CMPT 435 - Fall 2024 - Dr. Labouseur

Assignment One – Data Structures & Sorting

Stefano Farro
Stefano.Farro1@Marist.edu

October 4, 2024

1 Data Structures

1.1 Node

```
Node::Node(int value) {
    this->data = value;
    this->next = nullptr;
}
```

Nodes form the basis of both the Stack and the Queue. The Node has two attributes: an integer value storing the elemental data, and a pointer to the next node.

1.2 Stack

The Stack is a one-way, linear data structure with a single pointer to the topmost element. Nodes are linked together in a Last In, First Out (LIFO) fashion like that of a stack of dinner plates, where the most recent addition is the first to be removed.

```
Stack::Stack() { this->top = nullptr; }

bool Stack::isEmpty() {
    return top == nullptr;
}
```

1.2.1 Stack Operations

Push inserts an element at the top of the stack.

- Data is wrapped into a newly created node
- The new node points to the previous top of the stack
- The top pointer is updated to the new node

```cpp
void Stack::push(int newData) {
    Node* newNode = new Node(newData);
    newNode->next = top;
    top = newNode;
}
```

Pop removes the topmost element from the stack.

- The top pointer is updated to point to the next node
- The previous top node is deleted to free memory

```cpp
void Stack::pop() {
    if (isEmpty()) {
        std::cout << "Empty Stack";
    }
    else {
        Node* temp = top;
        top = top->next;
        delete temp;
    }
}
```

isEmpty checks if the Stack is empty.

```cpp
bool Stack::isEmpty() {
    return top == nullptr;
}
```

Peek returns the top of the stack without altering it.

- In the case of an empty stack, an error message is printed and `INT_MIN` is returned as a sentinel value (`Lines 5-8`)

```cpp
1 int Stack::peek() {
2     if (!isEmpty()) {
3         return top->data;
4     }
5     else {
6         std::cout << "Empty Stack";
7         return INT_MIN;
8     }
9 }
```

1.3 Queues

The Queue is linear data structure with a pointer to the queue's head and a pointer to it's tail. Elements are enter & depart under First in, First Out (FIFO) principles where new nodes are added to the tail, and removed from the top.

```cpp
Queue::Queue() { head = tail = nullptr;}

bool Queue::isEmpty() {
    return head == nullptr;
}
```

1.3.1 Queue Operations

Enqueue inserts an element to the rear (tail) of the queue

- If the queue is empty, the new sole element becomes both the head and tail (`Lines 4-6`)
- Otherwise, the previous tail is updated to point towards the new element, and the tail becomes the new element (`Lines 8-9`)

```cpp
1  void Queue::enqueue(int newData) {
2      Node* newNode = new Node(newData);
3
4      if (isEmpty()) {
5          head = tail = newNode;
6      }
7
8      else {
9          tail->next = newNode;
10         tail = newNode;
11     }
12 }
```

Dequeue removes the element "first in line" at the head.

- The queue is checked for the emptiness to ensure bounds are kept
- Otherwise, a temp node is used hold the original head node, allowing it to be deleted once the new head is in place (`Lines 6-7, 13`)

```cpp
1  void Queue::dequeue() {
2      if (isEmpty()) {
3          std::cout << "Queue Empty";
4          return;
5      }
6      Node* temp = head;
7      head = head->next;
8
9      if (head == nullptr) {
10         tail = nullptr;
11     }
12
13     delete temp;
14 }
```

isEmpty checks if the Queue is empty.

```cpp
bool Queue::isEmpty() {
    return head == nullptr;
}
```

Peek returns the top of the Queue without altering it.

- In the case of an empty queue, an error message is printed and `INT_MIN` is returned as a sentinel value (`Lines 5-8`)

```cpp
1 int Queue::peek() {
2     if (!this->isEmpty()) {
3         return head->data;
4     }
5     else {
6         std::cout << "Empty Queue";
7         return INT_MIN;
8     }
9 }
```

1.4 Palindrome Checking

Vector `items` of randomly sorted strings is iterated over. One string at a time, alphanumeric `Chars` are converted to their lowercase version and popped onto stack S and queue Q (`Lines 11-17`). Both the stack and queue are then peeked, with the stack returning the last character of the string at it's top and the queue returning the first character of that string at it's head (`Line 21`). If these match, the stack and queue are both decremented, and the rest of the characters are compared. The string is printed if the last removed elements of the stack and queue never differ (`Line 31`)

```
 1 void palindromes(std::vector<std::string>& items) {
 2     int size = items.size();
 3     int i = 0;
 4 
 5     std::cout << "\nPalindromes:" << std::endl;
 6     while (i < size) {
 7         Stack s;
 8         Queue q;
 9         std::string item = items[i];
10 
11         for(char c : item) {
12             if(isalnum(c)) {
13                 char lowC = tolower(c);
14                 s.push(lowC);
15                 q.enqueue(lowC);
16             }
17         }
18 
19         bool palindrome = true;
20         while (!s.isEmpty() && !q.isEmpty()) {
21             if(s.peek() == q.peek()) {
22                 s.pop();
23                 q.dequeue();
24             }
25             else {
26                 palindrome = false;
27                 break;
28             }
29         }
30 
31         if (palindrome) {std::cout << items[i] << std::endl;};
32         i++;
33     }
34     std::cout << std::endl;
35 }
```

2 Sorting Algorithms

2.1 Selection Sort

```
 1 void selectionSort(std:: vector<std::string>& A)
 2 {
 3     int comparisons = 0;
 4     int n = A.size();
 5 
 6     for (int i = 0; i < n - 1; ++i) {
 7         // index of lowest value element
 8         int low = i;
 9 
10         for (int j = i + 1; j < n; ++j) {
11             comparisons++;
12             // loop through unsorted part of array, updating new
      low index is one is found
13             if (makeLower(A[j]) < makeLower(A[low])) {
14                 low = j;
15             }
16         }
17 
18         // perform swap if necesary
19         if (low != i) {
20             std::swap(A[i], A[low]);
21         }
22     }
23     std::cout << "Selection Sort Comparisons: " << comparisons << "
      \n" << std::endl;
24 };
```

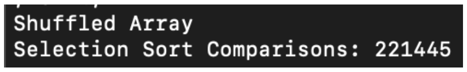
Selection Sort uses a comparison based sorting method similar to sorting cards in your hand. Starting at the first element at index `i`, a nested for loop compares the remaining indices from `i+1` (`Lines 6-10`). `low` holds the value of element `i` which is the current index to be sorted. Helper function `makeLower()` is used to sanitize the strings for comparison by putting all characters to lowercase (`Line 13`), allowing the characters that make them up to be compared as integer values. If vector index `j` is less then the value of `low`, `j` becomes the new low value (`Lines 12-13`). After all elements after `i` have been checked, swap the element with that of the new `low` value if a new lowest value has been found (`Line 19-20`).

```
std::string makeLower(std::string s) {
    std::transform(s.begin(), s.end(), s.begin(),
        [](unsigned char c){ return std::tolower(c); });
    return s;
};
```

Figure 1: makeLower Helper Function

2.1.1 Time Complexity

The asymptotic runtime of Selection sort is $O(n^2)$, where n is the number of elements in the input array. This is because of the nested loops that traverse the entire remaining unsorted array for each index of that array. The outer loop runs n-1 times, while the inner loop scans from the current position to the end of the array, scanning n-1 elements the first iteration, then n-2 the second, and so on. The total number of comparisons works out to be $n(n-1)/2$, or $(n^2 - n)/2$ simplified. Dropping the constant factors, we are left with n^2, which is thus the time complexity of the algorithm. Running the algorithm on a list of 666 strings, we can see that there are 221445 comparisons, which is exactly $666(666-1)/2$ evaluated.



```
Shuffled Array
Selection Sort Comparisons: 221445
```

Figure 2: Comparison count of Selection Sort on list of 666 items

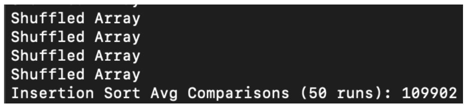
2.2 Insertion Sort

```
1  void insertionSort(std::vector<std::string>& A)
2  {
3      int comparisons = 0;
4      int n = A.size();
5      // Starting at the second element, traverse array
6      for (int i = 1; i < n; ++i) {
7          std::string key = A[i];
8          int j = i - 1;
9
10         /* Starting with 1 - the current index, compare key to
       previous element and swap if required,
11          working backwards until all previous elements have been
       compared  */
12         while (j >= 0) {
13             comparisons++;
14             if (makeLower(A[j]) < makeLower(key)) break;
15             A[j + 1] = A[j];
16             j = j - 1;
17         }
18         A[j + 1] = key;
19     }
20     std::cout << "Insertion Sort Comparisons: " << comparisons << "
       \n" <<std::endl;
21 };
```

Insertion Sort uses a similar approach as Selection Sort, however the main difference lies in the function of the inner nested loop. Rather than scanning the unsorted portion of the array, Insertion Sort compares each index to what's already been sorted, placing it in the correct position amongst the other already sorted elements and gradually filling it out. Specifically, the current index i of the outer loop is used as the key, and j is declared as one less than this current element (`Lines 7-8`). Decrementing from j to 0, each element in the unsorted portion is compared to the key, which breaks if the element at index j is less than the key, as the new insertion point has been found (`Line 14`). If the correct position has not been found, the element at j is shifted one to the right, and comparison continues (`Lines 15-16`). The once the while loop is broken, the key is inserted into the found correct position, and the outer loop increments to continue the sort (`Line 18`).

2.2.1 Time Complexity

The asymptotic runtime of Insertion Sort is $O(n^2)$, where n is the number of elements in the input array. Like Selection Sort, the nesting of loops for array traversal drastically increases the number of comparisons made to sort the list. In the worst case of a reversed order list, Insertion Sort takes n-1 iterations through the outer loop, and again the inner loop runs for 1+2+...(n-2)+(n-1) iterations. This simplifies and computes out to our time complexity of n^2. However, Insertion Sort does better than Selection Sort in the case of an already sorted list, where the runtime is $O(n)$ versus $O(n^2)$ for Selection Sort. Taking the average of 50 runs, it is seen that Insertion Sort performs the sort with roughly half the comparisons of Selection Sort, owing to how cuts down on constant factors in the runtime by only comparing through the sorted section rather than the entire remaining unsorted portion each step.



```
Shuffled Array
Shuffled Array
Shuffled Array
Shuffled Array
Insertion Sort Avg Comparisons (50 runs): 109902
```

Figure 3: Average comparison count of Insertion Sort on list of 666 items for 50 runs

2.3 Merge Sort

```cpp
1  void mergeSort(std::vector<std::string>& A, int left, int right,
       int& comparisons)
2  {
3      if (left < right) {
4          int mid = left + (right - left) / 2;
5  
6          // Recursively divide an array into sorted sub arrays of
       size 1, and then merge back the sorted subarrys
7          mergeSort(A, left, mid, comparisons);
8          mergeSort(A, mid + 1, right, comparisons);
9          merge(A, left, right, mid, comparisons);
10     }
11 };
```

Merge Sort employs a divide-and-conquer strategy to sort an array. The main function `mergeSort()` implements this strategy through recursive calls to itself. After ensuring valid bounds, a midpoint is calculated from the right and left indices of the array (`Line 3-4`). This midpoint is then used in two recursive calls to `mergeSort()`, one for each half of the array. This progressively halves the array until each array is of length 1, at which point the array can begin to merge back (`Line 7-8`). This is achieved through use of the `merge()` helper function, which is the crux of Merge Sort.

```
1  void merge(std::vector<std::string>& A, int left, int right, int
       mid, int& comparisons) {
2      int n1 = mid - left + 1;
3      int n2 = right - mid;
4  
5      // Create & populate temp arrays
6      std::vector<std::string> L(n1);
7      for (int i = 0; i < n1; i++){
8          L[i] = A[left + i];
9      };
10 
11     std::vector<std::string> R(n2);
12     for (int j = 0; j < n2; j++) {
13         R[j] = A[mid + 1 + j];
14     };
15 
16 
17     int i = 0;
18     int j = 0;
19     int k = left;
20 
21     while (i < n1 && j < n2) {
22         comparisons++;
23 
24         // Merge each half back together by comparing elements from
        each subarry and placing the smaller of the two
25         if (makeLower(L[i]) <= makeLower(R[j])) {
26             A[k] = L[i];
27             i++;
28         }
29         else {
30             A[k] = R[j];
31             j++;
32         }
33         k++;
34     }
35     while (i < n1) {
36         A[k] = L[i];
37         i++;
38         k++;
39     }
40 
41     while (j < n2) {
42         A[k] = R[j];
43         j++;
44         k++;
45     }
46 };
```

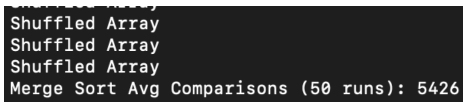
Figure 4: merge() Helper Function

The merge function is responsible for combining two sorted subarrays into a single sorted array. Temporary arrays L and R are created for the respective left and right subarrays (`Lines 2-14`). Elements from both subarrays are then compared, selecting the smaller (or equal) element to place in the original array

(`Lines 23-33`). This process continues until all elements from both subarrays have been placed back into the original array in sorted order. If there are any remaining elements in either subarray, they are appended to the end of the original array (`Lines 34-44`).

2.3.1 Time Complexity

In all cases, Merge Sort has an asymptotic runtime of $O(n \log n)$ The $\log n$ factor comes from the recursive dividing of the array into halves. Each division reduces the problem size by half, and this division continues until we reach subarrays of size 1. The number of such divisions is $\log n$. The n factor comes from the merge step. At each level of recursion, we need to merge all n elements. Even though we're working with smaller subarrays, when we add up all the merge operations across one level of the recursion tree, it sums to n. Multiplying these factors gives us the overall time complexity of $O(n \log n)$. Taking the average of 50 runs on 666 inputs, it is seen that Merge Sort performs significantly fewer comparisons (roughly 5500) than both Selection Sort and Insertion Sort, which holds especially true for larger input sizes. However, although is it much quicker than both previously discussed sort, it has the disadvantage of not being an in-place algorithm, requiring O(n) extra space to sort an array of size n.



```
Shuffled Array
Shuffled Array
Shuffled Array
Merge Sort Avg Comparisons (50 runs): 5426
```

Figure 5: Average comparison count of Merge Sort on list of 666 items for 50 runs

2.4 Quick Sort

```
1  void quickSort(std::vector<std::string>& A, int low, int high, int&
       comparisons)
2  {
3      if (low < high) {
4          //  Partition array around a chosen pivot into low and high
       subarray halves,
5          //  returning the partition to the middle of each
6          int part = partition(A, low, high, comparisons);
7
8          // Recursively sort each smaller subarray of either half by
       further partitioning
9          // and calls to quickSort
10         quickSort(A, low, part-1, comparisons);
11         quickSort(A, part+1, high, comparisons);
12
13     }
14 }
```

Quick Sort is another divide-and-conquer algorithm for sorting an array. The main function `quickSort()` implements this strategy through recursive calls to itself and a helper function `partition()`. After ensuring valid bounds, the function calls `partition()` to divide the array around a pivot (`Lines 3-4`). This partition index is then used in two recursive calls to `quickSort()`, one for each subarray on either side of the pivot (`Line 10-11`). This process continues until the subarrays have 1 or 0 elements.

```
 1 int partition(std::vector<std::string>& A, int low, int high, int&
       comparisons)
 2 {
 3     // Using high as pivot, sort array between elements below than
       the pivot and those above
 4     std::string pivot = A[high];
 5     int i = low - 1;
 6     for (int j = low; j < high; j++) {
 7         comparisons++;
 8         if (makeLower(A[j]) <= makeLower(pivot)) {
 9             i++;
10             std::swap(A[i], A[j]);
11         }
12     }
13 
14     // Return pivot to middle position between each half
15     std::swap(A[i+1], A[high]);
16     return (i+1);
17 };
```

Figure 6: partition() Helper Function

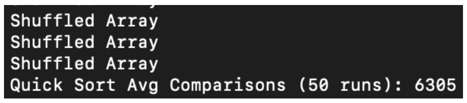
The `partition()` function is the core of Quick Sort. It chooses the last element as the pivot (`Line 3`) and places it in its correct position in the sorted array. It maintains two pointers: `i`, which keeps track of the boundary between elements smaller than or equal to the pivot and those larger than the pivot, and `j`, which scans through the array. Elements smaller than or equal to the pivot are swapped to the left side of the array(`Lines 6-12`). After the loop, the pivot is placed in its final sorted position (`Line 15-16`).

2.4.1 Time Complexity

The time complexity of Quick Sort varies depending on the choice of pivot and the input data:

- Best and Average Case: $O(n \log n)$
- Worst Case: $O(n^2)$

The $\log n$ factor in the best and average cases comes from the recursive division of the array. Each division reduces the problem size, and this division continues until we reach subarrays of size 1 or 0. The n factor comes from the partitioning step, which compares each element to the pivot. The worst case occurs when the pivot is always the smallest or largest element, leading to unbalanced partitions. This happens for already sorted or reverse sorted arrays if we always choose the first or last element as the pivot. While Quick Sort and Merge Sort both have an average time complexity of $O(n \log n)$, Quick Sort is often faster in practice due to its in-place nature and better cache performance. However, Merge Sort provides a guaranteed $O(n \log n)$ worst-case performance, making it preferable for certain performance critical applications. Quick Sort's partition step can be also be optimized for certain data distribution types, outperforming Merge Sort on these inputs. Taking the average of 50 runs on 666 inputs, it is seen that Quick Sort generally performs fewer comparisons than both Selection Sort and Insertion Sort, especially for larger input sizes. Unlike Merge Sort, Quick Sort is an in-place algorithm, requiring only $O(\log n)$ additional space for the recursion stack in the average case.



```
Shuffled Array
Shuffled Array
Shuffled Array
Quick Sort Avg Comparisons (50 runs): 6305
```

Figure 7: Average comparison count of Quick Sort on list of 666 items for 50 runs

2.5 Asymptotic Analysis

Algorithm	**Avg. Comparisons (666 inputs)**	**Time Complexity**			**Stable**	**In-Place**
		Best	**Average**	**Worst**		
Selection Sort	$\approx 221,000$	$O(n^2)$	$O(n^2)$	$O(n^2)$	No	Yes
Insertion Sort	$\approx 111,000$	$O(n)$	$O(n^2)$	$O(n^2)$	Yes	Yes
Merge Sort	$\approx 5,500$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Yes	No
Quick Sort	$\approx 6,800$	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	No	Yes

Table 1: Comparison of Sorting Algorithms