Assignment One – LaTeX Data

Patrick Tyler
Patrick.Tyler1@marist.edu

October 6, 2023

1 Nodes, Stacks, & Queues

1.1 Making a Node

A node needs to contain the data it stores and reference(s) to another node. In this case, just a single reference to a next Node is needed because the nodes will only be singly linked. A node struct could be used to implement this which wraps a piece of data with a reference to the next Node. A node is a building block for many different algorithms and structures; therefore, this node structure should be generic enough to easily create a node storing any type of data.

```cpp
// Wraps a generic, T, typed data in a node to reference other nodes
template <typename T>
struct Node{
    T data;
    Node<T>* next;
};
```

1.2 Stacks & Queues

Nodes can link together to form stacks and queues. Stacks and queues in programming should be intuitive as stacks (much like a stack of papers) are first in first out (FIFO) while queues (much like a line of people waiting for something) are last in first out (LIFO).

Well structured stack and queue classes encapsulate node structure they are made of only revealing their respective functionality to add and get data. Both stacks and queues should have method to check if the are empty, add data, and remove data. The interface for stacks minimally follows: push() and pop(). The interface for queues minimally follows: queue() and enqueue(). It should be obvious what each method does. Each method should happen in constant time.

The stack class will need to have contain a pointer to the head (the top of the stack) to implement its interface.

Push action:

1. Takes in a piece of data and creates a new node.
2. Sets the new node to this head.
3. Sets the new node's next value to the previous head (if there was one).

```
        void push(T value) {
            Node<T>* node = new Node<T>;
            node->data = value;
            node->next = top;
            top = node;
```

Pop action:

1. Make the new head the current head's next attribute.
2. Return the value of now the old head node.

```
        T pop(){
            if(isEmpty()){
                throw logic_error("Cannot pop from empty stack.");
            }
            Node<T>* trash = top;
            T value = top->data;
            top = top->next;
            delete trash;

            return value;
```

isEmpty action:

1. Returns if the head is currently points to a node.

The queue class will need to have a pointer to the first (start of the line) and the last (end of the line) node to implement its interface in constant time. It is possible to only have a pointer to the first node and then traverse the queue to obtain the last, but the trade off to store an additional pointer to the last element is almost worth it because it saves the queues from having to execute a linear time complexity operation when enqueuing.

Enqueue action:

1. Takes in a piece of data and creates a new node.

2. Sets the new node to the value of the tail.
3. Only if there was no head node, sets the new node equal to the head.
4. Sets the previous tail's next value to the new node (if there was one).

```
        void enqueue(T value){
            Node<T>* node = new Node<T>;
            node->data = value;
            node->next = nullptr;

            if(first == nullptr){
                first = node;
            }

            if(last == nullptr){
                last = node;
            } else {
                last->next = node;
                last = node;
            }
        }
```

Dequeue action:

1. Set the first to the current first's next attribute.
2. Return the value of now the old head node.

```
        T dequeue() {
            if(isEmpty()) {
                throw logic_error("Cannot dequeue from empty queue.");
            }
            Node<T>* trash = first;
            T value = first->data;
            first = first->next;
            delete trash;

            if (first == nullptr) {
                last = nullptr;
            }

            return value;
```

isEmpty action:

1. Returns if the head is currently points to a node.

1.3 A Simple Stack & Queue Use Case

A palindrome is phrase that is the same forward and backwards often only including alphanumeric characters and casing. "Race car" is a palindrome because the letters read the same front to back and back to front.

$$R_0a_1c_2e_3\ c_4a_5r_6 == r_6a_5c_4\ e_3c_2a_1R_0$$

This applies to stack and queues as if the letters of race car were pushed to a stack as well as enqueued to a queue, comparing, in order, the dequeued elements with the popping elements will render all equal comparisons. If any of these comparisons are false then then the phrase is not a palindrome.

2 Sorting

2.1 Introduction

An array of data (say n with length x) is said to be sorted when comparable elements are arranged in an order such that:

$$n_0 <= n_1 <= n_2 <= n_3 \ldots <= n_{x-1}$$

$$or$$

$$n_0 >= n_1 >= n_2 >= n_3 \ldots >= n_{x-1}$$

For future algorithms such as binary search, this relationship is foundational.

2.2 The "Opposite" of Sorted

Conceptually, the opposite of a sorted array is one in which the elements are randomly positioned. This does not mean the array is not sorted, just that the elements are placed randomly. One of the most intuitive random shuffling algorithms is the Fisher Yates Shuffle. It iterates through each index of the array picking a random index of out the current index and all indexes not yet iterated through.

```
void fisherYatesShuffle(string* items, int length = DEVILS_NUMBER){
    int swapIndex;

    // Reseeding the number generator with the psuedo random time
    // This means that multiple shuffles within the program can be linked through their delay between shuffle invocations
    // If "better" randomness is desired the random library could be considered
    srand(static_cast<unsigned>(time(nullptr)));

    for(int end = length-1; end > 0; end--){
        // +1 to include current index
        swapIndex = rand() % (end + 1);
        if(swapIndex == end) { continue; }
        // probably internally done using an XOR swap https://en.wikipedia.org/wiki/XOR_swap_algorithm
        //  of the chars in the string
        // Using the inbuilt swap is more elegant since we can interface an object type into this sort if needed
        swap(items[end], items[swapIndex]);
```

2.3 Selection Sort

Selection sort puts an array in an order using nested loops. The idea is to iterate through the array and each time find the minimal value from that index to the end index. After the minimal element is found swap it with the current index. This algorithm works in quadratic time because each the upper loop with have to carry out an inner loop relative to the size of the array. Even though the inner loop decreases in iterations as the sort progresses this is still quadratic time because number of the inner iterations depends in the size of the input; thus n * n comparisons (when n is size of the array), even if there are other constant factors.

```
int selectionSort(string* items, int length=DEVILS_NUMBER) {
    int comparisonCounter = 0;
    int minIndex;
    for(int i = 0; i < length; i++){
        minIndex = i;
        // finds the min index from i to length -
        for(int j = i + 1; j < length; j++){
            comparisonCounter++;
            if(items[j] < items[minIndex]){
                minIndex = j;
            }
        }
        if(minIndex == i){ continue; }
        swap(items[i], items[minIndex]);
        // items 0 to i is now sorted
    }

    return comparisonCounter;
```

2.4 Insertion Sort

Insertion sort utilizes an adaptive approach with nested loops. The idea is to iterate through the array starting at the second item and move each value down the array until the correct previous item is found (this comparison changes depends on what type of ordering). This is adaptive as the second loop only continues until the a correctly positioned item is found. The time complexity of this algorithm is difficult to calculate directly as it depends on the order the items are in, but the worst case is that the inner loop will have to iterate all the back to the start each time. This places this algorithm in its worst case at the same situation as selection sort since both nested loops depend on the length of the array making this sort also have quadratic time complexity.

```
int insertionSort(string* items, int length=DEVILS_NUMBER){
    int comparisonCounter = 0;
    string selectedItem;
    int prevIndex;
    for(int i=1; i < length; i++){
        selectedItem = items[i];
        prevIndex = i - 1;
        // finding the correct index to insert into array
```

```
        while(prevIndex >= 0) {
            // used to have this check reversed as a condition for the while
            //  put here just to ensure that comparisons are counted only for comparison between elements
            comparisonCounter++;
            if(items[prevIndex] <= selectedItem){
                break;
            }
            items[prevIndex + 1] = items[prevIndex];
            prevIndex--;
        }
        items[prevIndex + 1] = selectedItem;
        // items 0 to i are relatively in order that is items0 < items1...  < itemsi
        //   0 to i might not be in the right order for the whole list
    }

    return comparisonCounter;
}
```

2.5 Merge Sort

Merge sort utilizes a divide and conquer paradigm. It recursively splits the array in halves there are only subarrays of length 0 or 1. This guarantees that the subarrays start a sorted state since an array with length 0 or 1 is already in order. Now the algorithm can merge these subarrays back together creating large and larger subarrays at each step until it has merged the entire array together. Every divide step is constant time and there are $log_2 n$ because the array is halved each time. Each divide step must also be merged back together. There are n items to merge back together at each divide depth. Therefore, the time complexity is big oh of $nlog_2 n$.

The following code is an example merge function which purposefully uses more memory allocation to allow two individual lists to be taken as arguments and merged into one array. This approach may be helpful when the entire list will not be able to fit in memory.

```
pair<string*, int> mergeSorted(string* items1, string* items2, int size1, int size2){
    int count = 0;
    int mergedLength = size1 + size2;
    string* mergedItems = new string[mergedLength];
    int i1 = 0;
    string item1;
    int i2 = 0;
    string item2;
    for(int iMerged = 0; iMerged < mergedLength; iMerged++){
        // check for out of bounds
        if(i1 == size1){
            mergedItems[iMerged] = items2[i2];
            i2++;
```

```
            continue;
        }
        if(i2 == size2){
            mergedItems[iMerged] = items1[i1];
            i1++;
            continue;
        }

        // compare items to find min of the two and place in new array
        //  only iterates the the array index of the one that is added
        item1 = items1[i1];
        item2 = items2[i2];
        count++;
        if(item1 < item2){
            mergedItems[iMerged] = item1;
            i1++;
        } else {
            mergedItems[iMerged] = item2;
            i2++;
        }
    }
    return make_pair(mergedItems, count);
}
```

The following code is an example recursive case for merge sort which splits the array. Usually the subarrays would be done in the previous function but since this implementation receives the subarrays as arguments it must be done in this step.

```
void mergeSort(string* items, int* count, int istart=0, int iend=DEVILS_NUMBER-1){
    // base case as we have reached our sorted array of at 1 or 0 elements!
    if (istart >= iend){
        return;
    }

    int imiddle = istart + (iend - istart) / 2;
    int leftLength = imiddle - istart + 1;
    string left[leftLength];
    int rightLength = iend - imiddle;
    string right[rightLength];
    // recursively call mergeSort to ensure that we are using the sorted array when we merge
    //   with the two other half of the array
    mergeSort(items, count, istart, imiddle);
    mergeSort(items, count, imiddle + 1, iend);


    // populating left and right sub arrays to be merged
    for(int i = 0; i < leftLength; i++){
        left[i] = items[istart + i];
    }
    for(int i = 0; i < rightLength; i++){
        right[i] = items[imiddle + 1 + i];
```

```
    }

    auto mergedResult = mergeSorted(left, right, leftLength, rightLength);
    string* mergedArray = mergedResult.first;

    // this is the syntax to increment the value of the pointer
    (*count) += mergedResult.second;
    int mergedLength = leftLength + rightLength;

    for(int imerged = 0; imerged < mergedLength; imerged++){
        items[istart + imerged] = mergedArray[imerged];
    }
    delete[] mergedArray;
}
```

2.6 Quick Sort

Quick sort also utilizes a divide and conquer (at the same time) paradigm. It recursively divides the array around a pivot value and during each partition arranges the subarrays in their correct order relative to only the pivot (not relative to the rest of the array or the other elements in the array). Once the sufficiently small partitions (length 1-3) have been sorted the array is now sorted because each value is relatively placed in the correct partition after each recursive call (ideally the correct half of the subarrays but it will often not be perfect halves). It may be helpful to think the best case quick sort as a binary search for the correct position of the value for each element in the array. The time complexity for quick sort is also big oh of $nlog_2n$ because there are a little more than log_2n partitions (when a decent pivot value is chosen) and n comparisons for each partition to order the element around the partition at each depth.

The following code is used to obtain the median value of three indexes. This is needed to determine a pivot value that will not degrade the quick sort algorithm to a quadratic time complexity. Picking a pivot value that is either the smallest value of the partition or the largest will order the rest of the elements to one side of the partition which essentially only orders a single element. Thus, the median element of three elements from the partition ensure at least one item will be each side. Since the three pivot candidates are psuedo random on a shuffled array, the number of comparisons will change depending on the close the chosen pivot is to the true median of the partition.

```
int getiMiddleOfThree(string* items, int istart, int iend, int* counter){
    int iMedianOfThree;
    int imiddle = (istart + iend) / 2;
    // partial ordering shown after each comparison
    (*counter)++;
    if(items[istart] < items[iend]){
        // istart, iend
        (*counter)++;
        if (items[istart] >= items[imiddle]){
```

```
            // imiddle, istart, iend
            iMedianOfThree = istart;
        } else if (items[iend] < items[imiddle]){
            (*counter)++;
            // istart, iend, imiddle
            iMedianOfThree = iend;
        } else {
            (*counter)++;
            // istart, imiddle, iend
            iMedianOfThree = imiddle;
        }
    } else {
        // iend, istart
        (*counter)++;
        if (items[istart] < items[imiddle]){
            // iend, istart, imiddle
            iMedianOfThree = istart;
        } else if (items[iend] > items[imiddle]) {
            (*counter)++;
            // imiddle, iend, istart
            iMedianOfThree = iend;
        } else {
            (*counter)++;
            // iend, imiddle, istart
            iMedianOfThree = imiddle;
        }
    }
```

The following code partitions a range of the array ordering it relative to the pivot point.

```
int partition(string* items, int* counter, int istart, int iend){
    int ipivot = getiMiddleOfThree(items, istart, iend, counter);
    string pivotVal = items[ipivot];

    // arbitrarily Swapping pivot to end to ignore it when swapping around pivot value
    swap(items[ipivot], items[iend]);

    // index to put the lesser elements compared to the pivot
    int ilesser = istart - 1;

    for(int i = istart; i < iend; i++){
        (*counter)++;
        if(items[i] <= pivotVal){
            // preincrement since we start at istart - 1
            ilesser++;
            swap(items[i], items[ilesser]);
        }
        // do nothing if element is already less than pivot since it is in the correct place
    }

    // swapping pivot back to correct spot
    swap(items[ilesser + 1], items[iend]);
    return ilesser + 1;
}
```

The following code recursively calls itself to partition each partition until sorted.

```
void quickSort(string* items, int* counter, int istart=0, int iend=DEVILS_NUMBER-1){
    if (istart >= iend){
        return;
    }

    int ipivot = partition(items, counter, istart, iend);

    // Recursive calls for all elements around the pivot point
    quickSort(items, counter, istart, ipivot-1);
    quickSort(items, counter, ipivot+1, iend);
}
```

2.7 Sort Overview

Each sort has its own advantages for certain scenarios: selection sort has low overhead and minimal swapping, insertion sort works great on sorted or almost sorted arrays, merge sort looks cool, and quick sort is simply the GOAT. Merge sort and quick sort are both significantly better on large arrays because of their time complexity. Merge sort has more overhead than quick sort, but takes less comparisons. Most inbuilt sorting algorithms end of using a hybrid of quick sort and insertion sort.

Type of Sort	Time Complexity	Number of Comparisons
Selection Sort	$O(n^2)$	$221,445$
Insertion Sort	$O(n^2)$	$109,540^*$
Merge Sort	$O(nlog_2n)$	$5,399$
Quick Sort	$O(nlog_2n)$	$6,348^*$

*denotes adaptive comparison number