# **Advanced Data Structures**

## **Assignment-10**

Name: Ayush Sanjay Dhangar

Roll no: 42

Prn:12210406

Div: IT-A (Batch 2)

Problem statement:Write C/C++ program to implement Heap data structure. Create Max-heap and insert elements into it.

* **Swap Function**: The swap function facilitates swapping the values of two elements, used during heapifying and deletion operations.
* **Heapify Function**: This function ensures the heap property is maintained. It takes an array representing the heap, its size, and the index of the current node to heapify. It recursively compares the node with its children, swapping if necessary to maintain the heap property.
* **Insert Function**: The insert function adds a new element to the heap. If the heap is empty, the new element is placed at index 0; otherwise, it's appended to the end of the array and then adjusted to maintain the heap property.
* **DeleteRoot Function**: This function deletes a specified element (root) from the heap. It first finds the index of the element to be deleted, swaps it with the last element in the heap, reduces the size of the heap, and then heapifies the heap from the root to maintain the heap property.
* **PrintArray Function**: This function simply prints the elements of the array.
* **Main Function**: In the main function, a Max-Heap is initialized as an array. Several elements are inserted into the heap using the insert function. Then, one element (4) is deleted using the deleteRoot function. Finally, the contents of the heap are printed before and after the deletion.

Code:

// Max-Heap data structure in C

#include <stdio.h>

int size = 0;

void swap(int \*a, int \*b)

{

  int temp = \*b;

  \*b = \*a;

  \*a = temp;

}

void heapify(int array[], int size, int i)

{

  if (size == 1)

  {

    printf("Single element in the heap");

  }

  else

  {

    int largest = i;

    int l = 2 \* i + 1;

    int r = 2 \* i + 2;

    if (l < size && array[l] > array[largest])

      largest = l;

    if (r < size && array[r] > array[largest])

      largest = r;

    if (largest != i)

    {

      swap(&array[i], &array[largest]);

      heapify(array, size, largest);

    }

  }

}

void insert(int array[], int newNum)

{

  if (size == 0)

  {

    array[0] = newNum;

    size += 1;

  }

  else

  {

    array[size] = newNum;

    size += 1;

    for (int i = size / 2 - 1; i >= 0; i--)

    {

      heapify(array, size, i);

    }

  }

}

void deleteRoot(int array[], int num)

{

  int i;

  for (i = 0; i < size; i++)

  {

    if (num == array[i])

      break;

  }

  swap(&array[i], &array[size - 1]);

  size -= 1;

  for (int i = size / 2 - 1; i >= 0; i--)

  {

    heapify(array, size, i);

  }

}

void printArray(int array[], int size)

{

  for (int i = 0; i < size; ++i)

    printf("%d ", array[i]);

  printf("\n");

}

int main()

{

  int array[10];

  insert(array, 3);

  insert(array, 4);

  insert(array, 9);

  insert(array, 5);

  insert(array, 2);

  printf("Max-Heap array: ");

  printArray(array, size);

  deleteRoot(array, 4);

  printf("After deleting an element: ");

  printArray(array, size);

}

Output:



Conclusion:

This C code offers a practical implementation of a Max-Heap data structure, showcasing fundamental operations like insertion and deletion while adhering to the essential heap property where parent nodes hold greater values than their children. Through functions like insert and deleteRoot, it enables dynamic management of elements within the heap, adjusting the structure as necessary to maintain its integrity. The code's clarity and simplicity allow for easy comprehension and modification, making it a valuable resource for those seeking to understand Max-Heap concepts or integrate heap functionality into their C projects.