INDIAN INSTITUTE OF TECHNOLOGY MADRAS
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Course Name: ***Programming and Data Structure Lab*** **Course No:** ***CS2710***
Full Marks: 15 (LabTest-2) Date: 31 / 10 / 2016 Time: 3 *hours*

Instructions:

You are supplied with two files: `ROLLNO_labtest2_prob1.c` and `ROLLNO_labtest2_prob2.c`. First, replace these file-names with your Roll-Numbers where the `ROLLNO` keyword appears in the file-name. You can ***only modify*** the mentioned areas (where it is denoted as "`// Write C-code Here`") in these two `.c` files. You are ***not allowed*** to modify/change any other portion of the given code. Please make sure that your programs compile and run successfully. Finally, submit only two `.c` files (i.e., `ROLLNO_labtest2_prob1.c` and `ROLLNO_labtest2_prob2.c`) in the mentioned submission links given in MOODLE.

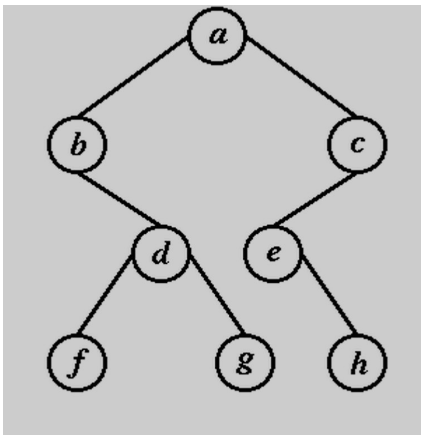
Problem-1: [5 *Marks*]

Consider a binary tree with each node carrying a (unique) label. In-order and pre-order listings of these labels are defined by the following recursive procedures:

```
inorder ( node )
{
     if (node != NULL) {
          inorder(left child of node);
          print(label of node);
          inorder(right child of node);
     }
}

preorder ( node )
{
     if (node != NULL) {
          print(label of node);
          preorder(left child of node);
          preorder(right child of node);
     }
}
```

A binary tree and the in-order and pre-order listings of the labels of its nodes are shown in the following:



In-order listing : *bfdgaehc*
Pre-order listing : *abdfgceh*

Given the in-order and pre-order listings for a binary tree, the tree can be uniquely reconstructed using the following *recursive* function:

```
node *gentree ( char *inlist , char *prelist )
```

Implement this *recursive* construction procedure, named as `gentree`. Read the `inlist` and `prelist` from the user, construct the tree and print it (`printtree` routine is already supplied).

Sample Output:

```
Inorder listing  : bfdgaehc
Preorder listing : abdfgceh
Node : a, Left child :    b, Right child :    c.
Node : b, Left child : NULL, Right child :    d.
Node : d, Left child :    f, Right child :    g.
Node : f, Left child : NULL, Right child : NULL.
Node : g, Left child : NULL, Right child : NULL.
Node : c, Left child :    e, Right child : NULL.
Node : e, Left child : NULL, Right child :    h.
Node : h, Left child : NULL, Right child : NULL.
```

Test input:

Show the output of your program on the following inputs:

```
Inorder listing  : GDHBAECF
Preorder listing : ABDGHCEF

Inorder listing  : abcdefghijkl
Preorder listing : abcdefghijkl

Inorder listing  : abcdefghijkl
Preorder listing : lkjihgfedcba

Inorder listing  : abcdefghijkl
Preorder listing : badcfehgjilk

Inorder listing  : cbafedihglkj
Preorder listing : abcdefghijkl

Inorder listing  : abcdefghijkl
Preorder listing : bcdefghijkla

Inorder listing  : bcdefghijkla
Preorder listing : abcdefghijkl

Inorder listing  : hdibjekalfmcngo
Preorder listing : abdhiejkcflmgno

Inorder listing  : ABCDEFGHIJKLMN
Preorder listing : HGFEDCBAIJKLMN

Inorder listing  : ABCDEFGHIJKLMN
Preorder listing : GABCDEFNMLKJIH
```

Problem-2: [10 *Marks*]

A max-heap (or a max-priority queue, to be more precise) supports constant-time max finding, and logarithmic-time insertion and deleteMax. A min-heap, on the other hand, supports constant-time min finding, and logarithmic-time insertion and deleteMin. In this exercise, you build a heap (let us call it a **med-heap**) that should support the following three operations:

- Constant-time median finding
- Logarithmic-time insertion
- Logarithmic-time deletion of the median

Using a med-heap, you are supposed to write a sorting function analogous to heap sort.

Let N be the current number of elements in a collection (a med-heap). We make the following convention. If $N = 2k + 1$ (that is, odd), then the median is the $(k + 1)$-st smallest element in the collection. If $N = 2k$ (even), then we call the k-th smallest element the median of the collection.

A med-heap is realized by maintaining two heaps: a max-heap consisting of the smaller half of the values, and a min-heap consisting of the larger half of the values. The following figure demonstrates how a single array can be used for the contiguous representation of both the heaps back to back.

Max-heap of smaller values	Empty space	Min-heap of larger values
Direction of growth →	(for future insertions)	←Direction of growth

Let N_1 be the number of elements stored in the max-heap, and N_2 the number of elements stored in the min-heap. We have $N_1 + N_2 = N$. We maintain the convention $N_1 = \lfloor N / 2 \rfloor$ and $N_2 = \lceil N / 2 \rceil$. Notice that the max-heap grows from the left, that is, the maximum in this heap can be found at the zeroth index of the array. On the other hand, the min-heap grows from the right end, that is, the minimum in this heap can be found at the $(n - 1)$-st index of the array, where n is the total capacity of the array (including the empty space).

We provide you with the `insMaxHeap` and `delMaxHeap` functions for insertion and deletion in the max-heap, and the `insMinHeap` and `delMinHeap` functions for insertion and deletion in the min-heap.

Now, ***complete*** the following C-functions.

- ***Write*** the `initMedHeap` function to initialize an empty med-heap.
- ***Write*** the `getMedian` function for finding the Median in the med-heap.
  Distinguish between the two cases $N_2 = N_1$ and $N_2 = N_1 + 1$.
- ***Write*** the `insMedHeap` function for inserting elements in the med-heap.
  Suppose that you want to insert x in a non-full med-heap. Let m be the current median in the med-heap. If $x \le m$, then you should insert x in the max-heap. But, then, if we already have $N_1 = N_2$, then this insertion will violate the above conventions regarding the new sizes of the two heaps. You therefore shift an element (which one is it?) from the max-heap to the min-heap, and then insert x in the max-heap. On the other hand, if $x > m$, then insert x in the min-heap after a size-adjusting transfer, if necessary.
- ***Write*** the `delMedHeap` function for deleting element in the med-heap.
  This boils down to a deletion in the max-heap or in the min-heap depending upon whether $N_2 = N_1$ or $N_2 = N_1 + 1$.
- ***Write*** the `medHeapSort` function (for sorting) that starts with a med-heap filled to the capacity (that is, with no empty space between the max-heap and the min-heap).
  The function iteratively deletes the current median. Each deletion creates an empty cell. The deleted median is copied there. After all of the n elements are deleted, the array should store the sorted list.

You are already provided with the `main` function which first reads n (the capacity of the array) from the user. It first initializes the array to an empty med-heap. Subsequently, it inserts n random elements in the med-heap. After each insertion, it prints the current median. After all the elements are inserted, the array is filled to the capacity. Now, the `medHeapSort` function is called. When it returns, the (sorted) array is to be printed (using already provided `printArray` function).

Sample Output:

```
Enter capacity (n) of the array: 20

+++ MedHeap initialized

+++ Going to insert elements one by one in MedHeap
    Insert(3064) done. Current median = 3064.
    Insert( 545) done. Current median =  545.
    Insert(2978) done. Current median = 2978.
    Insert(5176) done. Current median = 2978.
    Insert(7432) done. Current median = 3064.
    Insert(2687) done. Current median = 2978.
    Insert(9903) done. Current median = 3064.
    Insert(7991) done. Current median = 3064.
    Insert(7963) done. Current median = 5176.
    Insert(6184) done. Current median = 5176.
    Insert(5426) done. Current median = 5426.
    Insert(9981) done. Current median = 5426.
    Insert(8838) done. Current median = 6184.
    Insert(8053) done. Current median = 6184.
    Insert(1069) done. Current median = 6184.
    Insert(2950) done. Current median = 5426.
    Insert(3625) done. Current median = 5426.
    Insert(9130) done. Current median = 5426.
    Insert(8458) done. Current median = 6184.
    Insert(9070) done. Current median = 6184.

+++ Median Heap Sort done
      545 1069 2687 2950 2978 3064 3625 5176 5426 6184 7432 7963 7991 8053 8458
     8838 9070 9130 9903 9981
```

Remark: If our ultimate goal is the `medHeapSort` function, then we could have done better than inserting *n* elements one by one in the med-heap, which takes O(*n* log *n*) time. What we instead do is just starting with an array fully populated with elements. We then convert the array to a med-heap using a linear-time `makeMedHeap` function. Such a function can, for example, be implemented if we can *quickly* locate the median in the initial array. A linear-time median-finding algorithm does exist, and will be covered later in the theory course. We then partition the array using the median as pivot as in the quick sort algorithm. Finally, we call `makeMaxHeap` to convert the smaller half to a max-heap, and `makeMinHeap` to convert the larger half to a min-heap. The subsequent sequence of *n* deletions would anyway take O(*n* log *n*) time. But it is interesting to know that a linear-time `makeHeap` algorithm can be designed for med-heaps too.