University of Burgundy
Software Engineering
Tutorial No 5

SHAH Bhargav
DUDHAGARA Akshay

24 November 2018

1 Header file .h

Listing 1: Declaration file

```
#ifndef LAB_5_H
#define LAB_5_H


class Node
{
public:
    int Data;
    Node *left;
    Node *right;
};


class CArray
{
public:

    // 1. Preliminary work

    int* Array;
    unsigned int Number_of_Elements;

    CArray();
    CArray(unsigned int);

    // Randomly initializes the array between 10-20 elements
    void Builder();
    // Randomly initializes the array with given size
    void Builder(unsigned int);

    // To Display the array
    void Display() const;

    // 2. A First and Simple Algorithm: Bubble Sort
    void Swap(unsigned int, unsigned int);
    void BubbleSort();

    // 3. Quicksort
    void Recursively_Sort(int*, unsigned int, unsigned int);
    int Recursive_Sort_Partition(int*, unsigned int, unsigned int);
    void QuickSort();
```

```
42
43      // 4. Selection Sort
44      void SelectionSort();
45
46      // 5. Insertion Sort
47      void InsertionSort();
48
49      // 6. Sort using binary trees
50      Node *CreateNode(int);
51      void Store_value(Node *, int *, int &);
52      Node *Insert_value(Node *, int );
53      void BinarySort();
54
55
56  };
57
58  #endif // LAB_5_H
```

2 Preliminary work

Listing 2: Definition

```
1   /////////// 1. Preliminary work ////////////////
2
3   // Build Default constructor
4   CArray::CArray()
5   {
6       this -> Builder();
7   }
8
9   // Build Parameterized constructor
10  CArray::CArray(unsigned int Number_of_Elements)
11  {
12      this -> Builder(Number_of_Elements);
13  }
14
15  // Initializes the array
16  void CArray::Builder()
17  {
18      int a;
19      cout << "Enter the Number of Element " << endl;
20      cin >> a;
21
22
23      this -> Number_of_Elements = a;
24
25      this -> Array = new int [this -> Number_of_Elements];
26
27      for(unsigned int i = 0; i < this -> Number_of_Elements; i++)
28      {
29          this -> Array[i] = rand() % 10; // Randomly assign values for array
30      }
31  }
32
33  // Initializes the size of the Array which is given by user
34  void CArray::Builder(unsigned int Number_of_Elements)
35  {
36      this -> Number_of_Elements = Number_of_Elements;
37
38      this -> Array = new int[Number_of_Elements];
39
40      for(unsigned int i = 0; i < this -> Number_of_Elements; i++)
41      {
42          this -> Array[i] = rand() % 10;
43      }
```

```
44 }
45
46 // Display the Array
47 void CArray::Display() const
48 {
49
50     cout << "Values Stored in the Array:" << endl;
51
52
53     for(unsigned int i = 0; i < this -> Number_of_Elements; i++)
54     {
55         cout << this -> Array[i] << " ";
56     }
57
58     cout << endl;
59
60     for(unsigned int i = 0; i < this->Number_of_Elements; i++)
61     {
62         cout << "Value at Index " << i << " is " <<  this -> Array[i] << endl;
63     }
64     cout << endl;
65 }
```

Listing 3: Main file

```
1
2 cout << "            1. Preliminary work: " << endl;
3 cout << endl;
4 // Display the values of the array
5 CArray Sorting_Array;
6 Sorting_Array.Display();
7 cout << endl;
```

3 Bubble Sort

Bubble sort has a worst-case and average complexity of $O(n^2)$, where n is the number of items being sorted.
1. Worst case: $O(n^2)$
2. Best case: $O(n)$
3. Average case: $O(n^2)$
4. Worst case space complexity: $O(1)$

Listing 4: definition

```
1 ////////////// 2. A First and Simple algorithm: Bubble Sort  ///////////////
2 // On each pass, bubble sort scans the array, comparing each pair of adjacent elements.
3 // If two adjacent elements are out of order, they are swapped.
4 // As long as at least one swap is performed along the scan, another pass is computed
5 // First Swaps the values of the Array with Index
6 void CArray::Swap(unsigned int Index1, unsigned int Index2)
7 {
8     int Bubble = 0;
9     Bubble = this -> Array [Index1];
10    this -> Array[Index1] = this -> Array[Index2];
11    this -> Array[Index2] = Bubble;
12 }
13
14 // Perform Bubble sort
15 void CArray::BubbleSort()
16 {
17     for(unsigned int i = 0; i < this -> Number_of_Elements; i++)
```

```
18      {
19          for(unsigned int j = i + 1; j < this -> Number_of_Elements; j++)
20          {
21              if(this -> Array[i] > this -> Array[j])
22                  this -> Swap(i, j);
23          }
24      }
25  }
```

Listing 5: main file

```
1                   // 2. Bubble Sort main file
2
3   cout << "            2. Bubble Sort:" << endl;
4   cout << endl;
5   cout << "Performing Bubblesort... " << endl;
6   cout << endl;
7   cout << "After Bubblesort " << endl;
8   Sorting_Array.BubbleSort();
9   Sorting_Array.Display();
10  cout << endl;
```

4 Quicksort

Quicksort can be used over Bubble Sort which has greater performance.
1. Worst case: $O(n^2)$
2. Best case: $O(nlongn)$
3. Average case: $O(nlongn)$
4. Worst case space complexity: $O(n)$

Listing 6: Definition

```
1   // Quicksort works in a "divide and conquer" manner
2   // split the initial list of numbers into parts around a "pivot";
3   // all the values in the first part are less than the pivot;
4   // all the values in the second part are greater than or equal to the pivot.
5   // Recursively sort the two parts
6
7   // left is the Index of the Left Element of the subarray
8   // right is the Index of the Right Element of the subarray
9   // Number of Elements in subarray = right-left+1
10
11  void CArray::Recursively_Sort(int* Element, unsigned int left, unsigned int right)
12  {
13      if(left < right) //  If array has two or more elements
14      {
15          int Pivot_X = this->Recursive_Sort_Partition(Element, left, right);
16
17          if(Pivot_X != 0)
18              this->Recursively_Sort(Element, left, Pivot_X - 1); // Elements smaller than
                    the pivot
19
20          this->Recursively_Sort(Element, Pivot_X + 1, right); // Elements bigger than the
                pivot
21      }
22
23  }
24
25  // The Final step is to move the pivot between the two regions by swapping
26  int CArray::Recursive_Sort_Partition(int* Element, unsigned int left, unsigned int right)
27  {
28      int Pivot = Element[right];
```

```
29
30      unsigned int Index = left;
31
32      for(unsigned int i = left; i < right; i++)
33      {
34          if(Element[i] <= Pivot)
35          {
36              this -> Swap(i, Index); // If swapped , Increment
37              Index++;
38          }
39      }
40
41      this -> Swap(Index, right); // Move Pivot to end
42
43      return Index;
44  }
45
46  void CArray::QuickSort()
47  {
48      this -> Recursively_Sort(this -> Array, 0, this -> Number_of_Elements - 1);
49
50  }
```

Listing 7: Main file

```
1  cout << "            3. Quick Sort:" << endl;
2  cout << endl;
3  cout << "Performing Quicksort... " << endl;
4  cout << endl;
5  cout << "After quick sort " << endl;
6  Sorting_Array.QuickSort();
7  Sorting_Array.Display();
8  cout << endl;
```

5 Selection Sort

selection sort is a sorting algorithm, specifically an in-place comparison sort. It has $O(n^2)$ time complexity, making it inefficient on large lists, and generally performs worse than the similar insertion sort.
1. Worst case: $O(n^2)$
2. Best case: $O(n^2)$
3. Average case: $O(n^2)$
4. Worst case space complexity: $O(n), O(1)$ auxiliary

Listing 8: Definition

```
1  ////////////////  4. Selection Sort ////////////////////
2  // In selection sort the array is divided into two parts
3  // The first part that is sorted and the second part that is not sorted
4  // Initially the sorted part is empty and the unsorted part consists of the whole array
5  // In each step, the algorithm searches through the unsorted part,
6  // Finds the smallest element and puts it at the end of the sorted part
7
8  void CArray::SelectionSort()
9  {
10     unsigned int Sorted_Part = 0;
11
12     for(unsigned int i = 0; i < this -> Number_of_Elements; i++)
13     {
14         Sorted_Part = i;
15
16         for(unsigned int j = i + 1; j < this -> Number_of_Elements; j++)
```

```
17              if(this -> Array[j] < this -> Array[Sorted_Part]) // Finds the smallest
                    element
18                  Sorted_Part = j;
19
20          this -> Swap(i, Sorted_Part);
21      }
22  }
```

Listing 9: Main File

```
1  cout << "            4. Selection Sort:" << endl;
2  cout << endl;
3  cout << "Performing Selection Sort... " << endl;
4  cout << endl;
5  cout << "After Selection Sort " << endl;
6  Sorting_Array.SelectionSort();
7  Sorting_Array.Display();
8  cout << endl;
```

6 Insertion Sort

Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort.It is more efficient in practice than most other simple quadratic algorithms such as selection sort or bubble sort.

1. Worst case: $O(n^2)$ comparisons, swaps
2. Best case: $O(n)$ comparisons,$O(1)$ swaps
3. Average case: $O(n^2)$ comparisons, swaps
4. Worst case space complexity: $O(n), O(1)$ auxiliary

Listing 10: Definition

```
1  ////////////////// 5. Insertion Sort ///////////////////////
2  // The initialization of the algorithm is similar to the selection sort
3  // Dividing the array into a sorted and an unsorted part
4  // Each step of the algorithm picks the first item of the unsorted array and
5  // Inserts it into the right slot of the sorted array
6
7  void CArray::InsertionSort()
8  {
9      int Insert = 0;
10     unsigned int Sorted_null = 0;
11
12     for(unsigned int i = 1; i < this -> Number_of_Elements; i++)
13     {
14         Insert = this -> Array[i]; // Value will be inserted into the array
15         Sorted_null = i; // position i as the null Index
16
17         while(Sorted_null > 0 && Insert < this -> Array[Sorted_null -1])
18         {
19             // Shift the larger value up
20             this -> Array[Sorted_null] = this -> Array[Sorted_null - 1];
21             Sorted_null--;
22         }
23         // Inserts it into the right slot of the sorted array
24         this -> Array[Sorted_null] = Insert;
25     }
26 }
```

Listing 11: Main File

```
cout << "            5. Insertion Sort:" << endl;
cout << endl;
cout << "Performing Insertion Sort... " << endl;
cout << endl;
cout<< "After Insertion Sort " <<endl;
Sorting_Array.InsertionSort();
Sorting_Array.Display();
cout << endl;
```

7 Sort using binary tree

Listing 12: Definition

```
// Create a new node
Node *CArray::CreateNode(int done)
{
    Node *create = new Node;
    create -> Data = done;
    create -> left = create -> right = nullptr;
    return create;
}

// Store sorted elements in an array
void CArray::Store_value(Node *root, int *store, int &m)
{
    if (root != nullptr)
    {
        Store_value(root -> left, store, m);
        store[m++] = root -> Data;
        Store_value(root -> right, store, m);
    }
}

// Insert values to the new node
Node *CArray::Insert_value(Node *node, int data)
{
    // If the node is empty, return a new Node
    if (node == nullptr)
        return CreateNode(data);

    // Down the tree
    if (data < node -> Data)
        node -> left  = Insert_value(node -> left, data);

    else if (data > node -> Data)
        node -> right = Insert_value(node -> right, data);

    return node;
}

/*
// The Pre-Order traversal: at each node the root is evaluated first
// then the left sub tree, the the right subtree.
void CArray::PreOrder(Node *node, int data)
{
    if(node -> Data != 0)
        cout<<"array"<<node->data<<endl;

    if(node -> left != 0)
        node -> left = PreOrder(node -> left, data);

    if(node->right != 0)
        node -> right = PreOrder(node -> right, data);
*/
```

```
52
53 // Binary Sort Algorithm
54 void CArray::BinarySort()
55 {
56     Node *root = nullptr;
57
58     root = Insert_value(root, this -> Array[0]);
59
60     for (unsigned int i=0; i < this -> Number_of_Elements; i++)
61         Insert_value(root, this -> Array[0]);
62
63     // Store inoder traversal of the BST
64     int n = 0;
65     Store_value(root, this -> Array, n);
66 }
```

Listing 13: Main File

```
1 cout << "           6. Sort using Binary trees:" << endl;
2 cout << endl;
3 cout << "Performing Binary Sort... " << endl;
4 cout << endl;
5 cout<< "After Binary Sort " <<endl;
6 Sorting_Array.BinarySort();
7 Sorting_Array.Display();
8 cout << endl;
```