CS29003 ALGORITHMS LABORATORY
LABORATORY TEST (ODD PC)
Last Date of Submission: 14–Oct–2015

A ternary tree is a rooted tree in which each node can have three children (left, middle, and right). A ternary min-heap is an (almost) complete ternary tree in which the value stored at any node is no larger than the values stored at its child nodes. Assume that each node stores a positive integer value. Store your ternary min-heap in a static array H of integers. Maintain its size n separately.

```
int H[100], n;
```

Part 1 Write the ternary min-heap functions with the following prototypes:

```
void buildheap (int H[], int n);       /* Convert array H[] to a ternary min-heap */
void insert (int H[], int n, int x);   /* Insert x to a heap H[] of size n */
int deletemin (int H[], int n);        /* The return value is the deleted minimum */
```

Part 2 Write a function with the following prototype to print the k smallest elements stored in a ternary min-heap H:

```
void ksmallest (int H[], int n, int k);
```

The function should print the elements in sorted (increasing) order, and have a running time of $O(k \log n)$, where n is the number of elements stored in H. A heap is allowed store the same value at multiple nodes. Your function should print only k values including repetitions (see sample output where 23 is printed twice). After the function returns, the heap should continue to contain all of its n elements.

Part 3 Write a function with the following prototype to print all elements $\leq a$ in a ternary min-heap H:

```
void printsmall (int H[], int n, int a);
```

Printing need not be in a sorted order. If some element $\leq a$ appears multiple times in H, print it as many times as it occurs (not necessarily contiguously, see the printing of 23 in the sample output). Your function must run in $O(t)$ time, where t is the number of elements printed. The heap should continue to contain all of its n elements after the function returns.

main()
- Read the size n of the heap from the user.
- Read n positive integers from the user, and store them in an array H[].
- Run *buildheap*() to convert the array to a ternary min-heap. Print the array H[].
- Read k from the user. Call the function of Part 2 to print the k smallest elements in your heap. Print H[].
- Read a from the user. Invoke the function of Part 3 to print all heap elements $\leq a$. Print H[].

Submit a single C/C++ file. Do not use any global/static variable/array. If needed, you may use additional parameters in the functions mentioned above. **Follow the input sequence exactly as described above under main().**

Sample Output

```
n = 20

+++ Reading elements...
     95  53  32  35  84  68  73  23  94  70  54  46  86  23  48  24  38  18   6  78

--- Heap after buildheap():
      6  18  23  35  23  38  73  32  94  70  54  46  86  84  48  24  95  53  68  78

k = 8

+++ Finding the 8 smallest elements in heap
      6  18  23  23  24  32  35  38
--- Heap after ksmallest():
      6  18  53  46  23  24  38  78  94  70  86  68  54  84  48  23  95  32  35  73

a = 50

+++ Printing elements in heap <= 50
      6  18  23  48  23  24  32  35  38  46
--- Heap after printsmall():
      6  18  53  46  23  24  38  78  94  70  86  68  54  84  48  23  95  32  35  73
```

CS29003 ALGORITHMS LABORATORY
LABORATORY TEST (EVEN PC)
Last Date of Submission: 14–Oct–2015

A ternary search tree is a rooted tree in which every node stores two keys and three child pointers (left, middle, and right). For simplicity, assume that the keys are positive integers. Let u, v be the two values srored at a node, and r, s, t respectively be any values stored in the left, middle, and right subtrees of the node. We must have $r < u < s < v < t$. Only a leaf node (that is, a node with all three child pointers NULL) is allowed to store only one value u or v. The other value is called *undefined*. An undefined value can be represented by a negative integer, like –1, for a ternary search tree storing only positive key values. For each internal (that is, non-leaf) node, both the values u and v must be defined. Use the following data type to store a node in a ternary search tree.

```
typedef struct _node {
   int u, v;
   struct _node *L, *M, *R;
} node;
```

Part 1 Write tree functions with the following prototypes:

```
int search (node *T, int x);     /* Search for key x in a ternary search tree T */
node *insert (node *T, int x);   /* Insert key x in a ternary search tree T */
```

The key x is assumed to be positive in both these functions. Both search and insertion are analogous to those in binary search trees. A search tree is not allowed to store duplicate (repeated) keys. If you want to insert a key x which is already present in T, the original tree T should be returned. Otherwise, the insertion attempt eventually reaches a node p, and encounters a NULL child pointer of p. If both the values u and v are defined at p, create a new child node of p to store x. If only one of u and v is defined at p (in this case, p must be a leaf), store x in p itself (so after the insertion, both the values are defined at p). Your function must run in O(h) time, where h is the height of T.

Part 2 Write a function with the following prototype to print the largest and the second largest keys stored in T:

```
void print2largest (node *T);
```

Your function should run in O(h) time, where h is the height of T.

Part 3 Write a function with the following prototype to print all keys $\leq a$, stored in T:

```
void printsmall (node *T, int a);
```

The keys should be printed in the sorted (increasing) order. Your function should run in O($h + t$) time, where h is the height of T, and t is the number of keys to be printed.

main()
- Read *nins* (number of insertions) from the user.
- Insert *nins* positive integers supplied by the user in an initially empty ternary search tree T.
- Read two positive integer keys x and y from the user, and print the results of searching x and y in T.
- Invoke the function of Part 2 to print the largest and the second largest keys stored in T.
- Read a from the user, and call the function of Part 3 to print all keys $\leq a$, stored in T.

Submit a single C/C++ file. Do not use any global/static variable/array. If needed, you may use additional parameters in the functions mentioned above. **Follow the input sequence exactly as described above under main().**

Sample Output

```
nins = 20

+++ Inserting elements in tree
     58  77  46  81  47  16  93  27  84  46  57  59  72  69  54  32  17  39  57  85

+++ x = 56
    Search(56): FAILURE
+++ y = 57
    Search(57): SUCCESS

+++ The largest element in tree is 93
+++ The second largest element in tree is 85

a = 50

+++ Values in tree <= 50 are
     16  17  27  32  39  46  47
```