Proposal

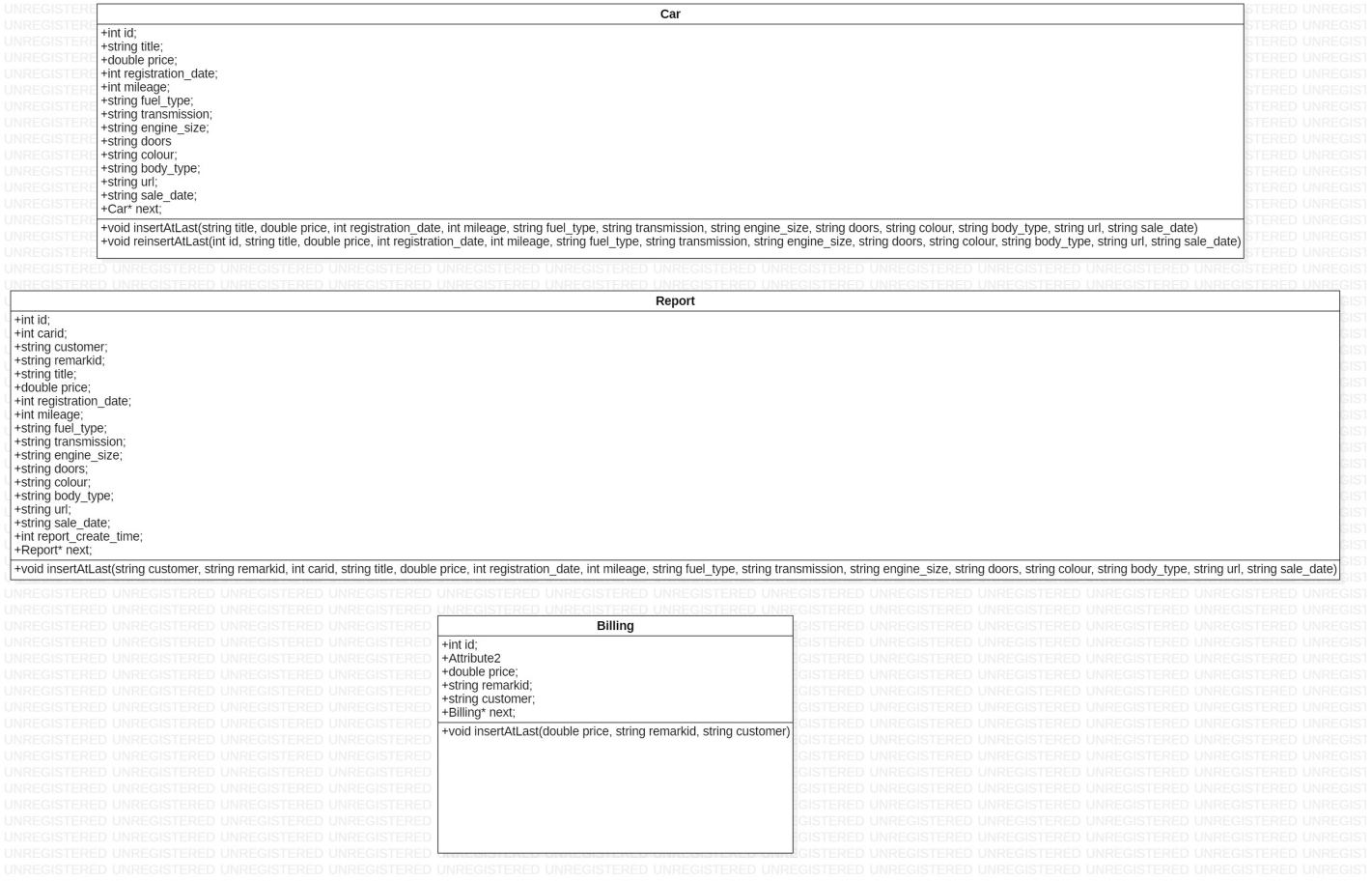
USED VEHICLE INVOCING SYSTEM

Groupmate

- Yeong Chee Chiew TP068860
- Chai Cheng Ti TP060723
- Siew Yung Hong TP060743
- Wong Yen Wei TP063782

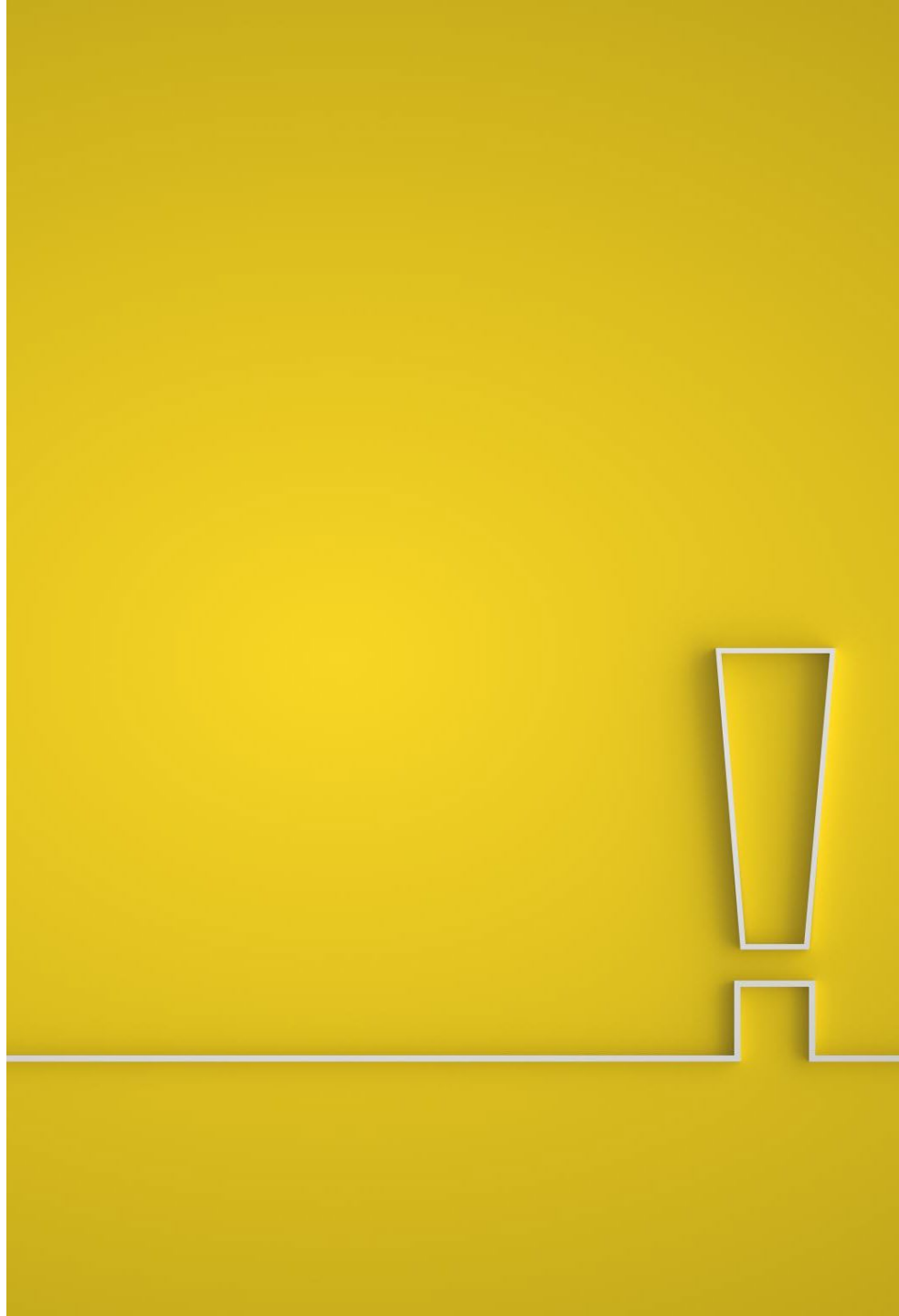
Data Structure

Visualization of data structures



Description

- In our data structure will include three structure it is car, report and bill struct.
- Every struct will have insert function for adding data.
- Every data will insert will add at the last of the struct link list.
- Only car struct will have reinsert function, it can use when delete the order we will reinsert the car detail back to the car struct link list.



Binary Search

Function:

- A searching algorithm used in a sorted array by repeatedly dividing the search interval in half.

Advantages:

- much quicker than a serial search because the data that needs to be searched halves with each step. For example, it is possible to search through 1024 values and find the one you want within 10 steps, every time.

Reason:

We choose Binary Search is to help make it easier to search for information in a big dataset, in this case we will be using it to search for client names in the dataset.

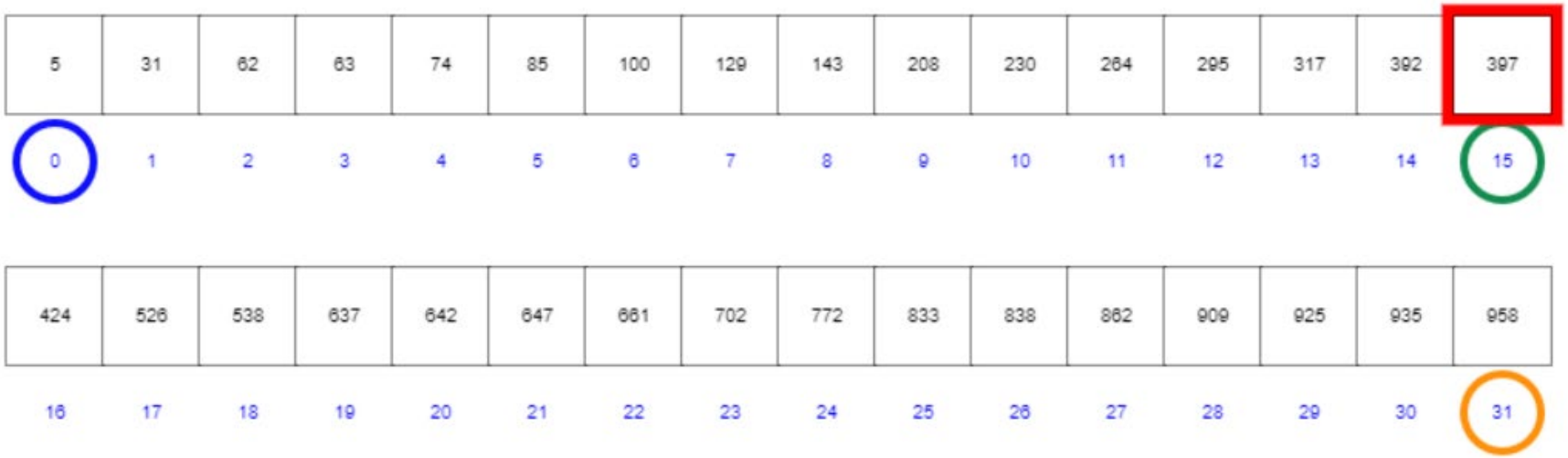
```
def binarySearch(listData, value)
    low = 0
    high = len(listData) - 1
    while (low <= high)
        mid = (low + high) / 2
        if (listData[mid] == value):
            return mid
        elif (listData[mid] < value)
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

5	31	62	63	74	85	100	129	143	208	230	264	295	317	392	397
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

424	526	538	637	642	647	661	702	772	833	838	862	909	925	935	958
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

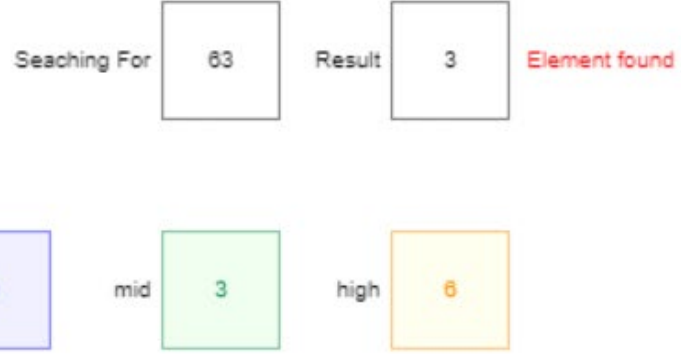
In this array we try to search 63. Now step 1 is identifying the low index to the first element of the array and the high index to the last element. As we can see the lowest element and highest element is highlighted up top.

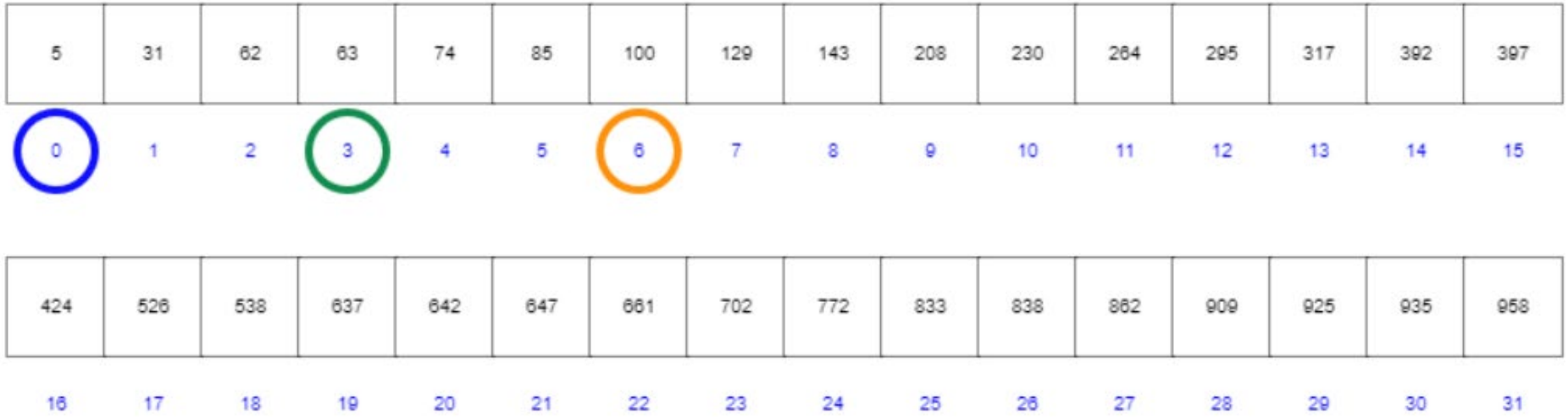
```
def binarySearch(listData, value)
    low = 0
    high = len(listData) - 1
    while (low <= high)
        mid = (low + high) / 2
        if (listData[mid] == value):
            return mid
        elif (listData[mid] < value)
            low = mid + 1
        else:
            high = mid - 1
    return -1
```



Step 2 is finding the mid point between element 1 and element 31, in this case element 15. If the number that we are searching for is not in element 15, then the highest element will be changed to one element lower than the mid point. That is element 14.

```
def binarySearch(listData, value)
    low = 0
    high = len(listData) - 1
    while (low <= high)
        mid = (low + high) / 2
        if (listData[mid] == value):
            return mid
        elif (listData[mid] < value)
            low = mid + 1
        else:
            high = mid - 1
    return -1
```





Step 3 is the same as step 2, if the value in the new midpoint of the element is not the value, we are searching for then step 2 will be repeated until the value of the mid point is the value we are searching for, thus in the pic shows the element 3 is where our value is. Value find is successful.

Linear Search

Function

- It will start at one end and go through each element of a list until the desired element is found; otherwise, the search will continue until the end of the data set.

Advantages

- Could be fast if the search item is towards the beginning of the list.

Choose Reason

- We choose linear search to search the invoce report id, mileage, and price. This is because every time we run the system, it will reset all the data to empty. As a result, the lists contain will contain fewer elements. It will search faster if you have fewer elements.

```
def linearSearch(listData, value)
    index = 0
    while (index < len(listData) and listData[index] < value):
        index++;
    if (index >= len(listData or listData[index] != value):
        return -1
    return index
```

Seaching For: 63

Result:

index: 0

18	22	63	66	102	104	147	215	237	346	392
0	1	2	3	4	5	6	7	8	9	10

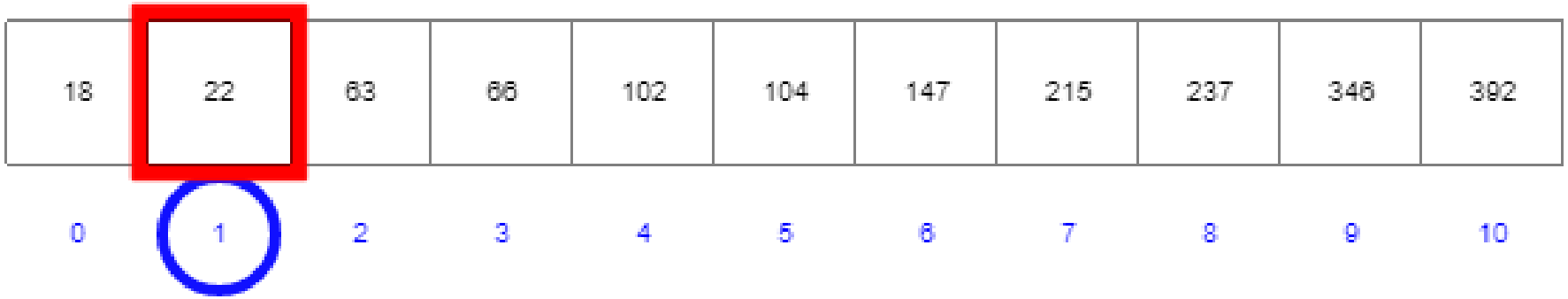
In this array we try to search 63. Now step 1 will check element 0 is same with 63; if no go to element 1.

```
def linearSearch(listData, value)
    index = 0
    while (index < len(listData) and listData[index] < value):
        index++;
    if (index >= len(listData or listData[index] != value):
        return -1
    return index
```

Seaching For: 63

Result:

index: 1



Step 2 is same with Step 1 check element 1 is same with 63; if no go to element 2.

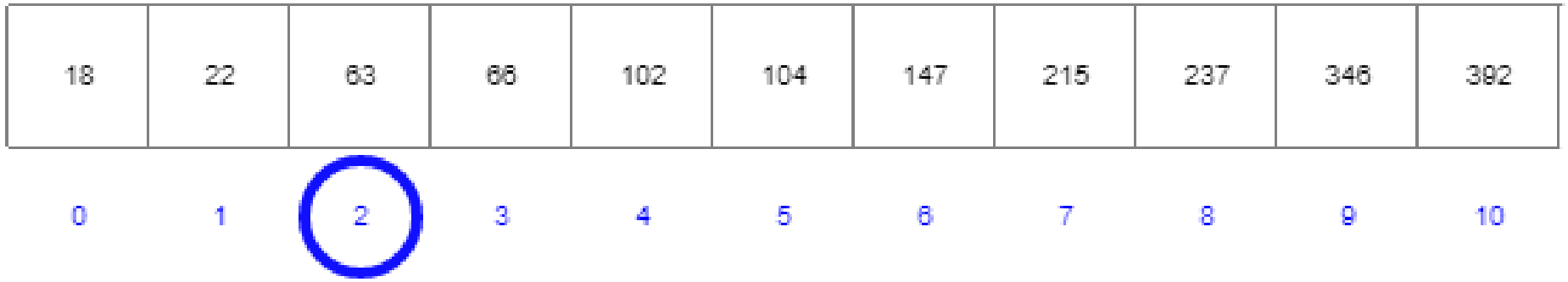
```
def linearSearch(listData, value)
    index = 0
    while (index < len(listData) and listData[index] < value):
        index++;
    if (index >= len(listData or listData[index] != value):
        return -1
    return index
```

Seaching For: 63

Result: 2

Element found

index: 2



Step 3 is same with Step 1 and Step 2 check element 2 is same with 63. Element 2 got same value with 63. Value Find Succesful.

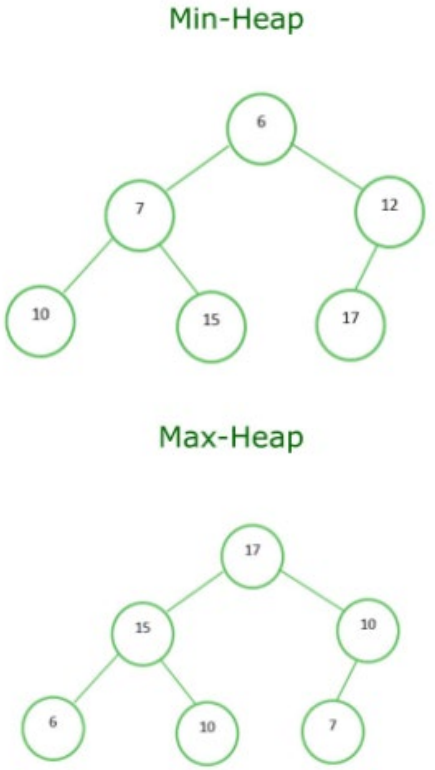
Heap Sort

What is Heap Sort? Heap Sort is considered one of the widely known technique to perform sorting of data in an array. How do we sort using heap Sort? First of all, we will find the minimum element and swap places with the maximum element so that it allows the maximum element node to be removed, the process is then repeated to acquire a sorted array.

Binary Heap is a binary tree with a few defining characteristics

- The tree must be a complete tree, meaning every level is filled with the lowest one as exception and all of the important ones are at the left side of the lowest level. This characteristics allows easy storage of data in the array
- There are two types of binary heap which is Min Heap or Max Heap. The example of Min Heap and Max Heap will be shown with the pictures on the side.

Example:



Advantages of Heap Sort

There are 3 advantages that can be found using Heap Sort algorithm over others which is:

- Simplicity - Since Heap Sort does not require any advanced computer science concepts. It is usually recommended to prioritize this algorithm over others as it is simple to understand it over other algorithms that has similar function to Heap Sort.
- Efficiency – The reason that this algorithm is preferred over the others is that it is very time efficient. One way to explain this is that when the number of items needed to be sort increases, other algorithms may grow very slow depending the amount of items added while Heap Sort increases logarithmically, this is why time usage in sorting is important.
- Memory Usage – Compared to other algorithm, Heap Sort only prioritize the required amount of memory needed to store the list of items into array, since it requires no additional memory to work as intended, the memory usage required by Heap Sort is very minimal.

Heapify Method

What is Heapify? Heapify is the process using an array to from a heap data structure from a binary tree. Min Heap and Max Heap is created using this method.

Algorithm for Heapify:

```
heapify(array)
    Root = array[0]

   Largest = largest( array[0] , array [2 * 0 + 1]/ array[2 * 0 + 2])
   if(Root != Largest)
    Swap(Root, Largest)
```

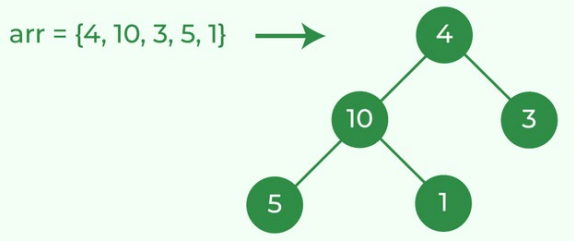
Working of Heap Sort

Step 1:

The array arr[] = {4, 10, 3, 5, 1} is used to create a binary tree for our example here

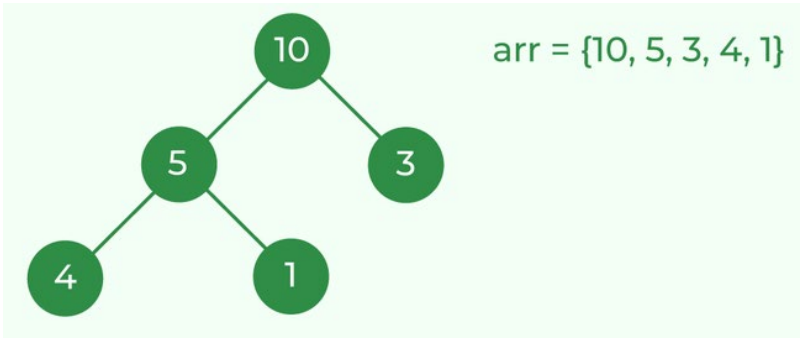
One of the condition for a heap to evolve into a mex heap is that the parent node should be larger than or equal to the child nodes

As shown in the example below, the parent node 4 is smaller than the child node 10, thus they are swapped to build a max-heap.



Step 2:

After that, as 4 as a parent node is smaller than the child 5, they are both swapped again and the resulted heap and array should be similar to the results shown below.

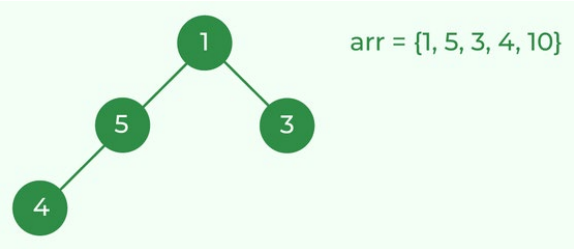


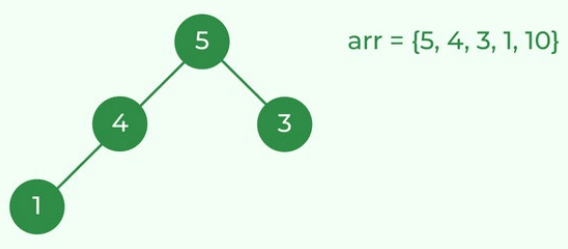
Step 3:

Heap sort is then performed by removing the maximum element is each step and then consider the remaining elements and transform it into a max heap

The root element 10 is deleted from the max heap. In order to delete this node, 10 is swapped with the last node which is 1. After removing the root element, heapify is used to convert it into max heap

The resulting heap and array are shown as below

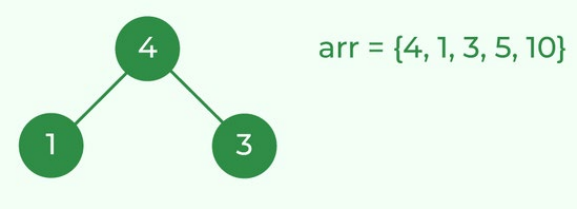




Max (10) is removed and heapify

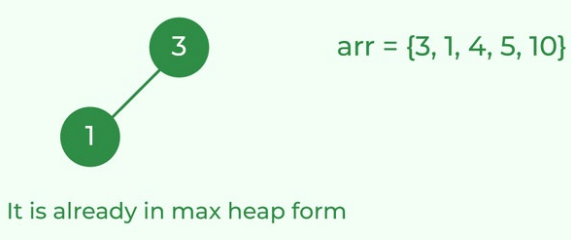
Step 4:

Step 3 is repeated and the results are as shown as below



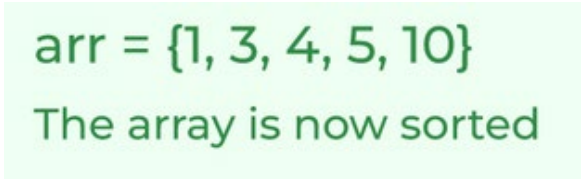
The current (5) is removed and heapify

Step 5



The current (4) is removed and heapify

Step 6



Max (3) is removed, the array is now sorted

Implementation of Heap Sort

```cpp
// C++ program for implementation of Heap Sort

#include <iostream>
using namespace std;

// To heapify a subtree rooted with node i
// which is an index in arr[].
// n is size of heap
void heapify(int arr[], int N, int i)
{

    // Initialize largest as root
    int largest = i;

    // left = 2*i + 1
    int l = 2 * i + 1;

    // right = 2*i + 2
    int r = 2 * i + 2;

    // If left child is larger than root
    if (l < N && arr[l] > arr[largest])
        largest = l;

    // If right child is larger than largest
    // so far
    if (r < N && arr[r] > arr[largest])
        largest = r;

    // If largest is not root
    if (largest != i) {
        swap(arr[i], arr[largest]);

        // Recursively heapify the affected
        // sub-tree
        heapify(arr, N, largest);
    }
}
```

```cpp
// Main function to do heap sort
void heapSort(int arr[], int N)
{

    // Build heap (rearrange array)
    for (int i = N / 2 - 1; i >= 0; i--)
        heapify(arr, N, i);

    // One by one extract an element
    // from heap
    for (int i = N - 1; i > 0; i--) {

        // Move current root to end
        swap(arr[0], arr[i]);

        // call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}

// A utility function to print array of size n
void printArray(int arr[], int N)
{
    for (int i = 0; i < N; ++i)
        cout << arr[i] << " ";
    cout << "\n";
}

// Driver's code
int main()
{
    int arr[] = { 12, 11, 13, 5, 6, 7 };
    int N = sizeof(arr) / sizeof(arr[0]);

    // Function call
    heapSort(arr, N);

    cout << "Sorted array is \n";
    printArray(arr, N);
}
```

```
Sorted array is
5 6 7 11 12 13

[Done] exited with code=0 in 1.803 seconds
```

Selection Sort

Function

- Sort data according to the condition user selected in ascending order

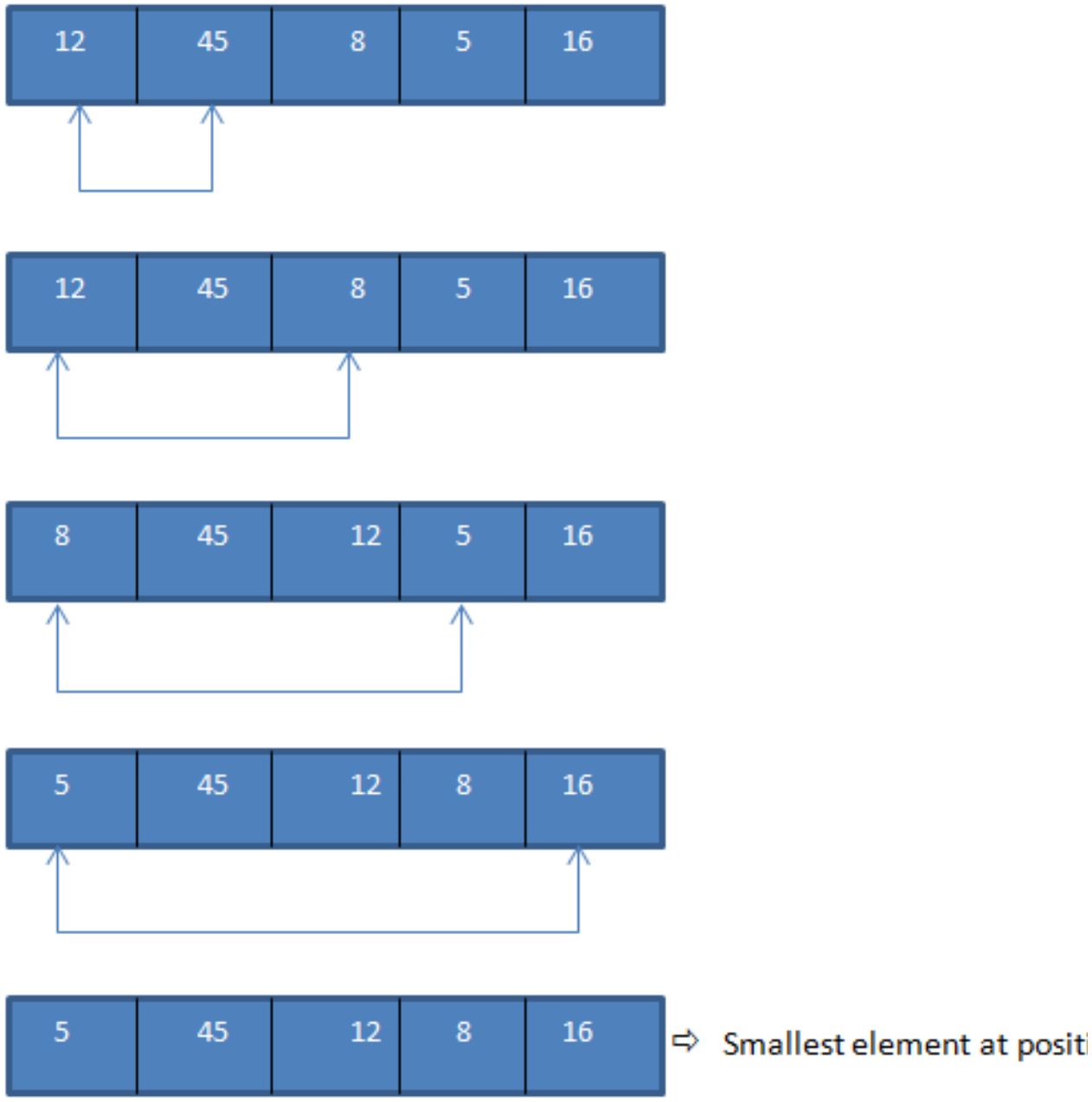
Advantages

- It is an in-place algorithm. It does not require a lot of space for sorting. Only one extra space is required for holding the temporal value
- Performs well on items that have already been sorted

Choose Reason

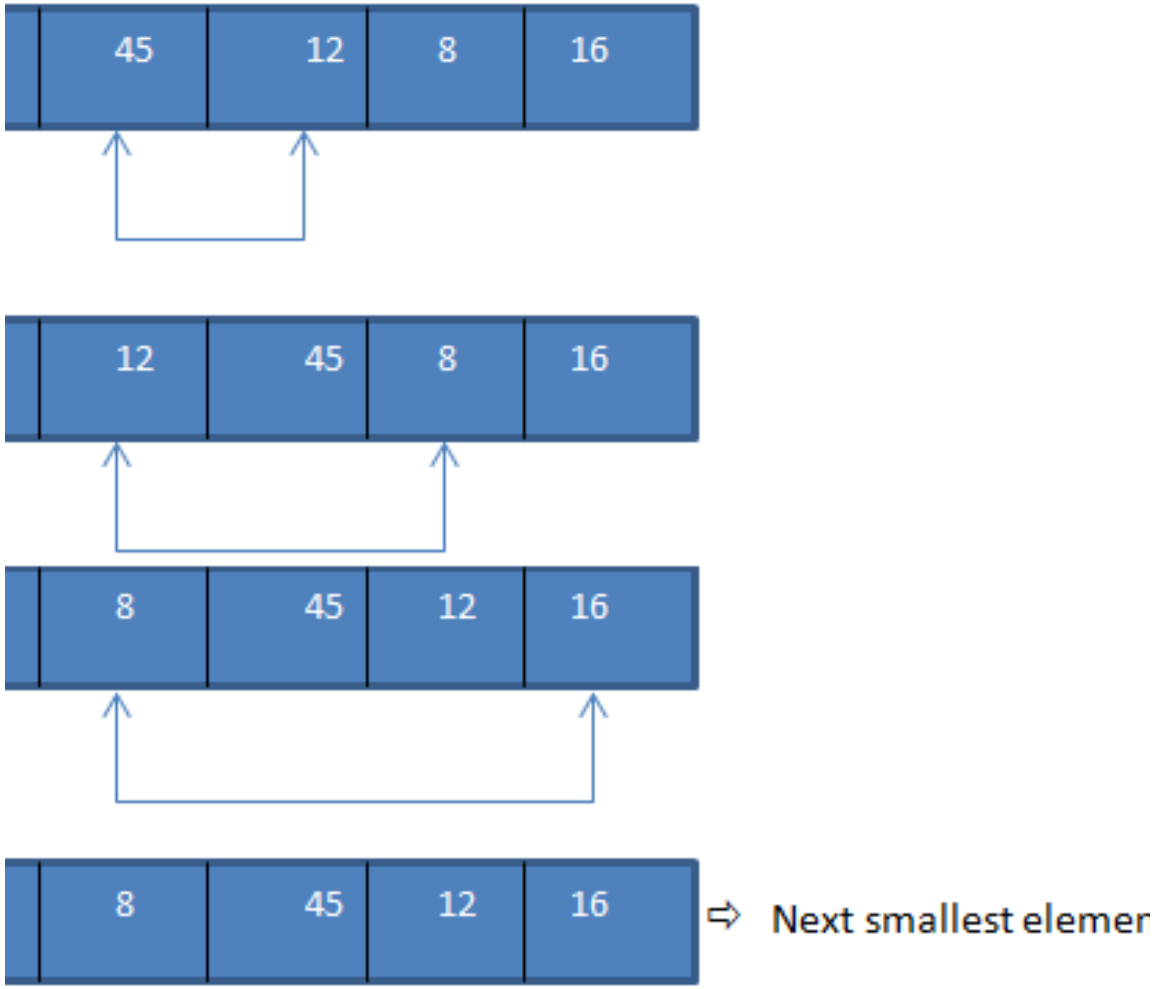
- We choose selection sort to sort the sale date. In every run of the program we only have a portion of data to be sorted so the effectiveness will not be affected as we are running in a small data.

Pass 1:



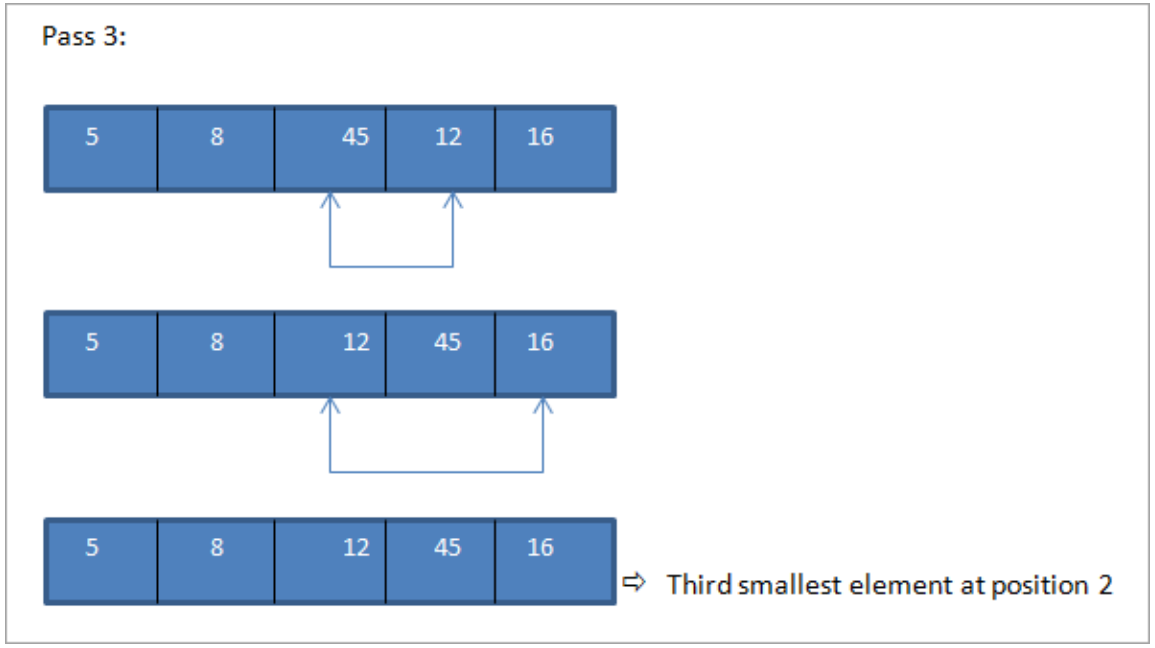
First Run

- Selection sort starts from one end and define it as the first item.
- Next it start searching for the lowest value in the data set to be switched
- Once the lowest value is selected, they switch places from the sorting zone to the sorted zone.



Second Run

- After the first element is confirmed, the algorithm look for the second lowest number from the dataset and switch their place with the second index of the data set.
- In this example, 8 is the second lowest value.

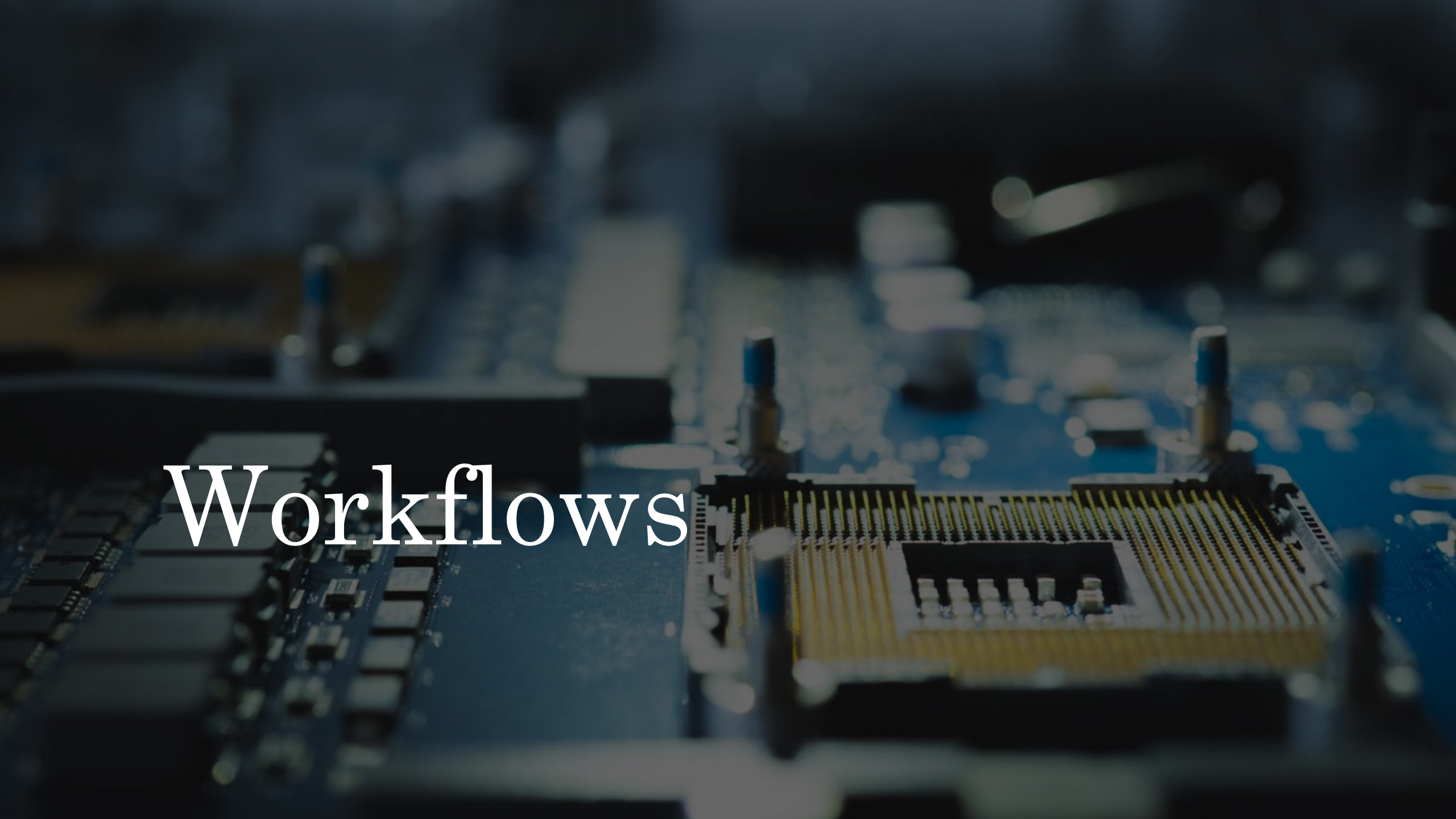


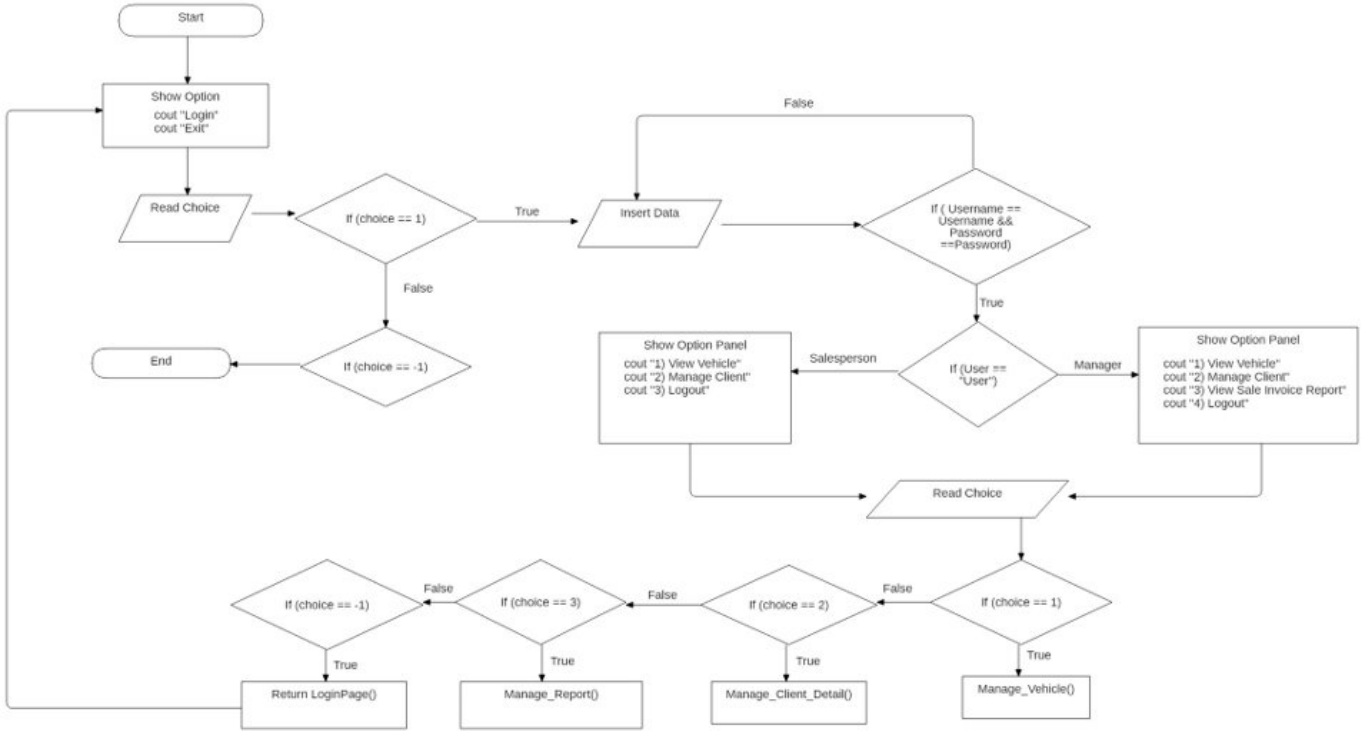
Third Run

- The selection sort keep looping until the dataset is fully sorted in ascending order.

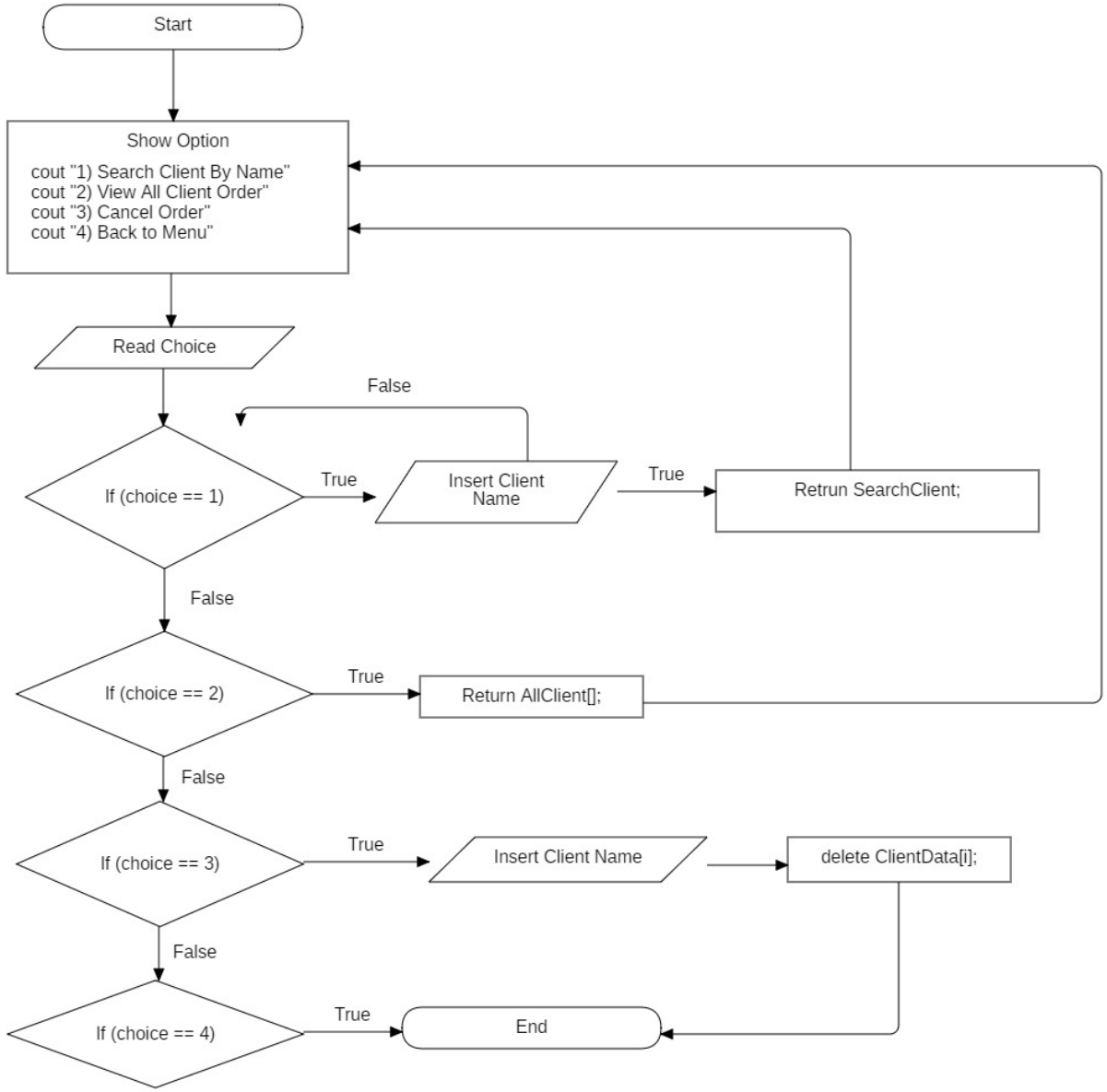
```
#include<iostream>
using namespace std;

int findSmallest (int[],int);
int main ()
{
   int myarray[5] = {12,45,8,15,33};
   int pos,temp;
   cout<<"\n Input list of elements to be Sorted\n";
   for(int i=0;i<5;i++)
   {
      cout<<myarray[i]<<"\t";
   }
   for(int i=0;i<5;i++)
   {
      pos = findSmallest (myarray,i);
      temp = myarray[i];
      myarray[i]=myarray[pos];
      myarray[pos] = temp;
   }
   cout<<"\n Sorted list of elements is\n";
   for(int i=0;i<5;i++)
   {
      cout<<myarray[i]<<"\t";
   }
return 0;
}
```

```
int findSmallest(int myarray[],int i)
{
   int ele_small,position,j;
   ele_small = myarray[i];
   position = i;
   for(j=i+1;j<5;j++)
   {
      if(myarray[j]<ele_small)
      {
      ele_small = myarray[j];
      position=j;
      }
   }
   return position;
}
```

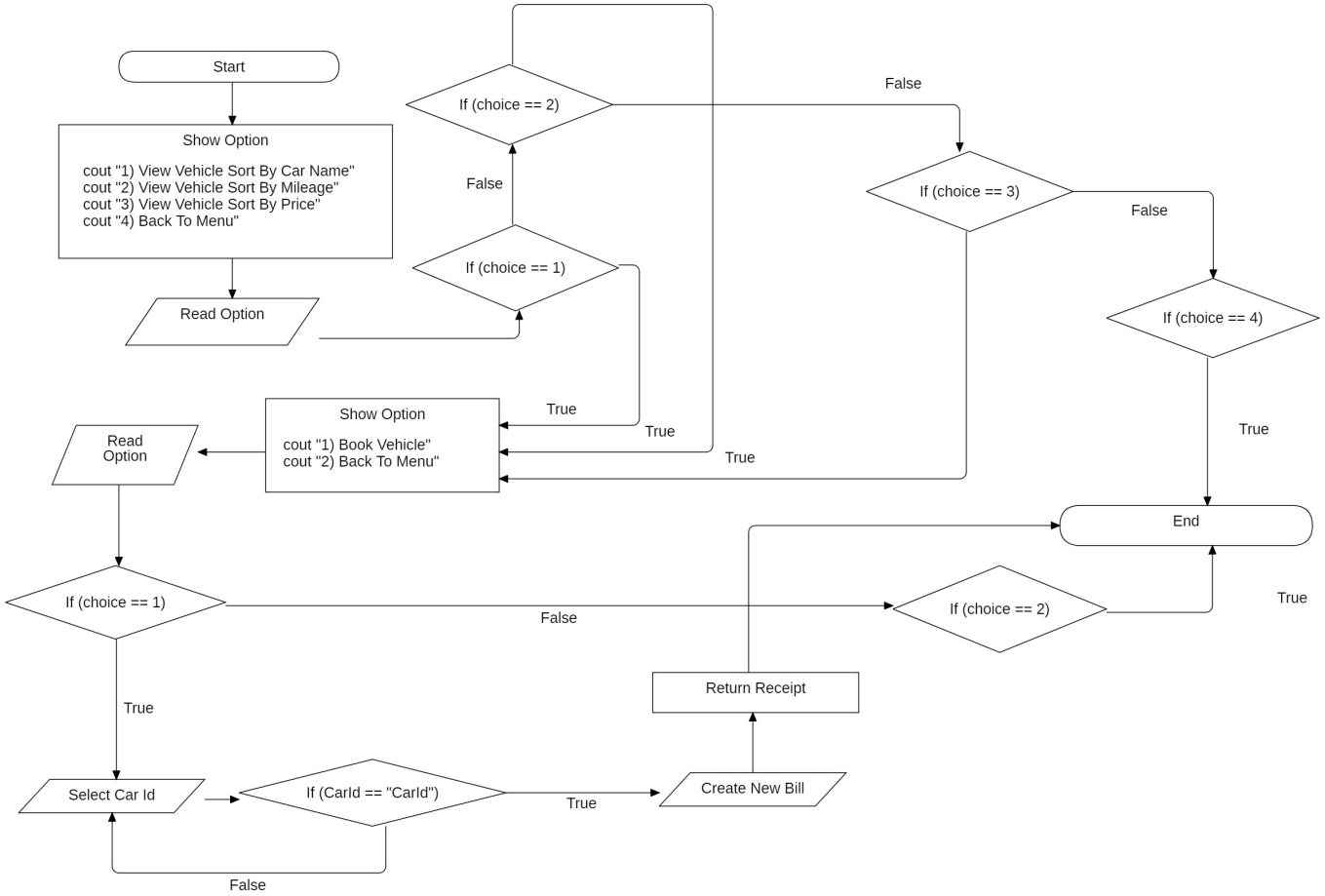

Workflows


Start
Show Option
cout "Login"
cout "Exit"
Read Choice
If (choice == 1)
True
Insert Data
False
If (Username == Username && Password ==Password)
True
False
End
If (choice == -1)
Show Option Panel
cout "1) View Vehicle"
cout "2) Manage Client"
cout "3) Logout"
Salesperson
If (User == "User")
Manager
Show Option Panel
cout "1) View Vehicle"
cout "2) Manage Client"
cout "3) View Sale Invoice Report"
cout "4) Logout"
Read Choice
If (choice == -1)
False
If (choice == 3)
False
If (choice == 2)
False
If (choice == 1)
True
True
True
True
Return LoginPage()
Manage_Report()
Manage_Client_Detail()
Manage_Vehicle()

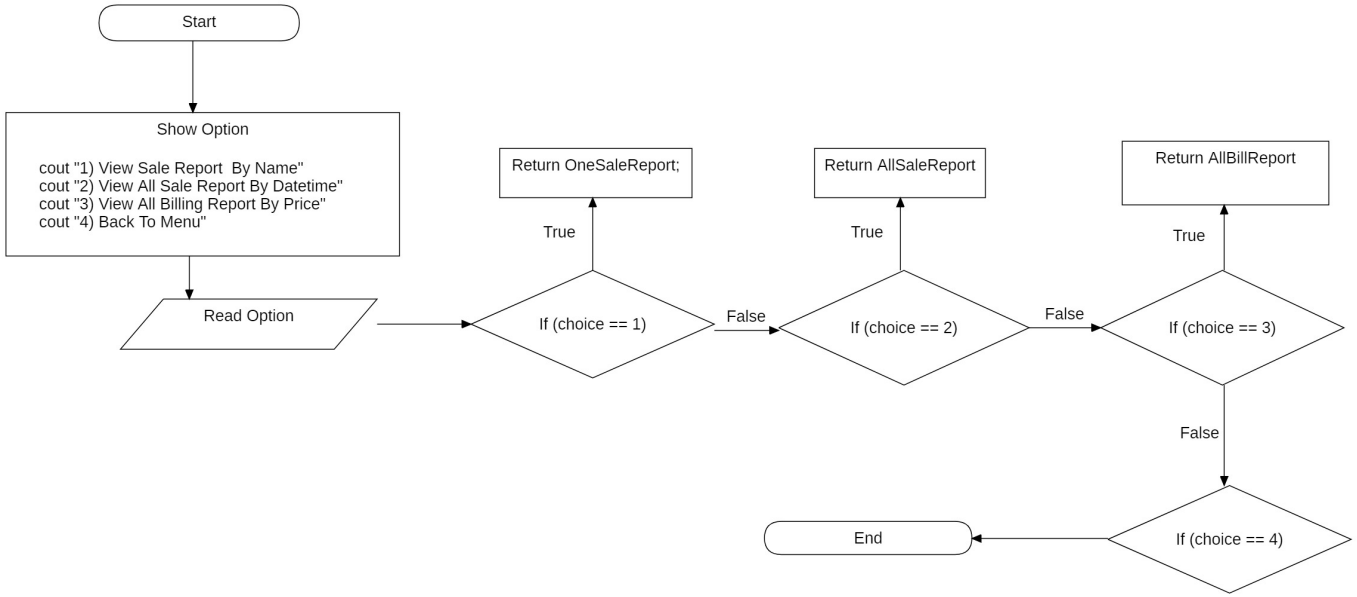
Menu Page



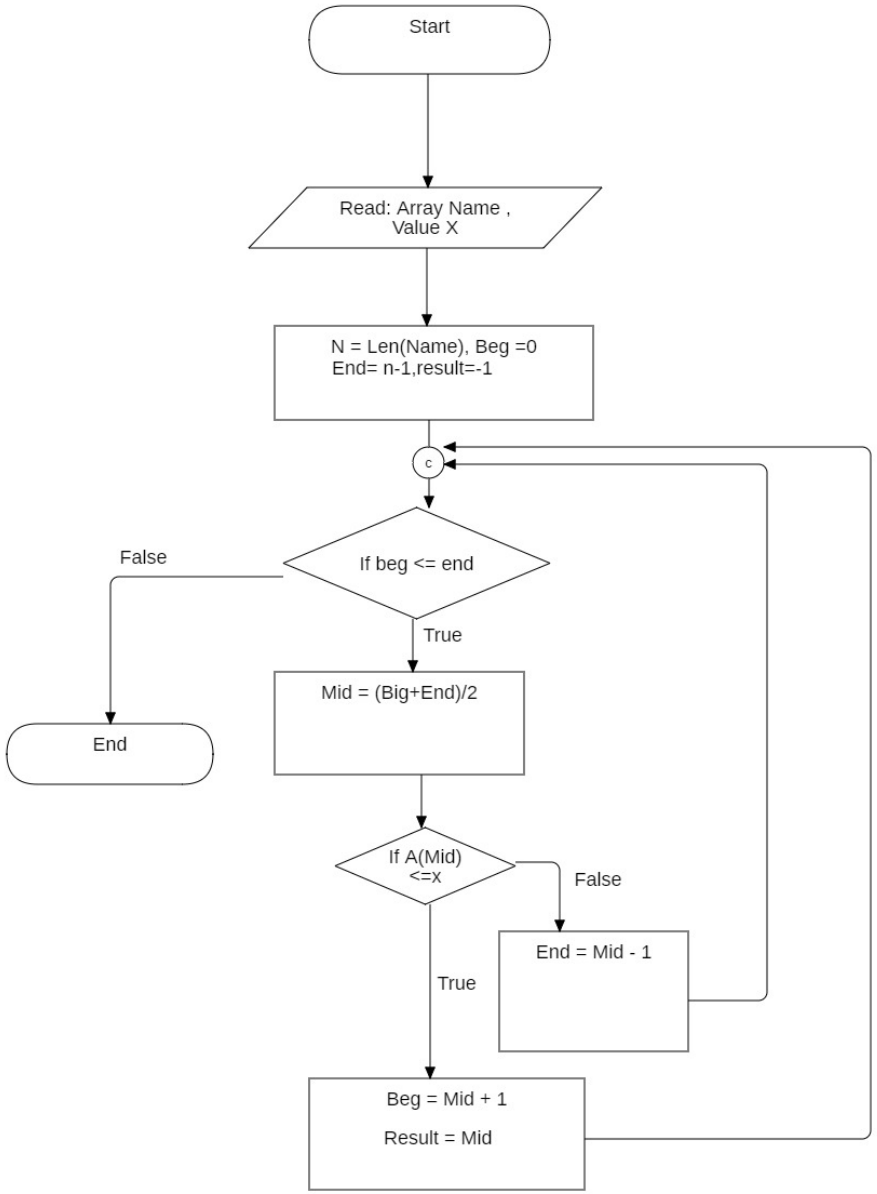
Manage Vehicle



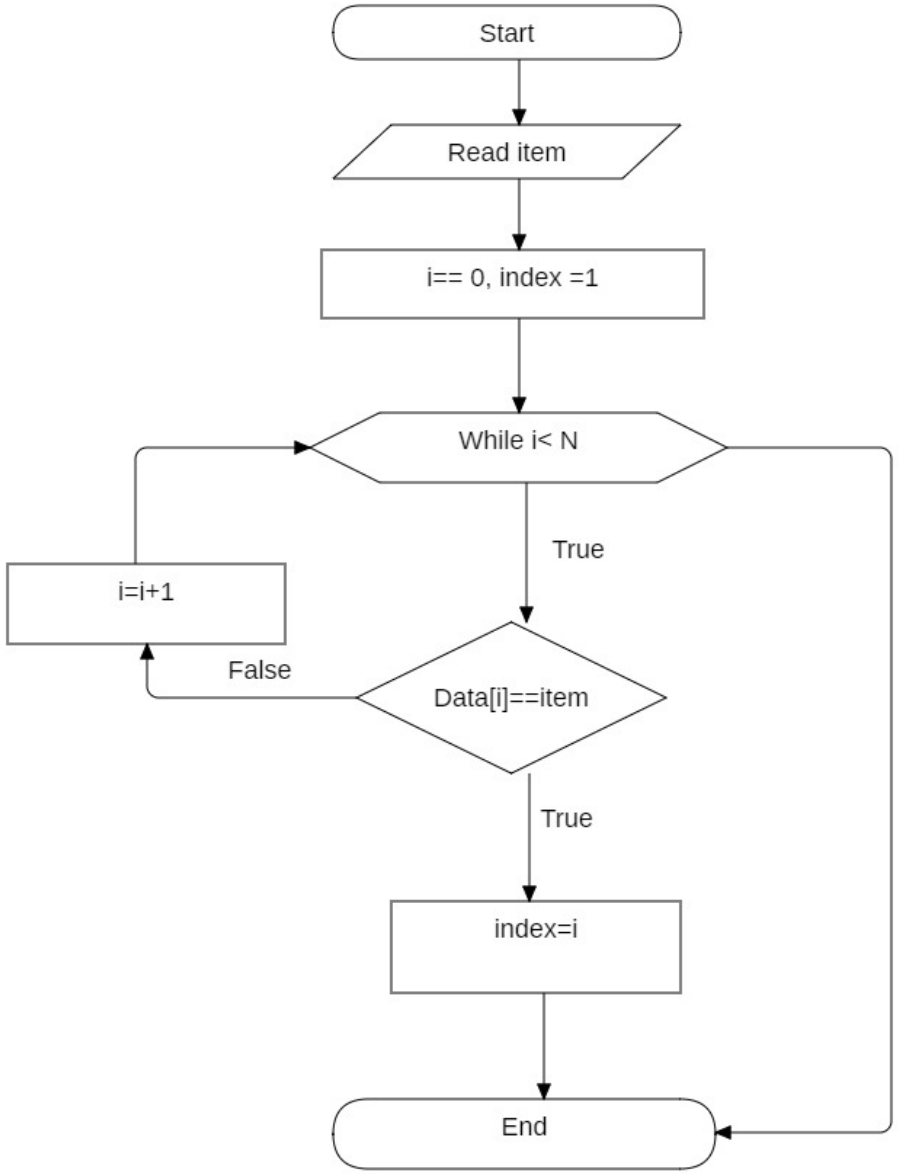
Manage Vehicle



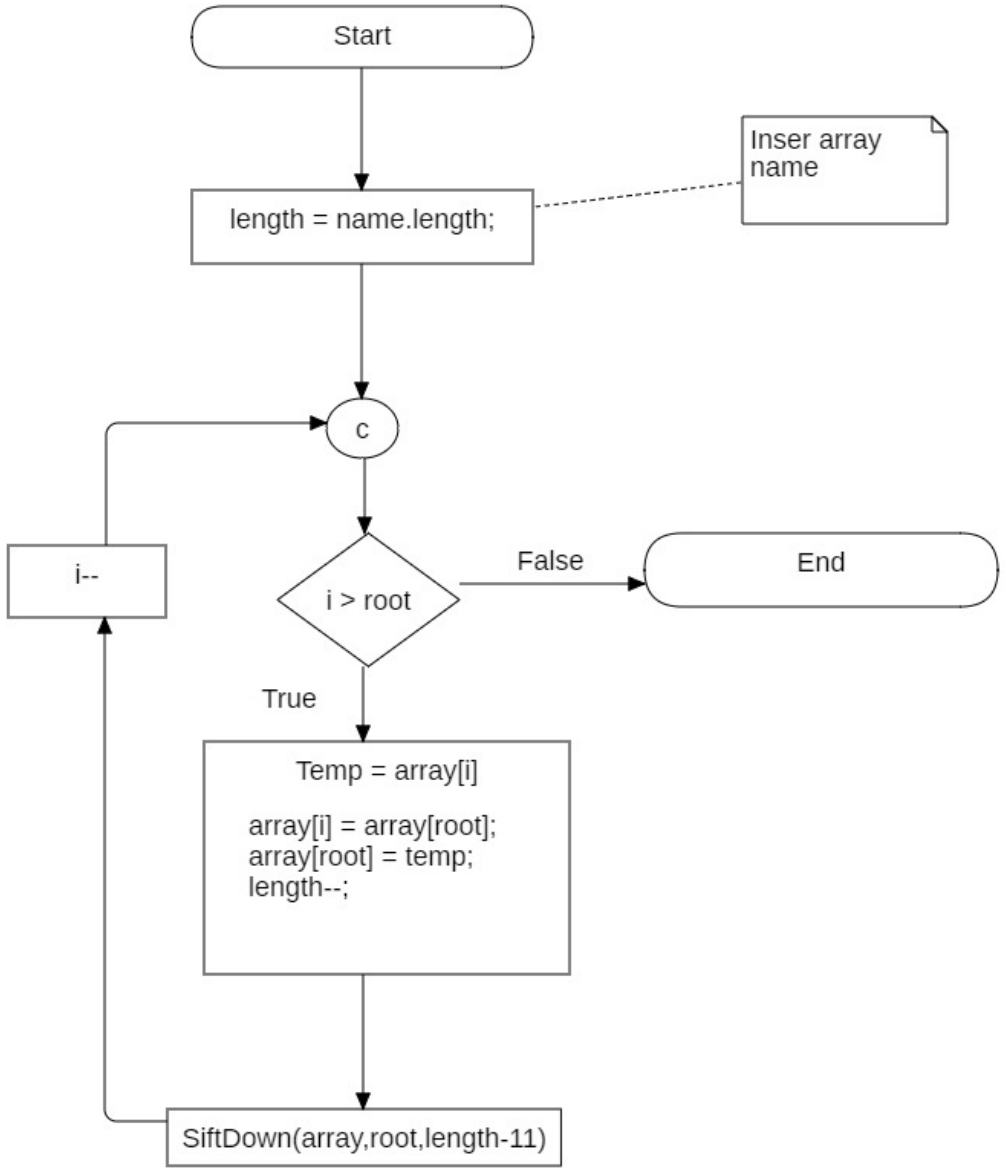
Manage Report



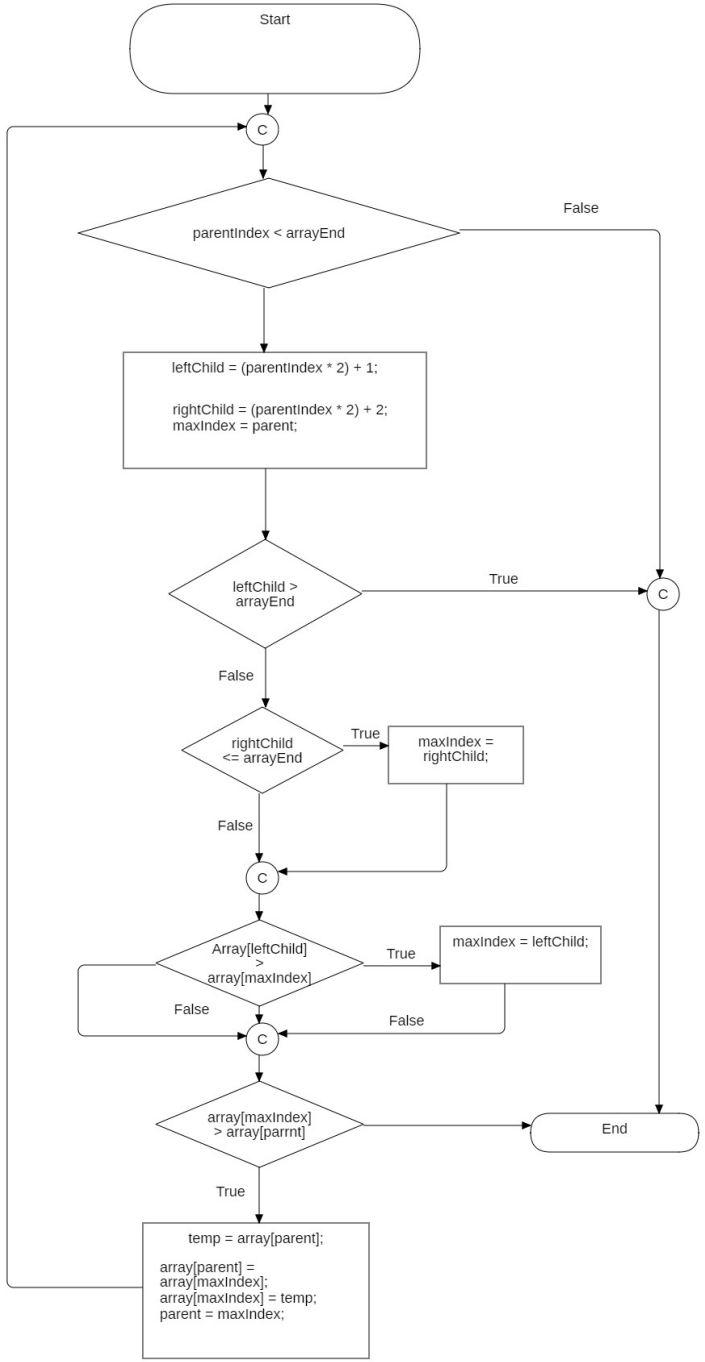
Binary Search-Search Client Name



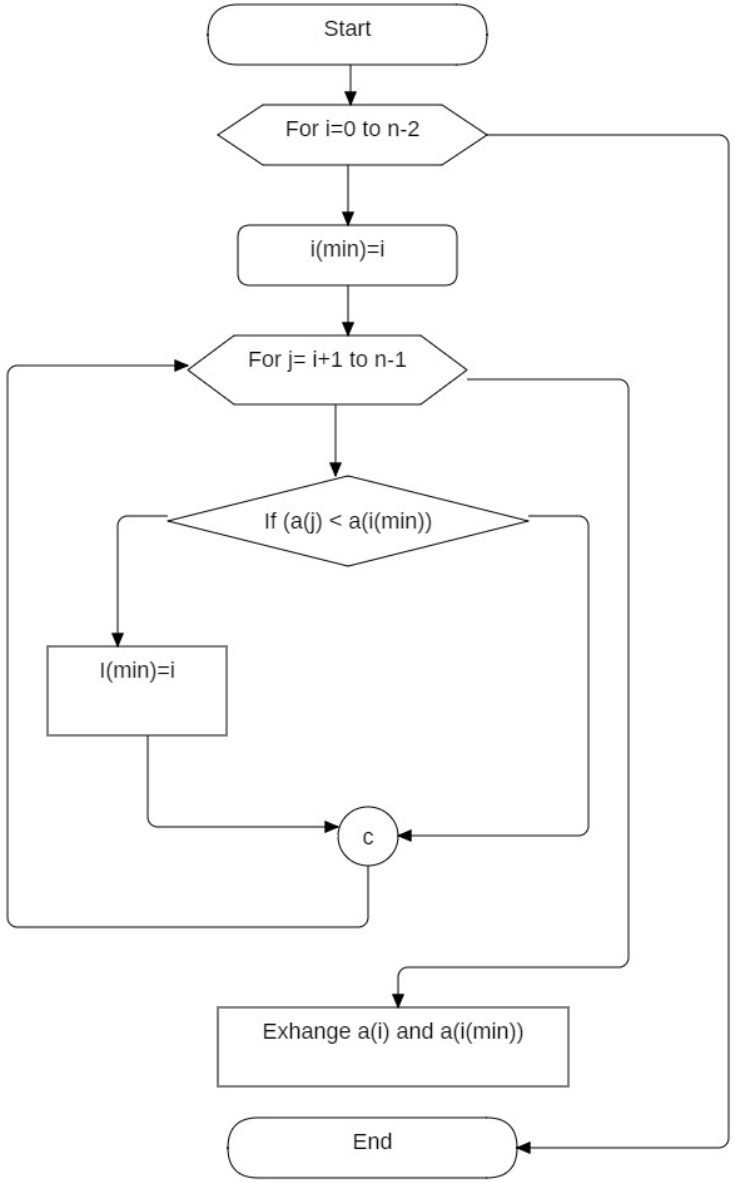
Linear Search- Search Car Id



Heap Sort-Sort Car Name



Shift Down Function for heap



Selection Sort - Sort Date Time