Homework 2 Robert Quan

due March 27 at 11:59 pm to eCampus.

1. (15 points) Write a recursive function that counts the number of nodes in a singly linked list. Write a recurrence relation that represents your algorithm. Solve the relation and obtain the running time of the algorithm in Big-O

```
int numNodes(ListNode* node){
    if(node == NULL)
        return 0;
    else
        return 1+numNodes(node->next());
}
```

The Time complexity is: $O(n)$

```
f(n) = f(n-1) +1
f(n-1) = f(n-2) +1
.
.
f(1) = 1
f(n) = n = O(n)
```

2. (15 points) Write a recursive function that finds the maximum value in an array of int values without using any loops. Write a recurrence relation that represents your algorithm. Solve the relation and obtain the running time of the algorithm in Big-O.

```
int max_val(int* array,int size,int* compare = NULL){
    if(compare==NULL) compare = array;
    if(size==0) return(*compare);
    return((*array)>(*compare)) ? max_val(array++,size-1,array++) :
        max_val(array++,size-1,compare);
}
```

Complexity is: $O(n)$

```
T(n) = T(n-1) +1
T(n-1) = T(n-2) +1
.
.
T(1) = 1
T(n) = n = O(n)
```

3. (10 points) What data structure is most suitable to determine if a string s is a palindrome, that is, it is equal to its reverse. For example, "racecar" and "gohan gasalamiimalasagnahog" are palindromes. Justify your answer. Use Big-O notation to represent the efficiency of your algorithm.

   3 - The most suitable ADT that we should use here is a stack ADT. We push each element of the string into the stack until we finish. After, we shall pop each element off and put this back into a new string. If the work is in fact a palindrome, we should see that the string are the same. The **Time Complexity** should be $O(n)$ because we must push and pop off each element from the stack.

4. (10 points) Describe how to implement the stack ADT using two queues. What is the running time of the push and pop functions in this case?

```
void push(T elem){      //normal push function
    queue Q1;
    Q1.enqueue(elem);   //enqueue in Q1
}

T pop(){
    queue Q2;           //create another queue
    while(!Q1.size>1){
        Q2.enqueue(Q1.dequeue());       //make sure there is one last element on the Q1
    }
    T obj = Q1.dequeue();       //Last element is saved in variable
    while(!Q2.isEmpty()){
        Q1.enqueue(Q2.dequeue());   // enqueue again now in Q1
    }
    return obj;         //POP off the last element
}
```

   Push: Insert each element into the queue.... $O(1)$

   Pop: We create a second Queue and then copy all the elements from the first queue onto the second one. We then remove the element and re copy all the elements now back into Q1. The total complexity is: 2n + 2n = 4n .... $O(n)$

5. (10 points) What is the best, worst and average running time of quick sort algorithm? Provide arrangement of the input and the selection of the pivot point at every case. Provide a recursive relation and solution for each case.

   The arrangement could be an array with size n.

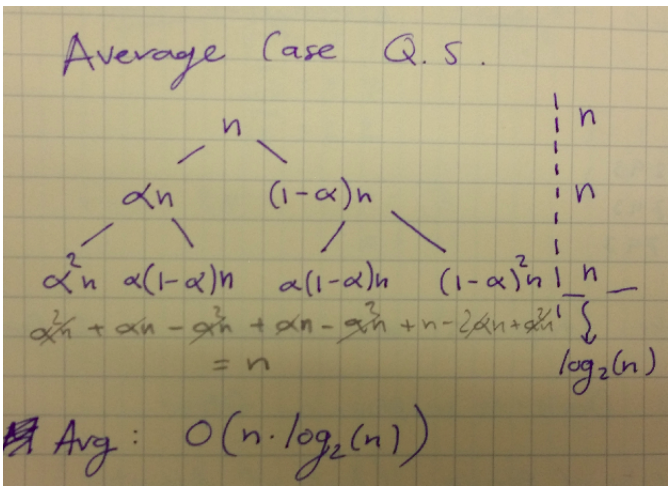
Best	$O(nlog(n))$
Worst	$O(n^2)$
Average	$O(nlog(n))$

In the **best** case, the Pivot point is right in the middle of the set with equal number of elements on each side.
The **worst** case is when the pivot point is a minimum or maximum value in the set.
The **average** case is when we assume the pivot point has an unequal number of elements of each side

```
Best:
T(n) = T(n/2) + T(n/2) + n
T(n) = 2T(n/2) + n
T(1) = 0
===========
T(n) = O(nlog(n))

Worst
T(n) = T(n-1) + T(1) + n
T(n) = T(n-2) + (n-1) + n
T(n) = T(n-3) + (n-2) + (n-1) + n
T(1) = 0
===========
T(n) = n(n+1)/2 +n = O(n^2)

Average
T(n) = T(an) + T((1-a)n) + n     a<0.5
==========
T(n) = O(nlog(n))
```



The above image is a tree proof for the **average case** for quick sort. We see that when we evaluate all the bottom elements they cancel out and whats left is n time the height of the tree. Thus $O(nlog(n))$. For the **Best case,** Here is a new image that substanciates the proof:

```
T(n) = 2T(n/2)+an
     = 2(2T(n/4) +an/2) + an
     = 2^2(T(n/4) + an/4) +2an
    ...
     = 2^k(T(n/2^k))+kan    //k = log(n)

     = O(nlog(n))
```

- 6. (10 points) What is the best, worst and average running time of merge sort algorithm? Use two methods for solving a recurrence relation for the average case to justify your answer.

There is no best case or worst case for the merge sort algorithm. By using this algorithm, we need to increase the memory because we are duplicating and creating new smaller arrays when we divide the main array and storing those values in there. The two proofs I will use are Telescoping and induction.

```
Proof by Telescoping
=====================
T(n) = 2T(n/2) + n
T(n)/n = 2T(n/2)/n +1 //Divide both sides by n
       = T(n/2)/(n/2) + 1   //Recursion
       = T(n/4)/(n/4) + 1 + 1
       = T(n/8)/(n/8) + 1 + 1 + 1
       ...//Base case: T(1)= 0
       = T(n/n)/(n/n) + 1 + 1 + ... + 1
T(n)/n = log(n)
T(n) = nlog(n)
```

```
Proof by Induction
==================
Claim: If T(n) satisfies this
recurrence, then T(n) = nlog(n)

Pf
Base case: n=1
Inductive hypothesis: T(n) = nlog(n)
Goal: show T(2n) = 2nlog(2n)

T(2n) = 2T(n) + 2n
      = 2nlog(n) + 2n
      = 2n(log(2n)-1) + 2n
      = 2nlog(2n)
     ...
      = O(nlog(n))
```

For the proof by **Telescoping** we simply divide both sides by n and then iterate recursively until we obtain the Base case where T(1)=0. From here we see that the right had has a sum of each level that we iterate which is the height of log(n). Then we simply multiply both sides again by n to obtain a O(nlog(n)). For one last method, I included the **masters method:**

```
Proof by Master Method:
=================
T(n) = 2T(n/2) + O(n) if n>1
T(1) = O(1)
According to Master Method:
T(n) = aT(n/b) + f(n)
Then T(n) can be bounded asymptotically
T(n) = O(n^logb(a)*log(n))
if f(n) = O(n^logb(a))

We have a=2, b=2, f(n)=O(n). Then
log2=1 and n^1=n. Therefor merge sort
requires T(n)=nlog(n)=O(nlog(n))
comparisons.
```

- 7) (10 points) R-10.17 p. 493
  For the following statements about red-black trees, provide a justification for each true statement and a counterexample for each false one.

  1. A subtree of a red-black tree is itself a red-black tree.

  – This is **FALSE** because the first red-black tree has a black root so any subtree that has a red root cannot be a red-black tree. A red-black tree must have a black root.

  1. The sibling of an external node is either external or it is red.

  – **True,** otherwise the black depth property would not be satisfies since the other external nodes would have a different number of black nodes on their path. If the external node had a internal black sibling then the **black depth** would not be the same, which would contradict the property of red-black trees that all leaves must have the same black depth.

  1. There is a unique (2,4) tree associated with a given red-black tree.

- **True**, every node in the red-black tree with two red children is uniquely represented as a 4-node in the (2,4) tree. Each node with one red child is uniquely represented as a 3-node in the (2,4) tree and each node with no children is represented as a 2-node (2,4) tree.

1. There is a unique red-black tree associated with a given (2,4) tree.

- **False,** a 3-node has two different possible representation in a red-black tree.

- 8) (10 points) R-10.19 p. 493
  Consider a tree T storing 100,000 entries. What is the worst-case height of T in the following cases?

1. 1.T is an AVL tree.

- For an AVL tree, we know from class that the height of any AVL tree will be h<1.44*log2(n+1). In this case n = 100,000. When plugging the value for n into our equation we obtain a height of 23.

$$h < 1.44 * log_2(100001) \approx 23$$

- Therefore we see that h $\approx$ 23

1. 2.T is a (2,4) tree.

- For the (2,4) tree we know that since there are at least 2^i items at depth i=0,1...h-1 and no items at depth h, we have n>= 1+2+3..+2^h+1 = 2^h -1 . Thus h<= log(n+1)

$$h \leq log_2(100,001) \leq 16.09$$

- So we see that h $\approx$ 16

1. 3.T is a red-black tree.

- From lecture we know that the height of the R-B-Tree is h(x)<=2log2(n(x)+1). Therefore we have:

$$h \leq 2 * log_2(100,001) \leq 32$$

- So we see that h $\approx$ 32

1. 4.T is a binary search tree.

- In the worst case for the Binary Search Tree, we would get an unbalanced three that has all the elements located on one of the branches and none on the other branch. In this case the height would be:

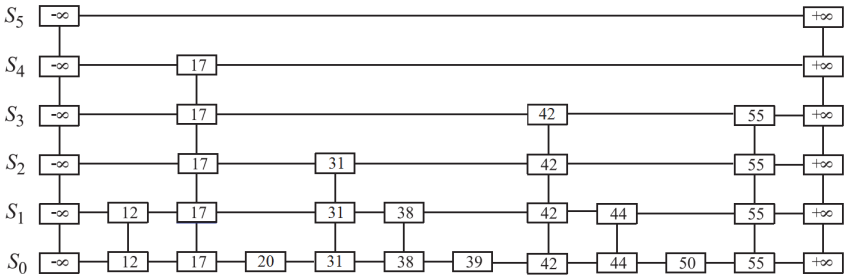
$$h = 100,000 - 1 = 99,999$$

1. (10 points) R-9.16 p. 418

   Draw an example skip list that results from performing the following series of operations on the skip list shown in Figure 9.12: `erase(38), insert(48,x), insert(24,y), erase(55)`. Record your coin flips, as well.

   (a) To begin, we must first erase the value 38 from out skip list in the figure below:

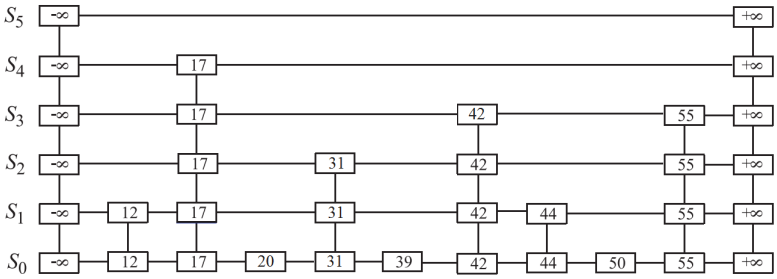
   Step 0

   (b)

   (c) To erase we proceed as following:

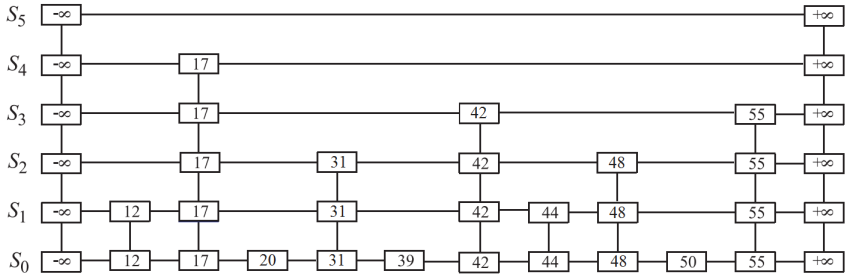
   Step 1 [erase(38)]

   (d)

   (e) Then to Insert the value 48, I used a number generator and took heads as if the outcome was even or tails if the outcome was odd.

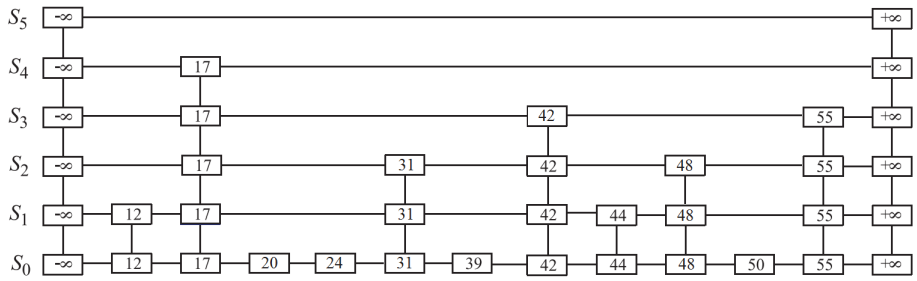
   Step 2 [insert(48,x) - 2 heads]

   (f)

   (g) Now to insert the value 24, we find the location needed and then use the random number generator modulus 2. I get 0 heads. So the value of 24 will only be placed in the complete list.

   Step 3 [insert(24,y) - 0 heads]

   (h)

(i) The final step is to erase the value 55 from our list. This is done below:

Step 4 [erase(55)]

(j)