Assignment One

Collin Drake

Collin.Drake1@Marist.edu

October 6, 2023

1 Linked List

1.1 The Data Structure

A linked list is a collection of objects that are linked together by pointers to addresses in memory. The objects, or nodes, of a linked list contain two fields: the data stored within the node, and the memory address of the next node in the list. A linked list can have either one or two entry points, denoted as the head or tail, which allow access from the beginning or end of the list. A linked list is an elementary data structure that can be used to build other fundamental data structures and algorithms. In this case, we will use the basic characteristics of a linked list to create a stack and queue. Therefore, we will only need one aspect from a linked list, the node class, to create these data structures. The node class creates the nodes of a data structure by taking in data and pairing it with a null pointer.

Node Class

```
1    #include "Node.hpp"
2    #include <stdexcept>
3
4    Node::Node(char val){ //Node class constructor
5        this->val = val;
6        this->next = nullptr;
7    }
```

1.2 Asymptotic Analysis

Given a linked list and assuming that there is a pointer to the head, there are four key operations that can be performed: accessing an element, adding an element to the head, adding an element at any point, and removing an element from any point. Each of these operations can be characterized by its time complexity. When it comes to accessing an element in a linked list, the worst-case scenario would be linear time, O(n), as the element could be located at the end, forcing you to iterate through the list. Adding an element to the head would be constant time, O(1) since we have direct access to the head. However, insertion or removal from any other point within the list signifies the time complexity to be linear, or O(n).

2 Stack

2.1 The Data Structure

A stack is a collection of objects that follows the LIFO (Last in First Out) principle. Initially, you can think of a stack as a vertical singly linked list where elements can only be added and removed from the head or top of the stack. An example of a stack would be a stack of plates, where plates are added and removed from the top of the stack. There are three fundamental operations that make up a stack: push, pop, and isEmpty.

Stack Constructor

```
1    Stack::Stack() { //Stack class constructor
2        top = nullptr;
3    }
```

2.1.1 Push

The push function of a stack is used to push or add an element to the top of a stack. This function takes one parameter, the data, creates a node, and then adds it to the top of the stack. In this process, if the stack is empty, the new object becomes the top of the stack. However, if the stack already contains elements, the new element is still positioned at the top, but this action requires a change in memory pointers. More specifically, the pointer for the current top element and the next pointer for the new element must be arranged in order for the new element to become the top, and for the top to become the next node.

Push

```
1    void Stack::push(char val) { // push node onto stack
2        Node* newNode = new Node(val);
3        if(isEmpty()){
4            top = newNode;
5        }
6        else{
7            newNode->next = top;
8            top = newNode;
9            newNode = nullptr;
10           delete newNode;
11       }
12   }
```

2.1.2 Pop

The pop function of a stack is used to remove the top element of a stack. It takes no parameters, but instead returns the data contained in the recently popped node. When performing a pop operation, there are two important actions that must be taken: the first one is to ensure we are not attempting to remove a non-existent node, and the second one is to properly reassign the pointers for both the new top element and the recently removed element.

Pop

```
1     char Stack::pop(){ //pop top node off stack
2         if (isEmpty()){
3             throw invalid_argument("The␣stack␣is␣empty");
4         }
5         else{
6             Node* temp = top;
7             top = top->next;
8             temp->next = nullptr;
9             return temp->val;
10            temp = nullptr;
11            delete temp;
12        }
13    }
```

2.1.3 isEmpty

The isEmpty function of a stack simply checks if a Stack is empty and returns true or false. It does this by checking if the top of the stack is equal to null, or if it contains an element.

isEmpty

```
1     bool Stack::isEmpty(){ //check if stack is empty
2         return top == nullptr;
3     }
```

2.2 Asymptotic Analysis

A stack maintains a consistent time complexity for all of its operations. When looking at the time complexity of the stack functions: push, pop, and isEmpty, it's easy to determine each time complexity since they all involve minimal calculations. When calling the push function to add a new element to the top of a stack, this operation runs in constant time, denoted as O(1), because we can only add elements to the top of the stack. Similarly, the pop function also runs in constant time, O(1), because it involves removing elements from the top of the stack. The isEmpty function also runs in constant time since it only performs a comparison. The time complexities of the stack functions all run in constant time, O(1), regardless of the size of the stack.

3 Queue

3.1 The Data Structure

A queue is a data structure that follows the FIFO (First in First Out) principle, meaning that the first element in the list is the first to be removed. A queue shares some similarities to a stack, as it is a collection of nodes, but it operates under different features and rules. You can visualize a queue as a real-life lunch line. Students stand in line, and the first students in the line are the first to get their food and the first to leave. Those who arrive later are added to the end of the line and have to wait their turn. A queue typically supports three main functions: enqueue, dequeue, and isEmpty.

Queue Constructor

```
1     Queue::Queue() { //Queue class constructor
2         head = nullptr;
3         tail = nullptr;
4     }
```

3.1.1 Enqueue

The enqueue function of a queue adds or enqueues nodes to the end of the queue. This function takes in one parameter, the data, creates a new node, and adds it to the tail or end of the queue. When adding an element to a queue, if there are no other nodes in the list, the newly added element becomes the head and tail. However, if the queue is already populated, the new node is added to the tail, forcing a change of memory pointers. More specifically, the tail pointer of the queue must be updated to point to this new node, and the next pointer to the previous tail.

Enqueue

```
1     void Queue::enqueue(char val){ //add node to back of line
2         Node* newNode2 = new Node(val);
3         if(isEmpty()){ //if empty make node head and tail
4             head = newNode2;
5             tail = newNode2;
6         }
7         else{ //if not make it rear
8             tail->next = newNode2;
9             tail = newNode2;
10            newNode2 = nullptr;
11            delete newNode2;
12        }
13    }
```

3.1.2 Dequeue

The dequeue function of a queue removes, or dequeues a node from a queue. This function accepts no parameters, but returns the data of the removed node. When dequeuing a node, we remove it from the head of the list and rearrange the pointers. After removing the current head, the next node becomes the new head, and the next pointer is updated to point to the node following it.

Dequeue

```
char Queue::dequeue(){ //remove node from front of line
    if(isEmpty()){
        throw invalid_argument("The queue is empty");
    }
    else{
        Node* temp = head;
        head = head->next; //change front
        if(head == nullptr){
            tail = nullptr;
        }
        return temp->val;
        temp = nullptr;
        delete temp;
    }
}
```

3.1.3 isEmpty

The isEmpty function for a queue is a straightforward check to see if a queue is empty or not. The function returns true or false, indicating whether there are any elements in the queue or not. When checking if a queue is empty, the function checks if the head points to null or a node.

Queue isEmpty

```
bool Queue::isEmpty(){ //check if queue is empty
    return head == nullptr;
}
```

3.2 Asymptotic Analysis

Given a standard queue with pointers to the head and tail, the operations are enqueue, dequeue, and isEmpty. Each of these operations has to have clearly defined time complexities of constant time, O(1). The reason these queue operations run in constant time is because each addition to the tail of a queue or removal from the head of a queue only involve minor calculations. Additionally, the isEmpty operation is a simple comparison, which also executes in constant time, O(1).

4 Selection Sort

4.1 The Algorithm

The algorithm selection sort is an algorithm used to sort a collection of items in ascending order. Selection sort works by iterating left to right and dividing the collection into two sections: a sorted section on the left, and an unsorted section on the right. During each iteration of selection sort, it finds the smallest element in the unsorted section and swaps it with the first element in that same section, effectively extending the sorted section further right. Selection sort continues to iterate and compare through the unsorted section until the entire collection becomes sorted.

Selection Sort

```
1     void Sorts:: selectionSort(vector<string>& magicItems){ //sort magic items
2         int sComparisons = 0; //comparisons counter for selection sort
3         for(int i = 0; i < magicItems.size(); i++){ //iterate through vector
4             int smallestPos = i; //smallest index
5             for(int j = i + 1; j < magicItems.size(); j++){ //increment and compare elements
6                 sComparisons++;
7                 if(magicItems[smallestPos].compare(magicItems[j]) > 0){
8                     //if smaller change smallest
9                     smallestPos = j;
10                }
11            }
12            //swap using temp variable
13            string temp = magicItems[i];
14            magicItems[i] = magicItems[smallestPos];
15            magicItems[smallestPos] = temp;
16        }
17        cout<<"Selection␣sort␣comparisons:␣" << sComparisons << "\n";
18    }
```

4.2 Asymptotic Analysis

When sorting an unsorted collection of elements using selection sort, the worst-case time complexity will be $O(n^2)$. The reason for this arises from the nested for-loops used within the algorithm. In terms of time complexity: basic calculations, assignments, and comparisons generally run in constant time, O(1). However, given that selection sort employs a nested loop (two for loops, one nested in the other), the function is forced to run in quadratic time. This signifies that as the input size increases, the time required to complete selection sort grows quadratically.

5 Insertion Sort

5.1 The Algorithm

Insertion sort is a sorting algorithm that shares some similarities with selection sort, but is overall different. Instead of comparing all elements to the right of the sorted section, insertion sort focuses on comparing each element in the unsorted section with all of the elements in the sorted section. The algorithm initially starts with one sorted element while all the others are unsorted. Insertion sort iterates through the unsorted section and inserts each unsorted element into its proper place in the sorted section until all elements are deemed sorted.

Insertion Sort

```
1     void Sorts:: insertionSort(vector<string>& magicItems){ //sort magic items
2         int iComparisons = 0;
3         for(int i = 1; i < magicItems.size(); i++){ //first element sorted
4             string current = magicItems[i];
5             int j = i - 1; //check with previous element
6             while(j >= 0 && current.compare(magicItems[j]) < 0){
7                 iComparisons++;
8                 magicItems[j + 1] = magicItems[j];
9                 j--;
10            }
11            magicItems[j + 1] = current;
12        }
13        cout<<"Insertion␣sort␣comparisons:␣" << iComparisons << "\n";
14    }
```

5.2 Asymptotic Analysis

When sorting an unsorted collection of elements using insertion sort the worst-case time complexity, similar to selection sort, will be $O(n^2)$. The quadratic time complexity is primarily due to the fact that insertion sort deploys a nested for-loop structure. This structure causes the algorithm to make multiple loops of varying lengths through an n range of elements. As a result, the efficiency of this algorithm decreases as the input size increases.

6 Merge Sort

The merge sort algorithm is a sorting method that uses the divide and conquer technique to sort a collection of elements. Merge sort sorts a collection of elements with two main functions: mergeSort and merge. The mergeSort function deploys the divide technique, dividing the collection into smaller sub-arrays by calling itself recursively. This recursive call continues until each element is in its own individual sub-array. On the contrary, the merge function deploys the conquer technique, merging these individual collections and sorting them in the process. This merging process continues until all the sub-collections are merged and sorted into a large collection of sorted objects.

6.1 The Algorithm

Merge Sort

```
1   void Sorts:: mergeSort(vector<string>& magicItems, int start, int end, int& comparisons){
2       if(start >= end){
3           return;
4       }
5       int middle = (start + end) / 2; //find middle point
6       mergeSort(magicItems, start, middle, comparisons); //sort left
7       mergeSort(magicItems, middle + 1, end, comparisons); //sort right
8       merge(magicItems, start, middle, end, comparisons); //merge sorted arrays
9   }
```

Merge

```
1   void Sorts:: merge(vector<string>& magicItems, int start, int middle, int end,
2   int& comparisons){ //merge sorted arrays together
3       //declare left and right pointer. Also create temp sub array of proper length
4       int left = start, right = middle + 1;
5       vector<string> subVec(end - start + 1);
6
7       //iterate through sub array
8       for(int i = 0; i < end - start + 1; i++){
9           comparisons++; //count comparisons
10          if(right > end){
11              //add element from left side
12              subVec[i] = magicItems[left];
13              left++;
14          }
15          else if (left > middle)
16          {
17              //add element from right side
18              subVec[i] = magicItems[right];
19              right++;
20          }
21          else if (magicItems[left].compare(magicItems[right]) < 0)
22          {
23              //add element from left side
24              subVec[i] = magicItems[left];
25              left++;
26          }
27          else{
28              //add element from right side
29              subVec[i] = magicItems[right];
30              right++;
31          }
32      }
33      //move subvector elements to main vector
34      for(int j = 0; j < end - start + 1; j++){
```

```
            magicItems[start + j] = subVec[j];
        }
    }
```

6.2 Asymptotic Analysis

When sorting an unsorted collection of objects using merge sort, the worst-case time complexity is O(n log n). This time complexity is determined due to merge sorts' two fundamental steps: divide and conquer. The divide step has a time complexity of O(log n) because it recursively divides the collection, making it smaller until they become single-element sub-arrays. On the other hand, the conquer operation has a time complexity of O(n) because it involves merging the individual sub-arrays back together, meaning the time it takes to merge them grows linearly with the arrays' size.

7 Quick Sort

The quick sort algorithm is a sorting algorithm that uses the divide and conquer approach when sorting a collection. Quick sort sorts a collection of elements using two functions: quickSort and partition. The quickSort function divides the collection in half by choosing a pivot element. It then recursively calls itself to divide the left and right sub-arrays continuously until it cannot be divided further. The partition function is responsible for the conquering and rearranging of elements in a sub-collection based on the pivot value. Elements smaller than the pivot are placed on the left, while elements larger than the pivot are placed on the right. This process is continuously called until the collection is completely sorted.

7.1 The Algorithm

Quick Sort

```
1     void Sorts:: quickSort(vector<string>& magicItems, int start, int end,
2     int& comparisons){ //quick sort algorithm
3         if(start >= end){
4             return;
5         }
6         int pivotIndex = (start + end) / 2; //choose pivot index/value
7         string pivot = magicItems[pivotIndex];
8
9         //partition based on pivot
10        pivotIndex = partition(magicItems, start, end, pivot, comparisons);
11
12        //sort left and right elements
13        quickSort(magicItems, start, pivotIndex - 1, comparisons);
14        quickSort(magicItems, pivotIndex + 1, end, comparisons);
15    }
```

Partition

```
1     int Sorts:: partition(vector<string>& magicItems, int start, int end, string pivot,
2     int& comparisons){
3         int l = start - 1; //less than pivot elements
4
5         for(int i = start; i <= end - 1; i++){ //iterate from start to end
6             comparisons++;
7             //check if less than pivot, swap, and increment
8             if(magicItems[i].compare(pivot) < 0){
9                 l++;
10                string temp = magicItems[l]; //swap
11                magicItems[l] = magicItems[i];
12                magicItems[i] = temp;
13            }
14        }
15        string temp1 = magicItems[l + 1]; //swap
16        magicItems[l + 1] = magicItems[end];
17        magicItems[end] = temp1;
18
19        return l+1; //return elements less than pivot
20
21    }
```

7.2 Asymptotic Analysis

The time complexity of quick sort, when applied to sort an unsorted collection of objects, is O(n log n). Much like merge sort, quick sort also uses the divide and conquer principle, but with randomly selected pivot points. In quick sort, when suitable pivot points are chosen, the algorithm is able to efficiently and recursively divide the input into two sub-arrays: one holding elements greater than the pivot, and the other possessing elements smaller than the pivot. This division process usually executes in O(log n) unless the worst pivot values are chosen. The conquer phase occurs when the partition function merges the pivoted arrays, which runs in O(n).

8 Appendix

8.1 Comparisons and Time Complexity

Insertion Sort	221445	$O(n^2)$
Selection Sort	108052	$O(n^2)$
Merge Sort	6302	O(n log n)
Quick Sort	6214	O(n log n)

Node cpp

```
1   //This file creates the linkedlist/node class
2   //Also creates stack and queue classes
3   #include "Node.hpp"
4
5   #include <stdexcept>
6
7   Node::Node(char val){ //Node class constructor
8       this->val = val;
9       this->next = nullptr;
10  }
11
12  Stack::Stack() { //Stack class constructor
13      top = nullptr;
14  }
15
16  void Stack::push(char val) { // push node onto stack
17      Node* newNode = new Node(val);
18      if(isEmpty()){
19          top = newNode;
20      }
21      else{
22          newNode->next = top;
23          top = newNode;
24          newNode = nullptr;
25          delete newNode;
26      }
27  }
28
29  char Stack::pop(){ //pop top node off stack
30      if (isEmpty()){
31          throw invalid_argument("The stack is empty");
32      }
33      else{
34          Node* temp = top;
35          top = top->next;
36          temp->next = nullptr;
37          return temp->val;
38          temp = nullptr;
39          delete temp;
40      }
41  }
42
43  bool Stack::isEmpty(){ //check if stack is empty
44      return top == nullptr;
45  }
46
47  Queue::Queue() { //Queue class constructor
48      head = nullptr;
49      tail = nullptr;
50  }
51
52  void Queue::enqueue(char val){ //add node to back of line
```

```
        Node* newNode2 = new Node(val);
        if(isEmpty()){ //if empty make node head and tail
            head = newNode2;
            tail = newNode2;
        }
        else{ //if not make it rear
            tail->next = newNode2;
            tail = newNode2;
            newNode2 = nullptr;
            delete newNode2;
        }
    }

    char Queue::dequeue(){ //remove node from front of line
        if(isEmpty()){
            throw invalid_argument("The queue is empty");
        }
        else{
            Node* temp = head;
            head = head->next; //change front
            if(head == nullptr){
                tail = nullptr;
            }
            return temp->val;
            temp = nullptr;
            delete temp;
        }
    }

    bool Queue::isEmpty(){ //check if queue is empty
        return head == nullptr;
    }
```

Sorts cpp

```
    //This file creates the sort classes
    #include "Sorts.hpp"

    #include <vector>
    #include <iostream>
    #include <string>

    void Sorts:: knuthShuffle(vector<string>& magicItems){
    //randomly shuffles all elements in magic items

        srand (time(NULL)); //initialize seed value so we get different random nums

        for(int i = 0; i < magicItems.size(); i++){
            //iterate through each line in magic items
            int random = rand() % magicItems.size();
            //generate random number between 1 and vector length
            string temp = magicItems[i]; //swap
            magicItems[i] = magicItems[random];
            magicItems[random] = temp;
        }
    }

    void Sorts:: selectionSort(vector<string>& magicItems){
        //selection sort to sort magic items
        int sComparisons = 0; //comparisons counter for selection sort
        for(int i = 0; i < magicItems.size(); i++){
            //iterate through vector
            int smallestPos = i; //smallest index
```

```
            for(int j = i + 1; j < magicItems.size(); j++){
                //increment and compare elements with smallest
                sComparisons++;
                if(magicItems[smallestPos].compare(magicItems[j]) > 0){
                    //if smaller change smallest
                    smallestPos = j;
                }
            }
            //swap using temp variable
            string temp = magicItems[i];
            magicItems[i] = magicItems[smallestPos];
            magicItems[smallestPos] = temp;
        }
        cout<<"Selection sort comparisons: " << sComparisons << "\n";
    }

    void Sorts:: insertionSort(vector<string>& magicItems){
        //insertion sort to sort magic items
        int iComparisons = 0;
        for(int i = 1; i < magicItems.size(); i++){
            //iterate through vector. first element sorted
            string current = magicItems[i];
            int j = i - 1; //check with previous element
            while(j >= 0 && current.compare(magicItems[j]) < 0){
                iComparisons++;
                magicItems[j + 1] = magicItems[j];
                j--;
            }
            magicItems[j + 1] = current;
        }
        cout<<"Insertion sort comparisons: " << iComparisons << "\n";
    }

    void Sorts:: mergeSort(vector<string>& magicItems, int start, int end,
    int& comparisons){ //merge sort to sort magic items
        if(start >= end){
            return;
        }
        int middle = (start + end) / 2; //find middle point
        mergeSort(magicItems, start, middle, comparisons); //sort left
        mergeSort(magicItems, middle + 1, end, comparisons); //sort right
        merge(magicItems, start, middle, end, comparisons); //merge sorted arrays
    }

    void Sorts:: merge(vector<string>& magicItems, int start, int middle, int end,
    int& comparisons){ //merge sorted arrays together
        //declare left and right pointer. Also create temp sub array of proper length
        int left = start, right = middle + 1;
        vector<string> subVec(end - start + 1);

        //iterate through sub array
        for(int i = 0; i < end - start + 1; i++){
            comparisons++; //count comparisons
            if(right > end){
                //add element from left side
                subVec[i] = magicItems[left];
                left++;
            }
            else if (left > middle)
            {
                //add element from right side
                subVec[i] = magicItems[right];
                right++;
            }
            else if (magicItems[left].compare(magicItems[right]) < 0)
```

```
94              {
95                  //add element from left side
96                  subVec[i] = magicItems[left];
97                  left++;
98              }
99              else{
100                 //add element from right side
101                 subVec[i] = magicItems[right];
102                 right++;
103             }
104         }
105         //move subvector elements to main vector
106         for(int j = 0; j < end - start + 1; j++){
107             magicItems[start + j] = subVec[j];
108         }
109     }
110
111
112     void Sorts:: quickSort(vector<string>& magicItems, int start, int end,
113     int& comparisons){ //quick sort algorithm
114         if(start >= end){
115             return;
116         }
117         int pivotIndex = (start + end) / 2; //choose pivot index/value
118         string pivot = magicItems[pivotIndex];
119
120         pivotIndex = partition(magicItems, start, end, pivot, comparisons);
121         //partition based on pivot
122
123         quickSort(magicItems, start, pivotIndex - 1, comparisons);
124         //sort left and right elements
125         quickSort(magicItems, pivotIndex + 1, end, comparisons);
126     }
127
128     int Sorts:: partition(vector<string>& magicItems, int start, int end,
129     string pivot, int& comparisons){
130         int l = start - 1; //less than pivot elements
131
132         for(int i = start; i <= end - 1; i++){ //iterate from start to end
133             comparisons++;
134             if(magicItems[i].compare(pivot) < 0){
135                 //check if less than pivot, swap, and increment
136                 l++;
137                 string temp = magicItems[l]; //swap
138                 magicItems[l] = magicItems[i];
139                 magicItems[i] = temp;
140             }
141         }
142         string temp1 = magicItems[l + 1]; //swap
143         magicItems[l + 1] = magicItems[end];
144         magicItems[end] = temp1;
145
146         return l+1; //return elements less than pivot
147
148     }
```

9 References

9.1 Links

Below are the resources I have used to create simple, readable, and beautiful code.

- Helped me create a Make File: bytes.usc.edu
- Bits and pieces helped me create my read File: stackoverflow.com
- I used this line of code to remove white spaces in my read File: stackoverflow.com
- Also helped me with white spaces: scaler.com
- Aided in understanding and using hpp and cpp files: geeksforgeeks.org
- A youtube playlist that helped me understand everything about C++: youtube.com
- Helped me with all sorts, stacks, and queues: Algorithms textbook
- Helped me form and articulate descriptions for each data structure and algorithm: Labouseur.com