**CptS 223 – Advanced Data Structures in C++**

**Fall 2020**

***Take-Home Exam 2***

Friday, November 13, 2020

**Your Name: John Sbur**

**ID Number: 11663921**

**TA’s Name: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

READ THE FOLLOWING INSTRUCTIONS:

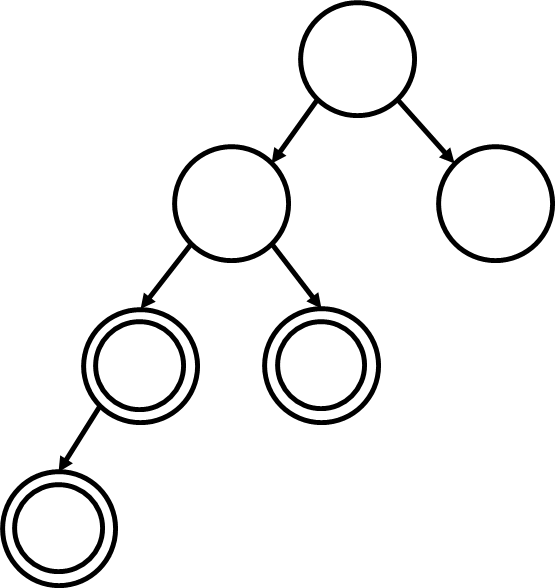
This exam is take-home. You must work **individually** on this exam. You may use your book, notes, and online resources if necessary. Please either handwrite or type your answers into this document. Late exam solutions will **not** be accepted! **Please show all work!!!**

You must submit the exam through Git by Monday, November 16, midnight. On your local file system, and inside of your Git repo for the class, create a new branch called Exam2. In the current working directory, also create a new directory called Exam2. Convert your exam solution file to a.pdf called Exam2.pdf. The Exam2.pdf file must be added and committed. Merge your Exam2 branch with your Master branch and push the Master to the remote origin. Be sure to answer each question precisely. Do not provide superfluous details in your answers. *NOTE: you do not need to comment your code solutions.*

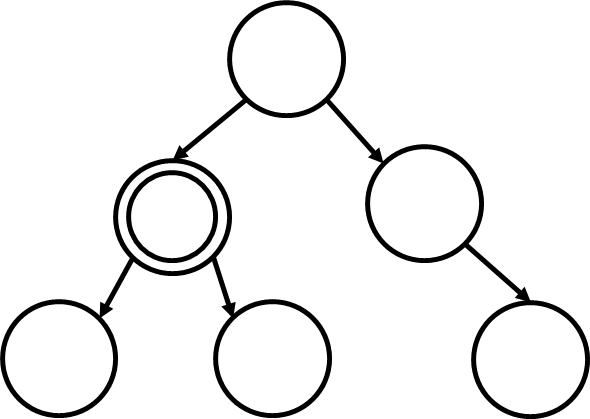
|  |  |  |
| --- | --- | --- |
| **Section** | **Points Possible** | **Points Earned** |
| 1 | 25 |  |
| 2 | 15 |  |
| 3 | 40 |  |
| 4 | 20 |  |
| **Total** | **100** |  |

**Section 1: Red-black Trees**

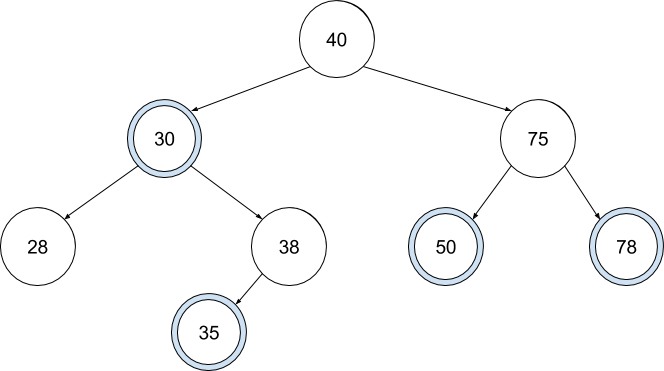
1. (5 pts) Is the following tree a valid red-black tree? Why or why not?

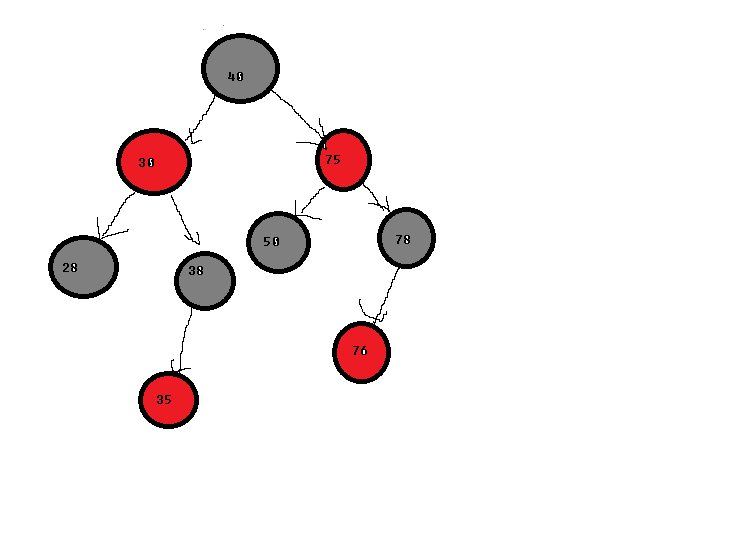
No. There is a red node whose child is red

1. (5 pts) Is the following tree a valid red-black tree? Why or why not?

 No, the distance to the furthest null pointer on the right in terms of black nodes is 3 while on the left it’s 2. For every path from a node to the null pointer, there has to be the same number of black nodes and since there isn’t, the tree is not a valid red-black tree.

1. (5 pts) Insert 76 into this Top Down Red Black tree:





1. (5 pts) When deleting a node from a red-black tree, we always want the node we remove to be colored \_\_\_\_red\_\_\_\_\_ [red/black/either]. Explain your answer.

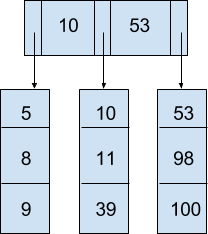
We want to color it red. If it’s already red, then we can do a simple BST deletion. If it was black, then we make it red and balance the tree with it being red before making the deletion.

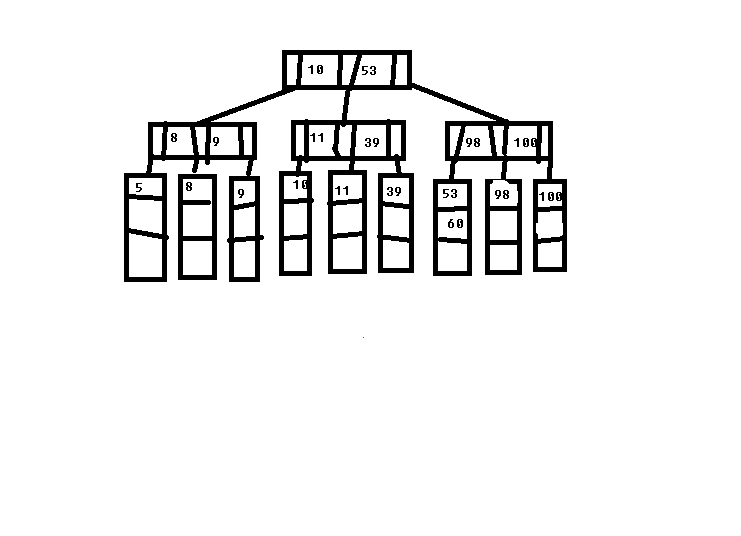
1. (5 pts) AVL vs. Red-Black trees: These are both balanced trees, but why would we use one over the other?

AVL trees are very strictly balanced and because of that, search and update functions are faster, however, insertion and deletion take longer because everything has to be strictly balanced whenever one of those occur. RB trees have less strict balancing requirements, so inserting and deleting take less time than AVL trees, but then search and update functions take longer. AVL trees are better for holding data and making less insertions and deletions while RB trees are better if you still want to adjust the tree often as those actions will take less time.

**Section 2: B+ Trees**

1. (10 pts) The following B+ tree has M = 3, L = 3. Insert 60 into the tree and draw the results. Make sure your final tree is a valid B+ tree.





1. (5 pts) What is the primary reason to use a B+ Tree?

If we want to store data on a disk and have it organized, we can use a B+ tree. The actual data is stored in the leaf nodes while pointers to said data is in the internal nodes so it’s less memory intensive, but the main use is for storing data on a disk.

**Section 3: Hash Tables**

1. (5 pts) List the key factors that affect the Big-O performance of a hash table:

The current number of elements in the hash table as well as the size of the hash table affect the performance of the table. They’re used to determine the load factor and when to rehash the table for better performance.

1. (10 pts – 5 pts/table) Given a hashing function hash(x) = ((x \* x) + x) % 11. Insert the value **17** into each hash table using the rules specified below. Note that some of the boxes in each hash table are already full. If there’s a collision, start the probing at i = 0 and increment from there.

Linear Probing having probe(i) = ( hash(x) + (i + 1) ) % 11

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 10 | 11 | 17 |  |  |  | 2 |  |  | 6 | 15 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

Quadratic Probing having probe(i) = ( hash(x) + (i^2 + i) + 1 ) % 11

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 10 | 11 |  |  |  | 17 | 2 |  |  | 6 | 15 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

1. (5 pts) Define the term "load factor" as it relates to hash tables.

Load factor is the ratio of the elements in the list to the size of the table or the max number of elements the table can hold. It’s used in determining when to rehash a hash table. It’s generally advised to rehash at around a load factor of a little higher than .5.

1. (5 pts) After we insert 17 into these hash tables, should we resize and rehash the values? Use Table Size and N to calculate the load factor (λ) and apply the general rule of when you rehash for linear and quadratic probing.

We should rehash after inserting 17. For open addressing style hash tables where you use quadratic or linear probing, you want to rehash when the load factor becomes greater than .5. Since now there are 6 elements and the table size is 10, the load factor is .6, so we should rehash.

1. (5 pts) If this was a hash table using separate chaining instead of linear probing, would it be time to rehash after we insert 17? Explain.

No since with separate chaining, instead of rehashing, we simply add the element that causes a collision onto a linked list. So we would start a linked list at index 9 and not rehash when we insert 17 into a hash table that uses separate chaining.

1. (5 pts) What is lazy deletion and why is it important for linear and quadratic probing?

Lazy deletion in hash tables is when if an object would be deleted, instead it is marked as deleted, rather than erasing everything and the spot is considered empty when an insertion is made on that index. Using this method, we don’t have to deallocate memory and then reallocate memory when we want to make an insertion as when we are inserting a new element, we can simply replace the value of a deleted index with new values and mark the bucket as being not deleted.

1. (5 pts) Which of the following hashing algorithms is likely to produce a faster HashTable implementation? Why?

|  |  |
| --- | --- |
| Hashing Algorithm A | Hashing Algorithm B |
| int hash(string text):  int result = 0;  for(char ch : text)  result += ch;  return result; | int hash(string text):  int result = 1;  for(char ch : text)  result \*= ch \* ch + (11 \* ch) + 7;  return result; |

Algorithm B will likely be faster. Algorithm B is a type of Quadratic Probing algorithm which in general is faster than Linear Probing, or Algorithm A. This is because linear probing can cause clustering more often and thus more time finding an open index from the one you start at if there is collision while with Quadratic Probing, it tends to cluster less so there will be less collisions and a faster insertion.

**Section 4: Parallel Programming**

1. (5 pts) What is parallel programming? Explain.

Parallel programming is a method of running a program in which tasks in the program are divided among different systems or cores to fulfill and then combine back together in the end to be processed. So, for example if I have to make a delivery of 100 tons of apples to 10 different towns, I could do them myself, but then it would take a long time. Instead, I assign 10 people to deliver the required number of apples to each town, or I could have 10 people bring a smaller number of apples to each town so it’s easier to transport.

1. (5 pts) When should we use critical regions in OpenMP? Explain.

Critical regions in Open MP make it so that threads created by OpenMP cannot access the region at the same time. It helps for when we want threads to work in order and not do operations in the wrong order. Maybe we want the first thread to do something and then the second thread needs to update that afterwards. The critical region associated with updating that variable helps the threads work in order whereas without, the second one could update the variable before the first one edits it.

1. (5 pts) What is meant by synchronizing tasks? Explain.

Synchronizing tasks means putting constraints on tasks so that they move more smoothly. The constraints can be things like the order in which they occur or protecting access from one task or thread from another. This is all done to ensure that nothing gets scrambled up between tasks and threads.

1. (5 pts) How can we apply parallel programming to hash tables? Explain.

We can use it to do multiple actions at once like a large amount of insertions, deletions, finding, or updating functions, and use synchronization with parallel programming to make sure that no collisions in the threads are there. This can make the code run as fast as possible while avoiding collisions of data.