# Lab-4_Documentation

## 11.c

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

//Code to parse integer from a string
int stoi(char *str)                    // converts string input str to integer
{
        int x;
        sscanf(str, "%d", &x);
        return x;
}

int heapSize=0;

void swap(int *x, int *y)                    // swap the input integer pointer values
{
   int temp;
   temp = *x;
   *x = *y;
   *y = temp;
        }

void maxHeapify(int A[],int i)            // takes an index i , makes it max_heapify until it
reach condition of max_heapify at that index or until it reaches a leaf (as leaf are
by default max_heapify).
{
   if(i>heapSize)
                        // checks if the index is greater than heapsize, if yes return -1
        return -1;
   int lar = i;            // store index in an integer lar
   if(A[i]<A[2*i+1] && 2*i+1<heapSize)
// if A[i] is smaller than A[2i+1] (it's child) then store the larger value in the integer
lar
         lar = 2*i+1;              // now lar can be i or 2*i+1
```

```
    if(A[2*i+2]>A[lar] && 2*i+2<heapSize)   // now if A[2*i+2] is the greater A[lar], means
A[2*i+2] is the largest , swap it with the A[i] and do max_heapify at 2*i +2 th
position
    {
        lar = 2*i+2;
        swap(&A[i], &A[lar]);
        maxHeapify(A, lar);
    }
    else if(lar!=i)                  // means A[2*i+1] is the largest , swap A[i] and A[2*i+1]
and do maxHeapify at 2*i+1 th position
    {
        swap(&A[i], &A[2*i+1]);
        maxHeapify(A, 2*i+1);
    }
    else
        // means A[i] is already maximum , therefore come out of the maxHeapify
        return;

        }

int insertKey(int A[], int key)             // insert key
{
    int flag=0;
    A[heapSize]=key;
                              // first insert key at the end , and increasing the heapSize
    int i = heapSize;
    heapSize++;
    while(i>0)              // now inserted value can be greater than its parent , so swap
    {
        if(A[i]>A[(i-1)/2])       // if greater than its parent swap
        {
            swap(&A[i], &A[(i-1)/2]);
            i= (i-1)/2;
        }
        else                      // else maxheap condition is verified , come out
            return 1;
        flag=1;
    }

    //printf("%d\n", A[heapSize-1]);
    if(flag==1 || i==0)
        return 1;
```

```
    return -1;
        }

int increaseKey(int A[], int i, int newVal)       // increase the value at index i to a newValue
{

    if(i>=heapSize)
        return -1;
    A[i]=newVal;                    // increase the value at index i to a newValue
    while(i>0)              // now increased value can be greater than its parent , so swap
    {
        if(A[i]>A[(i-1)/2])         // if greater than its parent swap
        {
            swap(&A[i], &A[(i-1)/2]);
            i= (i-1)/2;
        }
        else            // else maxheap condition is verified , come out
            return 1;
    }
    return 1;
}


int extractMax(int A[])         // takes out the maximum element out of the maxHeap
{
    int val;
    if(heapSize==0)
        return -1;
    val=A[0];               // A[0]  is the maximum element in the maxHeap, store in val
    A[0]=A[heapSize-1];         // now changing value at A[0]
    heapSize--;                 // decrease the heapSize
    maxHeapify(A,0);
                // now if A[0] is smaller than its child , swap , means maxheapify
    return val;             // extract the maximum
        }

void print(int A[])                 // print the elements of the maxheap
{
    int i=0;
    while(i<heapSize)           // while the i < heapSize , print the value of A[i]
```

```
    {
          printf("%d\n",A[i]);
          i++;
    }
    return;
        }

int main (int argc, char **argv)
{
        char line[128];
        char v1[15];
        char v2[15];
        char v3[15];

        int *A = NULL;
        int ret;
        int lineNo = 0;

        while (fgets(line, sizeof line, stdin) != NULL )
 // takes infinite line input from standard input
        {
        sscanf(line, "%s %s %s", v1, v2, v3);        // takes out the three strings v1, v2,
v3 out of the line . If only one string is present , it takes it in v1.
        lineNo++;

        if(lineNo == 1)                // first line gives the maximum size of heap
        {
        A = (int*) malloc(sizeof(int)* stoi(v1));        // allocating memory
        continue;
        }

        if(strcmp(v1,"INS") == 0)
        {
        ret = insertKey(A, stoi(v2));
        if(ret < 0)
                printf("%d\n", -1);
        }
        else if(strcmp(v1,"EXT") == 0)
        {
        ret = extractMax(A);
        printf("%d\n", ret);
        }
```

```
        else if(strcmp(v1,"PRT") == 0)
        {
        print(A);
        }
        else if(strcmp(v1,"INC")==0)
   {
         ret = increaseKey(A, stoi(v2) , stoi(v3));
         if(ret<0)
                printf("%d\n", -1);
   }
        else
        {
        printf("INVALID\n");
        }
        }
        if(A)
        free(A);

        return 0;
}
```

**Time Complexity:**

1. **Swap: O(1) - Takes constant amount of time to swap .**
2. **maxHeapify: O(logn) - If we start from first node at index 0 till to the end at a leaf (case in which A[0] is the minimum among all the elements), we need to go to its height level , height is logn .**
3. **insertKey: O(logn) - If inserted value is maximum , we need to go up till its first node.**
4. **increaseKey: O(logn) - If the increased index is the last one and the increased value is maximum , it will go up till its first node.**
5. **extractMax: O(1) - constant amount of time to access its first element**
6. **Print: O(n) - It will go through each element to print it, hence linear.**

**Space Complexity:**

1. **Swap: O(1) - Makes only one variable temp to swap .**
2. **maxHeapify: O(logn) - Start from index 0 till its base , its variables go only after the function call.**
3. **insertKey: O(1) - Initializes constant number of variables independent of n.**
4. **increaseKey: O(logn) - Initializes constant number of variables independent of n.**

5. **extractMax: O(logn) - Uses maxHeapify.**
6. **Print: O(1) - Initializes constant number of variables independent of n.**

**21.c**

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

//Code to parse integer from a string
int stoi(char *str)                // converts string input str to integer
{
        int x;
        sscanf(str, "%d", &x);
        return x;
}

typedef struct stack                    // defining a data type of type stack
{
   int top, capacity;
   int *s;                              // s contains the array on which we are working
} stack;

stack* make_stack(int capacity)                 // allocation of memory to the stack
{
   stack *stk;                                  // pointer to return
   stk = (stack*)malloc(sizeof(stack));         // giving memory to the pointer
   stk->top = -1;                               // initializing top to -1
   stk->capacity = capacity;    // storing maximum capacity of stack in capacity
variable
stk->s = (int*)malloc(capacity*sizeof(int));
                           // allocating memory to the array to work on
   return stk;
}

int push(stack *stk,int key)            // inserts a new key value into the stack
{
```

```
    if(stk->top==(stk->capacity)-1)               // check if stack is full
            return -1;
    (stk->top)++;
                                             // if no then increase the top and insert the key
    (stk->s)[stk->top]=key;
    return 1;

        }

int pop(stack *stk)              // takes out the element at the top in the stack
{
    int output=-1 ;
    if(stk->top==-1)             // if stack is empty , no item to take out
            return -1;
    output = (stk->s)[stk->top];
                                 // else increase the top and take out the value in output
    (stk->top)--;
    return output;
        }

int top(stack *stk)              // return the top element in the stack
{
    if(stk->top==-1)             // if stack is empty
            return -1;
    return (stk->s)[stk->top];           // else return the value at the top
        }

void print(stack *stk)           // prints the values in the stack
{
    int tpo;
    if(stk->top==-1)             // if stack is empty
            return;
    tpo = stk->top;
    while(tpo>=0)                // while tpo is greater than 0 print the value in the stack
    {
            printf("%d\n", (stk->s)[tpo]);
            tpo--;
    }

}

int size(stack *stk)             // gives the no of elements in the stack
```

```
{
   if(stk->top==-1)              // if stack is empty return 0
          return 0;
   return (stk->top)+1;          // else return top+1
         }

int isEmpty(stack *stk)                  // gives 1 if stack is empty
{
   if(stk->top==-1)              // if stack is empty

          return 1;
   return -1;
}

int isFull(stack *stk)           // gives 1 if stack is full
{
   if(stk->top==(stk->capacity)-1)              // if full
          return 1;
   return -1;
}

int main (int argc, char **argv)
{
         char line[128];
         char v1[15];
         char v2[15];
         char v3[15];

         stack* stk = NULL;
         int ret;
         int lineNo = 0;

         while (fgets(line, sizeof line, stdin) != NULL )
                  // takes line input from standard input
         {
         sscanf(line, "%s %s %s", v1, v2, v3);          // takes 3 strings from input line
         lineNo++;

         if(lineNo == 1)                         // first line gives the capacity of the stack
         {
         stk = make_stack(stoi(v1));
         continue;
```

```
}

if(strcmp(v1,"PSH") == 0)
{
ret =push(stk,stoi(v2));
if(ret < 0)
        printf("%d\n", -1);
}
else if(strcmp(v1,"POP") == 0)
{
ret = pop(stk);
        printf("%d\n", ret);
}
else if(strcmp(v1,"TOP") == 0)
{
ret = top(stk);
printf("%d\n", ret);
}
else if(strcmp(v1,"PRT") == 0)
{
print(stk);
}
else if(strcmp(v1,"SZE") == 0)
{
ret = size(stk);
        printf("%d\n", ret);
}
else if(strcmp(v1,"EMP") == 0)
{
ret = isEmpty(stk);
        printf("%d\n", ret);

}
else if(strcmp(v1,"FUL") == 0)
{
ret = isFull(stk);
        printf("%d\n", ret);
}
else
{
printf("INVALID\n");
}
}
```

```
    if(stk)
    free(stk);

    return 0;
}
```

**Time Complexity:**

1. **Push: O(1) - Access the element at top in constant time.**
2. **Pop: O(1) - Delete the top element in constant time.**
3. **Top: O(1) - gives the top element in constant time.**
4. **Print: O(n) - Access each element in the stack takes linear time.**
5. **Size: O(1) - value is top+1 , output in constant time.**
6. **isEmpty and isFull : O(1) - checks only one condition and returns, takes constant amount of time.**

**Space Complexity:**
**All functions take O(1) complexity to work.**

**22.c**

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

//Code to parse integer from a string
int stoi(char *str)            // converts string input str to integer
{
    int x;
    sscanf(str, "%d", &x);
    return x;
}

typedef struct Queue           // define data type named queue
{
  int front, rear, capacity, current_size;
  int *q;
} queue;
```

```
queue* make_queue(int capacity)
        // gives actual memory to the queue data type and return its pointer
{
   queue* que;                   // pointer to return which stores the queue data
   que = (queue*)malloc(sizeof(queue));      // memory to the pointer
   que->front=0;                             // front stores the index of first filled
element
   que->rear = -1;                           // rear stores the index of last filled element
   que->capacity = capacity;                 // stores the capacity of the queue
   que->current_size = 0;                    // stores the current size of the queue
   que->q = (int*)malloc(capacity*sizeof(int));              // actual array which stores the
values
   return que;
}

int enque(queue *que, int key)         // add one value to the queue
{
   if(que->current_size==que->capacity)              // if queue is full return -1
         return -1;

   (que->current_size)++;
        // else add the value of current size, rear value and insert the key into the
queue. Take mod by capacity to make it a circular queue
   (que->rear)++;
   que->rear = (que->rear)%(que->capacity);
   (que->q)[que->rear]=key;

   return 1;
}

int dequeue(queue *que)                // extract the value from the front of the queue
{
   int output ;
   if(que->current_size==0)            // if queue is empty return -1
         return -1;

   (que->current_size)--;
// else increase the current capacity , front and extract the element at the front
// Take mod by capacity to make it a circular queue
   output = (que->q)[que->front];
   (que->front)++;
```

```
    que->front = (que->front)%(que->capacity);
    return output;
}

int peekFront(queue *que)              // gives the element at the front of queue
{
    if(que->current_size==0)           // if queue is empty return -1
            return -1;
    return (que->q)[que->front];               // else return queue front element
}

void print(queue *que)                         // print the element of the queue
{
    int front=que->front;
    int rear=que->rear;
    int helper=0;
    int cap = que->capacity;
    int steps=0;
    if(que->current_size==0)           // if queue is empty
            return;
    if(front<rear)                     // if it is in linear array
    {
            steps=rear-front+1;        // number of steps to be taken
            while(steps--)
            {
                    printf("%d\n", (que->q)[(que->front)+helper]);        // printing the values
                    helper++;
            }
            return;
    }
    else if(front>rear)                // means it is in circular form now
    {
            steps = cap-(que->front-que->rear)+1;        // number of steps to be taken
            while(steps--)
            {
                    printf("%d\n", (que->q)[((que->front)+helper)%cap]);
          // printing the values and mod by capacity to ensure that it doesn't overflow
                    helper++;
            }
            return;
    }
    Else          // else it has only one element which is being printed here
```

```
        printf("%d\n", que->front);
    return;
}

int size(queue *que)   // gives the current size of the queue stored in current capacity
{
    return que->current_size;
        /* Decrease the value of A[i] to newVal. Return 1 if the
        operation is successful and -1 otherwise. */
}

int isEmpty(queue *que)                // empty if current size is 0
{
    if(que->current_size==0)
        return 1;
    return 0;
        /* Ensure that the subtree rooted at A[i] is a min heap. */
}

int isFull(queue *que)          // is full if current size is capacity
{
    if(que->current_size==que->capacity)
        return 1;
    return 0;
        /* Display the heap represented by A in the increasing order
        of their indices, one element per line.*/
}

int main (int argc, char **argv)
{
        char line[128];
        char v1[15];
        char v2[15];
        char v3[15];

        queue* que;
        int ret;
        int lineNo = 0;

        while (fgets(line, sizeof line, stdin) != NULL )//takes line input from standard input
        {
        sscanf(line, "%s %s %s", v1, v2, v3);
```

```
lineNo++;                              // first line gives the capacity of the queue.

if(lineNo == 1)
{
que = make_queue(stoi(v1)-1);
continue;
}

if(strcmp(v1,"ENQ") == 0)
{
ret = enque(que, stoi(v2));
if(ret < 0)
        printf("%d\n", -1);
}
else if(strcmp(v1,"DEQ") == 0)
{
ret = dequeue(que);
        printf("%d\n", ret);
}
else if(strcmp(v1,"FRN") == 0)
{
ret = peekFront(que);
printf("%d\n", ret);
}
else if(strcmp(v1,"PRT") == 0)
{
print(que);
}
else if(strcmp(v1,"SZE") == 0)
{
ret = size(que);
        printf("%d\n", ret);
}
else if(strcmp(v1,"EMP") == 0)
{
ret = isEmpty(que);
        printf("%d\n", ret);
}
else if(strcmp(v1, "FUL")==0)
{
         ret = isFull(que);
                 printf("%d\n", ret);
}
```

```
	else
	{
	printf("INVALID\n");
	}
	}

	if(que)
	free(que);

	return 0;
}
```

**Time Complexity: All take O(1) time except the print which takes O(n) time, time to access each element and printing it.**

**Space Complexity: All functions take O(1) complexity to work.**