CSCE 221 Cover Page

Homework Assignment #2

Oneal Abdulrahim UIN: 324007937

oneal.abdulrahim oneal.abdulrahim@tamu.edu

Please list all sources in the table below including web pages which you used to solve or implement the current homework. If you fail to cite sources you can get a lower number of points or even zero, read more on Aggie Honor System Office website: `http://aggiehonor.tamu.edu/`

Type of sources		
People	Lab TA's Dr. Leyk	PA2 Group & Peers
Web pages (provide URL)	cppreference.com	
Printed material	Textbook	
Other Sources	Lecture Slides **chegg.com for images/figures	

I certify that I have listed all the sources that I used to develop the solutions/codes in the submitted work.

On my honor as an Aggie, I have neither given nor received any unauthorized help on this academic work.

Your Name Oneal Abdulrahim Date 27 October 2016

Homework 2

1. (10 points) Describe (in pseudo code) how to implement the stack ADT using two queues. What is the running time of the push and pop functions in this implementation?

   We can use two queues to implement this. Let's call the first queue Q1 and use it to hold the elements, and call the second queue Q2 and use it as temporary storage. Then, we can use the following psuedocode for Stack operations.

   - **Push:**

     ```
     enqueue data to Q1
     ```

   The time complexity for pushing is $O(1)$ constant, since only one operation is needed to add an element to a queue.

   - **Pop:**

     ```
     from (1 to size - 1 of Q1) do:
         copy element by element Q2
     switch the names of Q1 and Q2 //since Q2 now has data, Q1 is temporary queue
     return final element in Q1
     ```

   The time complexity for popping is $O(n)$ and cannot be any faster, due to the nature of queue and how we must iterate through every element to reach the last.

2. (10 points) Solve C-5.8 on p. 224: Describe a non-recursive way of evaluating an expression in post-fix notation.

   Using just one Stack, you can easily store results and data from post-fix expression evaluation. Generally: we must parse each token (a token here is any operand or operator) one by one and make decisions on where to put it based on its type. So, if the current token is an operand (or digit): push this value onto Stack. Otherwise, if the token is an operator, pop two elements from Stack, and perform the operation. Then, push the result on Stack. In the end of the evaluation, there will be only one element in Stack, which will be the answer.

3. (10 points) Linked list questions.

(a) Write a recursive function in C++ that counts the number of nodes in a singly linked list.

```
/// LinkedList length
/* This method returns the number of nodes after the given node
in a singly linked list, inclusively
@param n      The node to start counting from */
int node_counter (const &Node n) {
      if (n.next == NULL) return 1;
      return 1 + node_counter(n.next);
}
// proper method invocation:  int size = node_counter(LinkedList.HeaderNode);
```

(b) Write a recurrence relation that represents the running time for your algorithm.

$$T(n) = T(n-1) + 1$$

Base case: $T(0) = 0$

(c) Solve this relation and provide the classification of the algorithm using the Big-O asymptotic notation.
Solving using backwards substitution:

We start at the definition	$T(n) = T(n-1) + 1$
Noticing recurrence relation	$T(n-1) = T(n-2) + 1$
Replacing $T(n-1)$ into $T(n)$	$T(n) = T(n-2) + 2$
One more step	$T(n-2) = T(n-3) + 1$
Replacing $T(n-2)$ into $T(n)$	$T(n) = T(n-3) + 3$
For general case $T(k)$ where $k \leq n$	$T(k) = T(n-k) + k$
In the worst case, max value of k is n, therefore	$T(k) = T(0) + k = n$

So, we can conclude that this algorithm runs in $O(n)$ time complexity.

4. (10 points) Write a recursive function that finds the maximum value in an array of integers without using any loops.

```
/// Max of two ints
int max(int a, int b) {
     return (a > b) ?  a :  b;
}

/// Find max value of array
/* Finds the max integer in an array.
@param a      The array being traversed
@param n      The current index */
int max_val(int a[], int n) {
     return (n > 0) ?  max(a[n], max_value(a, n - 1)) :  a[0];
}
```

(a) Write a recurrence relation that represents running time of your algorithm.

$$T(n) = T(n-1) + 2$$

Base case (no extra comparisons): $T(0) = 0$

(b) Solve this relation and classify the algorithm using the Big-O asymptotic notation.

We start at the definition	$T(n) = T(n-1) + 2$
Noticing recurrence relation	$T(n-1) = T(n-2) + 2$
Replacing $T(n-1)$ into $T(n)$	$T(n) = T(n-2) + 4$
One more step	$T(n-2) = T(n-3) + 2$
Replacing $T(n-2)$ into $T(n)$	$T(n) = T(n-3) + 6$
For general case $T(k)$ where $k \leq n$	$T(k) = T(n-k) + 2k$
Worst case:, max value is first element, so k is n, therefore	$T(k) = T(0) + 2k = 2n$

So, we can conclude that this algorithm runs in $O(n)$ time complexity.

5. (10 points) Consider the quick sort algorithm.

   (a) Provide an example of the inputs and the values of the pivot point for the best, worst and average cases for the quick sort.
   In the best case, the pivot is the exact median of the list.
   On average, the pivot should be somewhere between the first and last element.
   In the worst case, the pivot is the first or last element. This would cause the sort to change each element.

   (b) Write a recursive relation for running time function and its solution for each case.

Best Case	$T(n) = 2T(n/2) + n$	$\Omega(nlogn)$
Average Case	$T(n) = 2T(n/2) + n$	$\Theta(nlogn)$
Worst Case	$T(n) = T(n-1) + n$	$O(n^2)$

6. (10 points) Consider the merge sort algorithm.

   (a) Write a recurrence relation for running time function for the merge sort.

$$T(n) = 2T(n/2) + n$$

$$\text{Base case: } T(1) = 0$$

   (b) Use two methods to solve the recurrence relation.
   By the Master Theorem,

Relation	$T(n) = 2T(n/2) + n$
	$a = 2,\ b = 2,\ f(n) = n$
	$T(n) = \Theta(n^{log_b a} logn) = nlog_2 n$

   By Induction,

Base Case	$T(2) = 2T(1) + 2$
Inductive Step	Assume for $\frac{n}{2}$, there exists a positive constant c that $T(\frac{n}{2}) \leq c\frac{n}{2}log_2(\frac{n}{2})$
Prove	Such that for any $n \geq \frac{n}{2}$, $T(n) \leq c \cdot nlog_2 n$
Proof	$T(n) = 2T(\frac{n}{2}) + n \leq 2cnlog_2(\frac{n}{2}) + n = cn(log_2(n) - 1) + n =$
	$cnlog_2 n - n(1-c) \leq cnlog_2 n$ for $c \geq 1$
Combining	We conclude that $c \geq 2$, therefore $nlog_2 n$ for any positive $c \leq 1$

   (c) What is the best, worst and average running time of the merge sort algorithm? Justify your answer.
   In the best case, the list is already sorted (though all merges still happen), so $\Omega(nlogn)$
   On average, the list requires some rearranging (all merges still happen), so $\Theta(nlogn)$
   In the worst case, the list requires complete rearranging (all merges happen anyway), so $O(nlogn)$

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.

   (a) A subtree of a red-black tree is itself a red-black tree.
   This is false, because a subtree will have a red root, which is not a property of RB trees. Unless the tree rebalances, this is false.

   (b) The sibling of an external node is either external or it is red.
   This is true, as all external nodes (leaves) have the same black depth

   (c) There is a unique (2,4) tree associated with a given red-black tree.
   This is true, because each black node can be merged with however many children up to 4

   (d) There is a unique red-black tree associated with a given (2,4) tree.
   This is false, because the solution is not unique. A particular black node can have one red child, left or right, for instance.

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?

   (a) T is an AVL tree.
   $$h \leq 2 \cdot log_2(n+1)$$
   $$h \leq 34$$

   (b) T is a (2,4) tree.
   $$h \leq (1.44) log_2(n+1)$$
   $$h \leq 24$$

   (c) T is a red-black tree.
   $$h \leq log_2(n)$$
   $$h \leq 17$$

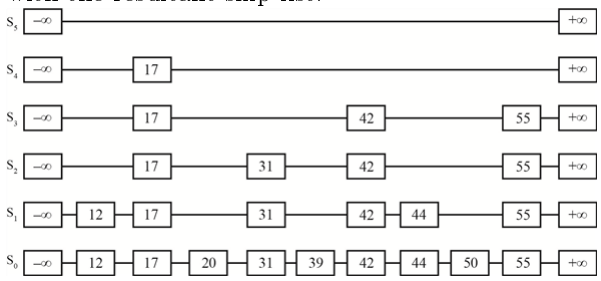
   (d) T is a binary search tree.
   In the worse case, $h \leq n$, so $h \leq 100000$

9. (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.

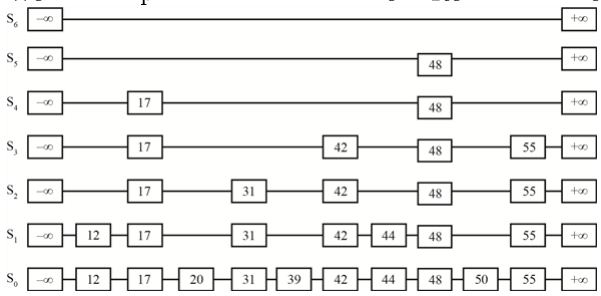
(a) `erase(38)`

There is no need to flip for erase, simply find and delete all instances of 38 in this case. We end up with the resultant skip list:



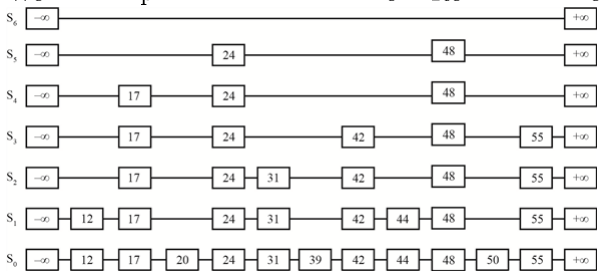
(b) `insert(48,x)`

We must flip until tails is obtained. Result of tosses: H H H H H T. The resultant skip list:



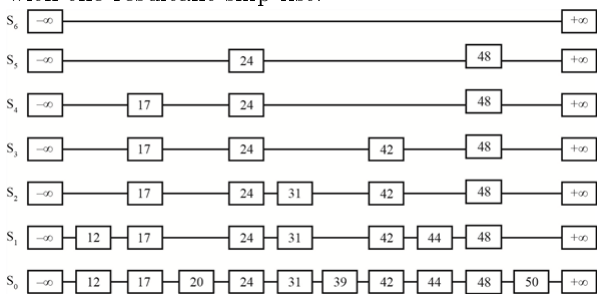
(c) `insert(24,y)`

We must flip until tails is obtained. Result of tosses: H H H H T. The resultant skip list:



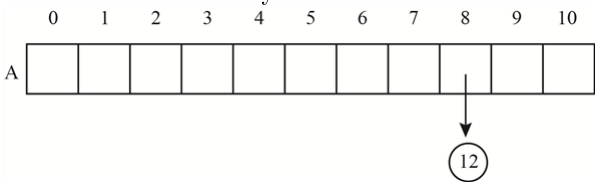
(d) `erase(55)`

There is no need to flip for erase, simply find and delete all instances of 55 in this case. We end up with the resultant skip list:

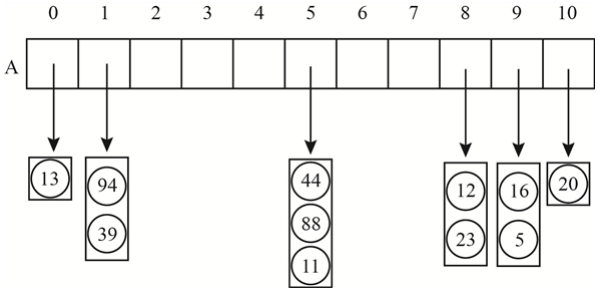


10. (10 points) R-9.7 p. 417

Draw the 11-entry hash table that results from using the hash function, $h(k) = (3k + 5) \bmod 11$, to hash the keys 12, 44, 13, 88, 23, 94, 11, 39, 20, 16, and 5, assuming collisions are handled by chaining.
We start with an array of size 10, since there is input of 10 integers. To hash 12, the resultant index is 8 by our formula. Insert key 12 at index 8.

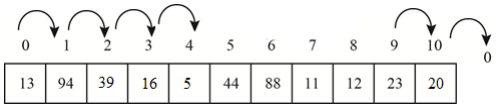


And so on, arriving at:



11. (10 points) R-9.8 p. 417

What is the result of the previous exercise, assuming collisions are handled by linear probing?
Linear probing addresses the issue of collisions by checking for empty buckets to place items in. Specifically, it checks first $(i+1)\%n$, then $(i+2)\%n$ and so on, where i is the current index, and $n$is the size of the hash table. We arrive at this hash table, after all of these calculations have been done



12. (10 points) R-9.10 p. 417

What is the result of Exercise R-9.7, when collisions are handled by double hashing using the secondary hash function $h_s(k) = 7 - (k \bmod 7)$?
When we double hash collisions, this means we use another hash function to define the location of a particular element. So, if we run into a collision using the first hash function, we must use the second hash function to decide its location. Thankfully, there were no collisions after the double hash was completed!

0	1	2	3	4	5	6	7	8	9	10
13	94	23	88	11	44	39	5	12	16	20