Chapter 4

Trees

From Linear ADTs to …

- For input of large size, the linear access time of linked lists is prohibitive.
- Here we look at a simple data structure for which the average running time of most operations is O(logN), and some modifications to get O(NlogN).

  ➔ *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

- See how trees are used to implement the file system of several popular OSs.
- See how trees can be used to evaluate arithmetic expressions.
- Show how to use trees to support search operations in $O(\log N)$ average time and how to refine these ideas to obtain $O(\log N)$ worst-case bounds. We will also see how to 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***

A has 3 *children*, B, C, D

A is ***parent*** of B

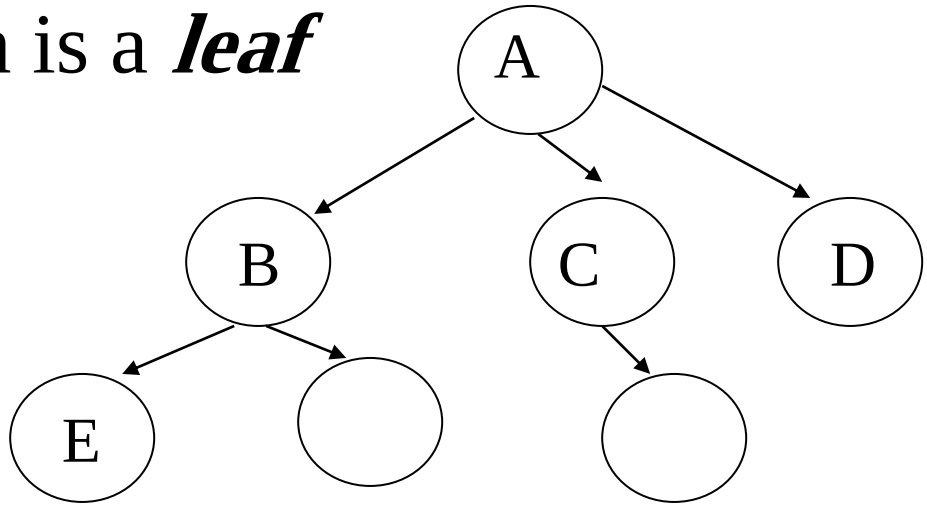
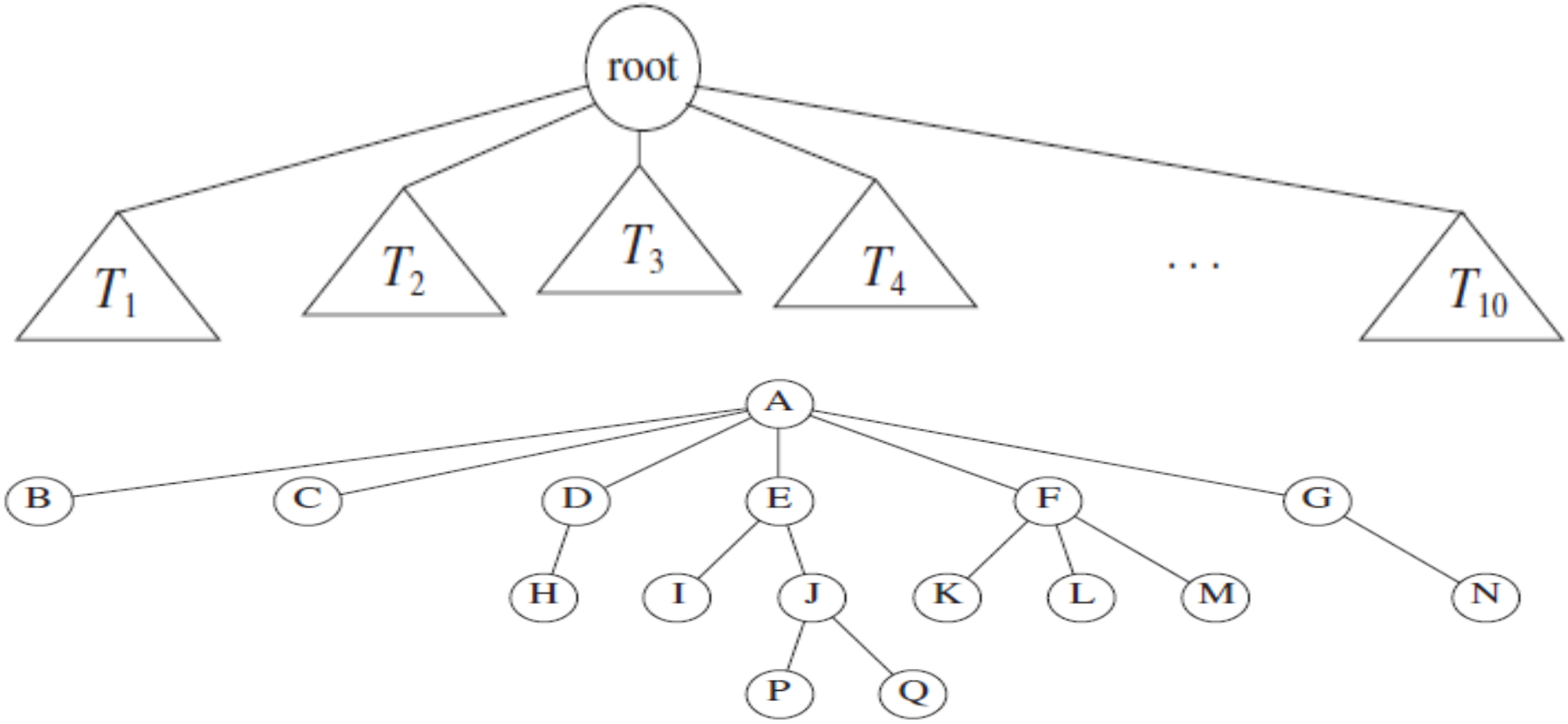


Illustration of a *tree* + terminology



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 root to any node.
- For any node *ni*, the **depth** of *ni* is the length of the unique path from the root to *ni*.

➔ 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.

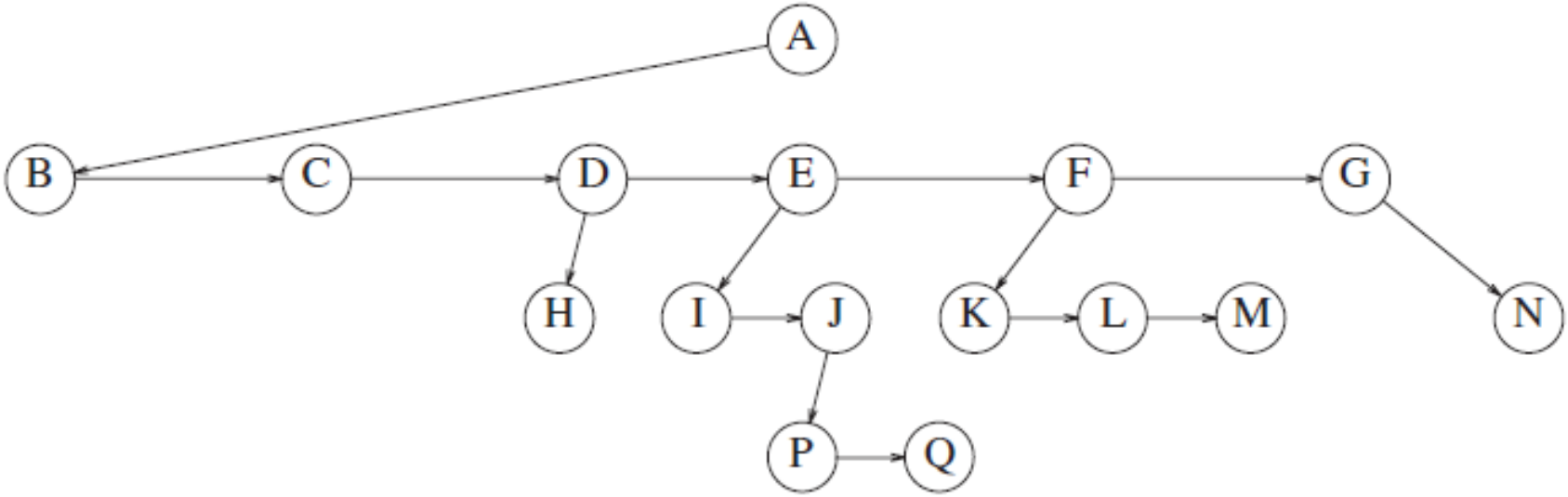
  ➔ height (tree) = height (root)
- If there is a path from $n1$ to $n2$, then $n1$ is an **ancestor** of $n2$ and $n2$ is a **descendant** of $n1$.
- If $n1 \neq n2$, then $n1$ is a **proper ancestor** of $n2$ and $n2$ is a **proper descendant** of $n1$.

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;
 };
```

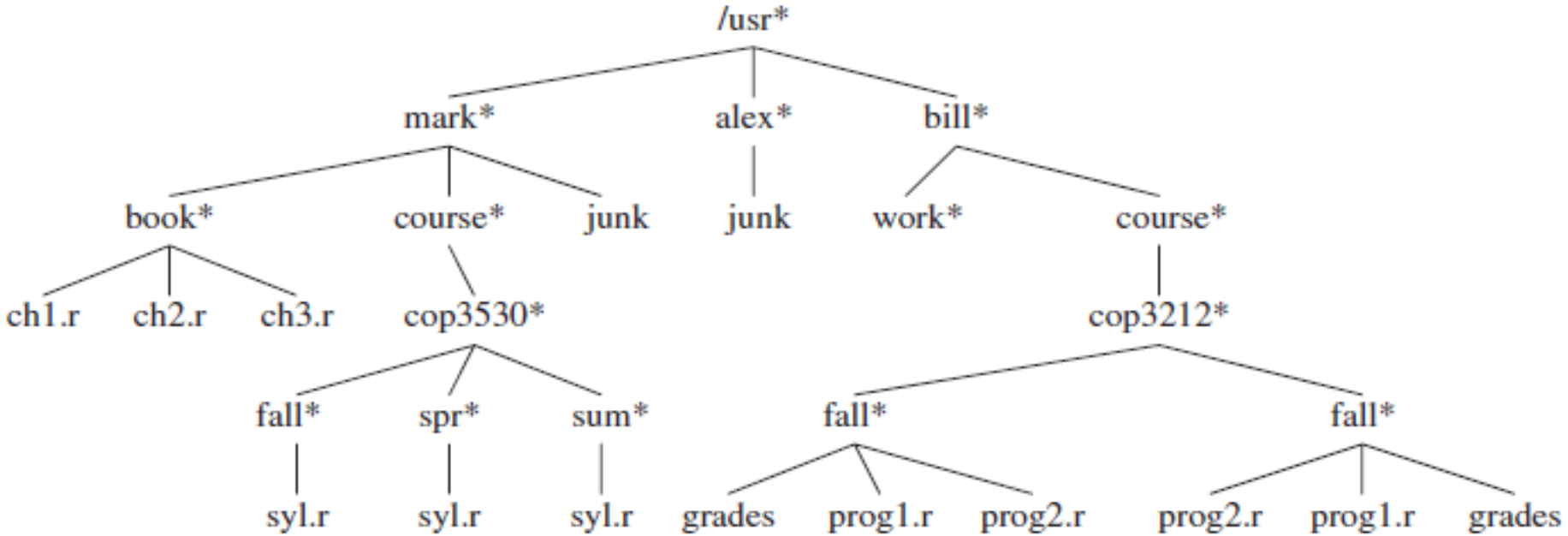
First child/next sibling representation of a tree



- 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

Application: 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; result is 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
1         printName( depth ); // Print the name of the object
2         if( isDirectory( ) )
3               for each file c in this directory
4                     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
```

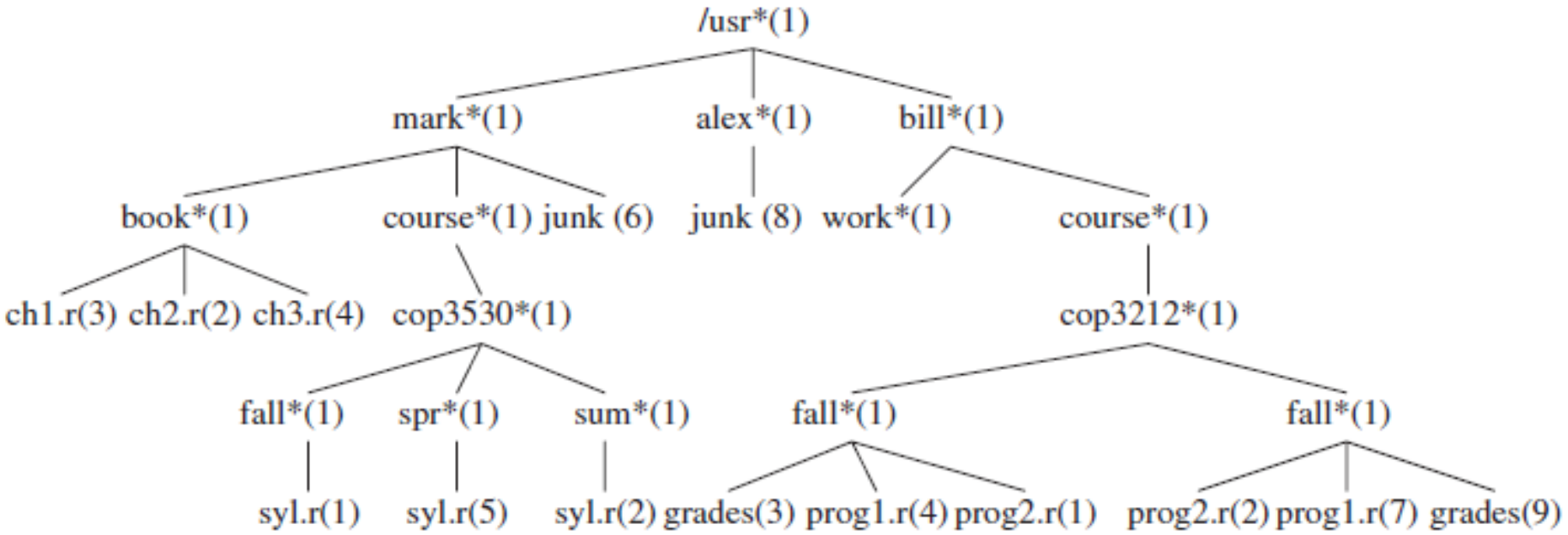
Traversal strategy

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

- **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;
}
```

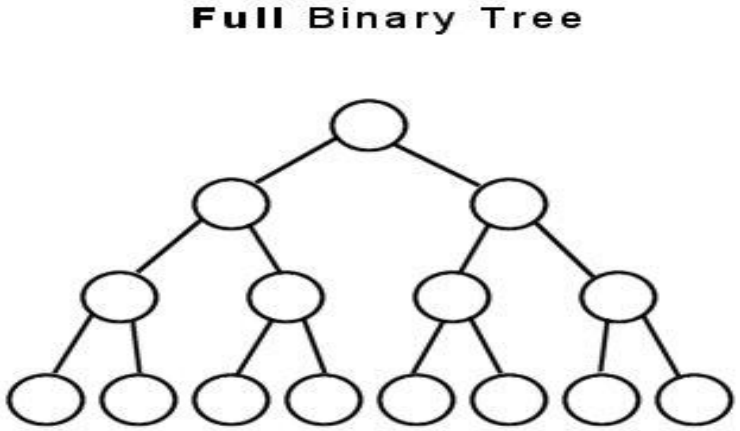
```
                ch1.r                    3
                ch2.r                    2
                ch3.r                    4
        book                            10
                        syl.r            1
                fall                     2
                        syl.r            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

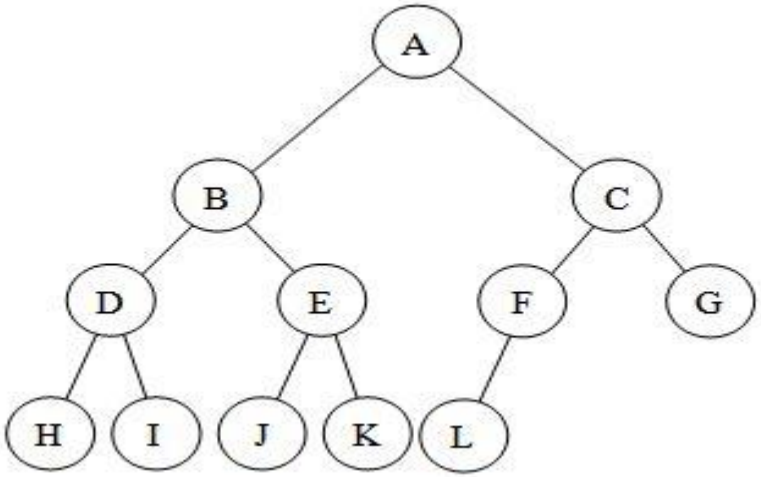
- A *Binary Tree* is a tree in which each node can have at most two children.
  - The depth of an average binary tree is generally considerably smaller than N.
  - An analysis shows that
    - the average depth of a binary tree is $O(\sqrt{N})$ (Exercise)
    - for a special type of binary tree, namely 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 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 $2^{d+1} - 1 = N$

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

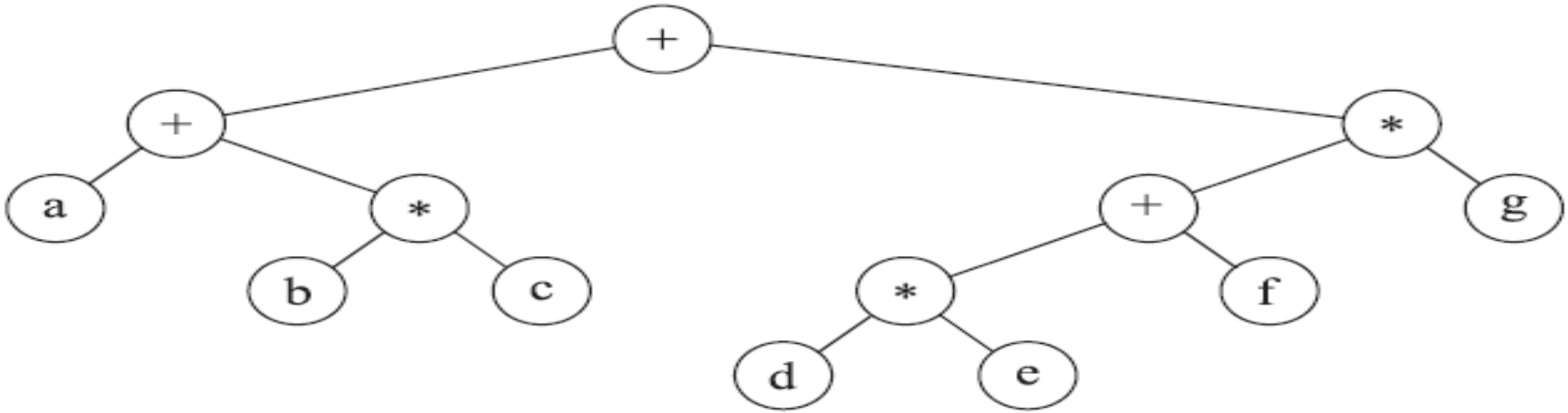
Implementation of a BT

```
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

- 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. Example?



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)

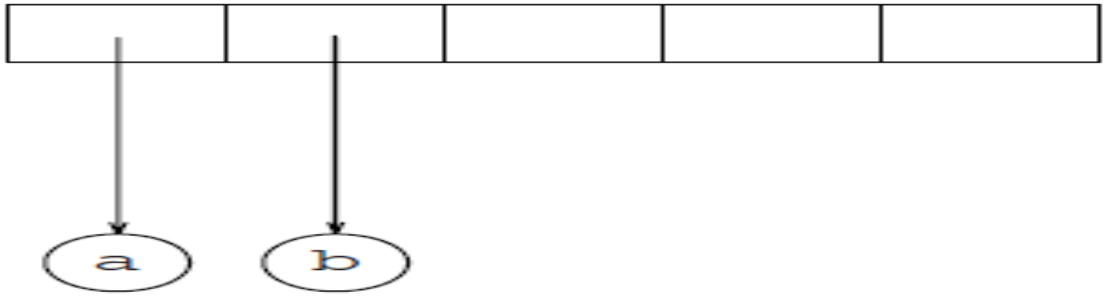
Constructing an ET

Algorithm to convert a postfix expression into an expression tree:

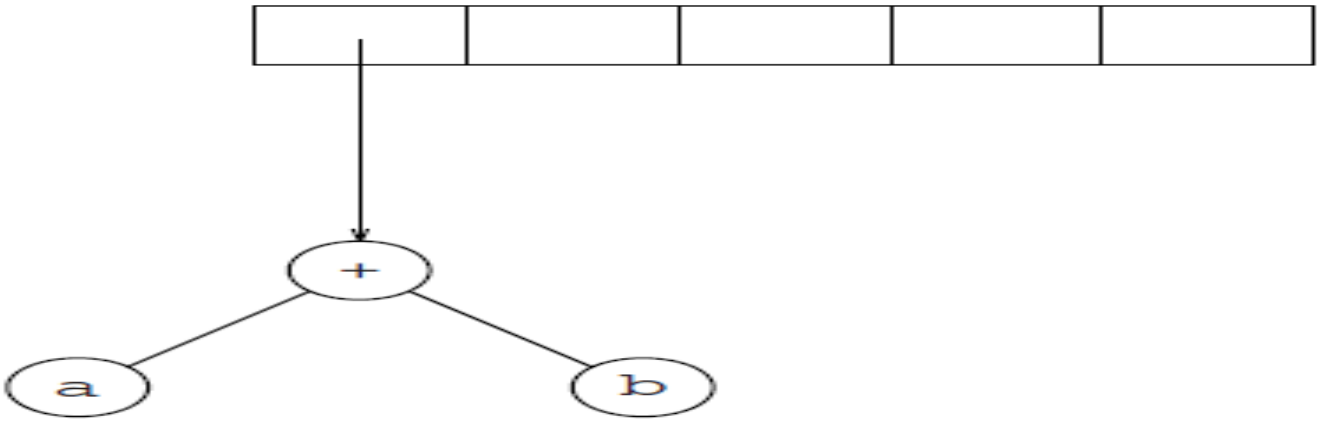
- Read the expression one symbol at a time.
- If the symbol is an operand, create a one-node tree and push a pointer to it onto a stack.
- If the symbol is an operator, pop (pointers) to two trees $T1$ and $T2$ from the stack ($T1$ is popped first) and form a new tree whose root is the operator and whose 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 + * *

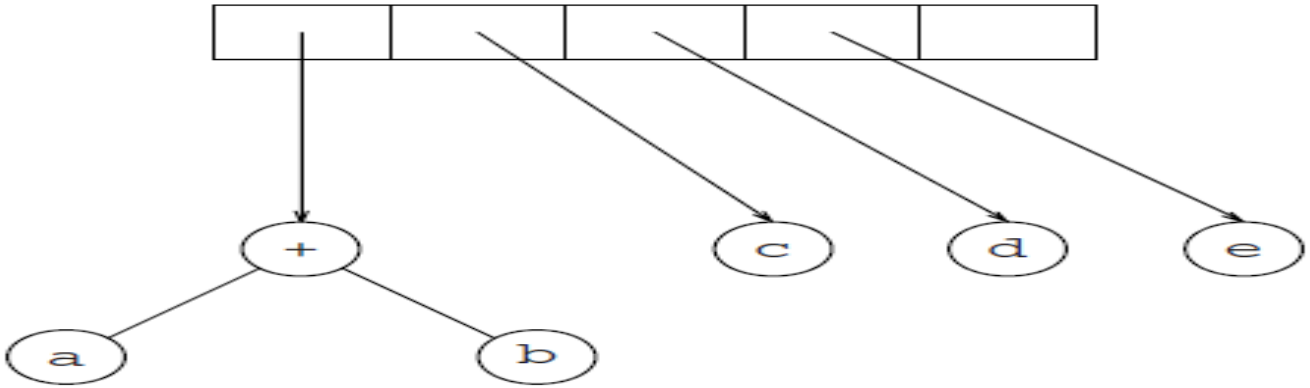
First two symbols are operands, so we create a one-node tree and push pointers to them onto a stack



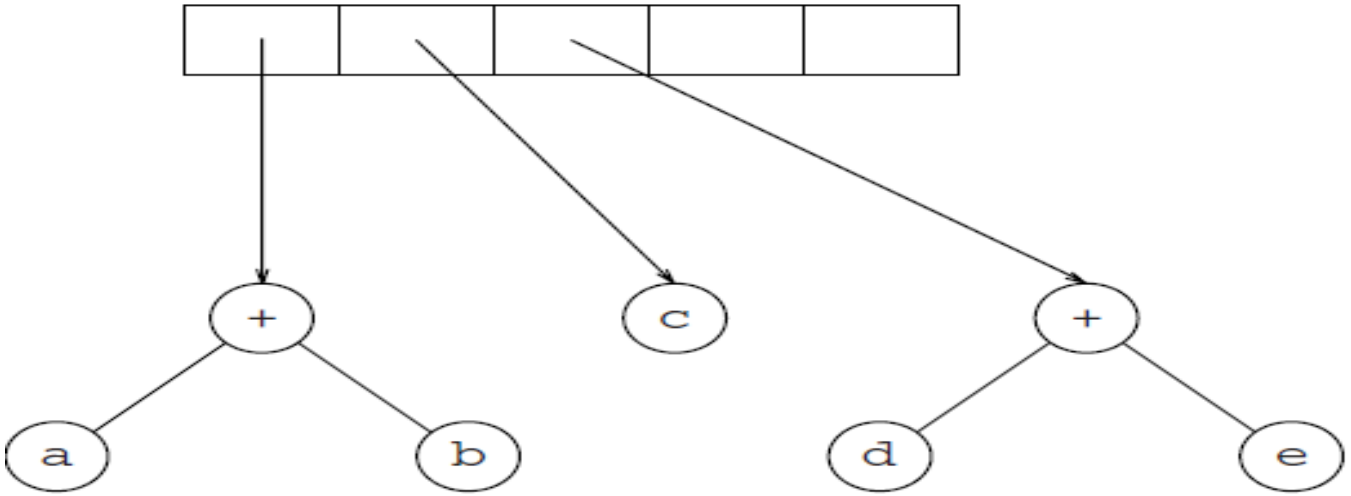
Next, + is read, so two pointers to trees are popped, a new tree is formed, and a pointer to it is pushed onto the stack.



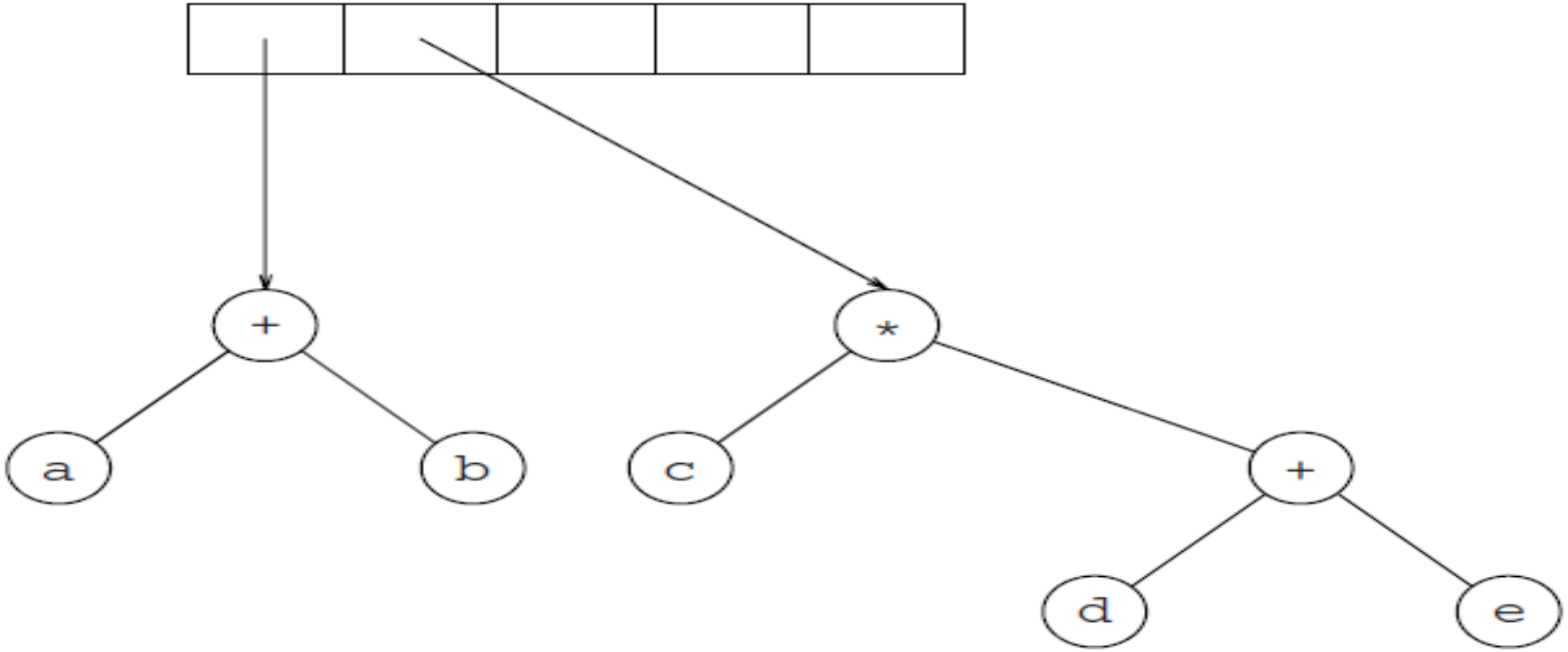
Next, c, d, and e are read, and for each, a one-node tree is created and a pointer to the corresponding tree is pushed onto the stack.



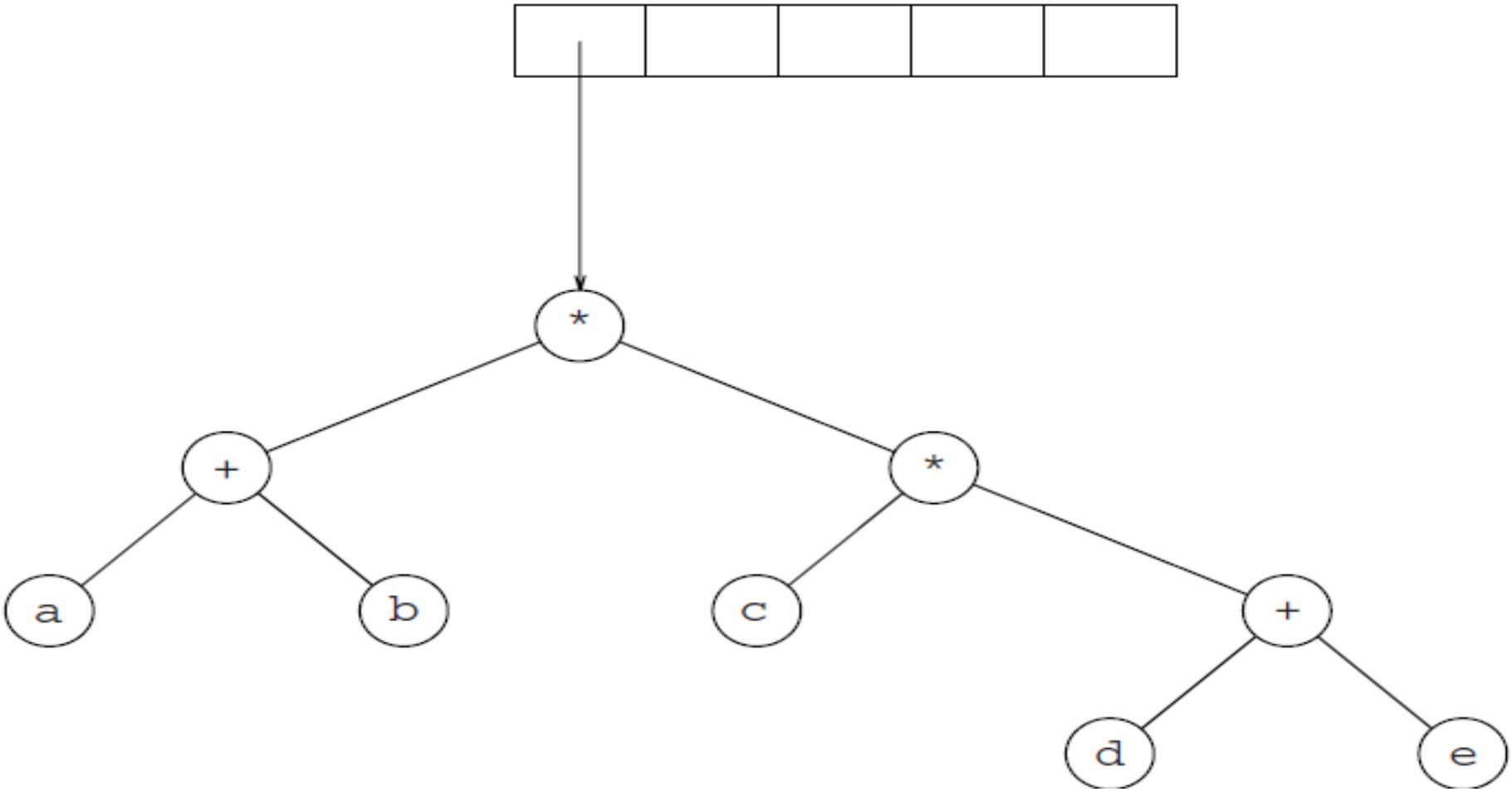
Now + is read, so two trees are merged.



* 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.

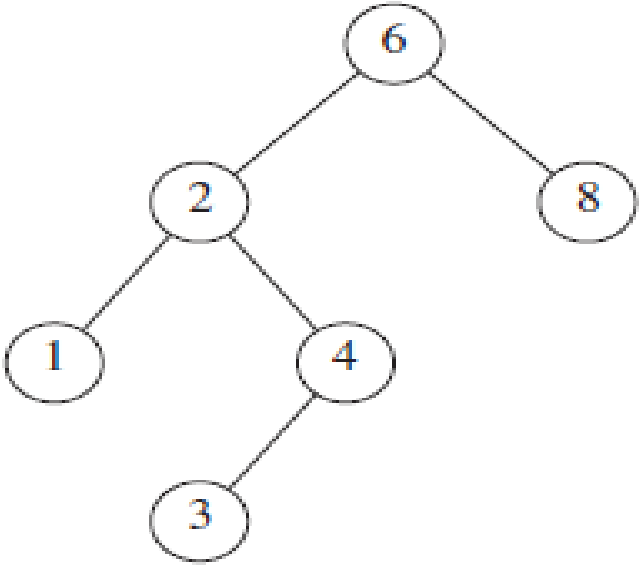


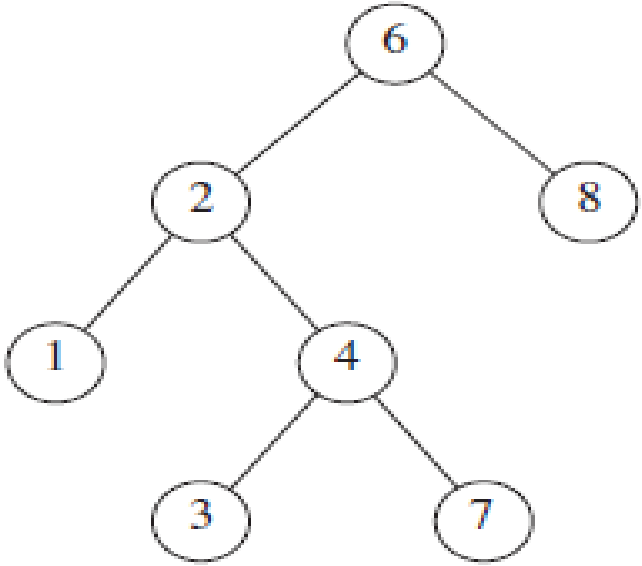
Binary Trees

- Important application of binary trees is their use in searching
- We will assume a tree of integers, though arbitrarily complex 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
- Examples: Are the following BSTs?





Operations on BSTs

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

➔ 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.

Binary Search Tree Code

The Big Five

- Classes come with five special functions that are already written for you:

  destructor, copy constructor, move constructor, copy assignment operator, and **move assignment operator**.

- These are the **big-five**.
- In many cases, you can accept the default behaviour provided by the compiler for the big-five.

Copy/Move Constructor

- Required to construct a new object, initialized to the same state as another object of the same type.
- *copy constructor* if the existing object is an lvalue,
- *move constructor* if the existing object is an rvalue (i.e., a temporary that is about to be destroyed anyway)

A copy constructor or move constructor is called in the following instances:

- a declaration with initialization, such as

  IntCell B = C; // Copy construct if C is lvalue;

  Move construct if C is rvalue

  IntCell B { C }; // Copy construct if C is lvalue;

  Move construct if C is rvalue

  but not B = C; // Assignment operator, discussed later
- an object passed using call-by-value (instead of by & or const &), which, should rarely be done anyway.
- an object returned by value (instead of by & or const &).
  - a copy constructor is invoked if the object being returned is an lvalue,
  - a move constructor is invoked if the object being returned is an rvalue.

Copy/Move Assignment Operators

- Assignment operator is called when = is applied to two objects that have both been previously constructed.
- *lhs=rhs* is intended to copy the state of rhs into lhs.
  - If rhs is an lvalue, this is done by using the copy assignment operator
  - if rhs is an rvalue, this is done by using the move assignment operator.
- By default, the copy assignment operator is implemented by applying the copy assignment operator to each data member in turn

The big five: Example signatures

- ~IntCell( ); // Destructor
- IntCell(const IntCell & rhs);
  // Copy constructor
- IntCell(IntCell && rhs);
  // Move constructor
- IntCell & operator= (const IntCell & rhs);
  // Copy assignment
- IntCell & operator= (IntCell && rhs);
  // Move assignment

Searching an element in the Tree

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.

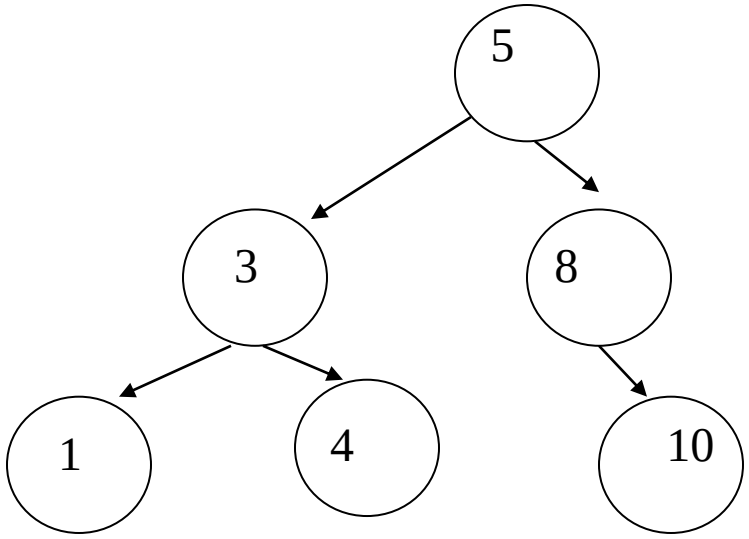
If it is more, go to the right subtree.

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

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

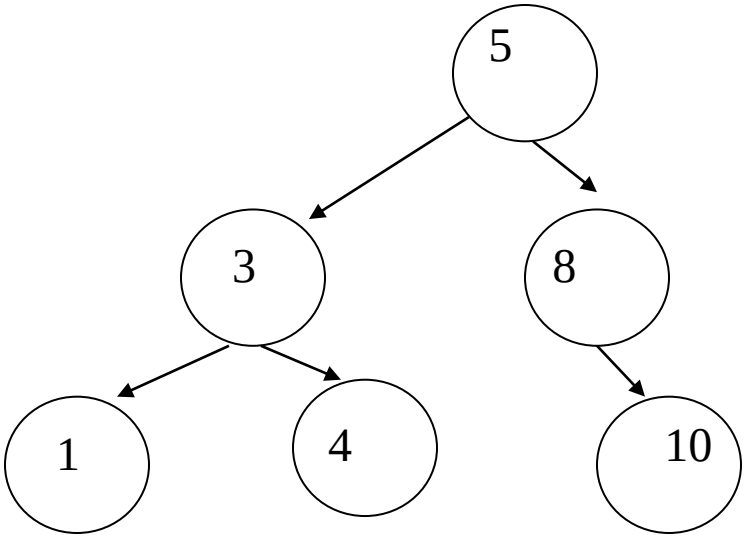
Not found!

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

```
Node = root;
For (node = root; node=node.leftchild; node!=NULL)
        prevnode = node;
Return (prevnode);
```

Complexity: O(d)



Travel 5, 3, 1

Return 1;

Insert an element

Try to find the element;

If the element exists, do nothing.

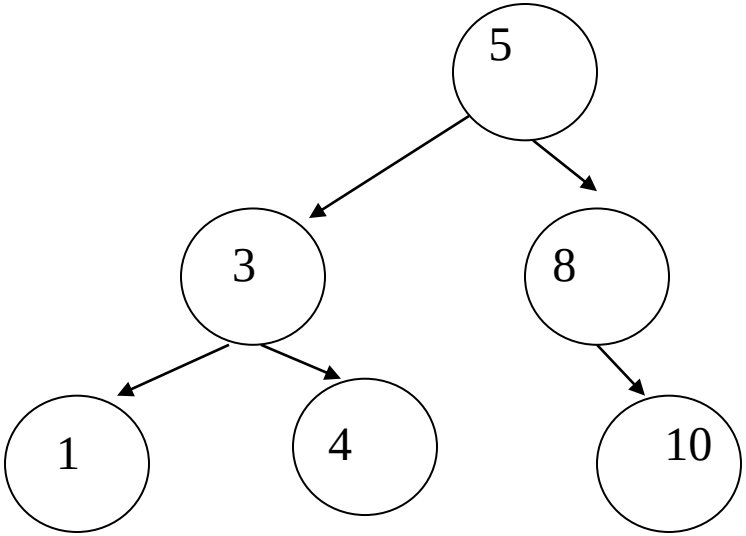
If it does not, insert it at the position of the returned null pointer;

```
Insert(node, element)
    {
        If (node.key = element) conclude FOUND and Return;
        Else If (element < node.key)
              {
                   If (node.leftchild =NULL)
                                 insert the new node at node.leftchild;
                   Else   Insert(node.leftchild, element);
               }
        Else If (element > node.key)
                              {
                                ??????????? }
```

Think about it!

}

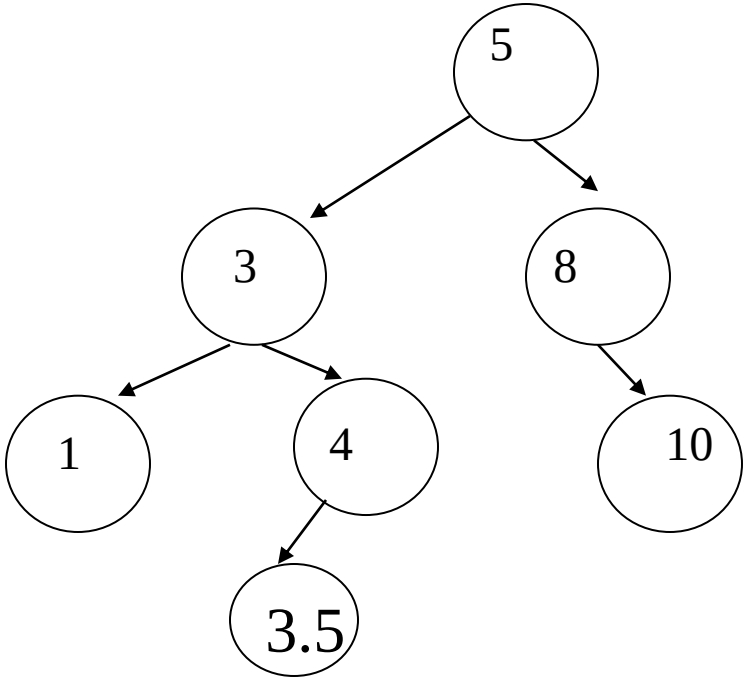
Complexity: O(d)



Insert 3.5

Sequence Traveled:

5, 3, 4



Insert 3.5 as left child of 4

DELETION

When we delete a node, we need to consider how we take care of the children of the deleted nodes.

This has to be done such that the property of the 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.

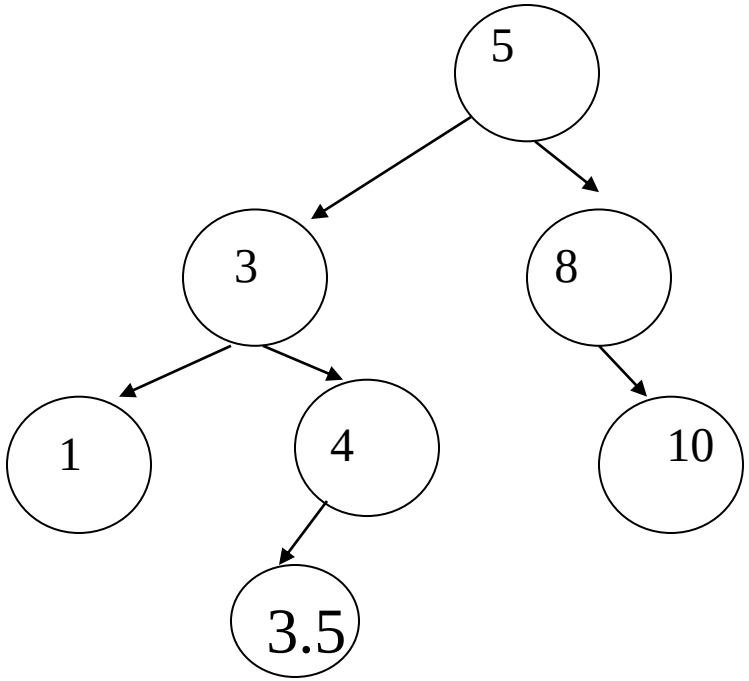
Pseudo Code

```
Delete(node) {
If node is childless, then
     {
          node->parent->ptr_to_node = NULL
          free node;
      }
If node has one child
      {
           node->parent->child = node->child;
           free node;
       }
```

If a node has 2 children,

```
    {
        minnode = findmin(rightsubtree)->key;
        node->key = minnode->key;
        delete(minnode);
     }
}
```

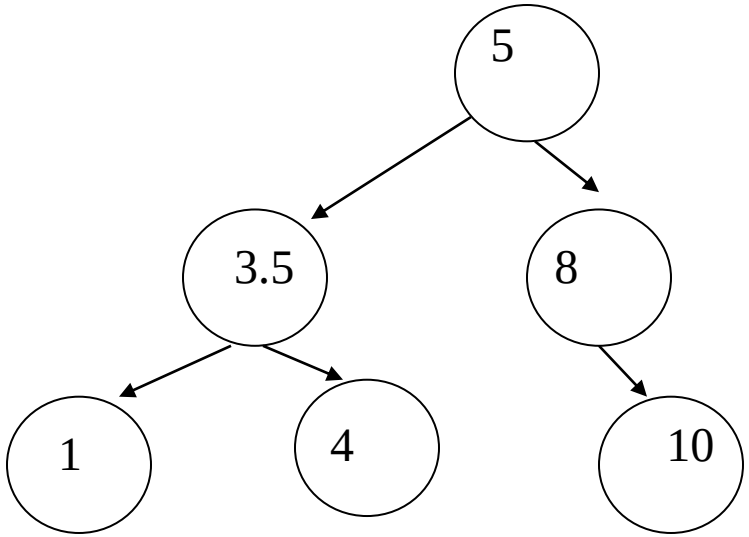
Complexity? **O(d)**



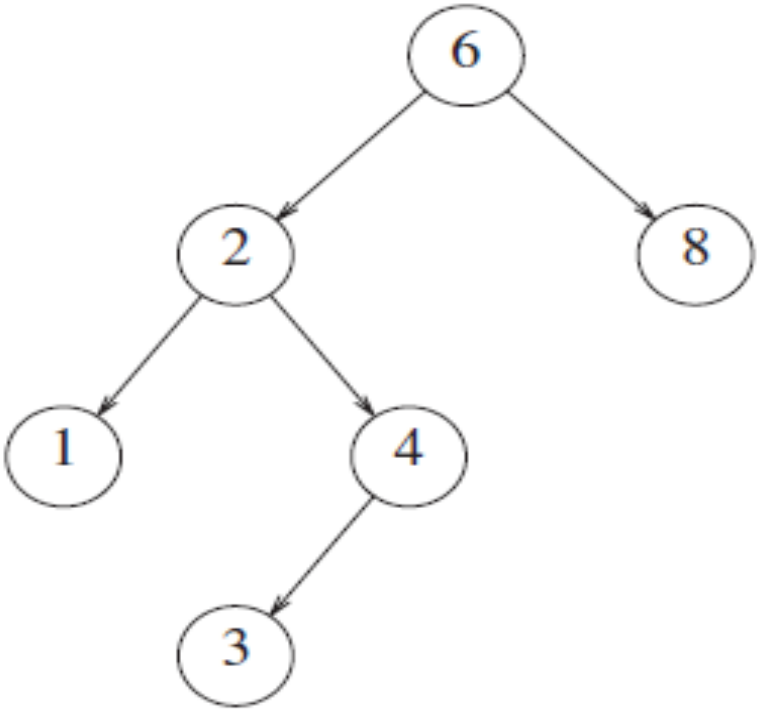
Delete 3;

3 has 2 children;

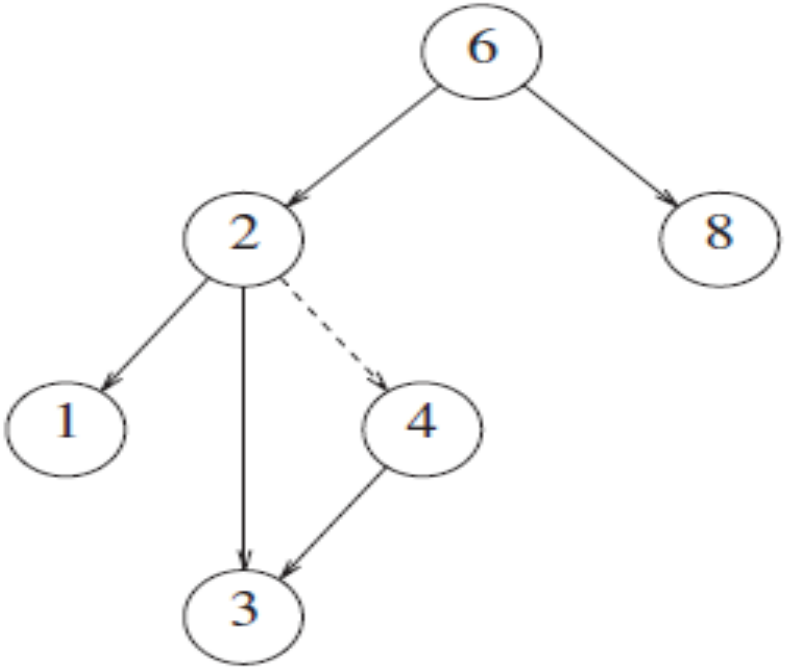
Findmin right subtree for 3 returns 3.5



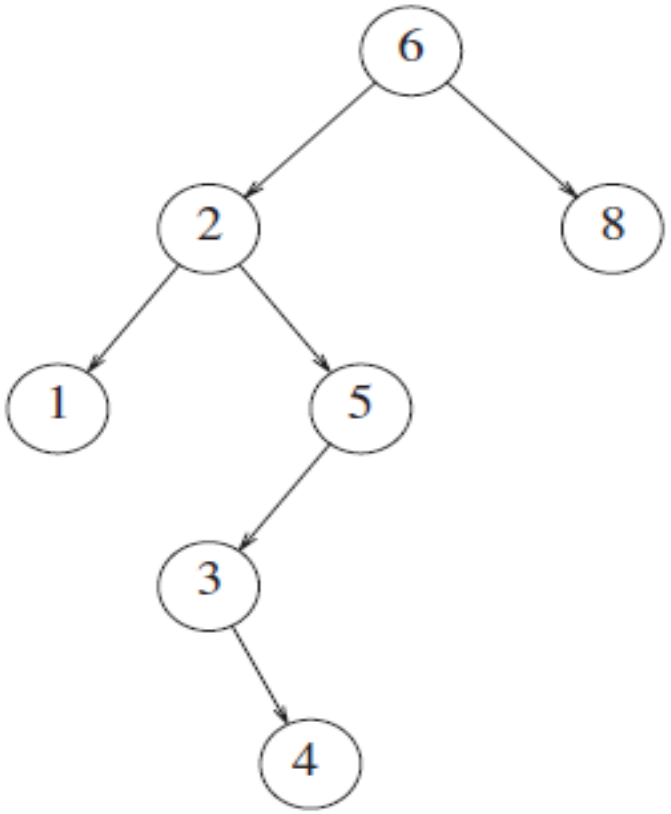
So 3 is replaced by 3.5, and 3.5 is deleted.



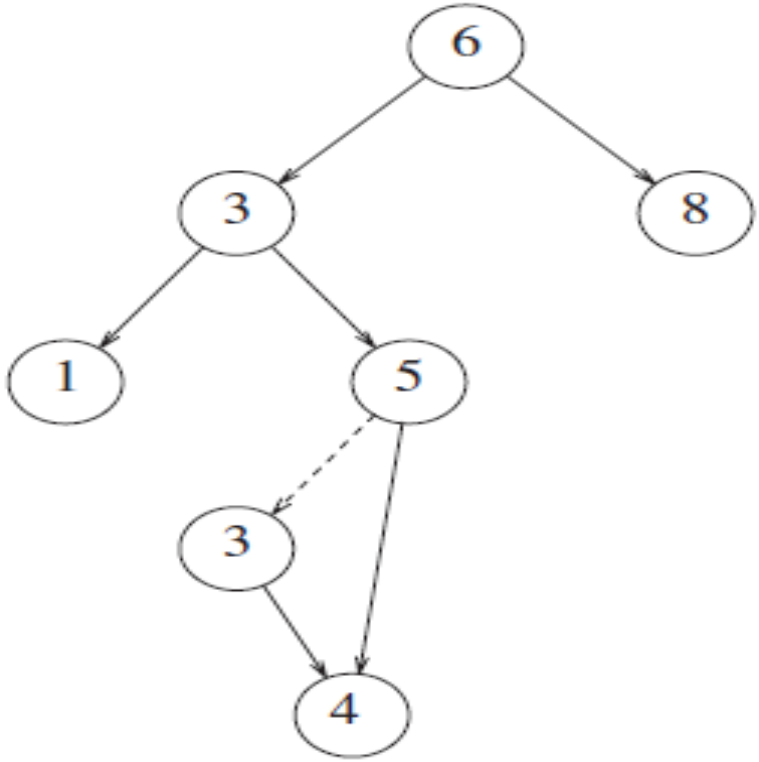
Before Delete 4
(with 1 child)



After Delete 4
(with 1 child)



Before Delete 2
(with 2 children)



After Delete 2
(with 2 children)

Operations on BSTs: Code

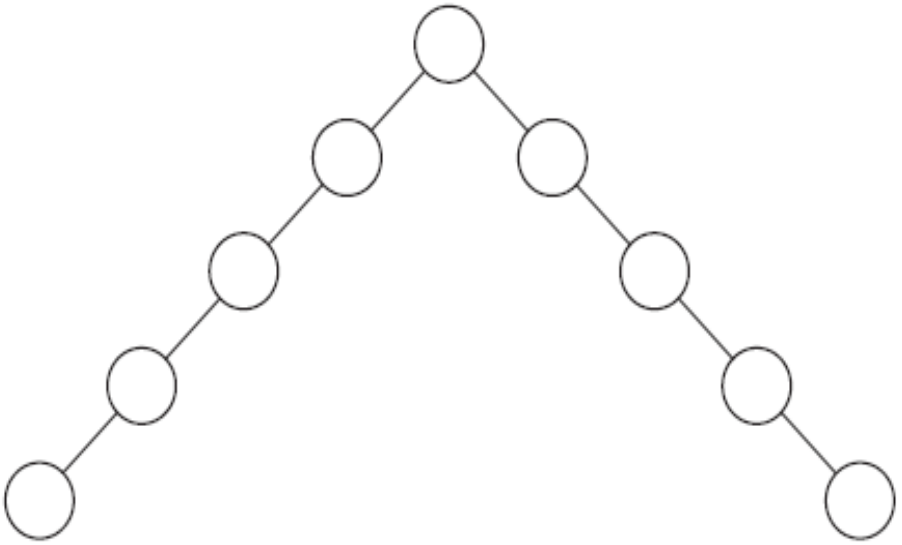
Binary Search Tree Code

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.

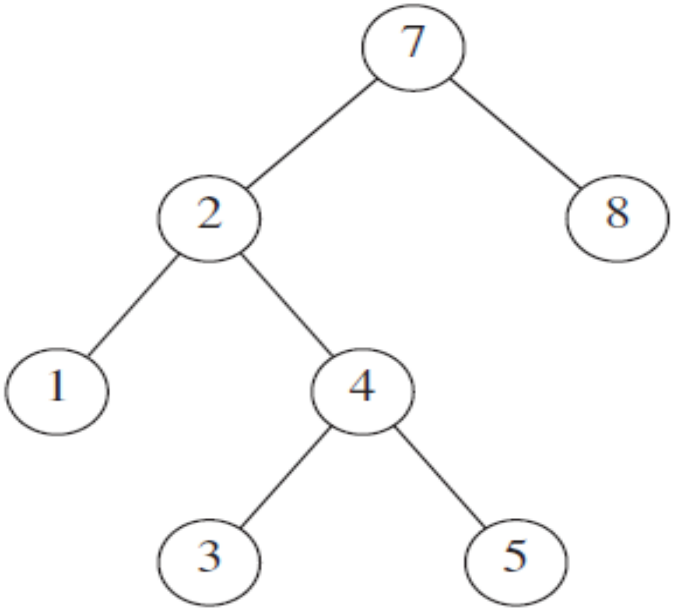
AVL Trees

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

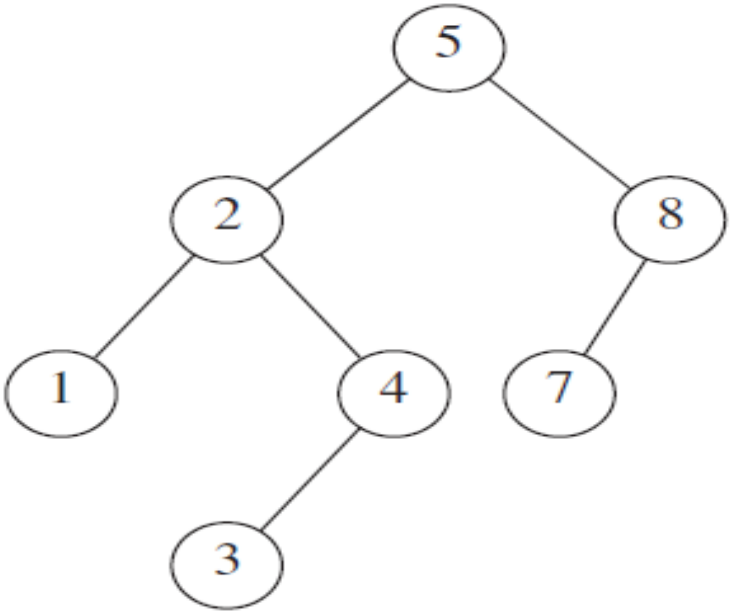


An AVL (Adelson-Velskii and Landis) tree is a BST where the heights of the two subtrees of any node differ by at most one

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 $\log N$.
- The minimum number of nodes, $S(h)$, in an AVL tree of height h: $S(h) = S(h-1)+S(h-2)+1$.

  For $h = 0$, $S(h) = 1$. For $h = 1$, $S(h) = 2$

➔ all the tree operations can be performed in ***O*(log*N*)** time, except insertion and deletion (need to update all the balancing information)

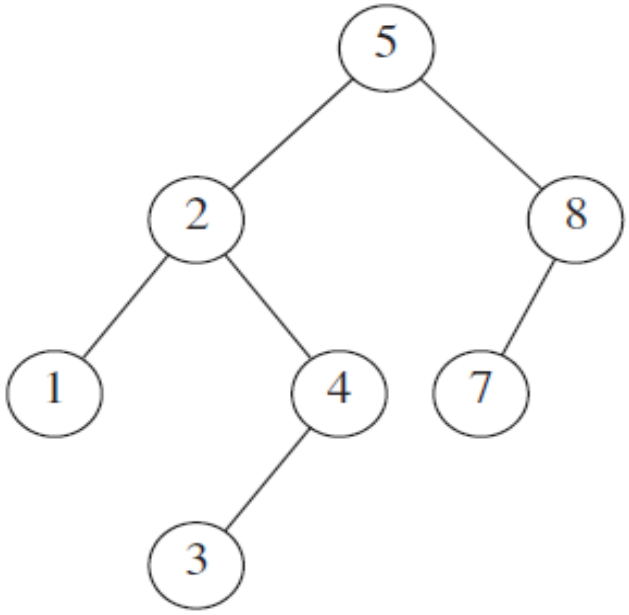
Operations in AVL Tree

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 an insertion, only nodes that are on the path from the 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

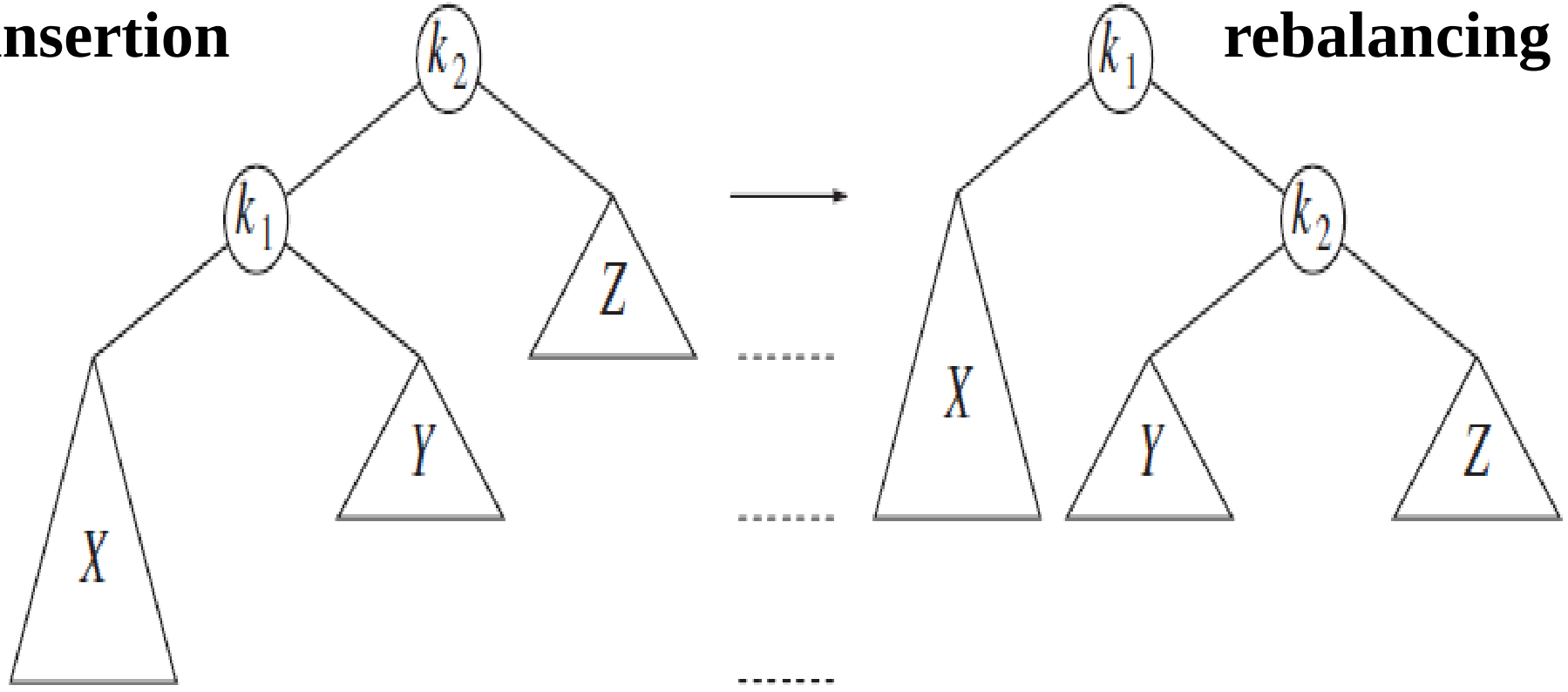
Rotations

- The first case, in which the insertion occurs on the “outside” (i.e., left–left or right–right), is fixed by a **single rotation** of the tree.
- The second case, in which the insertion occurs on the “inside” (i.e., left–right or right–left) is handled by the slightly more complex **double rotation.**
- These are fundamental operations on the tree that we’ll see used several times in balanced-tree algorithms

Single Rotation

After insertion

After rebalancing



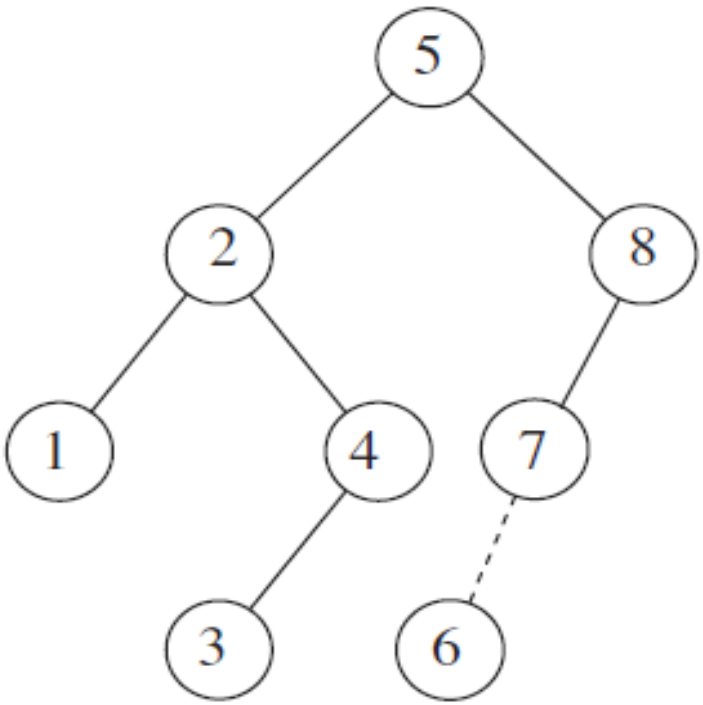
- *k*2 violates the AVL balance property because its left subtree is two levels deeper than its right subtree. (Generically: only possible case: Subtree *X* has grown to an extra level, causing it to be exactly two levels deeper than *Z*.)

Result of Single Rotation

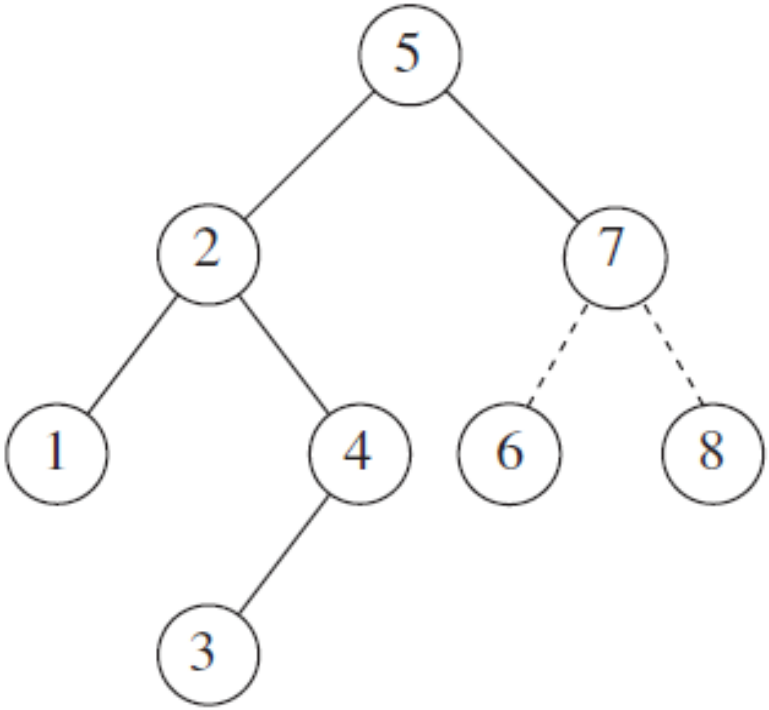
- Note that Single Rotation requires only a few pointer changes,
- We get another BST that is an AVL tree.
- This is because *X* moves up one level, *Y* stays at the same level, and *Z* moves down one level.
- *k*2 and *k*1 not only satisfy the AVL requirements, but they also have subtrees that are exactly the same height.
- New height of entire subtree is *exactly the same* as the height of the original subtree.

➔ no further updating of heights on the path to the root is needed, and consequently *no further rotations needed.*

Example of Single Rotation



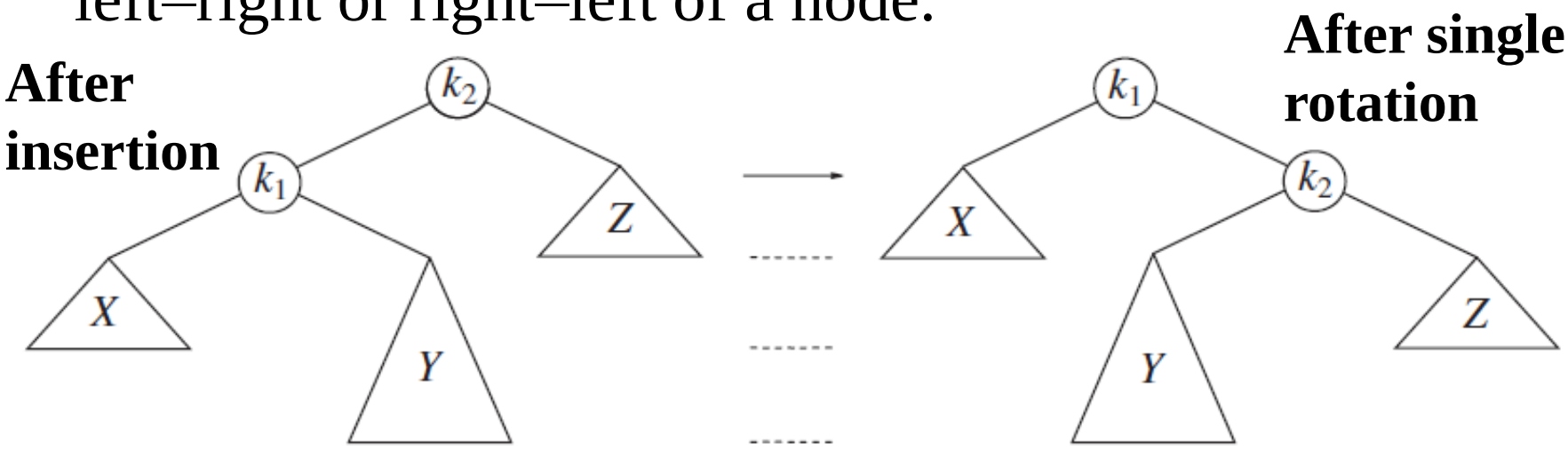
After insertion



After rebalancing

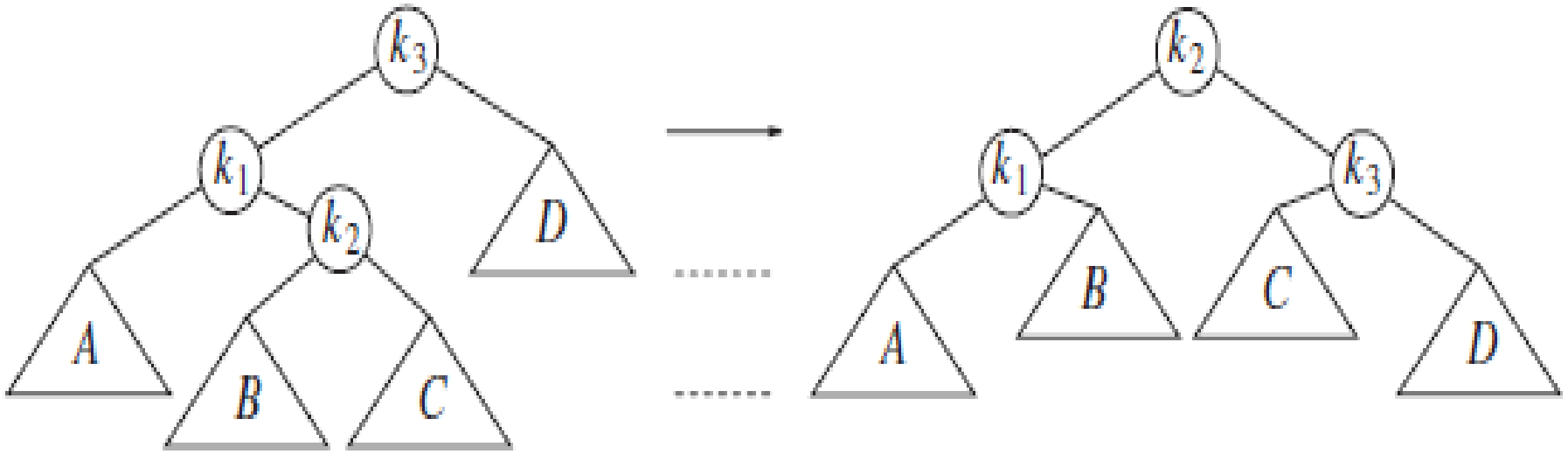
Double Rotation

- 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.



- Subtree Y has had an item inserted into it➔ guarantee that it is nonempty.
- We may assume that it has a root and two subtrees.
- ➔ the tree may be viewed as four subtrees connected by three nodes.

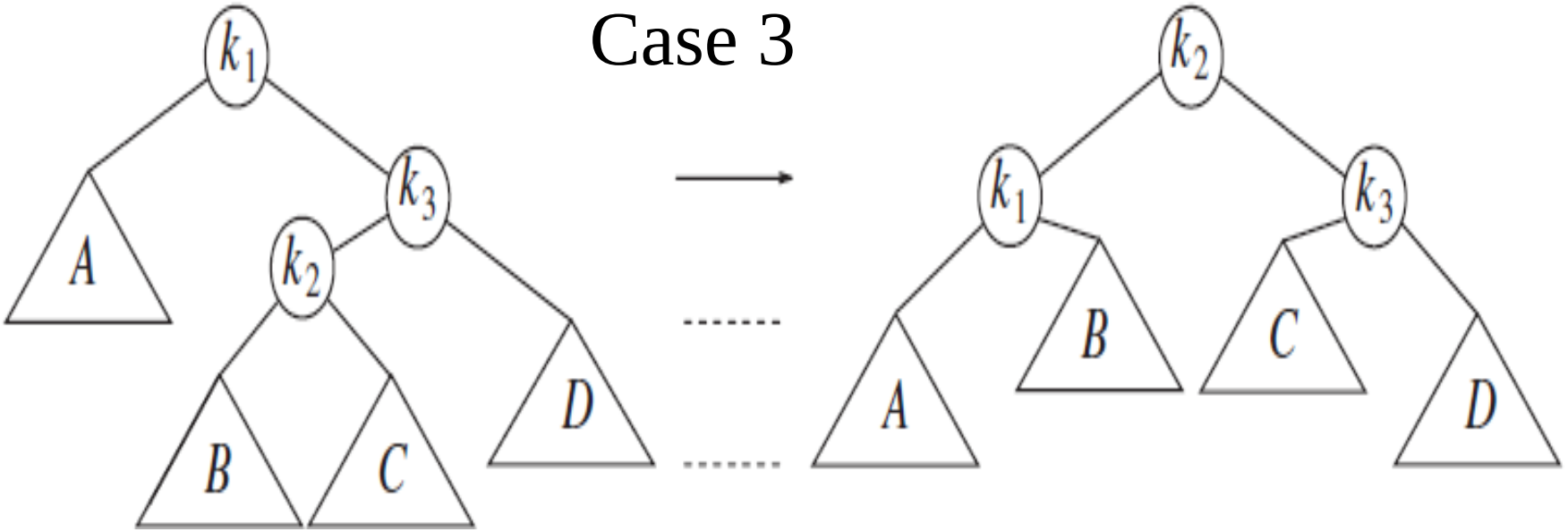
Case 2



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

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
                 // SingleRotate routine in Fig 4.41 (Weiss)
                 //         Separate for left and right nodes
                 // DoubleRotate routine in Fig 4.43 (Weiss)
                 //         Separate for left and right nodes
```

```
{
      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);  }
```

Extended Example

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

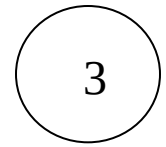


Fig 1

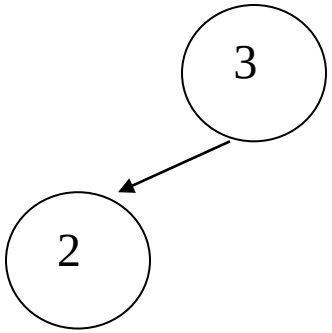


Fig 2

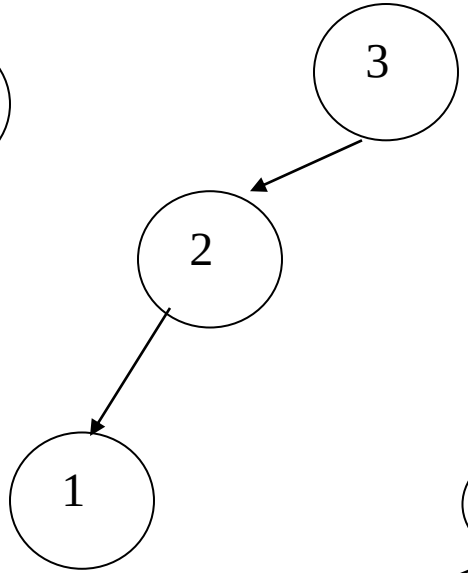


Fig 3

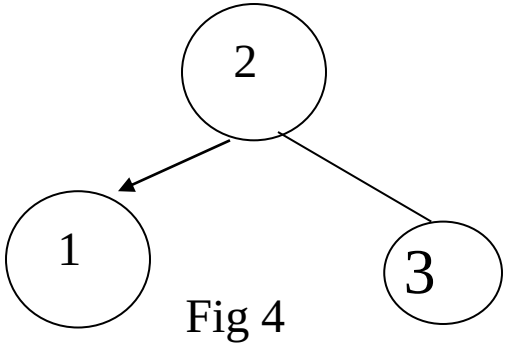


Fig 4

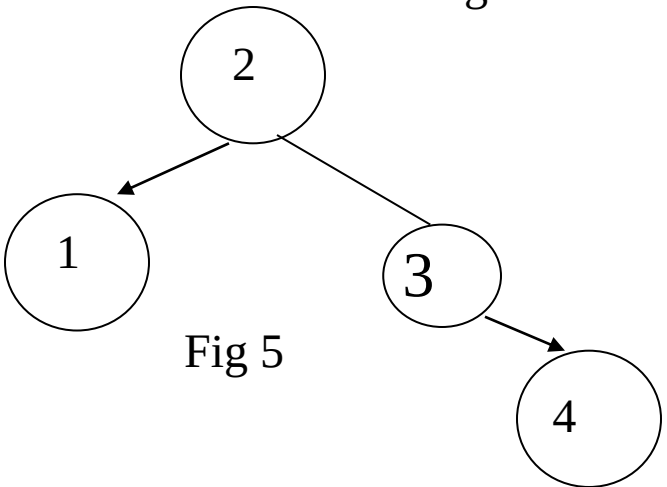


Fig 5

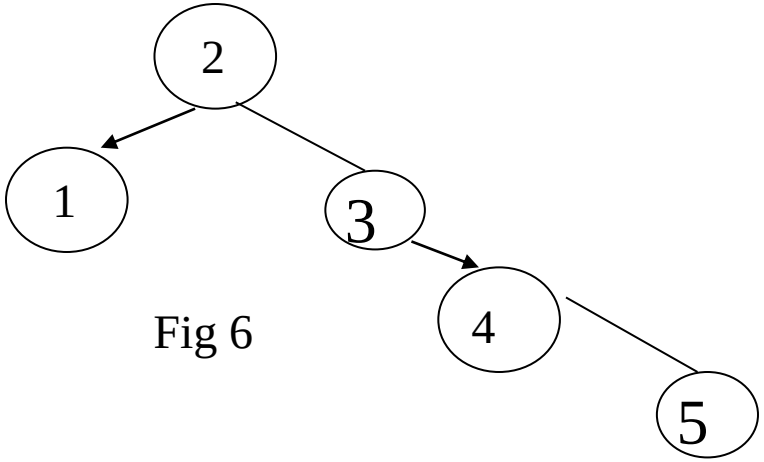


Fig 6

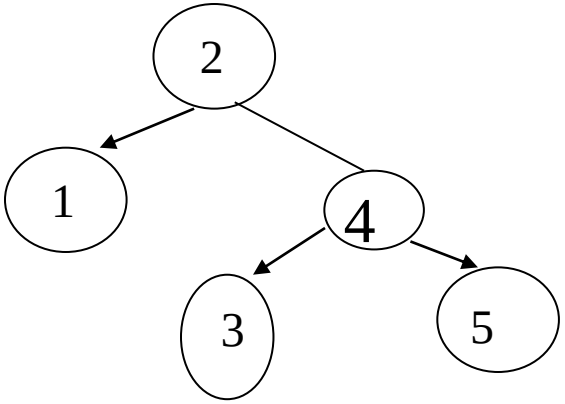


Fig 7

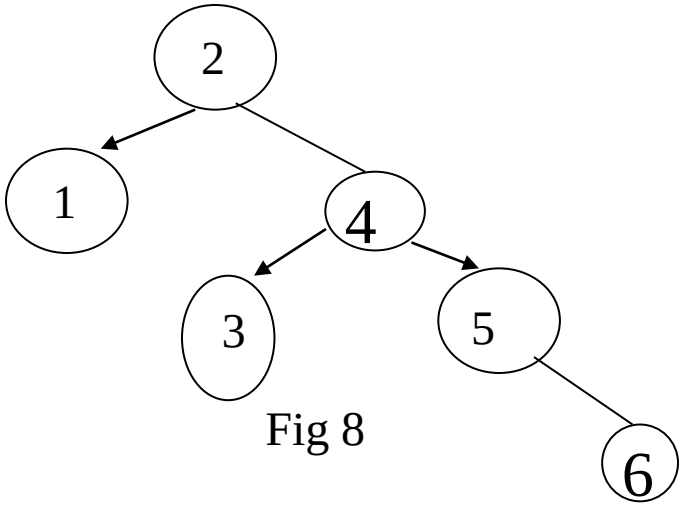


Fig 8

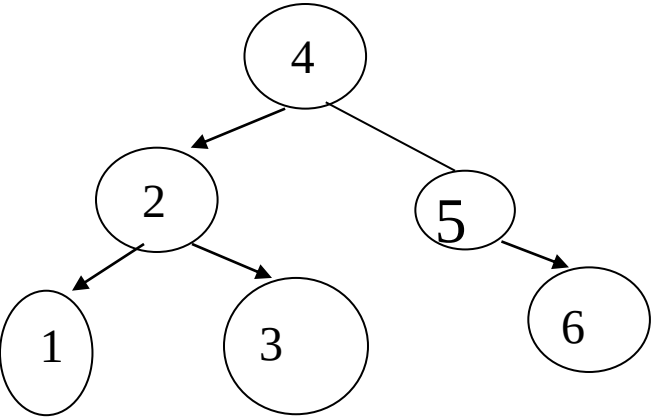


Fig 9

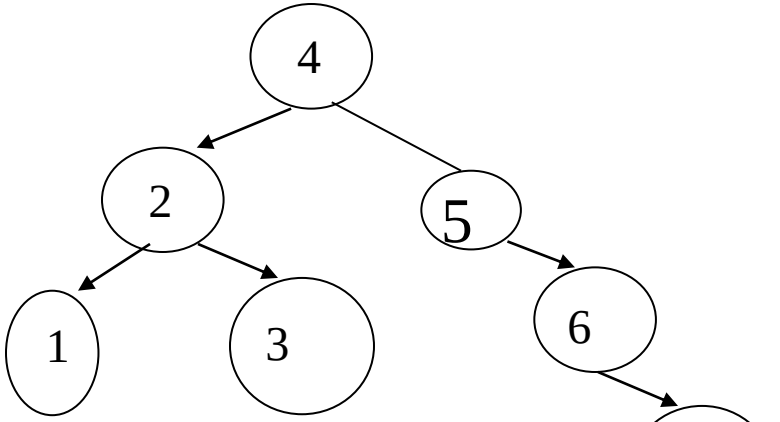


Fig 10

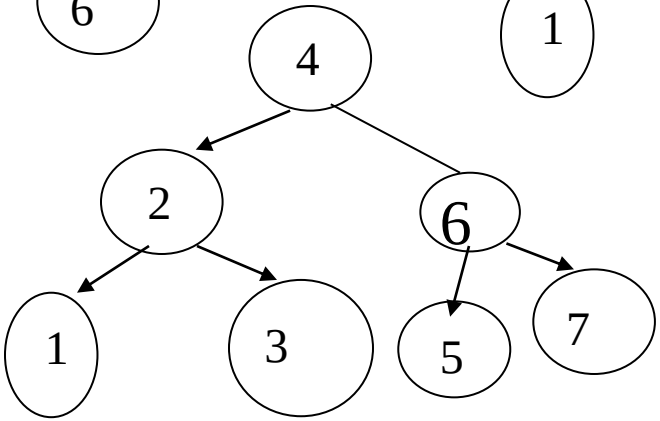


Fig 11

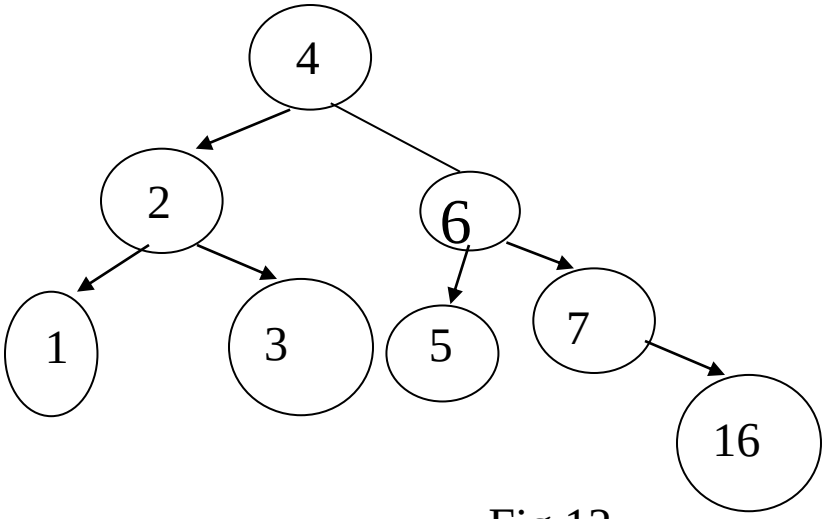


Fig 12

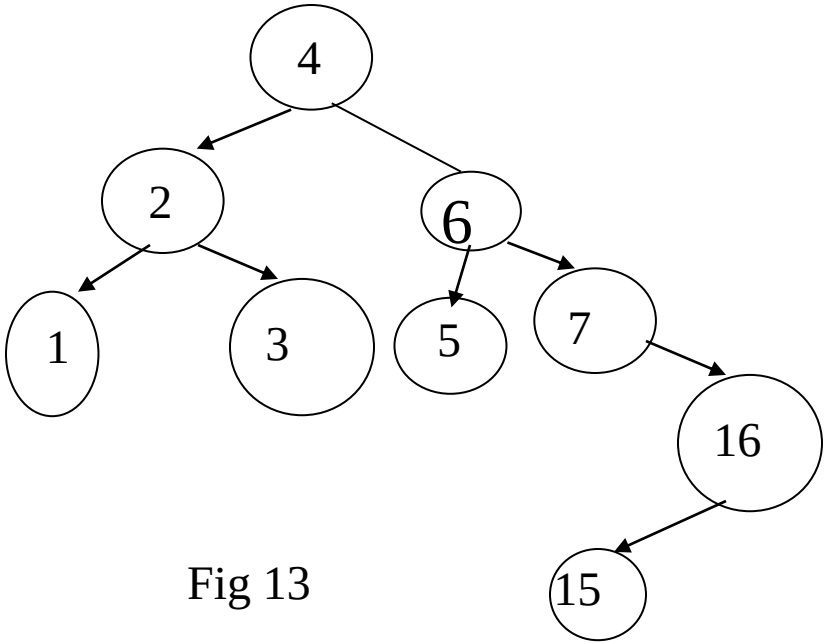


Fig 13

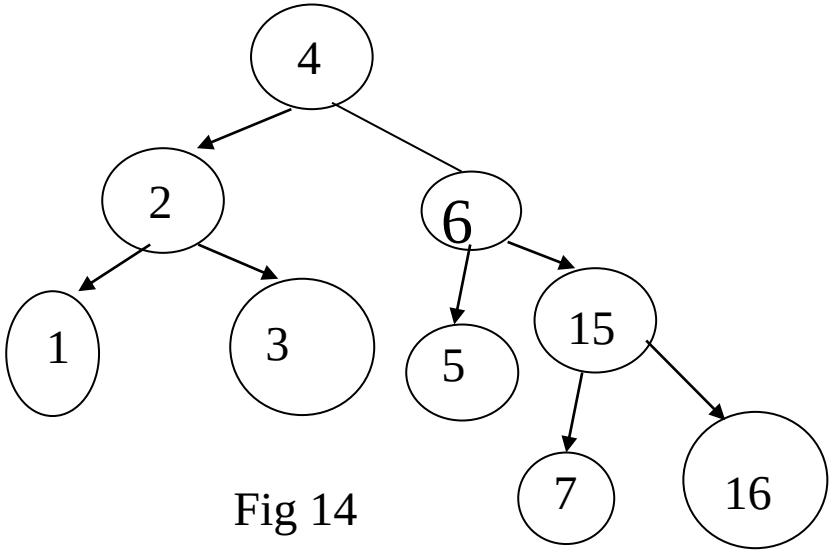


Fig 14

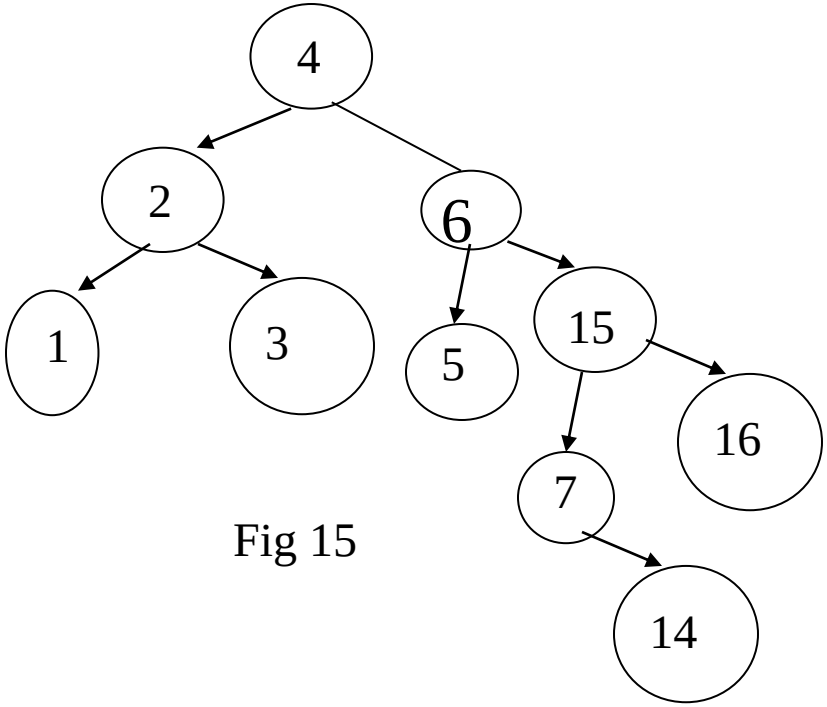


Fig 15

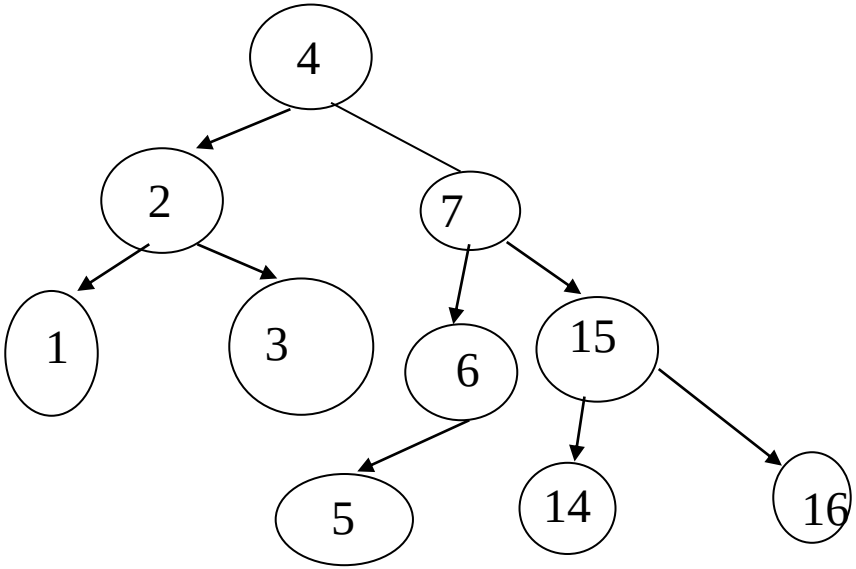


Fig 16

Continued in Book

Deletions can be done with similar rotations

Tree Traversal Revisited

- **Inorder traversal**: process left subtree, process current node, process right subtree. E.g. to list the elements of a BST inorderTraversalBST
  - **total running time: $O(N)$:** constant work performed at every node in the tree (testing against nullptr, setting up two function calls, and doing an output statement) & each node is visited once.
- **Postorder traversal**: when we need to process both subtrees first before we can process a node. E.g. to compute the height of a node➔ LTree, RTree, Node
  - **total running time: $O(N)$:** constant work performed at each node postOrderTraversalBST

Tree Traversal Revisited

- **PreOrder traversal**: Node is processed before the children. E.g. to label each node with its depth. (See file system example in this chapter)

- **Level-order traversal**. All nodes at depth d are processed before any node at depth $d + 1$.
  - Level-order traversal differs from the other traversals in that it is not done recursively; a queue is used, instead of the implied stack of recursion.
  - To be seen later in the course (Breadth-First Search)

B-Trees

- So far, we have assumed that we can store an entire data structure in the main memory of a computer.
- Often, the volume of data is too large

➔We must have the data structure reside on disk.

➔The rules of the game change, because the Big-Oh model is no longer meaningful.

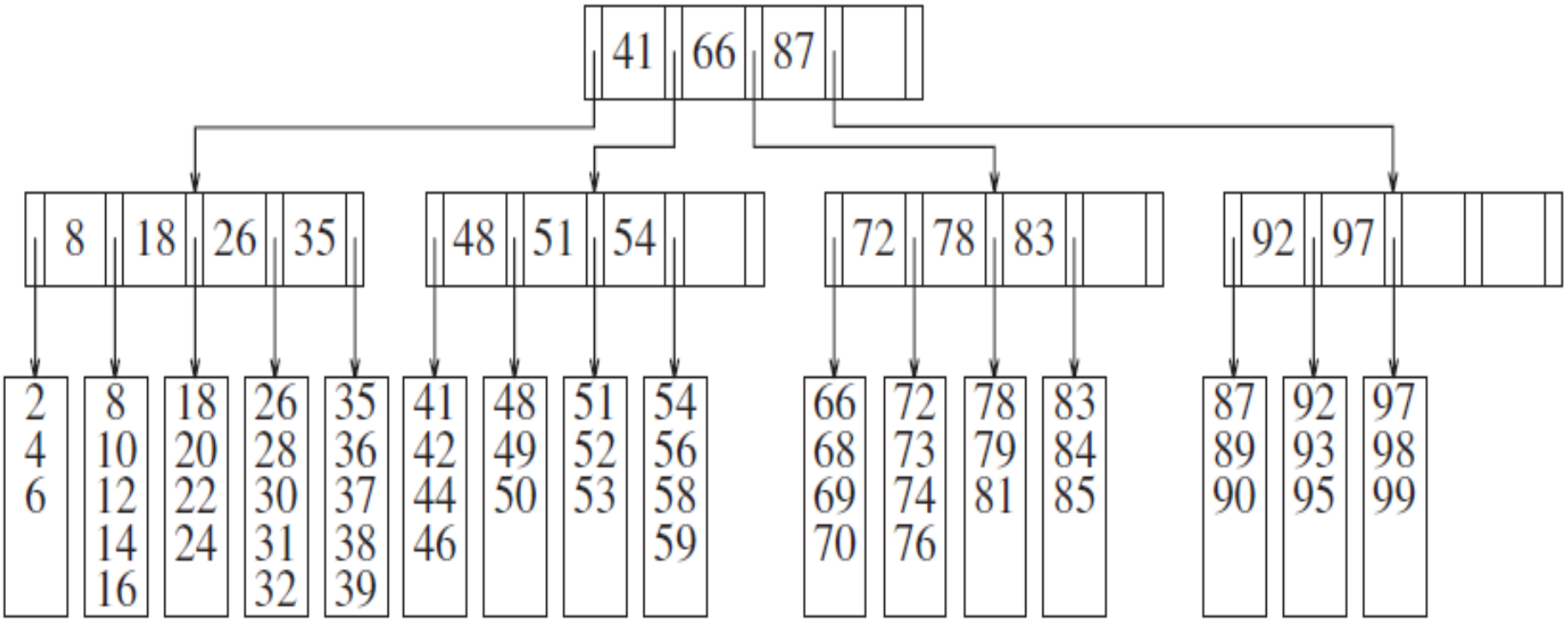
- Problem: a Big-Oh analysis assumes that all operations are equal, which is not true, especially when disk I/O is involved.
- Billions of instructions executed per second on modern computers; only ~120 disk I/Os per second
- Processor speeds increasing much faster than disk I/O

B-trees Motivation

- Need to reduce disk accesses as much as possible to reduce running time.
- We are willing to write complicated code to do this, because machine instructions are essentially free.
- A BST will not work, since the typical AVL tree is close to optimal height. $logN$ is the best we can reach with BST
- What is the solution?

Solution: M-ary search trees

- If we have more branching, we have less height.
- While a perfect binary tree of 31 nodes has five levels, a 5-ary tree of 31 nodes has only three levels



- An **M-ary search tree** allows M-way branching ➔ As branching increases, the depth decreases.
- Whereas a complete binary tree has height roughly $\log_2 N$, a complete M-ary tree has height roughly $\log_M N$.
- M-ary search tree can be created in the same way as a BST.
- In a BST, we need one key to decide which of two branches to take. In an M-ary search tree, we need $M - 1$ keys to decide.
- To make this scheme efficient in the worst case, we need to ensure that the M-ary search tree is balanced in some way. Otherwise, like a BST, it could degenerate into a linked list.
- In fact, we want an even more restrictive balancing condition so that an M-ary search tree does not degenerate to even a BST.

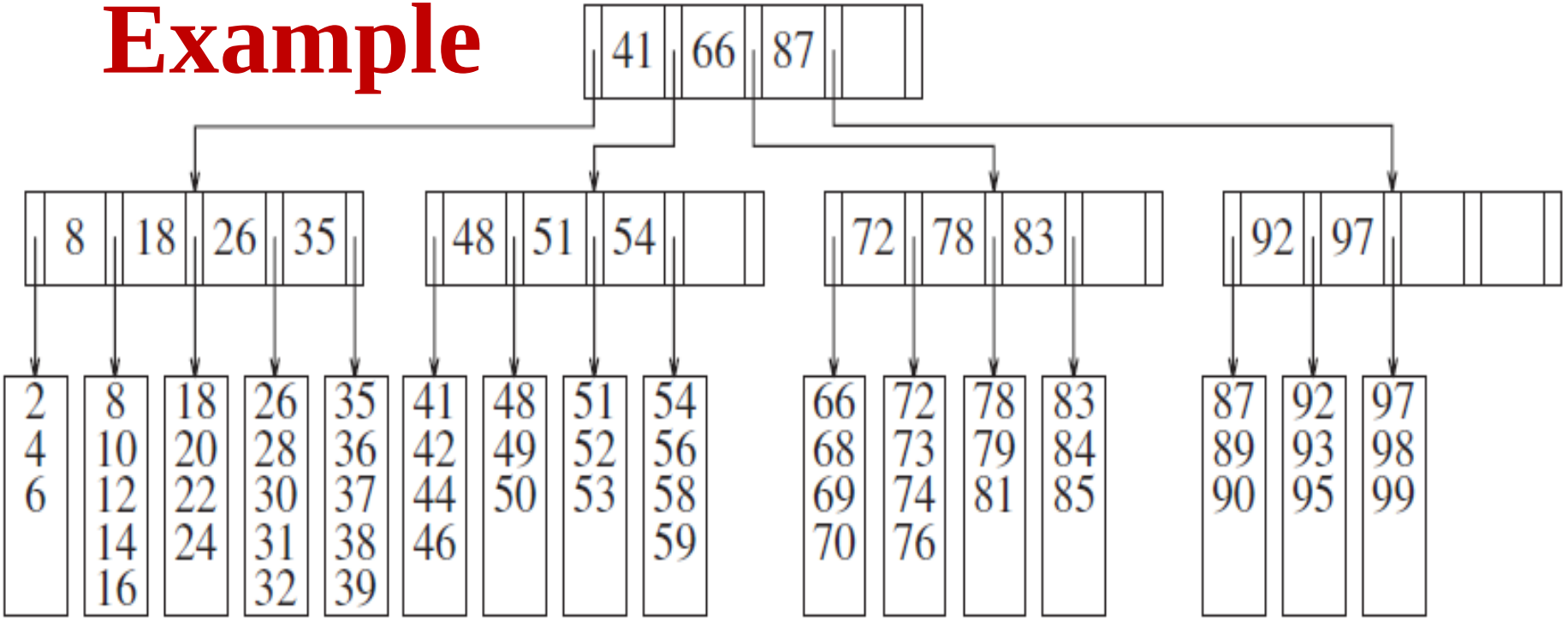
B-Trees (B+ Trees)

A B-tree of order M is an M-ary tree such that:

1. The data items are stored at leaves.
2. The nonleaf nodes store up to $M-1$ keys to guide the searching; key i represents the smallest key in subtree $i+1$
3. The root is either a leaf or has between two and M children.
4. All nonleaf nodes (except the root) have between $\lceil M/2 \rceil$ and M children. (avoids degeneration into binary tree)
5. All leaves are at the same depth and have between $\lceil L/2 \rceil$ and L data items, for some L (the determination of L is described shortly).

N.B.: Rules 3 and 5 must be relaxed for the first *L* insertions.

Example



- All nonleaf nodes have between three and five children (and thus between two and four keys);
- The root could possibly have only two children
- Here, L = 5. It happens that L = M
- L = 5 ➔ each leaf has between three and five data items

Choice of M and L: Example

- Each node represents a disk block, so we choose M and L on the basis of the size of the items that are being stored.
- Suppose that we have 10,000,000 data items, each key is 32 bytes (e.g. a name), a record is 256 bytes, and 1 block holds 8,192 bytes.
- In a B-tree of order M, we would have $M-1$ keys, for a total of $32M - 32$ bytes, plus M branches.
- Since each branch is essentially a ptr to another disk block, we can assume that a branch is 4 bytes ➔ the branches use $4M$ bytes.
- The total memory requirement for a nonleaf node is thus $36M-32$.
- The largest value of M for which this is no more than 8,192 is 228 ➔ we choose $M = 228$.
- Since each data record is 256 bytes, we would be able to fit 32 records in a block. Thus we would choose $L = 32$.

- $M = 228$. $L = 32$.

➔each leaf has between 16 and 32 data records

➔each internal node (except the root) branches in at least 114 ways.

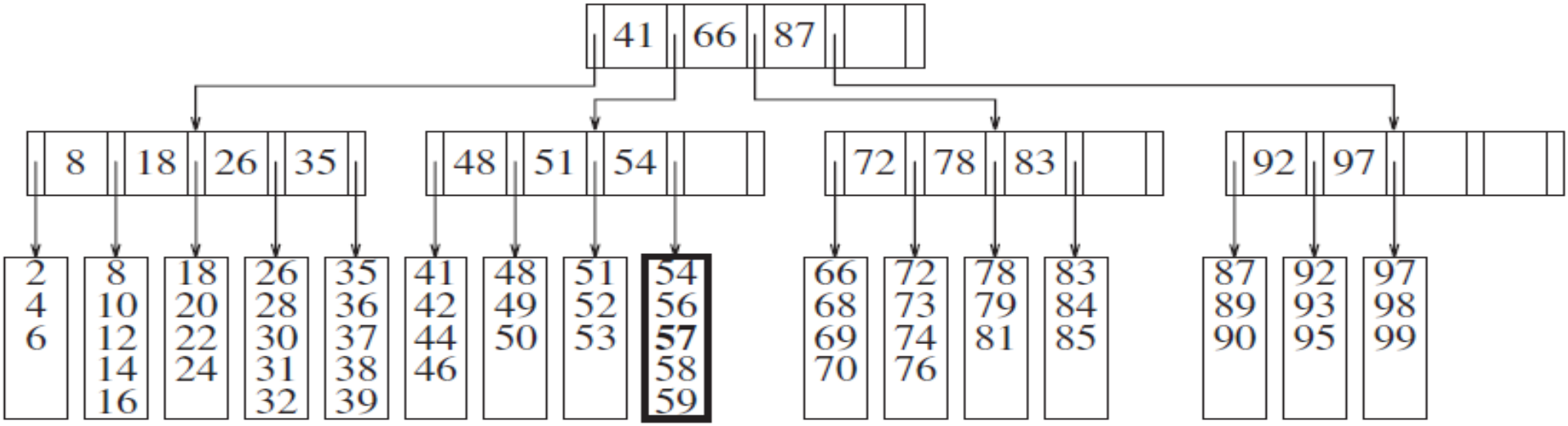
- Since there are 10,000,000 records, there are, at most, 625,000 leaves.

➔In the worst case, leaves would be on level 4.

➔In more concrete terms, the worst-case number of accesses is given by approximately $\log_{M/2} N$, give or take 1. (For example, the root and the next level could be cached in main memory, so that over the long run, disk accesses would be needed only for level 3 and deeper.)

Insertion into a B-Tree

Insert 57 into previous B-Tree

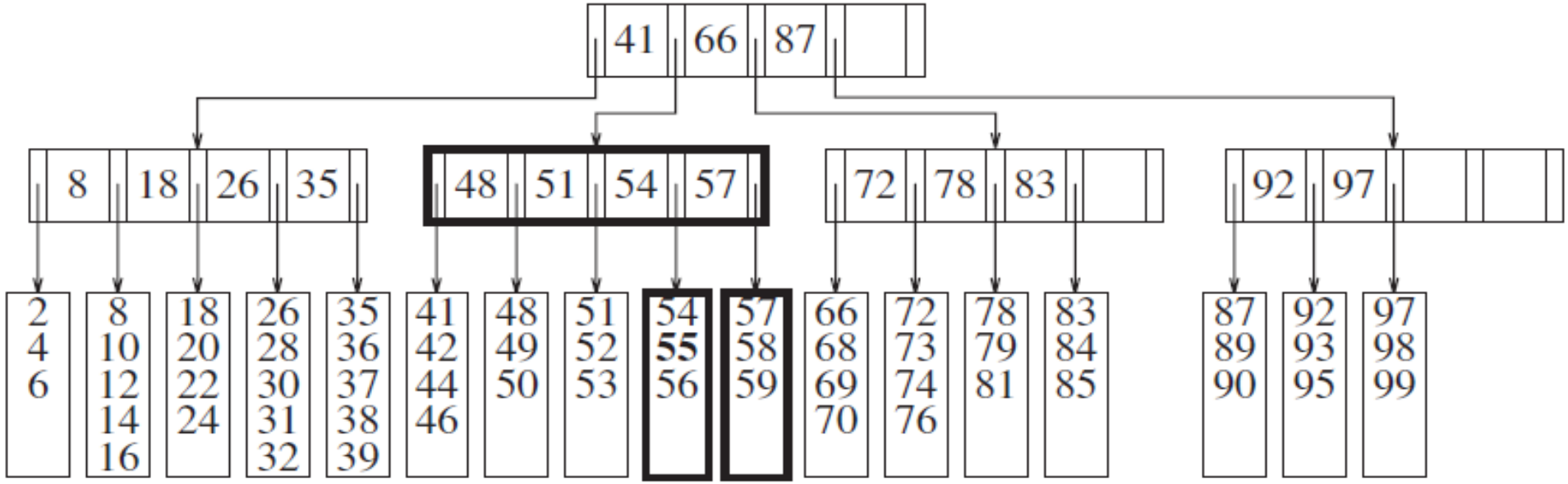


A search down the tree reveals that 57 not in the tree

Leaf node not full ➔ no problem. Can be added as fifth item then data is reorganised (negligible cost)

(Just an extra cost of writing on the disk)

- Insert 55 into previous (resulting) B-Tree
- Upon searching down the tree, leaf node found full
- Since we now have L+1 items, we split them into two leaves each containing the minimum required
- We form two leaves with three items each.
- Two disk accesses are required to write these leaves, and a third disk access is required to update the parent.
- Note that in the parent, both keys and branches change, but they do so in a controlled way that is easily calculated.



- Although splitting nodes is time-consuming (at least two additional disk writes), it is a relatively rare occurrence.
- E.g. if $L = 32$, then when a node is split, two leaves with 16 and 17 items, respectively, are created.

➔For the leaf with 17 items, we can perform 15 more insertions without another split.

➔More generally, for every split, there are roughly $L/2$ nonsplits.

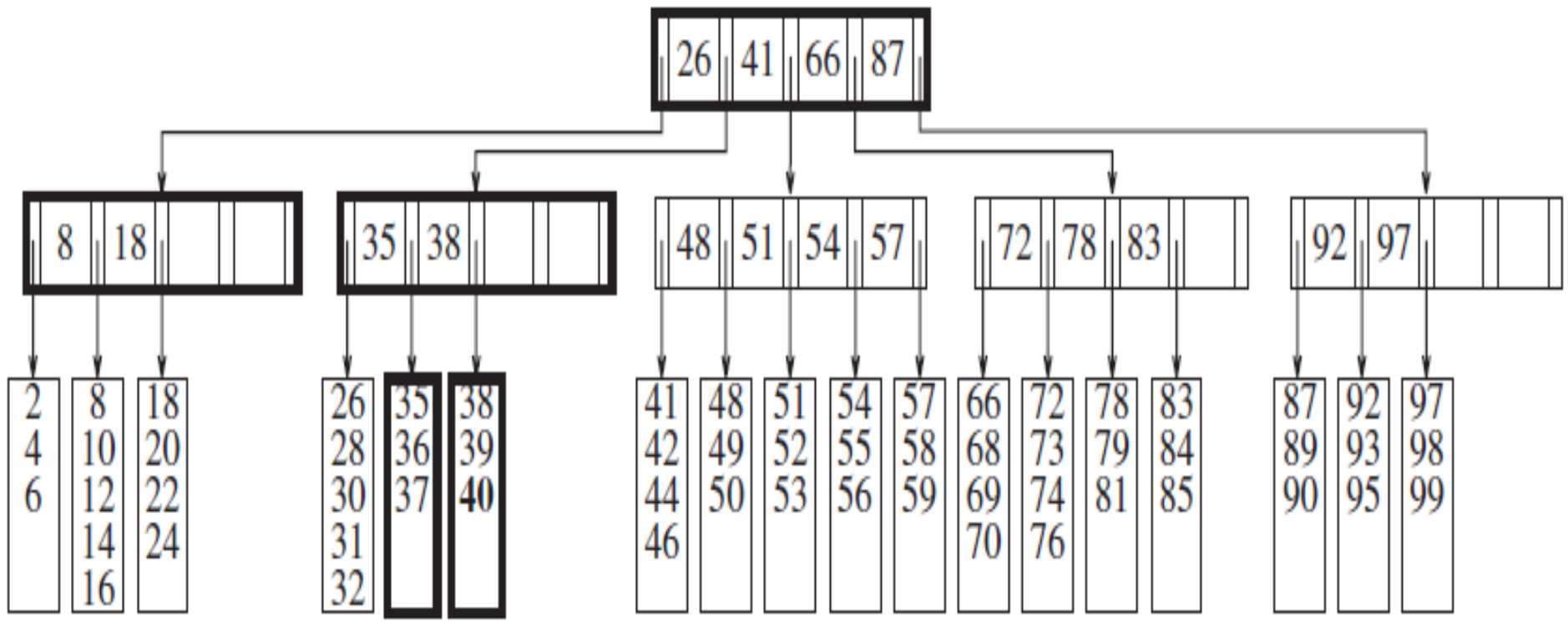
- Previous case: the node splitting worked because the parent did not have its full complement of children. What would happen if it did?
- Suppose, that we insert 40 into the previous (resulting) B-tree
- We must split the leaf containing the keys 35 through 39, and now 40, into two leaves.

➔ Parent has six children (only 5 allowed)

➔Split the parent.

➔When parent is split, we must update the values of the keys and also the parent's parent, thus incurring an additional two disk writes (so this insertion costs five disk writes).

- However, once again, the keys change in a very controlled manner, although the code is certainly not simple because of a lot of cases


26 41 66 87
8 18
35 38
48 51 54 57
72 78 83
92 97
2 4 6
8 10 12 14 16
18 20 22 24
26 28 30 31 32
35 36 37
38 39 40
41 42 44 46
48 49 50
51 52 53
54 55 56
57 58 59
66 68 69 70
72 73 74 76
78 79 81
83 84 85
87 89 90
92 93 95
97 98 99

- When a nonleaf node is split, its parent gains a child.
- If the parent already has reached its limit of children, then we continue splitting nodes up the tree until :
  - either we find a parent that does not need to be split or
  - we reach the root.
    - If we split the root, then we have two roots (unacceptable!)
    - Create a new root that has the split roots as its two children. (This is why the root is granted the special two-child minimum exemption. It also is the only way that a B-tree gains height.
- **Notes**: splitting all the way up to the root is an exceptionally rare event.
    - This is because a tree with four levels indicates that the root has been split three times throughout the entire sequence of insertions (assuming no deletions have occurred).
- The splitting of any nonleaf node is also quite rare.

Deletion from a B-Tree

- Deletion is performed by finding the item that needs to be removed and then removing it.
- Problem if the leaf it was in had the minimum number of data items ➔ it is now below the minimum.
- We can rectify this situation by adopting a neighboring item, if the neighbor is not itself at its minimum.
- If the neighbor is already at its minimum, then we can combine with the neighbor to form a full leaf.

➔ This means that the parent has lost a child. If this causes the parent to fall below its minimum, then it follows the same strategy.

- This process could percolate all the way up to the root.
- If a root is left with one child as a result of the adoption process, then we remove the root and make its child the new root of the tree.
- This is the only way for a B-tree to lose height.

Suppose we want to delete 99 from the previous resulting) B-Tree