**MECHTRON 2MD3 Assignment 5 – Michael Giancola**

**1.** Check your assignment 4 solution for memory leaks using valgrind on the pascal server. If you find leaks, debug your code until valgrind reports no leaks. You should follow the "rule of three" to make sure your LinkedBinaryTree class has working destructor, copy constructor, and overloaded copy assignment operator. Note that code supplied for question 2 below has a leak-free LinkedBinaryTree class. You may use this as a reference but please work to modify your own code rather than copying the class directly. You should submit your debugged code as "1234-asg4-debugged.cpp" where 1234 is your student ID. Please also write a brief description (1-3 sentences) about what was wrong, if anything, and how you fixed it. If you were not able to fix it, discuss where you got stuck.

**Code Corrections:**

Copy Assignment Operator:

```
// copy assignment operator
LinkedBinaryTree& LinkedBinaryTree::operator=(const LinkedBinaryTree& t) {
    if (this != &t) {
      // If tree already contains data, delete it
      if (_root != NULL)
      {
        destructorHelper(_root);
      }
      _root = copyPreOrder(t.root());
      score = t.getScore();
    }
    return *this;
  }

LinkedBinaryTree::LinkedBinaryTree()
: _root(NULL), n(0) { }
```

The above block of code copies the original tree as another object of the same type.

Destructor:

```
LinkedBinaryTree::~LinkedBinaryTree() { //destructor
    destructorHelper(_root);
}

void LinkedBinaryTree:: destructorHelper (LinkedBinaryTree:: Node* node){
    if (node == NULL)
    {
        return;
    }

    destructorHelper(node ->left);
    destructorHelper(node->right);
    delete node;
}
```

The original A4 code of mine didn't include a destructor to deallocate the previously allocated memory.

Copy Constructor:

```
// copy constructor
LinkedBinaryTree::LinkedBinaryTree(const LinkedBinaryTree& t) {
  _root = copyPreOrder(t.root());
  score = t.getScore();
}
```

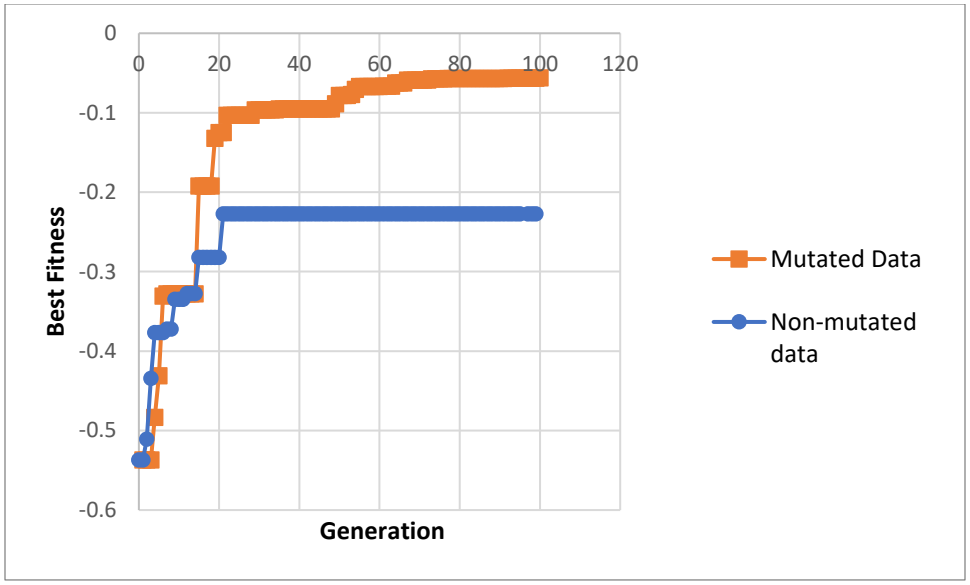
This function creates a new tree that is a copy of the original tree.

Changed Create Expression tree:

```
LinkedBinaryTree createExpressionTree(string postfix) {       //creates trees to be put into a vector
    stack<LinkedBinaryTree::Node*> stack;                        //stack of nodes to hold roots of tree
    stringstream ss(postfix);
    string tok;
    LinkedBinaryTree::Node *holder;                                      //temporary variable to hold the new tree every iteration
    int count = 0;                                               //a variable to count number of nodes in tree

    while (getline(ss, tok, ' '))                                //tokenizes string input for every space seperation
    {
        holder = new LinkedBinaryTree::Node();                           //creates new tree
        holder->elt = tok;
        holder->left = NULL;
        holder->right = NULL;
        count++;

        if ((isdigit(tok[0]) || isdigit(tok[1])) || (isalpha(tok[0]) && tok != "abs")) //if token is number or a/b, do nothing and push to stack
        {
        }
        else if (tok == "abs")                                   //the right node holds the other node in the stack if token is abs
        {
            holder->right = stack.top();
            holder->right->par = holder;
            stack.pop();

        }
        else                                                       //if operator takes two most recent stack elements and makes them the children of the root
        {
            holder->right = stack.top();                        //take the top of the stake and make right and left pointers to then pop the stack element
            holder->right->par = holder;
            stack.pop();
            holder->left = stack.top();
            holder->left->par = holder;
            stack.pop();

        }
        stack.push(holder);                                      //push the root of the tree to the stack
    }
    LinkedBinaryTree result;                                     //by this point their should only be only thing left if the stack
    result._root = stack.top();                                  //create a new tree and set the stack element to the root of the new tree
    result.n = count;
    return result;
}
```

The function in my original submission was creating a full tree whereas now it is creating individual nodes. These individual nodes that are linked together use less memory.

**2. Part 1:**




```
Best tree:
((a / (((abs((a - (a > a))) * ((a - (b / abs(((abs((b / b)) * (((b > b) > (a + (a + a))) * a))
 * (a > b))))) * (b / a))) * b) - (a / a))) > (abs(a) / (a / (a - a))))
Generation: 99
Size: 53
Depth: 15
Fitness: -0.0563453
```

**Part 2 -** Implement a comparator ADT called LexLessThan which performs a lexicographic comparison of two trees a and b as follows: If the scores of a and b differ by less than 0.01, then tree a is “less than” tree b if a has more nodes than b. Otherwise, compare the trees by their score. The goal of using this comparator is to favour simpler trees only when their scores are similar. Repeat the experiment form part 1 using the new comparator. To do so, you must uncomment the line under “sort using comparator class”. After doing so, does evolution produce simpler trees? If not, try to increase the number of generations.

The evolution produces a much simpler tree after implementing the LexLessThan operator. The fitness score becomes worse unless the experimental parameters are altered within the main function.

```
Best tree:
(a > (((a + (a + b)) - (a - a)) + a))
Generation: 53
Size: 13
Depth: 5
Fitness: -0.0714162
```

**Part 3.** Modify anything in genetic_programming_01.cpp in order to evolve a tree with high fitness and low complexity. What is the best fitness and complexity you can get in the single best tree?

```
Best tree:
((a - b) > (a / abs(b)))
Generation: 860
Size: 8
Depth: 3
Fitness: -0.030319
```

As shown in the printout above, the best fitness I can get with the changes I have made is -0.030319. Within the main function, I changed the experiment parameters, lowering the NUM_TREE, increasing the MAX_DEPTH, and changing the MAX_GENERATIONS. Increasing the number of trees results in more population because more comparisons can be made and it is more likely a better random expression is generated. Lowering the depth to ensure the expression trees were more simplified and contained fewer characters. Increasing the max generations also allowed for more comparisons to take place between expression trees to produce the best possible output. These alterations significantly improved the fitness of my program.

**3. Give an example of a worst-case sequence with n elements for insertion-sort using a priority queue. Show that insertion-sort runs in $n^2$ time on such a sequence.**

Insertion-sort is a type of Priority Queue sort where it is implemented with a sorted sequence. The insertion sort runs on an O(n^2) sequence because an insertion sort algorithm places each element of the sequence in a sorted priority queue. After, each item must be removed from the priority queue which takes O(n) run time. Therefore, the total run time is O(n^2+n) which simplifies to O(n^2).

Example: Consider the sequence [8,5,3,7,2,4].

| | **List L** | **Priority Queue P** |
|---|---|---|
| **Input** | (8,5,3,7,2,4) | () |
| Phase 1 | (5,3,7,2,4)<br>(3,7,2,4)<br>(7,2,4)<br>(2,4)<br>(4)<br>() | (8)<br>(5,8)<br>(3,5,8)<br>(3,5,7,8)<br>(2,3,5,7,8)<br>(2,3,4,5,7,8) |

| | | |
|---|---|---|
| Phase 2 | (2)<br>(2,3)<br>(2,3,4)<br>(2,3,4,5)<br>(2,3,4,5,7)<br>(2,3,4,5,7,8) | (3,4,5,7,8)<br>(4,5,7,8)<br>(5,7,8)<br>(7,8)<br>(8)<br>() |

The above insertion sort on list L = (8,5,3,7,2,4) is executed in the table above. In phase 1, the first element of the list L is repeatedly removed and inserted into the priority queue P using the insert() function. The list is scanned and once inserted into the priority queue, is placed in order from least to greatest. In phase, 2 the min() function is used to find the minimum value in the priority queue, and the removeMin() operation is repeated on P. This returns the first element of the list and adds each removed element after that to the end of L, resulting in a successful insertion sort.

The run time of Phase 1 of the above insertion sort can be described as:

$$O(1+2+\ldots+(n-1)+n) = O\left(\sum_{i=1}^{n} i\right).$$

Therefore, this phase runs in O(n^2) time, meaning the entire algorithm runs in O(n^2) time.

An example of a worst-case scenario would be if List L was unsorted in reverse order before it was in the priority queue. For example, a list L = {4,3,2,1} where n = 4. This is because it is a queue, so things are brought from the front of the list and placed at the back. Every element must be compared to the element being added.

**4. At which nodes of a heap can an entry with the largest key be stored?**

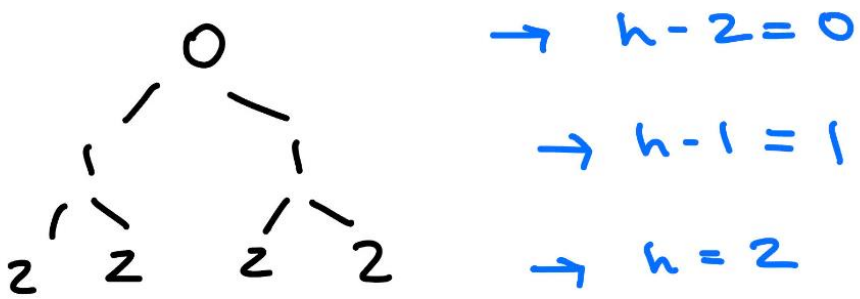
The largest key is always stored at h or h-1 because heaps must satisfy the condition that the key to a child node must be greater than the key to a parent key. This rule, however, does not say that the largest key has to be in the largest subtree, which is why it can be found in either h or h-1. This is shown in the diagrams below.

**5. Let T be a complete binary tree such that node v stores the key-entry pairs (f(v),0), where f(v) is the level number of v. Is tree T a heap? Why or why not?**

The heap property states that for every internal node, the key(v) must be greater than or equal to the key(parent(v)).

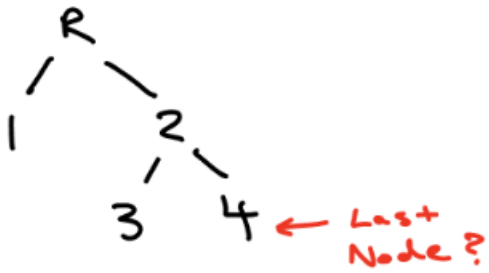
- **Heap-Order:** for every internal node v other than the root, key(v) ≥ key(parent(v))

In this case, tree T would be a heap because parent nodes will always be at a smaller level of the tree than the child. This means the key of the child will always be greater than the parent, agreeing with the heap-order property of heaps.




**6. Review Section "Down-Heap Bubbling after a Removal" in the Goodrich test chapter 8. Explain why the case where the right child of r is internal and the left child is external was not considered in the description of down-heap bubbling.**

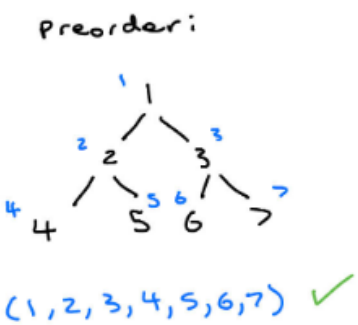
Down-Heap bubbling is a process where the child with the smaller key is chosen and swapped with the root is necessary, only if both children of the root are internal. This process takes place between nodes until the heap-order property is not violated. The case where r is the internal node, and the left child is the external node is not considered because down-heap bubbling algorithms can not be executed on this data structure. The last node of the heap is removed and replaced as the root in the algorithm but if the last node doesn't then this algorithm can't be applied. The last node is considered the rightest node of the final level of the tree, and as shown below, that does not exist in the situation described in the question.

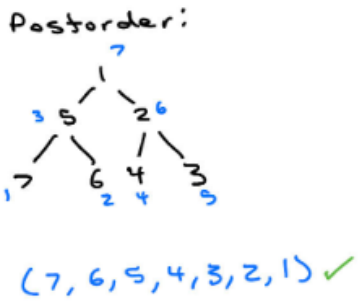


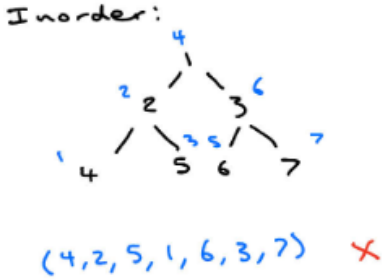
**7. Is there a heap T storing seven distinct elements such that a preorder traversal of T yields the elements of T in sorted order? How about an inorder traversal? How about a postorder traversal?**

Using the heap-order property that was defined in the lecture (the parents of a node must have a smaller key than its children), this is possible for preorder traversal in increasing order and postorder traversal in decreasing order.

Heap Examples:

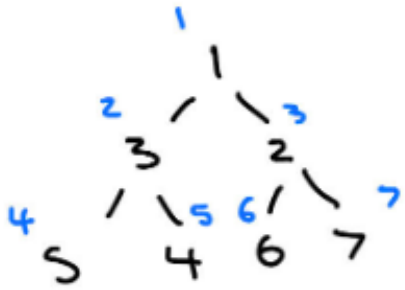


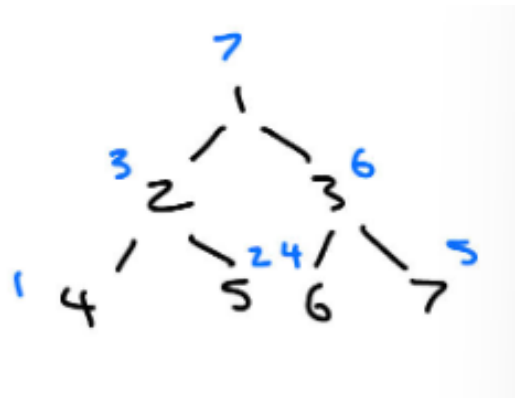

Therefore, the preorder and postorder traversals yields the elements of T in sorted order whereas the inorder traversal does not.

**8. Bill claims that a preorder traversal of a heap will list its keys in nondecreasing order. Draw an example of a heap that proves him wrong.**



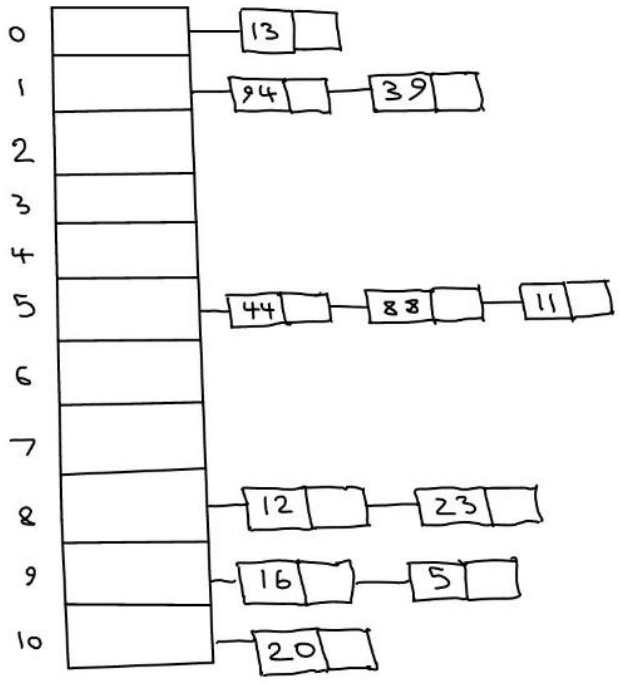
The pre-order traversal of this heap is {1,3,2,5,4,6,7}. Evidently, the keys are not listed in nondecreasing order.

**9. Hillary claims that a postorder traversal of a heap will list its keys in non-increasing order. Draw an example of a heap that proves her wrong.**



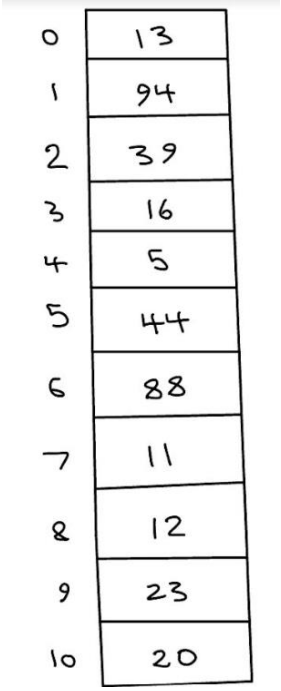

The post-order traversal of the heap is {4,5,2,6,7,3,1}. Therefore, the keys are not listed in non-increasing order.

**10. Draw the 11-entry hash table that results from using the hash function, h(i) = (3i + 5) mod 11, to hash the keys 12, 44, 13, 88, 23, 94, 11, 39, 20, 16, and 5, assuming collisions are handled by chaining.**

**11. What is the result of the previous exercise, assuming collisions are handled by linear probing?**

Since there are no indexes past 10, there are not enough empty buckets (indexes) to deal with the key values that have duplicate indexes within the hash table. So, when dealing with collisions with linear probing, some key values don't end up on the hash table.




**12. What is the worst-case running time for inserting n items into an initially empty hash table, where collisions are resolved by chaining? What is the best case?**

The worst case for putting n elements in an initially empty hash table by chaining is O(n). In the worst case, all elements have the same hash value so all are chained in the same spot. This occurs if no collisions occur and only simple operations are performed to add the values to the hash table. The best case is when there is no collision so for n elements, the insertion will be O(1). Only simple operations are performed to add the values to the hash table.