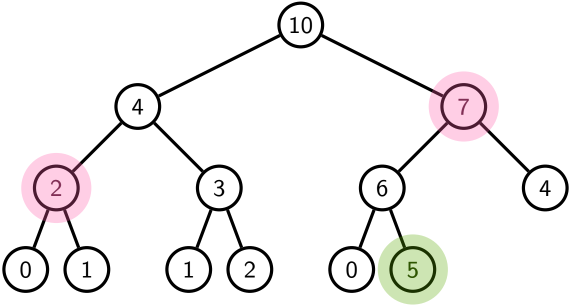
# bubbleDown (pseudocode)

```
private void bubbleDown(int i) {
    if ( left(i) > n )                    // no children
        return;
    else if ( right(i) > n )              // only left child
    {   if ( heap[i] < heap[left(i)] )
            swap heap[i] and heap[left(i)]
    }
    // else two children
    else if ( heap[left(i)] > heap[right(i)])
    {   // left child has higher priority than right
        if (heap[i] < heap[left(i)] )
        {    swap heap[i] and heap[left(i)]
             bubbleDown(left(i))
        }
    }
    else // right child has higher priority than left
        if ( heap[i] < heap[right(i)] )
        {   swap heap[i] and heap[right(i)]
            bubbleDown(right(i))
        }
    }
}
```

# Delete

Deleting an arbitrary node means the node might be anywhere in the tree.

1. Remove the last node and use it to replace the node to be deleted
2. This node may be smaller than its children or larger than its parent, so bubble up or bubble down as necessary



# Delete (pseudocode)

```
public void delete(int i) {
  if (n < 1)
    throw EmptyHeapException;
  if (i < 1 or i > n)
    throw IndexOutOfBoundsException;

  heap[i] = heap[n];
  n = n-1;
  bubbleUp(i)
  bubbleDown(i);
}
```

Note that only one of `bubbleUp` or `bubbleDown` is necessary, but since the one which turns out to be unnecessary will not do anything, it is safe to call both, one after the other, rather than test to check which one should be invoked.

# Update

Update means modifying the priority of a element in the tree. This works like `delete` except that we set the priority of the target node from a parameter, rather than from the last element in the tree, and we don't reduce the size of the tree.

```
public void update(int i, int priority) {
  if (n < 1)
    throw EmptyHeapException;
  if (i < 1 or i > n)
    throw IndexOutOfBoundsException;

  heap[i] = priority;
  bubbleUp(i)
  bubbleDown(i);
}
```

# Heapify and Merge

## Building a Binary Heap Tree

Inserting a set of $n$ items into an empty Binary Heap Tree is $n$ inserts of complexity $O(\log n)$ giving a total complexity of $O(n \log n)$.

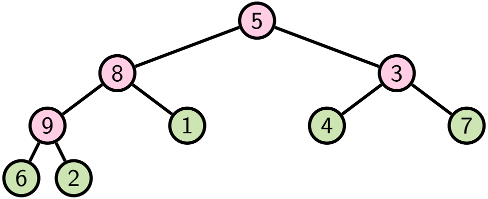
There is a more efficient way: If we have the items in an array in random order (starting at index position 1), then we already have them in Complete Binary Tree form, but not in Binary Heap Tree form. At this point, all the leaf nodes satisfy the heap tree properties, but the internal nodes do not.

We can therefore iterate over the internal nodes, starting with the last internal node and working up to the first, calling `bubbleDown` on each in turn. Each time we do, we ensure that the subtree based on that node becomes a valid Binary Heap Tree, so that in the end, the whole tree is a valid Binary Heap Tree.

# Building a Binary Heap Tree

The last node is node $n$. The parent of the last node is node $n/2$ and that must be the last internal node in the tree, because node $n/2+1$ would have left child $2*(n/2+1) = n+2$, which doesn't exist, so node $n/2+1$ must not be an internal node.

```
public void heapify() {
    for( int i = n/2 ; i > 0 ; i— )
        bubbleDown(i)
}
```



|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| heap: |  | 5 | 8 | 3 | 9 | 1 | 4 | 7 | 6 | 2 |

## Complexity of Heapify

Where $h$ is the height of the heap tree, the root node needs to swap with $h$-many nodes, the nodes on level 1 swap with $(h-1)$-many nodes and so on. In the **worst** case, the total number of swaps is

$$\begin{aligned} C(h) &= h + 2(h-1) + 4(h-2) + \ldots + 2^{h-1}(h-(h-1)) \\ &= \sum_{i=0}^{h} 2^i(h-i) = \frac{2^h}{2^h}\sum_{i=0}^{h} 2^i(h-i) = 2^h \sum_{i=0}^{h} \frac{h-i}{2^{h-i}} \\ &= 2^h \sum_{j=0}^{h} \frac{j}{2^j} \leq 2^h \sum_{j=0}^{\infty} \frac{j}{2^j} \end{aligned}$$

and since $\sum_{j=0}^{\infty} \frac{j}{2^j} = 2$ we obtain that $C(h) = 2^h \times 2 = 2^{h+1}$.

Hence the complexity of heapify is $O(2^{h+1}) = O(2^h) = O(n)$

# Merging Binary Heap Trees

Three major approaches to merge two BHTs of similar size $n$:

1. Insert each item of one tree into the other
   - $n$ inserts, hence $O(n \log n)$
2. Remove the last elements of the bigger tree and insert them into the smaller until the tree made from a dummy root node and the smaller and bigger trees as its left and right child respectively, is complete. Then use the standard delete method to delete the dummy root node[1].
   - On average, the smaller tree is half the size of the larger, about a quarter of the leaf nodes of the large must be inserted into the smaller to make them approx equal in size, so $O(n)$ inserts, hence complexity is also $O(n \log n)$. However, this will be about $\frac{1}{4}$ of the number of operations of the previous method so faster.
3. Concatenate the array forms and call `heapify`
   - $O(n)$

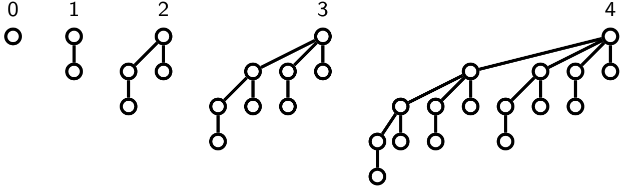
[1] Note: more complicated with array forms than with pointer based trees

# Binomial and Fibonacci Heaps

# Binomial Trees

**Definition**
A *binomial tree* is defined recursively as follows:

- A binomial tree of order 0 is a single node.
- A binomial tree of order $k$ has a root node with children that are roots of binomial trees of orders $k-1, k-2, \ldots, 2, 1, 0$



Note:

- A Binomial Tree of order $k$ has exactly $2^k$ nodes
- A Binomial Tree of order $k$ can be constructed from 2 Binomial Trees of order $k-1$ by attaching one of them as a new leftmost child of the root of the other: this is the basis of the efficient *merge* operation for Binomial Heaps.

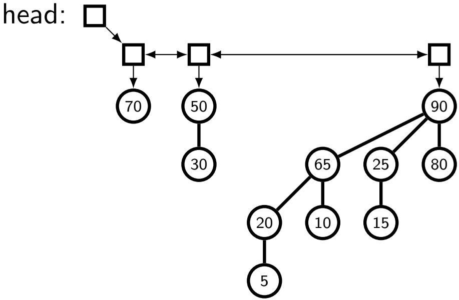
## Binomial Heaps

A *Binomial Heap* is a list of *Binomial Trees* with the properties:

- There can be only zero or one *Binomial Trees* of each order
- Each *Binomial Tree* satisfies the priority ordering property: each node has priority less than or equal to its parent.

A typical implementation would use a doubly linked list of pointers to the root of each component Binomial Tree kept in order of Binomial Tree orders.

# Binomial Heap Example, Find Max



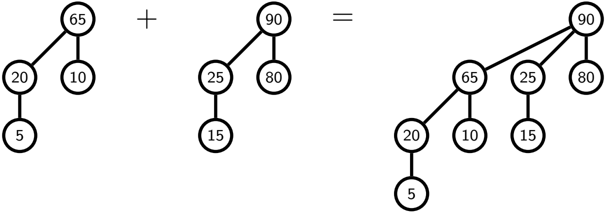
Note that there is no ordering between the keys in the **different** component Binomial Trees

Finding the node with highest priority is a linear search through the trees comparing the root values. Since, in the worst case, the number of nodes double in each consecutive tree, there will be $\log n$ trees required to store $n$ values, so this is $O(\log n)$

# Binomial Heap Merge

The key operation is merge, which merges two binomial heaps into one. Inserting a new value into a binomial tree works by merging a simple one node heap with the new value into the existing heap.

Note that if we have two binomial trees of the same order $k$, we can merge them into a single binomial tree of order $k+1$ by adding one of the trees as the leftmost child of the root of the other:

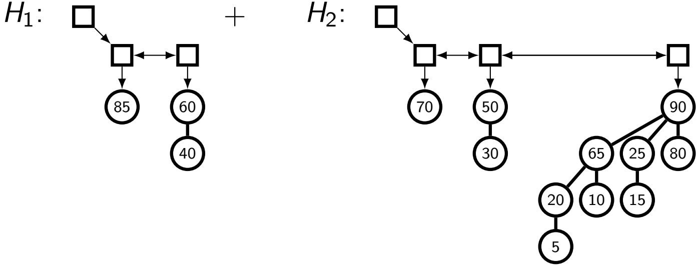


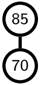
Binomial Heap Merge works in a way that is analogous to elementary addition of integers: Think of each Binomial Tree as a single binary digit in the addition.

# Binomial Heap Merge

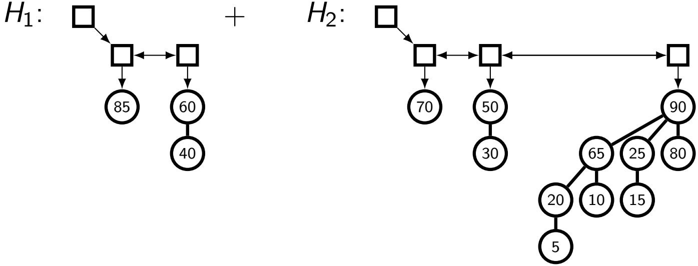
- "Adding" two trees of the same order $k$ produces a single tree of order $k+1$, and is treated as a "*carry out*"
- Iterate through the tree orders $0, 1, 2, \ldots$. For each order, set the resulting tree and carry out to be the merge of any carry in and the trees of that order in the two heaps
- For any particular order $k$, there may be between 0 and 3 input trees to be merged, with the following effect on the result heap:
  0: no output tree of order $k$ and there is no carry out.
  1: the output tree of order $k$ is the input tree and there is no carry out.
  2: no output tree of order $k$ and the carry out (of order $k+1$) is the merge of the two input trees
  3: the output tree of order $k$ is one of the input trees, and the carry out (of order $k+1$) is the merge of the other two input trees

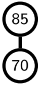
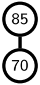
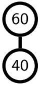
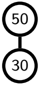
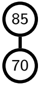
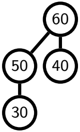
# Binomial Heap Merge Example



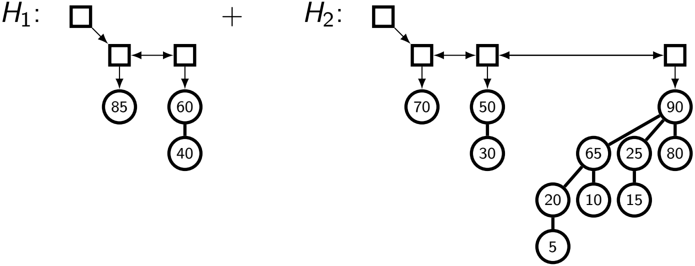
| Order | Carry-in | $H_1$ | $H_2$ | Out | Carry-out |
|---|---|---|---|---|---|
| 0 |  | | |  |  |

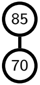
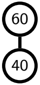
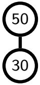
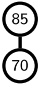
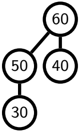
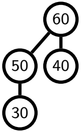
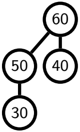
# Binomial Heap Merge Example



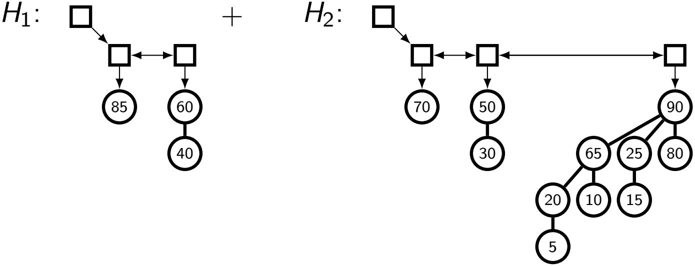
| Order | Carry-in | $H_1$ | $H_2$ | Out | Carry-out |
|---|---|---|---|---|---|
| 0 | | | | |  |
| 1 |  |  |  |  |  |

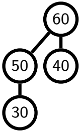
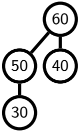
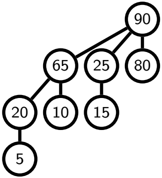
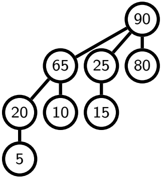
# Binomial Heap Merge Example



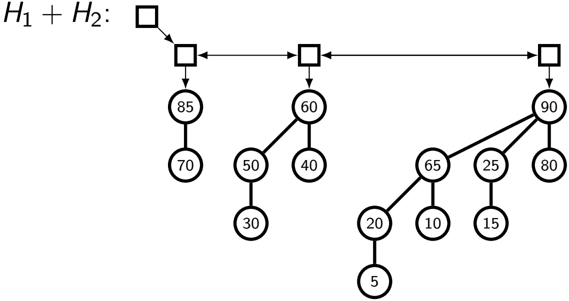
| Order | Carry-in | $H_1$ | $H_2$ | Out | Carry-out |
|---|---|---|---|---|---|
| 1 |  |  |  |  |  |
| 2 |  | | |  | |

# Binomial Heap Merge Example



| Order | Carry-in | $H_1$ | $H_2$ | Out | Carry-out |
|---|---|---|---|---|---|
| 2 |  | | |  | |
| 3 | | |  |  | |

# Binomial Heap Merge Example: Final Result



# Complexity

Merging two Binomial Trees is $O(1)$

Merging two Binomial Heaps requires, in general, the merging of $O(\log n)$ Binomial Trees, hence $O(\log n)$

Inserting merges two Binomial Heaps: one with a single Binomial Tree of order 0. There is a probability of

- $\frac{1}{2}$ that the other heap has a Binomial tree of order 0 and so requires merging of order 0 trees and a carry-out
- $\left(\frac{1}{2}\right)^2$ that the other heap has a Binomial tree of order 1 and requires a merge
- $\left(\frac{1}{2}\right)^3$ etc.

Full average (amortised) cost of insertion is

$$O(1) \times \sum_{i=1}^{\infty} \left(\frac{1}{2}\right)^i = O(1)$$

# Binomial Heap: Other operations

There is no fast way to build a Binomial Heap from a collection of $n$ key values, so simply insert $n$ times: $O(n)$

To change the priority of a node in a Binomial Heap, we can use a bubble up/down process similar to that of Binary Heaps: $O(\log n)$

Deleting the highest priority node requires finding it: a linear search through the roots of the Binomial trees: $O(\log n)$, and removing the root node of that tree and merging the subtrees back into the binomial heap: $O(\log n)$. Total cost:

$$O(\log n) + O(\log n) = O(\log n)$$

Deleting non-root nodes can be done by setting the node's priority to $\infty$, bubbling it up to the root and then deleting it as in the previous case: $O(\log n)$

# Fibonacci Heaps

*Fibonacci Heaps* are similar to Binomial Heaps in that they are also a collection of trees, but with different constraints on their structure. They are considerably more complex than Binomial Heaps, and make use of lazy modifications to keep themselves organised.

Their advantage over Binomial Heaps are that they achieve complexity of $O(1)$ for merge, and updating the priority of a node has amortised complexity of $O(1)$

# Priority Queue Complexity

# Comparison of priority queues implementations

| Operation | Binary Heaps | Binomial Heaps | Fibonacci Heaps |
|---|---|---|---|
| Insert | $O(\log n)$ | $O(1)^\star$ | $O(1)$ |
| Delete | $O(\log n)$ | $O(\log n)$ | $O(\log n)^\star$ |
| Update | $O(\log n)$ | $O(\log n)$ | $O(1)^\star$ |
| Merge | $O(n)$ | $O(\log n)$ | $O(1)$ |
| Heapify | $O(n)$ | $O(n)$ | $O(n)$ |

Where $\star$ means that it is the amortized complexity.

# Sorting

# Sorting

Given a set of records (or objects), we can sort them by many different criteria. For example, a set of student/mark records that contain marks for different students in different modules could be sorted:

- in decreasing order by mark
- in alphabetic order of their surname first, then by their firstname, if there are students with the same surname
- in increasing order of module name, then by decreasing order of mark

Sort algorithms mostly work on the basis of a *comparison* function that is supplied to them that defines the order required between any two objects or records.

In some special cases, the nature of the data means that we can sort without using a comparison function.

# Comparing objects in Java

Java provides two interfaces to implement comparison functions:

`Comparable`: A `Comparable` object can compare itself with another object using its `compareTo(...)` method. There can only be one such method for any class, so this should implement the default ordering of objects of this type. `x.compareTo(y)` should return a negative int, 0, or a positive int if `x` is less than, equal to, or greater than `y` respectively.

`Comparator`: A `Comparator` object can be used to compare two objects of some type using the `compare(...)` method. This does not work on the current object but rather both objects to be compared are passed as arguments. You can have many different comparison functions implemented this way. `compare(x, y)` should return a negative int, 0, or a positive int if `x` is less than, equal to, or greater than `y` respectively.

# Comparison-based Sorting Strategies

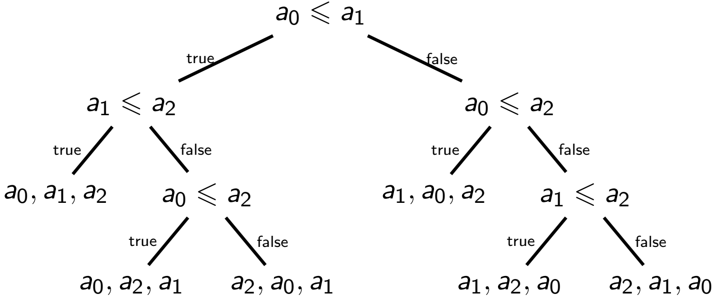
There are a number of basic strategies for comparison-based sorting, and different sorting algorithms based on each strategy:

- **Enumeration:** For each item, count the number of items less than it, say $N$, then put the current item at position $N + 1$.
- **Exchange:** If two items are found to be out of order, exchange them. Repeat until all items are in order.
- **Selection:** Find the smallest item, put it in position 1, find the smallest remaining item, put it in position 2, ...
- **Insertion:** Take the items one at a time and insert into an initially empty data structure such that the data structure continues to be sorted at each stage.
- **Divide and conquer:** Recursively split the problem into smaller sub-problems till you just have single items that are trivial to sort. Then put the sorted 'parts' back together in a way that preserves the sorting.

## Minimum number of Comparisons

For comparison-based sorting, the minimum number of comparisons necessary to sort $n$ items gives us a lower bound on the complexity of any comparison based sorting algorithm.

Consider an array $a$ of 3 elements: $a_0, a_1, a_2$. We can make a decision tree to figure out which order the items should be in (note: no comparisons are repeated on any path from the root to a leaf):



# Minimum number of Comparisons

- This decision tree is a binary tree where there is one leaf for every possible ordering of the items
- The **average** number of comparisons that are necessary to sort the items will be the average path length from the root to a leaf of the decision tree.
- The **worst case** number of comparisons that are necessary to sort the items will be the height of the decision tree.
- Given $n$ items, there are $n$ ways to choose the first item, $n-1$ ways to choose the second, $n-2$ ways to choose the third, etc. so there are $n(n-1)(n-2)\dots 3\cdot 2\cdot 1 = n!$ different possible orderings of $n$ items
- Thus the minimum number of comparisons necessary to sort $n$ items is the height of a binary tree with $n!$ leaves

# Minimum number of Comparisons

A binary tree of height $h$ has the most number of leaves if all the leaves are on the bottom-most level, thus it has at most $2^h$ leaves.

Hence we need to find $h$ such that

$$\begin{aligned} 2^h &\geqslant n! \\ \implies \quad \log_2 2^h &\geqslant \log_2 n! \\ \implies \quad h &\geqslant \log_2 n! \end{aligned}$$

But $\log_2 n! = \Theta(n \log n)$, thus we need at least $n \log n$ comparisons to complete a comparison based sort in general.[1]

[1] There are many ways to prove this but the easiest involves showing that $(\frac{n}{2})^{\frac{n}{2}} \leqslant n! \leqslant n^n$ and taking the log of all terms

# Stability in Sorting

A *stable* sorting algorithm does not change the order of items in the input if they have the same sort key.

Thus if we have a collection of student records which is already in order by the students' first names, and we use a stable sorting algorithm to sort it by students' surnames, then all students with the same surname will still be sorted by their firstnames.

Using stable sorting algorithms in this way, we can "*pipeline*" sorting steps to construct a particular order in stages.

In particular, a stable sorting algorithm is often faster when applied to an already sorted, or nearly sorted list of items. If your input is usually nearly sorted, then you may be able to get higher performance by using a stable sorting algorithms. However, many stable sorting algorithms have higher complexity than unstable ones, so the compexities involved should be carefully checked.

# Bubble Sort (Exchange)

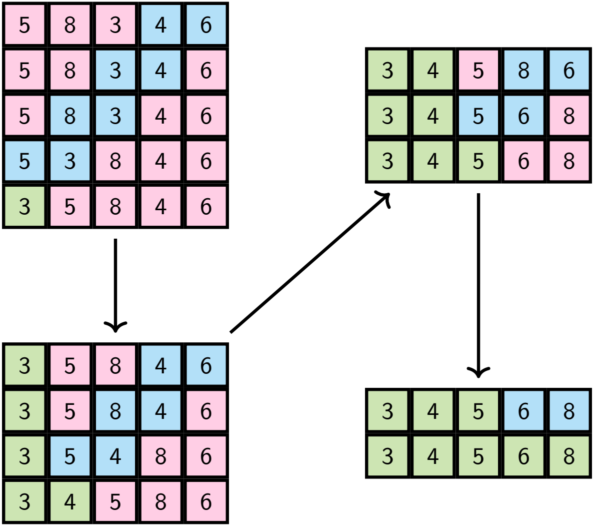
# Bubble Sort

Bubble sort does multiple passes over an array of items, swapping neighbouring items that are out of order as it goes.

Each pass guarantees that at least one extra element ends up in its correct ordered location at the start of the array, so consecutive passes shorten to work only on the unsorted part of the array until the last pass only needs to sort the remaining two elements at the end of the array.

```
bubblesort(a, n) {
  for ( i = 1 ; i < n ; i++ )
    for ( j = n-1 ; j >= i ; j-- )
      if ( a[j] < a[j-1] )
        swap a[j] and a[j-1]
}
```

# Example of a Bubble Sort run



| | | | | |
|---|---|---|---|---|
| 5 | 8 | 3 | 4 | 6 |
| 5 | 8 | 3 | 4 | 6 |
| 5 | 8 | 3 | 4 | 6 |
| 5 | 3 | 8 | 4 | 6 |
| 3 | 5 | 8 | 4 | 6 |

| | | | | |
|---|---|---|---|---|
| 3 | 5 | 8 | 4 | 6 |
| 3 | 5 | 8 | 4 | 6 |
| 3 | 5 | 4 | 8 | 6 |
| 3 | 4 | 5 | 8 | 6 |

| | | | | |
|---|---|---|---|---|
| 3 | 4 | 5 | 8 | 6 |
| 3 | 4 | 5 | 6 | 8 |
| 3 | 4 | 5 | 6 | 8 |

| | | | | |
|---|---|---|---|---|
| 3 | 4 | 5 | 6 | 8 |
| 3 | 4 | 5 | 6 | 8 |

## Bubble Sort Complexity

The outer loop is iterated $n - 1$ times.

The inner loop is iterated $n - i$ times, with each iteration executing a single comparison. So the total number of comparisons is:

$$\begin{aligned}\sum_{i=1}^{n-1}\sum_{j=i}^{n-1} 1 &= \sum_{i=1}^{n-1}(n - i)\\ &= (n-1) + (n-2) + \cdots + 1\\ &= \frac{n(n-1)}{2}.\end{aligned}$$

Thus best, average and worst case complexities are all $O(n^2)$

# Bubble Sort Stability

Consider what happens when two elements with the same value are in the array to be sorted.

Since only neighbouring pairs of values can be swapped, and the swap is only carried out if one is strictly less than the other, no pair of the same values will ever be swapped. Hence bubble sort can not change the relative order of two elements with the same value.

Hence bubble sort is *stable*.

# Insertion Sort (Insertion)

# Insertion Sort

Insertion sort works by taking each element of the input array and *inserting* it into its correct position relative to all the elements that have been inserted so far.

It does this by partitioning the array into a sorted part at the front and an unsorted part at the end.

Initially the sorted part is just the first cell of the array and the unsorted part is the rest.

In each pass it takes the first element of the unsorted part and inserts it into its correct position in the sorted part, simultaneously growing the sorted part and shrinking the unsorted part by one cell.

# Example of an Insertion Sort run

1. 5 | $\underline{12}$ , 6, 3, 11, 8, 4
2. 5, 12 | $\underline{6}$ , 3, 11, 8, 4
3. 5, 6, 12, | $\underline{3}$ , 11, 8, 4
4. 3, 5, 6, 12 | $\underline{11}$ , 8, 4
5. 3, 5, 6, 11, 12 | $\underline{8}$ , 4
6. 3, 5, 6, 8, 11, 12 | $\underline{4}$
7. 3, 4, 5, 6, 8, 11, 12 |

# Insertion Sort Pseudo-Code

```
insertionsort(a, n) {
  for ( i = 1 ; i < n ; i++ ) {
      j = i
      t = a[j]
      while ( j > 0 && t < a[j−1] ) {
          a[j] = a[j−1]
          j—
      }
      a[j] = t
  }
}
```

# Insertion Sort Complexity

The outer loop is iterated $n-1$ times

In the worst case, the inner loop is iterated 1 time for the first outer loop iteration, 2 times for the 2nd outer iteration, etc. Thus, in the worst case, the number of comparisons is:

$$\begin{aligned}\sum_{i=1}^{n-1}\sum_{j=1}^{i} 1 &= \sum_{i=1}^{n-1} i \\ &= 1 + 2 + \cdots + (n-1) \\ &= \frac{n(n-1)}{2}\end{aligned}$$

In the average case, it is half that (because on average the correct position for the insertion in each inner loop will be in the middle of the sorted part), i.e. $\frac{n(n-1)}{4}$

Hence average and worst case complexity is $O(n^2)$

## Insertion Sort Stability

Consider what happens when two elements with the same value are in the array to be sorted.

Since the value inserted in each outer loop is the first value in the unsorted part of the array, the first occurrence of two copies of the same value will be taken for insertion before the second.

Further, in the inner loop, we walk down from the end of the sorted part of the array until we find the first location that is not strictly greater than the value to be inserted before inserting there. That means that we will not insert a later copy of a value before an earlier copy that has already been inserted.

Hence insertion sort is *stable*.

# Selection Sort (Selection)

# Selection Sort

Selection sort works by *selecting* the smallest remaining element of the input array and *appending* it at the end of all the elements that have been inserted so far.

Just like Insertions sort, it does this by partitioning the array into a sorted part at the front and an unsorted part at the end.

Initially the sorted part is empty and the unsorted part is the whole input array.

In each pass it finds the smallest element of the unsorted part and swaps it with the first element of the unsorted part of the array. Then it moves the split position between the sorted and the unsorted parts of the array on by one cell.

## Example of a Selection Sort run

1. | 5, 12, 6, $\underline{3}$ , 11, 8, 4
2. 3 | 12, 6, 5, 11, 8, $\underline{4}$
3. 3, 4 | 6, $\underline{5}$ , 11, 8, 12
4. 3, 4, 5 | $\underline{6}$ , 11, 8, 12
5. 3, 4, 5, 6 | 11, $\underline{8}$ , 12
6. 3, 4, 5, 6, 8 | $\underline{11}$ , 12
7. 3, 4, 5, 6, 8, 11 | $\underline{12}$
8. 3, 4, 5, 6, 8, 11, 12 |

# Selection Sort (pseudocode)

```
selectionsort(a, n){
  for ( i = 0 ; i < n-1 ; i++ ) {
      k = i
      for ( j = i+1 ; j < n ; j++ )
          if ( a[j] < a[k] )
              k = j
      swap a[i] and a[k]
  }
}
```

# Selection Sort Complexity

The outer loop is iterated $n - 1$ times

In the worst case, the inner loop is iterated $n - 1$ times for the first outer loop iteration, $n - 2$ times for the 2nd outer iteration, etc. Note that this is for best, worst and average cases. Thus, the total number of comparisons is:

$$\begin{aligned}\sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 &= \sum_{i=0}^{n-2} (n - 1 - i) \\ &= (n-1) + \cdots + 2 + 1 \\ &= \frac{n(n-1)}{2}.\end{aligned}$$

Hence best, average and worst case complexity is $O(n^2)$

## Selection Sort Stability

Consider what happens when two elements with the same value are in the array to be sorted.

For example, consider when the input array contains $2_1, 2_2, 1$, where the subscript indicates the order of appearance of the two copies of the value 2.

In the first pass, we find the smallest element, in this case the 1, and swap it with the first element in the array, the $2_1$. This results in $1, 2_2, 2_1$. In other words, the 2 copies of 2 have changed order.

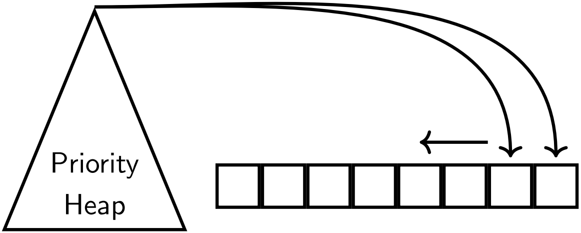
No matter how we change the condition for which element of a set of elements of the same (smallest) value we then select, we can easily produce counterexamples that show that the order of elements with the same value can change.

Hence selection sort is *unstable*.

# Heap Sort (Selection)

# Heap Sort

A priority heap structure allows efficient selecting and removal of the highest priority (i.e. largest value) from a collection of values. Heap sort uses a priority heap to sort all the values by first constructing a priority heap with the values to be sorted and then, repetitively removing the largest value from the heap and filling it in to the output array starting at the end of the array and working backwards towards the start of the array.



## Heap Sort

If we use the standard array implementation of the Binary Heap, then every time we remove the highest priority element from the heap, we reduce the size of the heap by one and we are therefore using less of the array to hold the heap values. We can therefore use the *SAME* array to put the sorted output values into, thus avoiding the need to use a separate extra array. Of course, the final sorted elements will be in index locations 1 to $n$ of the array, instead of 0 to $n - 1$:

```
heapSort(array a, int n) {
    heapify(a,n)
    for( j = n ; j > 1 ; j— ) {
        swap a[1] and a[j]
        bubbleDown(1,a,j−1)
    }
}
```

# Heap Sort Complexity and Stability

Heapify is $O(n)$. Then we have to do $n$ bubble down operations, each of complexity $O(\log n)$ which gives a cost of $O(n \log n)$ in total.

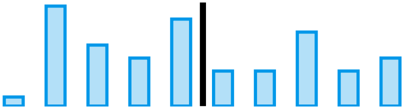
Because of the reordering in the bubble down operations, this sort is *unstable*

# Merge Sort (Divide & Conquer)

# Merge Sort

**Idea:**

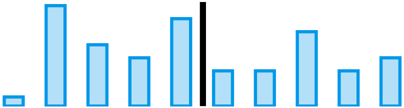
1. Split the array into two halves:



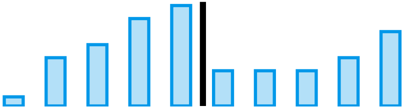
# Merge Sort

**Idea:**

1. Split the array into two halves:



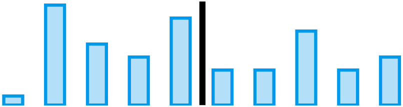
2. Sort each of them recursively:



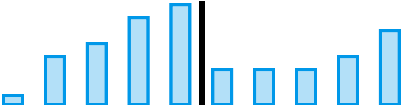
# Merge Sort

**Idea:**

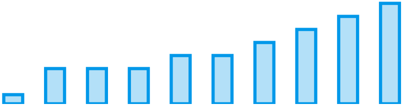
1. Split the array into two halves:



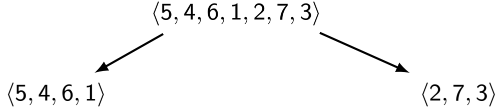
2. Sort each of them recursively:



3. Merge the sorted parts:



# Example: Merge Sort run

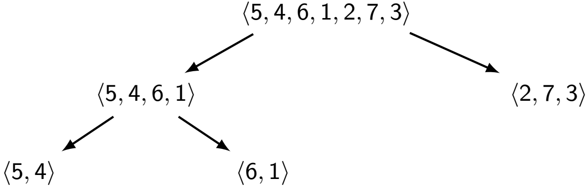


$\langle 5, 4, 6, 1, 2, 7, 3 \rangle$

$\langle 5, 4, 6, 1 \rangle$

$\langle 2, 7, 3 \rangle$

# Example: Merge Sort run



$\langle 5, 4, 6, 1, 2, 7, 3 \rangle$

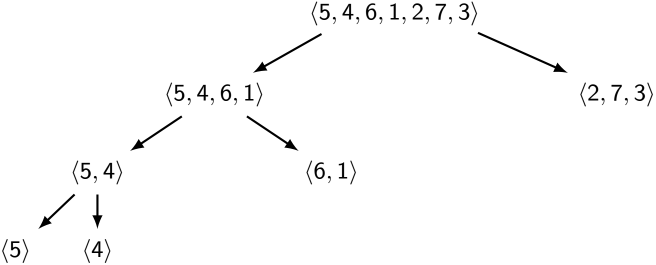
$\langle 5, 4, 6, 1 \rangle$

$\langle 2, 7, 3 \rangle$

$\langle 5, 4 \rangle$

$\langle 6, 1 \rangle$

# Example: Merge Sort run



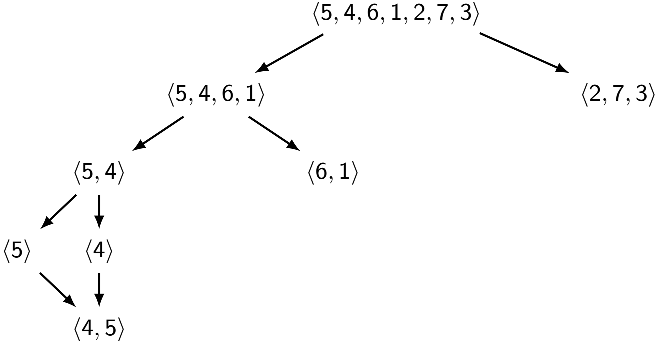
$\langle 5, 4, 6, 1, 2, 7, 3 \rangle$

$\langle 5, 4, 6, 1 \rangle$ $\langle 2, 7, 3 \rangle$

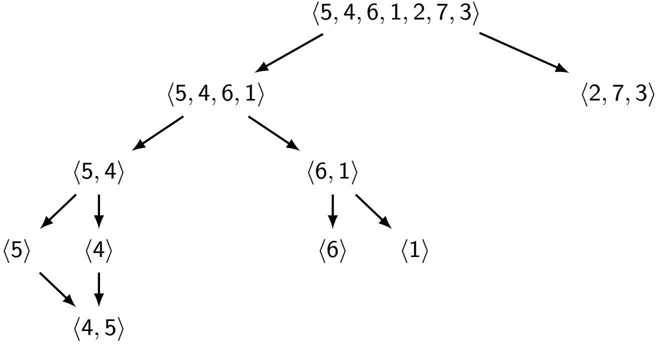
$\langle 5, 4 \rangle$ $\langle 6, 1 \rangle$

$\langle 5 \rangle$ $\langle 4 \rangle$

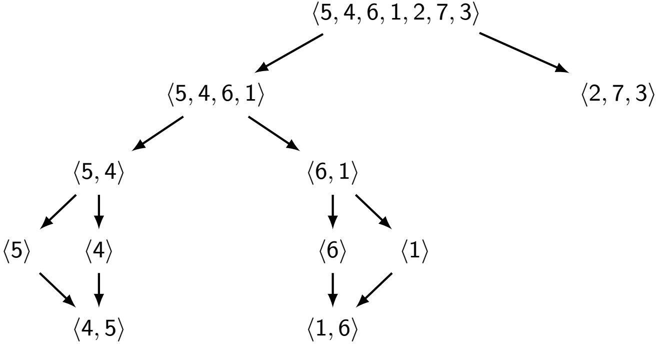
# Example: Merge Sort run



# Example: Merge Sort run



# Example: Merge Sort run



$\langle 5, 4, 6, 1, 2, 7, 3 \rangle$

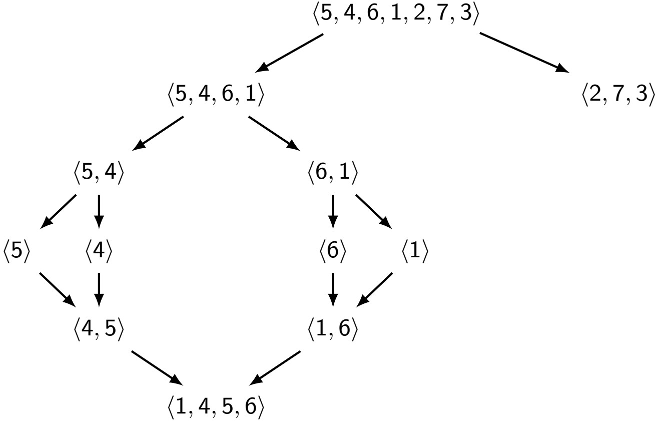
$\langle 5, 4, 6, 1 \rangle$ $\langle 2, 7, 3 \rangle$

$\langle 5, 4 \rangle$ $\langle 6, 1 \rangle$

$\langle 5 \rangle$ $\langle 4 \rangle$ $\langle 6 \rangle$ $\langle 1 \rangle$

$\langle 4, 5 \rangle$ $\langle 1, 6 \rangle$

# Example: Merge Sort run



$\langle 5, 4, 6, 1, 2, 7, 3 \rangle$

$\langle 5, 4, 6, 1 \rangle$ $\langle 2, 7, 3 \rangle$

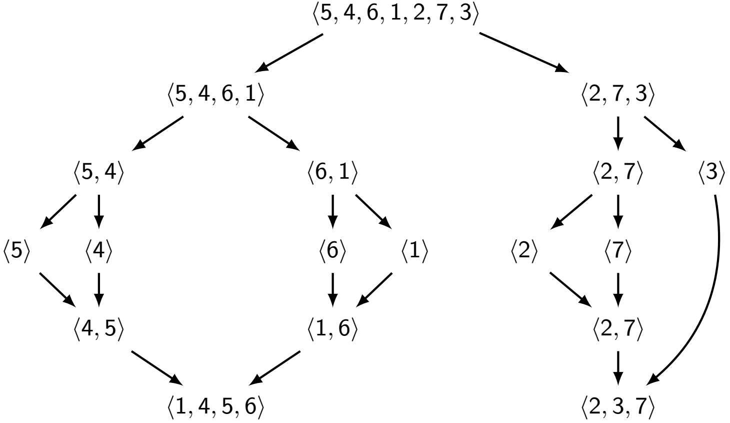
$\langle 5, 4 \rangle$ $\langle 6, 1 \rangle$

$\langle 5 \rangle$ $\langle 4 \rangle$ $\langle 6 \rangle$ $\langle 1 \rangle$

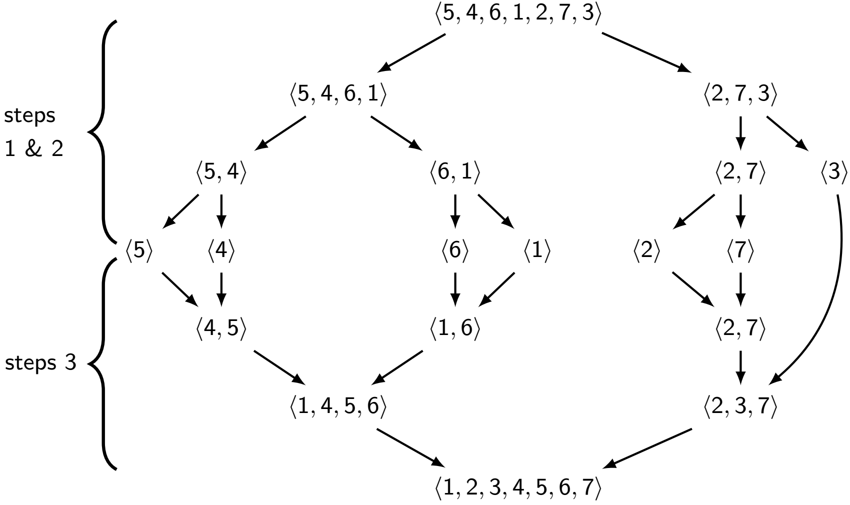
$\langle 4, 5 \rangle$ $\langle 1, 6 \rangle$

$\langle 1, 4, 5, 6 \rangle$

# Example: Merge Sort run



# Example: Merge Sort run

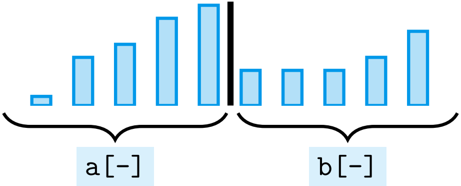


# Merging two sorted arrays `a[-]` and `b[-]` efficiently

**Idea:** In variables `i` and `j` we store the current positions in `a[-]` and `b[-]`, respectively (starting from `i=0` and `j=0`). Then:

1. Allocate a *temporary* array `tmp[-]`, for the result.
2. If `a[i] <= b[j]` then copy `a[i]` to `tmp[i+j]` and `i++`,
3. Otherwise, copy `b[j]` to `tmp[i+j]` and `j++`.

Repeat 2./3. until `i` or `j` reaches the end of `a[-]` or `b[-]`, respectively, and then copy the rest from the other array.



# Merging two sorted arrays `a[-]` and `b[-]` efficiently

Merging two sorted arrays is the most important part of merge sort and must be efficient. For example:

Take `a = [1,6,7]` and `b = [3,5]`. Set `i=0` and `j=0`, and allocate `tmp` of length `5`:

1. `a[0]` $\leqslant$ `b[0]`, so set `tmp[0] = a[0]` $(=$ `1` $)$ and `i++`.
2. `a[1]` $>$ `b[0]`, so set `tmp[1] = b[0]` $(=$ `3` $)$ and `j++`.
3. `a[1]` $>$ `b[1]`, so set `tmp[2] = b[1]` $(=$ `5` $)$ and `j++`.

At this point `i` $= 1$, `j` $= 2$ and the first three values stored in `tmp` are `[1,3,5]`.

Since `j` is at the end of `b`, we are done with `b` and we copy the remaining values from `a` into `tmp`. Then, `tmp` stores `[1,3,5,6,7]`.

# Merge Sort (pseudocode)

```
mergesort(a, n) {
    mergesort_run(a, 0, n-1)
}

void mergesort_run(a, left, right) {
    if (left < right){
      mid = (left + right) div 2

      mergesort_run(a, left, mid)
      mergesort_run(a, mid+1, right)

      merge(a, left, mid, right)
   }
}
```

The pseudocode we present here tries to avoid some of the unnecessary allocations of new arrays. Namely, when running recursive calls of merge sort, we do not allocate two new arrays for the two halves, we only compute the `left`-most and `right`-most positions of those halves, with respect to the original array `arr`.

Initially we call `mergesort_run` to sort all elements of the array, that is, we want to sort elements on positions

`0`, `1`, `2`, `3`, ..., `n-1`

In order to sort this, we run merge sort twice, first time for the positions

`0`, `1`, `2`, `3`, ..., `mid`

and the second time for positions

`(mid+1)+0`, `(mid+1)+1`, `(mid+1)+2`, ..., `n-1`.

(In further recursive calls those `left` and `right` bounds are recomputed accordingly.)

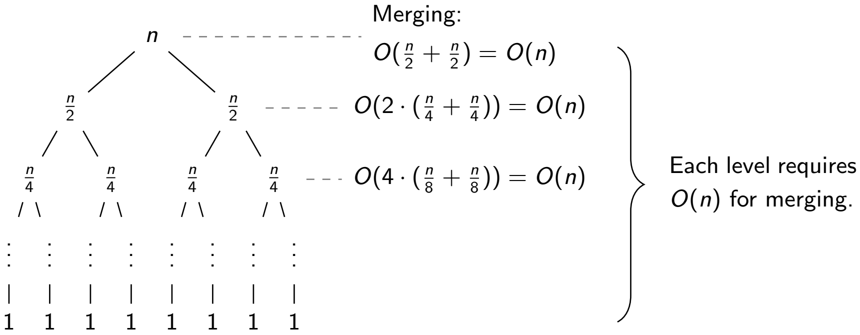
# Merging (pseudocode)

```
merge(array a, int left, int mid, int right) {
    create new array b of size right-left+1
    bcount = 0
    lcount = left
    rcount = mid+1
    while ( (lcount <= mid) and (rcount <= right) ) {
        if ( a[lcount] <= a[rcount] )
            b[bcount++] = a[lcount++]
        else
            b[bcount++] = a[rcount++]
    }
    if ( lcount > mid )
        while ( rcount <= right )
            b[bcount++] = a[rcount++]
    else
        while ( lcount <= mid )
            b[bcount++] = a[lcount++]
    for ( bcount = 0 ; bcount < right-left+1 ; bcount++ )
        a[left+bcount] = b[bcount]
}
```

# Time Complexity of Mergesort

Merging two arrays of lengths $n_1$ and $n_2$ is in $O(n_1 + n_2)$

Sizes of recursive calls:



$n$
$\frac{n}{2}$ $\frac{n}{2}$
$\frac{n}{4}$ $\frac{n}{4}$ $\frac{n}{4}$ $\frac{n}{4}$
1 1 1 1 1 1 1 1

Merging:

$O(\frac{n}{2} + \frac{n}{2}) = O(n)$

$O(2 \cdot (\frac{n}{4} + \frac{n}{4})) = O(n)$

$O(4 \cdot (\frac{n}{8} + \frac{n}{8})) = O(n)$

Each level requires $O(n)$ for merging.

If $n = 2^k$, then we have $k = \log_2 n$ levels $\Longrightarrow$ $O(n \log n)$ is the time complexity of merge sort.

(This is the Worst/Best/Average Case complexity.)

Let us analyse the running time of merge sort for an array of size $n$ and for simplicity we assume that $n = 2^k$. First, we run the algorithm recursively for two halves. Putting the running time of those two recursive calls aside, after both recursive calls finish, we merge the result in time $O(\frac{n}{2} + \frac{n}{2})$.

Okay, so what about the recursive calls? To sort $\frac{n}{2}$-many entries, we split them in half and sort both $\frac{n}{4}$-big parts independently. Again, after we finish, we merge in time $O(\frac{n}{4} + \frac{n}{4})$. However, this time, merging of $\frac{n}{2}$-many entries happens twice and, therefore, in total it runs in $O(2 \times (\frac{n}{4} + \frac{n}{4})) = O(2 \times \frac{n}{2}) = O(n)$.

Similarly, we have 4 subproblems of size $\frac{n}{4}$, each of them is merging their subproblems in time $O(\frac{n}{8} + \frac{n}{8})$. In total, all calls of `merge` for subproblems of size $\frac{n}{4}$ take $O(4 \times (\frac{n}{8} + \frac{n}{8})) = O(n)$. ... We see that it always takes $O(n)$ to merge all subproblems of the same size (= those on the same level of the recursion).

Since the height of the tree is $O(\log n)$ and each level requires $O(n)$ time for all merging, the time complexity is $O(n \log n)$. Notice that this analysis does not depend on the particular data, so it is the Worst, Best and Average Case.

## Stability of Mergesort

The splitting phase of mergesort does not change the order of any items.

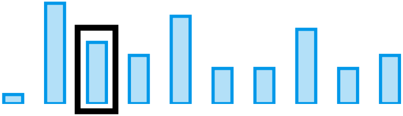
So long as merging phase merges the left with the right in that order and takes values from the leftmost sub-array before the rightmost one when values are equal (as the pseudocode above does) then different elements with the same values do not change their relative order.

Therefore mergesort is stable.

# Quick Sort (Divide & Conquer)

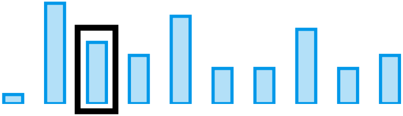
# Quick Sort

1. Select an element of the array, which we call the **pivot**.

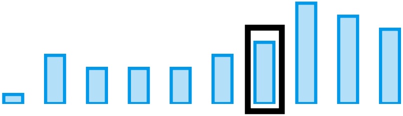


# Quick Sort

1. Select an element of the array, which we call the **pivot**.

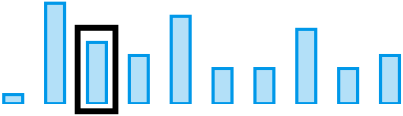


2. Partition the array so that the *"small entries"* ($\leq$ pivot) are on the left, then the pivot, then the *"large entries"* ($>$ pivot).

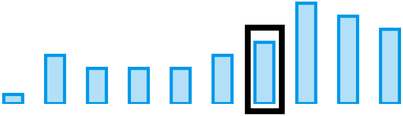


# Quick Sort

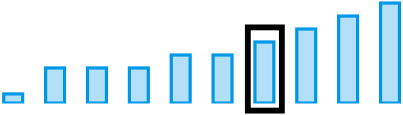
1. Select an element of the array, which we call the **pivot**.



2. Partition the array so that the *"small entries"* ($\leq$ pivot) are on the left, then the pivot, then the *"large entries"* ($>$ pivot).



3. Recursively (quick)sort the two partitions.



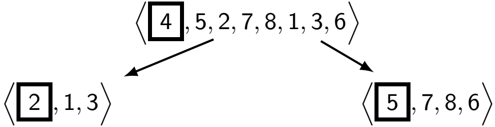
For the time being it is not important how the pivot is selected. We will see later that there are different strategies that select the pivot and they might affect the time complexity of quicksort.

**Remark:** In order for quicksort to be a *stable* sorting algorithm, it is useful to allow the *large entries* to also be $\geq$ pivot.

On the other hand, it is easier to understand how quicksort works if we require the large entries to be strictly larger than the pivot. Of course, this is only an issue if there are duplicate values in the array.

# Example: Quick Sort run

Initial pivot selection strategy: we always choose the leftmost entry.



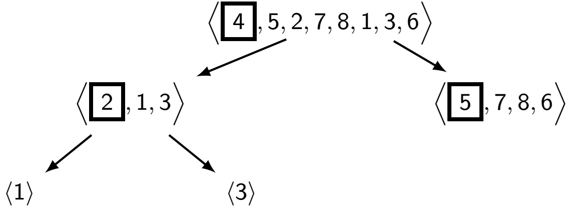
$\left\langle \boxed{4}, 5, 2, 7, 8, 1, 3, 6 \right\rangle$

$\left\langle \boxed{2}, 1, 3 \right\rangle$

$\left\langle \boxed{5}, 7, 8, 6 \right\rangle$

# Example: Quick Sort run

Initial pivot selection strategy: we always choose the leftmost entry.



$\left\langle \boxed{4}, 5, 2, 7, 8, 1, 3, 6 \right\rangle$

$\left\langle \boxed{2}, 1, 3 \right\rangle$

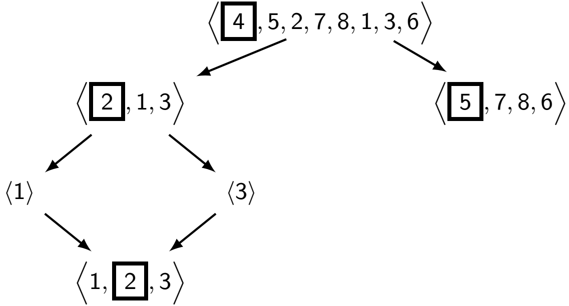
$\left\langle \boxed{5}, 7, 8, 6 \right\rangle$

$\langle 1 \rangle$

$\langle 3 \rangle$

# Example: Quick Sort run

Initial pivot selection strategy: we always choose the leftmost entry.



$\left\langle \boxed{4}, 5, 2, 7, 8, 1, 3, 6 \right\rangle$

$\left\langle \boxed{2}, 1, 3 \right\rangle$

$\left\langle \boxed{5}, 7, 8, 6 \right\rangle$

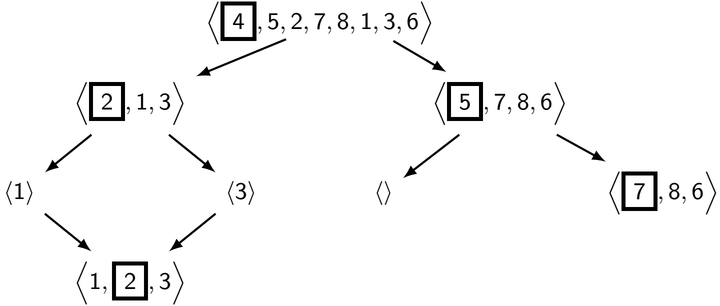
$\langle 1 \rangle$

$\langle 3 \rangle$

$\left\langle 1, \boxed{2}, 3 \right\rangle$

# Example: Quick Sort run

Initial pivot selection strategy: we always choose the leftmost entry.



$\left\langle \boxed{4}, 5, 2, 7, 8, 1, 3, 6 \right\rangle$

$\left\langle \boxed{2}, 1, 3 \right\rangle$

$\left\langle \boxed{5}, 7, 8, 6 \right\rangle$

$\langle 1 \rangle$

$\langle 3 \rangle$

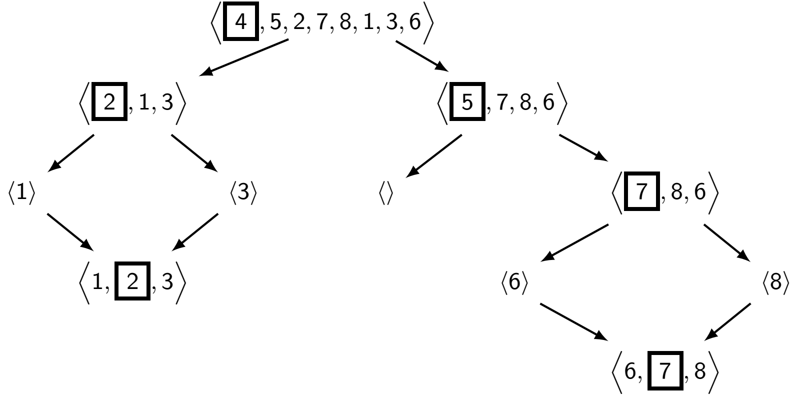
$\langle \rangle$

$\left\langle \boxed{7}, 8, 6 \right\rangle$

$\left\langle 1, \boxed{2}, 3 \right\rangle$

# Example: Quick Sort run

Initial pivot selection strategy: we always choose the leftmost entry.



$\left\langle \boxed{4}, 5, 2, 7, 8, 1, 3, 6 \right\rangle$

$\left\langle \boxed{2}, 1, 3 \right\rangle$

$\left\langle \boxed{5}, 7, 8, 6 \right\rangle$

$\langle 1 \rangle$

$\langle 3 \rangle$

$\langle \rangle$

$\left\langle \boxed{7}, 8, 6 \right\rangle$

$\left\langle 1, \boxed{2}, 3 \right\rangle$

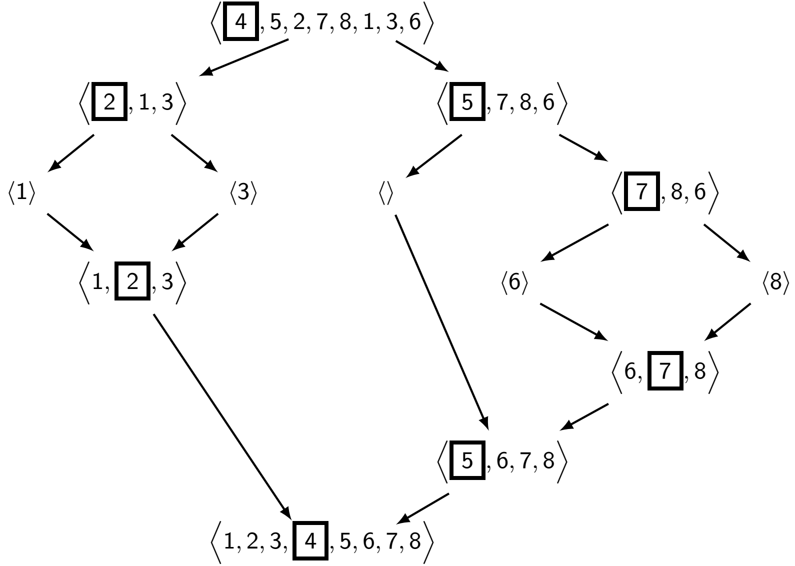
$\langle 6 \rangle$

$\langle 8 \rangle$

$\left\langle 6, \boxed{7}, 8 \right\rangle$

# Example: Quick Sort run

Initial pivot selection strategy: we always choose the leftmost entry.



$\left\langle \boxed{4}, 5, 2, 7, 8, 1, 3, 6 \right\rangle$

$\left\langle \boxed{2}, 1, 3 \right\rangle$

$\left\langle \boxed{5}, 7, 8, 6 \right\rangle$

$\langle 1 \rangle$

$\langle 3 \rangle$

$\langle \rangle$

$\left\langle \boxed{7}, 8, 6 \right\rangle$

$\left\langle 1, \boxed{2}, 3 \right\rangle$

$\langle 6 \rangle$

$\langle 8 \rangle$

$\left\langle 6, \boxed{7}, 8 \right\rangle$

$\left\langle \boxed{5}, 6, 7, 8 \right\rangle$

$\left\langle 1, 2, 3, \boxed{4}, 5, 6, 7, 8 \right\rangle$

# Quick Sort (pseudocode)

```
void quicksort(a, n){
    quicksort_run(a, 0, n-1)
}

quicksort_run(a, left, right) {
   if ( left < right ) {
      pivotindex = partition(a, left, right)
      quicksort_run(a, left, pivotindex-1)
      quicksort_run(a, pivotindex+1, right)
   }
}
```

Where `partition` rearranges the array so that

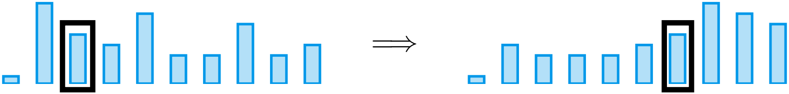
- the small entries are stored on positions `left, left+1, left+2, ..., pivot_index-1`,
- pivot is stored on position `pivot_index` and
- the large entries are stored on `pivot_index+1, pivot_index+2, ..., right`.

# Partitioning array `a`

**Idea:**

1. Choose a pivot `p` from `a`.
2. Allocate two temporary arrays: `tmpLE` and `tmpG`.
3. Store all elements *less than or equal to* `p` to `tmpLE`.
4. Store all elements *greater than* `p` to `tmpG`.
5. Copy the arrays `tmpLE` and `tmpG` back to `a` and return the index of `p` in `a`.

The time complexity of partitioning is $O(n)$.



# Partitioning array `a` in-place (unstable)

```
partition(array a, int left, int right) {
  pivotindex = choosePivot(a, left, right)
  pivot = a[pivotindex]
  swap a[pivotindex] and a[right]
  leftmark = left
  rightmark = right - 1
  while (leftmark <= rightmark) {
    while (leftmark <= rightmark  and
           a[leftmark] <= pivot)
      leftmark++
    while (leftmark <= rightmark  and
           a[rightmark] >= pivot)
      rightmark--
    if (leftmark < rightmark)
      swap a[leftmark++] and a[rightmark--]
  }
  swap a[leftmark] and a[right]
  return leftmark
}
```

## Partitioning array `a`, using temporary storage (stable)

```
partition(array a, int left, int right) {
  create new array b of size right-left+1
  pivotindex = choosePivot(a, left, right)
  pivot = a[pivotindex]
  acount = left
  bcount = 1
  for ( i = left ; i <= right ; i++ ) {
    if ( i == pivotindex )
      b[0] = a[i]
    else if ( a[i] < pivot ||
                (a[i] == pivot && i < pivotindex) )
      a[acount++] = a[i]
    else
      b[bcount++] = a[i]
  }
  for ( i = 0 ; i < bcount ; i++ )
    a[acount++] = b[i]
  return right-bcount+1
}
```

# Time Complexity of Quicksort

**Best Case:** If the pivot is the *median* in every iteration, then the two partitions have approximately $\frac{n}{2}$ elements.

$\Longrightarrow$ The time complexity is as for Merge Sort, i.e. $O(n \log n)$.

# Time Complexity of Quicksort

**Best Case:** If the pivot is the *median* in every iteration, then the two partitions have approximately $\frac{n}{2}$ elements.

$\implies$ The time complexity is as for Merge Sort, i.e. $O(n \log n)$.

**Worst Case:** If the pivot is always the *least* element in every iteration, then the second partition contains all elements except for the pivot; it has $n-1$ elements. In the consecutive iterations:

the second partition has $n-1, n-2, n-3, ..., 1$ elements.

$\implies$ The time complexity is $O(n^2)$.

# Time Complexity of Quicksort

**Best Case:** If the pivot is the *median* in every iteration, then the two partitions have approximately $\frac{n}{2}$ elements.

$\Longrightarrow$ The time complexity is as for Merge Sort, i.e. $O(n \log n)$.

**Worst Case:** If the pivot is always the *least* element in every iteration, then the second partition contains all elements except for the pivot; it has $n-1$ elements. In the consecutive iterations:

the second partition has $n-1, n-2, n-3, ..., 1$ elements.

$\Longrightarrow$ The time complexity is $O(n^2)$.

**Average Case:** Depends on the strategy which chooses the pivots! If there are $\geq 25\%$ many small entries or $\geq 25\%$ many large entries in almost every iteration, then the partitioning happens approximately $\log_{4/3} n$-many times

$\Longrightarrow$ The time complexity is $O(n \log n)$.

# Pivot-selection strategies

Choose pivot as:

1. the middle entry
   (good for sorted sequences, unlike the leftmost-strategy),
2. the median of the leftmost, rightmost and middle entries,
3. a random entry (there is 50% chance for a good pivot).

**Remark:** In practice, usually 3. or a variant of 2. is used.

Also, for both quicksort and mergesort, when you reach a small region that you want to sort, it's faster to use selection sort or other sort algorithms. The overhead of Q.S. or M.S. is big for small inputs.

Strategies (1) and (2) don't guarantee that the pivot will be such that $\geq$ 25% entries is small and $\geq$ 25% is large for *every input* sequence. However, this property holds *on average* (= for a random sequence).

Strategy (3), although it does not guarantee that we will find a perfect pivot every single time, we pick it *often* (with 50% probability) which suffices.

# Comparison of sorting algorithms

| | Selection Sort | Heap Sort | Merge Sort | Quick Sort temp array (stable) | Quick Sort in-place (unstable) |
|---|---|---|---|---|---|
| **Time Complexity:** | | | | | |
| Average C. | $O(n^2)$ | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ |
| Worst C. | $O(n^2)$ | $O(n \log n)$ | $O(n \log n)$ | $O(n^2)$ | $O(n^2)$ |
| **Space Complexity:** | | | | | |
| Average C. | $O(1)$ | $O(1)$ | $O(n)$ | $O(n)$ | $O(\log n)$ |
| Worst C. | $O(1)$ | $O(1)$ | $O(n)$ | $O(n)$ | $O(n)$ |
| **Stability** | No | No | Yes | Yes | No |

So why is quicksort used so much if its Worst Case complexity is as bad as that of selection sort?

It is because quicksort's constants hidden by the big-O are *smaller*. However, if guaranteed $O(n \log n)$ time complexity is required, it is probably better to use merge sort. Moreover, if we are working with very restricted memory, then it is reasonable to also consider heap sort.

# Non-Comparison Sorts

# Binsort

Binsort is a type of sort that is not based on comparisons between key values but instead simply assigns records to "*bins*" based of the key value of the record alone.

These bins, in Abstract Data Type terms, are Queue data structures, which maintain the order that records are inserted into them.

The final step of the binsort is to concatenate the queues together in order to get a single list of records with all records of the first bin followed by all records of the second bin, etc.

Binsort is a stable sort because values that belong in the same bin are enqueued in the order that they appear in the input.

Binsort does one pass through the input to fill the bins, and one pass through the bins to create the output list, so this is $O(n)$.

## Binsort Example

For example, with a shuffled deck of 52 playing cards you can do a pass through the deck separating out each card into one of 13 piles by their face value (Ace, 2, 3,. . . , Jack, Queen, King). Each pile, or *bin*, would end up with 4 cards with the same face value, but the suits (Hearts, Diamonds, Clubs, Spades) within each bin would still be mixed up. We can now put all the piles together to make a single pile of 52 cards, which are sorted by face value but not by suit.

If we now do another binsort on the pile obtained from our first binsort, but this time based on suits rather than face values, you end up with 4 piles of 13 cards, with one pile for each suit. This time, because of the stability of binsort, each pile **WILL** be sorted by face value: since the input was sorted by face value, cards are put into each suit pile in face value order.

# Further Binsort Examples

Dates are suitable values to do such "*multi-phase*" binsorts on: sort first by day, then by month, then by year to obtain the list of dates in Year, Month, Day order.

A variant on binsort is *bucketsort*, where instead of "*scattering*" records into bins based just on a value (which could be numeric or categorical), they are scattered into buckets based on a range of numeric values or a set of categories.

# Radix Sort

Radix sort is a multi-phase binsort where the key sorted on in each phase is a different, more significant base power of the integer key. For example, in base (or radix) 10, an integer has digits for units, 10s, 100s, 1000s, etc. In a radix sort, a binsort on the units digit is performed first, then on the 10s digit, then on the 100s digit etc. The result final result will be that the keys are sorted first by the most significant digit, then by the next most significant digit, . . . , until finally by the least significant digit. That is, they will be sorted into normal integer order.

# Radix Sort Example

Here we sort a set of numbers using a 3-phase binary radix sort, i.e. the base, or radix of the sort is 2, so there are two bins used: bin 0 and bin 1:

$$\begin{bmatrix}0\\5\\2\\7\\1\\3\\6\\4\end{bmatrix} = \begin{bmatrix}000\\101\\010\\111\\001\\011\\110\\100\end{bmatrix} \rightarrow \left[\begin{array}{c}000\\010\\110\\100\\\hline 101\\111\\001\\011\end{array}\right] \rightarrow \left[\begin{array}{c}000\\100\\101\\001\\\hline 010\\110\\111\\011\end{array}\right] \rightarrow \left[\begin{array}{c}000\\001\\010\\011\\\hline 100\\101\\110\\111\end{array}\right] = \begin{bmatrix}0\\1\\2\\3\\4\\5\\6\\7\end{bmatrix}$$

- Complexity is $O(kn)$, where $k$ is the number of bits in a key
- Reduce to $O\left(\frac{kn}{m}\right)$ by grouping m bits together and using $2^m$ bins, e.g. $m = 4$ and use 16 bins.

# Pigeonhole Sort

A special case is when the keys to be sorted are the numbers from 0 to $n - 1$. This sounds unnecessary, i.e. why not just generate the numbers in order from 0 to $n - 1$?, but remember that these keys are typically just fields in records and the requirement is to put the records in key value order, not just the key values.

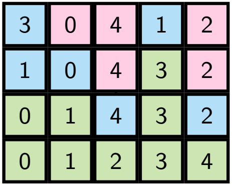
The idea here is to create an output array of size $n$, and iterate through the input list directly assigning the input records to their correct location in the output array. Clearly, this is $O(n)$.

```
pigeonhole_sort(a, n){
  create array b of size n
  for (i = 0 ; i < n ; i++ )
    b[a[i]] = a[i]
  copy array b into array a
}
```

# Pigeonhole Sort in-place

We can avoid allocating the extra array and doing the extra copy as follows:

```
pigeonhole_sort_inplace(a, n){
  for (i = 0 ; i < n ; i++ )
    while ( a[i] != i )
      swap a[a[i]] and a[i]
}
```



| | | | | |
|---|---|---|---|---|
| 3 | 0 | 4 | 1 | 2 |
| 1 | 0 | 4 | 3 | 2 |
| 0 | 1 | 4 | 3 | 2 |
| 0 | 1 | 2 | 3 | 4 |

Every swap results in at least one key in its correct position, and once a key is in its correct position, it is never again swapped, so there are at most $n - 1$ swaps, therefore the sort is $O(n)$

# Summary: Comparison Based Sort Properties

| Sorting Algorithm | Strategy employed | Worst case complexity | Average case complexity | Stable |
|---|---|---|---|---|
| Bubble Sort | Exchange | $O(n^2)$ | $O(n^2)$ | Yes |
| Selection Sort | Selection | $O(n^2)$ | $O(n^2)$ | No |
| Insertion Sort | Insertion | $O(n^2)$ | $O(n^2)$ | Yes |
| Heapsort | Selection | $O(n \log n)$ | $O(n \log n)$ | No |
| Quicksort | D & C | $O(n^2)$ | $O(n \log n)$ | Maybe |
| Mergesort | D & C | $O(n \log n)$ | $O(n \log n)$ | Yes |

# Summary: Empirical Sort Timings

| Algorithm | 128 | 256 | 512 | 1024 | O1024 | R1024 | 2048 |
|---|---|---|---|---|---|---|---|
| Bubble Sort | 54 | 221 | 881 | 3621 | 1285 | 5627 | 14497 |
| Selection Sort | 12 | 45 | 164 | 634 | 643 | 833 | 2497 |
| Insertion Sort | 15 | 69 | 276 | 1137 | 6 | 2200 | 4536 |
| Heapsort | 21 | 45 | 103 | 236 | 215 | 249 | 527 |
| Quicksort | 12 | 27 | 55 | 112 | 1131 | 1200 | 230 |
| Quicksort2 | 6 | 12 | 24 | 57 | 1115 | 1191 | 134 |
| Mergesort | 18 | 36 | 88 | 188 | 166 | 170 | 409 |
| Mergesort2 | 6 | 22 | 48 | 112 | 94 | 93 | 254 |

- Column titles show the number of items sorted
- O1024: 1024 items already in ascending order
- R1024: 1024 items already in descending order
- Quicksort2 and Mergesort2: sort switches to selection sort during recursion once size of array drops to 16 or less.

# Hash tables

# Basic idea

**Goal:** We would like to be able to *calculate* the location of our search target without having to actually search for it.

A **hash function** `hash(key)` computes the location from a `key`

Example: Storing international dialing codes in an array:

```
dialCode[hash("UK")] = "+44"
```

In practice, we will need an ADT, rather than a simple array:

```
dialCode.put("UK", "+44")
```

However, some programming languages with built-in hash ADTs, like **Python**, extend the array syntax:

```
dialCode["UK"] = "+44"
```

Maybe you have seen the syntax `dialCode["UK"]` in Python, JavaScript, PHP or other programming languages. Even though it looks like those languages allow indexing of arrays by strings, internally it is always implemented by using hash tables.

It is important that every time we compute the index of a key by a hash function, we always get the same index.

# Example: Storing students' assignments in $O(1)$

In Canvas, say we store students' submissions in a hash table:

- `value` = assignment submission
- `key` = student's ID number

Student IDs of the form 2183201, 1526020, ... 7-digit numbers

Allocate an array `submissions` of size $10^7$ and choose the hashcode to be the identity function: `hash(s)` returns s.

Then, to store an assignment:

```
submissions[hash(s)] = assignment
```

This is in $O(1)$ but **VERY** memory inefficient!

Even if we only need to store assignments of 500 students, we still allocate an array of size $10^7$

## Example: Hash function based on the size of the array

Allocate an array `arr` of size 500 and compute `hash(s)` as

`s mod 500`.

Now `hash(s)` is one of `0`, `1`, `2`, ... `submissions.length-1`.

**But** this might introduce **hash collisions**. That is, we can have

`hash(key1) == hash(key2)`

for two different keys `key1` and `key2`.

Collisions will happen even if we double/triple the size of `arr`.

⟹ We need a mechanism for dealing with hash collisions.

# Summary

In summary, a **hash table** consists of

1. an array `arr` for storing the values,
2. a hash function `hash(key)`, and
3. a mechanism for dealing with collisions.

It implements (at least) the operations:

`set(key, value)`, `delete(key)`, `lookupValue(key)`.

---

**NOTE:** We will consider a simplified situation where `key`s and `value`s are the same. For example, an assignment is always:

`arr[hash(key)] = key`.

And the operations change to: `insert(key)`, `delete(key)`, `lookup(key)`.

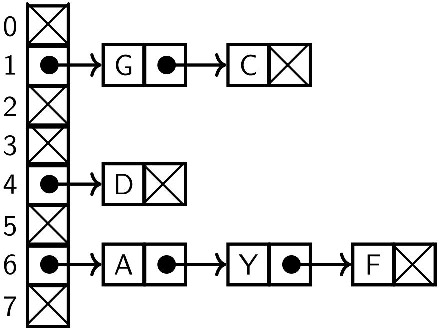
Whereas `lookupValue(key)` returns the value stored on the position given by `key`, `lookup(key)` returns `true` or `false` based on whether `key` is stored in the hash table.

The reason why we explain the simplified situation is because it is easier to illustrate the main ideas this way. However, this simplified situation is also often useful on its own. In Java there is even a class called `HashSet` which works exactly this way.

**Note:** The only difference between the simplified and unsimplified situations is that, instead of storing the key only, we need to store both the key and the value.

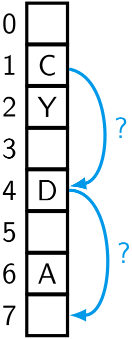
# Two types of solutions of hash collisions

## Chaining strategy



Entries with the same `hash(key)` are stored in a linked list.

## Open Addressing strategy



If the position is occupied, we try different "fallback" positions.

The *chaining* strategies store an extra data structure on each position of the hash table. Those could be linked lists, another hash table, or even something completely different. In the following we only consider one chaining strategy, called *direct chaining*, which uses linked lists to store the values with the same `hash(code)`.

The main idea behind *open addressing* strategies is that, in case of collisions, we find a different address (from a sequence of "fallback" addresses) in the same array for the colliding value to be stored, that is, an address which is currently unused, or *"open"*, hence the name *open addressing*. In this module we consider the following two open addressing strategies:

- Linear probing
- Double hashing

# Direct chaining (= a chaining strategy)

**Entries:** airport codes, e.g. BHX, INN, HKG, IST, ...

**Table size:** 10

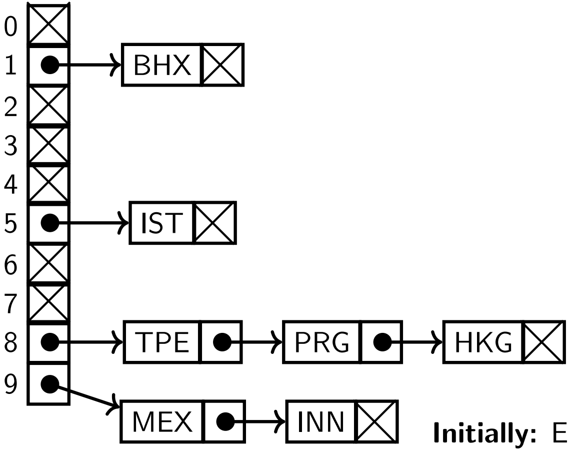
**Hash function:**

- We treat the codes as a number in base 26 (A=0, B=1, ..., Z=25).
  Example: $\text{ABC} = 0 * 26^2 + 1 * 26 + 2 = 28$
- The hashcode is computed `mod 10` (to make sure that the index is $0, 1, 2, 3, ...,$ or 9).
  Example:
  `hash(BHX) = 1*26*26 + 7*26 + 23 mod 10 = 1`

| `key` | BHX | INN | HKG | IST | MEX | PRG | TPE |
|---|---|---|---|---|---|---|---|
| `hash` | 1 | 9 | 8 | 5 | 9 | 8 | 8 |

## Direct chaining

| key | BHX | INN | HKG | IST | MEX | PRG | TPE |
|---|---|---|---|---|---|---|---|
| hash | 1 | 9 | 8 | 5 | 9 | 8 | 8 |



**Initially:** Empty lists on all positions.

To insert, we always first check if the `key` which we are inserting is in the linked list on position `hash(key)`. If it isn't, we insert the `key` at the beginning of that list.

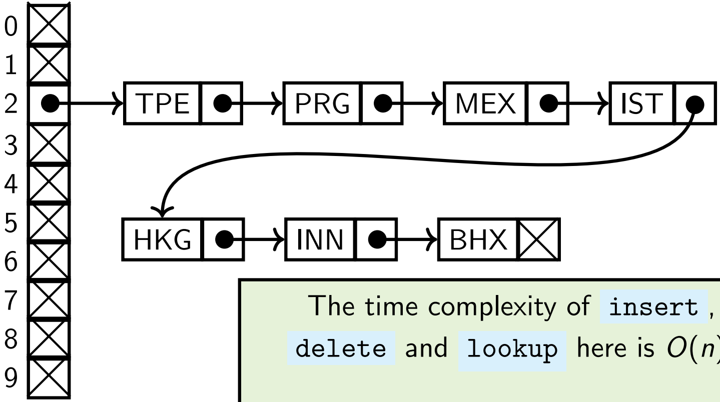
(We are inserting without duplicates.)

To `delete(key)` we delete `key` from the linked list stored on position `hash(key)`, if it is there. Similarly, `lookup(key)` returns `true`/`false` depending on if `key` is stored in the list on position `hash(key)`.

**Note:** The choice to insert the `key` at the beginning of the list and not at the end is not so important. Inserting at the beginning is more common (probably) because, in practice, the just inserted `key` is more likely to be accessed soon again, as opposed to the key at the end of the list.

# Bad hash functions

| `key` | BHX | INN | HKG | IST | MEX | PRG | TPE |
|---|---|---|---|---|---|---|---|
| `hash` | 2 | 2 | 2 | 2 | 2 | 2 | 2 |



The time complexity of `insert`, `delete` and `lookup` here is $O(n)$!

A **good** hash function `hash(key)` assigns indexes to keys **uniformly**.

We see that the hash function assigns `2` to all keys. Then, when inserting a new `key` we first check if `key` is stored in the linked list on position `hash(key) = 2`. This requires to go through all the elements already stored in the hash table $\implies O(n)$ time complexity.

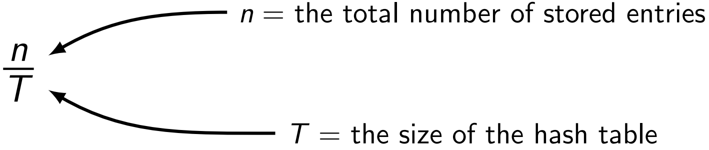
Similarly, `delete` and `lookup` are also in $O(n)$.

To tackle this, we need to have a **good** hash function which uniformly distributes the keys among positions. In other words, given a random `key`, it ought to have the same probability of being stored on every position.

**Remark:** Notice that whether a function is good or not also depends on the *distribution* of your data/keys. (You don't want the two most likely keys to share the same hash key, for example.) When the distribution is not known, one assumes that all keys are equally likely.

# Time Complexity of Direct Chaining, part 1

The **load factor** of a hash table is the *average* number of entries stored on a location:



$$\frac{n}{T}$$

$n =$ the total number of stored entries

$T =$ the size of the hash table

If we have a *good* hash function, a location given by `hash(key)` has the *expected* number of entries stored there equal to $\frac{n}{T}$.

**Unsuccessful lookup** of `key`:

- `key` is not in the table.
- Location `hash(key)` stores $\frac{n}{T}$ entries, *on average*.
- $\Longrightarrow$ We have to traverse them all.

The load factor represents how full the hash table is. Assuming we have a good hash function, the load factor 0.25 represents 25% probability of getting a collision.

A consequence of having a good hash function is that the linked list on position `hash(key)`, for a randomly selected `key`, has *expected* length $\frac{n}{T}$.

The word "expected" has a well-defined meaning in probability theory. Intuitively speaking, it means that the list stored on position `hash(key)` might be longer, it might be shorter, but it's length will most likely be approximately $\frac{n}{T}$ (for a randomly selected `key`).

# Time Complexity of Direct Chaining, part 2

**Successful lookup** of `key`:

- Location `hash(key)` stores $k = \frac{n}{T}$ entries on average.
- On average, A linear search in a linked list of $k$ elements takes $\frac{1}{k}(1 + 2 + \cdots + k) = \frac{k(k+1)}{2k} = \frac{(k+1)}{2}$ comparisons

Assume **maximal load factor** $\lambda$, that is, $\frac{n}{T} \leq \lambda$

(For example, in Java $\lambda = 0.75$)

The *average case* time complexities:

- unsuccessful lookup: $\frac{n}{T} \leq \lambda$ comparisons $\implies O(1)$
- successful lookup: $\frac{1}{2}(1 + \frac{n}{T}) \leq \frac{1}{2}(1 + \lambda)$ comparisons $\Rightarrow O(1)$

$\lambda$ is a constant number!

# Time Complexity of Direct Chaining, part 3

The time complexity of `insert(key)` is the same as unsuccessful lookup:

- First check if the `key` is stored in the table.
- If it is not, insert `key` at the beginning of the list stored on `hash(key)`.

In total: $\frac{n}{T} + 1 \leq \lambda + 1 \implies O(1)$.

The time complexity of `delete(key)` is the same as successful lookup.

$\implies$ The time complexities of `insert`, `delete`, `lookup` are all $O(1)$.

To summarise, we made two assumptions:

1. We have a *good hash function*.
2. We assume a *maximal load factor*.

A consequence of the first assumption is that the expected length of chains is $\frac{n}{T}$ and the second one is that $\frac{n}{T} \leq \lambda$, for some fixed constant number $\lambda$.

By assuming those two conditions, we have computed that the operations of hash tables are all in $O(1)$.

Whether a hash function is *good* depends on the distribution of the data. On the other hand, making sure that the load factor is bounded by some $\lambda$ can be done automatically. We will show how to do this later on. The consequence of our approach will be that the constant time complexity will be (only) *amortized*.

# Disadvantages of “chaining” strategies

1. Typically, there are a lot of hash collisions, therefore a lot of unused space.
2. Linked lists require a lot of allocations (`allocate_memory`), which is slow.

We will take a look at two **open addressing strategies** which avoid those problems:

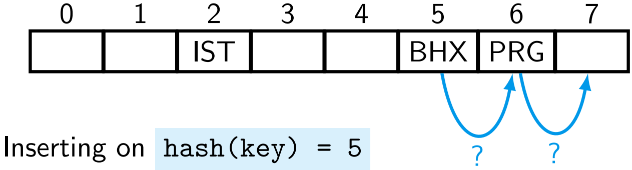
- Linear probing
- Double hashing

# Linear probing (= an open addressing strategy)

**Insertion (initial idea):** If the primary position `hash(key)` is occupied, search for the first *available* position to the right of it.

If we reach the end, we wrap around.

**Example**



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
|  |  | IST |  |  | BHX | PRG |  |

Inserting on `hash(key) = 5`

We use `mod` to compute the “fallback” positions:

`hash(key)+1 mod T`, `hash(key)+2 mod T`, `hash(key)+3 mod T`, ...

# Linear Probing: Deleting

**Deletion (idea):**

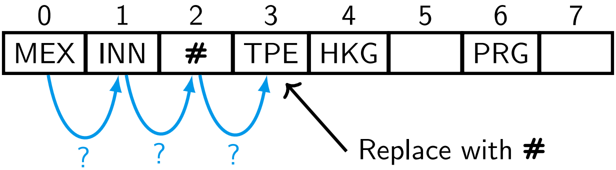
1. Find whether the `key` is stored in the table:

   Starting from the primary position `hash(key)`, go the right, until the `key` or an empty position is found.

2. If the `key` is stored in the table, replace it with a marker value, called a **tombstone** (marked as **#**).

**Example**

Deleting `key = TPE` such that `hash(key) = 0`:



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| MEX | INN | **#** | TPE | HKG | | PRG | |

? ? ?

Replace with **#**

# Linear Probing: Searching and Inserting

**Searching:**
Starting from the primary position `hash(key)`, search for the `key` to the right. We skip over all **tombstones #**.

If we reach an empty position, then the `key` is not in the table.

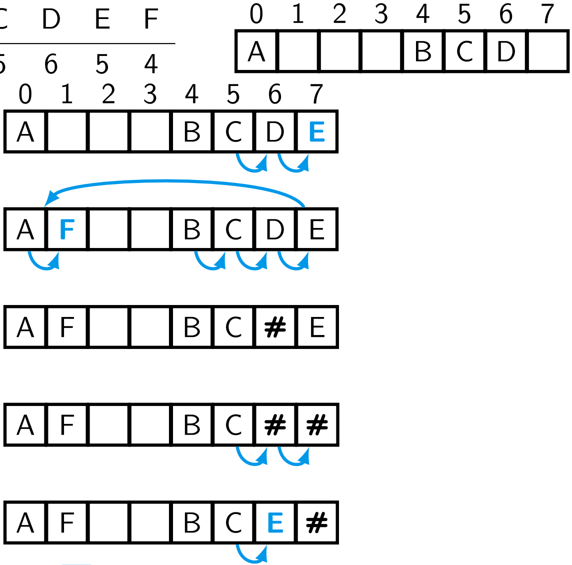
**Inserting (more accurately):**
Search for the `hash(key)` as above but note the location of the first tombstone we found, if any. If we find `key`, signal an error.

If we reach an empty position, then the `key` is not in the table, so insert the `key` in the noted tombstone location, if any, otherwise in the empty position found.

**Remark**
Every position is either **empty**, or it stores a **tombstone** or a **key**. Moreover, initially, all positions are marked as *empty*.

# Example: Linear probing

| key | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| hash | 0 | 4 | 5 | 6 | 5 | 4 |



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| A | | | | B | C | D | |

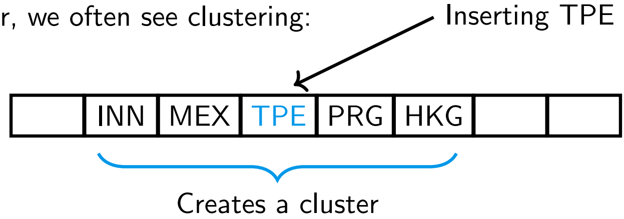
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 1. insert(E) | A | | | | B | C | D | **E** |
| 2. insert(F) | A | **F** | | | B | C | D | E |
| 3. delete(D) | A | F | | | B | C | # | E |
| 4. delete(E) | A | F | | | B | C | # | # |
| 5. insert(E) | A | F | | | B | C | **E** | # |

(Note we checked that E is not stored by searching until position 2)

# Time Complexity and Clustering

`insert`, `search` and `delete` have the time complexity $O(1)$. (This is much more difficult to calculate.)

However, we often see clustering:



Inserting TPE

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | INN | MEX | TPE | PRG | HKG | | |

Creates a cluster

**Primary clusters** are clusters caused by entries with the same hash code. **Secondary clusters** are caused when the collision handling strategy causes different entries to check the same sequence of locations when they collide.

Clusters are more likely to get bigger and bigger, even if the load factor is small. To make clustering less likely, use **double hashing**.

# Double hashing

Use primary and secondary hash functions `hash1(key)` and `hash2(key)`, respectively.

**Insertion:** We try the primary position `hash1(key)` first and, if it fails, we try fallback positions:

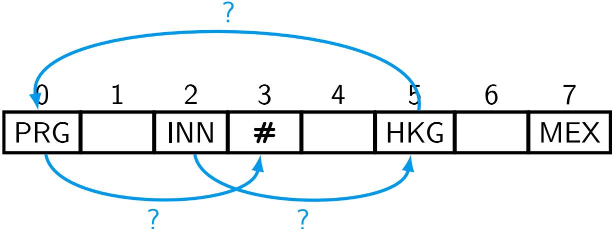
1. `(hash1(key) + 1*hash2(key)) mod T`
2. `(hash1(key) + 2*hash2(key)) mod T`
3. `(hash1(key) + 3*hash2(key)) mod T`
4. …

(until we find an available space)

`T` is the table size

**Example**

If `key = TPE`, `hash1(key) = 2`, `hash2(key) = 3`:



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| PRG |  | INN | # |  | HKG |  | MEX |

Double hashing is an improvement of linear probing. The only difference is that every `key` has a different sequence of "fallback" positions given by the secondary hash function.

Except for how we calculate the fallback positions, all the operations (`insert`, `delete` and `lookup`) work the same way; we use tombstones to mark deleted keys, when looking up we skip over those tombstones etc.

Linear probing's fallback positions are:

`(hash(key) + i) mod T` for `i = 1, 2, 3, ...`

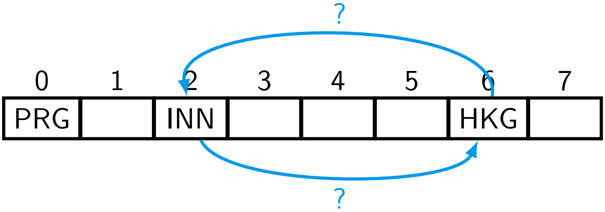
whereas double hashing's fallback positions are:

`(hash1(key) + i*hash2(key)) mod T` for `i = 1, 2, 3, ...`

# Avoiding short cycles

We can have short cycles!

Consider inserting a `key` such that `hash1(key) = 2` and `hash2(key) = 4` into a table of size 8:



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| PRG |  | INN |  |  |  | HKG |  |

The table size $T$ and `hash2(key)` have to be coprime!

**Two solutions:**

(a) $T$ is a prime number.

(b) $T = 2^k$ and `hash2(key)` is always an odd number. **(preferred)**

Maths break:

- Two numbers $a$ and $b$ are said to be *coprime* if no number, other than 1, divides both $a$ and $b$
- *Prime numbers* are the numbers which are divisible only by 1 and themselves.

# What to do if the table is full?

We say that a hash table is **full** if the load factor is more than the maximal load factor, that is,

$$\frac{n}{T} > \lambda.$$

**Rehashing (idea):** If the table becomes full after an insertion, allocate a new table twice the size and `insert` all elements from the old table into it.

Consequences for `insert`:

- the Worst Case time complexity is $O(n)$ (when rehashing) but
- the *amortized* time complexity is $O(1)$!
  - calculation is similar to that for dynamic arrays

(Rehashing can be used for direct chaining, linear probing, or double hashing and always leads to constant amortized time complexities.)

This combines well with our extra assumption that $T = 2^k$ in order to avoid short cycles. If we start from an empty hash table of such size (for example, we initially have $T = 2^3 = 8$), then doubling the size always ensures that $T = 2^k$ for some (natural) number $k$.

**Remark:** If we double the size of the hash table, we also need to change the (primary) hash function to make sure that it is *good* again. In practice, `hash(key)` is usually computed as `bigHash(key) mod T` (where `bigHash` computes a "big" hashcode).

Then, after doubling the size of our hash table we only modify `hash(key)` as follows

`bigHash(key) mod 2*T`.

# Summary

Hash tables are ADTs with an implementation consisting of an array `arr`, a primary hash function `hash1(key)` (and possibly a secondary hash function `hash2(key)`)

All operations are in $O(1)$ (amortized time) if

1. `hash1` (and `hash2`) computes indexes uniformly,
2. we `rehash` whenever the table becomes full,
3. ($T = 2^k$ for some $k$, and `hash2` gives odd numbers).

They do not offer an efficient way to obtain entries in key order

**Comparison with trees**
AVL Trees require keys to be *comparable* and the operations are in $O(\log n)$, best, worst and average case.

Hash tables, on the other hand, require *good hash functions*. Then, operations are in $O(1)$ *amortized* time complexity.

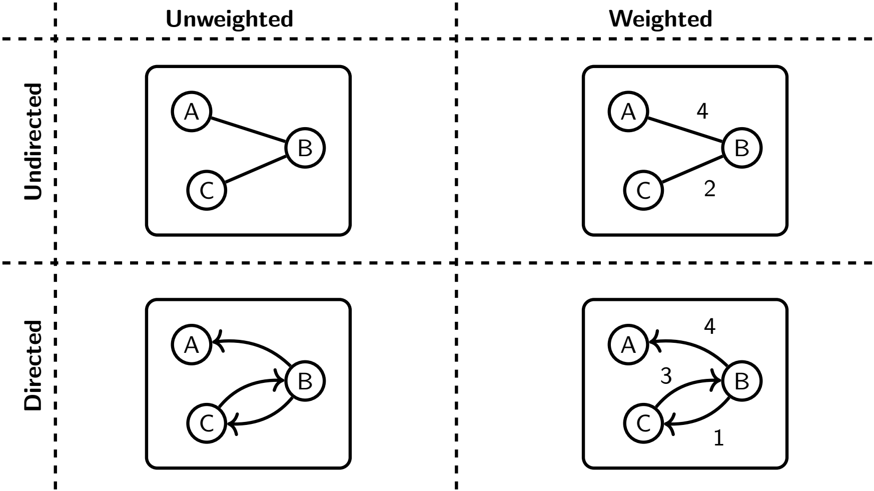
## Final thoughts

- Insert, delete and search in Direct Chaining, Linear Probing or Double Hashing all have $O(1)$ amortised complexity.
- Double hashing has an performance advantage because `allocate_memory` in chaining has a large constant cost and clustering in linear probing is worse.
- In chaining, if the load factor drops below a minimum threshold, we can rehash into a hash table half the size. This is rarely done because it does not speed up performance.
- In open addressing hash tables, We keep track of the number of tombstones in the table. If this exceeds some threshold, we also rehash but without doubling the size. With many tombstones, we might even halve the size of the hash table.
- As a consequence `delete` is also $O(1)$ *amortized* time complexity.

# Graphs and graph algorithms

# Graph Types

A **graph** is formed of a (finite) set of vertices/nodes and a set of edges between them. We distinguish four types of graphs:



# Examples

**Undirected unweighted graph:**

- vertices = registered people on Facebook
- edges = friendships between people (it is mutual!)

**Directed unweighted graph:**

- vertices = registered people on Twitter
- edges = who is following who

**Undirected weighted graph:**

- vertices = train stops/stations
- edges = rail lines connecting train stops with distance

**Directed weighted graph:**

- vertices = bank accounts
- edges = bank transfers with amounts transferred

## Graph Concepts

- In maths, we tend to use the term **vertex**, in computer science, we use both **node** and **vertex** interchangeably
- If the graph is undirected, an edge between nodes `u` and `w` can be thought of as having two edges `u` → `w` and `w` → `u`.
- A directed graph is often called a **digraph**
- A graph is called **simple** if
  1. it has no self loops, i.e. edges connected at both ends to the same node, and
  2. it has no more than one edge between any pair of nodes
- A **path** is a sequence of vertices $v_1, v_2, \ldots, v_n$ such that $v_i$ and $v_{i+1}$ are connected by an edge for all $1 \leq i \leq n - 1$.
- A **cycle** is a non-empty path whose first vertex is the same as its last vertex.
- A path is **simple** if no vertex appears on it twice (except for a cycle, where the first and last vertex may be the same)

## Graph Concepts

- An undirected graph is **connected** if every pair of vertices has a path connecting them.
- A directed graphs is:
  - **weakly connected** if for every two vertices $A$ and $B$ there is either a path from $A$ to $B$ or a path from $B$ to $A$.
  - **strongly connected** if there are paths leading both ways.
- A **tree** can be viewed as a simple connected graph with no cycles and one node identified as a root
- A graph, unlike a tree, does not have a **root** from which there is a unique path to each vertex, so it does not make sense to speak of parents and children in a graph.
- Two vertices $A$ and $B$ connected by an edge $e$ are called **neighbours**, and $e$ is said to be **incident** to $A$ and $B$.
- Two edges with a vertex in common (e.g., one connecting $A$ and $B$ and one connecting $B$ and $C$) are said to be **adjacent**.

## Graph represented as an Adjacency Matrix

Assume that graph's vertices are numbered $V = \{0, 1, 2, ..., n-1\}$.

*Adjacency matrix* `G` is a two-dimensional array/matrix $n \times n$ where each cell `G[v][w]` contains information about the connection from vertex v to vertex w

**Unweighted graphs:**

- `G[v][w]` $= 1$ if there is an edge going from `v` to `w`
- `G[v][w]` $= 0$ if there is no such edge

**Weighted graphs:**

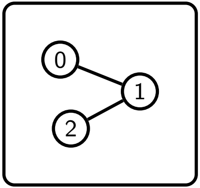
- `G[v][w]` $=$ weight of the edge going from `v` to `w`
- `G[v][w]` $= \infty$ if there is no such edge*
- `G[v][v] = 0` *

**Remark:** The graph is undirected if `G[v][w] = G[w][v]` for all vertices `v` and `w`.

*other values possible depending on application

# Example: Adjacency matrix

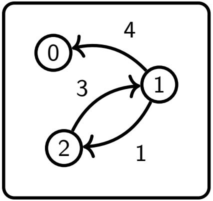
## Unweighted undirected:



$$\texttt{G} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

For example: `G[2][0] = 0` and `G[2][1] = 1`.

## Weighted directed:



$$\texttt{G} = \begin{pmatrix} 0 & \infty & \infty \\ 4 & 0 & 1 \\ \infty & 3 & 0 \end{pmatrix}$$

For example: `G[2][0] = infty` and `G[2][1] = 3`.

# Graph represented as Adjacency Lists

To represent a graph on vertices $V = \{0, 1, 2, \ldots, n - 1\}$ by *adjacency lists* we have an array `N` of *n*-many linked lists (one list for every vertex).

**Unweighted:**

- `N[v]` is the list of *neighbours* of `v`.

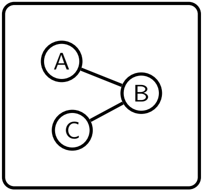
  ( `w` is a neighbour of `v` if there is an edge `v` $\to$ `w` )

**Weighted:**

- `N[v]` is the list of *neighbours* of `v` together with the weight of the edge that connects them with `v`.

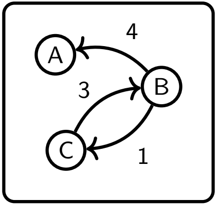
# Example: adjacency lists

## Unweighted undirected:



| `N[v]` | neighbours |
|---|---|
| A | B |
| B | A, C |
| C | B |

## Weighted directed:

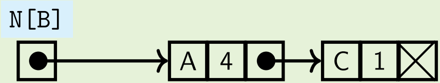


| `N[v]` | neighbours & weights |
|---|---|
| A | |
| B | (A, 4), (C,1) |
| C | (B, 3) |

We said that representing a graph by adjacency lists means that we will have an array `N` of *n*-many linked list (where *n* is the number of vertices). Then, for example, `N[2]` stores the address of the head of the linked list of all neighbours of the `2`nd vertex. If we name our vertices by letters `A`, `B`, `C`, for example, we need to find a way to assign indexes of the array `N` to the letters `A`, `B`, `C`. One way to do this is to use hash tables.

However, in the example given here, we don't care how this is done. We assume that we have lists of neighbours stored in `N[A]`, `N[B]`, `N[C]`.

In the weighted case, `N[B]` also stores the weights of the edges:



But instead of drawing this we just say that `N[B]` stores the list (A, 4), (C,1).

# Comparison of those two methods

Set $n =$ the number of vertices, $m =$ the total number of edges.

| | **Adjacency matrix** | **Adjacency lists** |
|---|---|---|
| Checking if there is an edge `v` → `w`: | Reading `G[v][w]` (which is in $O(1)$) | Checking if `w` is in in the list `N[v]` |
| Allocated space: | $n$ arrays of size $n$ $= O(n \times n)$ space | $n$ linked lists storing $m$ edges in total $= O(n + m)$ space |
| Traversing `v`'s neighbours: | Traversing all `G[v][0]`, `G[v][1]`, .., `G[v][n-1]`. $= O(n)$ time | Traversing only the linked list `N[v]` |

In the third case (with adjacency lists) we only traverse the actual neighbours of `v`. This is better whenever the graph is **sparse** (= not dense), that is, if there are relatively few edges.

A graph is **sparse** if it has few edges relative to its number of vertices, e.g. $\frac{m}{n}$ is small. For simple graphs, some authors say it is sparse if $m$ is $O(n)$ or less, rather than $O(n^2)$

An example of a sparse graph would be the graph of Facebook users with edges representing friendships. Facebook has hundreds of millions of users but each user has only a few hundreds of friends. In other words, every vertex of the graph has only a few hundreds of neighbours.

From the table we see that checking whether an edge exists is much faster for adjacency matrices than for adjacency lists. On the other hand, if our graph is sparse, then the allocated space of adjacency lists is much smaller than adjacency matrices and also traversing neighbours is faster for adjacency lists than adjacency matrices.

# Shortest Paths and Dijkstra's Algorithm

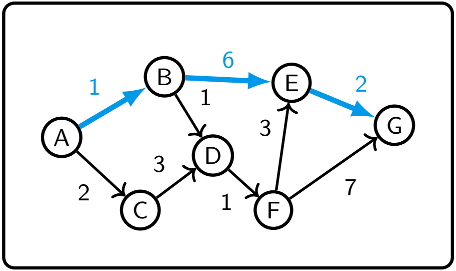
# Paths and shortest paths

Recall: A **path** is a sequence of vertices $v_1, v_2, \dots, v_n$ such that $v_i$ and $v_{i+1}$ are connected by an edge for all $1 \leq i \leq n-1$.

A **shortest path** from $A$ to $B$ is a path for which the sum of the weights along the path is less than or equal to the sum of the weights along any other path from $A$ to $B$. Note that there may be multiple different shortest paths from $A$ to $B$. (In unweighted graphs, set weights to 1.)

**Example**

1. $A \to B \to E \to G$



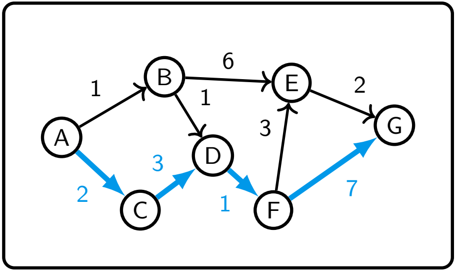
# Paths and shortest paths

Recall: A **path** is a sequence of vertices $v_1, v_2, \ldots, v_n$ such that $v_i$ and $v_{i+1}$ are connected by an edge for all $1 \leq i \leq n-1$.

A **shortest path** from $A$ to $B$ is a path for which the sum of the weights along the path is less than or equal to the sum of the weights along any other path from $A$ to $B$. Note that there may be multiple different shortest paths from $A$ to $B$. (In unweighted graphs, set weights to 1.)

**Example**

1. $A \rightarrow B \rightarrow E \rightarrow G$
2. $A \rightarrow C \rightarrow D \rightarrow F \rightarrow G$
3. ...



# Paths and shortest paths

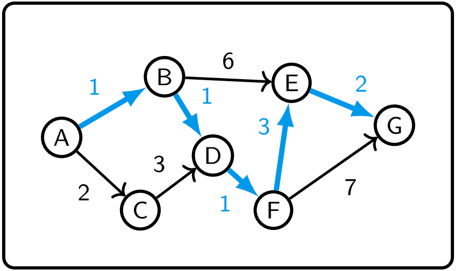
Recall: A **path** is a sequence of vertices $v_1, v_2, \ldots, v_n$ such that $v_i$ and $v_{i+1}$ are connected by an edge for all $1 \leq i \leq n-1$.

A **shortest path** from $A$ to $B$ is a path for which the sum of the weights along the path is less than or equal to the sum of the weights along any other path from $A$ to $B$. Note that there may be multiple different shortest paths from $A$ to $B$. (In unweighted graphs, set weights to 1.)

**Example**

1. $A \to B \to E \to G$
2. $A \to C \to D \to F \to G$
3. ...

The shortest: $A \to B \to D \to F \to E \to G$



## **Dijkstra's algorithm** to find the shortest path from `v` to `z`

For each vertex `w` of the graph other than `v`, we keep track of the following:

i. `d[w] =` the shortest distance from `v` to `w` so far
   (Initially: $\infty$, except `d[v]` $= 0$)

ii. `p[w] =` the predecessor on the path from `v`
   (initially: `w` itself, just a convention)

iii. `f[w] =` is computation of `d[w]` *finished*?
   (initially: `false`)

# The algorithm

The algorithm (idea):

```
set d[v] = 0                                                        (i.e. start on v)
while there are unfinished vertices:
    set w = the yet unfinished vertex with the smallest d[w]
    set f[w] = true                                        (i.e. mark w as finished)
    for every neighbour u of w:
        if d[w] + weight(w,u) < d[u]:
            set d[u] = d[w] + weight(w,u) and p[u] = w
```

(Where `weight(w,u)` is the weight of the edge `w` → `u`)

The input of the algorithm is a graph (represented as an adjacency matrix or adjacency lists) and two vertices `v` and `z`. The aim is to find the shortest path from `v` to `z`.

As the algorithm runs it changes the values `d[w]`, `p[w]` and `f[w]`. Initially `d[w] = infinity`, `p[w] = w` and `f[w] = false` for every vertex `w`.

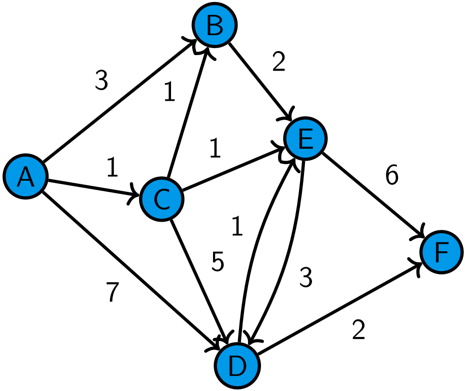
The arrays `d` and `f` obeys the following *invariant*:

- `d[w]` is the length of the shortest path from `v` to `w` when using only the finished vertices (i.e. those `w` such that `f[w] == true`).
- If `w` is finished then `d[w]` is the actual length of the shortest path from `v` to `w`.

After the algorithm finishes, we compute the found shortest path by using the array `p`. Lastly, `weight(w,u)` is the weight of the edge `w` → `u` obtained from the adjacency matrix/lists of the graph.

# Example: Execution of Dijkstra's algorithm

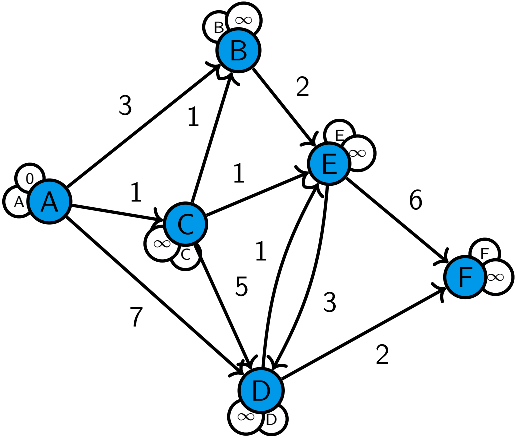
Shortest Path $A \rightarrow F$



# Example: Execution of Dijkstra's algorithm

Shortest Path $A \to F$

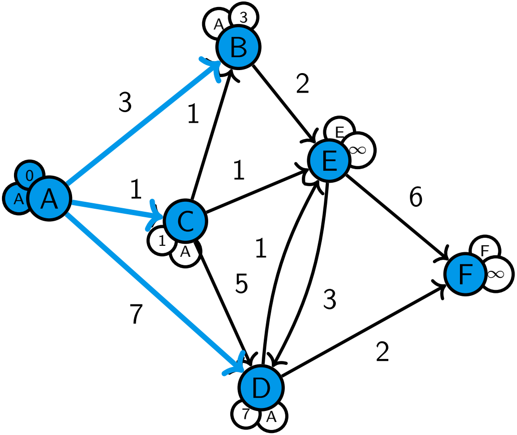
| A | B | C | D | E | F | finished |
|---|---|---|---|---|---|---|
| 0,A | $\infty$,B | $\infty$, C | $\infty$,D | $\infty$,E | $\infty$,F | |



# Example: Execution of Dijkstra's algorithm

Shortest Path $A \to F$

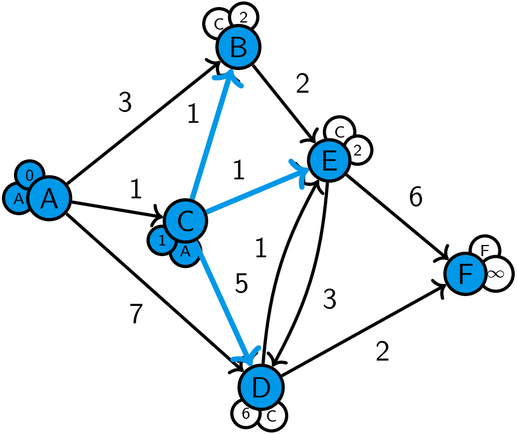
| A | B | C | D | E | F | finished |
|---|---|---|---|---|---|---|
| 0,A | $\infty$,B | $\infty$, C | $\infty$,D | $\infty$,E | $\infty$,F | |
| 0,A,✓ | 3,A | 1,A | 7,A | $\infty$,E | $\infty$,F | A |



# Example: Execution of Dijkstra's algorithm

Shortest Path $A \to F$

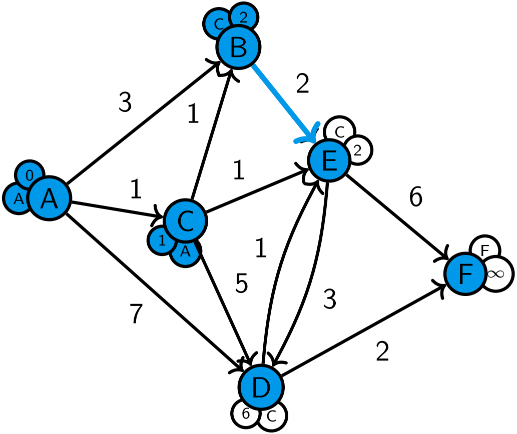
| A | B | C | D | E | F | finished |
|---|---|---|---|---|---|---|
| 0,A | ∞,B | ∞, C | ∞,D | ∞,E | ∞,F | |
| 0,A,✓ | 3,A | 1,A | 7,A | ∞,E | ∞,F | A |
| 0,A,✓ | 2,C | 1,A,✓ | 6,C | 2,C | ∞,F | C |



# Example: Execution of Dijkstra's algorithm

Shortest Path $A \rightarrow F$

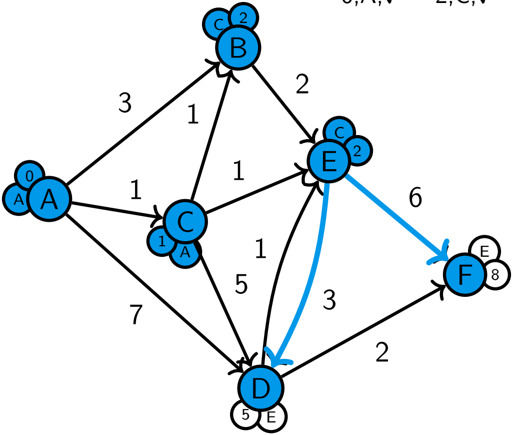
| A | B | C | D | E | F | finished |
|---|---|---|---|---|---|---|
| 0,A | ∞,B | ∞, C | ∞,D | ∞,E | ∞,F | |
| 0,A,✓ | 3,A | 1,A | 7,A | ∞,E | ∞,F | A |
| 0,A,✓ | 2,C | 1,A,✓ | 6,C | 2,C | ∞,F | C |
| 0,A,✓ | 2,C,✓ | 1,A,✓ | 6,C | 2,C | ∞,F | B |



# Example: Execution of Dijkstra’s algorithm

Shortest Path $A \to F$

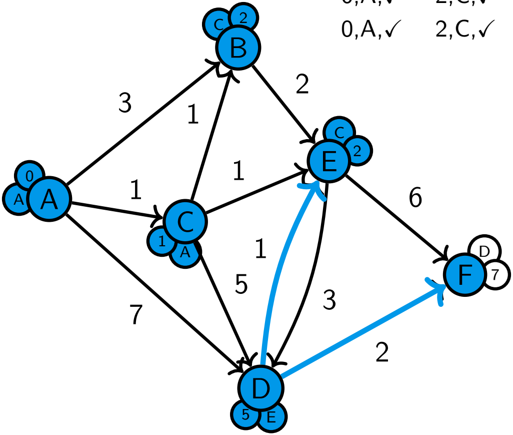
| A | B | C | D | E | F | finished |
|---|---|---|---|---|---|---|
| 0,A | ∞,B | ∞, C | ∞,D | ∞,E | ∞,F | |
| 0,A,✓ | 3,A | 1,A | 7,A | ∞,E | ∞,F | A |
| 0,A,✓ | 2,C | 1,A,✓ | 6,C | 2,C | ∞,F | C |
| 0,A,✓ | 2,C,✓ | 1,A,✓ | 6,C | 2,C | ∞,F | B |
| 0,A,✓ | 2,C,✓ | 1,A,✓ | 5,E | 2,C,✓ | 8,E | E |



# Example: Execution of Dijkstra's algorithm

Shortest Path $A \to F$

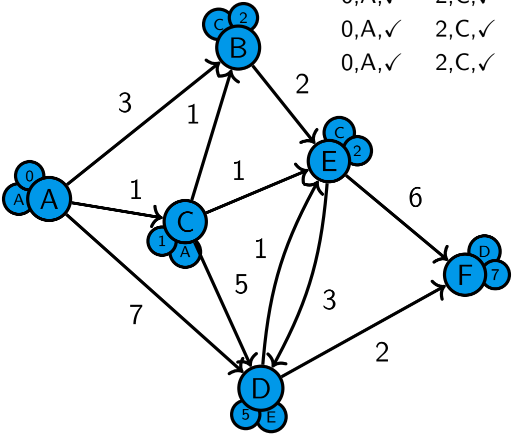
| A | B | C | D | E | F | finished |
|---|---|---|---|---|---|---|
| 0,A | ∞,B | ∞, C | ∞,D | ∞,E | ∞,F | |
| 0,A,✓ | 3,A | 1,A | 7,A | ∞,E | ∞,F | A |
| 0,A,✓ | 2,C | 1,A,✓ | 6,C | 2,C | ∞,F | C |
| 0,A,✓ | 2,C,✓ | 1,A,✓ | 6,C | 2,C | ∞,F | B |
| 0,A,✓ | 2,C,✓ | 1,A,✓ | 5,E | 2,C,✓ | 8,E | E |
| 0,A,✓ | 2,C,✓ | 1,A,✓ | 5,E,✓ | 2,C,✓ | 7,D | D |



# Example: Execution of Dijkstra's algorithm

Shortest Path $A \to F$

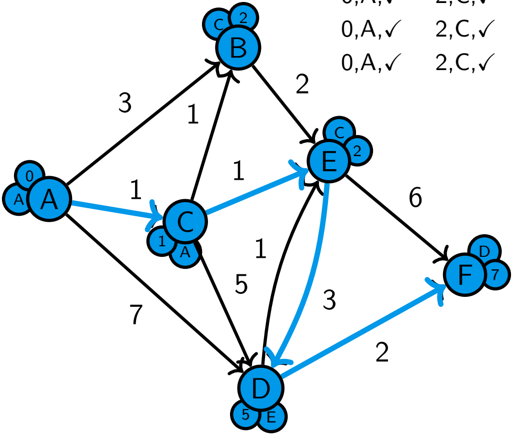
| A | B | C | D | E | F | finished |
|---|---|---|---|---|---|---|
| 0,A | $\infty$,B | $\infty$, C | $\infty$,D | $\infty$,E | $\infty$,F | |
| 0,A,✓ | 3,A | 1,A | 7,A | $\infty$,E | $\infty$,F | A |
| 0,A,✓ | 2,C | 1,A,✓ | 6,C | 2,C | $\infty$,F | C |
| 0,A,✓ | 2,C,✓ | 1,A,✓ | 6,C | 2,C | $\infty$,F | B |
| 0,A,✓ | 2,C,✓ | 1,A,✓ | 5,E | 2,C,✓ | 8,E | E |
| 0,A,✓ | 2,C,✓ | 1,A,✓ | 5,E,✓ | 2,C,✓ | 7,D | D |
| 0,A,✓ | 2,C,✓ | 1,A,✓ | 5,E,✓ | 2,C,✓ | 7,D,✓ | F |



# Example: Execution of Dijkstra's algorithm

Shortest Path $A \to F$

| A | B | C | D | E | F | finished |
|---|---|---|---|---|---|---|
| 0,A | $\infty$,B | $\infty$, C | $\infty$,D | $\infty$,E | $\infty$,F | |
| 0,A,✓ | 3,A | 1,A | 7,A | $\infty$,E | $\infty$,F | A |
| 0,A,✓ | 2,C | 1,A,✓ | 6,C | 2,C | $\infty$,F | C |
| 0,A,✓ | 2,C,✓ | 1,A,✓ | 6,C | 2,C | $\infty$,F | B |
| 0,A,✓ | 2,C,✓ | 1,A,✓ | 5,E | 2,C,✓ | 8,E | E |
| 0,A,✓ | 2,C,✓ | 1,A,✓ | 5,E,✓ | 2,C,✓ | 7,D | D |
| 0,A,✓ | 2,C,✓ | 1,A,✓ | 5,E,✓ | 2,C,✓ | 7,D,✓ | F |



The shortest path from A to F is obtained (in the reversed order) by reading out `p[w]` 's, starting from F:

$A \to C \to E \to D \to F$.

Every iteration of the algorithm corresponds to one row in the table and each such row shows the content of the three arrays `d[-]`, `p[-]` and `f[-]`. (Check marks denote finished vertices.)

In the graph, the two circles adjacent to a vertex mark the current state of `d[w]` and `p[w]`. They turn blue whenever the vertex is marked as finished.

# Dijkstra's time complexity (adjacency matrix)

$n =$ the number of vertices, $m =$ the total number of edges.

We do the following *up to $n$* times:

a. Mark `w` as finished.

b. Update every neighbour of `w`.

c. Find `w` which is unfinished and with the smallest `d[w]`.

Representing the graph by an *adjacency matrix*, means that, over all $n$ outer loops, it takes:

- $O(n)$ to do step a
- $O(n^2)$ to do step b
- $O(n^2)$ to do step c by going through all vertices.

$\Longrightarrow$ The time complexity is $O(n^2)$.

# Dijkstra's time complexity (adjacency lists)

We do the following *up to n*-times:

a. Mark `w` as finished.

b. Update every neighbour of `w`.

c. Find `w` which is unfinished and with the smallest `d[w]`.

With *adjacency lists*, executions of step b. will (in total) update

neighbours of the 1st selected `w`,
neighbours of the 2nd selected `w`,
neighbours of the 3nd selected `w`,
...

Over all iterations combined we update $m$-many times $\Rightarrow O(m)$

Representing the graph by an *adjacency list*, means that, over all $n$ outer loops, it takes:

- $O(n)$ to do step a
- $O(m)$ to do step b
- $O(n^2)$ to do step c by going through all vertices.

$\Longrightarrow$ The time complexity is $O(n^2)$ (Note: $m \leq n^2$ in a simple graph)

# Dijkstra’s time complexity (adjacency lists)

### Speeding up step c

Use min-priority queue: The priority of `u` is `d[u]`.

- Initialise the queue by inserting all nodes into it
- Call `deleteMin` to find the unfinished node with smallest `d[w]`
  - once per iteration, i.e. up to $n$ times in total
- Whenever `d[u]` changes, we `update` the priority of `u`.

$\Longrightarrow$ total time complexity of step c

$= O(n\times\text{“cost of deleteMin”} + m\times\text{“cost of update”})$

- Using Binary Heap: $O(n\log n + m\log n)$
- Using Fibonacci Heap: $O(n\log n + m)$

What is omitted in the analysis is the time complexity of initialising the heap. This is usually done by `heapify` and its time complexity was always $O(n)$ for all heaps we had. Alternatively, we can do `insert` $n$-times which will result in the time complexity $O(n \log n)$ or $O(n)$ depending on the heap that we are using. Either way, the initialisation will not play any role in the total time complexity.

# Dijkstra's time complexity – comparison

| Adjacency matrices | Adjacency lists | |
|---|---|---|
| | Binary Heaps | Fibonacci Heaps |
| $O(n^2)$ | $O((n+m)\log n)$ | $O((n\log n)+m)$ |

Min-priority queues:

- Binary heaps: both `update` and `deleteMin` are in $O(\log n)$.
- Fibonacci heaps: `update` is in $O(1)$ and `deleteMin` is in $O(\log n)$ (both amortized).

**Remark:** Dijkstra's algorithm works only if all weights are $\geq 0$.

**Remark:** If the graph is *dense*, that is if the number of edges, $m$, is approximately $n^2$, then using adjacency lists together with binary heaps has the time complexity $O((n + n^2) \log n) = O(n^2 \log n)$ which is slower than just using adjacency matrices. This problem disappears when using Fibonacci heaps where, for dense graphs, the time complexity becomes $O(n \log n + n^2) = O(n^2)$.

On the other hand, if the graph is not dense, using adjacency lists with Binary Heaps or Fibonacci Heaps is faster than using adjacency matrices.

# Dijkstra's algorithm (pseudocode with adjacency matrix)

```
dijkstra_with_matrix(int[][] G, int v, int z) {
    n = G.length;
    d = new int[n]; p = new int[n]; f = new bool[n];

    for (int w = 0; w < n; w++) {
        d[w] = infty;   p[w] = w;   f[w] = false;
    }
    d[v] = 0;

    while (true) {
        w = min_unfinished(d, f);
        if (w == -1)
            break;

        for (int u = 0; u < n; u++)
            update(w, u, d, p);

        f[w] = true;
    }
    // compute results in desired form
    return compute_result(v, z, G, d, p);
}
```

```
int min_unfinished(int[] d, bool[] f) {
    int min = infty;
    int idx = -1;

    for (int i=0; i < d.length; i++) {
        if ( (not f[i]) && d[i] < min) {
            idx = i;
            min = d[i]
        }
    }

    return idx;
}
```

```
void update(w, u, G, d, p) {
    if (d[w] + G[w][u] < d[u]) {
        d[u] = d[w] + G[w][u];
        p[u] = w;
    }
}
```

# Dijkstra's algorithm (pseudocode with adjacency lists)

```
dijkstra_with_lists(List<Edge>[] N, int v, int z) {
    n = G.length;
    d = new int[n];   p = new int[n];
    Q = new MinPriorityQueue();

    for (int w = 0; w < n; w++) {
        d[w] = infty;   p[w] = w;
        Q.add(w, d[w]);
    }
    d[v] = 0;
    Q.update(v, 0);

    while (Q.notEmpty()) {
        w = Q.deleteMin()

        for (Edge e : N[w]) { // iterate over edges to neighbours
            u = e.target;
            if (d[w] + e.weight < d[u]) { // should we update?
                d[u] = d[w] + e.weight;
                p[u] = w;
                Q.update(u, d[u]);
            }
        }
    }
    return compute_result(v, z, G, d, p);
}
```

```
class Edge {
  // target node
  int target;

  int weight;
}
```

The initialisation happens on lines 6–9.

Lines 10–11 make sure that the first selected `w` will be `v`.

We use the class `Edge` to store neighbours together with the weight of the edge that connects them. For example, if the vertex `A` has neighbours B, C and D with the edge $A \to B$ of weight 3, $A \to C$ of weight 1, and $A \to D$ of weight 8, then we will have that the linked list `N[v]` stores `Edge(B, 3)`, `Edge(C, 1)` and `Edge(D, 8)`.

# Minimal Spanning Trees and Jarník-Prim's Algorithm

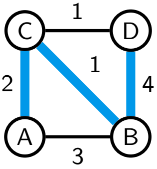
# Minimal spanning tree

Assumption: Consider only *undirected* and *connected* graphs!

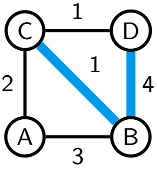
A **spanning tree** is a minimal possible selection of edges that connects all vertices. (That is, a spanning tree does not contain any cycles.)

**Minimum spanning tree** is a spanning tree such that the sum of weights of its edges is the minimal such.

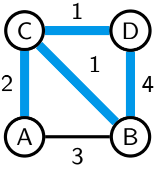
## Example



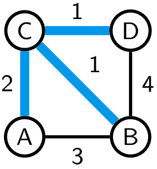
Spanning tree.



Not a spanning tree. A is not connected.



Not a spanning tree. Any of the edges CB, CD, BD could be removed.

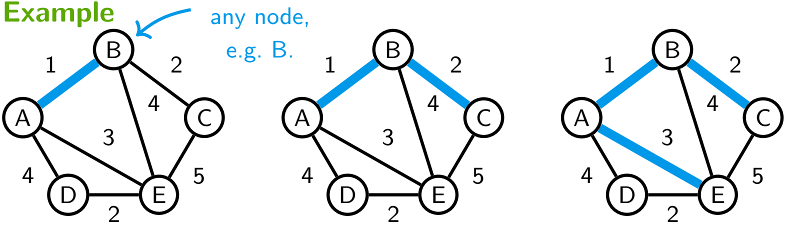


Minimum spanning tree!

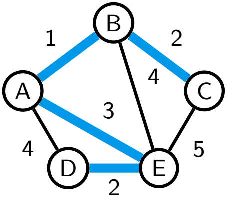
# Example: Execution of Jarník-Prim algorithm

**Idea:** Iteratively extend the tree with an edge which has the smallest weight and which connects a yet unconnected node.

**Example**



Start from any node, e.g. B.



The minimal spanning tree consists of edges: AB, BC, AE, ED.

# **Jarník-Prim algorithm** for finding the minimal spanning tree

For each vertex `w` of the graph, we keep track of the following:

i. `d[w] =` the current distance from the *tree* (initially: $\infty$)
ii. `p[w] =` the vertex which connects to the tree (initially: `w`)
iii. `f[w] =` has `w` been added to the tree? (initially: `false`)

The algorithm (idea):

```
set d[0] = 0          (vertex 0 could be replaced by any other vertex)
while there are unfinished vertices:
    set w = the yet unfinished vertex with the smallest d[w]
    set f[w] = true                          (i.e. mark w as finished)
    for every neighbour u of w:
        if weight(w,u) < d[u]:
            set d[u] = weight(w,u) and p[u] = w
```

(Where `weight(w,u)` is the weight of the edge `w` → `u`)

Jarník-Prim's algorithm works similarly to Dijkstra's algorithm. The differences are marked by red. Altough the principle is similar, the interpretation of the execution and the result is different. The main idea of Jarník-Prim is that we are growing a minimal spanning tree iteratively

The following is an invariant for Jarník-Prim:

1. The vertices marked as finished are connected/added to the tree.
2. For those vertices which are not connected yet, `d[w]` denotes the smallest weight of an edge that connects `w` to the tree.
3. `p[w]` denotes the vertex of the tree such that the edge between `w` and `p[w]` is the edge with weight `d[w]`.

In steps 5-7, we reduce the weights (from $\infty$) of the nodes that connect to last added finished node, so that they can be considered for adding in the next iteration.

After the algorithm finishes, i.e. all vertices are marked finished, we can read out the spanning tree from the array `p[-]`. To obtain the minimum spanning tree, for every vertex `w` (except for `w == 0`), add the edge `w`—`p[w]`.

# Jarník-Prim's time complexity

The time complexity is the same as for Dijkstra's algorithm!

| Adjacency matrices | Adjacency lists | |
|---|---|---|
| | Binary Heaps | Fibonacci Heaps |
| $O(n^2)$ | $O((n+m)\log n)$ | $O((n\log n)+m)$ |

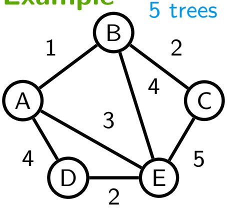
Remark: Unlike Dijkstra's algorithm, Jarník-Prim's algorithm would also work for graphs with edges that have negative weights.

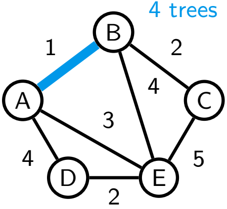
# Minimal Spanning Forests and Kruskal's Algorithm

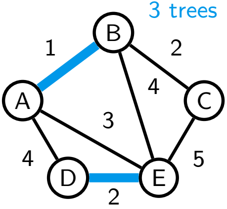
# Example: Execution of Kruskal's algorithm

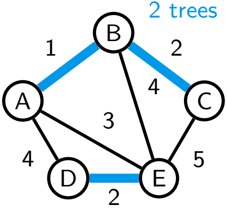
**Idea:** Starting with a forest where every node in the graph is a separate tree, greedily add edges with minimum weights such that each edge is between different trees.

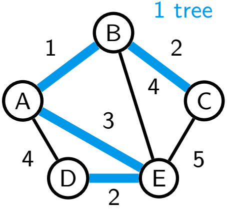
**Example**











The minimal spanning tree consists of edges: AB, BC, AE, DE.

- A *Forest* is a set of trees
- If the graph is *connected* then this creates a minimal spanning tree. Otherwise it creates a *minimal spanning forest*: a set of minimal spanning trees, one for each connected component of the graph
- The algorithm is greedy edge-based because it always chooses the best (i.e. minimal) edge to add at each step
- It finishes when no more edges can be added
    - Note: unless the graph is connected, this is **NOT** the same as finishing when there is only one tree
- Good performance is dependent on having an efficient way to
    1. Discover if an edge is between nodes in the same tree
    2. If an edge is between 2 different trees, which 2 trees these are

  For this purpose, we can use a **Union-Find** data structure

# Union-Find

The *Union-Find* ADT has 3 operations:

- `makeSet(x)`: Make a new set containing the element x
- `find(x)`: Find the set that x is an element of. It returns a particular element of the set that is used to identify the set
- `union(x,y)`: merge the two sets that contain elements x and y. This replaces the two pre-existing sets containing x and y by a single new set which contains the union of the two sets

In Kruskal: each tree is represented by the set of nodes in the tree:

- `makeSet(n)` is called on each node of the graph at the start
- To test if an edge (u,v) is between nodes in the same or different trees, call `find(u)` and `find(v)` and compare their results.
- When an edge (u,v) is added, `merge(find(u),find(v))` is called to merge the two trees joined by the edge

## Union-Find Implementation

There are pointer based implementations, but we will use a simple array based implementation, where we assume that the graph nodes are numbered $0, 1, \ldots, n-1$.

Sets will be represented by trees, where a `parent` array, indexed by the node number, identifies the parent of each node in the set, and the root of the tree is the set representative that identifies the set. The root of each tree has itself as its own parent

To find the set that a node is in, we recursively follow the chain of parents up to the root of the tree

To merge two sets, you find the roots of the two trees and set one to have the other as its parent

A good explanation of this structure can be found at: `https://cp-algorithms.com/data_structures/disjoint_set_union.html`. The code in these slides is based on that.

# Union-Find Naïve Implementation

```
void makeSet(int v) {
    parent[v] = v;
}

int find(int v) {
    if (v == parent[v])
        return v;
    return find(parent[v]);
}

void union(int a, int b) {
    a = find(a);
    b = find(b);
    if (a != b)
        parent[b] = a;
}
```

# Naïve Union-Find Problem 1

Unions can end up making very deep chains, i.e. the trees that represent sets can end up very deep and narrow, or even linear. This makes `find(x)` require a nearly linear search up a long chain to find the root.

This can be fixed by changing `find(x)` so that, on return from the recursive calls to find the root, it sets the parent of all nodes on the path just traversed to be the root, thus reducing the `find(x)` cost for all future calls on any of those nodes

# Naïve Union-Find Problem 2

The previous code for `union(u, v)` just makes the second tree be a child of the root of the first tree. If the second tree is deeper than the first, than this makes the resulting tree deeper still, again reducing performance of `find(x)`

Here we just need to choose the deeper tree to be the one that we add the shallower tree to. Actually, it works out that you can alternatively choose to make the smaller tree the child of the deeper one, and in practice you get the same improved performance

In either case, you need an extra array to record either (an approximation to) the rank (level) of the tree rooted at each node, or the size of the subtree rooted at each node.

One subtlety is that it is not necessary to maintain the depth/size of the subtree at *every* node, just that of the *root* nodes.

# Improved Union-Find (based on size)

```
void makeSet(int v) {
    parent[v] = v;
    size[v] = 1;
}

int find(int v) {
    if (v == parent[v])
        return v;
    return parent[v] = find(parent[v]);
}

void union(int a, int b) {
    a = find(a);
    b = find(b);
    if (a != b) {
        if (size[a] < size[b])
            swap(a, b);
        parent[b] = a;
        size[a] += size[b]
    }
}
```

The improved algorthm that is based on size does not need to do anything extra in the `find(x)` method as even the updates done there to shorten the paths from a node to its root still leaves the node in the same tree and hence does not change the size of the tree.

# Improved Union-Find (based on rank)

```
void makeSet(int v) {
    parent[v] = v;
    rank[v] = 0;
}

int find(int v) {
    if (v == parent[v])
        return v;
    return parent[v] = find(parent[v]);
}

void union(int a, int b) {
    a = find(a);
    b = find(b);
    if (a != b) {
        if (rank[a] < rank[b])
            swap(a, b);
        parent[b] = a;
        if (rank[a] == rank[b])
            rank[a]++
    }
}
```

The improved alorithm that is based on rank can not efficiently reduce the rank of the root correctly during a call to *find*$(x)$ that reduces path lengths because that would require paths of from all leaves to the root and not just the one that has been traversed.

However, it is sufficient to use the heuristic of tracking an upperbound of the depth, where no change to the rank is made during a call to `find(x)`.

# Union-Find performance

Without the two optimisations, the trees involved can have height O(n), so that `find(x)` and `union(x, y)` both have O(n) complexity

With both the above optimisations the amortised (over many operations) complexity of each operation will be $\Theta(\alpha(n))$

$\alpha(n)$ is the reverse Ackermann function which is extremely slow growing and will not exceed 4 for any realistic value of $n$.

Hence the amortised upperbound complexity for all operations is effectively $O(1)$

# Kruskal's Algorithm

Now that we have an efficient Union-Find ADT, we can use it in Kruskal's algorithm:

```
let result be a new empty list of edges
for each node n in G
  makeSet(n)
let E be a list of edges in G sorted by increasing weights
for each edge e = (u,v) in E in order
{
  if (find(u) != find(v))
  {
    result.add(e)
    union(find(u), find(v))
  }
}
return result
```

# Kruskal's Algorithm Complexity

This algorithm requires sorting the graph edges by weight, which is $O(e \log e)$, where $e$ is the number of edges.

Then it does a linear search through the edges, during which it carries out only operations of $O(1)$. This contributes $O(e)$

Hence its final complexity is $O(e \log e + e) = O(e \log e)$

$e$ is at most $n^2$ where $n$ is the number of nodes in the graph.

So $O(e \log e) = O(e \log n^2) = O(2e \log n) = O(e \log n)$

# Kruskal's Algorithm vs Jarník-Prim algorithm

If the graph is sparse, i.e. $e \approx n$, then

- Fib Adj List Jarník-Prim: $O(n \log n + n) = O(n \log n)$
- Adj Matrix Jarník-Prim: $O(n^2)$
- Kruskal: $O(n \log n)$

If the graph is dense, i.e. $e \approx n^2$, then

- Fib Adj List Jarník-Prim: $O(n \log n + n^2) = O(n^2)$
- Adj Matrix Jarník-Prim: $O(n^2)$
- Kruskal: $O(n^2 \log n)$

Finally, note that Kruskal's algorithm computes a spanning forest, if the graph is not connected, or a spanning tree if it is connected, while Jarník-Prim can only calculate a spanning tree on a connected graph.