Assignment1

Sorin Macaluso

October 2023

1 Note

The code with all comments is in the appendix. Latex did not like single quotes so for some of my returns just took them out and added a comment for the data type. Also all cout's had to be commented, latex did not know what to do with them.

2 Node Class/Stack & Queue

```
1  //stack class
2  struct Node{
3      char data;
4      Node* link;
5  };
6
7  //Queue Class
8  Node* front;
9  Node* back;
10
11 Queue(){
12     front = NULL;
13     back =  NULL;
14 }
```

2.1 Node/Stack

Line two and three in the code above show the two data values in the class called Node. I choose the data type char and Node* link because, we want one item to be the data (or character we are checking below) and one to be a link to the next (so it would be of Node* class, * being the c++ syntax for pointer value).We will use a different version of this class in order to make a Queue.

2.2 Queue

In the Queue version we have both a front (head) and a back (tail) of the singly linked list. Queues use a head and a tail because they are first in first out. So what every is the first thing added is the thing that will be taken out first (in order). While a stack is first in last out. What ever is the first thing to be pushed onto the stack is the last thing to come out (reverse order).

2.3 Creation

Next I will show how you can utilizes this small class in order to make a stack and a queue. First thing we have to do is initialize the head node so that we always know where the top is. This will allow us to have O(1) time on the Pop and Push function.

```
//stack start
Node* head;
head = new Node;

```

```
//Queue start
Queue queue;
```

2.4 Stack: Push & Queue: EnQueue

Then after the head is created we start to populate the stack by ”pushing” values onto the stack. Then we put everything in queue on the queue. See lines 2 & 3 of the code bellow.

```
1  for(int k = 0; k < PalanCheck.length(); k++){
2      push(&head, PalanCheck[k]);
3      queue.EnQueue(PalanCheck[k]);
4  }
```

Lets take a closer look at the push and EnQueue functions above.

```
//Push
void push(Node** front, char key) {

    front to the pushed value)
    Node* newFront = new Node;

    newFront->data = key;
    newFront->link = *front;
    *front = newFront;
}

//EnQueue
void EnQueue(char info){

    Node* temp = new Node;

    temp->data =  info;
        if(back == NULL){
            front = temp;
            back = temp;
            return;
        }

        back->link = temp;
        back = temp;
}
```

2.5 Stack: Push

Push:

Push takes in a pointer value to the front and the character that we are adding to the data. Makes a new front (since this new value will now become the front, REVERSE). Assigns the key to the data value and the link to the pointer of front. Then makes the pointer for front the new front value which is the new thing we added.

1 , 2 , 3 , 4

Turn into

4 , 3 , 2 , 1

(As we add a new thing its getting pushed to the front)

2.6 DeQueue: EnQueue

For Queue all we do is pass the info in that we want added. Make a temp Node so that we do not break anything in the original and also helps for swapping. Then we make sure the back == NULL to check if this is the first thing added to the list. If not make the link for the back equal to the temp and set it equal to temp.

1 , 2 , 3 , 4

Turn into

1 , 2 , 3 , 4

(As we add a new thing its getting pushed to the back (like a line) in order)

2.7 Stack: Pop & Queue: DeQueue

Getting the data out is pretty much just the reverse of all of this. We do this with a Pop for stacks and a DeQueue for Queues.

```
//Pop
if(front){
    Node* temp =  front;
    char data =  front->data;
    front = front->link;

    return data;
}
return 0;


//DeQueue
if (front == NULL) {
    return 0;
}

Node* temp = front;
front = front->link;
char data =  temp->data;
delete temp;
if(front == NULL){
    back = NULL;
}

return data;
```

2.8 Stack: Pop

For Pop all we do is take the front pointer and make a temp value. We don't want to actually delete it in memory just want to grab the data. We assign the data at that point to a value, reassign the link of front to the next link and then return back the data to be used. To check we can use the 0 as the bottom. If we return 0 its either the end or we know we got wrong data.

2.9 Queue: DeQueue

For DeQueue we first check to see if the front is NULL or we are done going thought the list. If so we get a 0 as the return. Otherwise same idea as Pop we get a temp change the front to the link of the front we are checking. Take the data for the temp (which is now the item we are getting info from) and return it back. We also make sure the back == NULL to check if this is the first thing added to the list.

3 Stack & Queue isEmpty

```
//Check if empty or not
bool EmptyOrNot = false;
Node* temp = head;
Node* CheckLink = temp->link;

if (CheckLink == NULL){
    EmptyOrNot = true;
} else{
    EmptyOrNot = false;
}

return EmptyOrNot;
```

3.1 Stack & Queue: IsEmpty Function

This is a simple function that just looks at the head of either the stack or the queue and checks if the link value is NULL or not. If the link value is NULL we can assume the linked list is empty therefor returning true if not then return false because the linked list is not empty.

3.2 Data Structures Chart

Data Structure	Appending	Time Complexity
Stack (Pop)	Instant	O(1)
Stack (Push)	Instant	O(1)
Queue (EnQueue)	Instant	O(1)
Queue(DeQueue)	Instant	O(1)

These are all O(1) because for the stack we know where the head of the stack is at all time so we can add to the front at instant time. We can also take from the top in instant time.

While for a Queue we can add to back because we know the back of the Queue at all times so we can add to there instantly. We can also take from the top of the Queue since we know what the head of the Queue is instantly.

4 Selection Sort

```
int numOfComp = 0;

int arrayLength = size;

for(int i = 0; i < arrayLength - 2; i++){

    int smallestNum = i;

    for(int k = i + 1; k < arrayLength; k++){

        onces
        numOfComp++;

        if (selectionSortsArray[k] < selectionSortsArray[smallestNum]){
            smallestNum = k;
```

```
            }
        }

        if(smallestNum != i){
            string temp = selectionSortsArray[i];
            selectionSortsArray[i] = selectionSortsArray[smallestNum];
            selectionSortsArray[smallestNum] = temp;
        }
    }
```

4.1 Selection Sort

Selection sort is a sort that is very simple to implement into any code but in terms of time complexity it is not the best. Selection Sort starts at the first item and goes through the whole list for the place that it is at. Checking at each place on the array if the value its on is less than the checked value. If so make that the smallest thing and then at the end do a swap to put the smallest thing in the right spot. The time complexity on this is $O(n^2)$. This is shown in the double for loop. First the selection sort will pick the value its on and then compare that to all the values of the array. Making it $n * n$ amount of comparisons. We do not factor in the if and swap since those are done instantly in constant time O(1).

4.2 Selection Sort: Amount of Comparisons & Time Complexity

Then number of comparisons for this type of sort is 221444. This makes sense since there are 666 thing in the magic items list so $O(n^2)$ for the array size of 666 would be 443,556. My answer is about $O(1/2(n^2))$ so it is valid to say that $O(n^2)$ is the time complexity for selection sort.

5 Insert Sort

```
    int numOfComp = 0;

    for(int i = 1; i < size; i++){
        string comparisonValue = insertSortArray[i];
        int k = i - 1;

        while(k >= 0 &&  insertSortArray[k] > comparisonValue){
            insertSortArray[k + 1] = insertSortArray[k];
            k = k - 1;
            numOfComp++;
        }
        insertSortArray[k + 1] = comparisonValue;
    }
    numOfComp++;
```

5.1 Insert Sort

Unlike Selection Sort Insert Sort goes from the second item and checks everything else. This is different than Selection Sort because Selection sort says that the last thing in the array is size 1 so it has to be sorted already. On the other hand Insert sort says that the front first thing is in a array of size one and already sorted, so it continues from there.

With the fact mentioned before Insert Sort acts differently in order to sort the array. Rather than it start from the front and go through he whole array. Insert Sort more works back words, it goes through everything behind it to check and then places the item where it should be behind the next value it is going to check.

5.2 Insert Sort: Amount of Comparisons & Time Complexity

Then number of comparisons for this type of sort is 106631. This makes sense since there are 666 thing in the magic items list so O(n^2) for the array size of 666 would be 443,556. My answer is about O(1/4(n^2)) so it is valid to say that O(n^2) is the time complexity for selection sort.

6 Merge Sort

```
//mergeing
int leftSize = middle - start + 1;
int rightSize = end - middle;

string* tempLeftSide = new string[leftSize];
string* tempRightSide = new string[rightSize];

for(int i = 0; i < leftSize; i++){
    tempLeftSide[i] = MergedArray[start + i];
}

for(int k = 0; k < rightSize; k++){
    tempRightSide[k] = MergedArray[middle + 1 + k];
}

int left = 0;
int right = 0;
int mergePlace = start;

while(left < leftSize && right < rightSize){

    numOfComp++;

    if(tempLeftSide[left] <= tempRightSide[right]){
        MergedArray[mergePlace] = tempLeftSide[left];
        left++;
    } else{
        MergedArray[mergePlace] = tempRightSide[right];
        right++;
    }
    mergePlace++;
}

while(left < leftSize){
    MergedArray[mergePlace] = tempLeftSide[left];
    left++;
    mergePlace++;
    numOfComp++;
}

while(right < rightSize){
    MergedArray[mergePlace] = tempRightSide[right];
    right++;
    mergePlace++;
    numOfComp++;
}

delete[] tempLeftSide;
delete[] tempRightSide;

```

```
    //recusion for MergeSort
    if(start < end){

        int middle = start + (end - start) / 2;

        MergeSort(mergeSortArray, start, middle);
        MergeSort(mergeSortArray, middle+1, end);

        Merge(mergeSortArray, start, end, middle);

    }

    return numOfComp;
```

6.1 Merge Sort: Part 1

In Merge sort there are two functions that we have to discuss in order to have a full explanation of the Merge sort functions. The first function to discuss is the one with the comment MergeSort. This function is the way that the recursion starts. It recalls the same MergeSort function with different start and end value so that we are ”splitting” the array, the quotes are because we are not really breaking the array more just working on smaller and smaller spots of the array. It is being split on the array index. Specifically the start to middle and the middle+1 to the end, we do middle+1 since the middle is already accounted for in the first recursive call.Finally the Merge part is the other function that will actually do the sorting.

6.2 Merge Sort: Part 2

The Merge function works by making temp arrays to have all the sorted parts. The arrays are the size of what the max left (start) and max right (end). Makes everything for the left and the right side to be in these temp array to be manipulated without touching the really array. Then there are place value, left, right, and mergePlace. These are used in the while loop. The while loop will go as the place is less than the actual value for both the left and the right. Then we re-add the now sorted array to the array we want sorted and delete the temp values.

6.3 Merge Sort: Amount of Comparisons & Time Complexity

Merge Sort is of time complexity O($n * log(n)$) because we have to do the splitting $nlog(n)$ and then we are still comparing at least everything in the array once after all the splitting is done, so n. Then number of comparisons for this type of sort is 6302. This makes sense since there are 666 thing in the magic items list so O($n * log(n)$) for the array size of 666 would be 5994. My answer is about O($1.1(n * log(n))$) so it is valid to say that O($n * log(n)$) is the time complexity for selection sort.

7 Quick Sort

```
    //partition
    string pivotValue = pivotArray[end];
    int place = start;

    for(int i  = start; i < end; i++){
        numOfCompQuick++;

        if(pivotArray[i] < pivotValue){
            swap(pivotArray[place], pivotArray[i]);
            place++;
        }
    }

    swap(pivotArray[place], pivotArray[end]);
    return place;
```

```
16
17      //recusion for QuickSort
18      if(start < end){
19          int pivotPoint = Partition(quickSortArray, start, end);
20
21          QuickSort(quickSortArray, start, pivotPoint-1);
22          QuickSort(quickSortArray, pivotPoint+1, end);
23
24      }
25
26      return numOfCompQuick;
```

7.1 Quick Sort: Part 1

In Quick sort there are two functions that we have to discuss in order to have a full explanation of the Quick sort functions. Unlike Merge sort the recusion start after the sorting. Though like Merge sort we are giving it a start to pivotPoint-1 and pivotPoint+1 to the end to split it. In Merge sort we acounted for the middle value while in Quick Sort we use it as a pivot point and nothing more. The first function to discuss is the function called Partition on line 19. This is how Quick Sort sorts the function as we are breaking the array into its smaller parts. Simply all it does is say if the value you are checking for this sub array is less than the pivot then move it to the left of pivot if not move it to right of pivot.

7.2 Quick Sort: Part 2

As mentioned above quick sort start by getting a partition value and separating all the value either left or right of that value. Then it will start splitting the array into it smaller parts. This is the recursive part of the sort. This is handled on lines 21 & 22. These two lines handle the recursion in the same way that Merge does just using a partition value instead of the middle. Although the pivot point in nothing more than that we do not factor it into the placing like we do with Merge sort, where middle was a value used to determine place.

7.3 Quick Sort: Amount of Comparisons & Time Complexity

Merge Sort is of time complexity O($n * log(n)$) because we have to do the splitting $nlog(n)$ and then we are still comparing at least everything in the array once as we split the array, so n. Then number of comparisons for this type of sort is 7022. This makes sense since there are 666 thing in the magic items list so O($n * log(n)$) for the array size of 666 would be 5994. My answer is about O(1.2($n * log(n)$)) so it is valid to say that O($n * log(n)$) is the time complexity for selection sort.

8 Comparisons Chart

Sort	Comparisons	Time Complexity
Selection	221444	O(1/2(n^2))
Insert	106631	O(1/4(n^2))
Merge	6302	O(1.1($n * log(n)$))
Quick	7022	O(1.2($n * log(n)$))

See; 3.2, 4.2, 5.3, and 6.3 section for the full explanation of complexity and how the number of comparisons relate to time complexity.

9 References

"Insertion Sort - Data Structure and Algorithm Tutorials." GeeksforGeeks, GeeksforGeeks, 31 May 2023, www.geeksforgeeks.org/insertion-sort/.

"Introduction to Stack - Data Structure and Algorithm Tutorials." GeeksforGeeks, GeeksforGeeks, 23 Mar. 2023, www.geeksforgeeks.org/introduction-to-stack-data-structure-and-algorithm-tutorials/.

"Merge Sort - Data Structure and Algorithms Tutorials." GeeksforGeeks, GeeksforGeeks, 6 July 2023, www.geeksforgeeks.org/merge-sort/.

"Queue - Linked List Implementation." GeeksforGeeks, GeeksforGeeks, 12 Jan. 2023, www.geeksforgeeks.org/queue-linked-list-implementation/.

Pretty much for all of these I like to use them because they usually give a really simple visual way of explaining how the code works. They also add other details like disadvantages and advantages, applications, as well as the time complexity for the sort or data structure. At the beginning of this lab I did a lot of drawing out the diagrams to help me understand. The helped to fill the mistakes that I made with my diagrams. Also the insert sort link had a FAQ section that branched into other interesting algorithms topics like stable sorting algorithms.

10 Appendix

10.1 Node/Stack/Queue

```cpp
//librarys that are always used in c++
#include <iostream>
using namespace std;

//This class will be the nodes that are made to make the link list
//The data type for data is char since there will characters passed to make the list
struct Node{
    char data;
    Node* link;
};

//push asks for a pointer to a pointer of the front and the key (value) that you want to add
void push(Node** front, char key) {

    //mks a new node pointer call newFront (becasue it will be use to reassign the front to the pushed value)
    Node* newFront = new Node;

    /*assigns the newFront the passed in key (the new data)
    then assigns the link to be the pointer value to the front
    then the pointer to front is equal to the newFront makeing is back at the top*/

    /* Diagram:
        (new front) new thing -> newFront*
            (new front) new thing -> to front**
                                  front -> first one***
    */


    newFront->data = key;
    newFront->link = *front;
    *front = newFront;




}

//pop the first iteam in the stack out of the stack
char pop(Node*& front) {
    //checks if the front is NULL or not
    if(front){
```

```

        //mks a temp variable
        Node* temp =  front;
        //gets the data
        char data =  front->data;
        //make front = to next thing
        front = front->link;

        //delets all that not used and returns the data to be used
        return data;
    }

    /* Diagram:

        front1 -> link* (takes the data from here)
            thing2 -> link** (makes this the new head)
                thing3 -> link***

        take the data from front and returns that and then makes the head what ever is
    below it
    */

   return 0 //string value;
}


//checks if the list is empty
bool isEmptyStack(Node*& head) {
    //value to be returned back
    bool EmptyOrNot = false;

    //mks a temp node value
    Node* temp = head;

    //assign the value of temp (head) to CheckLink
    Node* CheckLink = temp->link;

    //if null end of list (empty) if not not end of list still full
    if (CheckLink == NULL){
        EmptyOrNot = true;
    } else{
        EmptyOrNot = false;
    }

    //returns bool answer
    return EmptyOrNot;
}

struct Queue{
    //Node pointer for the front and back
    Node* front;
    Node* back;

    Queue(){
        front = NULL;
        back =  NULL;
    }

    //put the item of choice one after the last slot
    void EnQueue(char info){
```

```
    //creates a temp Node pointer
    Node* temp = new Node;

    //sets the data to the temp nodes data
    temp->data =  info;
        //checks if back is null
        if(back == NULL){
            front = temp;
            back = temp;
            return;
        }

        //sets the back link to temp and then back to equal temp
        back->link = temp;
        back = temp;
}

/* Diagram:
    head -> link
    tail -> link (new iteam)
    iteam -> link (becomes new tail) ...
*/

char DeQueue(){
    //checks if front is null or at the top
    if (front == NULL) {
        return 0 //string value;
    }

    //creates a temp node pointer
    Node* temp = front;

    //sets the front  variable to the link of front
    front = front->link;

    //set data of char type to the temp->datas data
    char data =  temp->data;

    //delets temp since not needed anymore
    delete temp;

    //makes it all null
    if(front == NULL){
        back = NULL;
    }

    //returns the data
    return data;

}

/* Diagram:
    head -> link
    tail -> link (new iteam)
    iteam -> link (becomes new tail) ...

    same thing as above but in reverse

    head -> link (takes out this item)
```

```
        item -> link (this becomes new head)
        tail -> link

    */
};


//checks if the Queue is empty
bool isEmptyQueue(Node*& head) {
    //value to be returned back
    bool EmptyOrNot = false;

    //mks a temp node value
    Node* temp = head;

    //assign the value of temp (head) to CheckLink
    Node* CheckLink = temp->link;

    //if null end of list (empty) if not not end of list still full
    if (CheckLink == NULL){
        EmptyOrNot = true;
    } else{
        EmptyOrNot = false;
    }

    //returns bool answer
    return EmptyOrNot;
}
```

10.2 Shuffle

```
#include <iostream>

using namespace std;

//takes in a array and shuffles it useing a random number
void shuffle(string* shuffleArray){
    //gets a random number from 0 to 665 since array size is 666
    for (int i = 0; i < 666; i++){
        //does the swap of the two things
        //because i do not give it a random seed as well the random will be the same
            random everytime
        int randNum = rand() % 665 + 0;
        string temp = shuffleArray[i];
        shuffleArray[i] = shuffleArray[randNum];
        shuffleArray[randNum] = temp;
    }
};
```

10.3 Sorts

```
#include <iostream> //object oriented library that allows input and output using
    streams

using namespace std;

int numOfComp = 0;
int paritionPlace = 0;
```

```cpp
int numOfCompQuick = 0;

//takes in a array uses selection sort to sort it
void selectionSort(string* selectionSortsArray, int size){

    //counts the number of comparisons
    int numOfComp = 0;

    //has the array size
    int arrayLength = size;

    //goes through the array from start to the second to last one
    //if there is one thing left than it is sorted
    for(int i = 0; i < arrayLength - 2; i++){

        //starts by saying the smallest thing is what ever is in the i'th place
        int smallestNum = i;

        //will start making comparisons for everything after the i'th place
        for(int k = i + 1; k < arrayLength; k++){

            //counts the number of comparisons
            //put it here since it will have to check everything in the array at least onces
            numOfComp++;

            //if smaller givea k to the smallestNum to variable
            if (selectionSortsArray[k] < selectionSortsArray[smallestNum]){
                smallestNum = k;
            }
        }

        if(smallestNum != i){
            //performs the swap
            string temp = selectionSortsArray[i];
            selectionSortsArray[i] = selectionSortsArray[smallestNum];
            selectionSortsArray[smallestNum] = temp;
        }
    }

    //prints out the number of comparisons
    //cout << "\n" << "Selection Sort's number of comparisons is: " << numOfComp << "\n";

};

//takes in a array uses insert sort to sort it
void insertSort(string* insertSortArray, int size){

    //counts the number of comparisons
    int numOfComp = 0;

    //does not start at one since a iteam with one thing in it is sorted
    for(int i = 1; i < size; i++){

        //take the string at that spot in the array and uses that to check the rest
        string comparisonValue = insertSortArray[i];
        int k = i - 1;

        //goes through until the value at k is less than comparison value
```

```
        while(k >= 0 &&  insertSortArray[k] > comparisonValue){
            //swap pt.1
            insertSortArray[k + 1] = insertSortArray[k];
            k = k - 1;
            //the while loop will always be entered if the num is greater than 0 or equal to zero or the k'th value is smaller than comparison
            //so that means that something was checked so but the value here
            numOfComp++;
        }
        //swap pt.2
        insertSortArray[k + 1] = comparisonValue;
    }

    //prints out the number of comparisons
    //cout << "\n" << "Insert Sort's number of comparisons is: " << numOfComp << "\n";

    numOfComp++;

};

//does the mergeing back at the end

void Merge(string* MergedArray, int start, int end, int middle) {

    //creats a left side and a right side
    int leftSize = middle - start + 1;
    int rightSize = end - middle;

    //temp string arrays to handle the storing of the left and the right value
    string* tempLeftSide = new string[leftSize];
    string* tempRightSide = new string[rightSize];

    //copy the data to temp arrays to manipulate for sorting (left)
    for(int i = 0; i < leftSize; i++){
        tempLeftSide[i] = MergedArray[start + i];
    }

    //copy the data to temp arrays to manipulate for sorting (right)
    for(int k = 0; k < rightSize; k++){
        tempRightSide[k] = MergedArray[middle + 1 + k];
    }

    //place holder variables to tell where you are at
    int left = 0;
    int right = 0;
    int mergePlace = start;

    //comparison to tell what is left and right
    //does it for time of place value < thand the actual size
    while(left < leftSize && right < rightSize){

        numOfComp++;

        //if the left is lessthan or equal to the right but is on the left side
        if(tempLeftSide[left] <= tempRightSide[right]){
            MergedArray[mergePlace] = tempLeftSide[left];
            left++;
        //else has to go on the right side
        } else{
            MergedArray[mergePlace] = tempRightSide[right];
```

```
            right++;
        }
        //advance foward on the place you are checking
        mergePlace++;
    }

    //copy the rest of the left values over
    while(left < leftSize){
        MergedArray[mergePlace] = tempLeftSide[left];
        left++;
        mergePlace++;
        numOfComp++;
    }

    //copy the rest of the right values over
    while(right < rightSize){
        MergedArray[mergePlace] = tempRightSide[right];
        right++;
        mergePlace++;
        numOfComp++;
    }

    //deletes the temp values
    delete[] tempLeftSide;
    delete[] tempRightSide;
}


//takes in a array uses merge sort to sort it
int MergeSort(string* mergeSortArray, int start, int end){

    //uses recursion to set up a sort of queue that breaks down the total array to single values
    //then when the reusion is done it brings it all back to the the total array and sorts it along the way
    if(start < end){

        //creates a accurate middle
        int middle = start + (end - start) / 2;

        //sets up the recursion
        MergeSort(mergeSortArray, start, middle);
        MergeSort(mergeSortArray, middle+1, end);

        //when all done starts the merging back
        Merge(mergeSortArray, start, end, middle);

    }

    return numOfComp;

}

int Partition(string* pivotArray, int start, int end){
    //create the pivot value at what is at the middle of the parted array
    string pivotValue = pivotArray[end];

    //to keep track of what place you are at
    int place = start;

```

```
    //goes from the start of the sub array to the end
    for(int i  = start; i < end; i++){

        //gives the num of comparisons
        numOfCompQuick++;

        //checks to see if the ith value is less than the end
        if(pivotArray[i] < pivotValue){

            //if so swap and make place move over
            swap(pivotArray[place], pivotArray[i]);
            place++;
        }
    }

    //swap the place with the end to make sure all is well
    swap(pivotArray[place], pivotArray[end]);

    //return the place you are at to become the new end for the rest
    return place;

}


//takes in a array uses quick sort to sort it
int QuickSort(string* quickSortArray, int start, int end){

    //creates the condition to make it start calling
    if(start < end){

        //creates a accurate pivotPoint (middle)
        int pivotPoint = Partition(quickSortArray, start, end);

        //sets up the recursion
        QuickSort(quickSortArray, start, pivotPoint-1);
        QuickSort(quickSortArray, pivotPoint+1, end);

    }

    //for comparisons
    return numOfCompQuick;

}
```