CSE 373 – Data Structures and Algorithms
Autumn 2003.
Mid-term Exam. 11/3/2003

Student name	Student number

Question	
1	/20
2	/35
3	/15
4	/30
Total	/100

<u>Question 1 (20 points)</u>

Let $T_{bar}(n)$ define the time complexity, as a function of n, of executing bar(n), and let $T_{foo}(n)$ define the time complexity, as a function of n, of executing foo(n).

```
foo( n : integer): void {
      m: integer;
      m := n * n * n;
      bar( m );
}
```

```
bar( k: integer ):void  {
   if ( k <= 1 )
      return;
   print( "X" );
   bar( k / 2 );
}
```

Complete: $T_{bar}(n) = \Theta($ __log n__ $)$

$T_{foo}(n) = \Theta($ __log n__ $)$

Explain your answers:

bar: In each iteration we perform $\Theta(1)$ operations and make a recursive call on a problem of half-size. Therefore, after log n iterations we reach a problem of size 1 and halt.

foo: We perform $\Theta(1)$ operations and call bar(n^3). The time complexity of bar(n^3) is $\Theta(\log (n^3))$ which equals $\Theta(3 \log n) = \Theta(\log n)$.

Question 2 (35 points)

Consider the following node structure in a binary tree (not necessarily a search tree).

```
struct node {
        key: integer;
        max: integer;
        left :struct node pointer;
        right :struct node pointer;
};
```

The 'key' field of each node is a **non-negative** integer, the 'max' field is initially 0, 'left' is a pointer to the left sub-tree of the node (or NULL if there is no such sub-tree), and 'right' is a pointer to the right sub-tree of the node (or NULL if there is no such sub-tree).

a. Complete the pseudocode of the **recursive** function mark(), that gets as input a pointer to a root of a tree, and fills the 'max' fields of all the tree nodes, such that for each node v, the value of v.max is the maximal value of a key in the sub-tree rooted at v.

```
mark( v : struct node pointer): integer {
        if (___ v == null ___)   // base case
                __return 0___ ;  // (because keys are non-negative; -1 will also work)
        v. max :=
            max3( __v.key__ , __mark(v.left)__ , __mark(v.right)__ );
        __return v.max_____;
}
```

Remark: the function max3 returns the maximal value among its three parameters.

b. What is the time complexity of mark( ) as a function of the number of nodes, n, in the tree? Explain.

> Θ(n) - Like in any other tree traversal, each node is visited exactly once. The above code is actually pre-order with some additional O(1) operations for each node. (It can also be in-order or post-order if we change the order of the three parameters of max3; for any order the solution is fine and the time complexity will be the same).

c. What is the space complexity of mark( ) as a function of the number of nodes, n, in the tree? Explain.

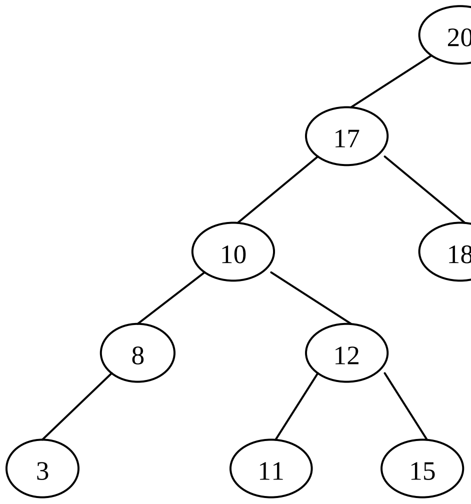
> Θ(n) – there are no local variables, but due to the recursion we need to store in the stack the details (parameters, etc.) of the currently executed functions. The space complexity is therefore Θ(depth of recursion). In the worst case, this can be Θ(n).

d. Do your answers to either of b or c change if it is known that the tree is balanced? Explain.

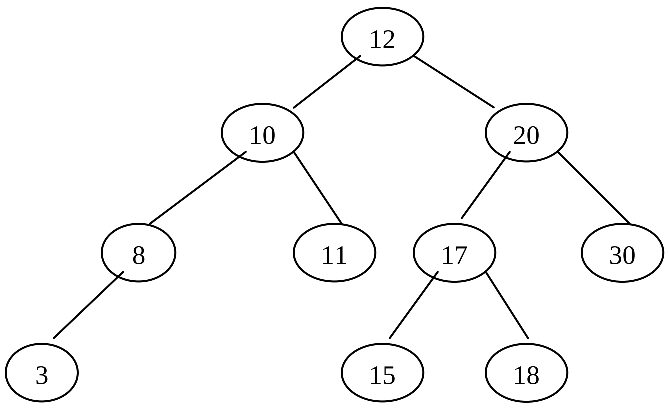
> The time complexity does not change – we still need to visit and mark each of the nodes. The space complexity is reduced to Θ(log n), since now the maximum depth of the recursion is Θ(log n) - the path from the root of the recursion tree to the currently marked node represents the currently open recursive calls at any moment.

Question 3 (15 points)

Draw the resulting **splay** tree after find(12) is performed. It is OK to draw only the final state. However, you can draw one intermediate state that will be considered if your final state is not correct. If you do provide an intermediate state please mark clearly what is the final state and what is the intermediate one.



Perform zig-zag first and then additional zig to get:
(it is **not** ok to perform zig first and then zig-zig)



<u>Question 4 (30 points)</u>

Consider a linked list in which every element has a ‘data’ field, which contains an integer, and a ‘next’ field, which points to the next element in the list.
Complete the function MidTermFunction( ) that gets as input a list, and **removes** from the list all the elements that have an odd-number data.

For example, if the list is *header -> 8 ->5 ->7 ->12 ->10 ->3* then after calling the function, the list should be *header -> 8 ->12 ->10*.

Remarks:
1. This is a destructive function; you do not need to keep the original list.
2. You need to free the space of the removed elements.
3. Assume that the list has a header, that is, p points to some dummy element, and p.next is the first element in the list.

```
MidTermFunction(p: struct node pointer) {

  temp : struct node pointer;
  prev: struct node pointer;  // a pointer to remember the previous node

  temp = p.next;          // temp starts at the first element of the list
  prev = p;               // prev starts at header of the list

  while ( temp != NULL ) {

        // check if data field is odd
        if ((temp.data mod 2) != 0) {
                // odd number, so remove the node
                temp = temp.next;
                free (prev.next);
                prev.next = temp;
        }else {
                // even number, so simply increment pointers
                temp = temp.next;
                prev = prev.next;
        }

   } // end of while loop

} // end of function
```

The logic is: If the data field is even, do nothing to the list and move on to the next node. If the data field is odd, remove that node by making the necessary changes to the link list. For this, we use two pointers – one that points to the data that is to be removed, while the other points to the previous node in the list, whose ‘next’ field needs to be adjusted.