

Practical: 1**25-07-2024****Write a program that implement array operations a) Insertion b) Deletion****1a) Inserting the element into the array at any specific position.****Program:**

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

int main()
{
    int arreySize_input;

    cout << "Enter the size of array: " << endl;
    cin >> arreySize_input;

    int arraySize = arreySize_input;
    int array[arraySize + 1];
    int i;

    cout << "Enter the array elements: " << endl;
    for (i = 0; i < arraySize; i++)
    {
        cin >> array[i];
    }

    int position, element;
    cout << "Enter the position to insert the new element (0 to " << arraySize << "): ";
    cin >> position;
    cout << "Enter the element to insert: ";
    cin >> element;

    for (i = arraySize; i > position; i--)
    {
        array[i] = array[i - 1];
    }

    array[position] = element;
    arraySize++;

    cout << "The array elements after insertion are: ";
    for (i = 0; i < arraySize; i++)
    {
        cout << array[i] << " ";
    }
    cout << endl;
```

```
    return 0;  
}
```

Output:

```
PS D:\MU\MU-DS\BASIC PROGRAM> cd "d:\MU\MU-DS\BASIC PROGRAM\" ; if ($?) { g++ insert_array.cpp -o insert_array } ; if ($?)  
{ .\insert_array }  
Enter the size of array:  
5  
Enter the array elements:  
1  
2  
3  
4  
5  
Enter the position to insert the new element (0 to 5): 3  
Enter the element to insert: 0  
The array elements after insertion are: 1 2 3 0 4 5  
PS D:\MU\MU-DS\BASIC PROGRAM> |
```



1b) Deleting the element from the array.

Program:

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

int main()
{
    int arreySize_input;

    cout << "Enter the size of array: " << endl;
    cin >> arreySize_input;

    int arraySize = arreySize_input;
    int array[arraySize];
    int i;

    cout << "Enter the array elements: " << endl;
    for (i = 0; i < arraySize; i++)
    {
        // Read elements into the array
        cin >> array[i];
    }

    int position;
    cout << "Enter the position of the element to delete (0 to " << arraySize - 1 << "): ";
    cin >> position;

    if (position < 0 || position >= arraySize)
    {
        cout << "Invalid position!" << endl;
    }
    else
    {
        // Shift left
        for (i = position; i < arraySize - 1; i++)
        {
            array[i] = array[i + 1];
        }
        arraySize--;

        // Update array
        cout << "The array elements after deletion are: ";
        for (i = 0; i < arraySize; i++)
        {
            cout << array[i] << " ";
        }
    }
}
```

```
    }  
    cout << endl;  
}  
  
return 0;  
}
```

Output:

```
PS D:\MU\MU-DS\BASIC PROGRAM> cd "d:\MU\MU-DS\BASIC PROGRAM\" ; if ($?) { g++ delete_array.cpp -o delete_array } ; if ($?) { .\del  
ete_array }  
Enter the size of array:  
4  
Enter the array elements:  
1  
2  
4  
3  
Enter the position of the element to delete (0 to 3): 2  
The array elements after deletion are: 1 2 3  
PS D:\MU\MU-DS\BASIC PROGRAM> █
```

Conclusion:

Deleting an element from an array involves shifting subsequent elements to fill the gap, maintaining contiguity, and has a time complexity of $O(n)$. Inserting an element at a specific position requires shifting elements to the right to make room, also with a time complexity of $O(n)$, and may require resizing the array if it is full.



Practical: 2

Write a program that implements the following sorting

a) Bubble sort b) Insertion sort c) Selection sort

01-08-2024

2 a) Program to implement Bubble sort.

Program:

```
#include <iostream>
using namespace std;
int main()
{
    int i, arr[50], n, x, y;

    cout << "Enter the size of array:";
    cin >> n;

    cout << "Enter the elements in an array:";
    for (i = 0; i < n; i++)
    {
        cin >> arr[i];
    }

    for (i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            if (arr[j] > arr[j + 1])
            {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }

    for (i = 0; i < n; i++)
    {
        cout << arr[i] << "\t";
    }
}
```



Output:

```
Enter the size of array:4
Enter the elements in an array:4
7
1
0
0      1      4      7
PS D:\MU\MU-DS\BASIC PROGRAM\Practical 2>
```



2 b) Program to implement Insertion sort.

Program:

```
#include <iostream>
using namespace std;
int main()
{
    int i, arr[50], n, j, current;

    cout << "Enter the size of array:";
    cin >> n;

    cout << "Enter the elements in an array:";
    for (i = 0; i < n; i++)
    {
        cin >> arr[i];
    }

    for (i = 1; i < n; i++)
    {
        current = arr[i];
        j = i - 1;

        while (arr[j] > current && j >= 0)
        {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = current;
    }
    for (i = 0; i < n; i++)
    {
        cout << arr[i] << "\t";
    }
}
```

Output:

```
Enter the size of array:5
Enter the elements in an array:1
7
8
3
0
0      1      3      7      8
PS D:\MU\MU-DS\BASIC PROGRAM\Practical 2>
```



2 c) Program to implement Selection sort.

Program:

```
#include <iostream>
using namespace std;
int main()
{
    int i, arr[50], n, x, y;

    cout << "Enter the size of array:";
    cin >> n;

    cout << "Enter the elements in an array:";
    for (i = 0; i < n; i++)
    {
        cin >> arr[i];
    }

    for (i = 0; i < n - 1; i++)
    {
        for (int j = i + 1; j < n; j++)
        {
            if (arr[i] > arr[j])
            {
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
    }

    for (i = 0; i < n; i++)
    {
        cout << arr[i] << "\t";
    }
}
```




Output:

```
Enter the size of array:6
Enter the elements in an array:9
2
0
7
1
4
0      1      2      4      7      9
PS D:\MU\MU-DS\BASIC PROGRAM\Practical 2> |
```

Conclusion:

In this experiment we have learned how to Sort elements in an Array using Bubble Sorting, Insertion Sorting and Selection sorting.



Practical: 3

Write a program that implements the following

a) Quick Sort b) Merge sort

3 a) Program to implement Quick sort.

Program:

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

// Function to print an array
void printArr(int a[], int n)
{
    for (int i = 0; i < n; i++)
        cout << a[i] << " ";
    cout << endl;
}

int partition(int a[], int start, int end)
{
    int pivot = a[end]; // pivot element
    int i = start - 1;

    for (int j = start; j <= end - 1; j++)
    {
        if (a[j] < pivot)
        {
            i++;
            swap(a[i], a[j]);
        }
    }
    swap(a[i + 1], a[end]);
    return i + 1;
}

void quick(int a[], int start, int end)
{
    if (start < end)
    {
        int p = partition(a, start, end);

        quick(a, start, p - 1);
        quick(a, p + 1, end);
    }
}

int main()
```



```
{  
  
int arraySize;  
  
    cout << "Enter the size of the array: ";  
    cin >> arraySize;  
  
    int a[arraySize];  
    cout << "Enter the array elements: " << endl;  
    for (int i = 0; i < arraySize; i++)  
    {  
        cin >> a[i];  
    }  
  
    cout << "The array elements are: ";  
    for (int i = 0; i < arraySize; i++)  
    {  
        cout << a[i] << " ";  
    }  
    cout << endl;  
  
    int n = sizeof(a) / sizeof(a[0]);  
  
    cout << "Before sorting, array elements are:" << endl;  
    printArr(a, n);  
  
    quick(a, 0, n - 1);  
  
    cout << "After sorting, array elements are:" << endl;  
    printArr(a, n);  
  
    return 0;  
}
```

Output:

```
PS D:\MU\MU-DS\Practical 3> cd "d:\MU\MU-DS\Practical 3\" ; if ($?) { g++ quickShort.cpp -o quickShort } ; if ($?) { .\quickShort  
}  
Enter the size of the array: 5  
Enter the array elements:  
33  
99  
88  
44  
11  
The array elements are: 33 99 88 44 11  
Before sorting, array elements are:  
33 99 88 44 11  
After sorting, array elements are:  
11 33 44 88 99  
PS D:\MU\MU-DS\Practical 3> |
```



3 b) Program to implement Merge sort.

Program:

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

void merge(int arr[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;

    int *L = new int[n1];
    int *R = new int[n2];

    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    i = 0;
    j = 0;
    k = l;
    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2)
    {

```



```
        arr[k] = R[j];
        j++;
        k++;
    }

    delete[] L;
    delete[] R;
}

void mergeSort(int arr[], int l, int r)
{
    if (l < r)
    {
        int m = l + (r - l) / 2;

        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}

void printArray(int A[], int size)
{
    for (int i = 0; i < size; i++)
        cout << A[i] << " ";
    cout << endl;
}

/* Driver code */
int main()
{
    int arraySize;

    cout << "Enter the size of the array: ";
    cin >> arraySize;

    int arr[arraySize];
    cout << "Enter the array elements: " << endl;
    for (int i = 0; i < arraySize; i++)
    {
        cin >> arr[i];
    }

    cout << "The array elements are: ";
    for (int i = 0; i < arraySize; i++)
    {
```

```
        cout << arr[i] << " ";
    }
    cout << endl;

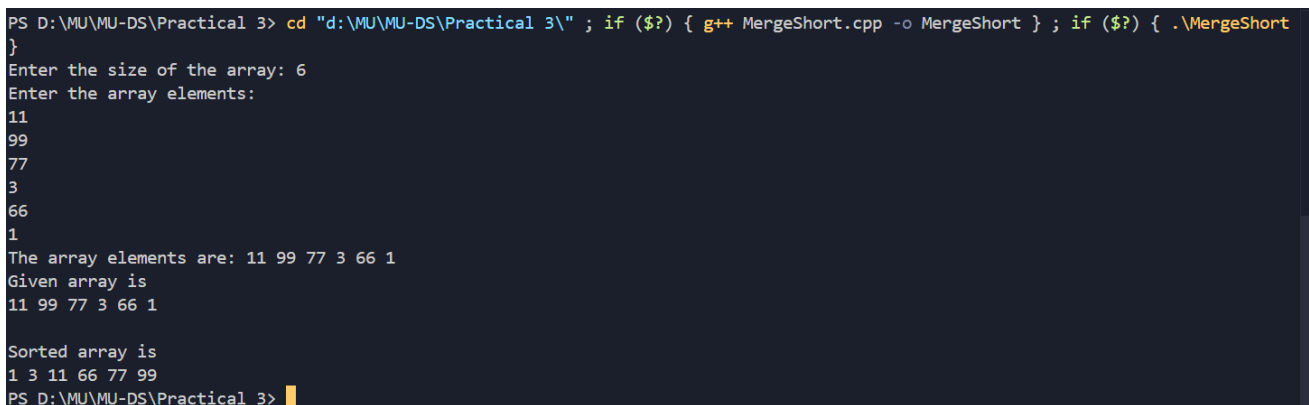
    int arr_size = sizeof(arr) / sizeof(arr[0]);

    cout << "Given array is \n";
    printArray(arr, arr_size);

    mergeSort(arr, 0, arr_size - 1);

    cout << "\nSorted array is \n";
    printArray(arr, arr_size);
    return 0;
}
```

Output:



```
PS D:\MU\MU-DS\Practical 3> cd "d:\MU\MU-DS\Practical 3\" ; if ($?) { g++ MergeShort.cpp -o MergeShort } ; if ($?) { .\MergeShort
}
Enter the size of the array: 6
Enter the array elements:
11
99
77
3
66
1
The array elements are: 11 99 77 3 66 1
Given array is
11 99 77 3 66 1

Sorted array is
1 3 11 66 77 99
PS D:\MU\MU-DS\Practical 3>
```

Conclusion:

Quick Sort and Merge Sort are both efficient sorting algorithms with distinct characteristics. Quick Sort uses a pivot to partition the array and sorts in place, offering an average time complexity of $O(n \log n)$ but with a worst-case of $O(n^2)$. It's generally faster and more memory-efficient. Merge Sort, on the other hand, consistently performs at $O(n \log n)$, as it splits the array, sorts the halves, and merges them. Though stable and predictable, Merge Sort requires additional memory. Quick Sort is preferred for its speed, while Merge Sort is favored when stability and worst-case guarantees are needed.

Practical: 4**Write a program for searching an element from the given list****a) Linear search b) Binary search.****4 a) Program to implement Linear sort.****Program:**

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

int search(int arr[], int N, int x)
{
    for (int i = 0; i < N; i++)
        if (arr[i] == x)
            return i;
    return -1;
}

// Driver's code
int main()
{
    int arraySize;

    cout << "Enter the size of the array: ";
    cin >> arraySize;

    int array[arraySize];
    cout << "Enter the array elements: " << endl;
    for (int i = 0; i < arraySize; i++)
    {
        cin >> array[i];
    }

    cout << "The array elements are: ";
    for (int i = 0; i < arraySize; i++)
    {
        cout << array[i] << " ";
    }
    cout << endl;

    int x;
    cout << "Enter the value to search: ";
    cin >> x;

    //TODO: Function call
    int result = search(array, arraySize, x);
```



```
if (result == -1)
    cout << "Element is not present in the array" << endl;
else
    cout << "Element is present at index " << result << endl;

return 0;
}
```

Output:

```
PS D:\MU\MU-DS\Practical 4> cd "d:\MU\MU-DS\Practical 4\" ; if ($?) { g++ LinearSearch.cpp -o LinearSearch } ; if ($?) { .\
LinearSearch }
Enter the size of the array: 5
Enter the array elements:
33
11
1
7
99
The array elements are: 33 11 1 7 99
Enter the value to search: 1
Element is present at index 2
PS D:\MU\MU-DS\Practical 4> |
```




4 b) Program to implement Binary sort.

Program:

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

int sorted(int a[], int n)
{
    int i, j;
    for (i = 0; i < n; i++)
    {
        for (int j = 0; j < n - 1; j++)
        {
            if (a[j] > a[j + 1])
            {
                int temp = a[j];
                a[j] = a[j + 1];
                a[j + 1] = temp;
            }
        }
    }
    return a[50];
}

int main()
{
    int i, arr[50], n, x, y;

    cout << "Enter the size of array:";
    cin >> n;

    cout << "Enter the elements in an array:";
    for (i = 0; i < n; i++)
    {
        cin >> arr[i];
    }

    sorted(arr, n);
    for (i = 0; i < n; i++)
    {
        cout << arr[i] << "\t";
    }

    int end = n - 1, beg = 0;
    cout << "\n\nEnter the elements to be found:";
    cin >> x;

    for (i = 0; i < n; i++)
    {
```



```
int mid = (beg + end) / 2;

if (arr[mid] == x)
{
    cout << "The element is found at index:" << mid;
    break;
}
else if (arr[mid] > x)
{
    end = mid - 1;
}
else
{
    beg = mid + 1;
}
}
```

Output:

```
Enter the size of array:5
Enter the elements in an array:9
4
1
6
3
1      3      4      6      9
Enter the elements to be found:4
The element is found at index:2
PS D:\MU\MU-DS\Practical 4> |
```

Conclusion:

Linear search is a simple algorithm that checks each element of the list sequentially, making it effective for small or unsorted lists. Its time complexity is $O(n)O(n)O(n)$. Binary search, on the other hand, is more efficient with a time complexity of $O(\log n)O(\log n)O(\log n)$, but it requires the list to be sorted. Binary search is preferred for large, sorted lists due to its faster performance.