

CSCE 221 Cover Page
Homework Assignment #2
Due March 25 at 23:59 pm to eCampus

Asa

Hayes

UIN: 525003952

User Name: AsaHayes

asahayes@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 Aggie Honor System Office <http://aggiehonor.tamu.edu/>

Type of sources			
People			
Web pages (provide URL)			
Printed material	This course's textbook	Rosen - Discrete Mathematics & Applications	
Other Sources			

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

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

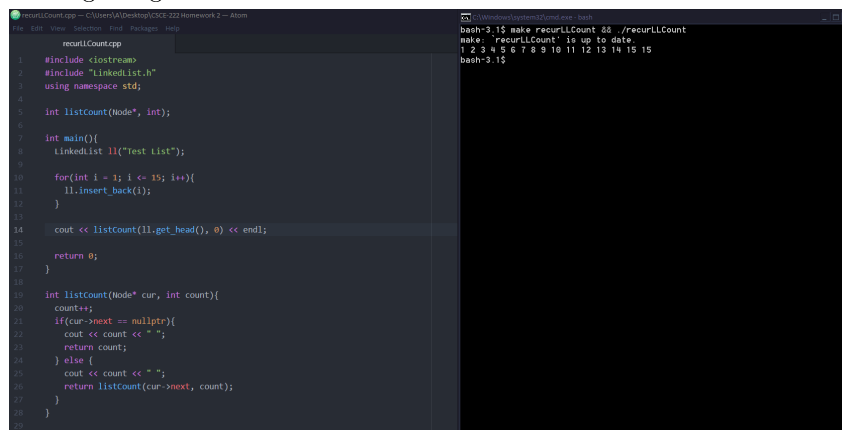
Your Name Asa R Hayes Date 27 March, 2018

Homework 2

due March 25 at 11:59 pm to eCampus.

- (15 points) Describe how to implement the stack ADT using two queues.
 - Write a C++ function that implements your solution. You can use the C++ STL queue container. Code can be run with “make stackFromQueues && ./stackFromQueues”. Output will be list of popped elements.
 - What is the running time of the push and pop functions in this case?
The push operation has a runtime of $1 + 2n$, or $O(n)$, where n is the size of the list before insertion. Pop is only $O(1)$, as it does not require the movement of any other elements besides the top of the stack.
- (15 points) Write a recursive function in C++ that counts the number of nodes in a singly linked list.
 - Test your function using different singly linked lists. Include the code and screenshots with testing cases.
Code can be run with “make recurLLCount && ./recurLLCount”. Output will be size of linked list in recurLLCount.cpp.

Testing Images:



The screenshot shows a C++ program named `recurLLCount.cpp` and its execution output. The code defines a `LinkedList` class with `insert_back` and `get_head` methods. The `main` function creates a linked list, inserts 15 nodes, and calls `recurLLCount`. The recursive function `recurLLCount` counts the number of nodes in the linked list.

```
1 #include <iostream>
2 #include "LinkedList.h"
3 using namespace std;
4
5 int listcount(Node*, int);
6
7 int main() {
8     LinkedList ll("Test List");
9
10    for(int i = 1; i <= 15; i++){
11        ll.insert_back(i);
12    }
13
14    cout << listcount(ll.get_head(), 0) << endl;
15
16    return 0;
17 }
18
19 int listcount(Node* cur, int count) {
20     count++;
21     if(cur->next == nullptr) {
22         cout << count << " ";
23         return count;
24     } else {
25         cout << count << " ";
26         return listcount(cur->next, count);
27     }
28 }
```

Output:

```
bash-3.1$ make recurLLCount && ./recurLLCount
make: 'recurLLCount' is up to date.
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 15
bash-3.1$
```

- Write a recurrence relation that represents your algorithm.
Relation: $a_n = a_{n-1} + 1$
 - Solve the relation using the iterating or a recursive tree method to obtain the running time of the algorithm in Big-O notation.
$$a_n = a_{n-1} + 1$$
$$a_{n-1} = a_{n-2} + 1$$
$$a_{n-2} = a_{n-3} + 1$$
$$a_{n-3} = a_{n-4} + 1 \dots$$
$$a_4 = a_3 + 1$$
$$a_3 = a_2 + 1$$
$$a_2 = a_1 + 1$$
$$a_1 = 1$$
$$a_n = 1 * na_n = O(n)$$
- (15 points) Write a C++ recursive function that finds the maximum value in an array of integers without using any loops.
 - Test your function using different input arrays. Include the code and screenshots with testing cases.
Code can be run with “make noLoopMax && ./noLoopMax”. Output will be incremental display of each recursion’s result.

Testing Images:

```

noLoopMax.cpp
1 #include <iostream>
2 using namespace std;
3
4 int maxRecur(int[], int, int, int);
5
6 int main(){
7     // Test 1: Values 1-10, chosen randomly, used for testing
8     int arr1[10] = {7,1,5,2,8,6,10,3,9,4};
9     cout << "Max 1: " << maxRecur(arr1, sizeof(arr1) / sizeof(arr1[0]), 0, 0) << endl;
10
11     // Test 2: Randomly generated ints from range 1-100, from random.org
12     int arr2[10] = {41,6,8,14,56,76,86,84,21,94};
13     cout << "Max 2: " << maxRecur(arr2, sizeof(arr2) / sizeof(arr2[0]), 0, 0) << endl;
14
15     // Test 3: Randomly generated ints from range 1-1000, from random.org
16     int arr3[10] = {848,895,67,930,723,392,139,174,889,863};
17     cout << "Max 3: " << maxRecur(arr3, sizeof(arr3) / sizeof(arr3[0]), 0, 0) << endl;
18 }
19
20 // note: passing arrSize as argument reduces amount of total calculations,
21 // also any constant value can be calculated inside of maxRecur.
22 int maxRecur(int* a, int arrSize, int index, int max){
23     cout << "Current Max: " << max << endl;
24     cout << "Remaining Elements: ";
25     // This loop does not count against the question, as it is only for testing/demo
26     for(int i = index; i < arrSize; i++){
27         cout << a[i] << " ";
28     }
29     cout << endl;
30
31     // productive code
32     if(index == arrSize){
33         return max;
34     }
35 }
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100

```

(b) Write a recurrence relation that represents your algorithm. Solve the relation and obtain the running time of the algorithm in Big-O notation.

Relation: $a_n = a_{n-1} + 1$

$$a_n = a_{n-1} + 1$$

$$a_{n-1} = a_{n-2} + 1$$

$$a_{n-2} = a_{n-3} + 1$$

$$a_{n-3} = a_{n-4} + 1 \dots$$

$$a_4 = a_3 + 1$$

$$a_3 = a_2 + 1$$

$$a_2 = a_1 + 1$$

$$a_1 = 1$$

$$a_n = 1 * na_n = O(n)$$

Despite the different problems being solved, this problem does the same amount of operations per list size as the LinkedList size function from (2).

4. (15 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.

(a) Best Case:

- Runtime: $O(n \log(n))$
 - Input Arrangement: Split into 2 parts, as closely as possible
 - Pivot Point: Median value of input list
- Recursive Relation: $T(n) = 2T(\frac{n}{2}) + n$

Solution (Master Theorem): $a = 2$, $b = 2$, $f(n) = n$; $\log_2(2) = 1$, $n^1 = n$, thus the runtime is $\Theta(n \log(n))$.

(b) Average Case:

- Runtime: $O(n \log(n))$
- Input Arrangement: Split into 2 parts, roughly even
- Pivot Point: Random/Last element in list

Recursive Relation: $T(n) = T(i-1) + T(n-i) + n$, where $0 \leq i \leq n-1$, i being the size of one of the partitions

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

Sum of all cases, divided by n as each element has $1/n$ chance of being pivot

$$T(n) = \frac{1}{n} \sum_{i=1}^n (T(i-1) + T(n-i) + n)$$

$$T(n) = \frac{2}{n} (T(0) + T(1) + T(2) + \dots + T(n-3) + T(n-2) + T(n-1)) + n$$

$$n * T(n) = 2(T(0) + T(1) + T(2) + \dots + T(n-3) + T(n-2) + T(n-1)) + n^2$$

$$(n-1) * T(n-1) = 2(T(0) + T(1) + T(2) + \dots + T(n-3) + T(n-2) + T(n-1)) + n(n-1)$$

$$(n-1) * T(n-1) = 2(T(0) + T(1) + T(2) + \dots + T(n-3) + T(n-2) + T(n-1))$$

$$n * T(n) - (n-1) * T(n-1) = (2(T(0) + T(1) + T(2) + \dots + T(n-3) + T(n-2) + T(n-1)) + n^2) - (2(T(0) + T(1) + T(2) + \dots + T(n-3) + T(n-2) + T(n-1)))$$

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

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

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2}{n+1}$$

$$\frac{T(2)}{n+1} = \frac{T(n)}{n+1} - \frac{T(n-1)}{n}$$

$$\frac{T(n)}{n+1} = \frac{T(0)}{1} + 2\left(\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n} + \frac{1}{n+1}\right)$$

$$T(n) = 2n((\text{HarmonicSeries}(n+1) - 1))$$

$$T(n) = \Theta(n \log n)$$

(c) Worst Case:

- Runtime: $O(n^2)$
- Input Arrangement: List is not split, or split into 2 lists with lengths n and 0
- Pivot Point: Largest/Smallest value in list

Recursive Relation: $T(n) = T(n-1) + n$

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

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

$$T(n-2) = T(n-3) + (n-2)$$

$$T(n-3) = T(n-4) + (n-3) \dots$$

$$T(4) = T(3) + 4$$

$$T(3) = T(2) + 3$$

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

$$T(1) = 0$$

$$T(n) = n + (n-1) + (n-2) + (n-3) + \dots + 3 + 2 \approx \frac{n^2}{2}$$

5. (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.

- (a) Best Runtime: $O(n \log(n))$
- (b) Average Runtime: $O(n \log(n))$
- (c) Worst Runtime: $O(n \log(n))$

Recurrence Relation (Avg.): $T(n) = 2T(\frac{n}{2}) + n$

Solution 1 (Master Theorem): $a = 2$, $b = 2$, $f(n) = n$; $\log_2(2) = 1$, $n^1 = n$, thus the runtime is $\Theta(n \log(n))$.

Solution 2:

$$T(n) = 2T(\frac{n}{2}) + n$$

$$\begin{aligned}
&= 2(2T(\frac{n}{4}) + \frac{n}{2}) + n = 4T(\frac{n}{4}) + 2n \\
&= 2(4T(\frac{n}{8}) + \frac{2n}{4}) + n = 8T(\frac{n}{8}) + 3n \\
&= 2^k T(\frac{n}{2^k}) + (k * n) \\
1 &= \frac{n}{2^k} \rightarrow k = \log_2(n) \\
T(n) &= 2^{\log_2(n)} T(\frac{n}{2^{\log_2(n)}}) + n(\log_2(n)) \\
&= nT(\frac{n}{n}) + n(\log_2(n)) \\
&= nT(1) + n\log_2(n) \\
&= n + n(\log_2(n)) \\
n + n(\log_2(n)) &= \Theta(n\log_2(n))
\end{aligned}$$

6. (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.

False: In the case that the subtree starts with a red node as its root, this disqualifies it from being a red-black tree as the root of a red black tree must be a black node.

- (b) The sibling of an external node is either external or it is red.

True: For the case of the sibling being an external node, this has the black-count at the same for both paths, keeping the validity of the red-black tree. For the case of the sibling being a red node, this is valid as well, as the black count is still the same as long as the children of that red node were external nodes, else the black count would not be constant.

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

True: To extend any red-black tree into a 2,4 tree, one can just merge all of the red child nodes of a black node into that node.

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

False: It is not possible to perfectly reverse the process described in (c) because the nodes in the 2,4 tree have different ways the expanded red nodes can be arranged.

7. (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.

AVL Trees have a maximum height of $\log(n)$, so $\log_2(100000) \approx 16.61$.

- (b) T is a (2,4) tree.

2,4 Trees have a maximum height of $\log(n)$, so $\log_2(100000) \approx 16.61$.

- (c) T is a red-black tree.

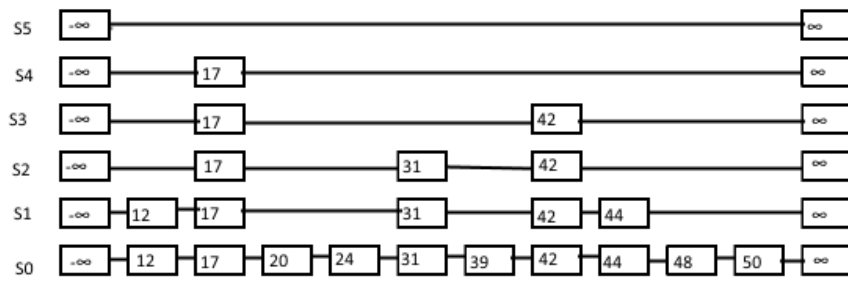
Red-Black trees have a maximum height of $2\log n$, so $2\log_2(100000) \approx 33.22$.

- (d) T is a binary search tree.

As a regular binary search tree is not necessarily balanced, it could be the case that all the nodes are on one side. A binary search tree only requires that all nodes to the left of root have lesser data values, and all nodes to the right have greater data values than root, so it can become unbalanced much more easily. If all nodes beginning with root are larger than the next, this would essentially become a linear tree, which would just have a height of n , or 100,000 in this case. For data like that, linked list or array would most likely be a better choice.

8. (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.



Coin Flips for insert(48, x): Tails on first flip

Coin Flips for insert(24, y): Tails on first flip (actually, I promise)