Assignement 3

PointerHeap and Sorting

**Part of the source code is included here, but all of it is in the zip folder*

Exercise 1

a. Create (write C++ code) PointerHeap class, which implements a Heap using pointers instead of using an array. Implement functions DeleteMin(…) and Insert(….). Use templates for type of object and Comparator to define comparison between two objects. Show an example of how you would use your class for an object of class IntCell (a class with just one integer member variable; hint: you need to implement Comparator for IntCell).
b. What is the big-Oh complexity of DeleteMin(…) and Insert(….)?

B. Hereunder is the whole code for Insert and DeleteMin functions:

```
template<typename T>
void HeapNode<T>::Insert(HeapNode* toInsert, HeapNode** root, HeapNode** las
t){

    //insert at the end
    if(*root == nullptr){
        *root = toInsert;
        *last = *root;
        return;
    }
    else if(*root == *last){
        (*root)->leftChild = toInsert;
        toInsert->parent = *root;
    }
    else if((*last)->parent->leftChild == *last){
        (*last)->parent->rightChild = toInsert;
        toInsert->parent = (*last)->parent;
    }
    else{
        if(!(*last)->parent->parent){
            HeapNode* temp = *root;
            while(temp->leftChild) temp = temp->leftChild;
            temp->leftChild = toInsert;
            toInsert->parent = temp;
        }
```

```
        else{
            if((*last)->parent->parent->leftChild == (*last)->parent){
                (*last)->parent->parent->rightChild->leftChild = toInsert;
                toInsert->parent = (*last)->parent->parent->rightChild;
            }
            else if((*last)->parent->parent->rightChild == (*last)->parent){
                HeapNode* temp = *last;
                while( temp->parent != *root && temp->parent-
>leftChild != temp) temp = temp->parent;
                if((*root)->rightChild == temp) temp = *root;
                else temp = temp->parent->rightChild;
                while(temp->leftChild) temp = temp->leftChild;
                temp->leftChild = toInsert;
                toInsert->parent = temp;
            }
        }
    }
    toInsert->previous = *last;
    *last = toInsert;

    // toggle up
    HeapNode* tup = *last;
    while(tup->parent != nullptr && tup->parent->value > tup->value){
          T temp = tup->value;
          tup->value = tup->parent->value;
          tup->parent->value = temp;
          tup = tup->parent;
    }
}
template<typename T>
void HeapNode<T>::DeleteMin(HeapNode** root, HeapNode** last){
    if(*root == nullptr) return;
    if(!(*root)->IsLeaf()){
        (*root)->value = (*last)->value;
        if((*last)->parent->leftChild == *last) (*last)->parent-
>leftChild = nullptr;
        else (*last)->parent->rightChild = nullptr;
        *last = (*last)->previous;
        this->trickleDown(*root);
    }
    else{
        *root = nullptr;
        *last = nullptr;
    }
```

For the $Insert()$ function, the first three branches are edge cases and move pointers to insert the newNode. Those operations are in done in $constant$ time. On the last else branch, we then have two cases: The first is when we are inserting at the second level of the three, meaning we don't have a grandparent to the node. In that case, we're traversing $O(log(n))$ left child nodes and insert the new Node. This yields a time complexity of $O(log(n))$ in that "worst case". The second branch (when we are below the second level of the three) has two cases, and similarly to the previous branch, the worst case is when our parent to the node we want to insert is a right child. In that case, we go up to the root and then find the place for our newNode in $O(log(n))$ time. The last part of our algorithm is trickling up the node we just inserted, and as we are going up one parent at a time, we also traverser $log(n)$ nodes. Finally, in the worst case, our $Insert()$ function has a $O(log(n))$ time complexity.

For the $DeleteMin()$ function, the time complexity is $constant$ if the Heap is empty or if we only have one leaf. If the root has children, the time complexity of the algorithm is entirely determined by the $trickleDown()$ function:

```
template<typename T>
void HeapNode<T>::trickleDown(HeapNode* root){
    HeapNode* temp = root;
    if(temp->leftChild){
        if(temp->leftChild->value < temp->value){
            T valt = temp->value;
            temp->value = temp->leftChild->value;
            temp->leftChild->value = valt;
            this->trickleDown(temp->leftChild);
        }
    }
    if(temp->rightChild){
        if(temp->rightChild->value < temp->value){
            T valt = temp->value;
            temp->value = temp->rightChild->value;
            temp->rightChild->value = valt;
            this->trickleDown(temp->rightChild);
        }
    }
}
```

For the two branches of the function, all the operations are made in $O(1)$ time, except for two instructions. The two recursive calls, on $temp-> leftChild$ and $temp-> rightChild$, make sure the node trickles down the right path, but since whatever branch we enter, we go down one level, we can't compare the $n\,nodes$ even in the worst case. So, the overall time complexity of the $DeleteMin()$ function stays at $O\big(log(n)\big)$ in the worst case.

Exercise 2

a. Implement functions for insertion sort, quicksort, heapsort and mergesort that input an array of integers and sort it.
b. Write a program that generates random integer arrays (hint: use seed appropriately to avoid generating same sequences) of lengths 10, 100, 1000, 10,000, 100,000, 1000,000, and then sorts each using each of the sorting functions from (a), and measures the time. The program will repeat this process 30 times and will compute the average execution time for each (arraysize,sorting-function) pair, over these 30 iterations. Finally, the program will output all these numbers in a readable format, e.g., as a table.
c. Are your computed numbers reasonable given your knowledge of the asymptotic complexity of each sorting algorithm? Explain.

Here are the numbers for each $< arraySize, sortingFunction >$ (in seconds):

<10, QuickSort>: 0	<10, HeapSort>: 0
<100, QuickSort>: 0	<100, HeapSort >: 0
<1000, QuickSort>: 8.81*e-005	<1000, HeapSort >: 0.000218233
<10000, QuickSort>: 0.00154577	<10000, HeapSort >: 0.0021039
<100000, QuickSort>: 0.0177466	<100000, HeapSort >: 0.0268195
<1000000, QuickSort>: 0.220972	<1000000, HeapSort >: 0.355106
<10, MergeSort>: 6.36*e-005	<10, InsertionSort>: 0
<100, MergeSort >: 0.000195133	<100, InsertionSort >: 0
<1000, MergeSort >: 0.00256353	<1000, InsertionSort >: 3.25333*e-005
<10000, MergeSort >: 0.0272044	<10000, InsertionSort >: 0.00256677
<100000, MergeSort >: 0.26379	<100000, InsertionSort >: 0.23696
<1000000, MergeSort >: 5.19896	<1000000, InsertionSort >: 25.944

The theorical time complexity for MergeSort, QuickSort and HeapSort is $O(n * log(n))$ with $O(n^2)$ as worst case for QuickSort. Insertion sort in $O(n)$ in best case and $O(n^2)$ in worst and average case.

The first thing to notice is that all the algorithms except Merge Sort have 0 secs time for 10 and 100 values. This is because the clock measuring the time elapsed has probably too little of an interval to measure a time for those sorts. Realistically, the time approaches 0 but is not 0. Comparing HeapSort and QuickSort which should run in $O(n * log(n))$ in the average case, we can see that they grow at the same rate (QuickSort being a little bit faster), and that for $n = 1000$, the time for HeapSort is 2.18233*e-004, and for $n = 1000 * 10 = 10000$ the time taken is 2.1039*e-003 seconds which corresponds to a increase by $10 * log(10)$ from the previous case. It means that the numbers for QuickSort and HeapSort are quite reprensetative of the theoretical analysis. It's the same for insertion sort, when the input n is multiplied by 10, the time taken is multiplied by 10^2, hence the $O(n^2)$ time complexity. The only big discrepancy is for MergeSort where, the relation between the sizes of n seems to be $n * log(n)$, but the overall time taken is much larger than QuickSort and HeapSort. That may be because the algorithm for MergeSort here is not in-place and allocating memory each time while splitting the array in half results in a space complexity of $O(n)$. That might influence the time taken for the function to sort and is quite different from the theoretical viewpoint.