# Assignment One – Structures & Sorting

Ryan Munger
Ryan.Munger1@marist.edu

September 24, 2024

## 1 Introduction

### 1.1 Goals

Assignment 1 instructed us to create the elementary data structures *Stack* and *Queue* using a node implementation. We then checked a list of items for palindromes using these data structures and also sorted the list using selection, insertion, merge, and quick sorts.

### 1.2 Write-up Format

In this report I will describe the logic being presented. Below the text explanation, relevant code will follow. I implemented the foundational parts of this assignment in both C++ and Ada. I was originally planning to use Pascal, but became frustrated with the nuances of different versions and compilers. As I was writing Pascal, I found it incredibly similar to a language I have seen before - Ada (Not like its based on Pascal or anything...). Before you ask, I work in the defense industry. I used this as an opportunity to actually learn Ada so that I can be better at my job with Lockheed Martin. I used Alire to build and run my Ada code. Alire is extremely similar to Cargo for Rust. Under the hood, it uses GNAT to build the Ada program. Long story short, walrus assignment is pretty cool. I will attempt to sprinkle in some jokes for you!

### 1.3 Limerick of Luck

Limerick of Luck would make a good magic item...
I suggest adding Nogard Dragon as a new palindrome!

All programmers revere the power of sorting
Especially after conducting it with no importing.
    Despite some having large big-oh,
    Merge sort hasn't yet delivered a deathblow,
Allowing us to experiment with algorithms so cavorting.

# 2 Nodes, Linked Lists, Stacks, & Queues

## 2.1 Making a Node

Nodes are the foundations of linked lists. Other data structures including stacks and queues are special forms of linked lists. To create nodes capable of forming a singly linked list (meaning nodes know where to find the next node but not the previous), they need a value and a pointer to the following node. I used a template in my implementation so that we can use the nodes with any type of data. A nodes value and next node can be accessed publicly. The Ada implementation is limited to char nodes.

```
template <typename T>
struct Node {
    T value;
    Node<T>* next;
};
```

```
   type Node;
   type Node_Ptr is access Node;

   type Node is
      record
         Data : Character;
         Next : Node_Ptr;
      end record;
```

## 2.2 Testing our Nodes

Code:

```
    cout << "Node Testing:" << endl;
    Node<string>* firstNode = new Node<string>;
    firstNode->value = "Magic Item1";
    Node<string>* secondNode = new Node<string>;
    secondNode->value = "Nogard Dragon";
    firstNode->next = secondNode;
    cout << "First Node Value: " << firstNode->value << endl;
    cout << "Second Node Value: " << secondNode->value << endl;
    if (firstNode->next == secondNode) {
        cout << "First node points to the second node." << endl;
    } else {
        cout << "First node does not point to the second node." <<
    endl;
    }
```

Output:

```
Node Testing:
First Node Value: Magic Item1
Second Node Value: Nogard Dragon
First node points to the second node.
```

Code:

```
   N1: Node_Ptr := new Node;
   N2: Node_Ptr := new Node;
begin
   N1.Data := 'a';
   N2.Data := 'b';
   N1.Next := N2;
   Put_Line("Node1 Value: " & N1.Data);
   Put_Line("Node2 Value: " & N2.Data);
   if N1.Next = N2 then
      Put_Line("Node1 Points to Node2!");
   else
      Put_Line("Node1 Does not Point to Node2!");
   end if;
```

Output:

```
Success: Build finished successfully in 1.11 seconds.
Node1 Value: a
Node2 Value: b
Node1 Points to Node2!
```

## 2.3 Stacks & Queues

Nodes can link together to form linked lists. Stacks and queues are linked lists with special rules. Stacks can only be managed from the top, similar to a stack of dining hall plates. You can add or remove from the top of the stack but nothing more (this is called LIFO - Last In First Out). Queues are like a line to enter a dining hall; you can only add things to the end of the line and take from the front (FIFO - First In First Out).
A stack needs - push (add to stack), pop (remove), isEmpty (see if it has anything in it), and optionally, size & peek. Queues need - enqueue (add to end of line), dequeue (remove from front), isEmpty, and optionally, size & peek. All of these operations are in constant time (as my queue has a tail pointer).

The stack needs a pointer to the top of the stack just as a queue keeps track of its head and tail.

**Stack Declaration:**

```
template <typename T>
class Stack {
    private:
        Node<T>* top;
        int size;

    public:
        // constructor
        Stack() {
            top = nullptr;
            size = 0;
        }
        // Destructor to free memory from nodes within stack if stack is deleted
        ~Stack() {
            while (!isEmpty()) {
                pop();
            }
        }
```

```
   type Stack is
      record
         Top : Node_Ptr := null;
      end record;
```

**Stack Push:**

1. Store new data into a node.
2. The new next node is the current top.
3. The new top is the new node.
4. Increase stack size (optional).

```
        void push(const T value) {
            // create new node, make it the top & point to the old top
            Node<T>* newItem = new Node<T>;
            newItem->value = value;
            newItem->next = top;
            top = newItem;
            size++;
        }
```

```
   procedure Push(S : in out Stack; Value : Character) is
      New_Node : Node_Ptr := new Node;
   begin
      New_Node.Data := Value;
      New_Node.Next := S.Top;
      S.Top := New_Node;
   end Push;
```

**Stack Pop:**

1. Ensure stack not empty.
2. Note the value of current top.
3. Make the node after the current top the new top.
4. Reduce stack size (optional).
5. Delete the old node from memory (garbage collection).
6. Return the value of the old top node.

```
        T pop() {
            if(isEmpty()) {
                throw std::runtime_error("Stack is empty, cannot pop.");
            }

            T value = top->value;
            Node<T>* newTop = top->next;
            delete top;
            top = newTop;
            size--;

            return value;
        }
```

```
   function Pop(S : in out Stack) return Character is
      Temp_Node : Node_Ptr;
      Value : Character;
   begin
      if S.Top = null then
         raise Program_Error;  -- Stack underflow?
      else
         Temp_Node := S.Top;
         Value := Temp_Node.Data;
         S.Top := S.Top.Next;
         return Value;
      end if;
   end Pop;
```

**Stack isEmpty:**

1. Returns true if the top of the stack is a nullptr.

**Stack getSize:**

1. Returns value of size attribute.

**Stack Peek:**

1. Returns the value of the top node without popping it.

---

Why did the stack go to therapy? - *It keeps pushing things down...*

---

**Testing Stack:**

Code:

```
    cout << "\nStack Test: [Hello, World, !]" << endl;
    Stack<string> stack;
    stack.push("Hello");
    stack.push("World");
    stack.push("!");
    int size = stack.getSize();
    for (int i = 0; i < size; i++) {
        cout << "Pop: " << stack.pop() << endl;
    }
```

Output:

```
Stack Test: [Hello, World, !]
Pop: !
Pop: World
Pop: Hello
```

As previously mentioned, the queue class will need to have a pointer to the first and the last node to work in constant time. It is possible traverse the queue to obtain the final node, but the trade off between memory and time is worth it in this case to store an additional pointer.

**Queue Implementation:**

```
template <typename T>
class Queue {
    private:
        Node<T>* head;
        Node<T>* tail;
        int size;

    public:
        Queue() {
            head = nullptr;
            tail = nullptr;
            size = 0;
        }

        ~Queue() {
            while (!isEmpty()) {
                dequeue();
            }
        }
```

```
   type Queue is
      record
         Front : Node_Ptr := null;
         Rear  : Node_Ptr := null;
      end record;
```

**Queue Enqueue:**

1. Create a new node with the new value.
2. If queue empty, new node is the head.
3. Otherwise, set the current tail's next value to the new node.
4. Make the new node the tail.
5. Increment size (optional).

```cpp
    void enqueue(const T value) {
        // make a new node
        Node<T>* newItem = new Node<T>;
        newItem->value = value;
        newItem->next = nullptr;

        // add the node to the back of the queue
        if (isEmpty()) {
            head = tail = newItem;
        } else {
            tail->next = newItem;
            tail = newItem;
        }
        size++;
    }
```

```ada
procedure Enqueue(Q : in out Queue; Value : Character) is
   New_Node : Node_Ptr := new Node;
begin
   New_Node.Data := Value;
   New_Node.Next := null;

   if Q.Rear = null then
      -- Queue is empty, so Front and Rear point to the new node
      Q.Front := New_Node;
      Q.Rear := New_Node;
   else
      -- Append to the rear of the queue
      Q.Rear.Next := New_Node;
      Q.Rear := New_Node;
   end if;
end Enqueue;
```

**Queue Dequeue:**

1. Make sure queue is not empty.
2. Note the value of the current head.
3. Set the new head to the current head's next.
4. Free up memory from the now old head.
5. Check if the head is now null, if so, set tail to null.

6. Decrement size (optional).
7. Return the value of now the old head node.

```
        T dequeue() {
            if(isEmpty()) {
                throw std::runtime_error("Queue is empty, cannot
 dequeue.");
            }

            T value = head->value;
            Node<T>* oldHead = head;
            head = oldHead->next;
            delete oldHead;
            size--;

            if (head == nullptr) {
                tail = nullptr;
            }
            return value;
        }
```

```
   function Dequeue(Q : in out Queue) return Character is
      Temp_Node : Node_Ptr;
      Value : Character;
   begin
      if Q.Front = null then
         raise Program_Error;  -- Queue underflow?
      else
         Temp_Node := Q.Front;
         Value := Temp_Node.Data;
         Q.Front := Q.Front.Next;

         --  if front null its empty so rear null
         if Q.Front = null then
            Q.Rear := null;
         end if;

         return Value;
      end if;
   end Dequeue;
```

**Queue isEmpty:**

1. Return true if the head and the tail are null.

**Queue peekFront:**

1. Return value of head node.

**Queue peekBack:**

1. Return value of tail node.

**Queue getSize:**

1. Return value of size attribute.

---

Why did the stack break up with the queue? - *It found a priority queue to put it first...*

---

**Testing Queue:**

Code:

```
    cout << "\nQueue Test: [Hello, World, !]" << endl;
    Queue<string> q;
    q.enqueue("Hello");
    q.enqueue("World");
    q.enqueue("!");
    size = q.getSize();
    for (int i = 0; i < size; i++) {
        cout << "Dequeue: " << q.dequeue() << endl;
    }
```

Output:

```
Queue Test: [Hello, World, !]
Dequeue: Hello
Dequeue: World
Dequeue: !
```

## 2.4 Palindrome Checking: A Stack & Queue Use Case

A palindrome is word/phrase that reads the same forward and backwards. For our implementation, we will ignore spaces and case. "Racecar" is a palindrome because the letters read the same front to back and back to front. "Nogard Dragon" should be in the ones we are checking because I think its cool (is that not reason enough?). We read in the list of words/phrases to check from magicitems.txt and determine if it is a palindrome by:

1. Declaring a stack and a queue.
2. For each char in the item, push/enqueue each char to each respective data structure in lower case if it is not a space.
3. Pop each element from the stack and dequeue each element from the queue. If they do not match, it is not a palindrome! (As stack has the word in reverse order).

**Read in magic items:**

```
vector<string> getMagicItems(const string& filename) {
    vector<string> magicItems;
    ifstream file(filename); //input file stream
    if (!file) {
        cerr << "File opening failed." << endl;
    }
    string line;
    while (getline(file, line)) {
```

```cpp
        magicItems.push_back(line); // add line to vector
    }
    file.close();
    return magicItems;
};
```

```ada
   procedure Check_Magicitems(File_Name : String) is
      File : File_Type;
      Line : String (1 .. 100);  -- assuming max line length is 100
     characters
      Length : Natural;
   begin
      Open (File, In_File, File_Name);
      while not End_Of_File (File) loop
         Get_Line (File, Line, Length);

         if Is_Palindrome(Line (1 .. Length)) then
            Put_Line(Line (1 .. Length) & " is a palindrome.");
         end if;

      end loop;
      Close (File);
   end Check_Magicitems;
```

**Check for palindromes:**

```cpp
bool isPalindrome(const string& word) {
    Stack<char> s; Queue<char> q;
    for (char letter: word) {
        if (!isspace(letter)) {
            if (isalpha(letter)) {
                letter = tolower(letter);
            }
            s.push(letter);
            q.enqueue(letter);
        }
    }

    int size = s.getSize();
    for (int i = 0; i < size; i++) {
        if (s.pop() != q.dequeue()) {
            return false;
        }
    }
    return true;
};
```

```ada
   function Is_Palindrome(Input : String) return Boolean is
      S : Stack;
      Q : Queue;
      Len : Integer := Input'Length;
   begin
      -- Load the string into both the stack and the queue
      -- Ingore case and space
      for I in 1 .. Len loop
         if Input(I) /= ' ' then
            Push(S, Ada.Characters.Handling.To_Lower(Input(I)));
            Enqueue(Q, Ada.Characters.Handling.To_Lower(Input(I)));
```

```
            end if;
         end loop;

         -- Pop, dequeue, compare
         for I in 1 .. Len loop
            if Input(I) /= ' ' then
               if Pop(S) /= Dequeue(Q) then
                  return False;
               end if;
            end if;
         end loop;

         return True;
      end Is_Palindrome;
```

**Results:**

```
Boccob is a palindrome.
Seuss Igniting Issues is a palindrome.
UFO tofu is a palindrome.
Ebuc Cube is a palindrome.
Aibohphobia is a palindrome.
Taco cat is a palindrome.
Was It A Rat I Saw is a palindrome.
Olah Halo is a palindrome.
CD case divides ACDC is a palindrome.
Dacad is a palindrome.
Dior Droid is a palindrome.
Robot Tobor is a palindrome.
Narc in a panic ran is a palindrome.
radar is a palindrome.
Golf flog is a palindrome.
```

---

Why is 'palindrome' not a palindrome? Dr. Labouseur and other detail oriented people surely don't get annoyed by this...

---

# 3 Sorting

## 3.1 Introduction

In this section we will sort our list alphabetically from least (a) to greatest (z). The $\leq$ relation is a **total order** and must uphold the following:

1. Reflexivity: For all x in S, $x \leq x$.
2. Antisymmetry: For all x, y in S, if $x \leq y$ and $y \leq x$, then $x = y$.
3. Transitivity: For all x, y, z in S, if $x \leq y$ and $y \leq z$, then $x \leq z$.

Good 'ol discrete math...

I implemented this section in c++ only.

## 3.2 Unsorting?

A truly unsorted array is one in which the elements are randomly positioned. This array can technically be sorted (the basis for the monkey/bogo sort), but we won't actually know until we check. Since we are implementing several sorting algorithms, we will need to unsort/shuffle the array between sorts. A good algorithm to do this randomly & in place is the Knuth/Fisher Yates Shuffle. It iterates through each data point in an array, picking a random index between current index and all remaining indices and swaps the values between the two.

```
void knuthShuffle(vector<string>& arr) {
    int n = arr.size();
    for (int i = n-1; i > 0; i--) {
        int random = rand() % (i + 1); // inclusive to swap in place
        swap(arr[i], arr[random]);
    }
};
```

## 3.3 Selection Sort

Selection sort is an in place sort that organizes an array using a nested loop. Selection sort iterates through the array except for the portion is has already sorted and finds the minimum value (when doing $\leq$). Once we find the minimal element, we swap it with the current index. This way, if we are on array index 2 (starting at 0) we would swap in the 3rd smallest value. This algorithm works in $O(n^2)$ time because the outer loop runs $n$ times and the inner loop has to iterate over the entire array it hasn't yet sorted. Despite the fact that 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; therefore $n*n$ comparisons (where $n$ is size of the array). A more precise characterization of this algorithm is as follows:
Total comparisons: $(n-1)+(n-2)+\cdots+1=\sum_{i=1}^{n-1} i$
$\sum_{i=1}^{n-1} i = \frac{(n-1)+1}{2}(n-1) = \frac{1}{2}n(n-1) = \frac{1}{2}(n^2-n)$
As we can see, in this case $n^2$ dominates therefore the algorithm is $O(n^2)$. Using this formula on our array of 666 elements, we can expect $\frac{1}{2}(666^2-666)$ or 221,445 comparisons to be made. It will always take this exact number of comparisons, even if the array is presorted or totally unsorted.

```
int selectionSort(vector<string>& arr) {
    int n = arr.size();
    int comparisons = 0;

    for (int i = 0; i < n; i++){
        int minIdx = i;
        for (int j = i+1; j < n; j++) {
            comparisons++;
            if (toLowerCase(arr[j]) < toLowerCase(arr[minIdx])){
                minIdx = j;
            }
        }
```

```
        if (minIdx != i) {
            swap(arr[i], arr[minIdx]);
        }
    }
    return comparisons;
};
```

---

Why did selection sort's girlfriend leave him? - *Turns out checking out every option before selecting her wasn't flattering...*

---

### 3.4 Insertion Sort

Insertion sort is an in place sort that is similar to selection sort. Instead of iterating through the entire array to find the item destined for the next position in the array, insertion sort simply grabs the next item and puts it in the correct location. Instead of searching forward for the most optimal value, insertion sort *inserts* the next value it is looking at in its correct relative position. The exact time complexity of this algorithm is difficult to calculate directly as it depends on the order the array is in, but the worst case is that the inner loop will have to iterate through the entire sorted portion of the array to find the correct location for each item. The worst case makes insertion sort $O(n^2)$ since the length of the inner array is approximately $n/2$ on average. A complexity of $n * \frac{1}{2}n$ without constants is $n^2$. For our array of 666 elements, we can expect between best case (sorted: $n-1$ comparisons) or worst case (same as selection sort), or 665 - 221,445. Averaging these, we should expect generally around 111,388 comparisons.

```
int insertionSort(vector<string>& arr) {
    int n = arr.size();
    int comparisons = 0;

    for (int i = 1; i < n; i++) {
        int insertIdx = i;
        string currentCheck = arr[i];
        for (int j = i-1; j >= 0; j--) {
            comparisons++;
            if (toLowerCase(arr[j]) > toLowerCase(currentCheck)) {
                arr[j+1] = arr[j];
                insertIdx = j;
            } else {
                break;
            }
        }
        arr[insertIdx] = currentCheck;
    }
    return comparisons;
};
```

## 3.5 Merge Sort

Merge sort is a divide and conquer algorithm that is not in place. It works by recursively dividing the array into smaller subarrays and sorting those subarrays. The subarrays are sorted by dividing them just as we did with the input array. The base case is an array of size one, which is sorted! Once we reach subarrays of size one, we then merge them back together all the way up the recursive tree until we have a completely sorted array.

To be concise, merge sort divides the array into halves, sorts each half, and then merges the sorted halves back together (and does so for each half of the half... etc.). This process is repeated until the entire array is sorted. Dividing the array is a constant time operation. There are $log_2 n$ divides as the array is cut in half each recursion. Each divide step must also be merged back together. There are n items to merge back together at each subarray size. This operation will take $n$ time. Therefore, the time complexity is $O(nlog_2 n)$. We should then expect $666 * log_2 666$ which is 6,244 comparisons.

Merge sort is driven by a recusrively called function that

1. Checks if the subarray can be divided further. If not, array is of size 1 and is sorted (base case).
2. Makes a recursive call to divide the subarray further if it can.
3. Calls a helper function to stitch the two halves back together in a sorted order.

```
void mergeSort(vector<string>& arr, int left, int right, int& comparisons) {
    // base case
    if (left >= right)
        return;

    // left < right, array can be divided further
    // calculate mid of subarray/array to split array
    int mid = left + (right - left) / 2;

    // recursively sort the halves, update comparisons
    mergeSort(arr, left, mid, comparisons);
    mergeSort(arr, mid + 1, right, comparisons);

    // merge sorted halves, update comparisons
    merge(arr, left, mid, right, comparisons);
};
```

The helper function first builds the two subarrays we are to work with (the only subarrays that get to this function are sorted). We then write the subarray data back into the original array in the correct order. Combining the two arrays is simple as we can just use a while loop to see which next element in either array is smaller since the subarrays are sorted. Once this is completed, the two subarrays have been merged successfully.

```
// referenced geeksforgeeks.org
void merge(vector<string>& arr, int left, int mid, int right, int&
    comparisons) {
    int nLeft = mid - left + 1; // size of left array
    int nRight = right - mid; // size of right array

    // dec temp arrays
    vector<string> leftArr(nLeft), rightArr(nRight);

    // populate temp vectors
    for (int i = 0; i < nLeft; i++)
        leftArr[i] = arr[left + i];
    for (int j = 0; j < nRight; j++)
        rightArr[j] = arr[mid + 1 + j];

    int i = 0, j = 0;
    int k = left;

    // merge temp vectors back into arr[left..right]
    while (i < nLeft && j < nRight) {
        comparisons++;
        if (toLowerCase(leftArr[i]) <= toLowerCase(rightArr[j])) {
            arr[k] = leftArr[i];
            i++;
        }
        else {
            arr[k] = rightArr[j];
            j++;
        }
        k++;
    }

    // copy the remaining elements of leftArr after merging
    while (i < nLeft) {
        arr[k] = leftArr[i];
        i++;
        k++;
    }

    // copy the remaining elements of right after merging
    while (j < nRight) {
        arr[k] = rightArr[j];
        j++;
        k++;
    }
};
```

---

A mad scientest has created a self sorting array! He pulls back the curtain and to the crowd's dismay yells: *"BEHOLD! A array of size one!!"*

---

## 3.6 Quick Sort

Quick sort also utilizes a divide and conquer methodology. Similar to merge sort, it recursively divides the array using a pivot value. It partitions the array by splitting the array at the pivot value by grouping items less than the pivot into one partition and the rest into another. There are many different versions of the partition algorithm, but I chose the Lomuto partition as it is easy to understand. Lomuto partition keeps track of the indices of smaller elements and keep swapping, always using the last element as the pivot. The image below from GeeksforGeeks.org does a great job of visualizing the process.

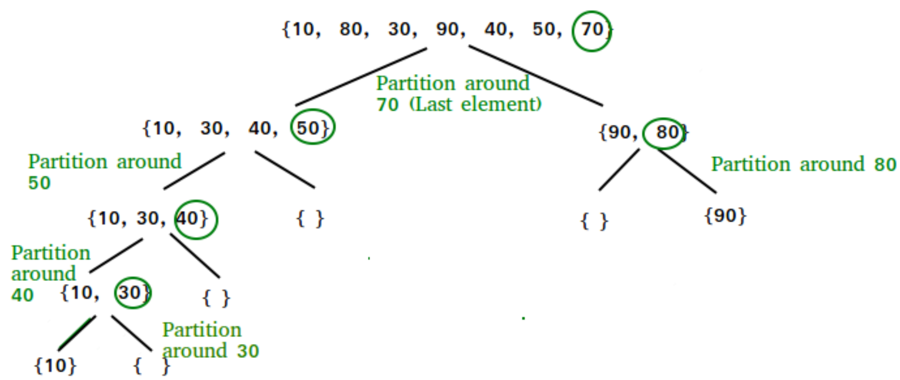


Figure 1: Lomuto Quick Sort

The time complexity for quick sort is $O(nlog_2n)$ as there are rarely more than $log_2n$ partitions (given we are choosing good pivots) and $n$ comparisons required for each partition to correctly order the elements around the pivot. The absolute worst case is that the pivot is always the largest or smallest element, which leaves us at $O(n^2)$, or 221,445 comparisons. On average however, we can expect $O(nlog_2n)$, or $666 * log_2 666$ which is 6,244 comparisons.

First, we have the recursive function:

1. Checks if the partition can be divided further. If not, array is of size 1 and is sorted (base case).

2. Call a helper function to partition the array.

3. Makes a recursive call to sort the partitions.

```
void quickSort(vector<string>& arr, int low, int high, int& comparisons) {
    // if low < high then we are not done
    if (low < high) {
        int pIdx = partition(arr, low, high, comparisons); // Partitioning index

        quickSort(arr, low, pIdx - 1, comparisons);  // recursively sort the elements before partition
        quickSort(arr, pIdx + 1, high, comparisons); // recursively sort the elements after partition
    }
};
```

The helper function is a bit more complex (we are using Lomuto).

1. Choose a pivot value (last element in partition).
2. Locate the start index of the partition.
3. Traverse partition and move all smaller elements to the left side of the pivot. This generally leaves the pivot toward the middle of the array. We can then again partition further.

Once again, GeeksforGeeks.org has an excellent graphic:

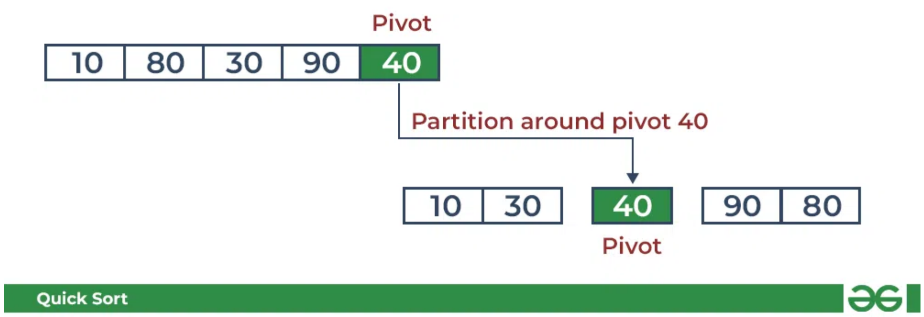


Figure 2: Pivoting

```
// referenced geeksforgeeks.org
int partition(vector<string>& arr, int low, int high, int& comparisons) {
    // choose pivot element
    string pivot = toLowerCase(arr[high]);
    // Index of smaller element + right position of pivot found so far
    int i = low - 1;

    // rearranges array so that elements less than or equal to the pivot are to the left of it
    // greater than are to the right
    for (int j = low; j <= high - 1; j++) {
```

```
        comparisons++;
        if (toLowerCase(arr[j]) < pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }

    // move pivot after smaller elements
    swap(arr[i + 1], arr[high]);
    return i + 1;
};
```

---

Why does quicksort have no friends? - *It always pivots the conversation to talk about itself...*

---

## 3.7 Comparing the Sorts

In terms of time complexity, quick sort and merge sort easily pull ahead of selection and insertion. This does not mean that selection and insertion sorts are not useful. Selection sort is an excellent teaching exercise, is easy to understand, and employs minimal overhead & swaps. Insertion sort is very quick to sort already sorted/near sorted arrays. Merge sort generally uses less comparisons than quick sort, but ends up with more overhead. Most in-built sorting libraries like sort() in Python use Timsort. Timsort is simply a hybrid sorting algorithm derived from ((merge sort or quick sort) and insertion sort). By using both algorithms, we can play on the strengths of each. By using merge sort to get the array almost sorted and insertion to finish the job, improvements are made. Since the code I implemented counts the comparisons for each sort, I will run it 20 times and average the results in a table below.

| Sort | Time Complexity | Expected | Actual |
|---|---|---|---|
| Selection Sort | $O(n^2)$ | $221,445$ | $221,445$ |
| Insertion Sort | $O(n^2)$ | $111,388$ | $112,609$ |
| Merge Sort | $O(nlog_2n)$ | $6,244$ | $5,424$ |
| Quick Sort | $O(nlog_2n)$ | $6,244$ | $6,760$ |

Selection sort is exactly as expected since it always checks every element. Insertion sort did vary a bit, but is right on par with the expected. Merge sort performed well, as it is usually a bit faster than its Big-Oh. Quick sort lagged behind, but this is likely due to my suboptimal choice of the Lomuto partitioning algorithm.

# 4 Miscellaneous Lessons Learned

If I learned anything, it is that the walrus operator is sick. I guess I learned a lot about Big Oh and sorting algorithms, but that pales in comparison. One thing I wished I knew is that you need to seed the random number generator in C++ or it will always give you the same numbers (seeding with current time is best). Aside from general Ada knowledge, I am satisfied with my newfound understanding of fundamental data structures and sorting algorithms.

---

You really read this whole thing? **WOW!**

---