Informatics II
Exercise 8

Learning Goals:

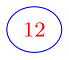
- Practise inserting items and deleting items in Binary Search Tree
- Implement functions of inserting items, deleting items, and searching elements in Binary Search Tree in C
- Implement Binary Search Trees in C.
- Validate implementation of Binary Search Trees.

Binary Search Trees

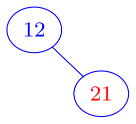
Task 1. Binary search tree is created by inserting nodes 12, 21, 15, 32, 13, 3, 8, 5, 1, 42, 17. Show structure of the tree after insertion of each node.

Initial tree
Empty binary search tree.

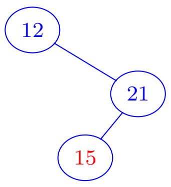
After insert 12



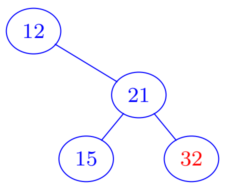
After insert 21



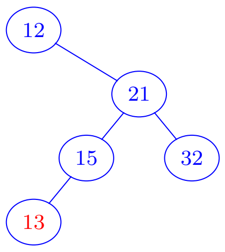
After insert 15



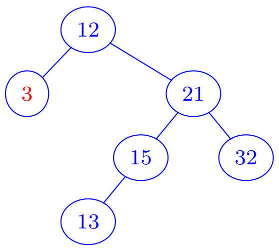
After insert 32



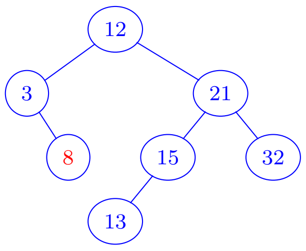
After insert 13



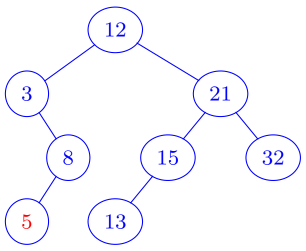
After insert 3



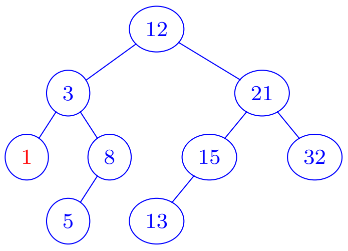
After insert 8



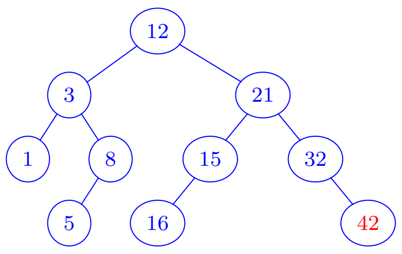
After insert 5



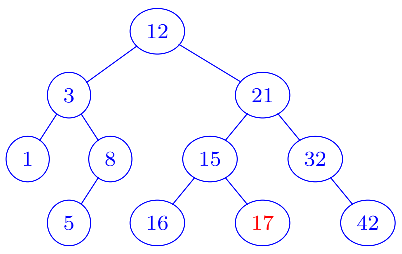
After insert 1



After insert 42

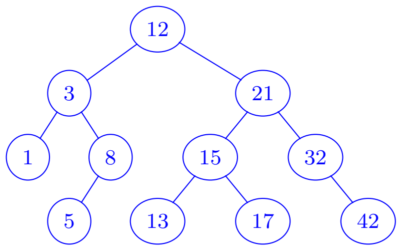


After insert 17

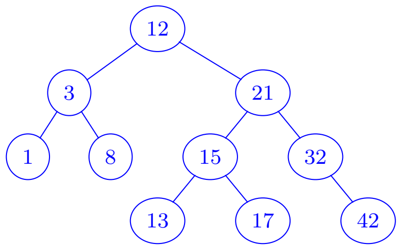


Task 2. Considering binary search tree created in `Task 1`, show the structure of the tree after deleting each of the following nodes 5, 32, 15, 21, 12. When deleting the node has two children, we first find the item with the largest value in the left subtree.

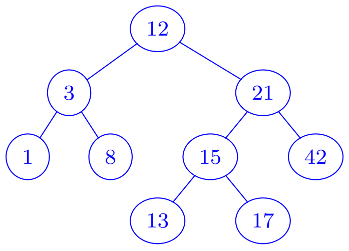
Initial tree



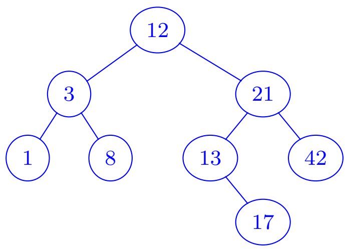
After delete 5



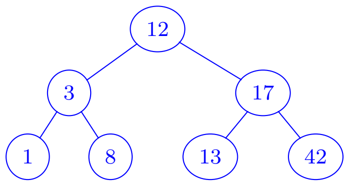
After delete 32



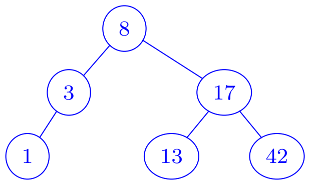
After delete 15



After delete 21



After delete 12



Task 3. The structure of the tree node is defined as follow. The entry to a binary search tree is the root that is also a tree node.

```
struct TreeNode {
    int val;
    struct TreeNode* left;
    struct TreeNode* right;
};
```

Write a C program that contains the following functions:

a) Write the function *void insert(struct TreeNode** root, int val)* that inserts an integer `val` into the binary search tree. Note that, you need to create a tree node for `val`, and find the correct position to insert the new tree node.

```
void insert(struct TreeNode** root, int val) {
  struct TreeNode* newTreeNode = NULL;
  struct TreeNode* prev = NULL;
  struct TreeNode* current = *root;
  newTreeNode = malloc(sizeof(struct TreeNode));
  newTreeNode->val = val;
  newTreeNode->left = NULL;
  newTreeNode->right = NULL;
  while (current != NULL) {
    prev = current;
    if (val < current->val){
      current = current->left;
    } else{
      current = current->right;
    }
  }
  if (prev == NULL) {
    *root = newTreeNode;
  } else if (val < prev->val) {
    prev->left = newTreeNode;
  } else {
    prev->right = newTreeNode;
  }
}
```

b) Write the function *struct TreeNode* search(struct TreeNode* root, int val)* that finds the tree node with value `val` and returns the node. If not exist, return NULL;

```
struct TreeNode* search(struct TreeNode* root, int val) {
  struct TreeNode* current = root;
  while (current != NULL && current->val != val) {
    if (val < current->val){
        current = current->left;
    } else{
        current = current->right;
    }
  }
  return current;
}
```

c) Write the function *void delete(struct TreeNode** root, int val)* that deletes the node with value `val` from the tree.

```
void delete(struct TreeNode** root, int val) {
  struct TreeNode* x = search(*root, val);
  if (x == NULL){ //search did not find an element, hence do nothing.
    return;
  }
  struct TreeNode* u = *root;
  struct TreeNode* prev = NULL; // parent of tree node with value = val
  while (u != x) {
    prev = u;
    if (x->val < u->val){
      u = u->left;
    } else{
      u = u->right;
    }
  }
  // Leaf and root case also handled in the no right or left branch. Since if it's leaf, its null anyway.
  if (u->right == NULL) { // there is no right branch
    if (prev == NULL){ // delete root
      *root = u->left;
    } else if (prev->left == u){ //if it's a left child, make left the new child
      prev->left = u->left;
    } else{
      prev->right = u->left;
    }
  } else if (u->left == NULL) { // there is no left branch
    if (prev == NULL){ // delete root
      *root = u->right;
    } else if (prev->left == u){ //if it's a left child, make right the new child
      prev->left = u->right;
    } else{
      prev->right = u->right;
    }
  } else{
    struct TreeNode* p = x->left;
    struct TreeNode* q = p;
    while (p->right != NULL) { //whilst right is null
      q = p;
      p = p->right;
    }
    if (prev == NULL){ // if we are at root
      *root = p;
    } else if (prev->left == u){ // if its a left child
      prev->left = p;
    } else{ //if its a right child
      prev->right = p;
    }
    p->right = u->right;
    if (q != p) {
      q->right = p->left;
      p->left = u->left;
    }
  }
  free(u);
```

d) Write *void printTree(tree *root)* which prints all edges of the tree from root in the console in the format `Node A -- Node B`, and each edge is printed in a separate line. The ordering of the printed edges does not matter and may vary based on your implementation.

```

void printTreeRecursive(struct TreeNode *root) {
  if (root == NULL)
    return;
  if (root->left != NULL) {
    printf("  %d -- %d\n", root->val, root->left->val);
    printTreeRecursive(root->left);
  }
  if (root->right != NULL) {
    printf("  %d -- %d\n", root->val, root->right->val);
    printTreeRecursive(root->right);
  }
}

void printTree(struct TreeNode *root) {
  printf("graph g {\n");
  printTreeRecursive(root);
  printf("}\n");
}
```

e) Write *struct TreeNode* maximum(struct TreeNode* node)*, which returns the node with the largest value in the subtree with root node `n`.

```

struct TreeNode* maximum(struct TreeNode* root) {
  struct TreeNode* current = root;
  while (current->right != NULL) { //keep going right until we find the maximum
        current = current->right;
  }
  return current;
```

f) Write *struct TreeNode* minimum(struct TreeNode* node)*, which returns the node with the smallest value in the subtree with root node `n`.

```

struct TreeNode* minimum(struct TreeNode* root) {
  struct TreeNode* current = root;
  while (current->left != NULL) { //keep going left until we find the minimum
        current = current->left;
  }
  return current;
```

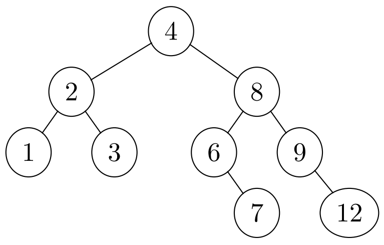
g) Write *int distanceToRoot(struct TreeNode* root, int val)* that returns the distance of the node with value `val` from the root node `root`. Assume that val exists in the Tree.

```

int distanceToRoot(struct TreeNode* root, int val) {
  if (root->val == val) return 0; // termination condition, else recursively search for solution
  else if (root->val>val) return 1+distanceToRoot(root->left,val);
  else return 1+distanceToRoot(root->right,val);
```

- Test your above implementations. For example, assume that the values 4, 2, 3, 8, 6, 7, 9, 12 and 1 are inserted into an empty tree, your program should produce the binary tree shown below.

Binary Tree Form



Test your program by performing the following operations:

- Create a root node `root` and insert the values 4, 2, 3, 8, 6, 7, 9, 12, 1.
- Print tree to the console.
- Print the minimum value of the tree.
- Print the distanceToRoot of node 7.
- Delete the values 4, 12, 2 from the tree.
- Print tree to the console.

- Print the maximum value of the tree.
- Print the distanceToRoot of node 6.

```

int main() {
  struct TreeNode* root= NULL;
  printf("Inserting: 4, 2, 3, 8, 6, 7, 9, 12, 1\n");
  insert(&root, 4);
  insert(&root, 2);
  insert(&root, 3);
  insert(&root, 8);
  insert(&root, 6);
  insert(&root, 7);
  insert(&root, 9);
  insert(&root, 12);
  insert(&root, 1);
  printTree(root);
  printf("Minimum Value: %d\n", minimum(root)->val);
  printf("Distance to root of node 7: %d\n",distanceToRoot(root,7));
  printf("Deleting: 4, 12, 2\n");
  delete(&root, 4);
  delete(&root, 12);
  delete(&root, 2);
  printTree(root);
  printf("Maximum Value: %d\n", maximum(root)->val);
```

Task 4. Given a rooted tree *T*, the *lowest common ancestor (LCA)* of two nodes *n1* and *n2* is defined as the node that is farthest from the root in *T* and has both *n1* and *n2* as descendants (where we allow a node to be a descendant of itself). Consequently, the LCA of *n1* and *n2* in *T* is the shared ancestor of *n1* and *n2* that is located farthest from the root. Note that *n1* and *n2* are in the tree *T*.

For example, given a tree as in Figure 1, the LCA of nodes *9* and *21* is node *15*.

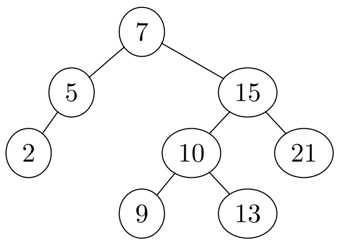


Figure 1: LCA of nodes 9 and 21 is node 15

Given values of two nodes in a Binary Search Tree, implement in C the function `struct TreeNode* lca(TreeNode* root, int n1, int n2)` that finds the *Lowest Common Ancestor (LCA)*. You may assume that both the values exist in the tree.

Write in C a program to test your implementation by performing the following operations:

- Create an empty tree and insert the nodes 7, 5, 6, 1, 9, 10, 8.
- Print the resulting tree using method described in **Task 3**.

- Print out the LCA of node *8* and node *9*

```c
struct TreeNode *lca(struct TreeNode *root, int n1, int n2)
{
    if (root == NULL) return NULL;

    // If both n1 and n2 are smaller than root, then LCA lies in left
    if (root->val > n1 && root->val > n2)
        return lca(root->left, n1, n2);

    // If both n1 and n2 are greater than root, then LCA lies in right
    if (root->val < n1 && root->val < n2)
        return lca(root->right, n1, n2);

    return root;
}
```

Initialization and Function Call:

```c
    int n1, n2;
    struct TreeNode *root = NULL;

    insert(&root, 7);
    insert(&root, 5);
    insert(&root, 6);
    insert(&root, 1);
    insert(&root, 9);
    insert(&root, 10);
    insert(&root, 8);

    printTree(root);

    n1 = 8;
    n2 = 9;
    struct TreeNode *t = lca(root, n1, n2);
    printf("LCA of %d and %d is %d \n", n1, n2, t->val);
```