

AIM: HEAP SORT & SPARSE MATRIX

THEORY:

Heap Sort:

Heap Sort is a popular and efficient sorting algorithm in computer programming. Learning how to write the heap sort algorithm requires knowledge of two types of data structures - arrays and trees. Heap sort works by visualizing the elements of the array as a special kind of complete binary tree called a heap.

Relationship between Array Indexes and Tree Elements

A complete binary tree has an interesting property that we can use to find the children and parents of any node.

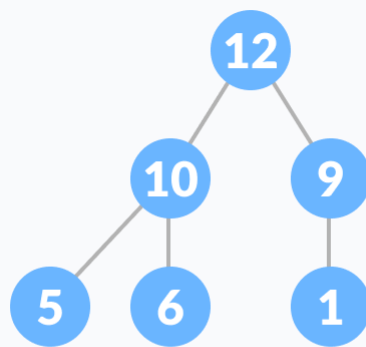
If the index of any element in the array is i , the element in the index $2i+1$ will become the left child and element in $2i+2$ index will become the right child. Also, the parent of any element at index i is given by the lower bound of $(i-1)/2$.

What is Heap Data Structure?

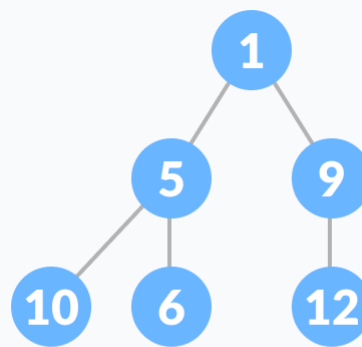
Heap is a special tree-based data structure. A binary tree is said to follow a heap data structure if

- it is a complete binary tree
- All nodes in the tree follow the property that they are greater than their children i.e. the largest element is at the root and both its children and smaller than the root and so on. Such a heap is called a max-heap. If instead, all nodes are smaller than their children, it is called a min-heap

The following example diagram shows Max-Heap and Min-Heap.



Max Heap

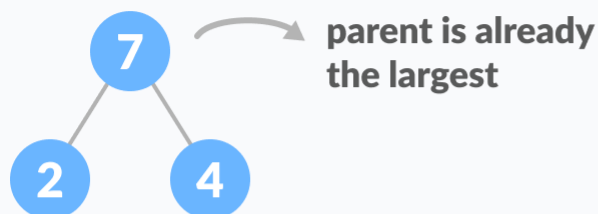


Min Heap

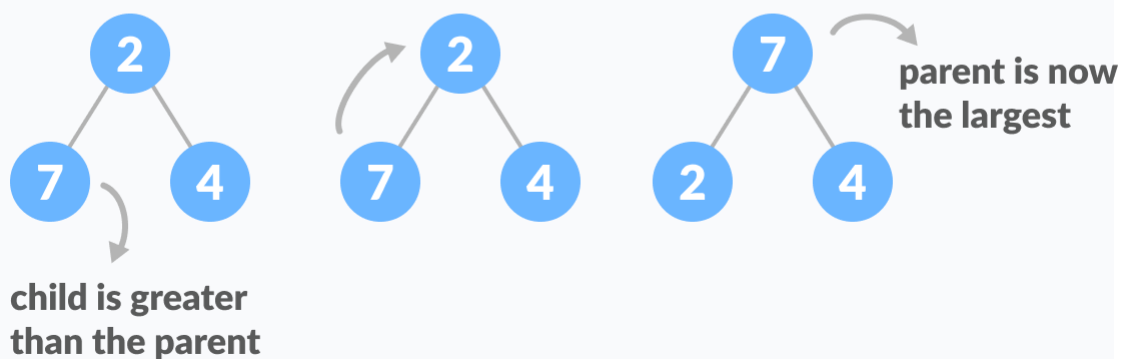
How to "heapify" a tree

Starting from a complete binary tree, we can modify it to become a Max-Heap by running a function called heapify on all the non-leaf elements of the heap. Since heapify uses recursion, it can be difficult to grasp. So let's first think about how you would heapify a tree with just three elements.

Scenario-1



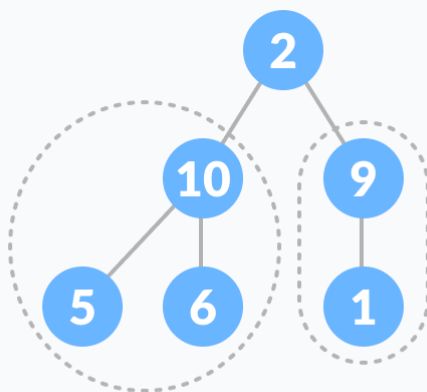
Scenario-2



The example above shows two scenarios - one in which the root is the largest element and we don't need to do anything. And another in which the root had a larger element as a child and we needed to swap to maintain max-heap property.

If you're worked with recursive algorithms before, you've probably identified that this must be the base case.

Now let's think of another scenario in which there is more than one level.

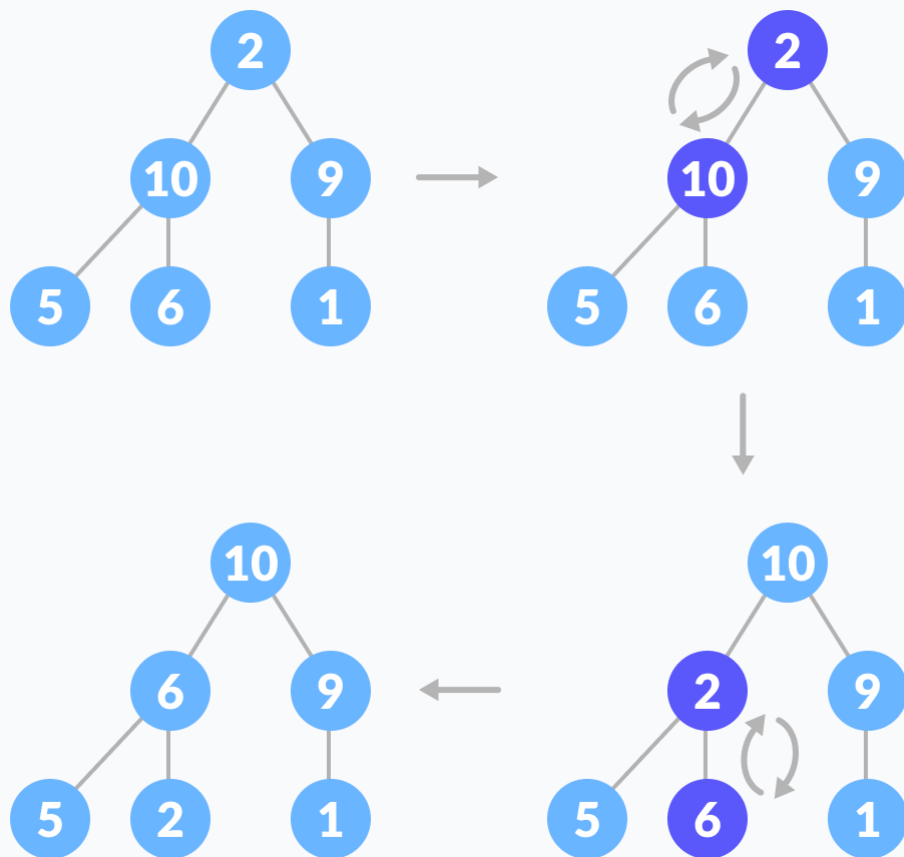


**both subtrees of the root
are already max-heaps**

How to heapify root element when its subtrees are already max heaps

The top element isn't a max-heap but all the sub-trees are max-heaps.

To maintain the max-heap property for the entire tree, we will have to keep pushing 2 downwards until it reaches its correct position.



Thus, to maintain the max-heap property in a tree where both sub-trees are max-heaps, we need to run heapify on the root element repeatedly until it is larger than its children or it becomes a leaf node.

This function works for both the base case and for a tree of any size. We can thus move the root element to the correct position to maintain the max-heap status for any tree size as long as the sub-trees are max-heaps.

Build max-heap

To build a max-heap from any tree, we can thus start heapifying each sub-tree from the bottom up and end up with a max-heap after the function is applied to all the elements including the root element.

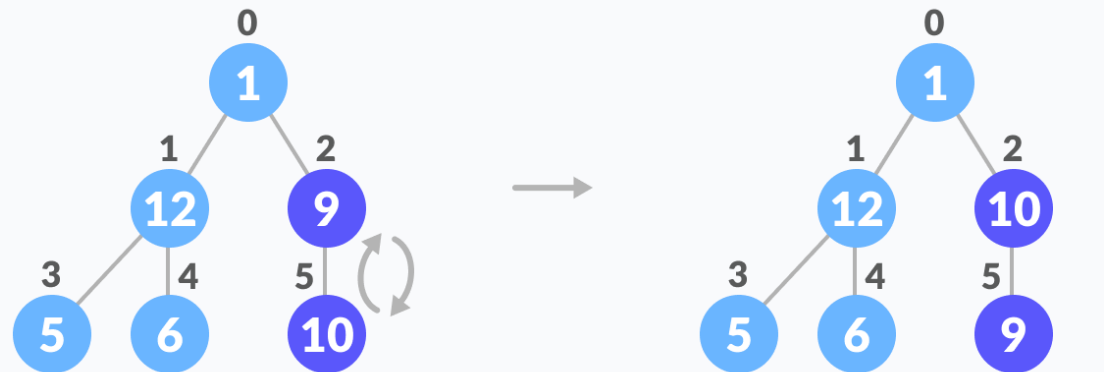
In the case of a complete tree, the first index of a non-leaf node is given by $n/2 - 1$. All other nodes after that are leaf-nodes and thus don't need to be heapified.

	0	1	2	3	4	5
arr	1	12	9	5	6	10

n = 6

i = $6/2 - 1 = 2$ # loop runs from 2 to 0

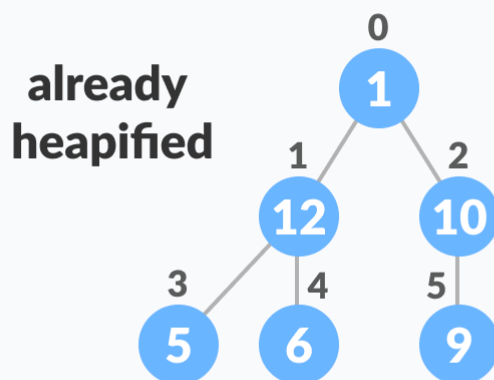
$i = 2 \rightarrow \text{heapify}(\text{arr}, 6, 2)$



0	1	2	3	4	5
1	12	9	5	6	10

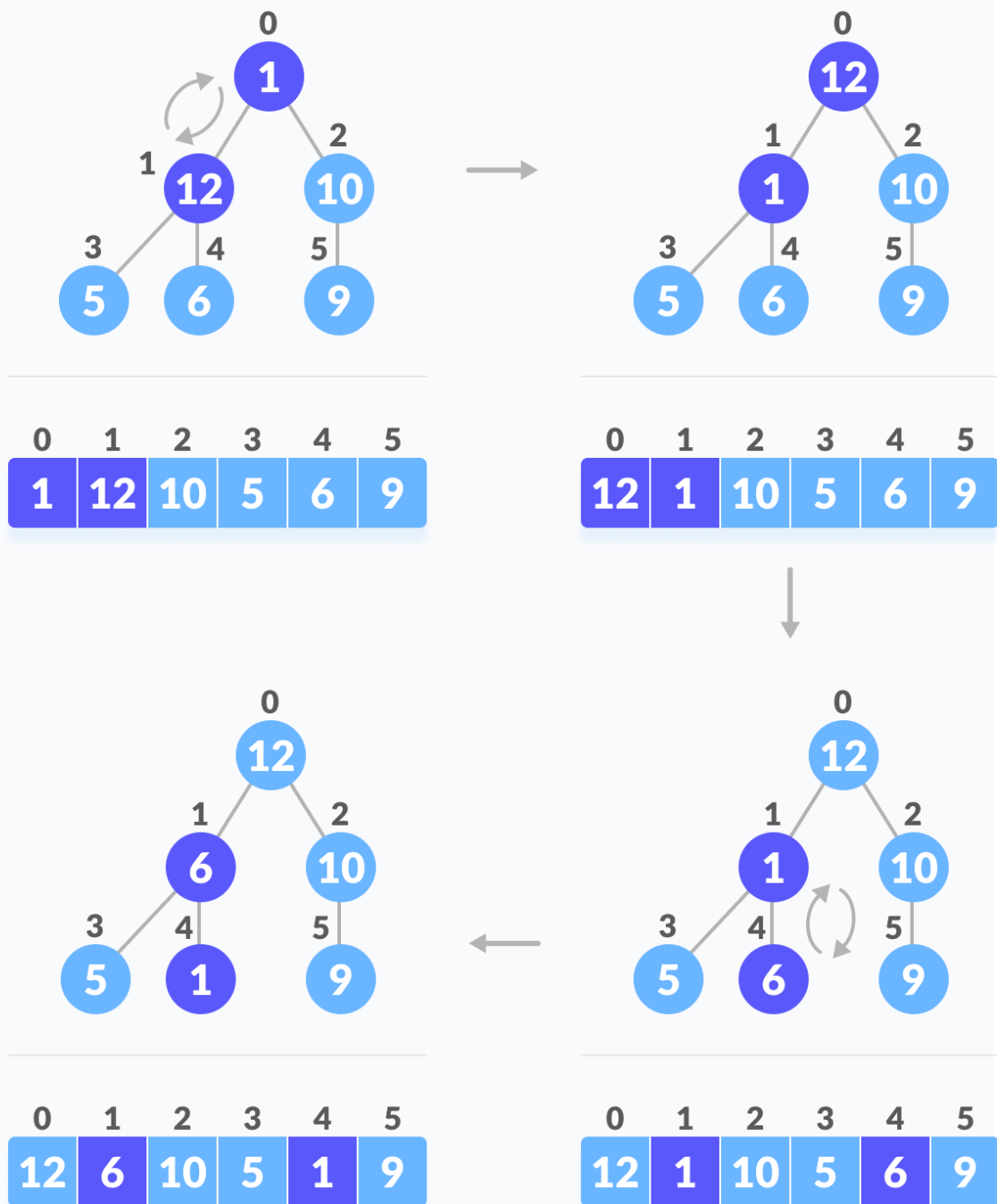
0	1	2	3	4	5
1	12	10	5	6	9

$i = 1 \rightarrow \text{heapify}(\text{arr}, 6, 1)$



0	1	2	3	4	5
1	12	10	5	6	9

$i = 0 \rightarrow \text{heapify}(\text{arr}, 6, 0)$



Steps to build max heap for heap sort

As shown in the above diagram, we start by heapifying the lowest smallest trees and gradually move up until we reach the root element.

If you've understood everything till here, congratulations, you are on your way to mastering the Heap sort.

Working of Heap Sort

1. Since the tree satisfies Max-Heap property, then the largest item is stored at the root node.
2. **Swap:** Remove the root element and put at the end of the array (nth position) Put the last item of the tree (heap) at the vacant place.
3. **Remove:** Reduce the size of the heap by 1.
4. **Heapify:** Heapify the root element again so that we have the highest element at root.
5. The process is repeated until all the items of the list are sorted.



Heap Sort Complexity**Time Complexity**

Best	$O(n \log n)$
Worst	$O(n \log n)$
Average	$O(n \log n)$

Space Complexity	$O(1)$
-------------------------	--------

Stability	No
------------------	----

Sparse Matrix:

A matrix is a two-dimensional data object made of m rows and n columns, therefore having total $m \times n$ values. If most of the elements of the matrix have **0 value**, then it is called a sparse matrix.

Why to use Sparse Matrix instead of simple matrix ?

- **Storage:** There are lesser non-zero elements than zeros and thus lesser memory can be used to store only those elements.
- **Computing time:** Computing time can be saved by logically designing a data structure traversing only non-zero elements..

Representing a sparse matrix by a 2D array leads to wastage of lots of memory as zeroes in the matrix are of no use in most of the cases. So, instead of storing zeroes with non-zero elements, we only store non-zero elements. This means storing non-zero elements with **triples- (Row, Column, value)**.

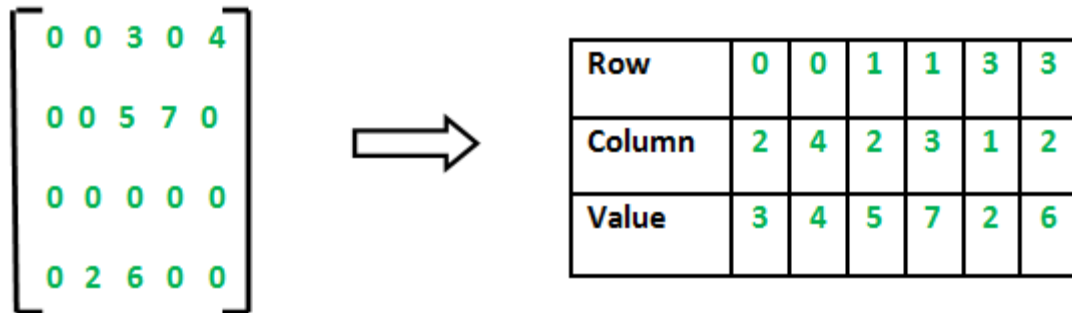
Sparse Matrix Representations can be done in many ways following are two common representations:

1. Array representation
2. Linked list representation

Method 1: Using Arrays:

2D array is used to represent a sparse matrix in which there are three rows named as

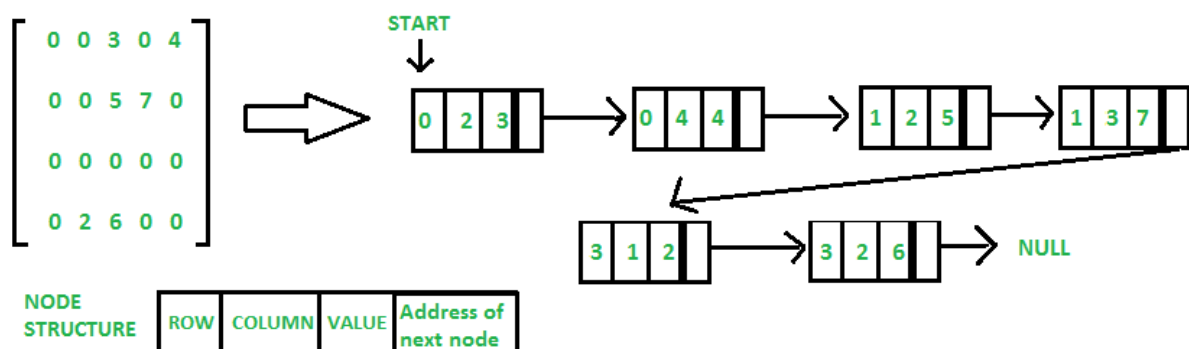
- **Row:** Index of row, where non-zero element is located
- **Column:** Index of column, where non-zero element is located
- **Value:** Value of the non zero element located at index – (row,column)



Method 2: Using Linked Lists

In linked list, each node has four fields. These four fields are defined as:

- **Row:** Index of row, where non-zero element is located
- **Column:** Index of column, where non-zero element is located
- **Value:** Value of the non zero element located at index – (row,column)
- **Next node:** Address of the next node



1) Implementation of Min Heap and Max Heap Application: Heap Sort

SOURCE CODE:

```
#include <iostream>

using namespace std;
void maxHeapify(int arr[], int n, int i)
{
    int largest = i;
    int l = 2 * i + 1;
    int r = 2 * i + 2;

    if (l < n && arr[l] > arr[largest])
    {
        largest = l;
    }

    if (r < n && arr[r] > arr[largest])
    {
        largest = r;
    }

    if (largest != i)
    {
        swap(arr[i], arr[largest]);
        maxHeapify(arr, n, largest);
    }
}

void minHeapify(int arr[], int n, int i)
{
    int smallest = i;
    int l = 2 * i + 1;
    int r = 2 * i + 2;

    if (l < n && arr[l] < arr[smallest])
    {
        smallest = l;
    }

    if (r < n && arr[r] < arr[smallest])
    {
        smallest = r;
    }

    if (smallest != i)
    {
        swap(arr[i], arr[smallest]);
    }
}
```

```
        minHeapify(arr, n, smallest);
    }
}

void maxHeapSort(int arr[], int n)
{
    for (int i = n / 2 - 1; i >= 0; i--)
    {
        maxHeapify(arr, n, i);
    }

    for (int i = n - 1; i > 0; i--)
    {
        swap(arr[0], arr[i]);
        maxHeapify(arr, i, 0);
    }
}

void minHeapSort(int arr[], int n)
{
    for (int i = n / 2 - 1; i >= 0; i--)
    {
        minHeapify(arr, n, i);
    }

    for (int i = n - 1; i >= 0; i--)
    {
        swap(arr[0], arr[i]);
        minHeapify(arr, i, 0);
    }
}

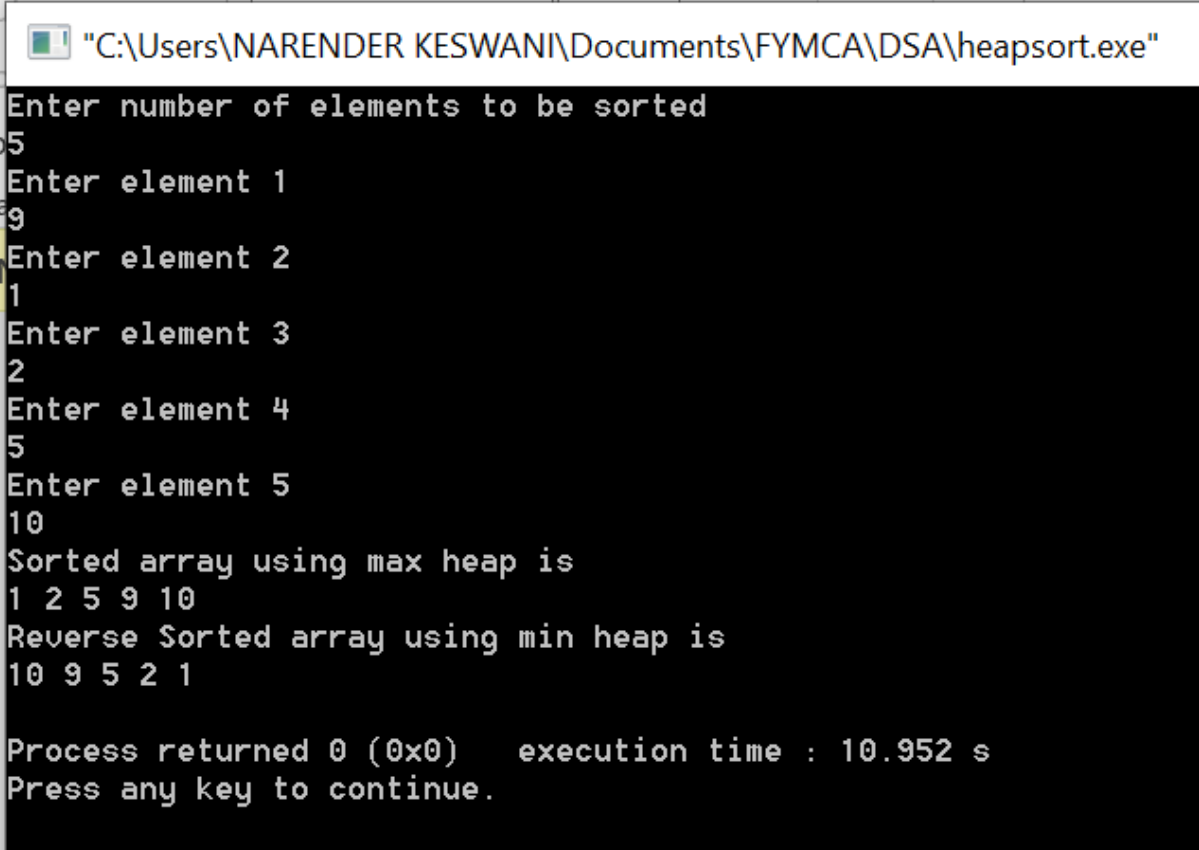
void printArray(int arr[], int n)
{
    for (int i = 0; i < n; ++i)
    {
        cout << arr[i] << " ";
    }
    cout << "\n";
}

int main()
{
    int n;
    cout<<"Enter number of elements to be sorted"<<endl;
    cin>>n;
    int arr[n];

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

```
        cout << "Enter element "<<i+1<<"\n";  
        cin >> arr[i];  
    }  
  
    cout << "Sorted array using max heap is \n";  
    maxHeapSort(arr, n);  
    printArray(arr, n);  
  
    cout << "Reverse Sorted array using min heap is \n";  
    minHeapSort(arr, n);  
    printArray(arr, n);  
}
```

OUTPUT:



```
"C:\Users\NARENDER KESWANI\Documents\FYMCA\DSA\heapsort.exe"  
Enter number of elements to be sorted  
5  
Enter element 1  
9  
Enter element 2  
1  
Enter element 3  
2  
Enter element 4  
5  
Enter element 5  
10  
Sorted array using max heap is  
1 2 5 9 10  
Reverse Sorted array using min heap is  
10 9 5 2 1  
  
Process returned 0 (0x0)   execution time : 10.952 s  
Press any key to continue.
```

2) Demonstrate application of linked list - Sparse matrix

SOURCE CODE:

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

class Node
{
    public:
    int row;
    int col;
    int data;
    Node *next;
};

void create_new_node(Node **p, int row_index,
                    int col_index, int x)
{
    Node *temp = *p;
    Node *r;

    if (temp == NULL)
    {
        temp = new Node();
        temp->row = row_index;
        temp->col = col_index;
        temp->data = x;
        temp->next = NULL;
        *p = temp;
    }

    else
    {
        while (temp->next != NULL)
            temp = temp->next;

        r = new Node();
        r->row = row_index;
        r->col = col_index;
        r->data = x;
        r->next = NULL;
        temp->next = r;
    }
}

void printList(Node *start)
{
    Node *ptr = start;
    cout << "row_position:";
    while (ptr != NULL)
    {
```

```

        cout << ptr->row << " ";
        ptr = ptr->next;
    }
    cout << endl;
    cout << "column_position:";

    ptr = start;
    while (ptr != NULL)
    {
        cout << ptr->col << " ";
        ptr = ptr->next;
    }
    cout << endl;
    cout << "Value:";
    ptr = start;

    while (ptr != NULL)
    {
        cout << ptr->data << " ";
        ptr = ptr->next;
    }
}

int main()
{
    int row, col;

    cout<<"Enter number of rows for sparse matrix"<<endl;
    cin>>row;
    cout<<"Enter number of columns for sparse matrix"<<endl;
    cin>>col;

    int sparseMatrix[row][col];

    for(int i = 0; i < row; i++)
    {
        for(int j = 0; j < col; j++)
        {
            cout<<"Enter the value for["<<i<<"]["<<j<<"]"<<endl;
            cin>>sparseMatrix[i][j];
        }
    }

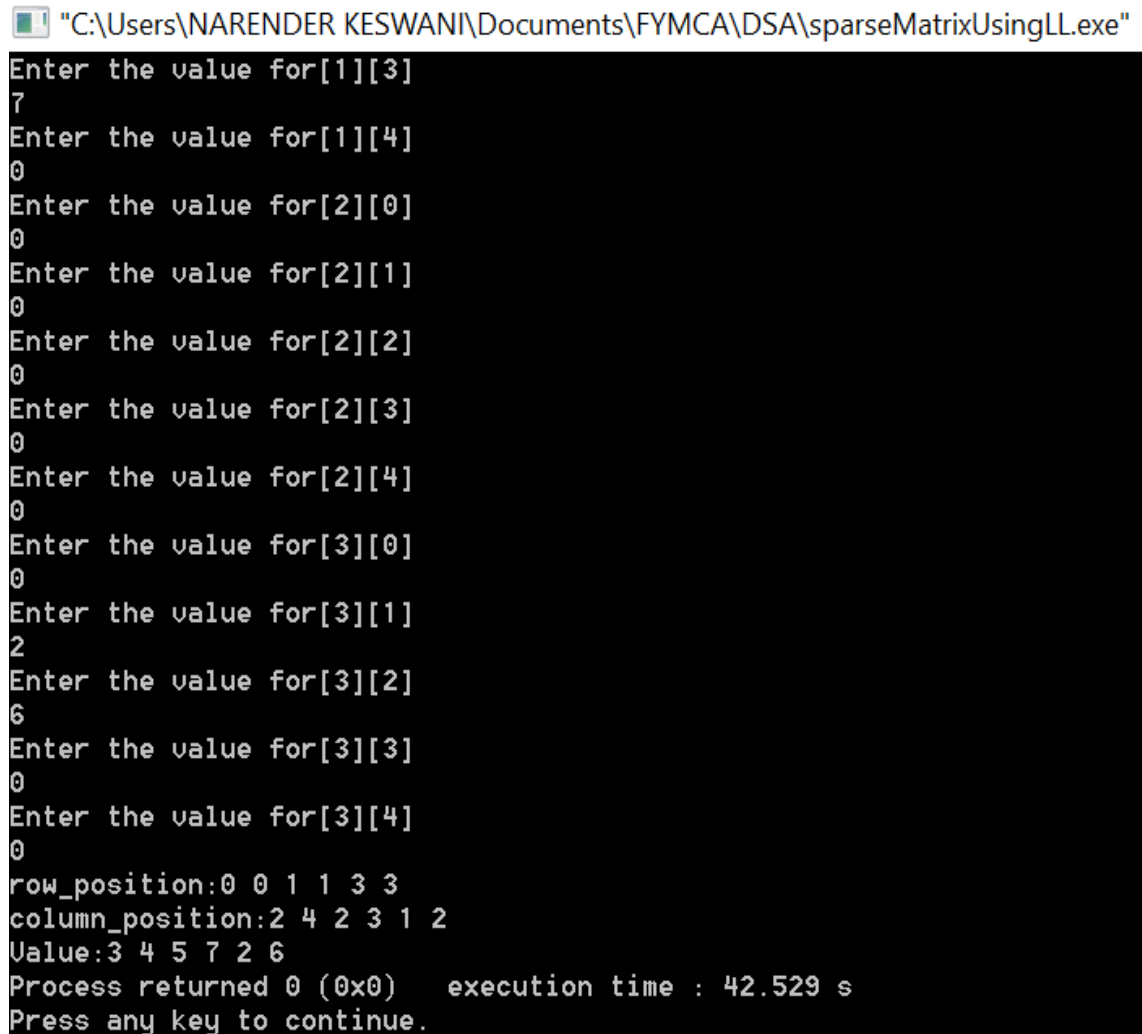
    Node *first = NULL;
    for(int i = 0; i < row; i++)
    {
        for(int j = 0; j < col; j++)
        {
            if (sparseMatrix[i][j] != 0)
            {
                create_new_node(&first, i, j, sparseMatrix[i][j]);
            }
        }
    }
}

```



```
    }  
    }  
    }  
    printList(first);  
  
    return 0;  
}
```

OUTPUT:



"C:\Users\NARENDER KESWANI\Documents\FYMCA\DSA\sparseMatrixUsingLL.exe"

```
Enter the value for[1][3]  
7  
Enter the value for[1][4]  
0  
Enter the value for[2][0]  
0  
Enter the value for[2][1]  
0  
Enter the value for[2][2]  
0  
Enter the value for[2][3]  
0  
Enter the value for[2][4]  
0  
Enter the value for[3][0]  
0  
Enter the value for[3][1]  
2  
Enter the value for[3][2]  
6  
Enter the value for[3][3]  
0  
Enter the value for[3][4]  
0  
row_position:0 0 1 1 3 3  
column_position:2 4 2 3 1 2  
Value:3 4 5 7 2 6  
Process returned 0 (0x0)    execution time : 42.529 s  
Press any key to continue.
```

CONCLUSION:

From this practical, I have learned how to implement heap sort using max and min heap, also learned about sparse matrix implementation using linked lists.