Week_3

11.c

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

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

int find_index(int *A, int n)
        // finds the index of first element which is greater than the A[n-1] element
{
   int i=0;
   for(i=n-2; i>=0; i--)
                // i starts from the end i.e. i-2 as input array is in decreasing order
   {
         //printf("%d", i);
                 if(A[i]>A[n-1])
                // when it finds that element is greater than A[n-1], it returns
                        return i;
         }
         return i;                  // i returns the index at which our A[n-1] should be
placed
}

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

        int *A, n, temp, ind=0 ;
        fgets(line, sizeof line, stdin);          // scans from standard input
        sscanf(line, "%s", v1);                   // v1 from the input line
```

```
        n=stoi(v1);                         //converts v1 to integer
        //scanf("%d", &n);
        A = (int*)malloc(n*sizeof(int));    // allocating size to array A
        for(int i=0;i<n;i++)
        {
                                    // scanning the array values and putting into the array
   fgets(line, sizeof line, stdin);
   sscanf(line, "%s", v1);
   A[i]= stoi(v1);
        }
 //  for(int i=0; i<n; i++)
        //    scanf("%d", A+i);

        ind = find_index(A, n);         // finding index to insert at
        //printf("index is:%d", ind);
        for(int i=n-1; i>ind+1; i--)
                                    // switching the values uptill index+1 starting from n-1
        {
         temp = A[i];
         A[i]=A[i-1];
         A[i-1]=temp;
        }

        for(int i=0; i<n; i++)          // printing the sorted array
         printf("%d\n", *(A+i));

        free(A);                    // free up the memory allocated to A

        return 0;
}
```

Time complexity: O(n)

Finds time to find index , same time for swapping the whole array.

Space complexity: O(n)

For initializing array size increases with the size of n.

12.c

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

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

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

void merge(int* ar1, int* ar2, int len1, int len2)
// takes two sorted arrays (pointer) as input and merge them and outputs one
sorted array.
{
   int i=0, j=0, k=0;
   int a[len1+len2];`                  // making of array of size len1+len2
   while(i<len1 && j<len2)                  // loop until any one of the array is not empty
   {
                if(ar1[i]<=ar2[j])
                                       // checking if array 2 has bigger element than array 1
                {
                                        a[k] = ar1[i];
                                // if yes it stores the value of array 2 in the initialized array a
                                        i++;    k++;
                }
                else
                {                               // else stores the value of array 1 in the array a
                                        a[k] = ar2[j];
                                        j++;     k++;
                }
   }

   while(i<len1)          // if array 2 got empty first , it will go untill array 1 is also empty
   {
                a[k] = ar1[i];                  // stores the value of array 1 in the array a
                i++;     k++;
   }

   while(j<len2)          // if array 1 got empty first , it will go untill array 2 is also empty

   {
                a[k]= ar2[j];     // stores the value of array 2 in the array a
```

```
                j++;    k++;
    }
    // only one of the two while loops will run, as at least one of the arrays has to be empty after the first while

    for(i=0; i<len1; i++)
                        // storing the sorted value of array a into array 1 and array 2
        ar1[i]=a[i];
    for(i=0; i<len2; i++)
        ar2[i]=a[i+len1];

}

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

    int *arr1, *arr2, n1, n2;

    fgets(line, sizeof line, stdin);          // scans from standard input
    sscanf(line, "%s", v1);                   // v1 from the input line
    n1=stoi(v1);                              //converts v1 to integer
    arr1 = (int *)malloc(n1*sizeof(int));     // allocating memory to the array 1

        for(int i=0;i<n1;i++)                 // takes the value into the array 1
        {
        fgets(line, sizeof line, stdin);
      sscanf(line, "%s", v1);
        arr1[i]= stoi(v1);
        }

        fgets(line, sizeof line, stdin);
        sscanf(line, "%s", v1);
        n2=stoi(v1);                          //scanf("%d", &n2);
    arr2 = (int*)malloc(n2*sizeof(int));      // allocating memory to the array 2
    for(int i=0;i<n2;i++)
    {                                         // takes the value into the array 2
        fgets(line, sizeof line, stdin);
      sscanf(line, "%s", v1);
```

```
        arr2[i]= stoi(v1);
    }
// two input  given arrays are already in sorted manner

    merge(arr1, arr2, n1, n2);          // calling the merge function

    for(int i=0; i<n1; i++)             // printing the two merged arrays
        printf("%d\n", arr1[i]);
    for(int i=0; i<n2; i++)
        printf("%d\n", arr2[i]);

    free(arr1);          // free up the allocated memory of arrays
    free(arr2);
    return 0;

}
```

Time Complexity: O(n)

Function merge takes O(n) time to execute. Rest the two loops in the main function also takes O(n) time to run.

Space Complexity: O(n)

Function merge takes O(n) space . It initializes array (a) of length n (len1+len2). As lengths of individual arrays increases it also increases linearly.

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 to integer value
{
        int x;
        sscanf(str, "%d", &x);
        return x;
}
```

```
int heapSize=0;

void swap(int *x, int *y)
                        // takes two integer pointers and swap the value at their
location
{
   int temp;
   temp = *x;
   *x = *y;
   *y = temp;
        /* Fill in */
}

void minHeapify(int A[],int i)
// takes an index i of an array and form it a node of a min heap. It goes from index
i to the point where both of its children are greater than the parent or upto a leaf
i.e. no children.
{
   int smt = i;
   if(A[i]>A[2*i+1] && 2*i+1<heapSize)
                        // checking the smaller value and storing the index int smt
        smt = 2*i+1;

   if(A[2*i+2]<A[smt] && 2*i+2<heapSize)
   {                    // if child A[2*i+2] is the smallest one swap A[i] and A[2*i+2]
        smt = 2*i+2;
        swap(&A[i], &A[smt]);
        minHeapify(A, smt);
   }
   else if(smt!=i)
                // else if child A[2*i+1] is the smallest one swap A[i] and A[2*i+1]
   {
        swap(&A[i], &A[2*i+1]);
        minHeapify(A, 2*i+1);
   }
   else
                        // else A[i] is minimum and nothing to be done, therefore return
        return;

        /* Ensure that the subtree rooted at A[i] is a min heap. */
}
```

```
int insertKey(int A[], int key)
{                   // insert a key in the heap
   int flag=0;
   A[heapSize]=key;            // insert a key at the last index
   int i = heapSize;
   heapSize++;                 // increasing the heap size
   while(i>0)
   {
                        // now checking that if the inserted key is smaller its parent
        if(A[i]<A[(i-1)/2])              // if yes then swap
        {
                swap(&A[i], &A[(i-1)/2]);
                i= (i-1)/2;
        }
        else
                        // if no then min heap is formed as key is greater than its
                parent
                return 1;
        flag=1;
   }

   if(flag==1 || i==0)
        return 1;
   return -1;
        /* Insert the element key into the heap represented by A.
        Return 1 if the operation is successful and -1 otherwise. */
}

int increaseKey(int A[], int i, int newVal)
{                       // increase the key at index i
   if(i>=heapSize)
                // if the index is greater than heapSize than operation is wrong
        return -1;
   A[i]=newVal;         // else increase the value at index i
   minHeapify(A, i);
                // if the new value is greater than it's child then we need to swap , as
now heap property may be lost . Therefore min heapify is called. (As the nodes at
the children are already a min heap)
   return;
}
```

```
int decreaseKey(int A[], int i, int newVal)
{               // decrease the value at index i
   if(i>=heapSize)
               // if the given index i is greater than heapSize , invalid operation
         return -1;
   A[i]=newVal;                  // else decrementing the value
   while(i>0)
   {                     // now decremented value may be smaller than it's parent
                         // so we need to swap the untill heap property is satisfied
         if(A[i]<A[(i-1)/2])
         {
               swap(&A[i], &A[(i-1)/2]);
               i= (i-1)/2;
         }
         else
               return 1;
         flag=1;
   }
   return 1;
        /* Decrease the value of A[i] to newVal. Return 1 if the
        operation is successful and -1 otherwise. */
}

int deleteKey(int A[], int i)      // delete the value at index i
{
   if(heapSize==0 || i>heapSize)
// if heapSize if 0 , nothing to delete or if index i is greater than heapSize , index is
out of bounds
         return -1;
   A[i]=A[heapSize-1];         // deleting the value at index i
   heapSize--;                 // decrementing the heapSize
   minHeapify(A, i);
// now A[i] can be greater than it's children , so min heap property is lost , so
minHeapify to restore the heap property at index i
   return 1;
        /* Delete the element A[i] from the heap represented by A.
        Return 1 if the operation is successful and -1 otherwise. */
}

int extractMin(int A[])
{               // same as delete key , here key is at last index
```

```
    int val;
    if(heapSize==0)
            return -1;
    val=A[0];
    A[0]=A[heapSize-1];
    heapSize--;
    minHeapify(A,0);
    return val;             // returning the value of min
            /* Delete the root of the min heap represented by A. Return
            the deleted element if the operation is successful and -1
            otherwise. */
}

int getMin(int A[])
{                   // return the min of the min heap
    if(heapSize==0)             // check if heap is empty
            return -1;
    return A[0];
            /* Get the root of the min heap represented by A. Return
            the element if the operation is successful and -1 otherwise. */
}

void print(int A[])
{                   // printing the values of min heap
    int i=0;
    while(i<heapSize)           // until heap size is greater than the index i
    {
            printf("%d\n",A[i]);
            i++;
    }
            /* 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];

            int *A = NULL;
```

```
        int ret;
        int lineNo = 0;

        while (fgets(line, sizeof line, stdin) != NULL )          // standard input in a line
        {
        sscanf(line, "%s %s %s", v1, v2, v3);          // taking out v1 , v2, v3 from line
        lineNo++;

        if(lineNo == 1)
        {
        A = (int*) malloc(sizeof(int)* stoi(v1));
                                                    // allocating maximum memory to the array
        continue;
        }

        if(strcmp(v1,"INS") == 0)
        {
        ret = insertKey(A, stoi(v2));
        if(ret < 0)
                printf("%d\n", -1);
        }
        else if(strcmp(v1,"DEL") == 0)
        {
        ret = deleteKey(A, stoi(v2));
        if(ret < 0)
                printf("%d\n", -1);
        }
        else if(strcmp(v1,"EXT") == 0)
        {
        ret = extractMin(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 if(strcmp(v1,"DEC") == 0)
        {
        ret = decreaseKey(A, stoi(v2), stoi(v3));
        if(ret < 0)
                printf("%d\n", -1);
        }
        else if(strcmp(v1,"MIN") == 0)
        {
        ret = getMin(A);
        if(ret < 0)
                printf("%d\n", -1);
        }
        else
        {
        printf("INVALID\n");
        }
        }
        if(A)           // if allocated free up the memory of A
        free(A);
        return 0;
}
```

Time Complexity:

1. **minHeapify - O(logn) : If we want to heapify top node, it will max go up to a leaf and travel to it's height, which is logn .**
2. **Insert - O(logn): If the inserted value is the smallest (smaller than the first element), then it will travel up to its height which is logn.**
3. **increaseKey - O(logn): if first node value is increased such that it is the biggest among all the nodes , then it will travel up to its base which is logn farther away.**
4. **decreaseKey - O(logn): Same asd insert . If we decrease a leaf to min of the heap so it will travel to its height.**
5. **deleteKey - O(logn): Delete the first node and travel to its base.**
6. **extractMIn - O(logn): same as deleting the first node.**
7. **getMin- O(1): first node contains the minimum value, we can access it in constant time.**
8. **Print- O(n): accessing all the elements one at a time takes linear time.**

Space Complexity:

1. **minHeapify - O(logn): uses recursion logn times , each time constant memory allocation.**
2. **Insert - O(1): uses constant amount of memory.**
3. **increaseKey - O(logn): uses minheapify.**
4. **decreaseKey- O(1): uses constant amount of memory.**
5. **deleteKey- O(logn): uses minheapify.**
6. **extractMin- O(long): uses minheapify.**
7. **getMin- O(1): first index , nothing to initialize**
8. **Print- O(1): no extra allocation based on input n.**