In this assignment you are going to construct and manipulate with Binary Search Tree (BST). BST is a rooted binary tree data structure. The nodes in the tree store keys and any internal node in BST has the following properties:

- The left subtree of a node contains nodes with keys less than or equal to the node's key
- The right subtree of a node contains nodes with keys greater than the node's key
- The left and right subtrees of a node are also BSTs

The construction of a BST is based on the rules above. For example, consider that we are given the following input [31,65,3,10,5,100,3,12]. The first element will be the root of the tree and the rest will be placed in correct positions within the tree based on the rules given above. The final BST constructed for this example is given in Figure 1.

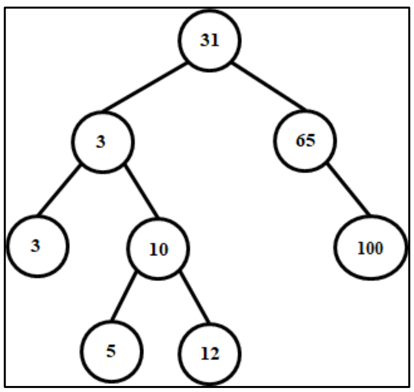

Figure 1

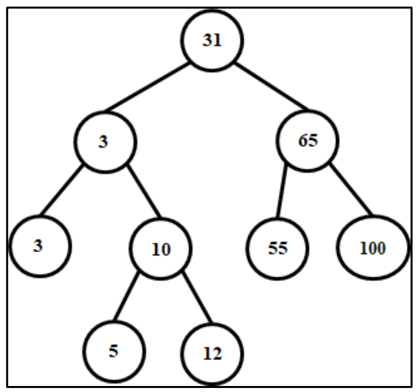

Figure 2

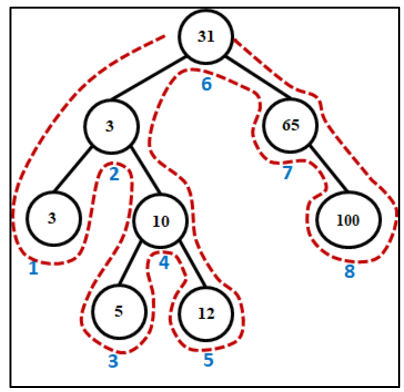

Figure 3

Implementation

The implementation language is C. Trees are dynamic data structures so your implementation will heavily rely on pointers. Also, for any operation to be done on the tree, it is easy if you implement it using recursion.

Input/Output Specifications

Your program should accept the set of commands from standard input.

1) CONSTRUCT [value1,value2,value3,…,valueN]

The CONSTRUCT command expects a set of integers in brackets separated by commas. The command takes the input and constructs a binary tree rooted at the first element of the input.
For example, for the input given below, your program should construct a tree as shown in Figure 1.
CONSTRUCT [31,65,3,10,5,100,3,12]

2) **INSERT value**

The INSERT command expects a single integer. It creates a new node and places the node in the correct position in the tree. For example, INSERT 55, places the value 55 in the correct position as shown in Figure 2 and outputs:
The parent of 55 is 65

3) **LIST**

The LIST command performs inorder traversal on the tree (see Figure 3) and prints the keys of every visiting node. Observe that inorder traversal of the BST prints sorted list in ascending order. For example, LIST command that is given after previous two example commands, the output will be:
3 3 5 10 12 31 55 65 100

4) **PARENT value**

The PARENT command takes the input value, searches it in the tree and prints its parent. For example:

PARENT 12
The parent of 12 is 10
PARENT 31
It is a root node

Note: You can assume that there are no errors in the input. Hence, you do not need to explicitly check for correctness of the provided input.

5) **EXIT**

Whenever the EXIT command is entered, the program must be terminated immediately.

6) **DELETE value**

The implementation of DELETE command is optional and comes as a bonus question.
The DELETE command expects one integer value. Its purpose is to find the given value in the tree and delete it. The state of the tree after deletion should still obey the rules specified above. Also observe that the root of the tree might change after delete operation. You should maintain the order of the tree and reconnect the parent/child connections. Figure 4 shows three different scenarios of delete operation: a node has no children, has single child and has two children.

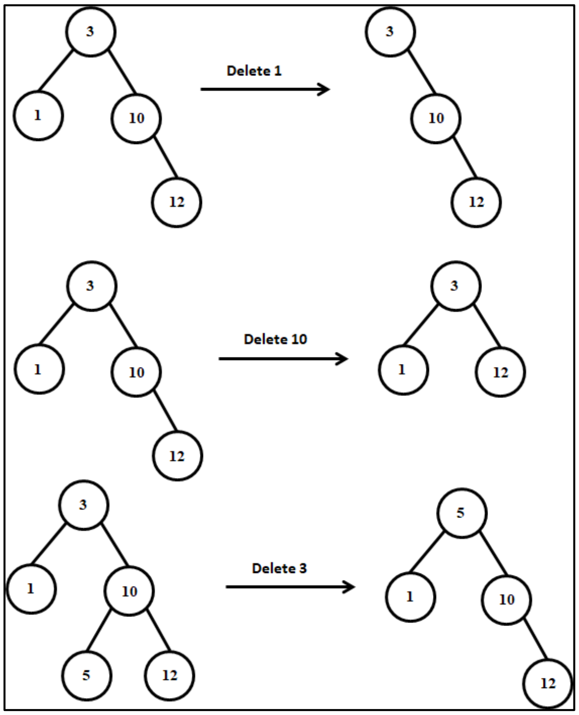

Figure 4

Sample Run 1:

```
CONSTRUCT [31,65,3,10,5,100,3,12]
INSERT 55
The parent of 55 is 65
LIST
3 3 5 10 12 31 55 65 100
PARENT 12
The parent of 12 is 10
PARENT 31
It is a root node
INSERT 1
The parent of 1 is 3
LIST
1 3 3 5 10 12 31 55 65 100
INSERT 500
The parent of 500 is 100
LIST
1 3 3 5 10 12 31 55 65 100 500
PARENT 5
The parent of 5 is 10
EXIT

Process returned 0 (0x0)   execution time : 2.679 s
Press any key to continue.
```

Sample Run 2:

```
CONSTRUCT [47,40,45,98,76,82,64]
LIST
40 45 47 64 76 82 98
INSERT 1
The parent of 1 is 40
INSERT 99
The parent of 99 is 98
LIST
1 40 45 47 64 76 82 98 99
PARENT 98
The parent of 98 is 47
PARENT 40
The parent of 40 is 47
LIST
1 40 45 47 64 76 82 98 99
DELETE 45
LIST
1 40 47 64 76 82 98 99
EXIT

Process returned 0 (0x0)   execution time : 10.403 s
Press any key to continue.
```