Lecture 11.1

Topics:

1. Recursive Insertion – **`insertRecur()`**
2. Binary Trees – **`removeNode()`**

---

1. Recursive Insertion – `insertRecur()`

Again, consider a recursive version of the insertion given below.

Example 1

```
/**
 * Program Name: cis27L1111.c
 * Discussion:   Binary Search Tree -- Recursive Stuff
 */
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

typedef struct TreeInt* TreePtr;
typedef struct TreeInt* IntPtr;

struct TreeInt {
  int iValue;
  IntPtr left;
  IntPtr right;
};

void create(TreePtr newTree);
void insert(int aValue, TreePtr* myTree);
int find(int aValue, TreePtr myTree);

void displayInOrder(TreePtr myTree);
void displayPreOrder(TreePtr myTree);
void displayPostOrder(TreePtr myTree);

void insertRecur(int aValue, TreePtr* myTree);
void insertRecurHelper(TreePtr* myTree, IntPtr newNode);

int main( ) {
  int ch;
  TreePtr workingTree = NULL;
  int iTest;

  printf("\n\nPreOrder Display:\n");  /* An empty tree */
  displayPreOrder(workingTree);

  printf("\n\nInOrder Display:\n");
  displayInOrder(workingTree);

  printf("\n\nPostOrder Display:\n");
  displayPostOrder(workingTree);

  insert(5, &workingTree);
  insert(15, &workingTree);
  insert(55, &workingTree);
```

```
  insert(25, &workingTree);
  insert(75, &workingTree);
  insert(65, &workingTree);
  insert(10, &workingTree);
  insert(-10, &workingTree);

  printf("\n\nPreOrder Display:\n");  /* Displaying the
                                        Binary Search Tree */
  displayPreOrder(workingTree);

  printf("\n\nInOrder Display:\n");
  displayInOrder(workingTree);

  printf("\n\nPostOrder Display:\n");
  displayPostOrder(workingTree);

  iTest = 25;

  if (find(iTest, workingTree)) {  /* Searching (finding)
                                      a node */
    printf("\n\nThe integer %d is found!\n", iTest);
  } else {
    printf("\n\nThe integer %d is not found!\n", iTest);
  }

  printf("\n\nEnter an integer + ENTER to continue ...\n");
  scanf("%d", &ch);

  return 0;
}

void insert(int aValue, TreePtr* myTree) {
  IntPtr newNode, nodePtr;

  newNode = (IntPtr) malloc(sizeof(struct TreeInt));
  assert(newNode);

  newNode->iValue = aValue;
  newNode->left = newNode->right = NULL;

  if (*myTree == NULL) {
    *myTree = newNode;
  } else {
    nodePtr = *myTree;
    while (nodePtr != NULL) {
      if (aValue < nodePtr->iValue) {
        if (nodePtr->left) {
          nodePtr = nodePtr->left;
        } else {
          nodePtr->left = newNode;
          break;
        }
      } else if (aValue > nodePtr->iValue) {
        if (nodePtr->right) {
          nodePtr = nodePtr->right;
        } else {
          nodePtr->right = newNode;
          break;
        }
      } else {
```

```
        printf("\nDuplicate value found!\n");
        break;
      }
    }
  }
  return;
}

int find(int aValue, TreePtr myTree) {
  TreePtr nodePtr = myTree;

  while (nodePtr) {
    if (nodePtr->iValue == aValue) {
      return 1;
    } else if (aValue < nodePtr->iValue) {
      nodePtr = nodePtr->left;
    } else {
      nodePtr = nodePtr->right;
    }
  }

  return 0;
}

void insertRecur(int aValue, TreePtr* myTree) {
  IntPtr newNode1;

  newNode1 = (IntPtr) malloc(sizeof(struct TreeInt));
  assert( newNode1 );

  newNode1->iValue = aValue;
  newNode1->left = newNode1->right = NULL;

  insertRecurHelper(myTree, newNode1);
}

void insertRecurHelper(TreePtr* myTree, IntPtr newNode) {
  if (*myTree == NULL) {
    *myTree = newNode;
  } else if (newNode->iValue < (*myTree)->iValue) {
    insertRecurHelper(&((*myTree)->left), newNode);
  } else {
    insertRecurHelper(&((*myTree)->right), newNode);
  }
}

void displayPreOrder(TreePtr myTree) {
  if (myTree) {
    printf("\n\tValue of node : %d", myTree->iValue);
    displayPreOrder(myTree->left);
    displayPreOrder(myTree->right);
  }

  return;
}

void displayInOrder(TreePtr myTree) {
  if (myTree) {
    displayInOrder(myTree->left);
    printf("\n\tValue of node : %d", myTree->iValue);
```

```
    displayInOrder(myTree->right);
  }

  return;
}

void displayPostOrder(TreePtr myTree) {
  if (myTree) {
    displayPostOrder(myTree->left);
    displayPostOrder(myTree->right);
    printf("\n\tValue of node : %d", myTree->iValue);
  }

  return;
}

/*OUTPUT


PreOrder Display:


InOrder Display:


PostOrder Display:


PreOrder Display:

        Value of node : 5
        Value of node : -10
        Value of node : 15
        Value of node : 10
        Value of node : 55
        Value of node : 25
        Value of node : 75
        Value of node : 65

InOrder Display:

        Value of node : -10
        Value of node : 5
        Value of node : 10
        Value of node : 15
        Value of node : 25
        Value of node : 55
        Value of node : 65
        Value of node : 75

PostOrder Display:

        Value of node : -10
        Value of node : 10
        Value of node : 25
        Value of node : 65
        Value of node : 75
        Value of node : 55
        Value of node : 15
        Value of node : 5
```

```
The integer 25 is found!


Enter an integer + ENTER to continue ...
1
*/
```

Let's compile, run, and look at the output of the above code in debugging mode.

2. Binary Trees – `removeNode()`

Recall the integer tree from previous lecture given in **Figure 1** below.

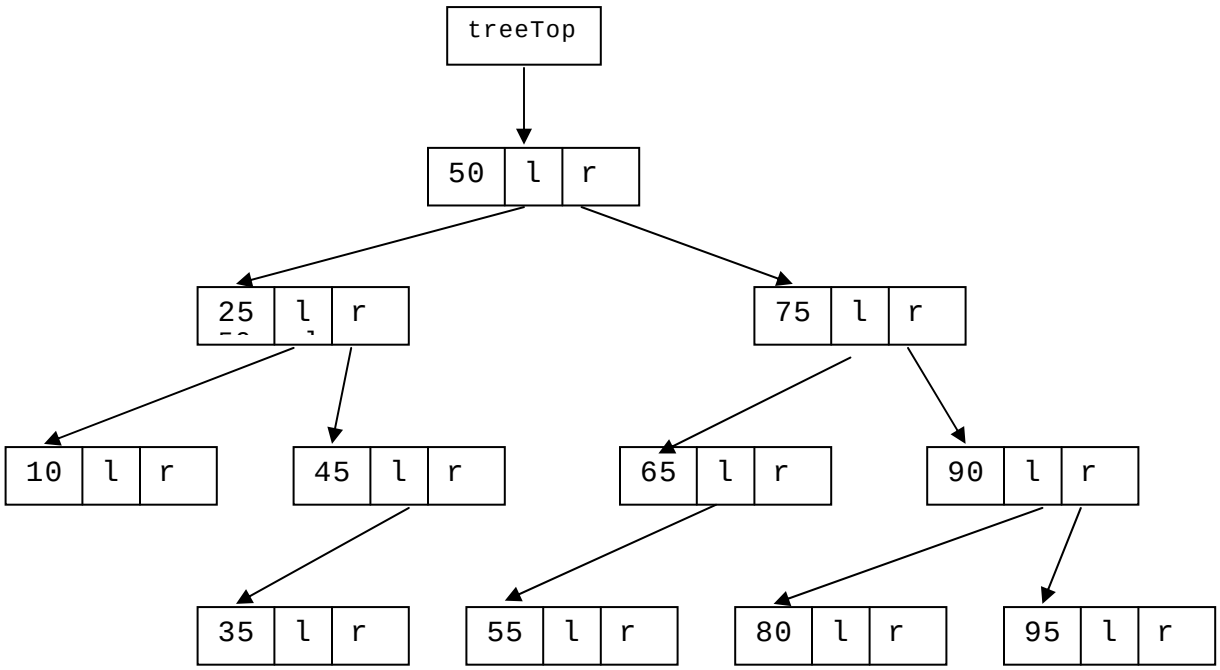

Figure 1 An integer tree

There are some leaves and nodes with child nodes in the above tree. How should one remove a node from this tree?

Clearly, removing a leaf is not hard. To do this, its parent must be found and set the child pointer to NULL, and the removed node is cleaned up (dynamic memory issue here).

Removing a node that has child nodes presents some challenges. The process must be accomplished by deleting the node and preserving the subtrees that the node links to. There are two possible cases to surface when removing a non-leaf node: (1) the node has one child, or (2) the node has two children.

Removing a non-leaf node with one child is easy to depict, as given in **Figure 2** below where node 45 is removed. The result can be obtained by re-attaching the link to the child from the parent of the parent (grand-something?) and deleting the node (which is the parent).

The other case with two children is not as easily obtained; for example, removing node 90. Clearly, the two subtrees cannot be attached to a single parent's link. So then, which link should be re-attached and to where?

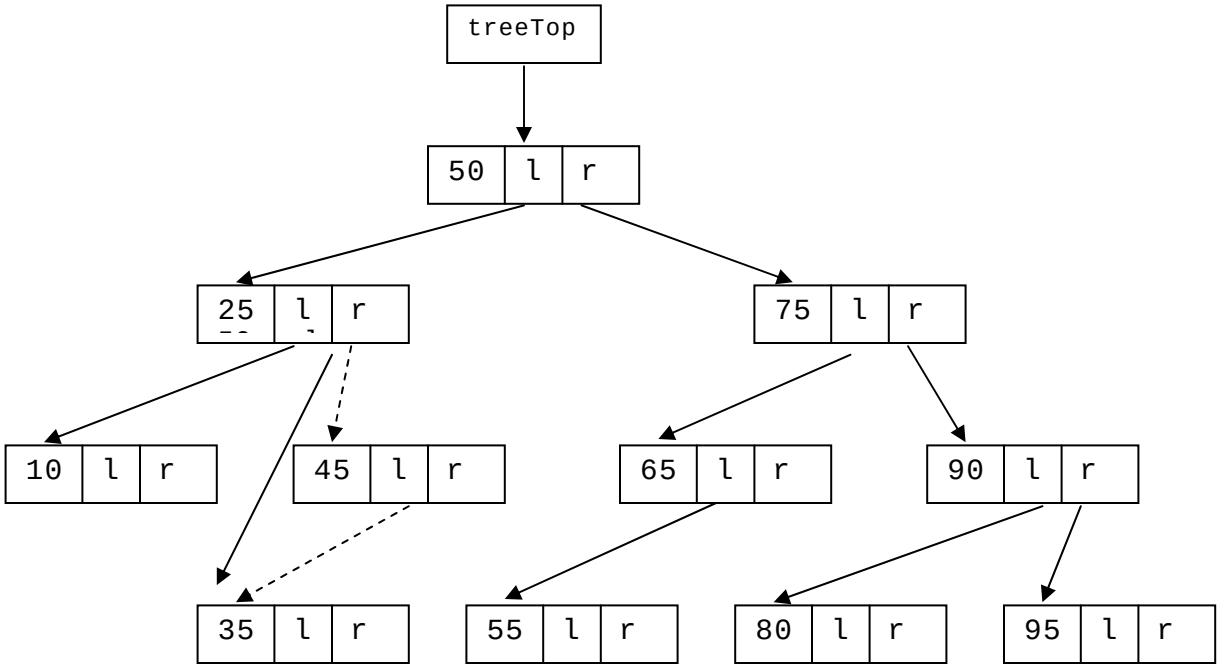

Figure 2 Removing a non-leaf node that has one child

One way to do it is to attach the node's right subtree to the parent, and then find a position in the right subtree to attach the left subtree. The result is shown in **Figure 3.**

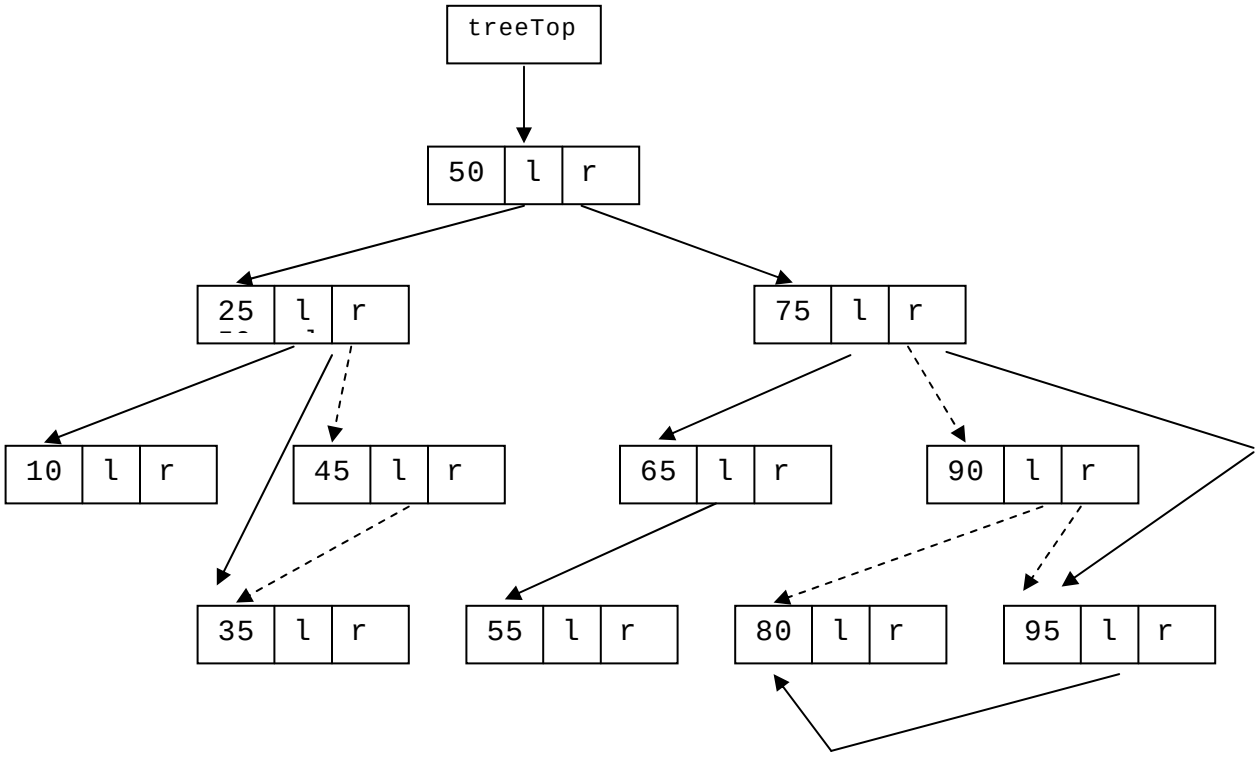

Figure 3 Removing a non-leaf node that has two children

Let's consider an implementation given as below.

Example 2

```
/**
 * Program Name: cis27L1112.c
 * Discussion:   Binary Search Tree -- Noda Removal
 */
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

typedef struct TreeInt* TreePtr;
typedef struct TreeInt* IntPtr;

struct TreeInt {
  int iValue;
  IntPtr left;
  IntPtr right;
};

void create(TreePtr newTree);
void insert(int aValue, TreePtr* myTree);
int find(int aValue, TreePtr myTree);

void removeNode(int oldValue, TreePtr* myTree);
IntPtr* findNode(int oldValue, TreePtr* myTree);
void removeUtil(IntPtr* ptrToNodeAddress);

void displayInOrder(TreePtr myTree);
void displayPreOrder(TreePtr myTree);
void displayPostOrder(TreePtr myTree);

void insertRecur(int aValue, TreePtr* myTree);
void insertRecurHelper(TreePtr* myTree, IntPtr newNode);

int main( ) {
  int ch;
  TreePtr workingTree = NULL;
  int iTest;

  printf("\n\nPreOrder Display:\n");  /* An empty tree */
  displayPreOrder(workingTree);

  printf("\n\nInOrder Display:\n");
  displayInOrder(workingTree);

  printf("\n\nPostOrder Display:\n");
  displayPostOrder(workingTree);

  insert(5, &workingTree);
  insert(15, &workingTree);
  insert(55, &workingTree);
  insert(25, &workingTree);
  insert(75, &workingTree);
  insert(65, &workingTree);
  insert(10, &workingTree);
  insert(-10, &workingTree);
```

```
  printf("\n\nPreOrder Display:\n");  /* Displaying the
                                        Binary Search Tree */
  displayPreOrder(workingTree);

  printf("\n\nInOrder Display:\n");
  displayInOrder(workingTree);

  printf("\n\nPostOrder Display:\n");
  displayPostOrder(workingTree);

  iTest = 25;

  if (find(iTest, workingTree)) {  /* Searching (finding)
                                      a node */
    printf("\n\nThe integer %d is found!\n", iTest);
  } else {
    printf("\n\nThe integer %d is not found!\n", iTest);
  }
  printf("\n\nEnter an integer + ENTER to continue ...\n");
  scanf("%d", &ch);

  printf("\nRemoving 25 ...\n");  /* Removing a Node */
  removeNode(25, &workingTree);

  printf("\n\nPreOrder Display:\n");  /* Displaying the
                                        Binary Search Tree */
  displayPreOrder(workingTree);

  printf("\n\nInOrder Display:\n");
  displayInOrder(workingTree);

  printf("\n\nPostOrder Display:\n");
  displayPostOrder(workingTree);

  printf("\n\nEnter an integer + ENTER to continue ...\n");
  scanf("%d", &ch);

  printf("\nRemoving 10 ...\n");  /* Removing a node */
  removeNode(10, &workingTree);

  printf("\n\nPreOrder Display:\n");  /* Displaying the
                                        Binary Search Tree */
  displayPreOrder(workingTree);

  printf("\n\nInOrder Display:\n");
  displayInOrder(workingTree);

  printf("\n\nPostOrder Display:\n");
  displayPostOrder(workingTree);

  printf("\n\nEnter a character + ENTER to stop ...\n");
  scanf("%d", &ch);

  return 0;
}

void insert(int aValue, TreePtr* myTree) {
  IntPtr newNode, nodePtr;

  newNode = (IntPtr) malloc(sizeof(struct TreeInt));
```

```
  assert(newNode);

  newNode->iValue = aValue;
  newNode->left = newNode->right = NULL;

  if (*myTree == NULL) {
    *myTree = newNode;
  } else {
    nodePtr = *myTree;
    while (nodePtr != NULL) {
      if (aValue < nodePtr->iValue) {
        if (nodePtr->left) {
          nodePtr = nodePtr->left;
        } else {
          nodePtr->left = newNode;
          break;
        }
      } else if (aValue > nodePtr->iValue) {
        if (nodePtr->right) {
          nodePtr = nodePtr->right;
        } else {
          nodePtr->right = newNode;
          break;
        }
      } else {
        printf("\nDuplicate value found!\n");
        break;
      }
    }
  }
  return;
}

int find(int aValue, TreePtr myTree) {
  TreePtr nodePtr = myTree;

  while (nodePtr) {
    if (nodePtr->iValue == aValue) {
      return 1;
    } else if (aValue < nodePtr->iValue) {
      nodePtr = nodePtr->left;
    } else {
      nodePtr = nodePtr->right;
    }
  }

  return 0;
}

void insertRecur(int aValue, TreePtr* myTree) {
  IntPtr newNode1;

  newNode1 = (IntPtr) malloc(sizeof(struct TreeInt));
  assert( newNode1 );

  newNode1->iValue = aValue;
  newNode1->left = newNode1->right = NULL;

  insertRecurHelper(myTree, newNode1);
}
```

```
void insertRecurHelper(TreePtr* myTree, IntPtr newNode) {
  if (*myTree == NULL) {
    *myTree = newNode;
  } else if (newNode->iValue < (*myTree)->iValue) {
    insertRecurHelper(&((*myTree)->left), newNode);
  } else {
    insertRecurHelper(&((*myTree)->right), newNode);
  }
}

IntPtr* findNode(int oldValue, TreePtr* myTree) {
  if (*myTree) {
    if (oldValue < (*myTree)->iValue)
      myTree = findNode( oldValue, &(*myTree)->left);
    else if (oldValue > (*myTree)->iValue)
      myTree = findNode(oldValue, &(*myTree)->right);
  }

  return myTree;
}
void removeNode(int oldValue, TreePtr* myTree) {
  IntPtr* ptrToNodeAddress;

  ptrToNodeAddress = findNode(oldValue, myTree);
  if (!(*ptrToNodeAddress)) {
    printf("\n%d is not found in the tree.\n", oldValue);
  } else if (*ptrToNodeAddress == *myTree) { /* The root node is
                                                being removed */
    removeUtil(myTree);
  } else {
    removeUtil(ptrToNodeAddress);
  }
  return;
}

void removeUtil(IntPtr* ptrToNodeAddress) {
  IntPtr tempNodePtr;

  if (*ptrToNodeAddress == NULL)
    printf("\nCannot remove empty node!\n");
  else if ((*ptrToNodeAddress)->right == NULL) {
    tempNodePtr = *ptrToNodeAddress;
    *ptrToNodeAddress = (*ptrToNodeAddress)->left; /* Reattaching left
                                                      child */
    free(tempNodePtr);
  } else if ((*ptrToNodeAddress)->left == NULL) {
    tempNodePtr = *ptrToNodeAddress;
    *ptrToNodeAddress = (*ptrToNodeAddress)->right; /* Reattaching right
                                                       child */
    free(tempNodePtr);
  } else { /*If node has two children*/
    tempNodePtr = (*ptrToNodeAddress)->right; /* Moving one node to the
                                                 right */

    while (tempNodePtr->left) /* Going to the end left node */
      tempNodePtr = tempNodePtr->left;

    tempNodePtr->left = (*ptrToNodeAddress)->left; /* Reattaching left
                                                      subtree */
```

```
    tempNodePtr = *ptrToNodeAddress;

    *ptrToNodeAddress = (*ptrToNodeAddress)->right; /* Reattaching right
                                                       subtree */
    free(tempNodePtr);
  }
  return;
}

void displayPreOrder(TreePtr myTree) {
  if (myTree) {
    printf("\n\tValue of node : %d", myTree->iValue);
    displayPreOrder(myTree->left);
    displayPreOrder(myTree->right);
  }

  return;
}

void displayInOrder(TreePtr myTree) {
  if (myTree) {
    displayInOrder(myTree->left);
    printf("\n\tValue of node : %d", myTree->iValue);
    displayInOrder(myTree->right);
  }

  return;
}

void displayPostOrder(TreePtr myTree) {
  if (myTree) {
    displayPostOrder(myTree->left);
    displayPostOrder(myTree->right);
    printf("\n\tValue of node : %d", myTree->iValue);
  }

  return;
}

/*OUTPUT


PreOrder Display:


InOrder Display:


PostOrder Display:


PreOrder Display:

        Value of node : 5
        Value of node : -10
        Value of node : 15
        Value of node : 10
        Value of node : 55
        Value of node : 25
        Value of node : 75
```

```
        Value of node : 65

InOrder Display:

        Value of node : -10
        Value of node : 5
        Value of node : 10
        Value of node : 15
        Value of node : 25
        Value of node : 55
        Value of node : 65
        Value of node : 75

PostOrder Display:

        Value of node : -10
        Value of node : 10
        Value of node : 25
        Value of node : 65
        Value of node : 75
        Value of node : 55
        Value of node : 15
        Value of node : 5

The integer 25 is found!


Enter an integer + ENTER to continue ...
1

Removing 25 ...


PreOrder Display:

        Value of node : 5
        Value of node : -10
        Value of node : 15
        Value of node : 10
        Value of node : 55
        Value of node : 75
        Value of node : 65

InOrder Display:

        Value of node : -10
        Value of node : 5
        Value of node : 10
        Value of node : 15
        Value of node : 55
        Value of node : 65
        Value of node : 75

PostOrder Display:

        Value of node : -10
        Value of node : 10
        Value of node : 65
        Value of node : 75
        Value of node : 55
```

```
        Value of node : 15
        Value of node : 5

Enter an integer + ENTER to continue ...
2

Removing 10 ...


PreOrder Display:

        Value of node : 5
        Value of node : -10
        Value of node : 15
        Value of node : 55
        Value of node : 75
        Value of node : 65

InOrder Display:

        Value of node : -10
        Value of node : 5
        Value of node : 15
        Value of node : 55
        Value of node : 65
        Value of node : 75

PostOrder Display:

        Value of node : -10
        Value of node : 65
        Value of node : 75
        Value of node : 55
        Value of node : 15
        Value of node : 5

Enter a character + ENTER to stop ...
3
*/
```