


# Trees

Topics: *Binary Trees, BST*

Date: 9 November, 2020

“ Trees sprout up just about everywhere in computer science. ”

Donald Knuth, *Art of Computer Programming*

Typically we associate the word ‘tree’ with some object looking like  having branches, leaves and possibly with fruits or flowers. But that is only what we see above the surface, the roots stays underground. From a data-structure designer perspective, we are often interested in what’s under the hood. In data-structures, a tree is generally considered as a root and its branches as depicted in figure 1. A tree is typically used to depict some kind of hierarchy or ordering, e.g. family tree. Syntax trees (see figure 2) and expression trees (see figure 3) are frequently used in various field of studies.

## 1 Tree

A *tree* data-structure is used to organize a collection of data values which are stored in *nodes* or *vertices*. The nodes are linked with each other forming the hierarchy. The links are known as *edges* or *branches*. In figure 1, the letters *A, B* etc. denote the nodes of the tree and the lines connecting two nodes denote the edges of the tree. Every two nodes in a tree, connected via an edge, forms a *parent-child* relationship among themselves. For example, *A* is parent of *B, C* and *D*, while each one of *B, C* and *D* is a child of *A*. We call two nodes to be *sibling* of each other if their parents are one and the same. For example, *B, C* and *D* are siblings of each other. The parent-child relationship can be generalized into *ancestor-descendant*. In this example, *A* is ancestor of *F* while *F* is descendant of *A*. There is a special designated node having no parent to it, is known as *root* of the tree. Here node *A* is the root of this tree.

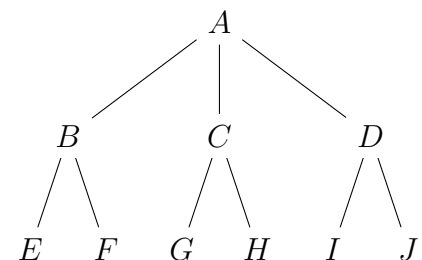


Figure 1: A sample tree structure

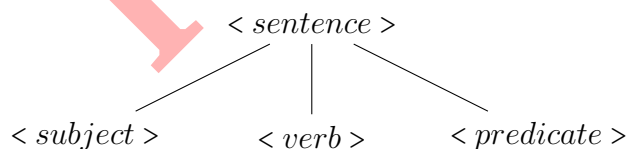


Figure 2: Grammar syntax for a simple english sentence

operands reside at leaves (drawn as square nodes) and all operators are placed at internal nodes (drawn as circles).

There exist some nodes in a tree, which do not have any child to it. Such nodes are called *external nodes* or *leaf nodes* or simply leaves of the tree. All other nodes, having at least one child, are called *internal nodes*. In this example, *E, F, G* and *H* are leaves while *A, B, C* and *D* are internal nodes. Also note that, for an expression tree (see figure 3) all

Now we should formally define the tree data-structure. A tree is recursively defined as a collection of nodes storing data values.

A tree or rooted tree  $T$  is a collection (finite set) of nodes which

- i) can either be empty, or
- ii) contains a special designated node called root, and rest of the nodes are partitioned into  $k$  ( $k \geq 1$ ) disjoint subsets  $T_1, T_2, \dots, T_k$  of  $T$ . Each  $T_i$  ( $1 \leq i \leq k$ ) is called a subtree of the root node.

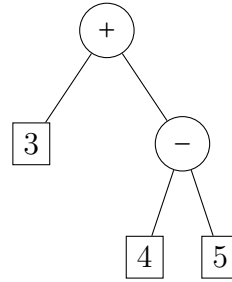


Figure 3: Expression tree for  $3 + (4 - 5)$

Following this definition, we can represent a tree using nested set notation. The tree in figure 1 would be represented as  $A(B(E, F), C(G, H), D(I, J))$ . As shown, here we first write the label of the root node and all subtrees of the root node is written recursively in nested form where siblings are separated by commas. The above definition can be further formalized using the concept of *abstract data-types (ADT)*.

A tree of abstract type  $T$

- i) is either empty (NULL), or
- ii) contains a special designated node called root, and one or more subtrees of the root node, each is of same type  $T$ .

We might think ordering of the elements in a collection is irrelevant and only thing that matters is whether an element exists in some collection or not. There are situations when order of the elements does matter. For example, in case of a syntax tree (see figure 2) position and ordering of subject, verb etc. is important. Similarly, for an expression tree (see figure 3) position of the two operands of a binary operator dictates the evaluated value, e.g. while  $4 - 5$  evaluates to  $-1$ ,  $5 - 4$  evaluates to  $+1$ . Thus we also define *ordered tree* where order of the children of each node is specified by their relative position in the tree.

An ordered tree of abstract type  $T$

- i) is either empty (NULL), or
- ii) contains a special designated node called root, and one or more ordered subtrees of the root node, each is of same type  $T$ .

We define *degree* of a node in a tree as the number of children of it. Therefore, degree of the leaves are always zero. The *degree of a tree*, also known as *arity*, is defined as the maximum possible degree a node of that tree can have. The *level* or *depth* of a node in a tree is defined as its distance from the root node, i.e. root node itself is at level or depth 0, its immediate children are at level or depth 1, grandchildren are at level or depth 2 and so on. In general, level or depth of a node is one more than that of its parent, starting with root node being at level or depth 0. Similarly, *height* of a node is defined as one more than maximum height among its children, with leaf nodes having height of zero. We define *height of a tree* as the height of the root node of that tree.

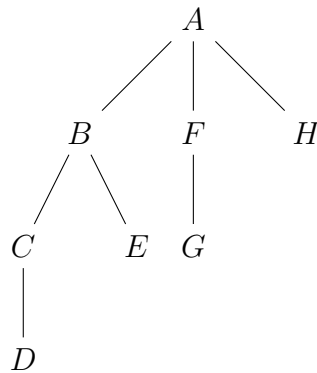


Figure 4: Another tree

## Exercise

- 1.1. For the tree in figure 4 write the nested set representation.
- 1.2. For the tree in figure 4 specify root node, leaf nodes, internal nodes.
- 1.3. For the tree in figure 4 specify the degree of the tree.
- 1.4. For the tree in figure 4 specify depth and height of each node.
- 1.5. For the tree in figure 4 how many non empty subtrees are there? Calculate height for each subtrees and also for the given tree.
- 1.6. Draw a tree of degree three with twelve nodes having height four, and one having height two.

## 2 Binary Tree

In data-structures, the simplest kind of trees are trees of arity two, known as *binary trees*. A binary tree is an ordered tree where each node has a left child and a right child. Formally, we define a binary tree as follows.

A binary tree of abstract type  $T$

- i) is either empty ( $NULL$ ), or
- ii) contains a special designated node called *root*, and exactly two subtrees, namely a left subtree and a right subtree of the the root, each is of same type  $T$ .

A *complete binary tree* is the one in which leaf nodes are present only in the last two levels. Here each level, except possibly the last, is filled and last level is filled from left to right order. A *full binary tree* is the one in which every node, except the leaves, has exactly two non-empty child. From the definition it is evident that a full binary tree is always a complete binary tree. Figure 5a and 5b depicts a complete binary tree and a full binary tree respectively, whereas the tree in figure 5c is neither full nor complete.

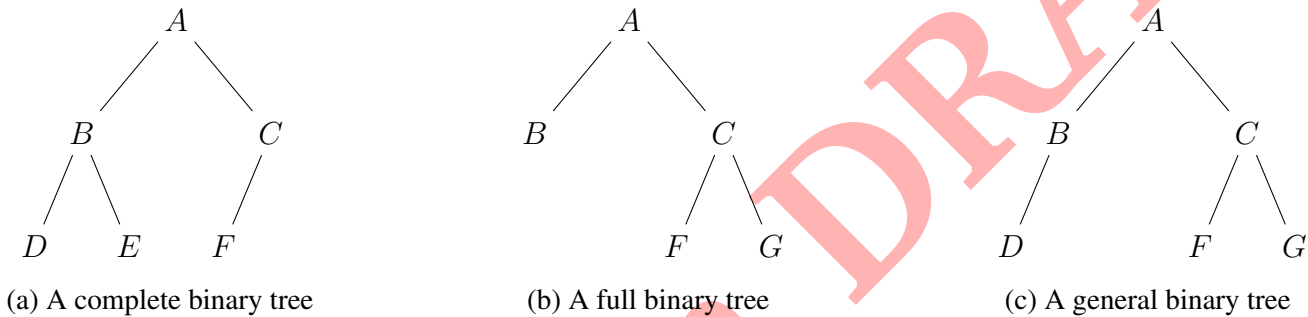


Figure 5: Caption

For a binary tree we will now prove a series of lemmas.

**Lemma 2.1.** *In a non empty binary tree the maximum number of node possibly exists on a level  $l$  is  $2^l$ .*

*Proof.* We will use mathematical induction to prove the lemma.

**base case** At level 0, there can only be a single node, the root node. Since  $2^0 = 1$ , the statement holds.

**inductive hypothesis** Let us assume the statement holds for some level  $k \geq 0$ , i.e. on  $k$ -th level maximum  $2^k$  nodes exists.

**inductive step** Now for the  $(k + 1)$ -th level, we may at most have two (child) nodes for each node in level  $k$ . Therefore, maximum possible nodes in this level is  $2 \times 2^k = 2^{(k+1)}$ . Thus the statement also holds for this level.

Therefore the statement holds for any level  $l \geq 0$ . ■

**Lemma 2.2.** *In a non empty binary tree of height  $h$  the maximum number of node possibly exists is  $2^{h+1} - 1$ .*

*Proof.* The maximum number of node possible in a binary tree when each level contains maximum possible number of nodes, i.e. the tree is a complete binary tree. The total number of node in a complete binary tree of height  $h$  is  $\sum_{l=0}^h 2^l = \frac{2^{h+1}-1}{2-1} = 2^{h+1} - 1$ . ■

**Lemma 2.3.** *In a non empty binary tree of height  $h$  the minimum number of node possible is  $h + 1$ .*

*Proof.* The minimum number of node possible in a binary tree when each level contains exactly a single node. Skewed trees (see figure 6a and 6b) or trees with zigzag pattern (see figure 6c) are such examples. In a height  $h$  tree the number of levels is  $h + 1$ , thus the lemma is proven. ■

**Corollary 2.3.1.** *Maximum possible height of a binary tree with  $n$  nodes is  $n - 1$*

*Proof.* This directly follows from lemma 2.3. ■

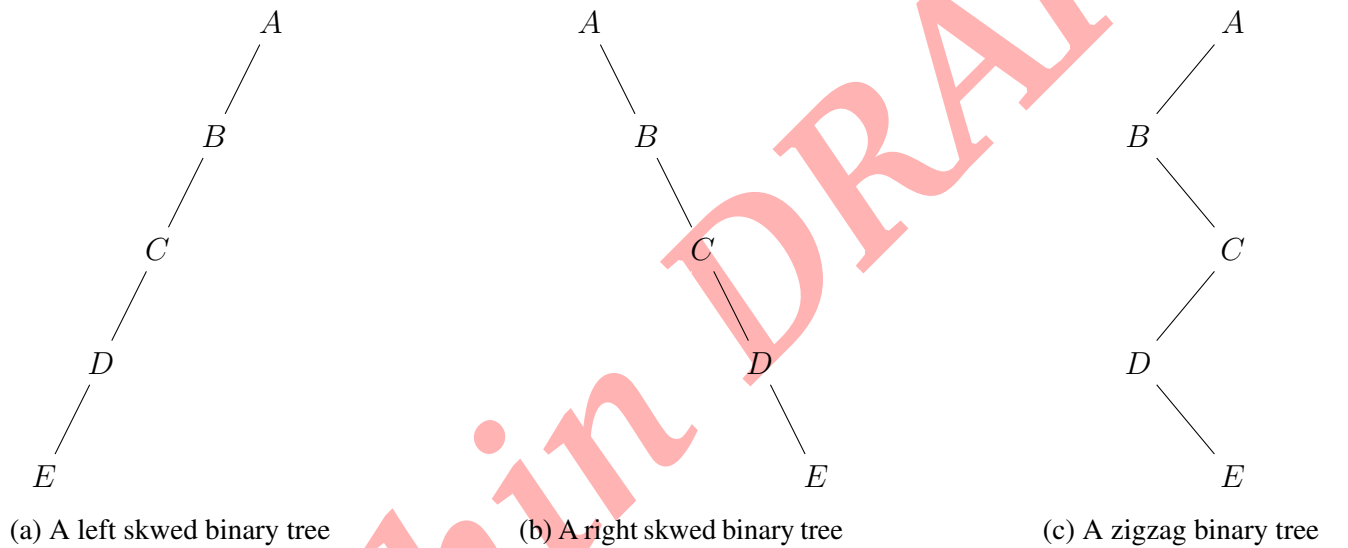


Figure 6: Special cases of binary trees

**Lemma 2.4.** *Minimum possible height of a binary tree with  $n$  nodes is  $\lceil \log_2(n + 1) \rceil - 1$*

*Proof.* Here we need to prove a lower bound of height. More precisely, we need to show that  $h \geq \lceil \log_2(n + 1) \rceil - 1$  for any binary tree with  $n$  nodes and height  $h$ . From lemma 2.2 we have an upper bound of  $n$ , i.e.  $n \leq 2^{h+1} - 1$ . Following algebraic manipulation proves the lemma.  $n \leq 2^{h+1} - 1$  implies  $n + 1 \leq 2^{h+1}$ . Taking  $\log_2$  on both sides we have  $\log_2(n + 1) \leq h + 1$ . Since  $h$  is an integer so is  $h + 1$ , thus  $\log_2(n + 1) \leq h + 1$  implies  $\lceil \log_2(n + 1) \rceil \leq h + 1$  which follows the result. ■

**Corollary 2.4.1.** *When height of a binary tree with  $n$  nodes is minimum, lets call this height  $h_{min}$ , we have  $h_{min} \leq \lceil \log_2 n \rceil$ .*

*Proof.* Here we are considering a binary tree with  $n$  nodes organized in such a way that minimum height is attained. Since by lemma 2.2 we can accommodate  $2^{h_{min}} - 1$  in a tree of height  $(h_{min} - 1)$ , we must have  $n \geq 2^{h_{min}}$ . Taking  $\log_2$  on both sides we have  $\log_2 n \geq h_{min}$ . Since  $h$  is an integer we can write  $\lceil \log_2 n \rceil \geq h_{min}$ . ■

**Lemma 2.5.** For a non empty binary tree with  $n$  nodes and  $e$  edges,  $n = e + 1$

*Proof.* We will prove this by mathematical induction.

**base case** When  $n = 1$  there can be no edge in the tree, i.e.  $e = 0$ , thus the statement trivially holds.

**inductive hypothesis** Let us assume the statement holds for all trees with  $n = k$  for some  $k \geq 1$ . Since we have  $n = e + 1$ , the number of edges in such a tree is  $e = k - 1$ .

**inductive step** Now if we add a new node as a leaf in a tree with  $k$  nodes, we also introduce a new edge to link this new node to some existing node with zero or one child. Thus new values of  $n$  and  $e$  are  $k + 1$  and  $(k - 1) + 1 = k$  respectively. This clearly implies that the statement also holds for the updated tree with  $k + 1$  nodes.

Therefore the statement holds for any tree with  $n$  nodes with  $n \geq 1$ . ■

**Lemma 2.6.** For a non empty binary tree we have  $n_0 = n_2 + 1$  where  $n_i$  denoted number of degree  $i$  nodes in the binary tree ( $i \in \{0, 1, 2\}$ ).

*Proof.* For a binary tree with total  $n$  nodes we have,  $n = n_0 + n_1 + n_2$ . Now a degree  $i$  node in a binary tree has exactly  $i$  number of children. Thus total links in a binary tree can be calculated as  $e = n_0 \times 0 + n_1 \times 1 + n_2 \times 2$ . Now from lemma 2.5 we have  $n = e + 1$ . Substituting the values on both sides and simplifying we have  $n_0 = n_2 + 1$ . ■

**Lemma 2.7.** There are  $\frac{1}{n+1} 2^n C_n$  possible (unlabeled) binary tree with  $n$  nodes.

*Proof.* The expression  $\frac{1}{n+1} 2^n C_n$  is known as *Catalan Number*. The proof is a bit lengthy and out of scope of this discussion. You may look on [web](#) for a formal proof. ■

## Storing a Binary Tree

### Array based representation of a binary tree

One possible way to store a binary tree using an array where contents of each node is placed onto the array. For a binary tree of height  $h$  by lemma 2.2 we know there can possibly be  $2^{h+1} - 1$  nodes at the most. We create an array of size  $2^{h+1} - 1$  and store each node level wise into the array. For an empty child, we consider there exists a NULL node entry and reflect that into the array. Figure 7 depicts array representation of a binary tree. As shown here, we leave out spaces for null links except for the children of the nodes in the very last level. This ensures that given an array representation of some binary tree, we can uniquely construct the binary tree.

The array notation allows us to access children and parent of a node in constant time. More precisely, for a node stored at index  $i$ , its left child is located at index  $2i$  and right child is located at index  $2i + 1$ , if they exists. Similarly, the parent of a node residing at index  $i$  can be found at index  $\lfloor i/2 \rfloor$ . This can be verified from the example in figure 7.

```
procedure LChild (i) :
  [ return 2 * i
```

```
procedure RChild (i) :
  [ return 2 * i + 1
```

```
procedure Parent (i) :
  [ return [i/2]
```

Although, there are several benefits in this representation like fast access of parent/children of a node, no storage space wastage due to pointers for linking nodes with each other, no dynamic memory allocation

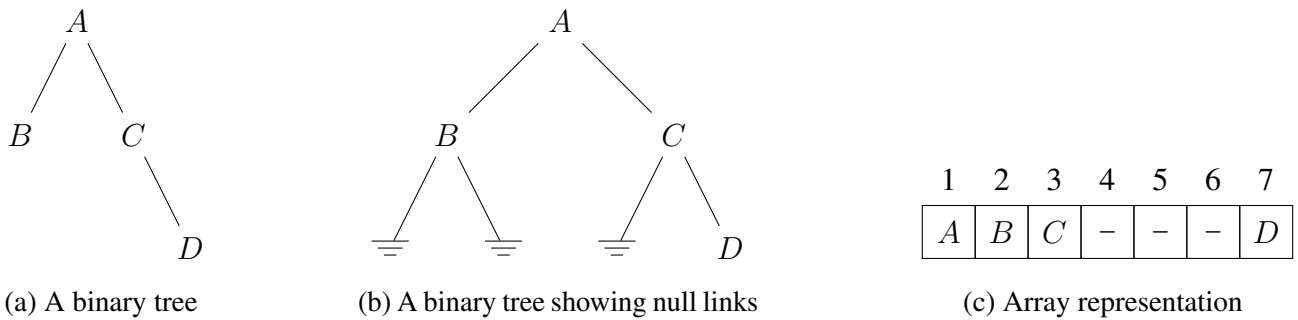


Figure 7: Array representation of a binary tree

requirement; this storage strategy has some drawbacks. In most cases, like in figure 7, there are many nodes before the last level which do not have two non empty children. This causes the array to have many unused cells as seen here causing storage space wastage. Furthermore, this scheme is not at all suitable for storing dynamic data where frequent insertion/deletion happens. This storage representation is ideal for cases like a complete binary tree with total number of nodes known before hand. We will revisit this in section 6 for storing Heap trees.

### Linked representation of a binary tree

In this storage scheme we define a node structure to store data as well as two links for the left child and right child of that node. Definition of a node structure in this tree representation can be given as following C code. Here we are assuming *data* part of a node contains integer data for simplicity, in practice it can be of any type even complex user-defined ones. Figure 8 depicts the structure of a single node while figure 9 shows the tree in figure 7a using linked representation.

```
typedef struct node {
    int data;
    struct node *left;
    struct node *right;
} Node;
```

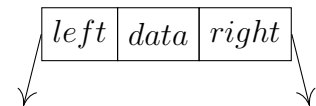


Figure 8: Node structure

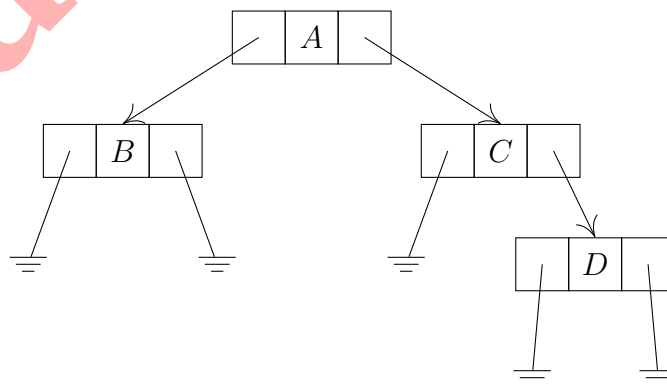


Figure 9: Linked representation

Now we can define a C function to create a new tree node containing some given data and another to free up the allocated space for a given node.

```
Node* make_node(int data) {
    Node *n = (Node*)malloc(sizeof(Node));
    n->data = data;
    n->left = n->right = NULL;
    return n;
}
```

```
void free_node(Node *n) {
    n->left = n->right = NULL;
    free(n);
}
```

The following segment of code will construct the tree in figure 7a. We are storing the label of the nodes as character data stored inside them.

```
Node *root = make_node('A');
root->left = make_node('B');
root->right = make_node('C');
root->right->right = make_node('D');
```

A tree in this storage representation is represented and accessed via its root node. The above piece of code also demonstrate how different nodes can be accessed through the root node.

**Lemma 2.8.** *For a non empty binary tree with  $n$  nodes there are  $n + 1$  null links.*

*Proof.* We will prove this by induction.

**base case** When  $n = 1$  there is a single node with its left and right links being null, thus the statement trivially holds.

**inductive hypothesis** Let us assume the statement holds for all binary trees with  $n = k$  for some  $k \geq 1$ . The number of null links in such a tree is  $k + 1$ .

**inductive step** Now if we want to add a new node as a leaf in a tree with  $k$  nodes, we must attach the new node to some existing node with free (null) link. The new node, being a leaf node, introduces two new null links. Thus number of null links in this new tree with  $k + 1$  nodes, is  $(k + 1) - 1 + 2 = (k + 1) + 1$ . This clearly implies that the statement also holds for the updated tree with  $k + 1$  nodes.

Therefore the statement holds for any tree with  $n$  nodes with  $n \geq 1$ . ■

## Exercise

2.1. For an array representation of a binary tree, prove relationship between the parent and child indices.



- 2.2. Derive the relationship between the parent and child indices if we follow array indexing starting with 0 like in C language.
- 2.3. Rewrite the definition of `make_node` so it can now accept pointer references of its left child and right child. Using this new definition, construct the tree in figure 7a in bottom-up fashion.

```
Node* make_node(int data, Node *left, Node *right) {//your code}
```

- \*2.4. Consider the definition of `Node` structure so that it can now accommodate any type of data. Such generic structures generally uses generic `void` pointer to hold any type of data. We extend the definitions of `make_node` and `free_node` function accordingly.

```
typedef struct node {
    void *data;
    struct node *left;
    struct node *right;
} Node;
```

```
Node* make_node(void *data, int size) {
    Node *n = (Node*)malloc(sizeof(Node));
    n->data = malloc(size);
    memcpy(n->data, data, size);
    n->left = n->right = NULL;
    return n;
}
```

```
void free_node(Node *n) {
    n->left = n->right = NULL;
    free(n->data);
    free(n);
}
```

Using these functions, demonstrate how information of students (name, roll etc.) can be stored into such a tree structure.

- 2.5. Compare and contrast the two storage techniques described above.
- \*2.6. There is another possible implementation of a binary tree where we store the nodes in an array. To avoid storage wastage, we use array indices as node references instead of pointers. This has an advantage of contiguous node allocation and fast access over conventional linked based sparsed allocation of nodes. Details of this alternative implementation can be found in [1]. Implement this alternative representation binary trees.

### 3 Binary Tree Traversals

*Traversal* of a tree refers to visiting each node of the tree exactly once. For any node, we can perform one of the three operations:

**V** visiting the node itself, process the data part of the node or display its contents

**L** traversing to its left child

**R** traversing to its right child

There are  $3! = 6$  possible ordering of these three (V, L and R) operations. Out of these six permutations, we consider only three orderings, namely *preorder* (VLR), *inorder* (LVR) and *postorder* (LRV).

```

procedure preorder (root):
    if root is not NULL then
        visit the root node
        traverse left subtree of root using preorder
        traverse right subtree of root using preorder
  
```

```

void preorder (Node *root) {
    if (root != NULL) {
        visit (root->data);
        preorder (root->left);
        preorder (root->right);
    }
}
  
```

Listing 1: Recursive preorder traversal

The above pseudocode and corresponding C code describe the *preorder* traversal of a binary tree. The traversal is described as a recursive process with the implicit base case (stopping condition) being *root* is *NULL*. The *visit* method can perform any operation on the data, typically some constant operation like displaying the data. Since at any given node we are only doing constant amount of work, the total time complexity of the preorder procedure is  $O(n)$ , where  $n$  is the number of nodes in a given tree. Similarly, we can define methods for *inorder* and *postorder*.

```

void inorder (Node *root) {
    if (root != NULL) {
        inorder (root->left);
        visit (root->data);
        inorder (root->right);
    }
}
  
```

```

void postorder (Node *root) {
    if (root != NULL) {
        postorder (root->left);
        postorder (root->right);
        visit (root->data);
    }
}
  
```

Listing 2: Recursive inorder and postorder traversal

### Traversal on Expression Trees

A preorder traversal on an expression tree with binary operators (and possibly unary ones also) results into *prefix* notation of the corresponding expression. For example preorder traversal of the tree in 3 results into:  $+3 - 45$  which is prefix expression of  $3 + (4 - 5)$ . Similarly, inorder and postorder traversal gives *infix* and *postfix* expressions respectively.

## Non-recursive Implementations of Tree Traversals

Observe that last statement of the *preorder* traversal is a tail-recursion which can easily be replaced by an while loop as follows.

```
void preorder(Node *root) {
    while (root != NULL) {
        visit(root->data);

        preorder(root->left);

        root = root->right;
    }
}
```

The remaining recursive call is not a tail recursion. Thus eliminating this recursive call requires a Stack and little extra work. The fully non-recursive implementation of preorder traversal is given below. Here we are taking one extra parameter *n* denoting number of nodes in the tree. We create a Stack of size *n* to store node references.

Listing 3: Preorder traversal with single recursion

```
void preorderIterative(Node *root, int n) {
    //create a Stack of size n
    Node *node = root;
    while ( node != NULL or !stackEmpty() ) {
        //visit and move to left child while keeping track
        while (node != null) {
            visit(node->data);

            push(n);
            node = node->left;
        }

        // backtrack and go to the right child
        node = pop();
        node = node->right;
    }
}
```

Listing 4: Non-recursive implementation of preorder traversal

The non-recursive implementation of inorder traversal is given in code listing 5 which is very similar to the non-recursive implementation of preorder traversal. Where as eliminating recursion from postorder traversal (without any tail recursive) is bit complicated. We first define a *mirrored tree* as a mirror image of a binary tree. Therefore, for each node in a binary tree its left and right child switch places in the mirrored tree, that is they become right and left child respectively. Now observe that preorder traversal on a mirrored tree is exactly the reverse of postorder traversal on the original tree. Formally, the pseudocode for postorder traversal is given in code listing 6.

```

void inorderIterative(Node *root, int n) {
    //create a Stack of size n
    Node *node = root;
    while ( node != NULL or !stackEmpty() ) {
        //move to left child while keeping track
        while (node != null) {
            push(n);
            node = node->left;
        }

        // backtrack, visit and go to the right child
        node = pop();
        visit (node->data);

        node = node->right;
    }
}

```

Listing 5: Non-recursive implementation of preorder traversal

```

procedure postorderIterative (root):
    mirror the tree rooted at root
    obtain preorder traversal of the mirrored tree
    output the preorder traversal in reverse

```

Listing 6: Non-recursive implementation of preorder traversal

## Reconstruction of a Binary Tree from Traversals

Given the inorder and preorder traversals of a binary tree, the tree can easily be reconstructed. Consider the following a binary tree with inorder traversal visiting in order  $D, B, E, A, F, C$  and preorder traversal visiting nodes in order  $A, B, D, E, C, F$ . Since  $A$  visited first in preorder traversal, it must be the root of the tree. Now all node visited before  $A$  in inorder traversal must belong to the left subtree of  $A$  and all nodes visited after  $A$  must belong to the right subtree of  $A$ . Therefore, inorder traversal on the left subtree of the root would give  $D, B, E$ . The preorder traversal on this subtree would preserve their relative order with respect to the preorder traversal of the entire tree. Thus preorder traversal on the left subtree of  $A$  must be  $B, D, E$ . Similarly, for the right subtree of  $A$ , the inorder traversal would be  $F, C$  and preorder traversal would be  $C, F$ . The entire tree can be reconstructed by recursively following this procedure. Figure 10 depicts the step by step reconstruction of the binary tree.

Similarly, given the postorder and inorder traversals of a binary tree the tree can be reconstructed. However, we may fail to uniquely reconstruct the original tree if only preorder and postorder traversals are provided.

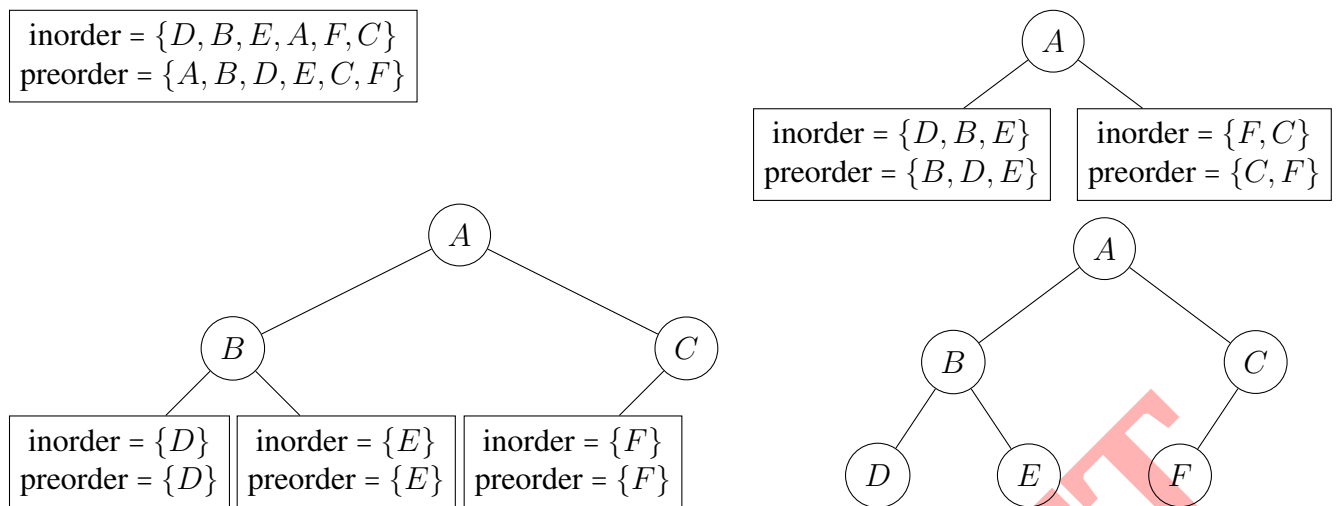


Figure 10: Construction of a binary tree from inorder and preorder traversals

## Exercise

- 3.1. The visit operation in a traversal procedure can be modified to perform various actions in linear time.
  - a) Given a binary tree and a key to be searched in the binary tree, we can devise a recursive searching algorithm by modifying the preorder traversal. The key is found if root contains its value, otherwise we can recursively search on the left and right subtrees of the root node. Write a C function to implement this algorithm.
  - b) Height of a binary tree can be defined recursively as  $1 + \max(\text{height of left subtree, height of right subtree})$ . Provide a C function to compute height of a given binary tree.
  - c) Write a C function to count the leaf nodes in a given binary tree.
- 3.2. Implement a recursive procedure that mirrors a given binary tree with  $n$  nodes in  $O(n)$  time. Also devise a non-recursive implementation of it.
- 3.3. Show that iterative implementations of the traversals takes  $O(n)$  running time on a tree with  $n$  number of nodes.
- 3.4. Consider the following non-recursive implementation of preorder traversal in code listing 7. Compare this with the one defined in code listing 4.
- 3.5. Observe that, preorder traversal on a mirrored tree is just like a preorder traversal on the original tree with left and right child swapped. Also observe that, a stack of size  $n$  can easily reverse a sequence of  $n$  elements by first pushing all the elements into the stack, and then popping them out from it. Using these two observations write a single C function for iterative postorder traversal which uses two stacks by modifying the iterative preorder traversal.
- \*3.6. Show that the iterative postorder traversal can be directly implemented using a single stack.
- \*3.7. Devise a C function to traverse a binary tree in level order by using a Queue.
- 3.8. It might be possible to uniquely reconstruct a tree from its traversals.

```
void preorderIterative2(Node *root, int n) {
    //create a Stack of size 2*n
    Node *n;
    push(root);
    while ( ! stackEmpty() ) {
        node = pop();
        if (node != null) {
            visit(n->data);

            push(node->right);
            push(node->left);
        }
    }
}
```

Listing 7: Another non-recursive implementation of preorder traversal

- a) Write a formal algorithm and prove that given inorder and preorder traversals of a tree, the tree can be uniquely reconstructed.
- b) Write a formal algorithm and prove that given inorder and postorder traversals of a tree, the tree can be uniquely reconstructed.
- c) Show an example where given preorder and postorder traversals of a tree the tree cannot be uniquely reconstructed.
- d) Devise an algorithm and show that given preorder and postorder traversals of a tree, the tree can be uniquely reconstructed if and only if all non leaf nodes in the tree has exactly two children and all nodes are uniquely labeled.

## 4 Binary Search Tree

Up until now, we have seen that data can be organized in a tree like structure. Since there is no particular restriction where a data item can be stored, therefore to look-up for any particular data item in the tree we may have to search the entire tree. Thus in order to quickly locate a data item, it is wiser to restrict where in the tree that item can be stored. Keeping this in mind, we formally define a *binary search tree* (BST).

*A binary search tree is a special kind of binary tree which satisfies the following property*

**BST property:** *for every node, the value at the node is greater than all the values at its left subtree and value at the node is smaller or equal to all the values in its right subtree.*

The equality condition allows duplicate elements to be stored in the BST. One can modify the BST property to allow repeated elements to be stored in the left subtree instead of the right subtree of a node. Figure 11 shows structure of a binary search tree. Here node  $R$ , which is also the root, is represented by a circle and its two subtrees  $L$  and  $R$  are represented by triangles. The labels  $<$  and  $\geq$  on the links denotes that smaller elements are stored in the left subtree and larger or equal items are stored in the right subtree respectively. We should note that definition the node structure for a BST is same as that of a binary tree, described above.

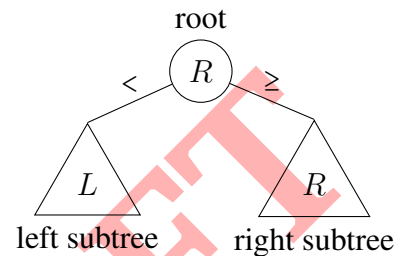


Figure 11: BST structure

**Lemma 4.1.** *Inorder traversal of a BST gives the data item in ascending order.*

*Proof.* We can prove this by mathematical induction on the number of nodes  $n$  in the tree.

**base case** When  $n = 1$  there there is a single node, thus the statement trivially holds.

**inductive hypothesis** Let us assume the statement holds for all trees with  $n \leq k$  for some  $k \geq 1$ .

**inductive step** Now consider a tree with  $n = k + 1$  nodes. We can decompose this tree into root and its two subtrees. The *inorder* traversal will first list all nodes in the left subtree, then the root, and finally all nodes in the right subtree. Since left and right subtree can have at most  $k$  nodes in it, by our inductive hypothesis, *inorder* traversal on each of these subtree will list the nodes in ascending order of their values. Combining this with the BST property, we observe that *inorder* traversal of this tree with  $n = k + 1$  nodes, visits the nodes in ascending order of their values.

Thus the statement is true for any non empty BST. ■

**NB:** *inorder* traversal on any binary tree is just a left to right scan of the tree structure when drawn properly.

### Searching

Now we should justify why using BST is beneficial over a normal binary tree for storing data. One can guess from the name itself that BST is an efficient structure for searching any stored data. More specifically, given *key* value to be searched in a BST given by its *root*, we proceed as follows. The value at the root node is compared against the *key*, if they are equal then we are done. If the *key* is smaller than the root then it cannot be in the right subtree of the root due to the BST property, thus we can recursively search on the left subtree only. Similarly, if the *key* is larger or equal to the root then it

cannot be in the left subtree, and we can recursively search on the right subtree only. Formally, the search procedure for a BST is given below.

```
Node* search(Node *root, int key) {
    if (root == NULL) return NULL;
    if (root->data == key) return root;
    if (root->data > key) return search(root->left, key);
    else return search(root->right, key);
}
```

For a successful search the best possible scenario occurs when the key is present at the root node itself requiring only constant ( $O(1)$ ) amount of work. In the unfavorable cases we may have to traverse down the given tree up to some leaf node, and the search successfully terminates. Whereas a unsuccessful search terminates at a (NULL) child of some leaf node. In both these cases we may have to traverse to the deepest level of the tree. Since we are doing only constant amount of work at each level of the tree, the total work done in the worst case is  $O(h)$  where  $h$  is the height of the given BST. Alternatively we can write a recurrence for the above procedure. Let  $T(h)$  denotes the time required to search an element in a height  $h$  BST. As per the procedure we perform a few checking at this level, and recursively move to one of the subtree which are of height at most  $h - 1$ . Thus  $T(h) \leq T(h - 1) + O(1)$ . Solving this we get  $T(h) = O(h)$  in the worst case. Since  $h$  is typically smaller than the total number of nodes in the tree, searching in a BST is much efficient than searching from unorganized data. The search procedure can easily be converted into a non-recursively form as follows.

```
Node* searchIterative(Node *root, int key) {
    while (root != NULL) {
        if (root->data == key) return root;
        if (root->data > key) root = root->left;
        else root = root->right;
    }
    return NULL;
}
```

## Insertion

In a BST new data elements must be added without violating the BST property. A new node is generally attached to an existing tree as a leaf node. If the given BST is empty the new node becomes the new root of that tree. If the tree is not empty, then the new node must be added either to the left subtree or to the right subtree of the root. Depending on the values of the root and the new node, we can recursively proceed to one of the subtree such that the BST property is satisfied after the insertion. The following C code formalizes this procedure. already

```
Node* insert(Node *root, int data) {
    if (root == NULL) return make_node(data);
    if (root->data > data) root->left = insert(root->left, data);
    else root->right = insert(root->right, data);
    return root;
}
```



In the above code, notice that the new node is always added into an empty subtree, making the newly added node the root of that subtree. Since the root of a subtree can be updated upon calling the `insert` procedure, the root is always returned and in the calling method the link is updated accordingly. Like the search procedure insertion procedure also runs in  $O(h)$  time.

### Deletion

Deletion of a node from a BST is a completely different story. The node to be deleted may not be a leaf node, which makes things little complicated. If there is a single child of that node, the child can safely be pushed up without violating BST property. Figure 12 depicts the scenario. Whereas, if the node to be

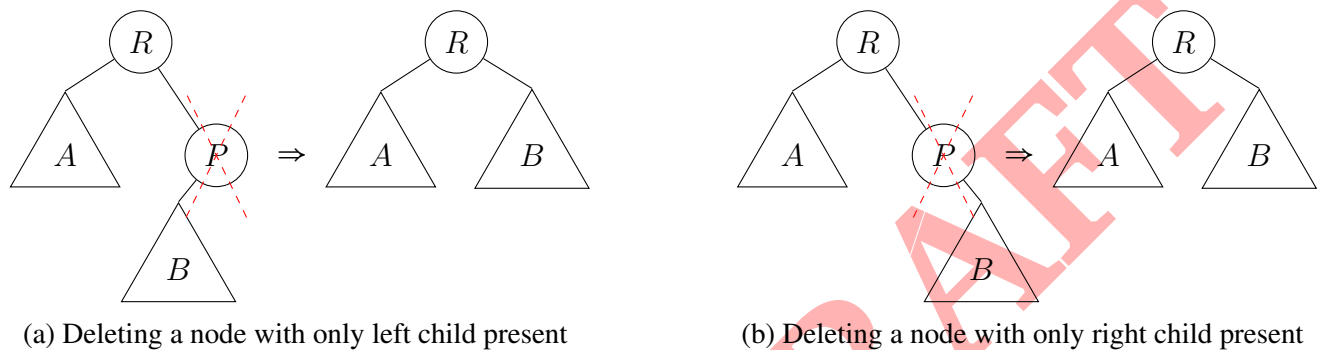


Figure 12: Deletion of a BST node having a single child

deleted has both left child and right child present, then we need to do some extra work. From a sorted list, if one element is deleted its place is filled by its successor in the list. By lemma 4.1 we can claim that if a node is deleted, its place should be filled by the node listed immediately next to the deleted one in the inorder traversal of the tree, known as *inorder successor* of the node being deleted.

**Lemma 4.2.** *Inorder successor of a node having both children present, is always a leaf node or a node with only right child present.*

*Proof.* It is sufficient to show that the inorder successor is the minimum most element in its right subtree. The proof is left as an exercise. ■

Therefore, we have the following three cases:

**case 1** [leaf node with zero child] If the node to be deleted has no child, it can safely be removed from the BST without violating the BST property.

**case 2** [node with only one child] If the node to be deleted has only one child (either left or right, but not both), we can remove the node and attach this child to the parent of the deleted node as shown in figure 12. This also preserves the BST property.

**case 3** [node with both children] If the node to be deleted has both left child and right child present, we replace this node by its inorder successor and delete the original inorder successor from the right subtree of the deleted node. This deletion of inorder successor, can be done via a recursive call. Observe that this manipulation does not violates the BST property.

The following C code formalizes this procedure. The `findMin()` procedure returns the node with the minimum most value in a given nonempty BST.

```

Node* delete(Node *root, int data) {
    if (root == NULL) return NULL;
    if (root->data > data)
        root->left = delete(root->left, data);
    else if (root->data < data)
        root->right = delete(root->right, data);
    else { //delete the root
        if (root->left == NULL && root->right == NULL) { //case 1
            free_node(root);
            root = NULL;
        } else if (root->left != NULL && root->right != NULL) { //case 3
            Node *n = findMin(root->right); //inorder successor
            root->data = n->data;
            root->right = delete(root->right, n->data);
        } else { //case 2
            Node *n = (root->left == NULL ? root->right : root->left);
            free_node(root);
            root = n;
        }
    }
    return root;
}

Node* findMin(Node *root) {
    while (root->left != NULL) root = root->left;
    return root;
}

```

## Exercise

- 4.1. Write a C function to find the minimum most element from a BST. Also specify the time complexity of this procedure. How would you modify this function if you wish to find the largest element from a BST.
- 4.2. Show that the C function in code listing 8 correctly verifies whether a given binary tree with all unique values is a BST or not when invoked as `isBST(root, INT_MIN, INT_MAX)`.

```

int isBST(Node *root, int min, int max) {
    if (root == NULL) return 1;
    if (root->data < min || root->data > max)
        return 0;
    return isBST(root->left, min, root->data) &&
           isBST(root->right, root->data, max);
}

```

Listing 8: Checking whether a given binary tree is a BST or not

- 4.3. Extend the code in code listing 8 to handle binary trees with duplicate elements.
- 4.4. Let us define a *special binary tree* where for every node in the tree, the value at the node is greater than the value at its left child (if exists) and value at the node is smaller or equal to the value at its right child (if exists). Verify that every BST is also a special binary tree. Is the converse also true?
- 4.5. Show that insertion of a new data into a height  $h$  BST takes  $O(h)$  time.
- 4.6. Compare the following non-recursive insertion procedures.

```
Node* insertIterative1(Node *root, int data) {
    Node *ptr = root;
    if (root == NULL) return make_node(data);
    while (1) {
        if (ptr->data > data) {
            if (ptr->left == NULL) {
                ptr->left = make_node(data);
                break;
            }
            ptr = ptr->left;
        } else {
            if (ptr->right == NULL) {
                ptr->right = make_node(data);
                break;
            }
            ptr = ptr->right;
        }
    }
    return root;
}
```

```
Node* insertIterative2(Node* root, int data) {
    Node *ptr = root, *ptr_parent = NULL;
    while (ptr != NULL) {
        ptr_parent = ptr;
        if (ptr->data > data)
            ptr = ptr->left;
        else
            ptr = ptr->right;
    }
    if (ptr_parent == NULL)
        root = make_node(data);
    else if (ptr->data > data)
        ptr_parent->left = make_node(data);
    else
        ptr_parent->right = make_node(data);
    return ptr_parent;
}
```

```
Node* insertIterative3(Node* root, int data) {
    Node **ptr = &root;
```

```
while (*ptr != NULL) {
    if (data > (*ptr)->data)
        ptr = &(*ptr)->right;
    else
        ptr = &(*ptr)->left;
}
*ptr = make_node(data);
return root;
```

- 4.7. Prove that the minimum most element in a BST resides in the leftmost node, i.e. the node does not have any left child attached to it. Also prove that, the maximum element resides in the rightmost node in a BST.
- 4.8. Inorder successor of a node having both children present, is always a leaf node or a node with only right child present. [hint: show that the inorder successor is the minimum most element in its right subtree]
- 4.9. Formally prove that all three cases for the deletion procedure preserves the BST property.
- 4.10. Prove that the case 3 of the deletion procedure can be handled by using the inorder predecessor instead of the inorder successor. Write C code demonstrating this modification.

## 5 Height Balanced Trees

Given  $n$  values we can arrange them into  $\frac{1}{n+1} \binom{2n}{n}$  (the  $n$ -th Catalan Number) possible BST structures. To search an item in a BST, we need to actually go to that level where the item actually resides. Therefore, different items requires different searching time depending on its position in a given BST. Therefore structure, more precisely the height of a BST influences the lookup time for an item. Let us define *average search time* of a BST as the average of search times of all the items in that BST. We implicitly assume that each item in the BST is equally likely to be searched for.

$$\begin{aligned} \text{average search time} &= \frac{1}{n} \sum_{i=1}^n \text{access time of } i\text{-th element} \\ &= \frac{1}{n} \sum_{i=1}^n (\text{depth level of } i\text{-th element} + 1) \\ &= \frac{1}{n} \sum_{l=0}^h (\text{number of nodes in level } l) * (l + 1) \quad [h \text{ is the height of the BST}] \end{aligned}$$

This clearly suggests that a BST with shorter height would have smaller average search time compared to a BST with longer height containing same elements. Figure 13 shows a BST with 15 nodes arranged in minimum possible height. We also calculate the average search time for this tree.

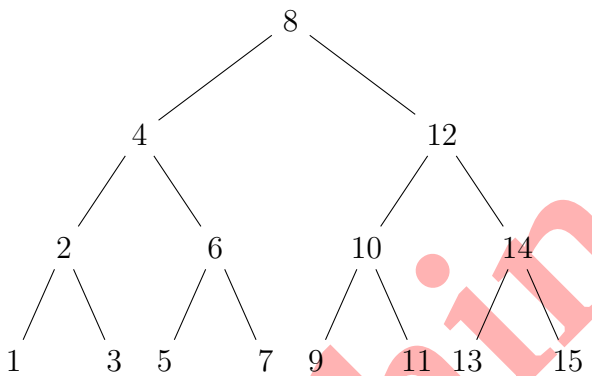


Figure 13: A BST with 15 nodes arranged in minimum possible height

$$\begin{aligned} \text{average search time} &= \frac{1}{15} \sum_{i=1}^{15} \text{access time of } i\text{-th element} \\ &= \frac{1}{15} (4 + 3 + 4 + 2 + 4 + 3 + 4 + 1 + \\ &\quad 4 + 3 + 4 + 2 + 4 + 3 + 4) \\ &= \frac{1}{15} (1 + 2 \times 2 + 3 \times 4 + 4 \times 8) \\ &= \frac{1 + 4 + 12 + 32}{15} \\ &= \frac{49}{15} \approx 3.2667 \end{aligned}$$

Now let us consider few different arrangements of these 15 nodes and calculate the average search time for those BST structures. Figure 14 depicts some possible arrangements of a BST with 15 nodes. As evident the BST with minimal height, given in figure 13 has the lowest average search time among few other possible BST structures with same 15 nodes given in figure 14.

We know that height of a binary can be computed recursively as  $1 + \max(\text{height of the left subtree, height of the right subtree})$ . Therefore in order to minimize the height a tree we need to minimize the height of its two subtrees. Following this recursive pattern, we can say that height of the tree is minimized as the tree gets balanced, i.e. nodes are almost equally distributed in the subtrees.

We say a binary tree is *height balanced* if for each node in the tree, the heights of the left subtree and the right subtree are within a constant difference from each other. The most common choice for the constant is 1. The higher the value of the constant, more the unbalanced tree structure. It can be shown

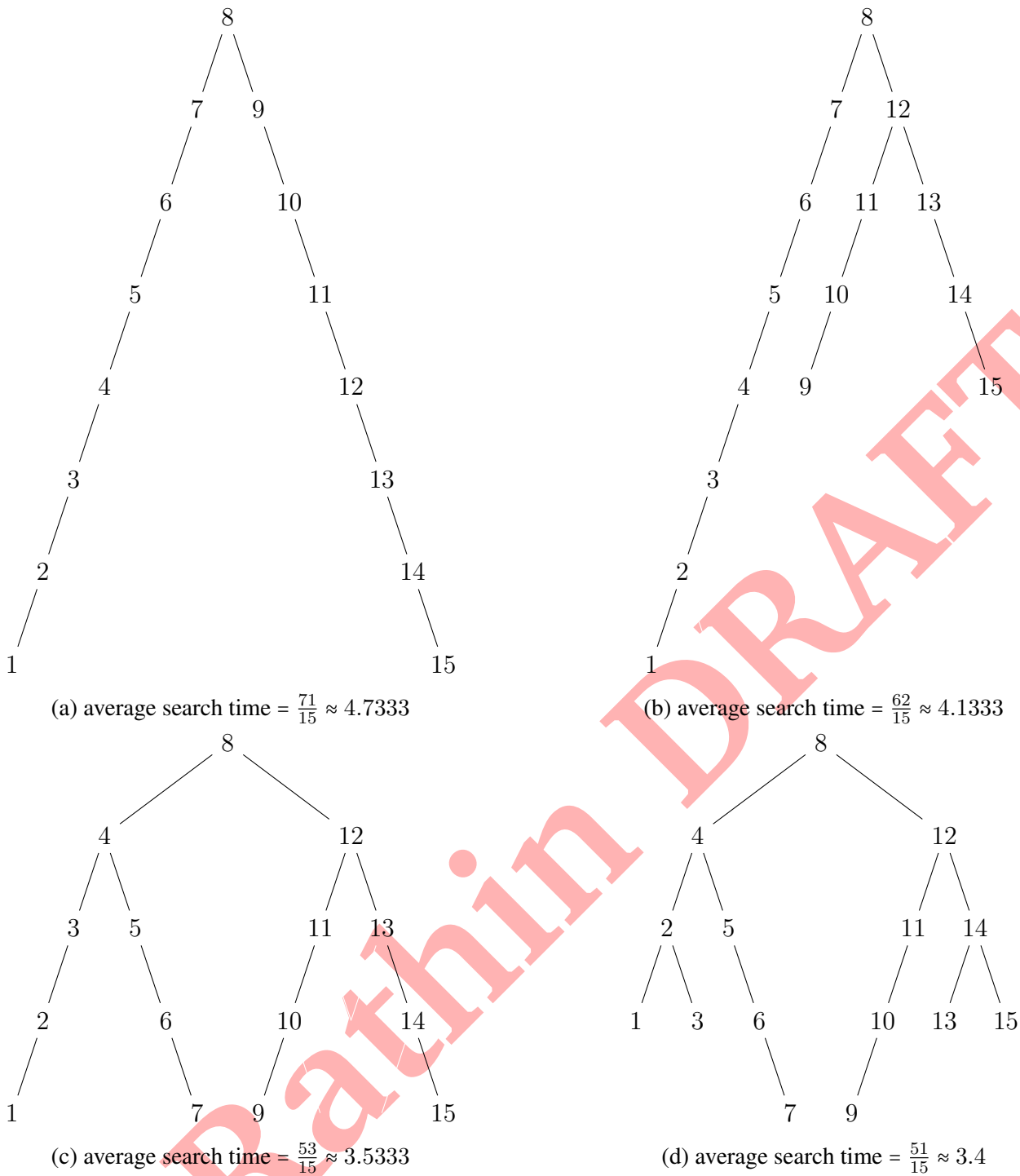


Figure 14: Various BST structures with 15 nodes

that height of such a height balanced tree with  $n$  nodes is  $O(\log n)$ . This implies that, if we can maintain a height balanced BST even after some updation (insertion or deletion) in the tree, the searching can be done efficiently in  $O(\log n)$  time.

We define *balance factor* ( $bf$ ) of a node as the difference between height of its left subtree ( $h_L$ ) and the height of its right subtree ( $h_R$ ), i.e.  $bf = h_L - h_R$ . Therefore, we say a BST is height balanced if for every node the balance factor lies in  $\{-1, 0, +1\}$ , i.e.  $|bf| \leq 1$ . Whenever a new node is inserted into a

height balanced BST, it may increase the height of a subtree where it gets inserted. This may cause some ancestor node of this new node to have  $|bf| = 2$ , which signifies that the tree has now become unbalanced. It might just be possible to make the tree balanced again by rearranging some nodes. Figure 15 depicts an example of such a rearrangement. Here the balance factors are also shown for every node. In case of a deletion the height may reduce of a subtree which may also result into a unbalanced tree. The resultant tree can similarly be made balanced.

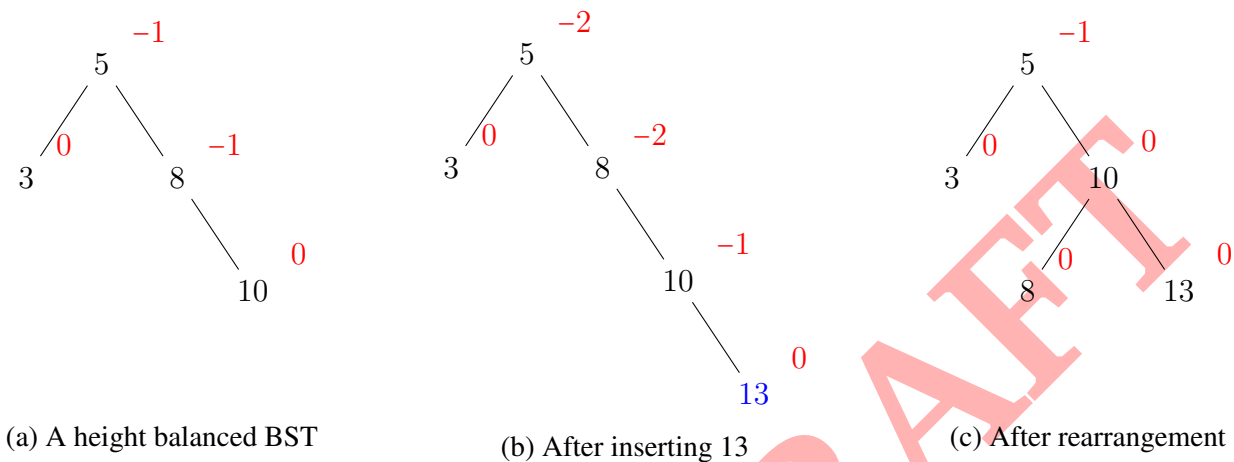


Figure 15: Balancing height of a BST after insertion

**Lemma 5.1.** Show that the height of a height balanced tree with  $n$  nodes is at most  $O(\log n)$ .

*Proof.* We assume the balance factor ( $bf$ ) of each node satisfies  $|bf| \leq 1$ . Let  $f(h)$  denotes the minimum number of nodes that must be stored in a height  $h$  height balanced tree. We can trivially verify that  $f(0) = 1$  and  $f(1) = 2$ . Now such a tree will have minimum number of nodes when one subtree of the root is of height  $h - 1$  and another is of height  $h - 2$ . Therefore,  $f(h) = 1 + f(h - 1) + f(h - 2)$ .

$$\begin{aligned}
 f(h) &= 1 + f(h - 1) + f(h - 2) \\
 &> 2f(h - 2) > 4f(h - 4) > 8f(h - 6) > \dots > 2^k f(h - 2 * k) = 2^k f(0) \quad [\text{when } k = h/2] \\
 \therefore f(h) &> 2^{h/2} \\
 \therefore h &< 2 \log_2 f(h)
 \end{aligned}$$

Now setting  $f(h) = n$  we have our required result. ■

## AVL Tree

In 1962 two Soviet mathematicians G. M. Adelson-Velsky and E. M. Landis proposed a self-balancing binary search tree in which they described a scheme for rebalancing a BST after an insertion or deletion operation is performed. This scheme is named as *AVL rotation* in their honor, and the self-balancing BST is commonly referred to as *AVL tree*.

### Insertion into an AVL tree

To insert a new element into an AVL tree (already height balanced), we first follow the procedure of BST insertion described in section 4. As depicted in figure 15 this insertion may result into a unbalanced BST

and thus a rebalancing is required. This rebalancing is done by *rotations*. The four possible cases where the tree gets unbalanced due to an insertion and how they are taken care of by rotations, are discussed below.

### Case LL (left child's left subtree)

Suppose an insertion into the left subtree ( $A$ ) of the left child ( $Q$ ) of a node ( $P$ ) has made the tree unbalanced by causing the node  $P$  to have balance factor of  $+2$ . Before discussing how rotation helps to rebalance the tree, let us analyze what were the tree structure before the insertion did happen.

Since, a single insertion into a already balanced BST can increase height of any subtree by at most one, change in balance factor must also be at most one. Also, observe that insertion into a BST only possibly causes change in balance factors of its ancestor nodes in the tree. Balance factors of all other nodes remain unaltered.

We assume that  $P$  is the nearest ancestor of the new node to get a balance factor outside of  $\{-1, 0, +1\}$ . Due to our specification, balance factor of  $P$  can now only be  $+2$ . Therefore it must have been  $+1$  before the insertion. Since the balance factor of  $P$  has increased from  $+1$  to  $+2$ , the insertion must have had increased the height of the left subtree of  $P$  i.e. subtree rooted at  $Q$ . This implies, the insertion must have had increased the height of the left subtree ( $A$ ) of  $Q$ . Thus balanced factor of  $Q$  must also have been increased. If it had been increased from  $-1$  to  $0$  the height the subtree rooted at  $Q$  cannot have increased. Therefore, it must be the case that balanced factor of  $Q$  was  $0$  before the insertion and now it is  $+1$ . If we assume that the height of the subtree  $A$  was  $h$  before the insertion, we can derive the heights of other subtrees from the above facts. The situations before and after the insertion happened is depicted in figure 16. Now if we perform a *right rotation* at  $P$  the tree becomes as shown in figure 17b. As evident, after the



Figure 16: Scenario when a LL insertion requires rebalancing

rotation the parent-child relationship between  $P$  and  $Q$  is reversed.  $Q$  becomes the new root of this tree. Subtree  $B$  is now present as the left child of  $P$ . The balance factors of this new tree suggest that the tree is now balanced. All that remains, is to show that the new tree is indeed a BST. For this we draw two imaginary vertical lines containing the subtree  $B$  as depicted in figure 17. It is now evident that inorder traversal of each these trees in figure 17 will visit the nodes in identical order. Thus the tree in figure 17b is a balanced BST with the newly added node.

### Case RR (right child's right subtree)

The mirror case of LL occurs when we insert an element into the right subtree ( $C$ ) of the right child ( $Q$ ) of a node ( $P$ ) and  $P$  is the nearest ancestor of the new node to get a balance factor outside of  $\{-1, 0, +1\}$ ,



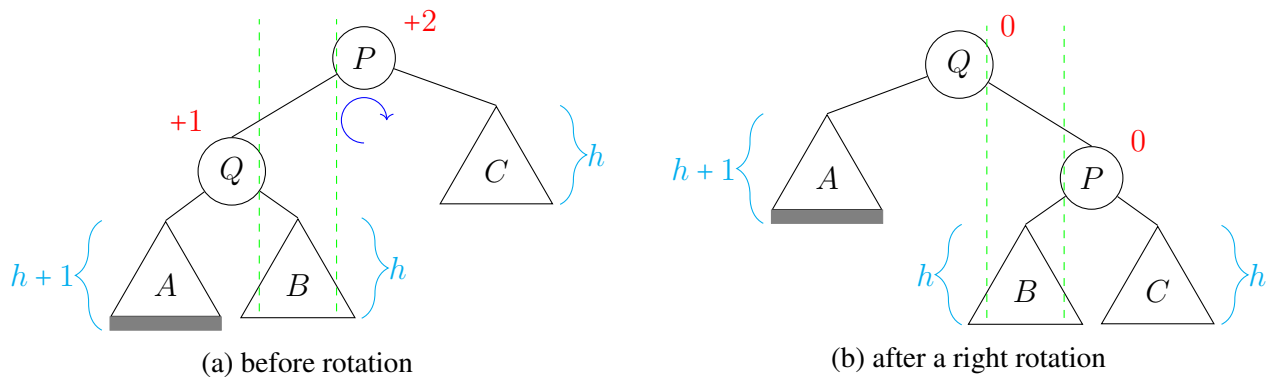


Figure 17: Rebalancing after an LL insertion

i.e.  $bf = -2$ . This unbalanced tree can be rebalanced by a *left rotation*, which is just the mirror of a right rotation, at  $P$ . Figure 18 depicts the scenario of a RR insertion and how the resultant BST can be rebalanced.

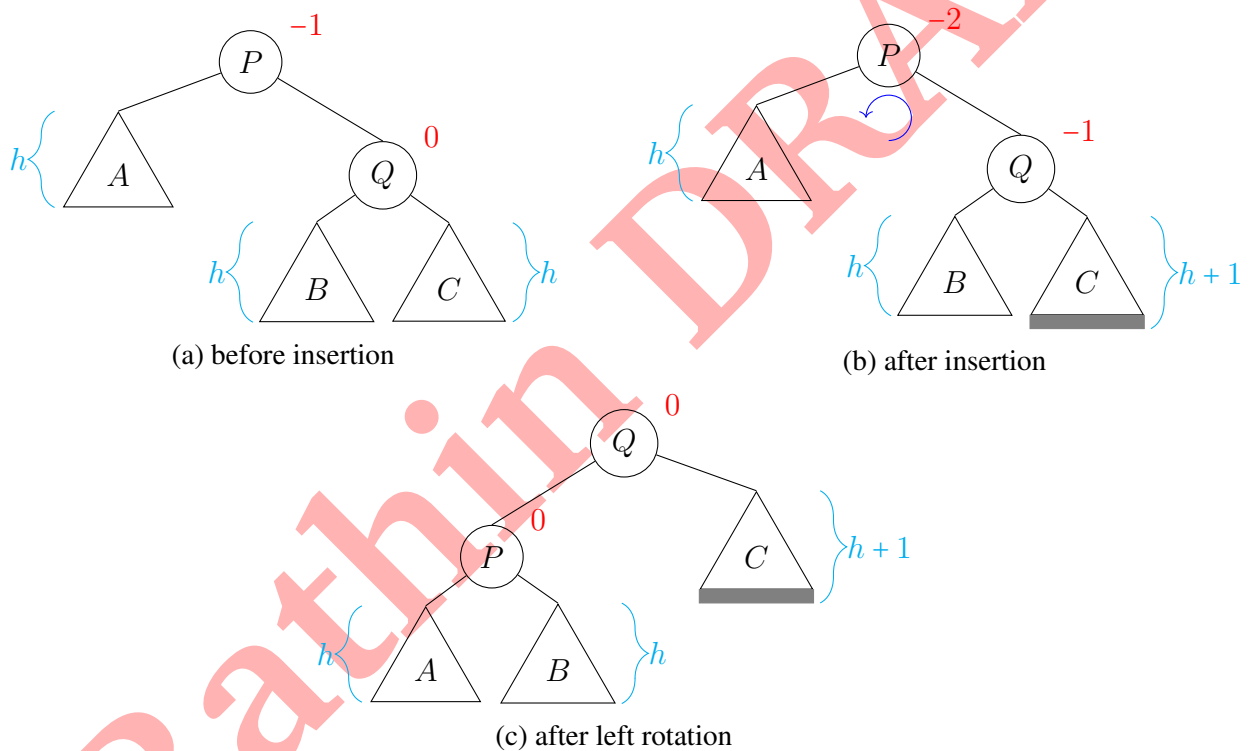


Figure 18: Scenario of a RR insertion and its rebalancing

**Case LR (left child’s right subtree)**

Suppose we insert an element into the right subtree ( $B$ ) of the left child ( $Q$ ) of a node ( $P$ ) and  $P$  is the nearest ancestor of the new node to get a balance factor outside of  $\{-1, 0, +1\}$

Let us consider that the root of the subtree  $B$  is  $R$  and its two subtrees are  $B_L$  and  $B_R$ . We assume for now that the new element is inserted into  $B_L$ . Let us now analyze balance factors of the nodes before and after the insertion happened. Since  $P$  is the first ancestor whose balance factor does not belong to

$\{-1, 0, +1\}$  after the insertion into an already balanced tree, the only possibility is that balance factor of  $P$  has changed from  $+1$  to  $+2$  due to the insertion into left subtree of  $P$ . Therefore, the insertion must have increased the height of  $B_L$  by one, which in turn increased the height of the subtree  $B$  and in turn increased the height of the subtree rooted at  $Q$  (left subtree of  $P$ ). Thus balance factor of  $Q$  must have changed from  $0$  to  $-1$ , and balance factor of  $R$  must have changed from  $0$  to  $+1$ . If we assume height of  $B_L$  was  $h$  before the insertion, we the heights of  $B_R$ ,  $A$ ,  $C$  must be  $h$ ,  $h + 1$ , and  $h + 1$  respectively. This is depicted in figure 19.

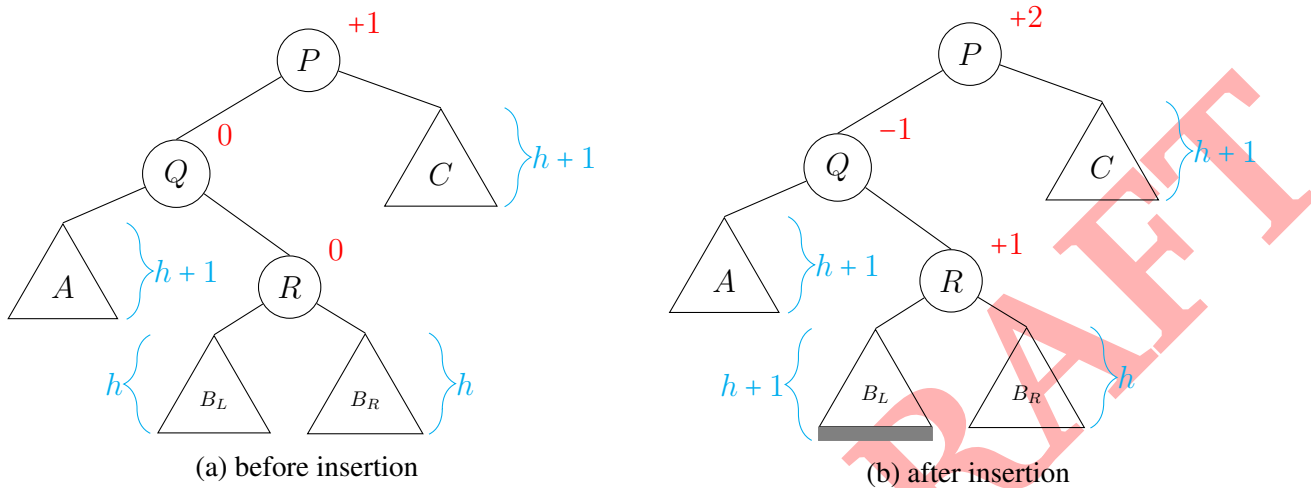
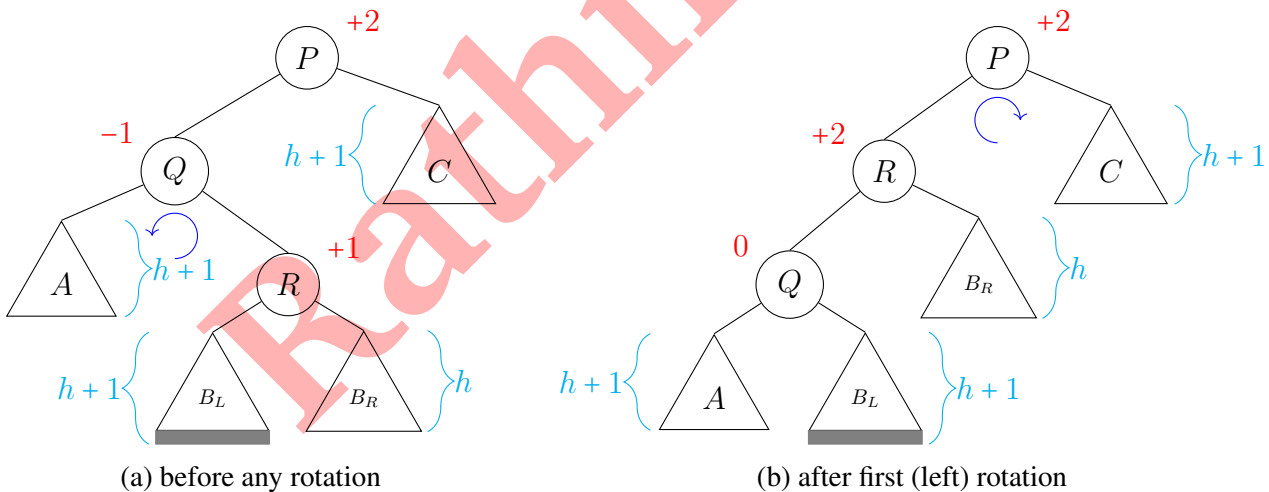


Figure 19: A scenario when a LR insertion requires rebalancing

In this situation, if we apply a left rotation at  $Q$  we get the scenario of 20b, which is a LL case from perspective of  $P$ . This can be taken care of by a right rotation at  $P$ . Note that, although here  $P$  not the nearest ancestor of the newly added node, we are still applying the right rotation in this artificially created LL case. The scenario after applying a right rotation at  $P$  is presented in figure 20c.



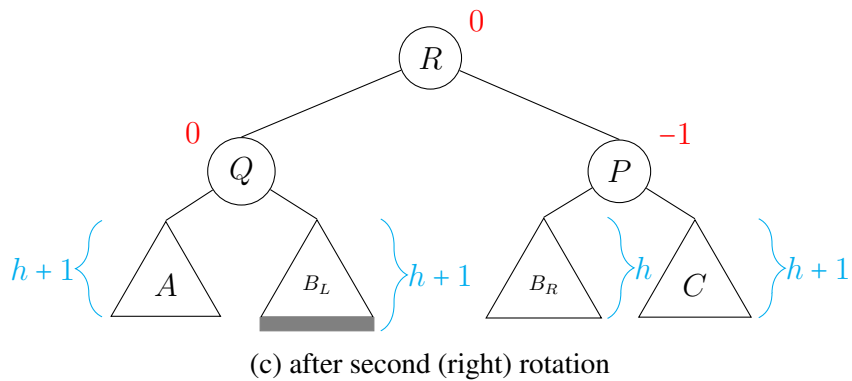


Figure 20: Rebalancing a LR case

Figure 21a depicts the scenario when the new node is inserted into  $B_R$  instead of  $B_L$ . Figure 21b is obtained after applying a left rotation (first rotation) at  $Q$  while the tree in figure 21c is obtained after applying the second rotation (right rotation) at  $P$ . As evident, in both the cases the final tree obtained after applying the two rotations successively, are balanced BST with the newly added node.

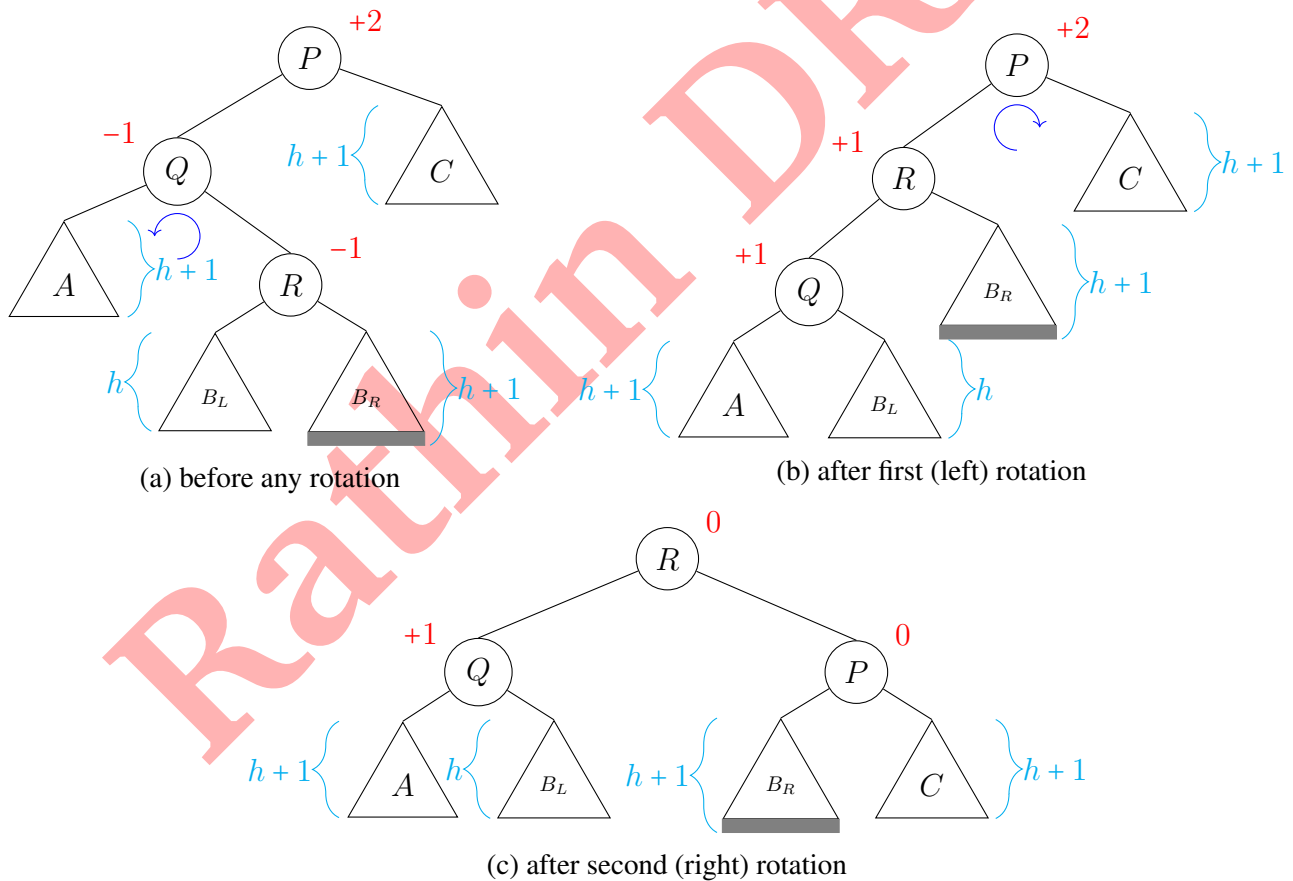


Figure 21: Rebalancing another LR case

### Case RL (right child's left subtree)

The mirror case of LR occurs when we insert an element into the left subtree ( $B$ ) of the right child ( $Q$ ) of a node ( $P$ ) and  $P$  is the nearest ancestor of the new node to get a balance factor outside of  $\{-1, 0, +1\}$ , i.e.  $bf = -2$ . Let us again consider that the root of the subtree  $B$  is  $R$  and its two subtrees are  $B_L$  and  $B_R$ . Figure 22 and figure 23 depicts the two RL insertion scenarios and subsequently they are dealt by the rotations. As evident, in each of two cases applying a right rotation at  $Q$  followed by a left rotation at  $P$  makes the resultant tree balanced.

Figure 22: Rebalancing a RL case

Figure 23: Rebalancing another RL case

## Exercise

- 5.1. Show that given  $n$  values, the BST(s) with minimum possible height has the lowest average search time among all possible BSTs with those  $n$  values.
- \*5.2. For an AVL tree with  $n$  nodes whose balance factor lies in  $\{-1, 0, +1\}$ , the height  $h$  lies in the interval  $\log_2(n+1) - 1 \leq h < \log_\varphi(n+2) + b$ , where  $\varphi$  is the golden ratio and  $b \approx -1.3277$ .
- 5.3. Formally show that a left rotation can correctly balance a RR case, where the BST has become unbalanced due to an insertion. Also show that the BST property is maintained in the resultant tree.
- 5.4. Show that the balanced trees obtained in figure 20c and 21c after applying the two rotations successively are indeed BSTs with the newly added node.
- 5.5. Formally show that a right rotation followed by a left rotation can correctly balance a RL case, where the BST has become unbalanced due to an insertion. Also show that the BST property is maintained in the resultant tree.

## 6 Heap

### References

- [1] A. Aho, J. Hopcroft, and J. Ullman. "Data Structures and Algorithms". In: 1983.
- [2] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844.
- [3] Ellis Horowitz, Sartaj Sahni, and Sanguthevar Rajasekaran. *Computer Algorithms*. 2nd. USA: Silicon Press, 2007. ISBN: 0929306414.
- [4] Tim Roughgarden. *Online Courses*. URL: <http://timroughgarden.org/videos.html>.

- [5] Debasis Samanta. *CLASSIC DATA STRUCTURES, 2nd ed.* Prentice-Hall Of India Pvt. Limited, 2008. ISBN: 9788120337312. URL: <https://books.google.co.in/books?id=law2E-LPScIC>.

Rathin DRAFT