Data Structures and Algorithms 2

Prof. Ahmed Guessoum
The National Higher School of AI

Chapter 4

Trees

From Linear ADTs to …

- For input of large size, the linear access time of linked lists is prohibitive.
- In this chapter, we look at a simple data structure for which the average running time of most operations is O(log n), and some simple modification to get O(log n) in the worst case.
- ➔ *Binary Search Trees*
- *Trees* are very useful abstractions in computer science
- We will discuss their use in other, more general applications.

Aims of this chapter

- We will also see how trees are used to:
  - implement the file system of several popular OSs.
  - evaluate arithmetic expressions.
  - support search operations in $O(\log n)$ average time
  - refine these ideas to obtain $O(\log n)$ worst-case bounds.
  - implement these operations when the data are stored on a disk.

Formal Definition

A ***tree*** is a sequence of nodes.

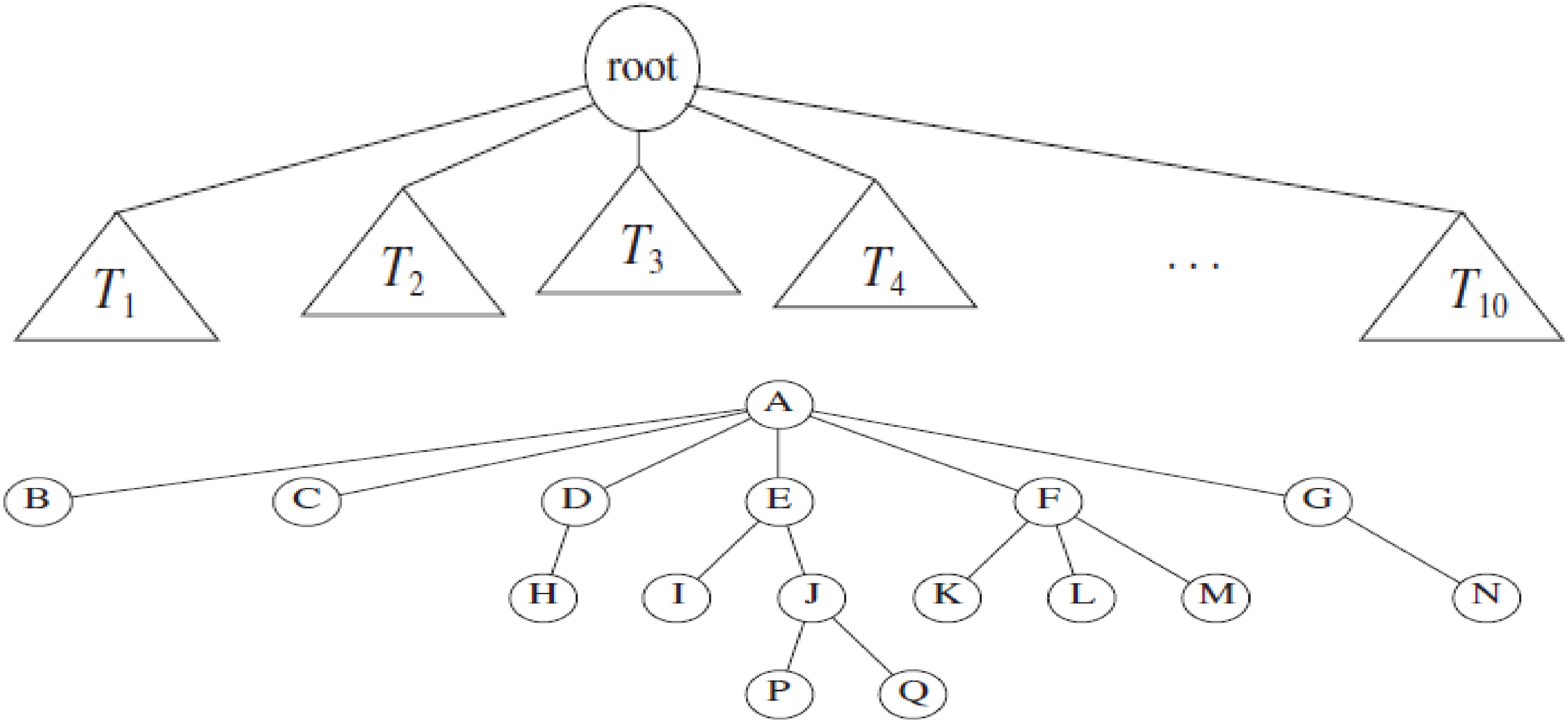
There is a starting node known as ***root*** *node*.

Every node other than the root has a **parent** node.

The nodes may have any number of ***children***, themselves being roots of *trees*.

A node that has no children is a ***leaf***

Illustration of a *tree* + terminology



A **root**; B, H, Q: **leaves**; I, J **children** of E; F **parent** of K, L, M; K, L, M: **siblings**; E **grandparent** of P; Q **grandchild** of E

More terminology

- A **path** from node n_1 to n_k is defined as a sequence of nodes $n_1, n_2, \ldots, n_k$ such that n_i is the parent of n_{i+1} for $1 \leq i < k$.
- The **length** of this path is the number of ***edges*** on the path, namely, $k - 1$.
- There is a ***path of length zero*** from every node to itself.
- In a tree there is exactly one path from the root to any node.
- For any node n_i, the **depth** of n_i is the length of the unique path from the root to n_i.

➔ the root is at depth 0.

More terminology

- The **height** of n_i is the length of the longest path from n_i to a leaf.

  ➔ all leaves are at height 0.

  ➔ height (tree) = height (root)

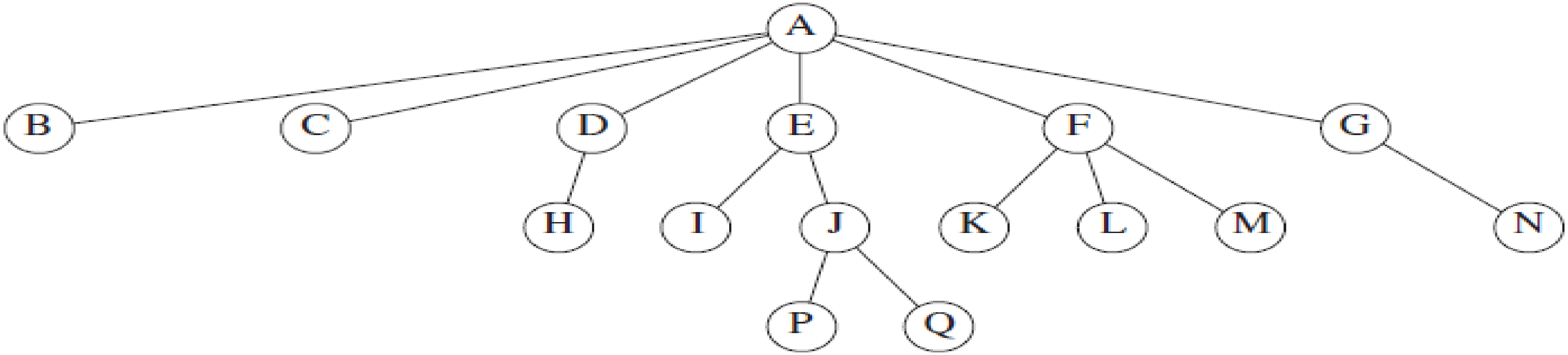
- If there is a path from n_1 to n_2, then n_1 is an **ancestor** of n_2 and n_2 is a **descendant** of n_1.
- If $n_1 \neq n_2$, then n_1 is a **proper ancestor** of n_2 and n_2 is a **proper descendant** of n_1.

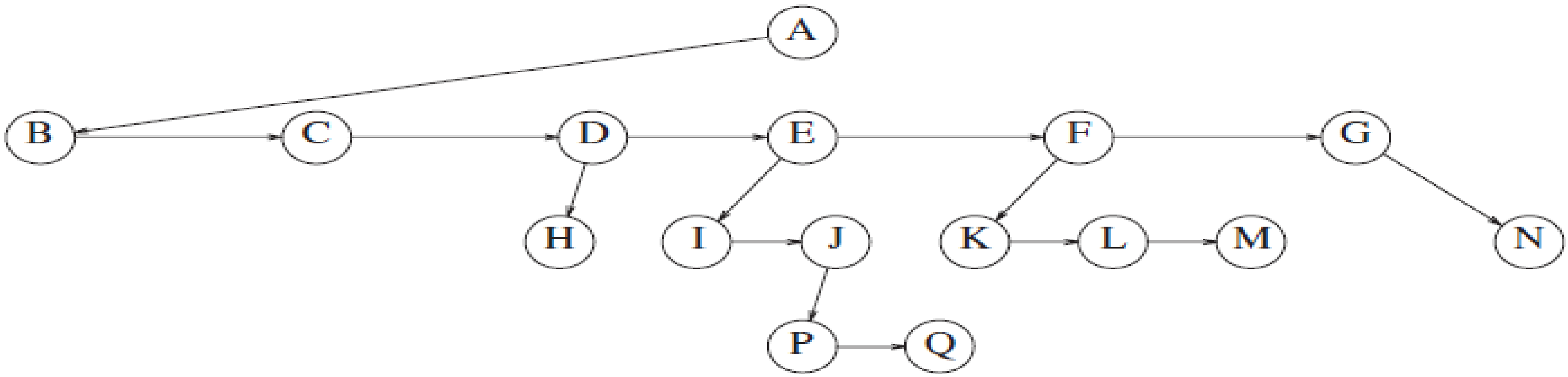
Implementation of Trees

- One intuitive approach: have in each node, besides its data, a link to each child of the node
- Can be very costly: number of children per node can vary greatly and is not known in advance
- Simple solution : Keep the children of each node in a linked list of tree nodes.

```
struct TreeNode
{
        Object element;
        TreeNode *firstChild;
        TreeNode *nextSibling;
};
```

First child/next sibling representation of a tree

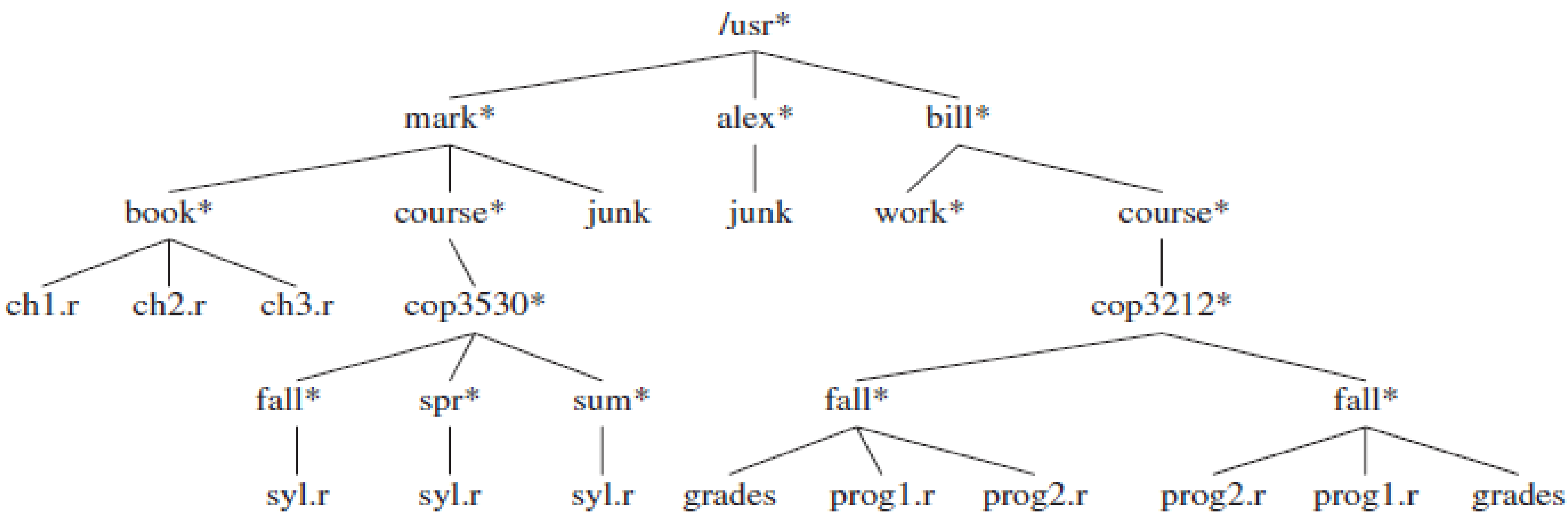




- Arrows that point downward are firstChild links.
- Arrows that go left to right are nextSibling links.
- Null links are not drawn (too many).
- Ex.: node *E* has both a link to a sibling (*F*) and a link to a child (*I*); some nodes have neither

Tree Traversal with Applications

<u>Application</u>: File system on Linux (or DOS)



/usr is the root directory

Filename "*/usr/mark/book/ch1.r*" is obtained by following the leftmost child 3 times.

Each "/" after the first indicates an edge;

The result is the full **pathname**

Pseudocode to list a directory in a hierarchical file system

```
void FileSystem::listAll( int depth = 0 ) const
{    // Preorder traversal to print directory filenames
        printName( depth ); // Print the name of the object
        if( isDirectory( ) )
                for each file c in this directory
                        c.listAll( depth + 1 );
}
```

- Prints all the names of files in the directory
- files at depth *di* will have their names indented by *di* tabs (function starts with depth = 0 ➔ no indent for root)
- Any children are one level deeper, and thus need to be indented an extra tab with respect to their parent.

```
/usr
    mark
        book
            ch1.r
            ch2.r
            ch3.r
        course
            cop3530
                fall
                    syl.r
                spr
                    syl.r
                sum
                    syl.r
        junk
    alex
        junk
    bill
        work
        course
            cop3212
                fall
                    grades
                    prog1.r
                    prog2.r
                fall
                    prog2.r
                    prog1.r
                    grades
```

Traversal strategies

- **Preorder traversal**: process Root BEFORE processing children from left to right (recursively)

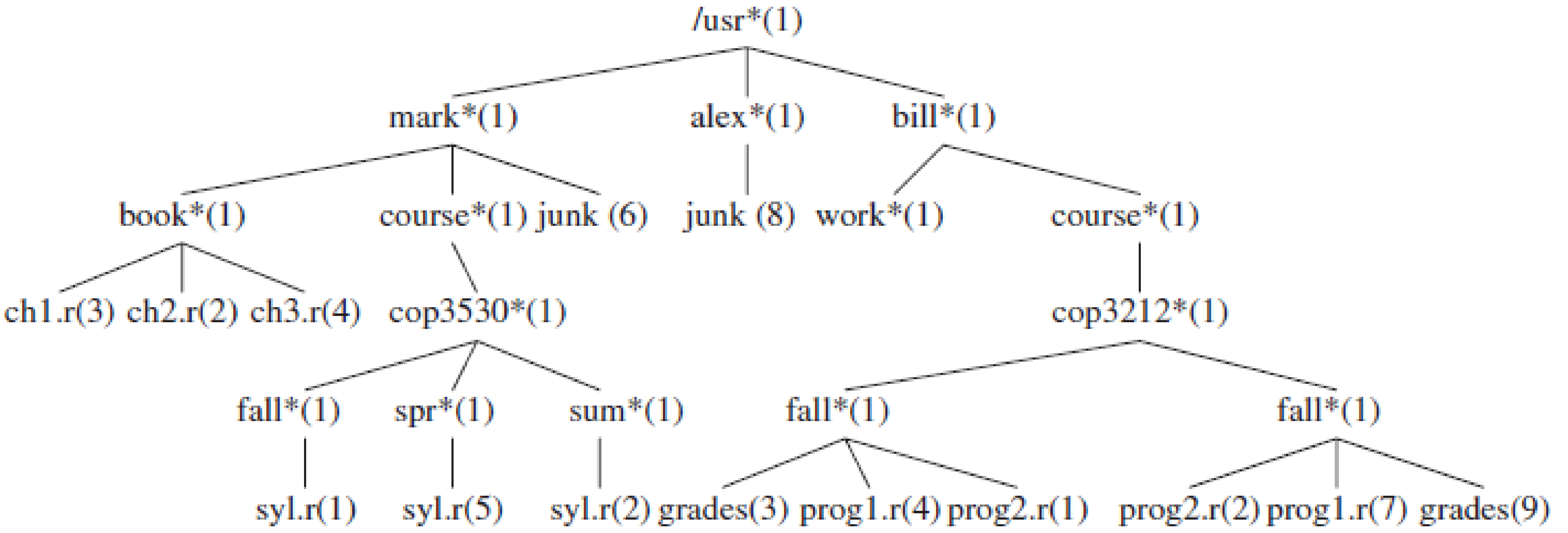
```
void FileSystem::listAll( int depth = 0 ) const
{ // Preorder traversal to print directory filenames
1       printName( depth ); // Print object name
2       if( isDirectory( ) )
3           for each file c in this directory
4                   c.listAll( depth + 1 );
}
```

- Analysis:
  - Line 1 executed exactly once per node; same for line 2
  - Line 4 executed at most once for each child of each node
  - The loop (Line 3) is iterated at most N times

  ➔ **O(N)** where N is the number of file names in the directory

Traversal strategies

- **Postorder traversal**: process Root AFTER processing children from left to right (recursively)
- Example:
  - Having a file directory with information about the number of disk blocks used in memory by each file
  - we would like to calculate the total number of blocks used by all the files in the tree
  - Compute the total number of blocks by adding up the numbers for each file/directory




```
int FileSystem::size( ) const
{    // Postorder traversal: total size of directory files
        int totalSize = sizeOfThisFile( );
        if( isDirectory( ) )
            for each file c in this directory
                totalSize += c.size( );
        return totalSize;
}
```

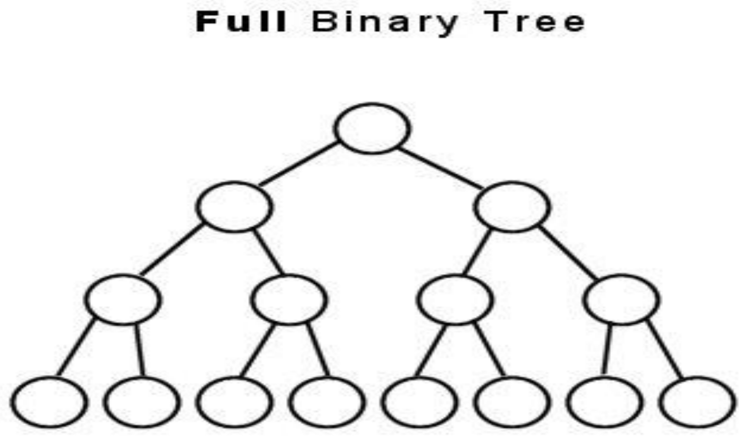
```
                  ch1.r                         3
                  ch2.r                         2
                  ch3.r                         4
          book                                 10
                                    syl.r       1
                        fall                    2
                                    syl.r       5
                        spr                     6
                                    syl.r       2
                        sum                     3
                  cop3530                      12
          course                               13
          junk                                  6
     mark                                      30
          junk                                  8
     alex                                       9
          work                                  1
                                    grades      3
                                    prog1.r     4
                                    prog2.r     1
                        fall                    9
                                    prog2.r     2
                                    prog1.r     7
                                    grades      9
                        fall                   19
                  cop3212                      29
          course                               30
     bill                                      32
/usr                                           72
```

Binary Trees

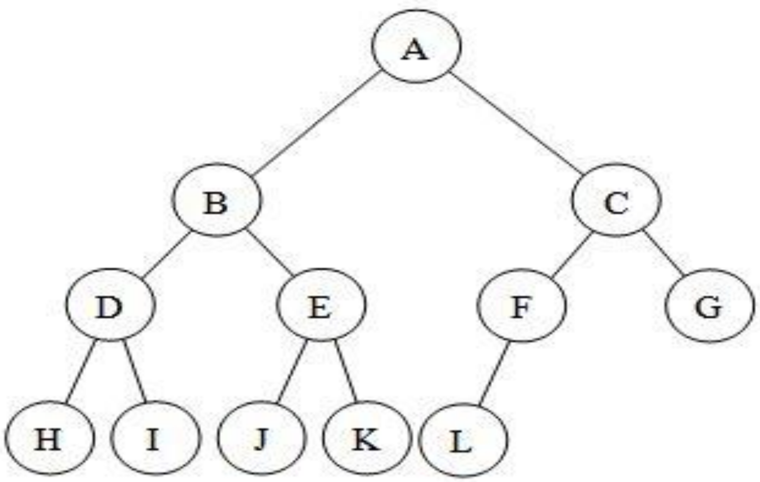
- A *Binary Tree* is a tree in which each node can have at most two children.
  - The depth of an average binary tree is generally considerably smaller than N.
  - An analysis shows that
    - the average depth of a binary tree is $O(\sqrt{N})$ (Exercise)
    - for a special type of binary tree, namely the *binary search tree*, average value of the depth is $O(\log N)$.
  - Unfortunately, the depth can be as large as $N - 1$ (Worst case: each node has exactly one child except leaf)

Different Types of BTs

- A *full binary tree* (sometimes proper binary tree or 2-tree) is a tree in which every node other than the leaves has two children



- A *complete binary tree* is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.



- Binary Search Trees will be presented later in this chapter

Complexity Analysis of CBT

A complete binary tree of N nodes has depth $O(\log N)$

Prove by induction that the number of nodes at depth d is 2^d

Total number of nodes of a complete binary tree of depth d is $1 + 2 + 4 + \ldots\ldots + 2^d = 2^{d+1} - 1$

Thus $2^{d+1} - 1 = N$

$d = \log(N + 1) - 1$

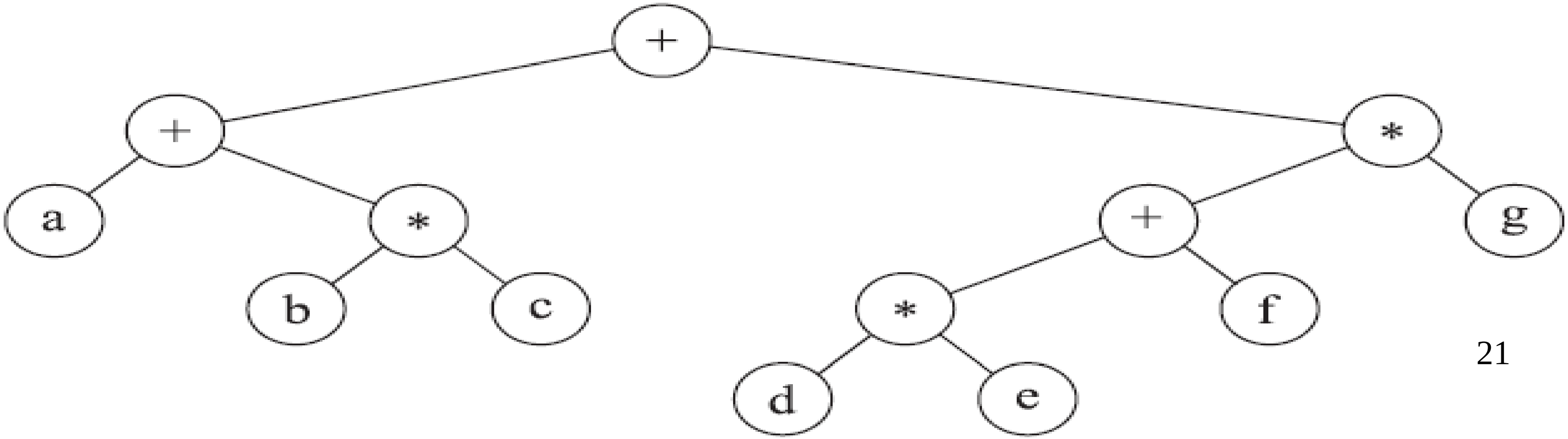
Implementation of a BT

```
struct BinaryTreeNode
{
        Object element;              // data in the node
        BinaryTreeNode *left;    // Left child
        BinaryTreeNode *right;  // Right child
 };
```

- Various interesting uses of BTs
- Next example in the area of Compiler Design

Expression Trees

- The leaves of an expression tree are **operands**, such as constants or variable names;
- The other nodes contain **operators**
- An ET is not necessarily binary:
  - e.g. case of unary operators (- and +)
  - Nodes may have more than 2 children: e.g. ternary operators.



Inorder/Postorder/Preorder Traversal

- **Inorder traversal strategy:** (left, node, right)
  - Produce an overly paranthesised expression by
    - recursively processing a parenthesized left expression
    - then printing out the operator at the root, and finally
    - recursively processing a parenthesized right expression.

  (a + (b * c)) + (((d * e) + f) * g)

- **Postorder traversal strategy:** (left subtree, right subtree, operator)

  a b c * + d e * f + g * + (postfix notation of Chapter 3)

- **Preorder traversal strategy:** (operator, left subtree, right subtree)

  + + a * b c * + * d e f g (prefix notation)

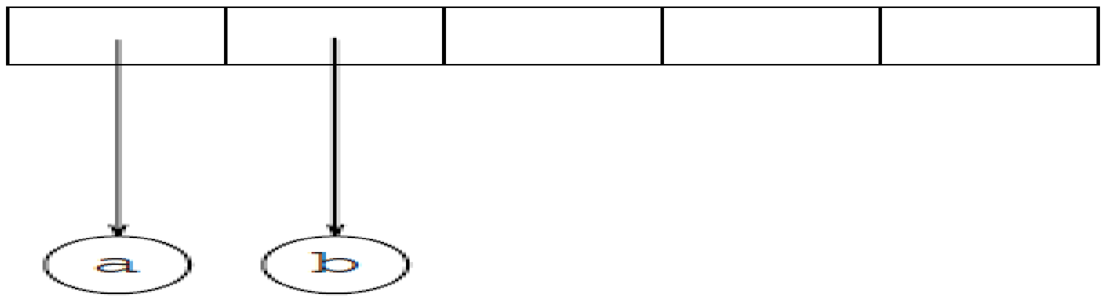
Constructing an ET

Algorithm to convert a postfix expression into an expression tree:

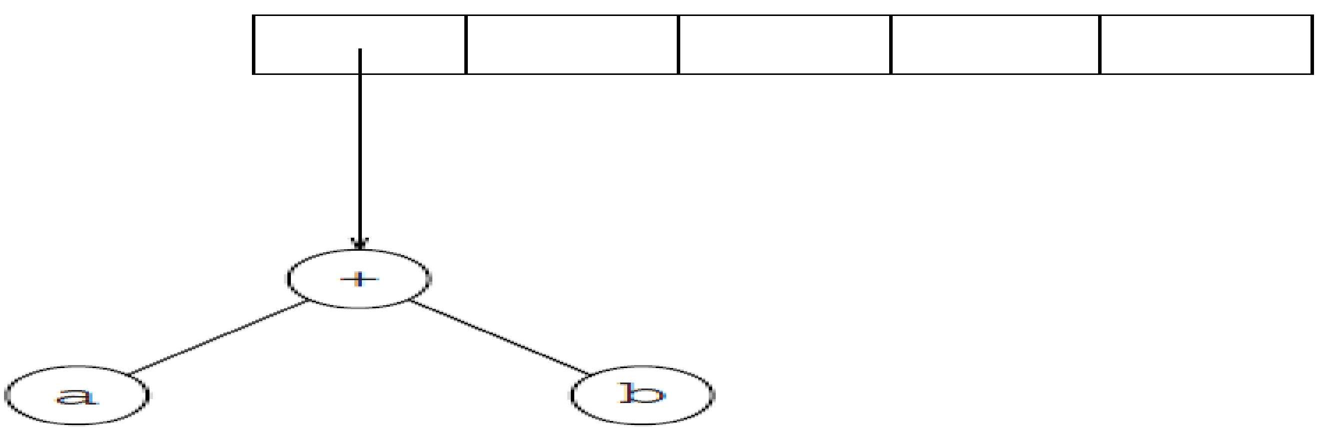
- Read the expression one symbol at a time.
- If the symbol is an operand, create a one-node tree and push a pointer to it onto a stack.
- If the symbol is an operator, pop (pointers) to two trees $T1$ and $T2$ from the stack ($T1$ is popped first) and form a new tree whose root is the operator and whose left and right children point to $T2$ and $T1$, respectively.
- A pointer to this new tree is then pushed onto the stack.

Example: input a b + c d e + * *

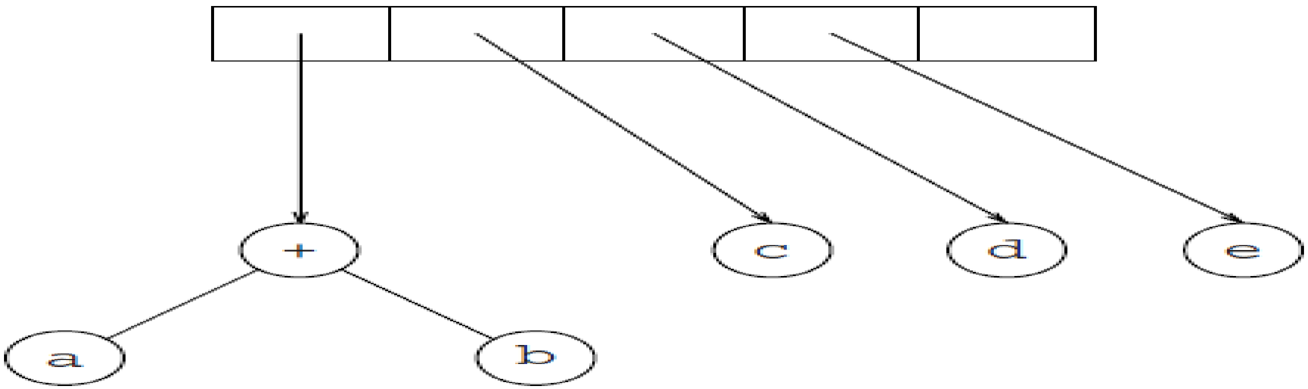
First two symbols are operands, so we create a one-node tree for each of them and push pointers to them onto a stack



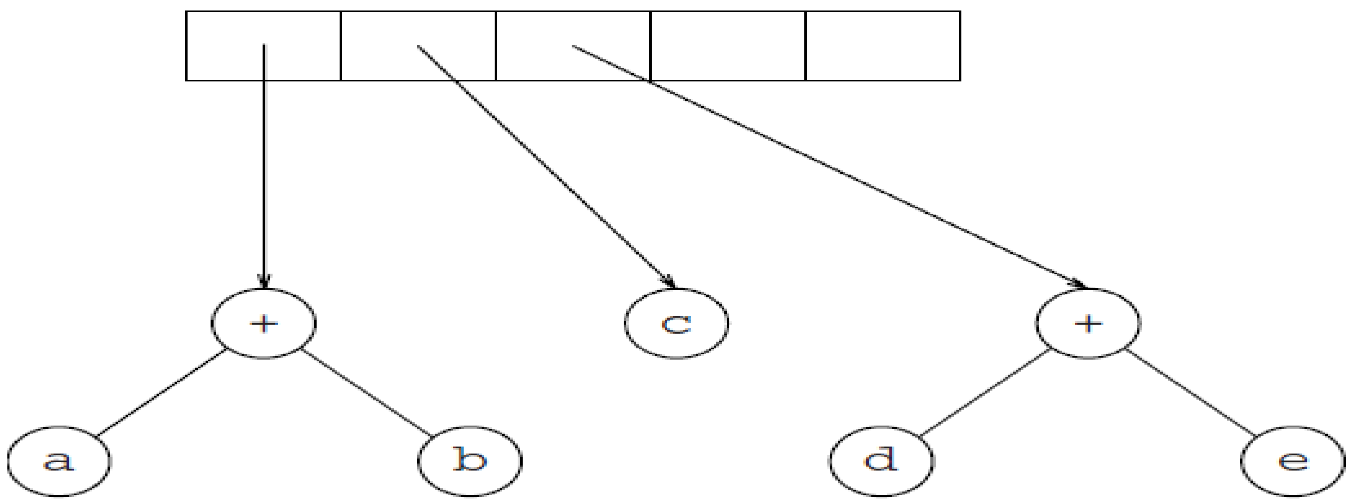
Next, + is read, so two pointers to trees are popped, a new tree is formed, and a pointer to it is pushed onto the stack.



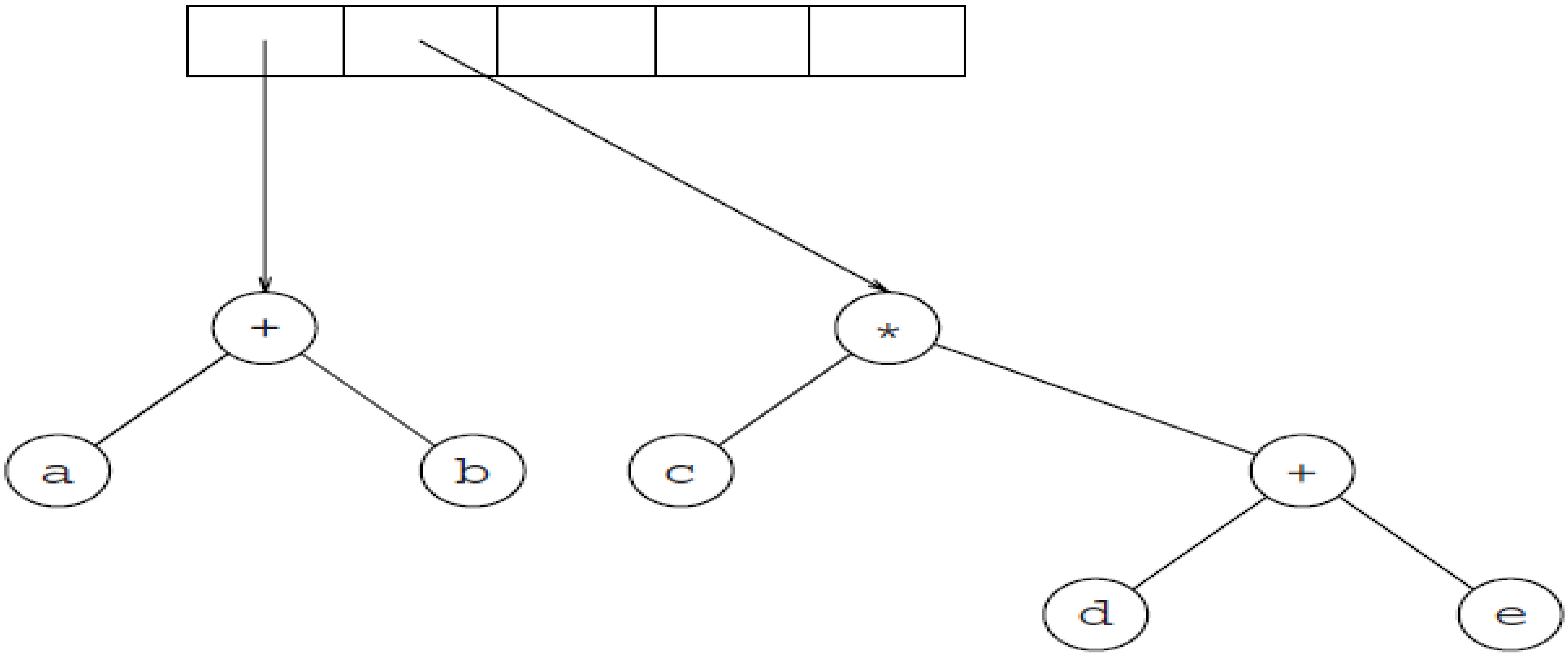
Next, c, d, and e are read, and for each, a one-node tree is created and a pointer to the corresponding tree is pushed onto the stack.



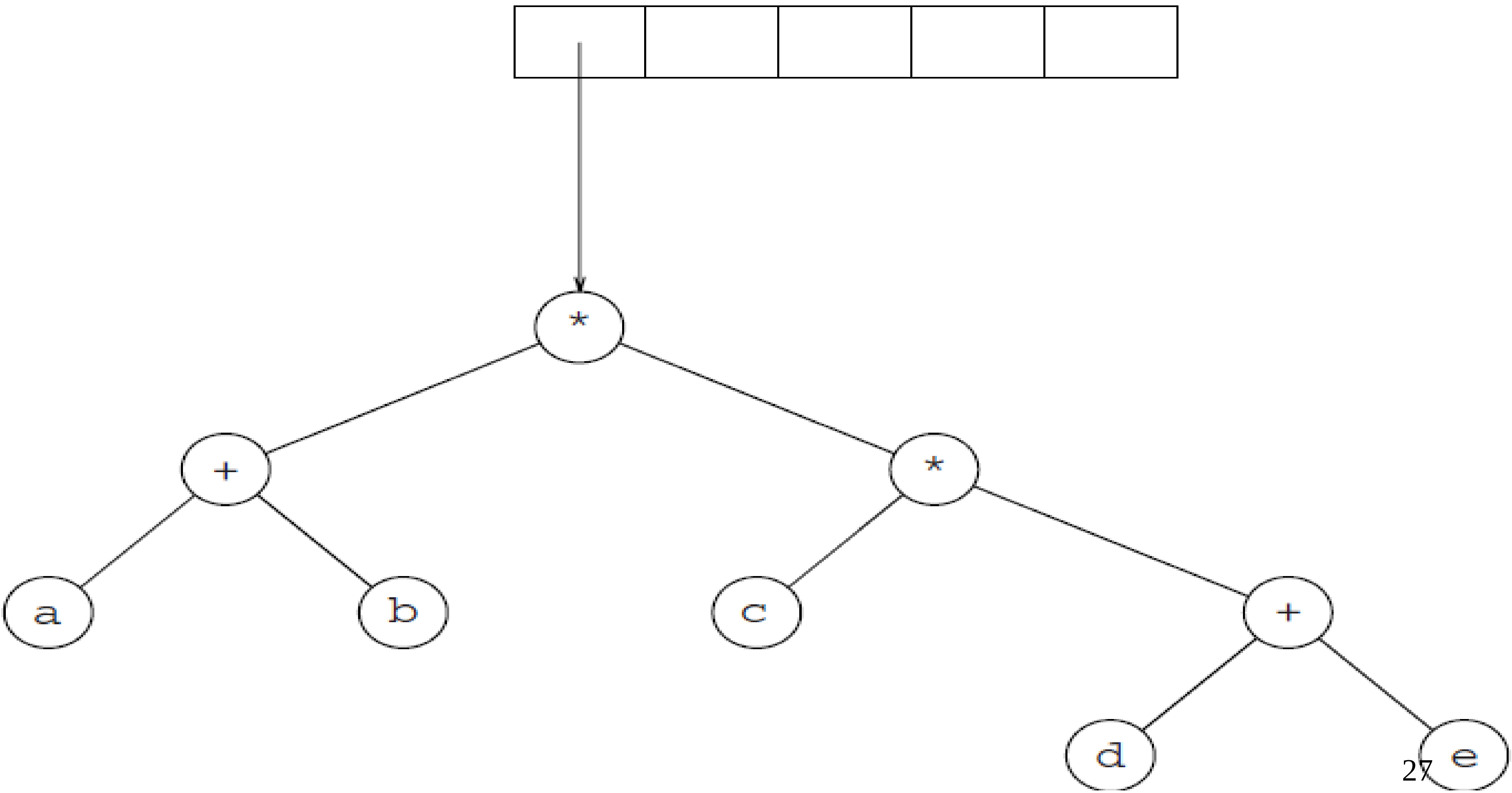
Now + is read, so two trees are merged.



* is read, so we pop two tree pointers and form a new tree with a * as root.



Finally, the last symbol * is read, the two trees are merged, and a pointer to the final tree is left on the stack.

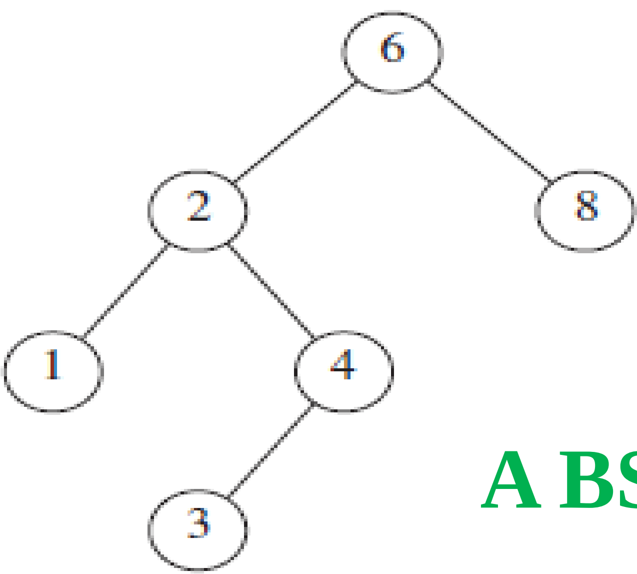


Binary Trees

- Important application of binary trees is their use in searching
- We will assume a tree of integers, though arbitrarily complex (nodes) elements are possible
- We will also assume that all the items are distinct (duplicates dealt with later)

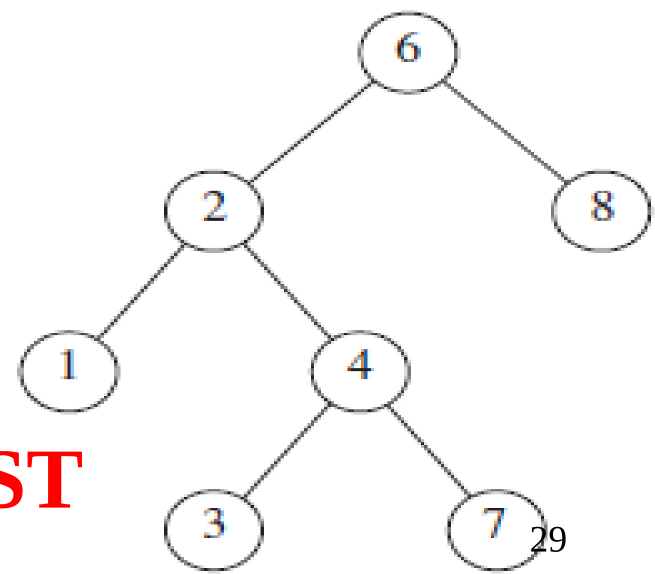
Binary Search Tree

- B*inary search tree* (BST): a BT where every node in the left subtree is less than the root, and every node in the right subtree is larger than the root.
- Properties of a BST are recursive



A BST

Not a BST



Operations on BSTs

- Implemented recursively
- Average depth of a binary search tree is O(log N)

→ no need to worry in general about running out of stack space

- The data member is a pointer to the root node; this pointer is nullptr for empty trees.

Code for Binary Search Tree Interface

Refresher: lvalue vs rvalue

```
std::vector<int> createVector() {
    std::vector<int> v{ 1, 2, 3, 4, 5 };
    return v;
}
int main() {
    std::vector<int> v1 = createVector();
                    // copy constructor
    std::vector<int>&& v2 = createVector();
                    // move constructor
    return 0;
}
```

Refresher: lvalue vs rvalue

- Function createVector() returns a vector of integers.
- In main(), we call createVector() twice: once to initialise v1 and once to initialise v2.
- When v1 is initialised, **copy constructor** is called:
  - creates a new vector,
  - copies contents of vector returned by createVector() into it.
- When v2 is initialised, **move constructor** is called:
  - moves contents of vector returned by createVector() in to v2.
  - Since original vector not needed anymore, this is more efficient than copying it.

Searching an element in a BST

Start from the root.

Each time we encounter a node, see if the key in the node equals the element. If yes stop.

If the element is less, go to the left subtree.

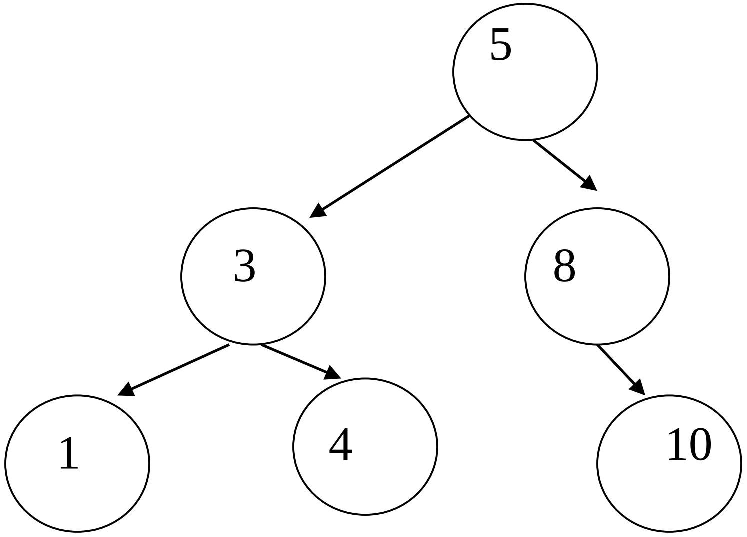
If it is more, go to the right subtree.

Conclude that the element is not in the list if we reach a leaf node and the key in the node does not equal the element.

```
Search(node, elt)
  {
      If (node = NULL) conclude NOT FOUND;
       Else If (node.key = elt) conclude FOUND;
       Else If (elt < node.key) Search(node.leftchild, elt);
       Else If (elt > node.key) Search(node.rightchild, elt);
    }
```

Complexity: **O(d)**, d is the depth of the element being searched for

For complete binary search trees: **O(log N)** where N is the number of nodes



Search for 10

Sequence Traveled:

5, 8, 10

Found!

Search for 3.5

Sequence Traveled:

5, 3, 4

Not found!

Find Min

- Returns a pointer to the node containing the smallest element in the tree
- Start at the root and
  - go left as long as there is a left child.
  - The stopping point is the smallest element

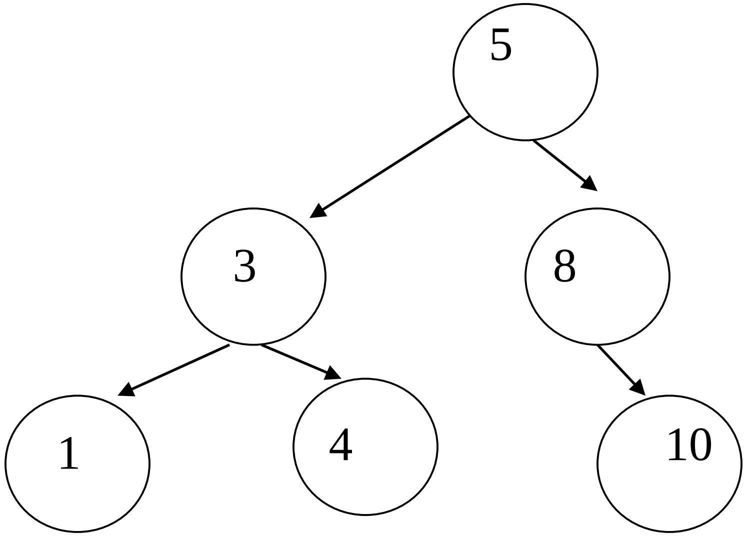
```
BinaryNode * findMin( BinaryNode *t ) const { // recursive
        if( t == nullptr )
                return nullptr;
        if( t->left == nullptr )
                return t;
        return findMin( t->left );                    }
```

Complexity: O(d)

Find Min

```
// non-recursive version

BinaryNode * findMin( BinaryNode *t ) const
{
        if( t != nullptr )
                while( t->left != nullptr )
                        t = t->left;
        return t;
}
```



Travel 5, 3, 1

Return 1;

Insert an element

Try to find the element;

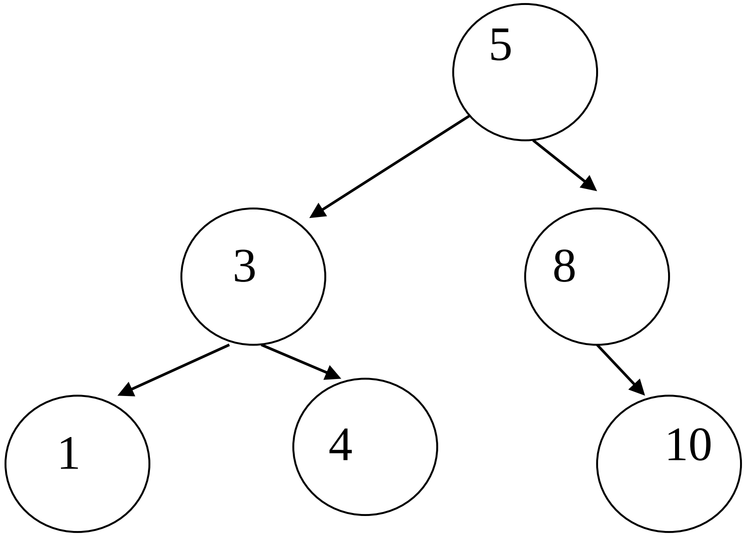
If the element exists, do nothing.

If it does not, insert it at the position of the returned null pointer;

Insertion function

```
void insert(const Comparable & x, BinaryNode * & t){
        if( t == nullptr )
            t = new BinaryNode{ x, nullptr, nullptr };
        else if( x < t->element )
                insert( x, t->left );
            else if( t->element < x )
                    insert( x, t->right );
                else
                    ; // Duplicate; do nothing
}
```

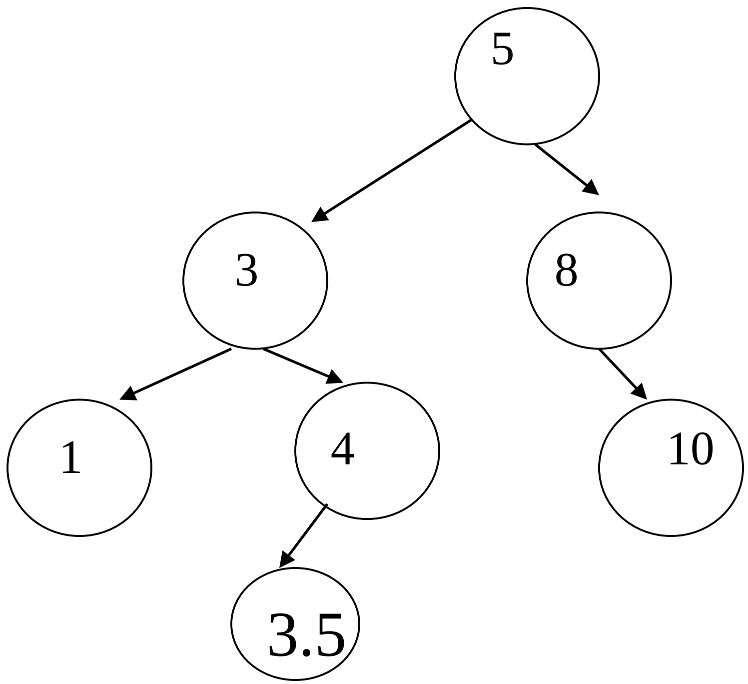
Complexity: O(d)



Insert 3.5

Sequence Traveled:

5, 3, 4



Insert 3.5 as left child of 4

Insert an element by moving

```
void insert( Comparable && x, BinaryNode * & t )  {
    if( t == nullptr )
        t = new BinaryNode{ std::move( x ), nullptr, nullptr };
  // std::move is exactly equivalent to a static_cast to
  // an rvalue reference type
    else if( x < t->element )
            insert( std::move( x ), t->left );
        else if( t->element < x )
                insert( std::move( x ), t->right );
            else
                ;     // Duplicate; do nothing
}
```

Complexity: O(d)

DELETION

Deleting a node, has to be done such that the property of the Binary Search Tree is maintained.

If the node has no child, **simply delete it**

If the node has only one child, **simply replace it with its child**

If the node has two children:

Look at the right subtree of the node (subtree rooted at the right child of the node).

Find the Minimum there.

Replace the key of the node to be deleted by the minimum element.

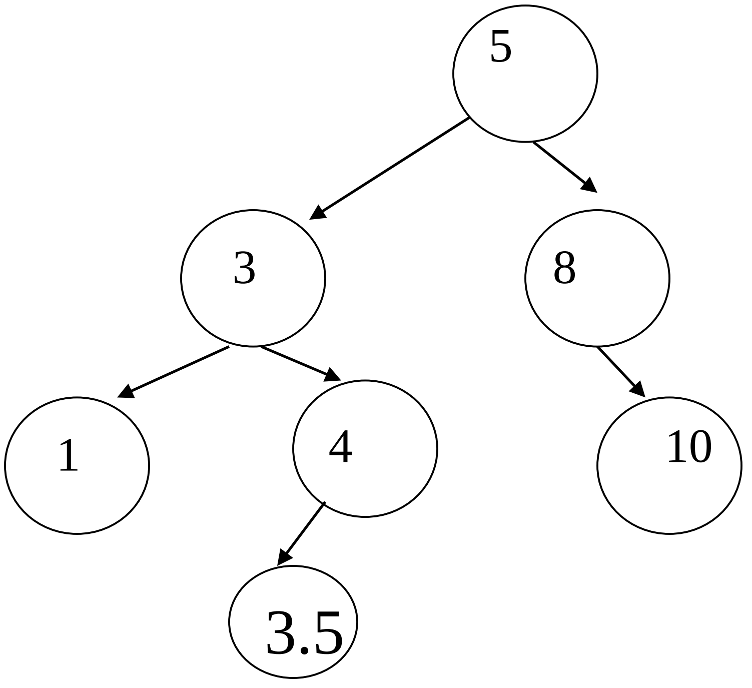
Delete the minimum element.

Any problem deleting it?

Need to take care of the children of this min. element,

(The min element can have at most one child.)

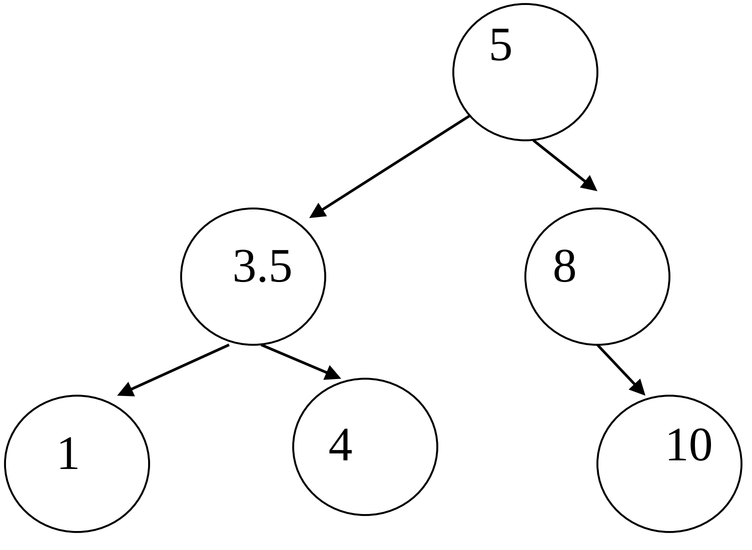
For deletion convenience, always have a pointer from a node to its parent.



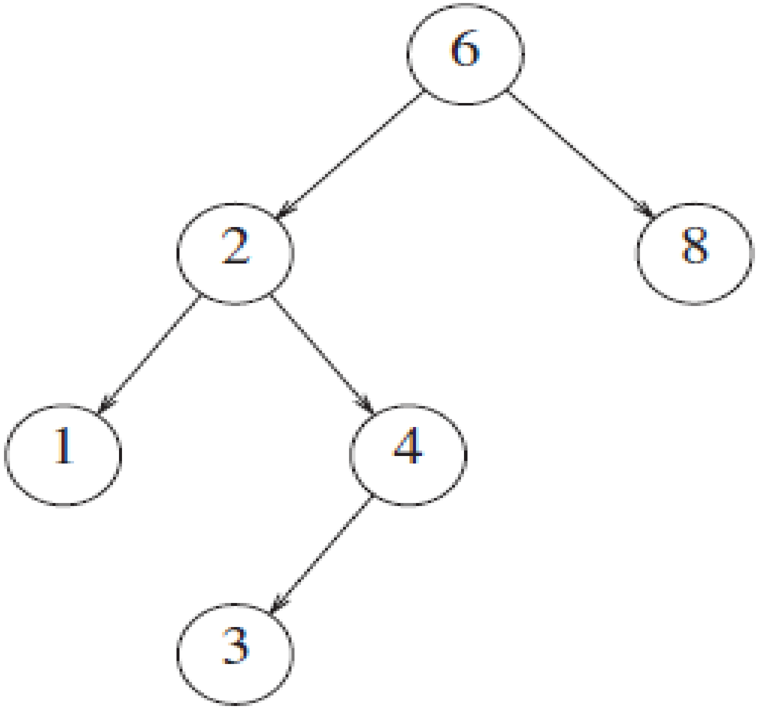
Delete 3;

3 has 2 children;

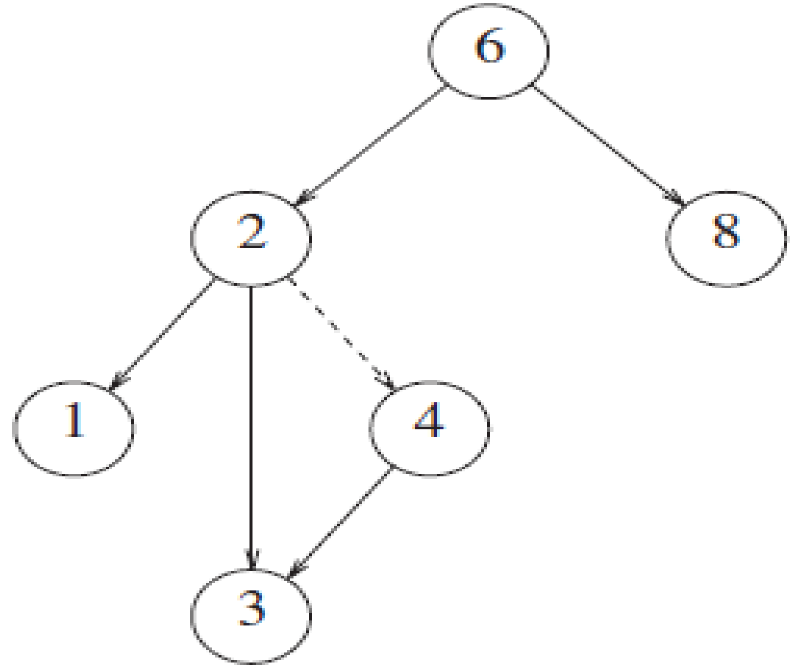
Findmin in right subtree of 3 returns 3.5



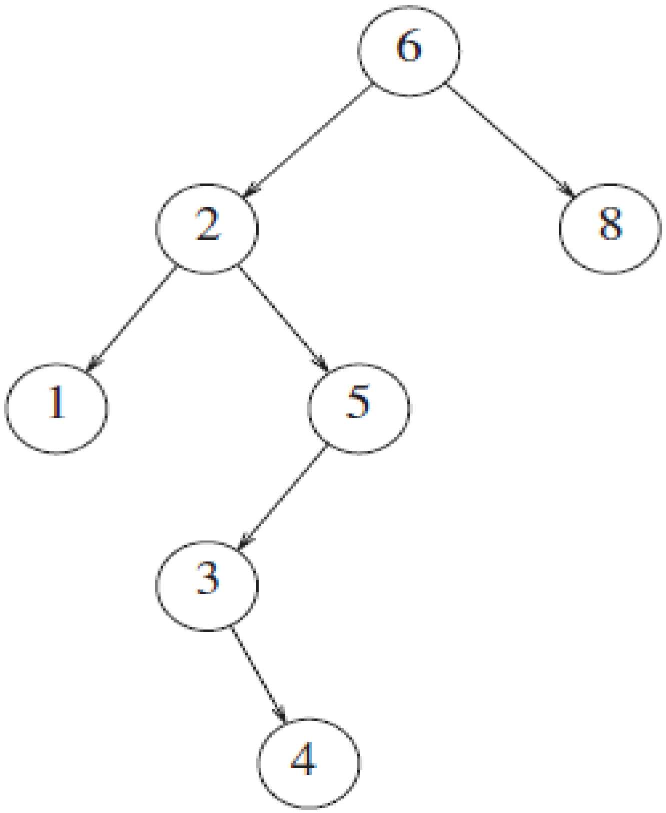
So 3 is replaced by 3.5, and 3.5 is deleted.

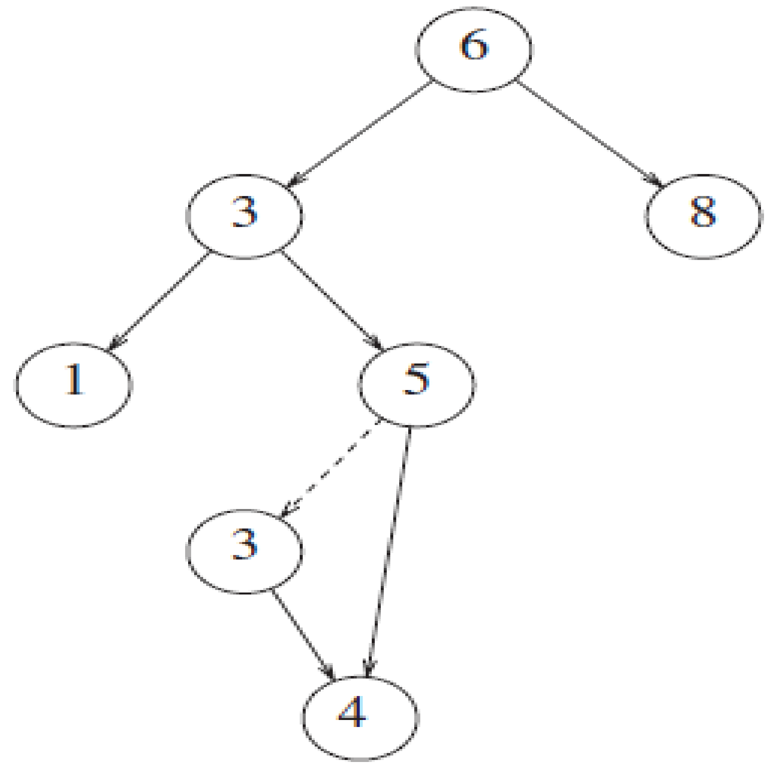


Before Delete 4
(with 1 child)



After Delete 4
(with 1 child)





Before Delete 2
(with 2 children)

After Delete 2
(with 2 children)

Convince yourselves with more sophisticated cases!

```
void remove( const Comparable & x, BinaryNode * & t ) {
    if( t == nullptr )
        return;                          // Item not found; do nothing
    if( x < t->element )
        remove( x, t->left );
    else if( t->element < x )
            remove( x, t->right );
        else if( t->left != nullptr && t->right != nullptr ) // Two children
{
                t->element = findMin( t->right )->element;
                remove( t->element, t->right );                    }
            else {
                BinaryNode *oldNode = t;
                t = ( t->left != nullptr ) ? t->left : t->right;
                delete oldNode;                          }      }
```

O(d)

Operations on BSTs: Code

Binary Search Tree Code

Average-Case Analysis for BSTs

- One expects: all operations on BSTs (except makeEmpty and copying) should take $O(\log N)$ time, because each time one subtree is traversed thus operating on a tree that is roughly half as large.
- Indeed running time (contains, insert, delete) is $O(d)$, where d is the depth of the node
- Assuming insertion sequences are equally likely, the average depth over all nodes in a tree is $O(\log N)$.

Average-Case Analysis for BSTs

- Definition: the **internal path length** is the sum of the depths of all nodes in a tree.
- What is the average internal path length of a BST, taken over all possible insertion sequences.
- Let $D(N)$ be the internal path length for some tree T of N nodes. $D(1) = 0$.
- An N-node tree consists of an i-node left subtree and an $(N - i - 1)$-node right subtree, plus a root depth zero for $0 \leq i < N$.
- $D(i)$ is the internal path length of the left subtree with respect to its root.

So $$D(N) = D(i) + D(N - i - 1) + N - 1$$

Average-Case Analysis for BSTs

- If all subtree sizes are equally likely, which is true for BSTs, then average value of both $D(i)$ and $D(N-i-1)$ is $\left(\frac{1}{N}\right)\sum_{j=0}^{N-1} D(j)$

So $\quad D(N) = \left(\frac{2}{N}\right)\left[\sum_{j=0}^{N-1} D(j)\right] + N - 1$

(Recurrence relation solved in Chapter 7: Sorting)

- It gives an average value of **$D(N) = O(N \log N)$**
- Because *Deletion* favours making the left subtrees deeper than the right, this does not imply that average running time of all the operations is $O(\log N)$
- If no deletions or BST roughly balanced, then average running times of the operations above are $O(\log N)$

AVL Trees

- We have seen that all operations depend on the depth of the tree.
- We don't want trees with large-height nodes
- This can be attained if both subtrees of each node have roughly the same height.

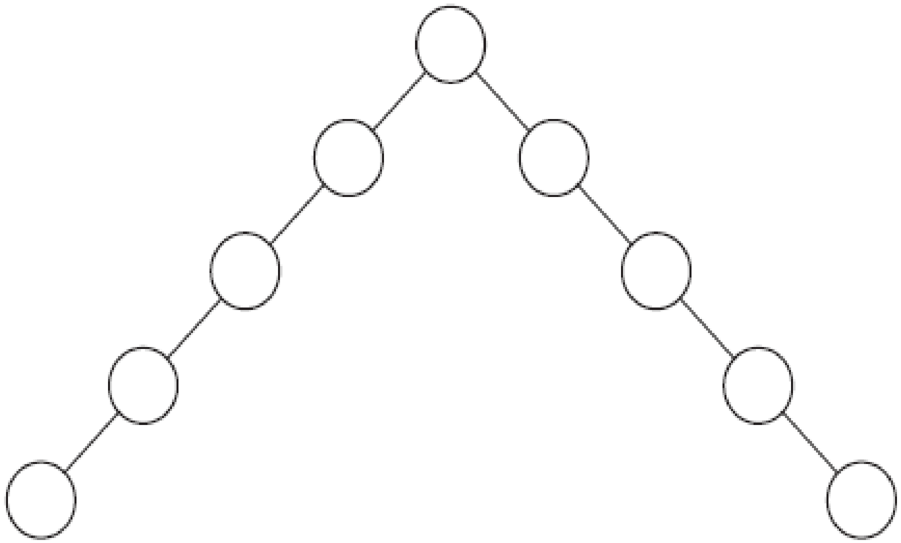
• An AVL (Adelson-Velskii and Landis) tree is a BST with a **balance condition**.

• The balance condition must be easy to maintain, and it ensures that the depth of the tree is $O(\log N)$.

• Simplest idea: require left and right subtrees have same height.

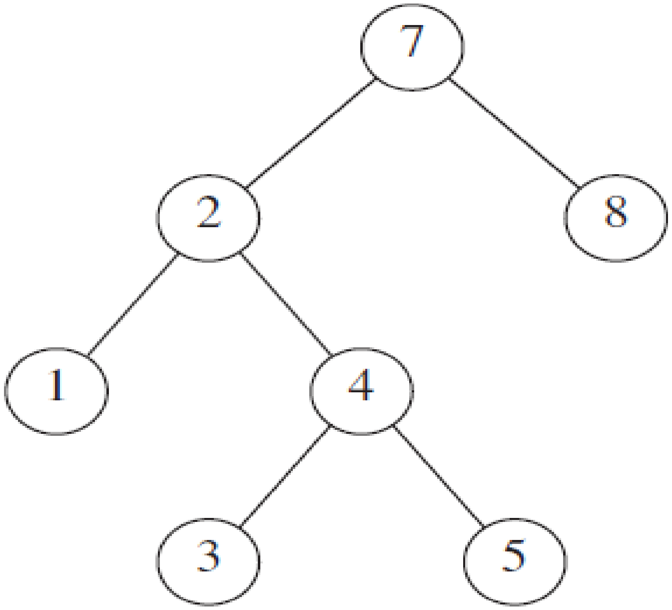
AVL Trees

Idea that left and right subtrees have roughly the same height does not force the tree to be shallow

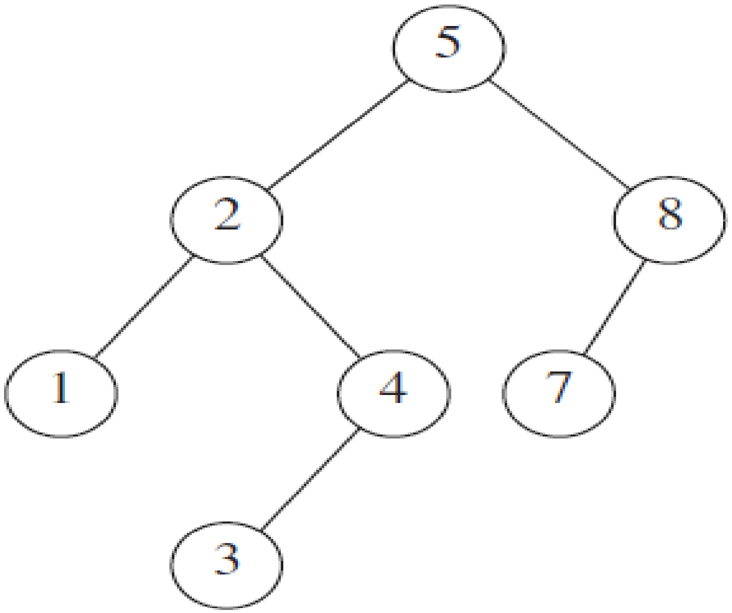


An AVL (Adelson-Velskii and Landis) tree is a BST where the heights of the two subtrees of any node differ by at most one

Height of a null tree is -1



Not AVL Tree



AVL Tree

Some AVL Tree Properties

- Height information is kept for each node (in the node structure).
- It can be shown that the height of an AVL tree is at most roughly $1.44 \log(N + 2) - 1.328$, but, in practice, only slightly more than $\log N$.
- all the tree operations can be performed in **$O(\log N)$** time, except insertion and deletion (need to update all the balancing information)

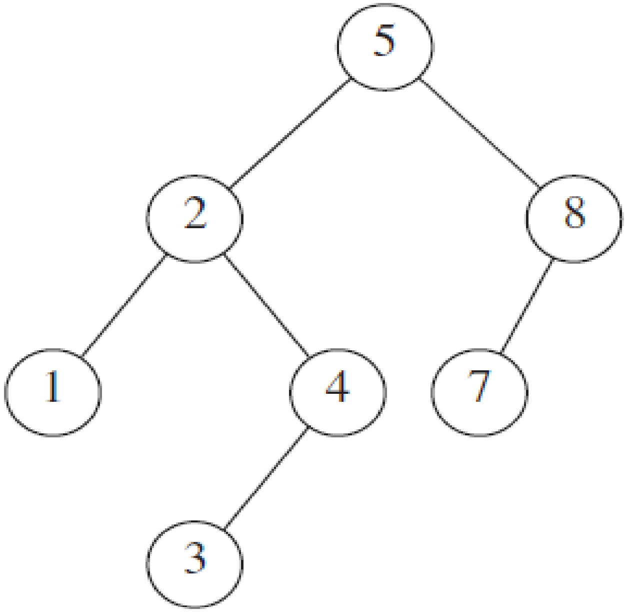
Operations on AVL Tree

Searching, Complexity? **O(log N)**

FindMin, Complexity? **O(log N)**

Deletion? Insertion?

Insertion into an AVL Tree



Insert 6

➔ Tree not AVL anymore

- The AVL property has to be restored before the insertion step is considered over.
- This can be done with a simple modification to the tree, known as a **rotation.**

Insertion into AVL Tree

- After insertion, only nodes on the path from insertion point to the root might have their balance altered because only those nodes have their subtrees altered.
- As we follow the path up to the root and update the balancing information, we may find a node whose new balance violates the AVL condition.
- So we will **rebalance** the tree at the first (i.e., deepest) such node
- This **rebalancing** guarantees that the entire tree satisfies the AVL property

Node rebalancing

- Let α *be* the node that must be rebalanced.
- Since any node has at most two children, It is easy to see that a violation might occur in four cases:

1. Insertion into the left subtree of left child of α
2. Insertion into the right subtree of left child of α
3. Insertion into the left subtree of right child of α
4. Insertion into the right subtree of right child of α

Note: 1 and 4 are mirror cases; same for 2 and 3

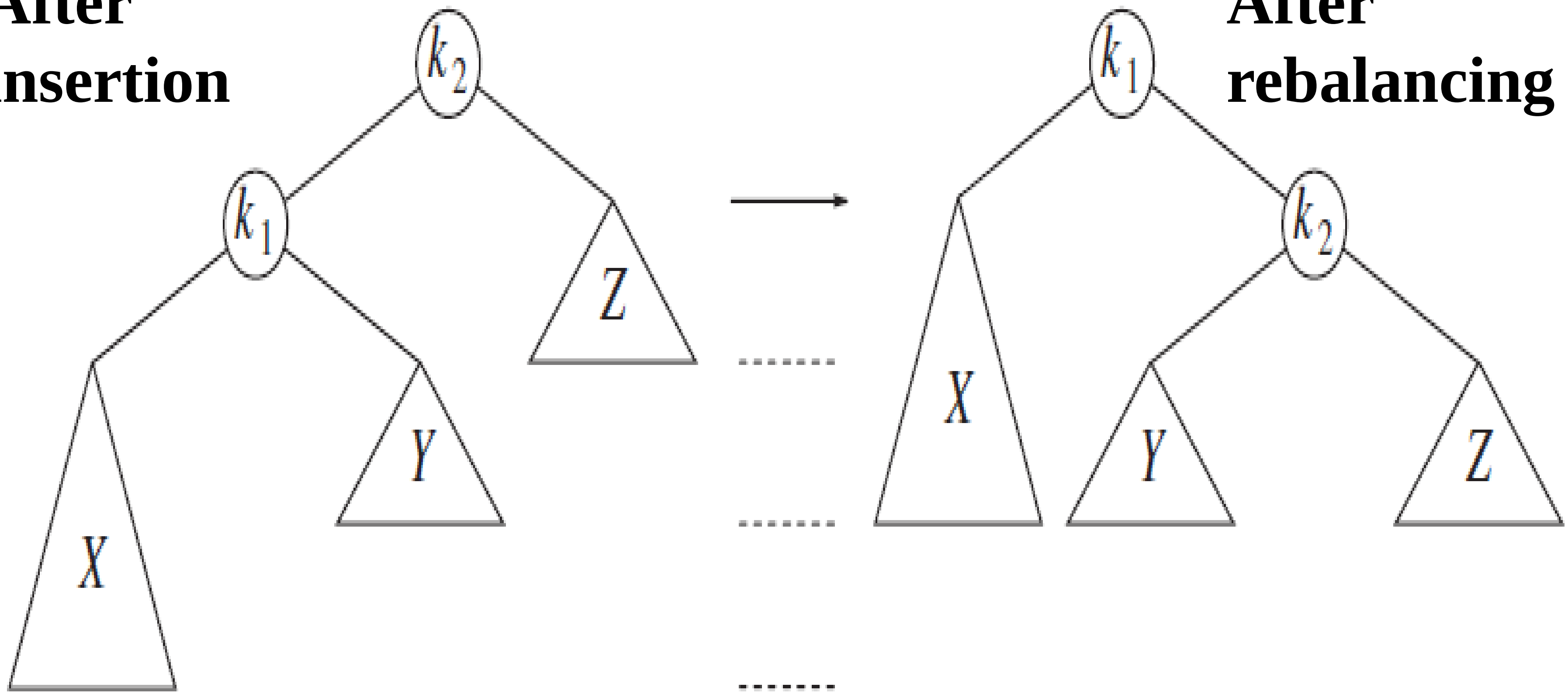
Rotations

- <u>First case</u>: insertion occurs on the "outside" (i.e., left–left or right–right); fixed by a **single rotation** of the tree.
- <u>Second case</u>: insertion occurs on the "inside" (i.e., left–right or right–left); handled by the slightly more complex **double rotation.**
- These are fundamental operations on the tree that we will see used several times in balanced-tree algorithms

Single Rotation

After insertion

After rebalancing

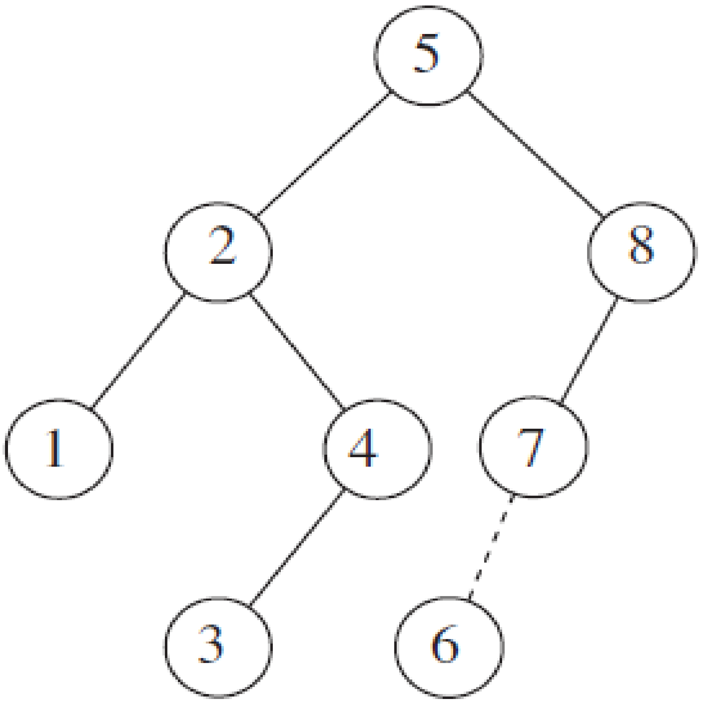


k_2 violates the AVL balance property because its left subtree is two levels deeper than its right subtree. (Generically: only possible case: Subtree *X* has grown to an extra level, causing it to be exactly two levels deeper than *Z*.)

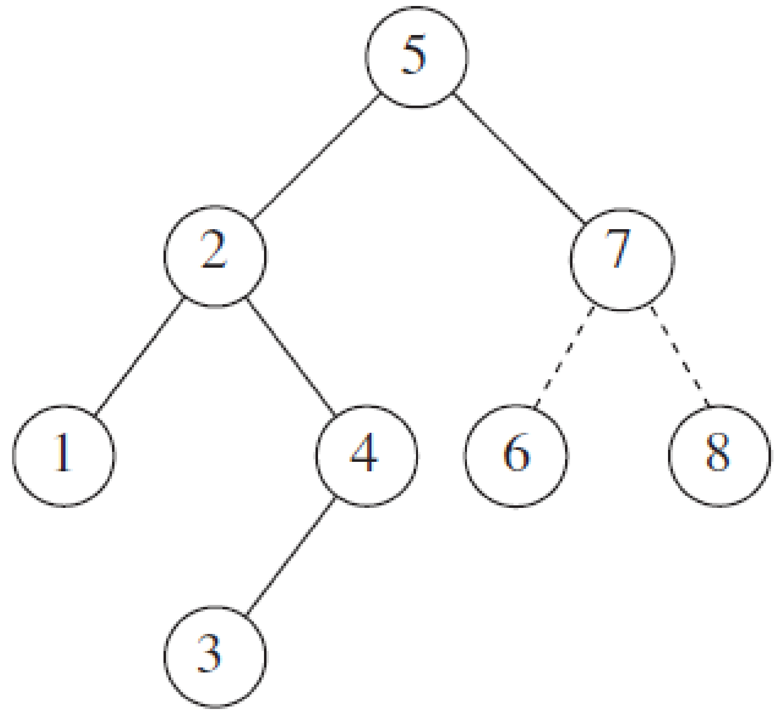
Result of Single Rotation

- Note that Single Rotation requires only a few pointer changes,
- We get another BST that is an AVL tree.
- This is because X moves up one level, Y stays at the same level, and Z moves down one level.
- k_2 and k_1 not only satisfy the AVL requirements, but they also have subtrees that are exactly the same height.
- ➔ no further updating of heights on the path to the root is needed, and consequently *no further rotations needed.*

Example of Single Rotation



After insertion of 6



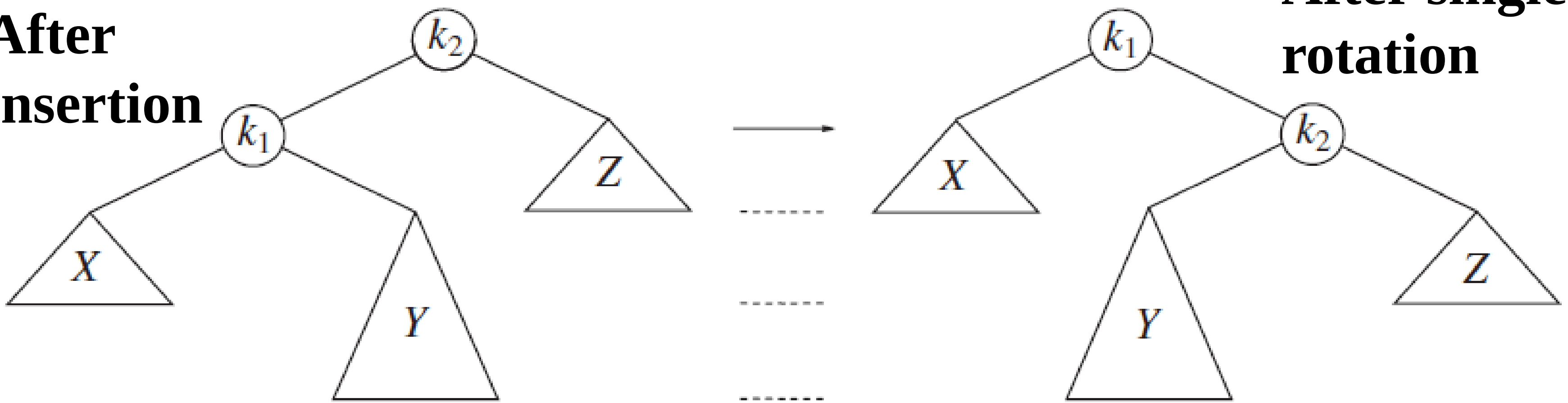
After rebalancing

Double Rotation

- Single Rotation does not work for cases 2 and 3 (in which the insertion has occured on the "inside", i.e., left–right or right–left of a node.

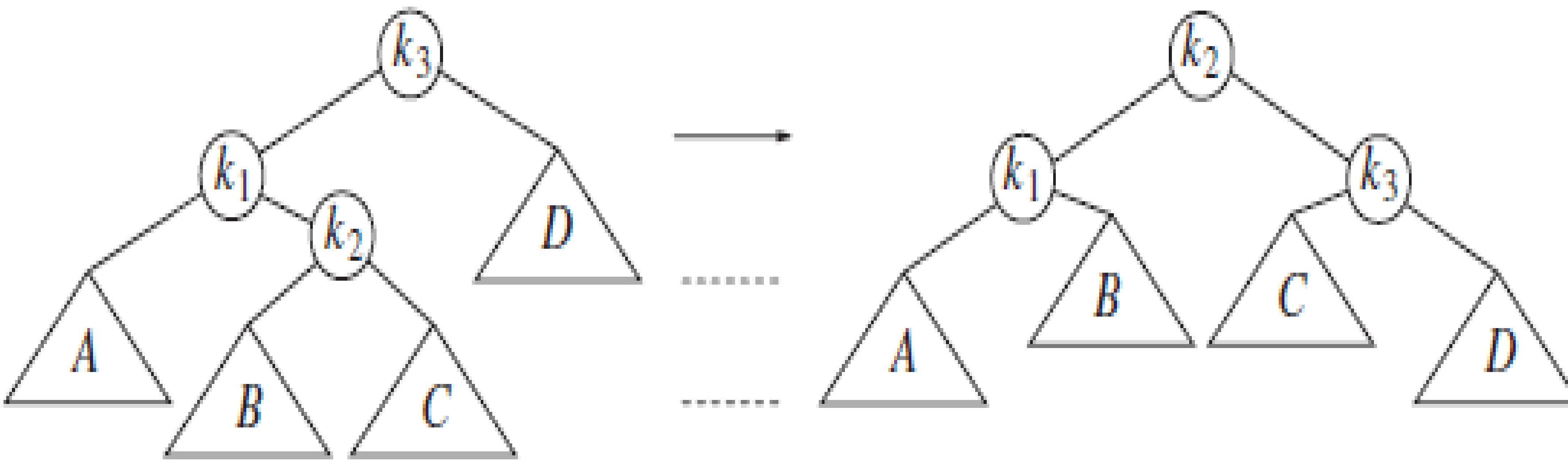
After insertion

After single rotation



- Item was inserted into subtree *Y*.
- One may assume it has a root and two subtrees.
- ➔ the tree may be viewed as four subtrees connected by three nodes.

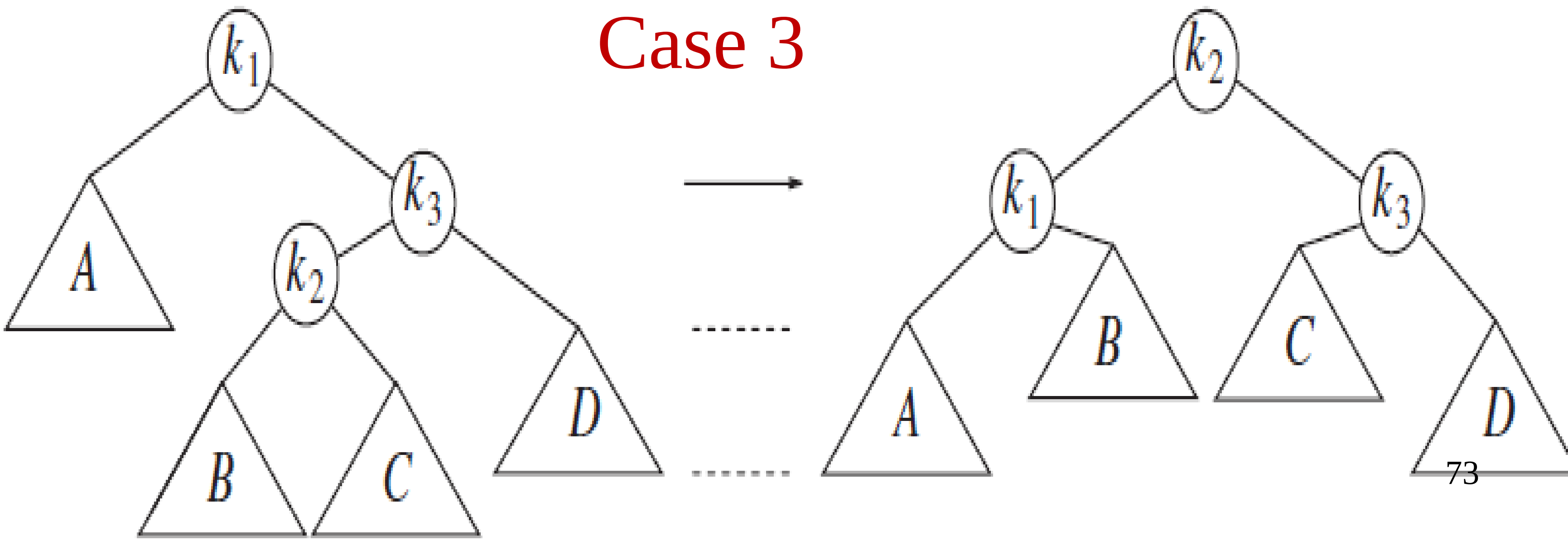
Case 2



After insertion

After double rotation

Case 3



Result of Double Rotation

It is easy to see that the resulting tree

- satisfies the AVL tree property, and
- restores the height to what it was before the insertion

➔ guarantee that all rebalancing and height updating is complete

Extended Example

Insert 3,2,1,4,5,6,7, 16,15,14

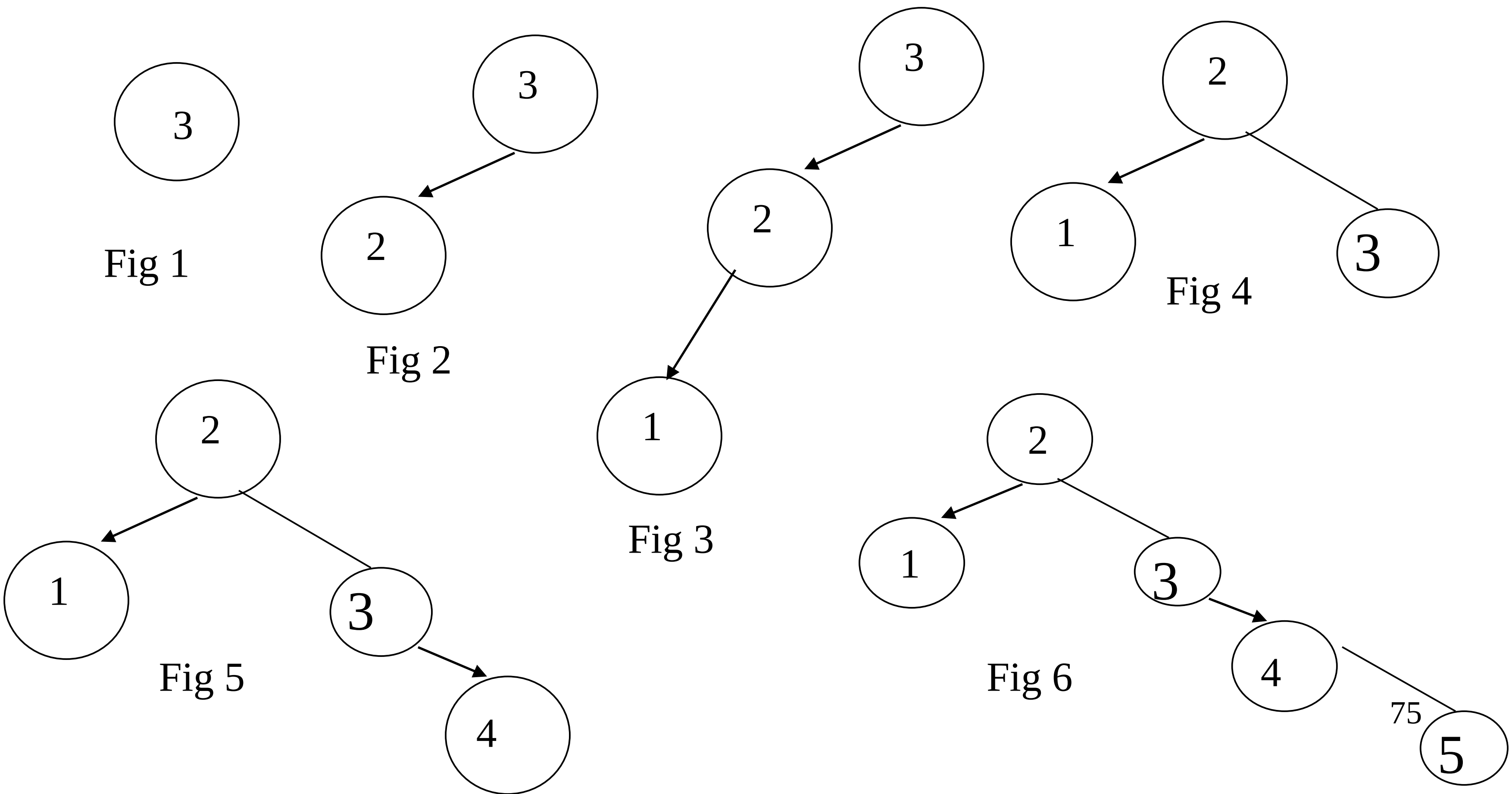


Fig 1 Fig 2 Fig 3 Fig 4 Fig 5 Fig 6

Insert 6,7, 16,15,14

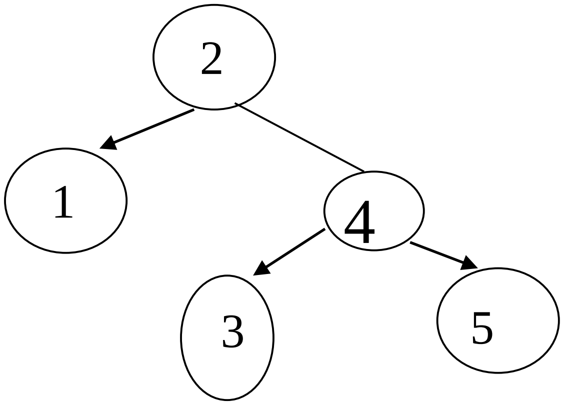


Fig 7

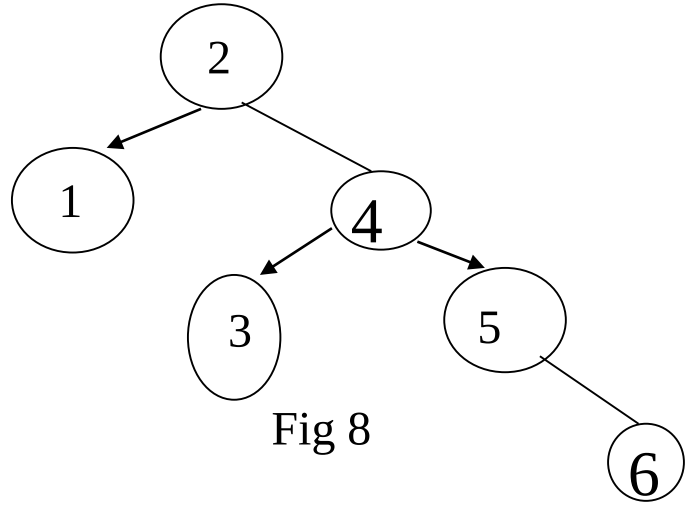


Fig 8

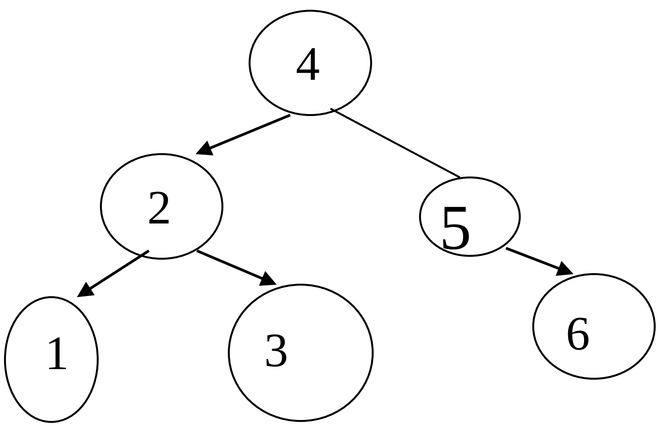


Fig 9

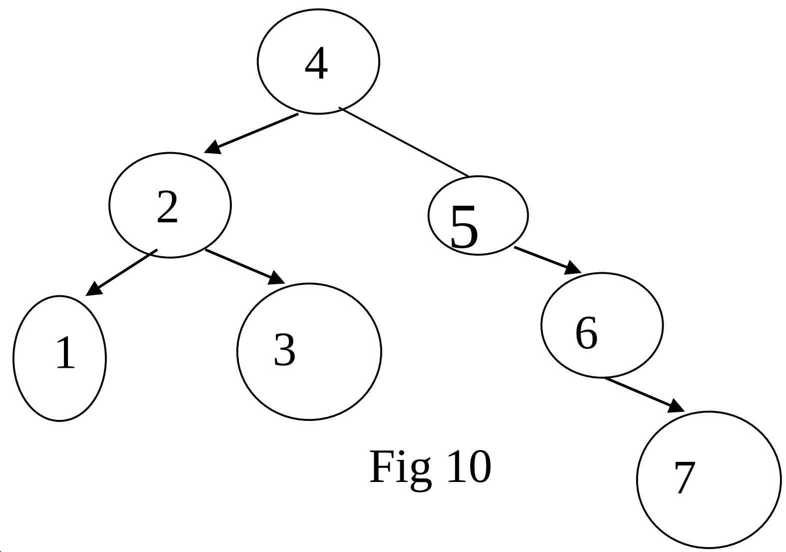


Fig 10

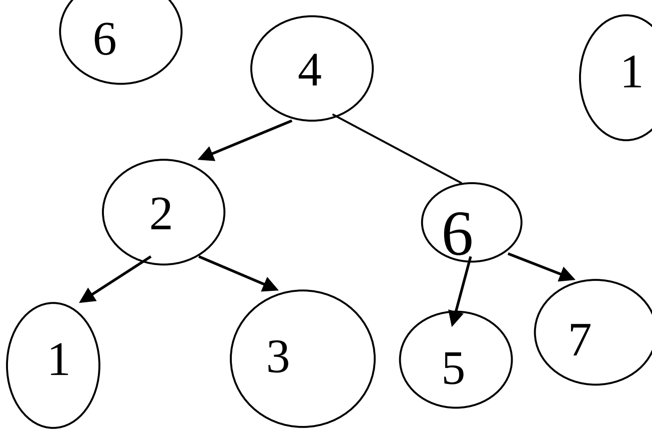


Fig 11

Insert 16,15,14

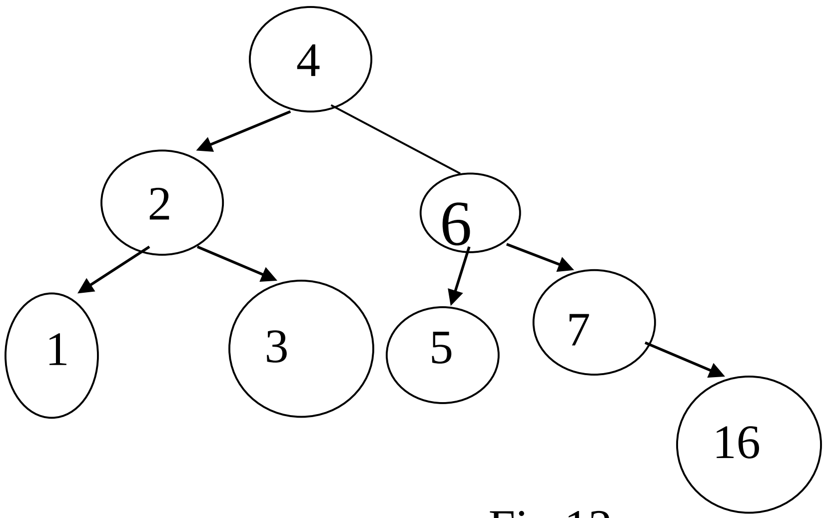


Fig 12

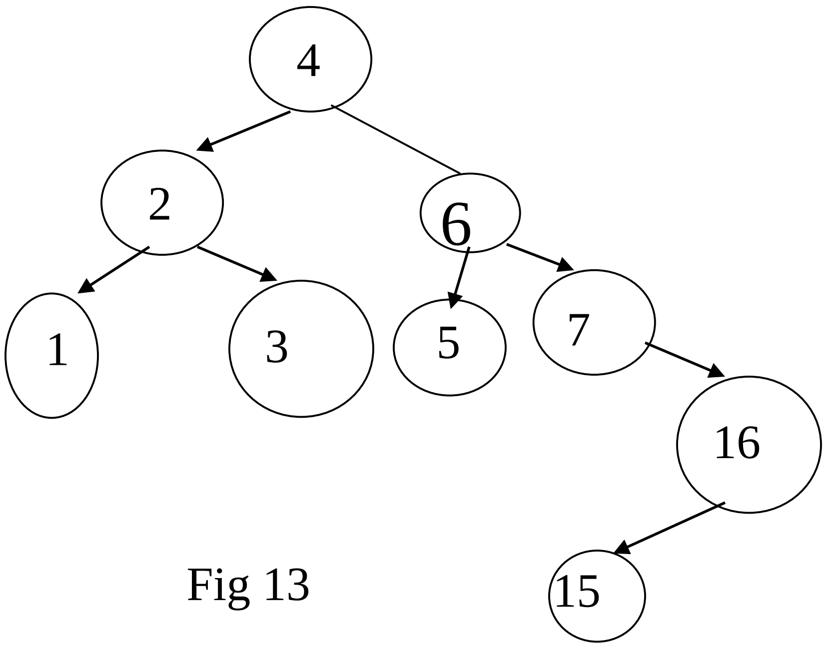


Fig 13

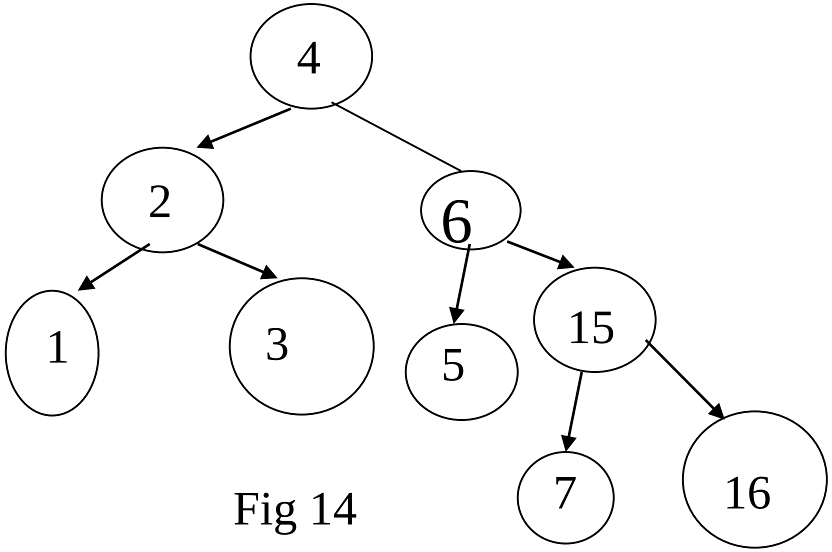


Fig 14

Insert 14

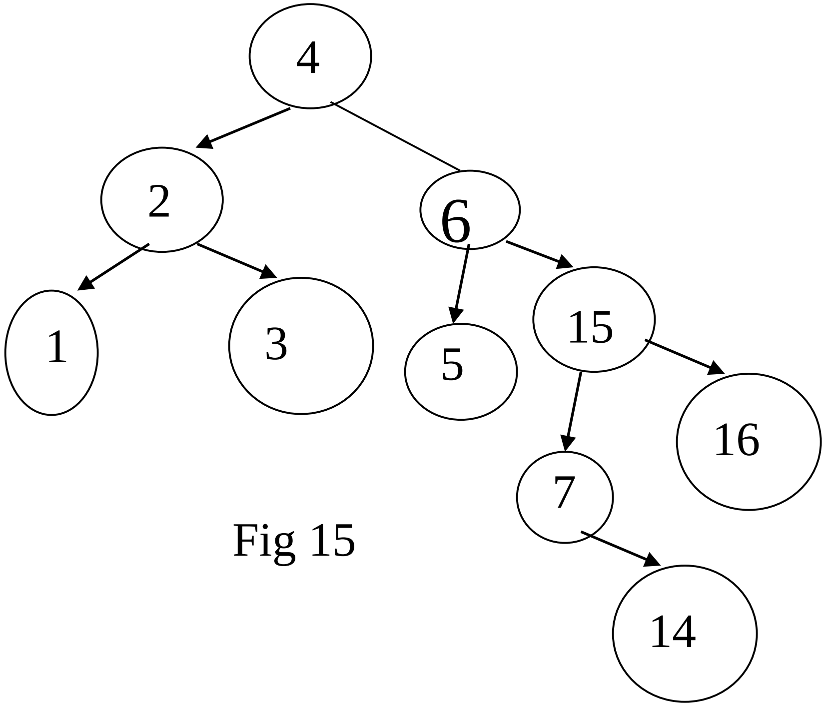


Fig 15

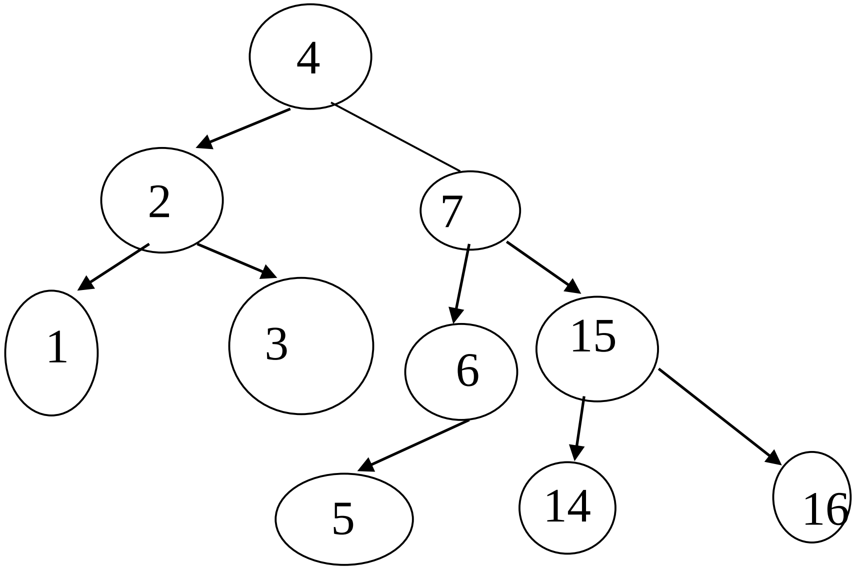


Fig 16

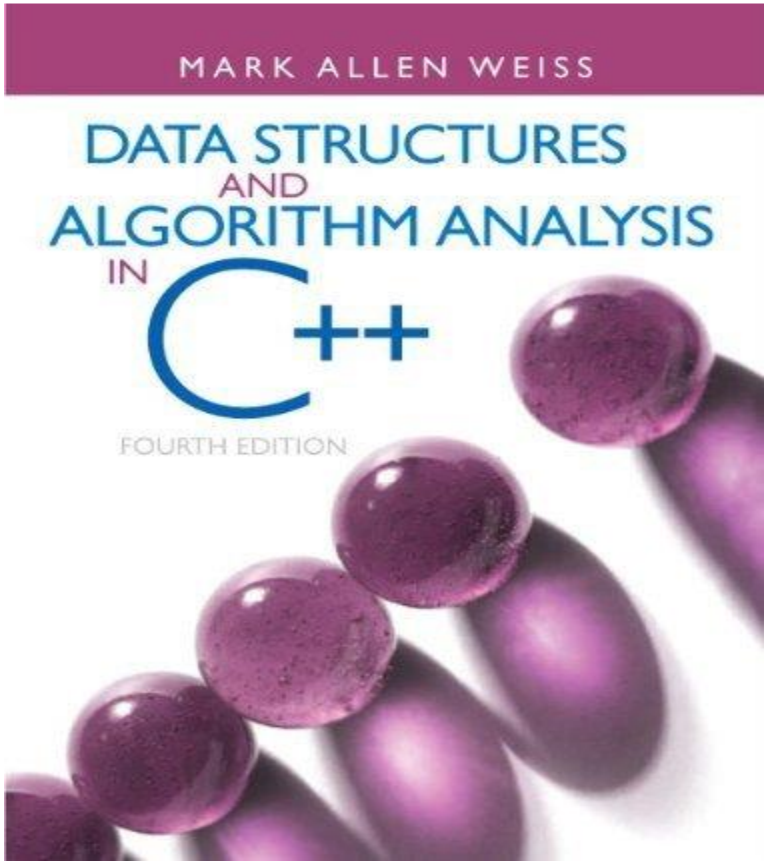
Continued in Book

Deletions can be done with similar rotations

Pseudocode

```
Insert(X, T) {
      If (T = NULL)
          insert X at T;   T->height = 0;
      If (X < T.element) {
              Insert(X, T ->left)
              If  Height(T ->left) - Height(T ->right)  = 2 {
                 If (X < T.leftchild.element)
                        T =singleRotatewithleft(T);
                 else
                       T =doubleRotatewithleft(T);        }
       }
      else If (X>T.element) {
             Insert(X, T ->right)
             If Height(T ->right) - Height(T ->leftt)  = 2 {
                If (X > T.righchild.element)
                       T =singleRotatewithright(T);
                 else
                       T =doubleRotatewithright(T);     }
             }
        T->height = max(height(T->left), height(T->right)) + 1;
        Return(T);  }
```

Slides based on the textbook



Mark Allen Weiss, (2014) Data Structures and Algorithm Analysis in C++, 4^{th} edition, Pearson.

Acknowledgement: This **course PowerPoints** make substantial (non-exclusive) use of the PPT chapters prepared by Prof. Saswati Sarkar from the University of Pennsylvania, USA, themselves developed on the basis of the course textbook. Other references, if any, will be mentioned wherever applicable.