Name	Shubhan Singh
UID no.	2022300118
Experiment No.	9

Program 1	
PROBLEM STATEMENT :	*Create a max-heap ADT using array and implement various operations*
ALGORITHM:	1. **initHeap(int capacity): This function initializes a new heap with a given capacity.** o Allocate memory for a new MaxHeap structure. o Set the capacity of the heap to the provided capacity. o Set the size of the heap to 0. o Allocate memory for the array of the heap with size capacity + 1. o Return the pointer to the new heap. 2. **destroyHeap(MaxHeap* heap): This function frees the memory allocated for the heap.** o Free the memory allocated for the array of the heap. o Free the memory allocated for the heap. 3. **heapify(MaxHeap* heap, int i): This function restores the heap-order property for the heap array at index i.** o Initialize pointers to the parent, left child, and right child of the node at index i. o If both children are NULL, return. o If only the left child is NULL, swap the parent and right child if the right child is greater than the parent, and recursively call heapify on the right child. o If only the right child is NULL, swap the parent and left child if the left child is greater than the

	parent, and recursively call heapify on the left child. o If both children are not NULL, find the maximum of the parent, left child, and right child. If the maximum is the parent, return. If the maximum is the right child, swap the parent and right child and recursively call heapify on the right child. If the maximum is the left child, swap the parent and left child and recursively call heapify on the left child. 4. **insert(MaxHeap* heap, int value): This function inserts a value into the heap.** o If the heap is full, print a message and return. o Increment the size of the heap and insert the value at the end of the heap array. o While the parent of the inserted node is greater than or equal to 1, call heapify on the parent. 5. **peek_max(MaxHeap* heap):** This function displays the maximum element in the heap. o If the heap is empty, print a message and return. o Print the first element of the heap array. 6. **extractMax(MaxHeap* heap):** This function extracts the maximum element from the heap. o Store the first element of the heap array in a variable. o Replace the first element of the heap array with the last element and decrement the size of the heap. o If the size of the heap is greater than 1, call heapify on the first element. o Return the stored variable. 7. **display_heap(MaxHeap* heap, int stop_idx):** This function displays the elements of the heap up to a given index. o Print the elements of the heap array from index 1 to stop_idx. 8. **constructHeap(int *arr, int arr_length): This function builds a max-heap from an array using Floyd's method.** o Initialize a new heap with capacity equal to the length of the array. o Copy the elements of the array to the heap array.

	○ Set the size of the heap to the length of the array. ○ For each node from the middle of the heap to the first node, call heapify on the node. ○ Return the heap. **9. heapSort_ascending(MaxHeap* heap): This function sorts the elements of the heap in ascending order.** ○ Store the initial size of the heap. ○ Swap the first and last elements of the heap array and decrement the size of the heap. ○ For each node from the first node to the second last node, call heapify on the node, swap the node with the last node of the heap, and decrement the size of the heap. ○ Set the size of the heap to the initial size.
PROGRAM:	

```c
/*
 * File: max_heap.c
 * Author: Siddhartha Chandra
 * Email: siddhartha_chandra@spit.ac.in
 * Created: November 5, 2023
 * Description: Create a max-heap ADT using array and implement various
operations
 */


//We will be assuming 1 based indexing in the heap array


#include <stdio.h>
#include <stdlib.h>
```

```c
#include<math.h>
#include <stdbool.h>

// Define MaxHeap ADT
typedef struct {
    int *array;
    int size;
    int capacity;
} MaxHeap;

void swap(int *a,int*b){
    int temp;
    temp=*a;
    *a=*b;
    *b=temp;
}

/// @brief Creates an empty max-heap of size 'capacity'
/// @param capacity - max size of heap
/// @return Pointer to MaxHeap
MaxHeap *initHeap(int capacity){
    MaxHeap* newheap=malloc(sizeof(MaxHeap));
    newheap->capacity=capacity;
    newheap->size=0;
    newheap->array=calloc(newheap->capacity+1,sizeof(int));
    return newheap;
}
```

```
// Delete and free the max-heap structure
void destroyHeap(MaxHeap* heap){
    free(heap->array);
    free(heap);
}

/// @brief This restores the heap-order property for max-heap array at index
'i'
/// @param heap
/// @param i - index of the max-heap array, which might potentialy be breaking
the heap order
void heapify(MaxHeap* heap, int i){
    int *parent,*left,*right;
    parent=heap->array+i;
    if(2*i<=heap->size){
        left=heap->array+2*i;
    }
    else{left=NULL;}
    if((2*i+1)<=heap->size){
        right=heap->array+(2*i+1);
    }
    else{right=NULL;}
    if(left==NULL && right==NULL){
        return;
    }
    if(left==NULL){
```

```
        if(*right>*parent){
            swap(parent,right);
            heapify(heap,2*i+1);
        }
        else{return;}
    }
    else if(right==NULL){
        if(*left>*parent){
            swap(parent,left);
            heapify(heap,2*i);
        }
        else{return;}
    }
    else{
        int *max;
        if(*parent>*left){max=parent;}
        else{max=left;}
        if(*right>*max){max=right;}
        if(max==parent){return;}
        if(max==right){
            swap(right,parent);
            heapify(heap,2*i+1);
        }
        else{
            swap(left,parent);
            heapify(heap,2*i);
        }
```

```
    }
}

/// @brief This inserts a value into a max-heap
/// @param heap
/// @param value
void insert(MaxHeap* heap, int value){
    if(heap->size==heap->capacity){
        printf("Heap is full!\n");
        return;
    }
    heap->size++;
    heap->array[heap->size]=value;
    int parent=heap->size/2;
    while(parent>=1){
        heapify(heap,parent);
        parent=parent/2;
    }
}

// displays the max element in the MaxHeap array
void peek_max(MaxHeap* heap){
    if(heap->size==0){
        printf("Heap is empty!\n");
        return;
    }
    printf("The maximum element in the given MaxHeap array is: %d\n",heap-
```

```
>array[1]);
}

// extracts the max element from the MaxHeap array
int extractMax(MaxHeap* heap){
    int retval=heap->array[1];
    heap->array[1]=heap->array[heap->size];
    heap->size--;
    if(heap->size>1){
        heapify(heap,1);
    }
    return retval;
}


/// @brief - Display the given MaxHeap in a visually clear way.
/// @param heap
/// @param stop_idx - This index in the MaxHeap array corresponds to the last
item of the heap
void display_heap(MaxHeap* heap, int stop_idx){
    printf("[");
    for(int i=1;i<stop_idx;i++){
        printf("%d, ",heap->array[i]);
    }
    printf("%d]\n",heap->array[stop_idx]);
}
```

```
/// @brief Build max-heap using the Floyd's method. This method should call
initHeap
/// @param heap
MaxHeap *constructHeap(int *arr, int arr_length){
    MaxHeap* heap=initHeap(arr_length);
    for(int i=0;i<arr_length;i++){
        heap->array[i+1]=arr[i];
    }
    heap->size=arr_length;
    for(int i=arr_length/2;i>=1;i--){
        heapify(heap,i);
    }

    return heap;
}


// Sorts the given MaxHeap array in ascending order.
// Post running this function, heap->array should contain the elements in the
sorted order
// NOTE: This function should not use any additional data structures
void heapSort_ascending(MaxHeap* heap){
    int initialsize=heap->size;
    swap(&(heap->array[1]),&(heap->array[heap->size]));
    heap->size--;
    for(int i=0;i<initialsize-1;i++){
```

```
        heapify(heap,1);
        swap(&(heap->array[1]),&(heap->array[heap->size]));
        heap->size--;
    }
    heap->size=initialsize;
}

int main(){
    printf("Enter capacity of heap: ");
    int cap,n;
    scanf("%d",&cap);
    MaxHeap* Heap=initHeap(cap);
    int choose=1,val;
    while(choose){
        printf("Enter\n1 for insert\n2 for extract_max\n3 for peek_max\n4 for
heapsort\n5 to display heap\n6 to construct heap\n0 to exit\n");
        scanf("%d",&choose);
        switch(choose){
            case 1:
                printf("Enter number of values to insert: ");
                scanf("%d",&n);
                printf("Enter all %d values\n",n);
                for(int i=0;i<n;i++){
                    scanf("%d",&val);
                    insert(Heap,val);
                }
                while((getchar())!='\n');//deletes any extra integers if
```

```
entered
                break;
            case 2:
                if(Heap->size==0){
                    printf("Heap is empty!\n");
                }
                else{
                    val=extractMax(Heap);
                    printf("The extracted value is %d\n",val);
                }
                break;
            case 3:
                peek_max(Heap);
                break;
            case 4:
                heapSort_ascending(Heap);
                printf("The array sorted using heapsort is:\n");
                display_heap(Heap,Heap->size);
                break;
            case 5:
                printf("The heap array is as follows(starting from index 1,
the first index is empty):\n");
                display_heap(Heap,Heap->size);
                break;
            case 6:
            {
                printf("Enter size of heap: ");
```

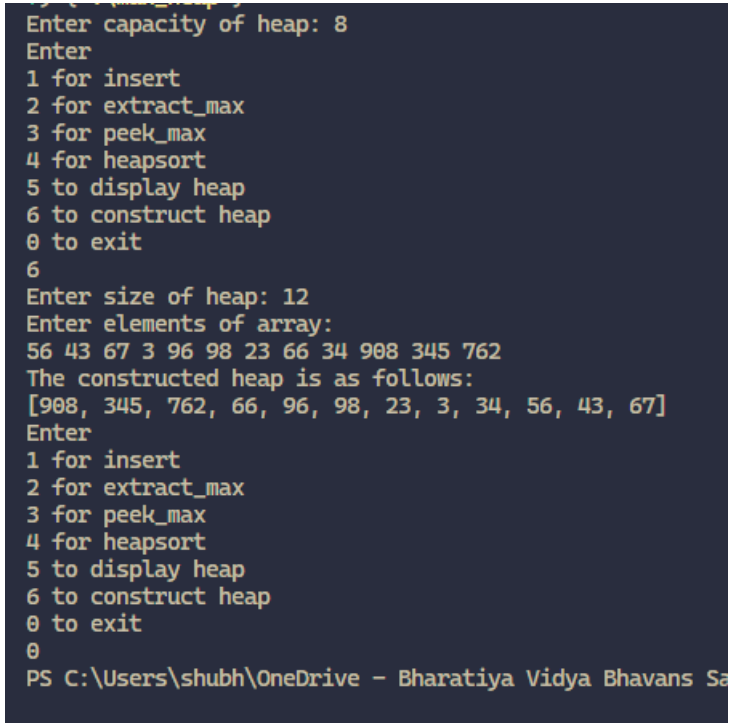
```
                scanf("%d",&n);
                int arr[n];
                printf("Enter elements of array:\n");
                for(int i=0;i<n;i++){
                    scanf("%d",&arr[i]);
                }
                MaxHeap* floyds=constructHeap(arr,n);
                printf("The constructed heap is as follows:\n");
                display_heap(floyds,n);
                break;
            }
            default:
                break;
        }
    }
    destroyHeap(Heap);

return 0;
}
```

RESULT:

```
Enter capacity of heap: 15
Enter
1 for insert
2 for extract_max
3 for peek_max
4 for heapsort
5 to display heap
0 to exit
1
Enter number of values to insert: 13
Enter all 13 values
1 43 6 22 87 2 9 278 65 46 33 12 456
Enter
1 for insert
2 for extract_max
3 for peek_max
4 for heapsort
5 to display heap
0 to exit
5
The heap array is as follows(starting from index 1, the first index is empty):
[456, 87, 278, 65, 46, 12, 6, 1, 43, 22, 33, 2, 9]
Enter
1 for insert
2 for extract_max
3 for peek_max
4 for heapsort
5 to display heap
0 to exit
3
The maximum element in the given MaxHeap array is: 456
Enter
1 for insert
2 for extract_max
3 for peek_max
4 for heapsort
5 to display heap
0 to exit
2
The extracted value is 456
```

```
Enter
1 for insert
2 for extract_max
3 for peek_max
4 for heapsort
5 to display heap
0 to exit
5
The heap array is as follows(starting from index 1, the first index is empty):
[278, 87, 12, 65, 46, 9, 6, 1, 43, 22, 33, 2]
Enter
1 for insert
2 for extract_max
3 for peek_max
4 for heapsort
5 to display heap
0 to exit
4
The array sorted using heapsort is:
[1, 2, 6, 9, 12, 22, 33, 43, 46, 65, 87, 278]
Enter
1 for insert
2 for extract_max
3 for peek_max
4 for heapsort
5 to display heap
0 to exit
3
The maximum element in the given MaxHeap array is: 1
Enter
1 for insert
2 for extract_max
3 for peek_max
4 for heapsort
5 to display heap
0 to exit
0
PS C:\Users\shubh\OneDrive - Bharatiya Vidya Bhavans Sardar Patel Institute Of Te
```

(after heapsort, peek_max shows the minimum element, as the heap array is sorted)



THEORY:	**Heap Data Structure:** A heap is a specialized tree-based data structure that satisfies the heap property. Typically, heaps are implemented as arrays, where each element's index determines its relationship with its parent and children. There are two main types of heaps: min-heaps and max-heaps. • **Min-Heap:** In a min-heap, for every node 'i' other than the root, the value of 'i' is greater than or equal to the value of its parent. • **Max-Heap:** In a max-heap, for every node 'i' other than the root, the value of 'i' is less than or equal to the value of its parent. **Use in Priority Queues:** Priority queues are abstract data types similar to regular queues but with each element having a priority assigned to it. The elements with higher priorities are served before elements with lower priorities.

	Heaps are commonly used to implement priority queues due to their efficient nature in maintaining the highest (or lowest) priority element at the root, allowing for quick access and removal of the highest priority item. • **Enqueue (Insertion):** When adding an element to the priority queue, it's inserted as a leaf node in the heap and then "bubbled up" or "sifted up" to the correct position to maintain the heap property (i.e., parent has higher priority than its children in a max-heap or lower priority in a min-heap). • **Dequeue (Removal):** Removing an element from the priority queue involves extracting the root element (the one with the highest or lowest priority) and then rearranging the remaining elements to restore the heap property. The last element in the heap usually replaces the root and is then "sifted down" or "bubbled down" to its appropriate position. **Performance:** Heaps offer efficient insertion and extraction of elements: • Both enqueue and dequeue operations in a heap-based priority queue typically run in O(log n) time complexity, where 'n' is the number of elements in the heap. • Finding the highest or lowest priority element (peeking) can be done in constant time, as it's always at the root of the heap.
CONCLUSION:	Heaps are a useful data structure which allow fast access to the maximum or minimum element in an array, and are used to implement priority queues. They can also be used to implement heap sort algorithm, which sorts an array in O(nlogn) time.