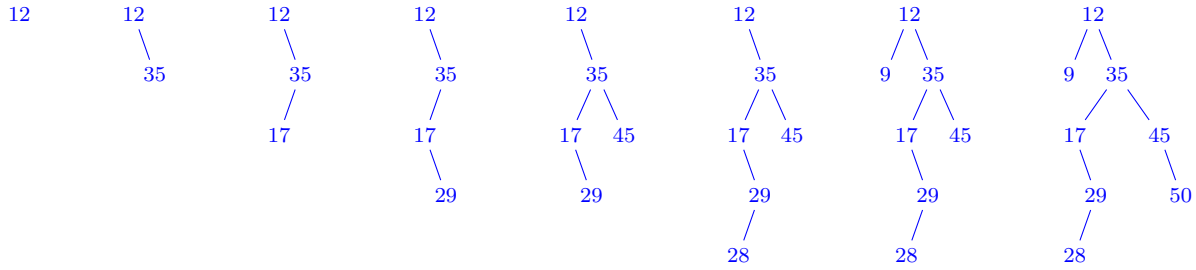
Informatics II
Exercise 8

Week 9

Binary Search Tree

Task 1. (Properties of Binary Search Tree) Binary search tree is constructed by inserting nodes one by one. Assume there is a sequence of nodes `[12, 35, 17, 29, 45, 28, 9, 50]`,

a. Draw the tree structure after inserting each node starting with 12.



Example: Taking `29` as an example.

(a) We start from the root, compare it with `12`. Since `29` is larger than `12`, we go to the right subtree.

(b) Then we compare it with `35` and `29` is less than `35`, so we go to the left subtree.

(c) At this time, we reach an end (current leaf). Since `29` is larger than `17`, we insert it to the right child.

b. Calculate the following properties of the tree and justify your answers.

(a) Height of the tree.

By definition, height of the tree is defined as the height of the root node, i.e., the length of the longest path from root to a leaf. Hence, in the above tree, the height of the tree is the length of the path from root to the leaf node `28`, which is 4.

(b) The depth of node `45`.

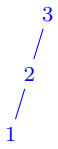
By definition, the depth of the node is the length of the path from root to this node. Here, the path from root to the node `45` is `12-35-45` and hence the depth is 2.

c. Is it true that the time complexity of search in a binary search tree is $O(1)$ in the best case? Identify when this best case is achieved.

If the node that we need to find is coincidently the root node, then we can return the root node within a constant time. Hence, in this case, the time complexity is $O(1)$.

d. Is it true that the time complexity of search in a binary search tree is $O(\log n)$ in the worst case where n is the number of nodes? Identify when this worst case is achieved.

In the worst case, the binary search tree is skewed like below:



In this case, the time complexity for searching for the node `1` is $O(n)$. Hence, this statement is false.

Task 2. Binary Search Tree in C The structure of the tree node is defined as follow. The entry to a binary search tree (the root) 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 *struct TreeNode* insert(struct TreeNode* root, int val)* that inserts an integer `val` into the binary search tree. The function returns the root of the tree.

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

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

```
1  struct TreeNode* delete (struct TreeNode* root, int val) {
2    struct TreeNode* x = search(root, val);
3    if (x == NULL) { // search did not find an element, hence do nothing, just simply return the root.
4      return root;
5    }
6    struct TreeNode* u = root;
7    struct TreeNode* parent = NULL; // parent of tree node with value = val
8    while (u != x) {
9      parent = u;
10     if (x->val < u->val) {
11       u = u->left;
12     } else {
13       u = u->right;
14     }
15   }
16   // Leaf and root case also handled in the no right or left branch. Since if
17   // it's leaf, its null anyway.
18   if (u->right == NULL) { // there is no right branch
19     if (parent == NULL) { // delete root
20       root = u->left;
21     } else if (parent->left == u) { // if it's a left child, make left the new child
22       parent->left = u->left;
23     } else {
24       parent->right = u->left;
25     }
26   } else if (u->left == NULL) { // there is no left branch
27     if (parent == NULL) { // delete root
28       root = u->right;
29     } else if (parent->left == u) { // if it's a left child, make right the new child
30       parent->left = u->right;
31     } else {
32       parent->right = u->right;
33     }
34   } else {
35     struct TreeNode* p = x->left;
36     struct TreeNode* q = p;
37     while (p->right != NULL) { // whilst right is null
38       q = p;
39       p = p->right;
40     }
41     if (parent == NULL) { // if we are at root
42       root = p;
43     } else if (parent->left == u) { // if its a left child
44       parent->left = p;
45     } else { // if its a right child
46       parent->right = p;
47     }
48     p->right = u->right;
49     if (q != p) {
50       q->right = p->left;
51       p->left = u->left;
52     }
53   }
```

```
54    free(u);
55    return root;
56  }
```

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.

```
1  void printTreeRecursive(struct TreeNode* root) {
2    if (root == NULL) return;
3    if (root->left != NULL) {
4      printf("  %d -- %d\n", root->val, root->left->val);
5      printTreeRecursive(root->left);
6    }
7    if (root->right != NULL) {
8      printf("  %d -- %d\n", root->val, root->right->val);
9      printTreeRecursive(root->right);
10   }
11 }
12
13 void printTree(struct TreeNode* root) {
14   printf("graph g {\n");
15   printTreeRecursive(root);
16   printf("}\n");
17 }
```

Task 3. Balance Binary Search Tree The time complexity for searching a key is $O(\log n)$ for a balanced binary search tree. Hence, it is useful if we can transform a binary search tree to a balanced binary search tree. Write a function that converts binary search tree to a balanced binary search tree, with the following sub-tasks:

a. Write a pseudocode for this function.

The idea of this function contains two steps:

(a) We first apply inorder tree walk. By doing so, we get a sorted array of all the nodes in the tree.

(b) From the sorted array, we generate a balanced binary search tree. To do so, we need the following steps:

i. First find the middle of the array and make it the root node.

ii. Recursively repeat the process for the left and right subtrees.

The pseudocode for inorder tree walk is already given in the lecture:

Algo: InorderTreeWalk(root)

```
arr ← [] ;
if p≠NIL then
    InorderTreeWalk(root → left);
    arr.push(root → val);
    InorderTreeWalk(root → right);
```

Then we show the algorithm for constructing binary search tree from the sorted array.

Algo: ConstructBSTFromArray(arr, start, end)

```
root ← NIL ;
if start > end then
  return NIL
mid = (start + end) / 2 ;
root ← arr[mid] ;
root.left ← ConstructBSTFromArray(arr, start, mid - 1) ;
root.right ← ConstructBSTFromArray(arr, mid + 1, end) ;
return root
```

Combining the above two functions together, we first apply inorder tree walk to get the sorted array. Then we construct the balanced binary search tree from the sorted array, as shown below:

Algo: BalanceBST(root)

```
sortedArray ← InorderTreeWalk(root) ;
balancedTree ← ConstructBSTFromArray(sortedArray, 0,
  sortedArray.length - 1) ;
```

b. Implement the function in C.

Similarly, we first present the inorder tree walk function in C.

```c
void InorderTraversal(struct TreeNode* root) {
  if (root == NULL) {
    return;
  }
  InorderTraversal(root->left);
  inorder_array[inorder_id] = root->val;
  inorder_id++;
  InorderTraversal(root->right);
}
```

Then we present the function for constructing binary search tree from the sorted array.

```c
struct TreeNode* ConstructBSTFromArray(int start, int end) {
  if (start > end) {
    return NULL;
  }
  struct TreeNode* root = malloc(sizeof(struct TreeNode));
  int mid = (start + end) / 2;
  root->val = inorder_array[mid];
  root->left = ConstructBSTFromArray(start, mid - 1);
  root->right = ConstructBSTFromArray(mid + 1, end);
  return root;
}
```

Hint: If you perform the inorder tree walk to a binary search tree, what pattern can you find in the output array? With the help of this pattern, can you construct the balanced binary search tree?

Task 4. Range Query by Trimming Binary Search Tree One important application of the binary search tree is for finding elements. Suppose you need to find all elements in the range $[\texttt{low}, \texttt{high}]$, write a pseudocode that returns the binary search tree with all its element lies in the range $[\texttt{low}, \texttt{high}]$. Write a pseudocode and C implementation for this task.

Note: You should not change the relative structure of the elements that will remain in the tree. For example, assume the original binary search tree is shown in Figure 1 and suppose you need to find all elements in the range $[17, 45]$, then the trimmed binary search tree should be shown in Figure 2.

The intuition of the solution is to recursively visit the entire tree. There will be 3 cases:

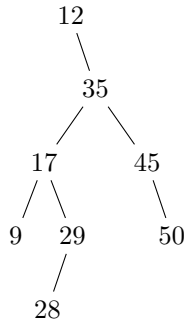


Figure 1: Original Binary Search Tree

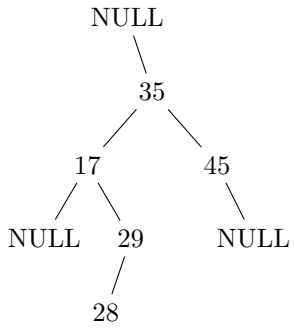


Figure 2: Trimmed Tree, the `NULL` node is used to indicate the nodes that are deleted from the original tree.

1. If the current node is larger than the `high`, then the trimmed tree must be to the left of the node. Hence, we only need to recursively trim the left subtree.
2. If the current node is smaller than the `low`, then the trimmed tree must be to the right of the node. Hence, we only need to recursively trim the right subtree.
3. Otherwise, we keep the current node and recursively trim its left and right subtrees.

Algo: TrimBST(root, low, high)

if *root==NIL* **then**
  return *root*
if *root.val > high* **then**
  return *TrimBST(root.left, low, high)*
if *root.val < low* **then**
  return *TrimBST(root.right, low, high)*
root.left ← TrimBST(root.left, low, high) ;
root.right ← TrimBST(root.right, low, high) ;
return root

The C implementation is shown below:

```
1  struct TreeNode* TrimBST(struct TreeNode* root, int low, int high) {
2    if (root==NULL) {
3      return NULL;
4    }
5    if (root->val > high) {
6      return TrimBST(root->left, low, high);
7    }
8    if (root->val < low) {
9      return TrimBST(root->right, low, high);
10   }
11   root->left = TrimBST(root->left, low, high);
12   root->right = TrimBST(root->right, low, high);
13   return root;
14 }
```