STUDENT NAME: _____

STUDENT ID: _____

McGill University
Faculty of Science
School of Computer Science

FINAL EXAMINATION

COMP-250: Introduction to Computer Science - Winter 2006

April 25, 2006
2:00-5:00

Examiner: Prof. Doina Precup

Sample Solutions

1. [30 points, 3 points each] **Short questions**

   Indicate whether the following statements are true or false. Give a two-line justification for each. Credit will be given only if the justification is correct.

   (a) $2^{\log_{10} n}$ is $O(n)$

   Solution: True, using the logarithm conversion rule we have:

   $$\log_{10}(n) = \log_1 0(2) \log_2(n)$$

   So we have:

   $$2^{\log_{10} n} = 2^{\log_1 0(2) \log_2(n)} = n^{\log_1 0(2)} < n^{\log_1 0(10)} = n$$

   So $2^{\log_{10} n}$ is (On).

   (b) All sorting algorithms have a worst-case running time of $O(n \log n)$.

   Solution: False (selection sort, for example, does not).

   (c) Suppose that some algorithm A has running time $O(n \log n)$ and algorithm B has running time $O(n^2)$. Then A is always preferred to B.

   Solution: False (see 2008 final solution).

   (d) Problems in the class P are also part of the class NP.

   Solution: True. NP means that a solution has to be guessed and then can be checked in polynomial time. For any P problem, we can simply solve it and use the solution as the guess.

   (e) Suppose I have a hash table with 50 buckets. Then I cannot store more than 50 items in this hash table.

   Solution: False. Each bucket is a container (list, tree, another hash table) which can hold more than one element.

   (f) $n \cos n$ is not $O(n)$.

   Solution: False. $\cos n \leq 1$ so $n \cos n \leq n$ and we can prove the statement using the definition of $O()$.

   (g) Quicksort's worst case input is when the array is already sorted.

   Solution: True (then the pivot will create one empty partition, and all other elements will be in the other partition).

   (h) An algorithm that does not contain recursive calls always terminates.

   Solution: False. E.g: **while true do** print("Hi") will loop forever.

   (i) Removing an desired element from a linked list takes $O(n)$ running time, where n is the number of elements in the list.

   Solution: True (you have to find the element, and it may be all the way at the end; see also list lectures).

   (j) Queues are used to in web servers instead of stacks because they ensure that all customers are treated equally.

   Solution: True (you want the first arriving job to be served first).

2. [10 points] **A simple algorithm**

   You are given an array of n positive integers. The numbers do not appear in any particular order. Write an algorithm that will move all the numbers that are multiples of 13 (if any) to the front of the array. You algorithm should have running time $O(n)$.

   For example, your initial array might be: 2 13 4 26 5 1 2 After the rearrangement, the array should have 13 and 26 in the first two positions. The order does not matter.

   Solution: We will use one index, i, for traversing the array and another index, j, to remember where to write. I am writing this in Java, just for variety:

```
public class MyProblem {
    public static void MyMethod(int[] a, int k) {
        int j=0;
        for (int i=0; i<a.length; i++)
            if (a[i]%k==0) {
                int temp = a[i];
                a[i] = a[j];
                a[j]=temp;
                j++;
            }
    }
    public static void main(String args[]) {
        int[] a = {2,13,4,26,5,1,2};
        for (int i = 0; i< a.length; i++)
            System.out.print(a[i]+' ');
        System.out.println();
        MyMethod(a,13);
        System.out.println("After processing:");
        for (int i = 0; i < a.length; i++)
            System.out.print(a[i]+' ');
        System.out.println();
    }
}
```

3. [10 points] **Merging in triplets**

Recall that the merge sort algorithm proceeds by recursively splitting an array of n numbers, sorting each part recursively, and then merging the resulting array. The pseudocode of the algorithm is as follows:

Algorithm mergeSort(a,p,q)
Input: An array a and two indices p and q between which we want to do the sorting.
Output: The array a will be sorted between p and q as a side effect
if $p < q$ **then**
  int $m \leftarrow \frac{\lfloor p+q \rfloor}{2}$ //this is the middle of the part of interest
  mergeSort(a,p,m)
  mergeSort(a,$m+1$,q)
  merge(a,p,m,q)

Algorithm merge(a,p,m,q)
Input: An array a, in which we assume that the halves from $p \ldots m$ and $m+1 \ldots q$ are each sorted
Output: The array should be sorted between p and q
Array tmp // this is an array of size $q-p+1$ (same as a) which will hold the temporary result
int $i \leftarrow p$ //these are the two indices in the two halves of the array
int $j \leftarrow m+1$
int $k \leftarrow 1$ //this is the index we will use in the tmp array
while ($i \leq m$ **or** $j \leq q$) **do**
  if ($j = q+1$ **or** $a[i] \leq a[j]$ **then**
    $tmp[k] \leftarrow a[i]$
    $i \leftarrow i+1$
  else
    $tmp[k] \leftarrow a[j]$
    $j \leftarrow j+1$
  $k \leftarrow k+1$
for $k = p$ **to** q **do**
  $a[k] \leftarrow tmp[k]$

Now suppose that you want to cut the array in 3 parts instead of 2. Write the pseudocode for the mergeSort and merge functions in this case. Does this make a difference in terms of the big-oh of the algorithm? Justify your answer.

Solution: The basic idea is to keep two indices, $m1$ and $m2$, which mark $1/3$ and $2/3$ of the array. Otherwise, nothing really changes. The new pseudocode is as follows:
Algorithm mergeSort(a,p,q)
Input: An array a and two indices p and q between which we want to do the sorting.
Output: The array a will be sorted between p and q as a side effect
if $p < q$ **then**
  int $m1 \leftarrow \lfloor \frac{p+q}{3} \rfloor$ //this is the first third of the part of interest
  int $m2 \leftarrow \lfloor \frac{(p+q)*2}{3} \rfloor$ //this is the first third of the part of interest
  mergeSort(a,p,$m1$)
  mergeSort(a,$m1$,$m2$)
  mergeSort(a,$m2$,q)

merge(a,p,$m1$,$m2$)
merge(a,p,$m2$,q)

Note that merge does not need to change. For the $O()$, let us write the recurrence:

$$T(n) = 3T\left(\frac{n}{3}\right) + c\frac{2}{3}n + cn$$

where the last 2 terms are from the calls to merge. Note that this is a bit wasteful, because we go over the first $2/3$ of the array twice. But it saves us the re-writing of merge (otherwise, we could just do a 3-way merge). Now we expand the recurrence (analogously to the lecture notes):

$$\begin{aligned} T(n) &= 3T\left(\frac{n}{3}\right) + c\frac{5}{3}n \\ &= 3\left(3T\left(\frac{n}{3^2}\right) + c\frac{5}{3}\frac{n}{3}\right) + c\frac{5}{3}n \\ &= 3\left(3T\left(\frac{n}{3^2}\right) + c\frac{5}{3}\frac{n}{3}\right) + c\frac{5}{3}n \\ &= 3^2T\left(\frac{n}{3^2}\right) + c\frac{5}{3}(2n) \\ &= \dots \\ &= 3^kT\left(\frac{n}{3^k}\right) + c\frac{5}{3}(kn) \end{aligned}$$

The recursion will hit the base case when $k = \log_3(n)$, so we get:

$$T(n) = nc + c\frac{5}{3}n\log_3 n$$

which is still $O(n\log n)$.

4. [10 points] **Tree node counting**

Suppose you have a binary tree. Write an algorithm in pseudocode that counts the number of nodes with exactly one child. State the big-oh of your algorithm as a function of the number of nodes in the tree, N.

Solution: We will do this recursively.

Algorithm CountOneChild (TreeNode n)
Input: A TreeNode n
Output: The number of nodes, in the subtree rooted at n, that have exactly one child

if ((n.left=null) **and** (n.right=null)) **then return** 0;
if ((n.left=null) **and** (n.right≠null)) **then return** 1+CountOneChild(n.right);
if ((n.left≠null) **and** (n.right=null)) **then return** 1+CountOneChild(n.left);
return CountOneChild(n.right) + CountOneChild(n.left);

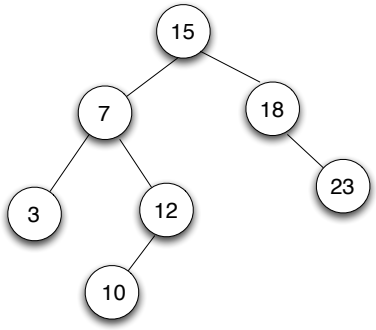
For the $O()$, the intuitive way to see the answer is that we will need to check all the nodes in the tree, so this will be $O(N)$. For the recurrence, the worst-case is the last line, in which case we have:

$$T(n) = c + T(k) + T(n-k)$$

Check the solution to the 2008 final for the solution.

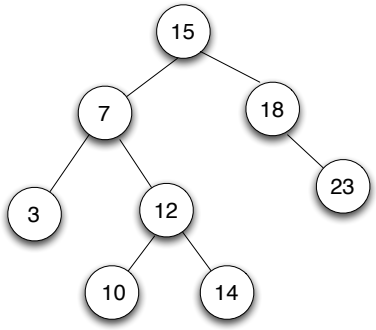
5. [5 points] **Binary search trees**

   Consider the following binary search tree:



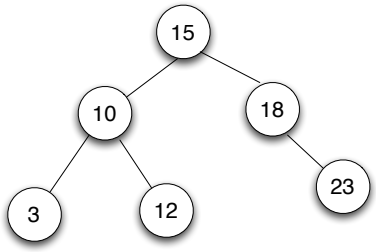
   (a) [2 points] Draw the binary search tree after an element with key 14 has been inserted

   Solution:



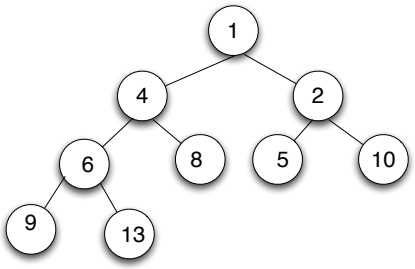
   (b) [3 points] Draw the binary search tree after remove(7) has been executed. Start from the original tree, not the tree you got in (a)

   Solution:

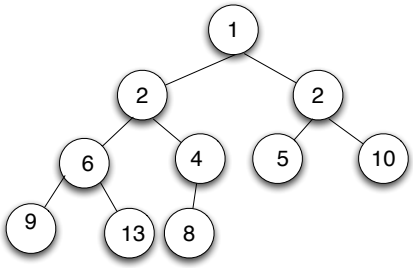


6. [5 points] **Heaps**

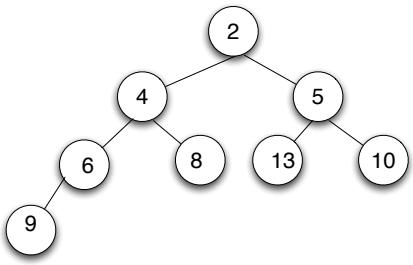
   Consider the following heap:



(a) [2 points] Draw the heap after the element with key 2 has been inserted.
Solution:

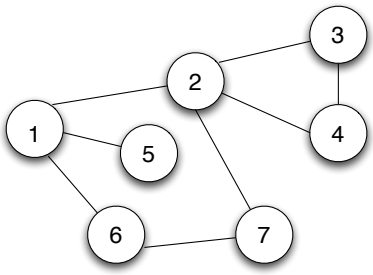


(b) [3 points] Draw the heap after removeMin() has been executed. Start from the original heap, not the one you got in (a)
Solution:



7. [10 points] **Search in graphs**

Consider the graph in the figure below.



(a) [4 points] Suppose that you are using breadth-first search to find a path from vertex 1 to vertex 4. Show the state of the queue after each node expansion, assuming nodes are enqueued in increasing order. What path do you find?

Solution:

First enqueue 1, so the queue is: 1 (leftmost element is the head)
Dequeue 1 and enqueue its successors: 2,5,6
Dequeue 2 and enqueue its successors: 5.6,3,4,7
Dequeue 5; its successor (1) is already visited, so the queue becomes: 6,3,4,7
Dequeue 6 and enqueue its successors: 3,4,7,7 (the first 7 was enqueued as 2's successor, the second as 6's successor)
Dequeue 3 and enqueue 4: 4,7,7,4
Dequeue 4: This gives the path 1-2-4.

(b) [4 points] Suppose that you are using depth-first search for the same task. Show the state of the stack after each node expansion, assuming that successors of the same node are pushed in increasing order. What path do you find?

Solution:

Push 1
Pop 1, push 2,5,6, so the stack is: 6,5,2 (leftmost element is the top of the stack)
Pop 6, push 7: 7,5,2
Pop 7, push 2: 2,5,2
Pop 2, push 3, push 4: 4,3,5,2
Pop 4; This gives the path 1-2-4.

(c) [2 points] Based on this example, what is the main advantage of depth-first search? What is the main advantage of breadth-first search?

Solution: See final of 2008.

8. [10 points] **Which data structure to pick?**

You have been hired by a large computer company as a summer intern. They have a huge database of customers, containing the customer's user name, their age group (under 20, 20-29, 30-39, 40-49, 50-59, more than 60), and the songs they have downloaded so far, along with the song's genre (one of 20 categories like classic rock, jazz etc). The company believes that customers in the same age group have similar preferences. They want to set up a new feature which will propose to new customers songs that they are likely to buy. The feature should display 5 songs at a time, and it should do this without perceivable delays. It is desirable to propose different songs every time (to the extent possible).

You are given initially a large list containing all transactions, in the format:
user-name age-group song genre

You know that using this list directly will be very slow, so you consider building a different data structure(s) based on this data. You are allowed to summarize the data you have in any way you want. Describe, in at most 7 sentences, what data structure(s) you would build, why you are making this choice, and how you would design the desired feature. No pseudocode is necessary.

Solution: The simplest idea is to have a 2D array , where for each age group, you keep something like the top 1000 most downloaded songs. Also, for each song, you will keep a count of how many times it has been downloaded by each age group (this is an array associated with the song). You can use a fast data structure to sort the songs for each age group (e.g. one heap for each age group). These are updated rarely (e.g. once per day) and the arrays that are used during queries might get updated then as well.

9. [5 points] **Proof by induction**

Recall that a binary tree is called *proper* if every node has either 2 or 0 descendents. Prove by induction (on the height of the tree) that in a proper binary tree, the number of leaves is always equal to the number of internal nodes + 1.

Solution:

Base case: $d = 0$ means that we have 1 leaf and no internal nodes, so the statement holds.

Induction step: Consider a tree of depth $d+1$. Cut out the root. This leaves 2 proper subtrees of depth at mode d (se we can use the induction hypothesis on them. Let N_l, N_r be the numbers of internal nodes on the

left and right respectively, and let L_l, L_r be the numbers of leaves in the left and right sub-trees. If N and L are the number for the whole trre, we have:

$$N = N_l + N_r + 1 \text{ and } L = L_l + L_r$$

By the induction hypothesis, we have:

$$L_l = N_l + 1 \text{ and } L_r = N_r + 1$$

Putting it all together, we get:

$$L = L_l + L_r = N_l + 1 + N_r + 1 = N + 1$$

which concludes the proof.

10. [5 points] **A linear-time sorting algorithm**

Suppose you are given an array of length n, containing integer elements. You know that each element is equal to 1, 2, 3, 4 or 5: $a[i] \in \{1,2,3,4,5\}, \forall i = 1 \ldots n$. Give a linear-time ($O(n)$) algorithm for sorting the array.

Solution: The idea is to make an array with 5 elements, and count how many times each value (1 through 5) occurs in the array. Then we can just re-write the array putting the appropriate number of 1s, 2s etc. This idea is easily generalizable if we know