Data Structures & Algorithms

Solutions

Sample Exam Questions

Data Structures & Algorithms

1. One can, inefficiently, implement a queue using two stacks, without using any further data structures such as arrays, linked lists etc.

   (a) Explain the basic approach to implementing a queue this way. **[5 marks]**

   (b) Write the pseudocode for the enqueue function. **[5 marks]**

   (c) Write the pseudocode for the dequeue function. **[5 marks]**

   (d) Analyse, in terms of $O(g(n))$, the complexity of each operation and explain, briefly, your reasoning. **[5 marks]**

   Topics: 02 - Stacks and Queues, 03 - Efficiency and Complexity

2. The following is the pseudocode for a recursive version of a function to calculate the size of a binary tree.

```
size(t)
{
    if ( isEmpty(t) )
        return 0
    else
        return (1 + size(left(t)) + size(right(t)))
}
```

   Write a non-recursive pseudocode version using a stack to keep track of the work still to be done. **[20 marks]**

   Topics: 04 - Trees

3. Construct a **minimal** AVL tree of height 5 of positive integers with no duplicate values allowed.

   (a) Draw that AVL tree. **[5 marks]**

   (b) Identify a value that on insertion into the tree would not trigger any rotation. **[5 marks]**

   (c) Identify a value that on insertion into the original tree that would trigger precisely one rotation. **[5 marks]**

   (d) Identify an insertion into the original tree that would trigger precisely one double rotation, i.e. a right rotation followed by a left rotation or a left rotation followed by a right rotation. **[5 marks]**

   Topics: 05 - AVL-Trees

4. For each of the following functions to calculate the largest element of an array, work out the formula for the number of operations (assignments, comparisons and arithmetic operations) executed as a function of the size of the array, n, state which complexity class in big O notation the function belongs to and justify your answer with a short explanation. Assume that the sort algorithm used in (c) takes $O(n \log n)$ operations.

(a)
```
int largest1(int[] arr) {
  int n = arr.length;
  int max = 0;
  for (int i=0; i<n; i++) {
    bool largest = true;
    for (int j=0; j<n; j++) {
      if (arr[i] < arr[j])
        largest = false;
    }
    if (largest)
      max = arr[i];
  }
  return max;
}
```

[7 marks]

(b)
```
int largest2(int[] arr) {
  int n = arr.length;
  int max = 0;
  if (arr.length == 0) {
    return 0;
  } else {
    max = arr[0];
    for (int i=0; i<n; i++) {
      if (arr[i] > max)
        max = arr[i];
    }
    return max;
  }
}
```

[7 marks]

(c)
```
int largest3(int[] arr) {
  sort(arr);
  if (arr.length == 0) {
    return 0;
  } else {
    int last = arr[arr.length - 1];
    return last;
  }
}
```

[6 marks]

Topics: : 03 - Efficiency and Complexity

5. Consider a binary search tree used to store integers with methods `isEmpty(t)`, `left(t)`, `right(t)` and `root(t)`, to return whether the tree `t`, is empty, the left child tree, the right child tree and the integer value stored at the root respectively.

   - Write a **recursive** function `sum_rec(tree)` to calculate and return the sum of all integers stored in the tree. **[8 marks]**
   - A `Queue` ADT has, for a queue `q`, methods `q.enqueue(val)` and `q.dequeue()` to enqueue a value val in the queue and to dequeue and return a value from the queue respectively. It also has a constructor which can be invoked with the commmand `new Queue()` to create and return a new empty queue.

     Write the pseudocode for a **non-recursive** function `sum_nonrec(tree)` using the `Queue` ADT, to calculate and return the sum of all integers stored in the tree. **[12 marks]**

   Topics: 02 - Stacks and Queues, 04 - Trees

Model answer

1.

(a) Have two stacks, `in` and `out`. In order to `enqueue` a value, iterate, popping each element off the `out` stack and pushing it onto the `in` stack until the `out` stack is empty. Then push the new value onto the `in` stack. When you want to `dequeue` a value, iterate, popping each element off the `in` stack and pushing it onto the `out` stack until the `in` stack is empty. Then obtain the value to be dequeued by popping it off the `out` stack.

(b) [Note: any reasonable pseudocode ADT for stacks can be used so long as it is used consistently. Here I assume an object oriented ADT such as might be written in Java, but you can use the ADT in the handouts if you prefer.]

```
enqueue(val)
{
  while (not out.isEmpty())
    in.push(out.pop())
  in.push(val)
}
```

(c)

```
dequeue()
{
  while (not in.isEmpty())
    out.push(in.pop())
  if (out.isEmpty())
    throw QueueEmptyException("Tried to dequeue from an empty queue")
  return out.pop()
}
```

(d) Here the cost of enqueue and dequeue are both $O(n)$ in the worst case, as you may have to transfer all elements from one stack onto the other to complete each operation. You can make it $O(1)$ for one of the operations by making one operation do the transfer, carry out its operation (enqueue or dequeue) and transfer back, leaving the other operation not needing to do any transfers between stacks. However, the amortised cost of enqueue and dequeue over a random sequence of such operations would remain $O(n)$

2. [There are multiple different solutions. This one allows pushing empty (sub-)trees onto the stack.]

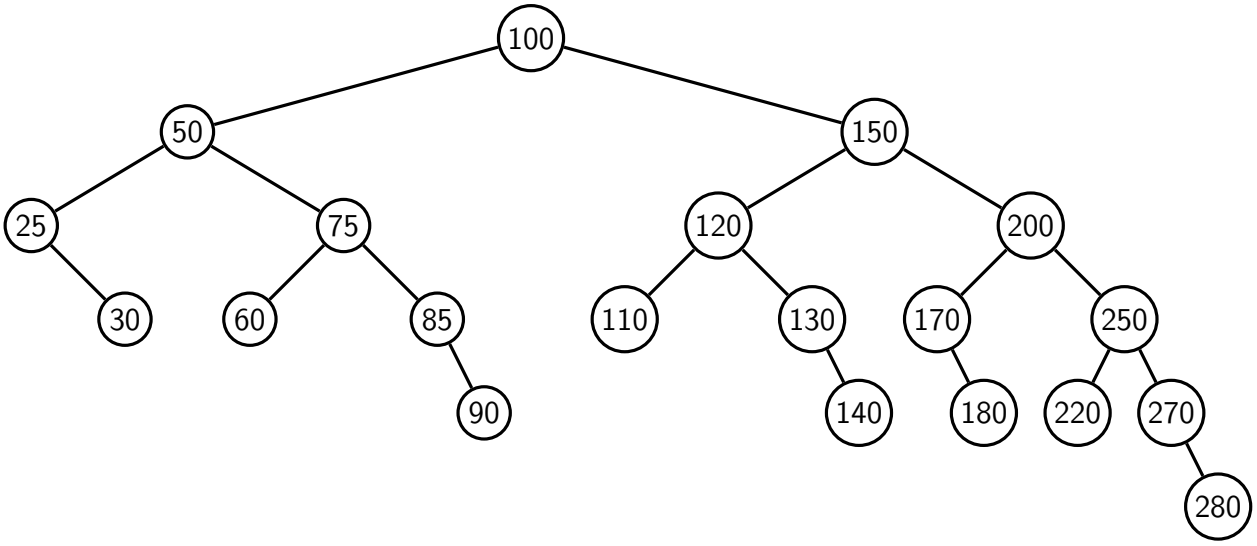
```
size(t)
{
    Stack todo = new Stack()
    todo.push(t)
    size = 0

    while (not todo.isEmpty())
    {
       tree = todo.pop()
       if (not tree.isEmpty()
       {
          size++
          todo.push(tree.left())
          todo.push(tree.right())
       }
    }

    return size
}
```

[This version only pushes non-empty trees on the stack:]

```
size(t)
{
    if (t.isEmpty())
        return 0

    Stack todo = new Stack()
    todo.push(t)
    size = 0

    while (not todo.isEmpty())
    {
       tree = todo.pop()
       size++
       if (not tree.left().isEmpty())
          todo.push(tree.left())
       if (not tree.right().isEmpty())
          todo.push(tree.right())
    }

    return size
}
```

3.

(a) [This is a minimal AVL tree of height 5. You can switch the left and right children of any node and it remains a minimal AVL tree (so long as you also correct the values in the nodes so that it observes the Binary Search Tree ordering requirements).]



(b) Inserting the value 20 would not cause any rotation because it would be entered as a new left leaf of the node with value 25, and not cause any imbalance higher up the tree.

(c) Inserting the value 95 would cause a single rotation. The new leaf would be inserted as the right child of the node with value 90, causing a -2 imbalance at the node containing 85. This corresponds to a RR case which would be fixed with a left rotation at 85.

(d) Inserting the value 87 would cause a double rotation. The new leaf would be inserted as the left child of the node with value 90, causing a -2 imbalance at the node containing 85. This corresponds to a RL case which would be fixed with a right rotation at 90 followed by a left rotation at 85.

4. Justification should analyse the steps and the loops in the code

(a) `largest1` : $O(n^2)$

(a) `largest2` : $O(n)$

(a) `largest3` : The dominant complexity will be the sort, so $O(n \log n)$ by default but other complexities will be accepted if a sort with a different complexity is used, or simply saying the complexity is that of the sort.

5. (a)

```
sum_rec(t)
{
  if (isEmpty(t) )
    return 0
  else
    return sum_rec(left(t)) + root(t) + sum_rec(right(t))
}
```

(b)

```
sum_nonrec(t)
{
  Queue todo = new Queue()
  todo.enqueue(t)
  sum = 0

  while ( not todo.isEmpty() )
  {
    tree = todo.dequeue()
    if ( not tree.isEmpty() )
    {
       sum += root(tree)
       todo.enqueue(tree.left())
       todo.enqueue(tree.right())
    }
  }
  return sum
}
```