

Assignment One – L^AT_EX 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.

```
27 // Wraps a generic, T, typed data in a node to reference other
    nodes
28 template <typename T>
29 struct Node{
30     T data;
31     Node<T>* next;
32 };
```

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 last in first out (LIFO) while queues (much like a line of people waiting for something) are first in first out (FIFO).

Well structured stack and queue classes encapsulate their node structure only exposing their respective functionality to add and get data. Both stacks and queues should have methods to check if they 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. The methods ideally all run in constant time.

The stack class will need to have 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).

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

Pop action:

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

```
105     T pop(){
106         if(isEmpty()){
107             throw logic_error("Cannot pop from empty stack.");
108         }
109         Node<T>* trash = top;
110         T value = top->data;
111         top = top->next;
112         delete trash;
113
114         return value;
```

isEmpty action:

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

Why are stacks often the life of the party? - *They get it popping.*

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 queues from having to execute a linear time complexity operation when enqueueing.

Enqueue action:

1. Takes in a piece of data and creates a new node.
2. Sets the new node to be equal to 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).

```
47     void enqueue(T value){
48         Node<T>* node = new Node<T>;
49         node->data = value;
50         node->next = nullptr;
51
52         if(first == nullptr){
53             first = node;
54         }
55
56         if(last == nullptr){
57             last = node;
58         } else {
59             last->next = node;
60             last = node;
61         }
62     }
```

Dequeue action:

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

```
66     T dequeue() {
67         if(isEmpty()) {
68             throw logic_error("Cannot dequeue from empty queue.
69         ");
70         }
71         Node<T>* trash = first;
72         T value = first->data;
73         first = first->next;
74         delete trash;
75
76         if (first == nullptr) {
77             last = nullptr;
78         }
79         return value;
80     }
```

isEmpty action:

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

```
1 public class Queue<T> {  
2 ...
```

Guy: *Hey girl, are you my generically typed queue declaration?*

Girl: *What?*

Guy: *Because you're a Queue<T>.*

Girl: ...

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 excluding casing. "Race car" is a palindrome because the letters read the same front to back and back to front.

$$R_0 a_1 c_2 e_3 c_4 a_5 r_6 == r_6 a_5 c_4 e_3 c_2 a_1 R_0$$

This applies to stacks 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 popped elements renders all equal comparisons. In other words, if any of these comparisons are false, then the phrase is not a palindrome. Implementation of this in code is left as an exercise to the reader.

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:

$$n_0 \leq n_1 \leq n_2 \leq n_3 \dots \leq n_{x-1}$$

or

$$n_0 \geq n_1 \geq n_2 \geq n_3 \dots \geq n_{x-1}$$

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 and swaps those indices.

```

225 void fisherYatesShuffle(string* items, int length = DEVILS_NUMBER){
226     int swapIndex;
227
228     // Reseeding the number generator with the psuedo random time
229     // This means that multiple shuffles within the program can be
    linked through their delay between shuffle invocations
230     // If "better" randomness is desired the random library could
    be considered
231     srand(static_cast<unsigned>(time(nullptr)));
232
233     for(int end = length-1; end > 0; end--){
234         // +1 to include current index
235         swapIndex = rand() % (end + 1);
236         if(swapIndex == end) { continue; }
237         // probably internally done using an XOR swap https://en.
    wikipedia.org/wiki/XOR_swap_algorithm
238         // of the chars in the string
239         // Using the inbuilt swap is more elegant since we can
    interface an object type into this sort if needed
240         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 will 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 the number of the inner iterations depends on the size of the input; thus $n * n$ comparisons (when n is size of the array), even if there are other constant factors.

```

247 int selectionSort(string* items, int length=DEVILS_NUMBER) {
248     int comparisonCounter = 0;
249     int minIndex;
250     for(int i = 0; i < length; i++){
251         minIndex = i;
252         // finds the min index from i to length -
253         for(int j = i + 1; j < length; j++){
254             comparisonCounter++;
255             if(items[j] < items[minIndex]){
256                 minIndex = j;
257             }
258         }
259         if(minIndex == i){ continue; }
260         swap(items[i], items[minIndex]);
261         // items 0 to i is now sorted
262     }
263
264     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 is adaptive because 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 puts the algorithm, in its worst case, in the same situation as selection sort since both nested loops depend on the length of the array. Thus, this sort also have quadratic time complexity.

```
268 int insertionSort(string* items, int length=DEVILS_NUMBER){
269     int comparisonCounter = 0;
270     string selectedItem;
271     int prevIndex;
272     for(int i=1; i < length; i++){
273         selectedItem = items[i];
274         prevIndex = i - 1;
275         // finding the correct index to insert into array
276         while(prevIndex >= 0) {
277             // used to have this check reversed as a condition for
the while
278             // put here just to ensure that comparisons are
counted only for comparison between elements
279             comparisonCounter++;
280             if(items[prevIndex] <= selectedItem){
281                 break;
282             }
283             items[prevIndex + 1] = items[prevIndex];
284             prevIndex--;
285         }
286         items[prevIndex + 1] = selectedItem;
287         // items 0 to i are relatively in order that is items0 <
items1... < itemsi
288         // 0 to i might not be in the right order for the whole
list
289     }
290
291     return comparisonCounter;
292 }
```

2.5 Merge Sort

Merge sort utilizes a divide and conquer paradigm. It recursively splits the array in halves until there are only subarrays of length 0 or 1. This guarantees that the subarrays start in 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 $\log_2 n$ divides 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 $n \log_2 n$.

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

```
300 pair<string*, int> mergeSorted(string* items1, string* items2, int
    size1, int size2){
301     int count = 0;
302     int mergedLength = size1 + size2;
303     string* mergedItems = new string[mergedLength];
304     int i1 = 0;
305     string item1;
306     int i2 = 0;
307     string item2;
308     for(int iMerged = 0; iMerged < mergedLength; iMerged++){
309         // check for out of bounds
310         if(i1 == size1){
311             mergedItems[iMerged] = items2[i2];
312             i2++;
313             continue;
314         }
315         if(i2 == size2){
316             mergedItems[iMerged] = items1[i1];
317             i1++;
318             continue;
319         }
320
321         // compare items to find min of the two and place in new
array
322         // only iterates the the array index of the one that is
added
323         item1 = items1[i1];
324         item2 = items2[i2];
325         count++;
326         if(item1 < item2){
327             mergedItems[iMerged] = item1;
328             i1++;
329         } else {
330             mergedItems[iMerged] = item2;
331             i2++;
332         }
333     }
334     return make_pair(mergedItems, count);
335 }
```

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.

```
338 void mergeSort(string* items, int* count, int istart=0, int iend=
    DEVILS_NUMBER-1){
339     // base case as we have reached our sorted array of at 1 or 0
    elements!
340     if (istart >= iend){
341         return;
342     }
343
344     int imiddle = istart + (iend - istart) / 2;
345     int leftLength = imiddle - istart + 1;
346     string left[leftLength];
347     int rightLength = iend - imiddle;
348     string right[rightLength];
349     // recursively call mergeSort to ensure that we are using the
    sorted array when we merge
350     // with the two other half of the array
351     mergeSort(items, count, istart, imiddle);
352     mergeSort(items, count, imiddle + 1, iend);
353
354
355     // populating left and right sub arrays to be merged
356     for(int i = 0; i < leftLength; i++){
357         left[i] = items[istart + i];
358     }
359     for(int i = 0; i < rightLength; i++){
360         right[i] = items[imiddle + 1 + i];
361     }
362
363     auto mergedResult = mergeSorted(left, right, leftLength,
    rightLength);
364     string* mergedArray = mergedResult.first;
365
366     // this is the syntax to increment the value of the pointer
367     (*count) += mergedResult.second;
368     int mergedLength = leftLength + rightLength;
369
370     for(int imerged = 0; imerged < mergedLength; imerged++){
371         items[istart + imerged] = mergedArray[imerged];
372     }
373     delete[] mergedArray;
374 }
```

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 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 of best case quick sort (picking true median values for pivots) 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 $n \log_2 n$ because there are a little more than $\log_2 n$ partitions (when a decent pivot value is chosen) and n comparisons for each partition to order the element around the pivot 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 ensures at least one item will be on each side. Since the three pivot candidates are pseudo random on a shuffled array, the number of comparisons will change depending on how close the chosen pivot is to the true median of the partition.

```

377 int getMiddleOfThree(string* items, int istart, int iend, int*
    counter){
378     int iMedianOfThree;
379     int imiddle = (istart + iend) / 2;
380     // partial ordering shown after each comparison
381     (*counter)++;
382     if(items[istart] < items[iend]){
383         // istart, iend
384         (*counter)++;
385         if (items[istart] >= items[imiddle]){
386             // imiddle, istart, iend
387             iMedianOfThree = istart;
388         } else if (items[iend] < items[imiddle]){
389             (*counter)++;
390             // istart, iend, imiddle
391             iMedianOfThree = iend;
392         } else {
393             (*counter)++;
394             // istart, imiddle, iend
395             iMedianOfThree = imiddle;
396         }
397     } else {
398         // iend, istart
399         (*counter)++;
400         if (items[istart] < items[imiddle]){
401             // iend, istart, imiddle
402             iMedianOfThree = istart;
403         } else if (items[iend] > items[imiddle]) {
404             (*counter)++;
405             // imiddle, iend, istart
406             iMedianOfThree = iend;
407         } else {
408             (*counter)++;
409             // iend, imiddle, istart

```

```

410         iMedianOfThree = imiddle;
411     }
412 }
413
414     return iMedianOfThree;
415 }

```

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

```

418 int partition(string* items, int* counter, int istart, int iend){
419     int ipivot = getiMiddleOfThree(items, istart, iend, counter);
420     string pivotVal = items[ipivot];
421
422     // arbitrarily Swapping pivot to end to ignore it when swapping
423     // around pivot value
424     swap(items[ipivot], items[iend]);
425
426     // index to put the lesser elements compared to the pivot
427     int ilesser = istart - 1;
428
429     for(int i = istart; i < iend; i++){
430         (*counter)++;
431         if(items[i] <= pivotVal){
432             // preincrement since we start at istart - 1
433             ilesser++;
434             swap(items[i], items[ilesser]);
435         }
436         // do nothing if element is already less than pivot since
437         // it is in the correct place
438     }
439
440     // swapping pivot back to correct spot
441     swap(items[ilesser + 1], items[iend]);
442     return ilesser + 1;
443 }

```

The following code recursively calls itself until partitions are small enough to have fully sorted the array.

```

445 void quickSort(string* items, int* counter, int istart=0, int iend=
446     DEVILS_NUMBER-1){
447     if (istart >= iend){
448         return;
449     }
450
451     int ipivot = partition(items, counter, istart, iend);
452
453     // Recursive calls for all elements around the pivot point
454     quickSort(items, counter, istart, ipivot-1);
455     quickSort(items, counter, ipivot+1, iend);
456 }

```

What type of wedding did quick sort and merge sort have?

- *An arranged wedding.*

What was their first married argument? - *How to divide the cake.*

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. In all seriousness, 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 up 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	$O(n\log_2 n)$	5,399
Quick Sort	$O(n\log_2 n)$	6,348*

*denotes adaptive comparison number

The above chart is counting the number of comparisons when sorting an array of length 666. Although data about comparisons counted with differently ordered starting arrays and different length arrays could lead to more comprehension conclusions, this data shows each sort's number of comparison is proportional to its time complexity by some constant factor. Of course, adaptive comparison numbers will change depending on the starting array; still, their comparisons average out to be some proportional value.

3 My Code Presentation

I am not sure if is needed but the assignment says ALL code must be documented in latex so I will document my presentation code here. But, this section will not read like a textbook because this is strictly about my own presentation decisions

To ensure the node, queue, and stack code works properly I wrote testQueue() and testStack() functions. These functions add and remove 10 items from the queue and stack twice. I did this way to ensure the structures worked in cases where they were adding and removing with no nodes.

```
123 // Add and remove elements from a queue twice
124 void testQueue() {
125     int numbers[10] = {1,2,3,4,5,6,7,8,9,10};
126
127     Queue<int> queue;
128     for(int num: numbers ){
129         queue.enqueue(num);
130     }
131
132     int value;
133     while(!queue.isEmpty()) {
134         value = queue.dequeue();
135         cout << value << " ";
```

```

136     }
137     cout << endl;
138
139     for(int num: numbers ){
140         queue.enqueue(num);
141     }
142
143     while(!queue.isEmpty()) {
144         value = queue.dequeue();
145         cout << value << ", ";
146     }
147     cout << endl;
148 }
149
150 // add and remove elements from a stack
151 void testStack() {
152     int numbers[10] = {1,2,3,4,5,6,7,8,9,10};
153
154     Stack<int> stack;
155     for(int num: numbers ){
156         stack.push(num);
157     }
158
159     int value;
160     while(!stack.isEmpty()) {
161         value = stack.pop();
162         cout << value << ", ";
163     }
164     cout << endl;
165
166     for(int num: numbers ){
167         stack.push(num);
168     }
169
170     while(!stack.isEmpty()) {
171         value = stack.pop();
172         cout << value << ", ";
173     }
174     cout << endl;
175
176 }

```

To get the magic items used in further tests and sorting I read in the .txt file putting each line into a statically sized array.

```

210 // reading in magic items from relative path
211 string* getMagicItems(){
212     string* magicItems= new string[DEVILS_NUMBER];
213     ifstream magicItemsStream("magicitems.txt");
214     string line;
215     int i = 0;
216     while(getline(magicItemsStream, line)){
217         magicItems[i] = line;
218         i++;
219     }
220     magicItemsStream.close();
221     return magicItems;
222 }

```

To display the palindromes I iterated through the entire items array and used my function described but not displayed earlier to check if the phrase was a palindrome.

```
178 bool isPalindrome(string &candidate){
179     Queue<char> fifo;
180     Stack<char> lifo;
181     for(const char ch: candidate) {
182         if(ch == ' '){
183             continue;
184         }
185         // same case
186         fifo.enqueue(tolower(ch));
187         lifo.push(tolower(ch));
188     }
189
190     // asymmetrical patten since fifo and lifo should be empty at
191     // the same iteration
192     while(!fifo.isEmpty() ){
193         if(fifo.dequeue() != lifo.pop()){
194             return false;
195         }
196     }
197     return true;
198 }
199
200 }
201
202 void cpalindromes(string* candidates){
203     for(int i=0; i < DEVILS_NUMBER; i++){
204         if(isPalindrome(candidates[i])){
205             cout << candidates[i] << endl;
206         }
207     }
208 }
```

To view the array in the console to ensure proper sorting, I created a helper function which displays up to the certain count but to over the length. This allowed me to quickly check if part of the list was sorted when making the sorting functions.

```
457 // helper function to display only part of the list
458 void cTruncatedItems(string* magicItems, int count=10, int length
    =666){
459     for(int i=0; i < min(count, length); i++){
460         cout << magicItems[i] << endl;
461     }
462     if(length > count){
463         cout << "..." << endl;
464     }
465 }
```

To run these test I created a main function which: tests the queues, stacks, and node; prints the palindromes; runs each sort shuffling in between as well as displaying the first n elements and the count of comparisons.

```
467 // Run tests
468 int main(){
469     // display for node, stack, and queue
470     cout << "Testing node class by implementing a stack and a queue
        and stack..." << endl << endl;
471     cout << "Enqueuing 1-10 and dequeuing x2:" << endl;
472     testQueue();
473     cout << endl;
474     cout << "Pushing 1-10 and popping x2:" << endl;
475     testStack();
476     cout << endl;
477     string* magicItems = getMagicItems();
478
479     // Display for palindromes
480     cout << "Showing all 12 of the palindromic magic items: " <<
        endl;
481     cpalindromes(magicItems);
482     cout << endl;
483
484     // prep for sort
485     cout << endl << "Shuffling the list to prepare for sorting..."
        << endl << endl;
486     fisherYatesShuffle(magicItems);
487
488
489     // Display for selection sort
490     int selectionSortCounter = selectionSort(magicItems);
491     cout << "Showing first " << truncationCount << " sorted items
        by selection sort. Entire list sorted in ";
492     cout << selectionSortCounter << " comparisons:" << endl;
493     cTruncatedItems(magicItems, truncationCount);
494     cout << endl << "Shuffling the list..." << endl << endl;
495     fisherYatesShuffle(magicItems);
496
497     // Display for insertion sort
498     int insertionSortCounter = insertionSort(magicItems);
499     cout << "Showing first " << truncationCount << " sorted items
        by insertion sort. Entire list sorted in ";
```

```

500     cout << insertionSortCounter << " comparisons:" << endl;
501     cTruncatedItems(magicItems, truncationCount);
502     cout << endl << "Shuffling the list..." << endl << endl;
503     fisherYatesShuffle(magicItems);
504
505     // Display for mergesort
506     int mergeSortCounter = 0;
507     mergeSort(magicItems, &mergeSortCounter);
508     cout << "Showing first " << truncationCount << " sorted items
509     by mergesort. Entire list sorted in ";
510     cout << mergeSortCounter << " comparisons:" << endl;
511     cTruncatedItems(magicItems, truncationCount);
512     cout << endl << "Shuffling the list..." << endl << endl;
513     fisherYatesShuffle(magicItems);
514
515     // Display for quicksort
516     int quickSortCounter = 0;
517     quickSort(magicItems, &quickSortCounter);
518     cout << "Showing first " << truncationCount << " sorted items
519     by quicksort. Entire list sorted in ";
520     cout << quickSortCounter << " comparisons:" << endl;
521     cTruncatedItems(magicItems, truncationCount);
522     return 0;
523 }

```