- Nodes of a binary tree
- Binary tree representation
- Binary tree traversal

1.1 INTRODUCTION OF BINARY TREES

A binary tree is a finite set of nodes that is either empty or it consists of a root and two disjoint binary trees called the left sub tree and the right sub tree.

A **binary tree** is a **rooted** tree in which every node has at most two children. A **full binary tree** is a tree in which every node has zero or two children. Also known as a **proper binary tree**. A **perfect binary tree** is a full binary tree in which all **leaves** (vertices with zero children) are at the same **depth** (distance from the **root**, also called **height**).
Sometimes the **perfect binary tree** is called the **complete binary tree**. An **almost complete binary tree** is a tree where for a right child, there is always a left child, but for a left child there may not be a right child.

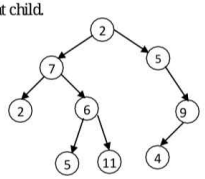
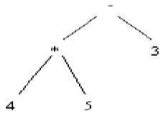


Figure: Binary Tree

A **binary tree** is similar to a tree, but not quite the same. For a binary tree, each node can have zero, one, or two children. In addition, each child node is clearly identified as either the left child or the right child.

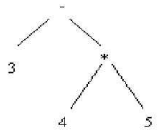
As an example, here you can see the **binary expression tree** for the expression 4 * 5 - 3. Note that a child drawn to the left of its parent is meant to be the left child and that a child drawn to the right of its parent is meant to be the right child.



Expression: 4*5-3

Figure: Binary Expression Tree

Reversing the left to right order of any siblings gives a different binary tree. For example, reversing the sub trees rooted at * and at 3 gives the following different binary tree.
It happens to be the binary expression tree for 3 - 4 * 5.



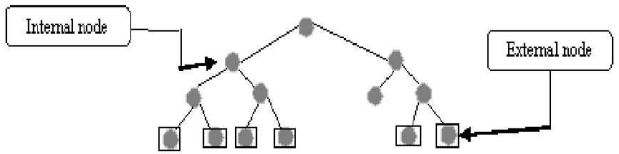
Expression: 3-4*5

Figure: Binary Expression Tree

1.2 BINARY TREES – EXTERNAL AND INTERNAL NODES

In a binary tree, all the nodes must have the same number of children. All internal nodes have two children and external nodes have no children. A binary tree with n internal nodes has n + 1 external nodes. Let s1, s2, …, sn be the internal nodes, in order. Then,

key(s1) < key(s2) < … < key(sn).



- Internal nodes - External nodes

Figure: Internal and External node representation

Minimum number of nodes in a binary tree whose height h is **h+1** and maximum number of nodes is about $2^{h+1} - 1$. A binary tree with N nodes (internal and external) has N-1 edges. A binary tree with N internal nodes has N+ 1 external node.

1.3 BINARY TREE REPRESENTATIONS

A full binary tree of depth k is a binary tree of depth k having 2^k-1 nodes. This is the maximum number of the nodes such a binary tree can have. A very elegant sequential representation for such binary trees results from sequentially numbering the nodes, starting with nodes on level 1, then those on level 2 and so on. Nodes on any level are numbered from left to right. This numbering scheme gives us the definition of a complete binary tree. A binary tree with n nodes and a depth k is complete iff its nodes correspond to the nodes which

are numbered one to n in the full binary tree of depth k. The nodes may be represented in an array or using a linked list.

1.3.1 Array Representation of Trees

This method is easy to understand and implement. It's very useful for certain kinds of tree applications, such as heaps, and fairly useless for others.
Steps to implement binary trees using arrays:

- Take a complete binary tree and number its nodes from top to bottom, left to right.
- The root is 0, the left child 1, the right child 2, the left child of the left child 3, etc.
- Put the data for node i of this tree in the ith element of an Array.
- If you have a partial (incomplete) binary tree, and node i is absent, put some value that represents "no data" in the ith position of the array.

Three simple formulae allow you to go from the index of the parent to the index of its children and vice versa:

- if index(parent) = N, index(left child) = 2*N+1
- if index(parent) = N, index(right child) = 2*N+2
- if index(child) = N, index(parent) = (N-1)/2 (integer division with truncation)

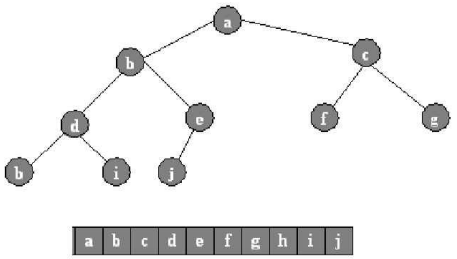


Figure: Array Representation of Trees

Advantages of linear representation:

1. Simplicity.
2. Given the location of the child (say, k), the location of the parent is easy to determine (k / 2).

Disadvantages of linear representation:

1. Additions and deletions of nodes are inefficient, because of the data movements in the array.
2. Space is wasted if the binary tree is not complete. That is, the linear representation is useful if the number of missing nodes is small.

Figure: Linked Trees

1.3 BINARY TREE TRAVERSAL

There are many operations that we often want to perform on trees. One notion that arises frequently is the idea of traversing a tree or visiting each node in the tree exactly once. Traversing a tree means visiting all the nodes of a tree in order. A full traversal produces a linear order for the information in a tree. This linear order may be familiar and useful. If we traverse the standard ordered binary tree *in-order*, then we will visit all the nodes in sorted order.

Types of Traversals are:

- Preorder traversal
    1. Visit the root
    2. Traverse the left sub tree
    3. Traverse the right sub tree

- In order traversal
    1. Traverse the left sub tree
    2. Visit the root
    3. Traverse the right sub tree

1. Post order traversal
    1. Traverse the left subtree
    2. Traverse the right subtree
    3. Visit the root

1.3.1 Pre-order traversal

1. Start at the root node
2. Traverse the left subtree
3. Traverse the right subtree

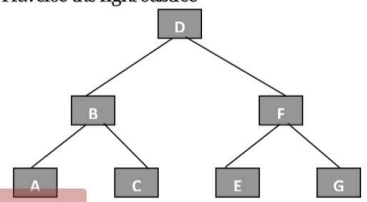


Figure: Preorder Traversal

The nodes of this tree would be visited in the order: **D B A C F E G**

Figure: Linked Trees

1.3 BINARY TREE TRAVERSAL

There are many operations that we often want to perform on trees. One notion that arises frequently is the idea of traversing a tree or visiting each node in the tree exactly once. Traversing a tree means visiting all the nodes of a tree in order. A full traversal produces a linear order for the information in a tree. This linear order may be familiar and useful. If we traverse the standard ordered binary tree *in-order*, then we will visit all the nodes in sorted order.

Types of Traversals are:

- Preorder traversal
    1. Visit the root
    2. Traverse the left sub tree
    3. Traverse the right sub tree

- In order traversal
    1. Traverse the left sub tree
    2. Visit the root
    3. Traverse the right sub tree

1. Post order traversal
    1. Traverse the left subtree
    2. Traverse the right subtree
    3. Visit the root

1.3.1 Pre-order traversal

1. Start at the root node
2. Traverse the left subtree
3. Traverse the right subtree

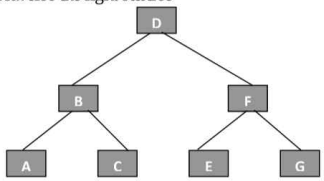


Figure: Preorder Traversal

The nodes of this tree would be visited in the order: **D B A C F E G**

Procedure for Pre-Order Traversal

Preorder (current node: tree pointer);

{Current node is a pointer to a node in a binary tree. For full
Tree traversal, pass preorder, the pointer to the top of the tree}

{//preorder

```
If current node <> nil then
{
Write (current node ^.data);

Preorder (current node ^.left child);

Preorder (current node ^.right child);
 }
}// preorder
```

1.3.2 In-order traversal

1. Traverse the left sub tree
2. Visit the root node
3. Traverse the right sub tree

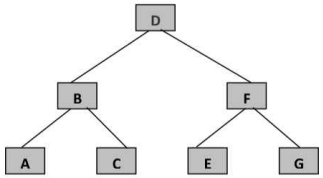


Figure: In-Order Traversal

The nodes of this tree would be visited in the order: **A B C D E F G**

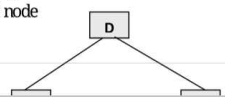
Procedure for In Order Traversal:

```
In order (currentnode: treepointer);
{Current node is a pointer to a node in a binary tree. For full

Tree traversal, pass in order, the pointer to the top of the tree}
{// in order
If current node <> nil
Then
{
In order (current node ^. left child);
Write (currentnode^.data);
In order (currentnode^.rightchild);
}
}// in order
```

1.3.3 Post-order traversal

1. Traverse the left subtree
2. Traverse the right subtree
3. Visit the root node



```
If current node <> nil then
{
Write (current node ^.data);

Preorder (current node ^.left child);

Preorder (current node ^.right child);
 }
}// preorder
```

1.3.2 In-order traversal

1. Traverse the left sub tree
2. Visit the root node
3. Traverse the right sub tree

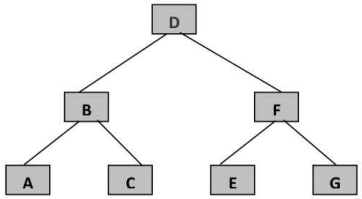


Figure: In-Order Traversal

The nodes of this tree would be visited in the order: **A B C D E F G**

Procedure for In Order Traversal:

```
In order (currentnode: treepointer);
{Current node is a pointer to a node in a binary tree. For full

Tree traversal, pass in order, the pointer to the top of the tree}
{// in order
If current node <> nil
Then
{
In order (current node ^. left child);
Write (currentnode^.data);
In order (currentnode^.rightchild);
}
}// in order
```

1.3.3 Post-order traversal

1.3.3 Post-order traversal

1. Traverse the left subtree
2. Traverse the right subtree
3. Visit the root node

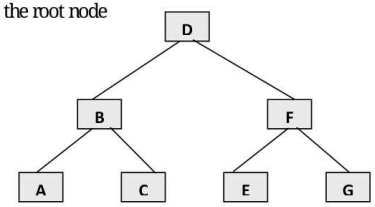


Figure: Post order Traversal

The nodes of this tree would be visited in the order: **A C B E G F D**

Procedure for Post Order Traversal

```
Post order (current node: tree pointer);
{Current node is a pointer to a node in a binary tree. For full
Tree traversal, pass post order, the pointer to the top of the tree}
{// post order
If current node<> nil then
{
Post order (current node ^.left child);
Post order (currentnode^.rightchild);
Write (currentnode^.data);
}
}//post order
```

1.4 BINARY SEARCH TREES

1.4.1 Introduction of Binary Trees

A **binary search tree** is a binary tree in which the data in the nodes are orderec particular way. To be precise, starting at any given node, the data in any nodes of its le tree must all be less than the item in the given node, and the data in any nodes of its rig tree must be greater than or equal to the data in the given node. For numbers th obviously be done. For strings, alphabetical ordering is often used. For records of c comparison based on a particular field (the **key field**) is often used. An example of a search tree is shown in Figure.

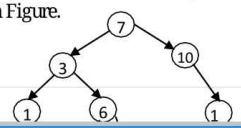


Figure: Binary Search Tree

The above figure shows a binary search tree of size 9 and depth 3, with root 7 and leaves 1, 4, 7 and 13. Every node (object) in a binary tree contains information divided into two parts. The first one is proper to the structure of the tree, that is, it contains a key field (the part of information used to order the elements), a parent field, a left child field, and a right child field. The second part is the object data itself. It can be endogenous (that is, data resides inside the tree) or exogenous (that is nodes only contains a references to the object's data). The root node of the tree has its parent field set to null. Whenever a node does not have a right child or a left child, then the corresponding field is set to null.

A binary search tree is a binary tree with more constraints. If x is a node with key value key [x] and it is not the root of the tree, then the node can have a left child (denoted by left [x]), a right child (right [x]) and a parent (p [x]). Every node of a tree possesses the following *Binary* **Search Tree properties:**

1. For all nodes y in left sub tree of x, key [y] < key [x]
2. For all nodes y in right sub tree of x, key[y] > key [x]

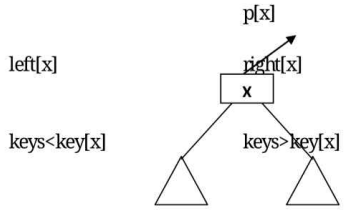
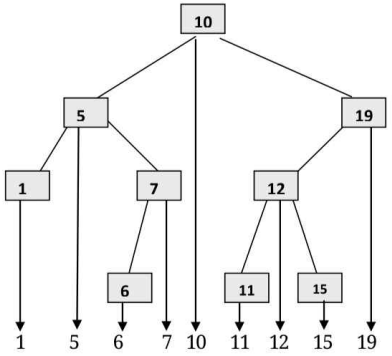


Figure: Keys of Binary Search Tree



2.2.1 Finding Minimum and Maximum

The minimum element of a binary search tree is the last node of the left roof, maximum element is the last node of the right roof. Therefore, we can find the minim the maximum by tracking on the left child and the right child, respectively, until ar sub tree is reached.

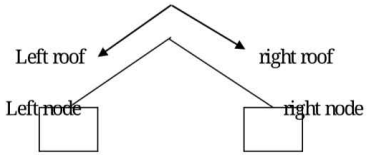


Figure: Left and Right Nodes

2.2.2 Searching in a Binary Search Tree

You can search for a desirable value in a Binary search tree through the following procedure.

Search for a matching node

1. Start at the root node as current node
2. If the search key's value matches the current node's key then found a match
3. If search key's value is greater than current node's key
    1. If the current node has a right child, search right
    2. Else, no matching node in the tree
4. If search key is less than the current node's key
    1. If the current node has a left child, search left
    2. Else, no matching node in the tree

Example:
Search for 45 in the tree:

1. Start at the root, 45 is greater than 25, search in right sub tree
2. 45 is less than 50, search in 50's left sub tree
3. 45 is greater than 35, search in 35's right sub tree
4. 45 is greater than 44, but 44 has no right sub tree so 45 is not in the BST

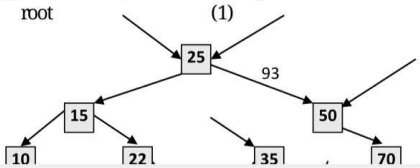


Figure: Insertion in a Binary Search Tree

We can insert a node in the binary search tree using the following procedure:

1. Always insert new node as leaf node
2. Start at root node as current node
3. If new node's key < current's key
   1. If current node has a left child, search left
   2. Else add new node as current's left child
4. If new node's key > current's key
   1. If current node has a right child, search right
   2. Else add new node as current's right child

TREE-INSERT (T, z)

```
y ← NIL
x ← root [T]  // Assigning root of tree
while x ≠ NIL do
y ← x
if key [z] <  key[x]
then x ← left[x] // Traversing left tree
else x ← right[x] // Traversing right tree
p[z] ← y
if y = NIL
then root [T] ← z
else if key [z] < key[y]
then left [y] ← z  // Inserting as left child
else right [y] ← z // Inserting as right child
```

3.2.2.4 Deleting an element

Deleting an item from a binary search tree is little harder than inserting one. Before writing code, let's consider how to delete nodes from a binary search tree in an abstract fashion Here's a BST from which we can draw examples during the discussion:

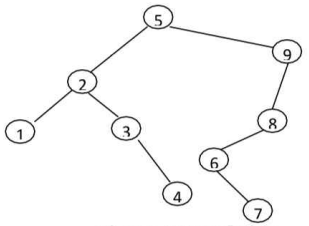


Figure: BST Deletion

It is more difficult to remove some nodes from this tree than to remove others. Here, let u see three distinct cases, described in detail below in terms of the deletion of a nod designated *p*.

Case 1: *p* has no right child

It is trivial to delete a node with no right child, such as node 1, 4, 7, or 8 above. We replac the pointer leading to *p* by *p*'s left child, if it has one, or by a null pointer, if not. In othe words, we replace the deleted node by its left child. For example, the process of deleting nod 8 looks like this:

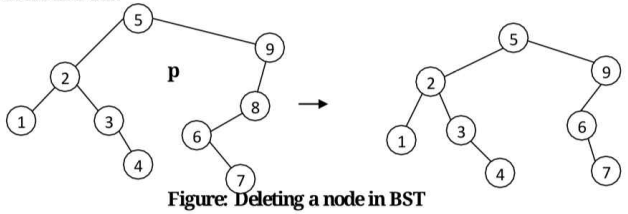
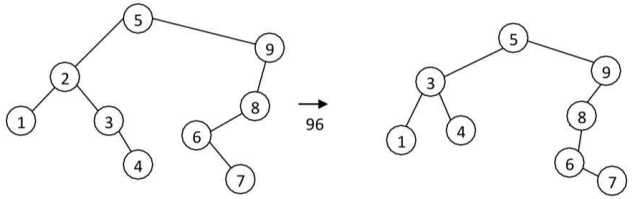


Figure: Deleting a node in BST

Case 2: p's right child has no left child

This case deletes any node *p* with a right child *r* that itself has no left child. Nodes 2, 3, and in the tree above are examples. In this case, we move *r* into *p*'s place, attaching *p*'s former le subtree, if any, as the new left subtree of *r*. For instance, to delete node 2 in the tree above we can replace it by its right child 3, giving node 2's left child 1 to node 3 as its new le child. The process looks like this:



p r

Figure: Deleting a node in BST

Case 3: p's right child has a left child

This is the "hard" case, where p's right child r has a left child. But if we approach it prope we can make it make sense. *Let p's* in order successor that is, the node with the smallest va greater than p, be s.
Then, our strategy is to detach s from its position in the tree, which is always an easy thing do, and put it into the spot formerly occupied by p, which disappears from the tree. In example, to delete node 5, we move inorder successor node 6 into its place, like this:

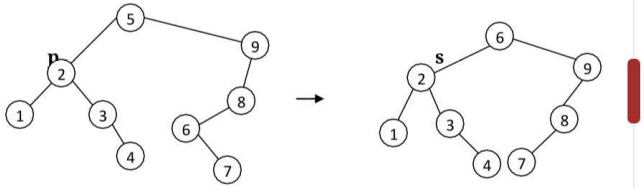


Figure: Deleting a node in BST

But how to know that node *s* exists and that we can delete it easily? We know that it exi because otherwise, this would be case 1 or case 2 (consider their conditions). We can eas detach from its position for a more subtle reason: *s* is the inorder successor of *p* and therefore has the smallest value in *p*'s right subtree, so *s* cannot have a left child. (If it d then this left child would have a smaller value than *s*, so it, rather than *s*, would be *p*'s inor successor.) Because *s* doesn't have a left child, we can simply replace it by its right child any. This is the mirror image of case 1.

TREE-DELETE (T, z)

```
if left [z] = NIL    .OR.    right[z] = NIL
then y ← z
else y ← TREE-SUCCESSOR (z)
if left [y] ≠ NIL
then x ← left[y] // Assigning left node if it exists
else x ← right [y] // Assigning right node if it exists
if x ≠ NIL
then p[x] ← p[y]
if p[y] = NIL
```

then root [T] ← x
else if y = left [p[y]]
then left [p[y]] ← x
else right [p[y]] ← x
if y ≠ z
then key [z] ← key [y]
if y has other field, copy them, too return y

Tree-Successor(x)

if $right[x] \neq NIL$
then return Tree-Minimum(*right*[*x*])
$y \leftarrow p[x]$
while $(y \neq NIL)$ and (x = *right*[*y*])
do $x \leftarrow y$
$y \leftarrow p[y]$
return *y*

1.4 LET US SUM UP

Non-linear data structure is a structure in which data's are not stored in a sequentia or Trees and Graphs are examples of Non linear data structure. A tree comprises o arrangement of *nodes* each of which holds information. Nodes are linked by *arcs* (or *edge* Each node has zero or more **child nodes**, which are below it in the tree (by conventio computer science, trees grow down - not up as they do in nature). A **child** is a node conne directly below the starting node. Nodes with the same parent are called **siblings**. The topr node in a tree is called the **root node**. A **branch** is a sequence of nodes such that the fir the parent of the second; the second is the parent of the third, etc. The **leaves** of a (sometimes also called **external nodes**) are those nodes with no children. The other node the tree are called **non-leaves** (or sometimes **internal nodes**). The **height** of a tree is maximum length of a branch from the root to a leaf.

A **binary tree** is a **rooted** tree in which every node has at most two children. A **full bin tree** is a tree in which every node has zero or two children. Also known as a **proper bin tree**. A binary tree with N nodes (internal and external) has N-1 edges. A binary tree wi internal nodes has N+ 1 external node. Binary trees can be represented in array representa as well as linked representation. Binary trees can be traversed in preorder, in order and order traversals.

A **binary search tree** is a binary tree in which the data in the nodes are ordered particular way. Insertion, Deletion and Search operations can be performed in a binary se

2.2 AVL TREES

Balanced binary tree

- The disadvantage of a binary search tree is that its height can be as large as N-1
- This means that the time needed to perform insertion and deletion and many other operations can be O(N) in the worst case
- We want a tree with small height
- A binary tree with N node has height at least $\Theta(\log N)$
- Thus, our goal is to keep the height of a binary search tree O(log N)
- Such trees are called balanced binary search trees. Examples are AVL tree, red-black tree.

AVL tree

Height of a node

- The height of a leaf is 1. The height of a null pointer is zero.
- The height of an internal node is the maximum height of its children plus 1

Note that this definition of height is different from the one we defined previously (we defined the height of a leaf as zero previously).

- An AVL tree is a binary search tree in which
- for *every* node in the tree, the height of the left and right subtrees differ by at most 1.

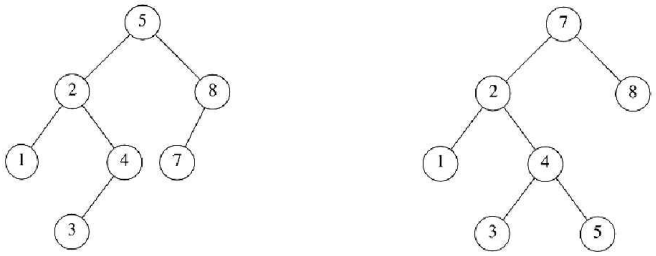


Figure 4.32 Two binary search trees. Only the left tree is AVL.

- Let x be the root of an AVL tree of height h
- Let N_h denote the minimum number of nodes in an AVL tree of height h
- Clearly, $N_i \geq N_{i-1}$ by definition
- We have

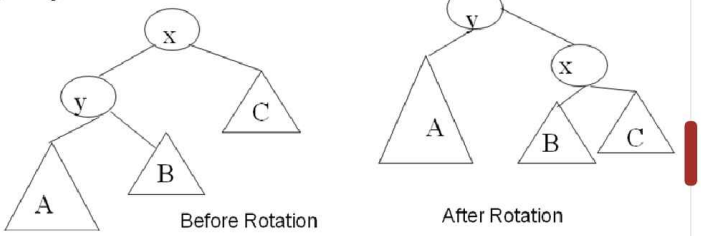
$$N_h \geq N_{h-1} + N_{h-2} + 1$$
$$\geq 2N_{h-2} + 1$$
$$> 2N_{h-2}$$

- By repeated substitution, we obtain the general form
- $N_h > 2^i N_{h-2}$
- The boundary conditions are: $N_1=1$ and $N_2=2$. This implies that $h = O(\log N_h)$.
- Thus, many operations (searching, insertion, deletion) on an AVL tree will take O(log N) time.

Rotations

- When the tree structure changes (e.g., insertion or deletion), we need to transform the tree to restore the AVL tree property.
- This is done using single rotations or double rotations.

e.g. Single Rotation



- Since an insertion/deletion involves adding/deleting a single node, this can only increase/decrease the height of some subtree by 1
- Thus, if the AVL tree property is violated at a node x, it means that the heights of left(x) ad right(x) differ by exactly 2.
- Rotations will be applied to x to restore the AVL tree property.

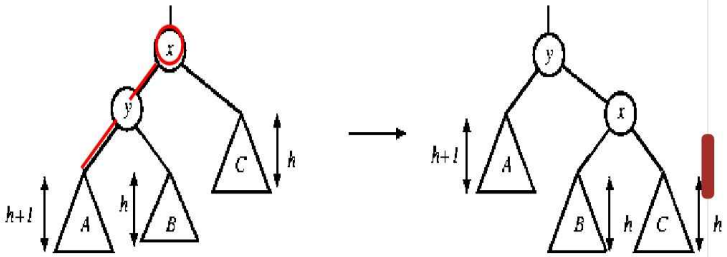
Insertion

- First, insert the new key as a new leaf just as in ordinary binary search tree
- Then trace the path from the new leaf towards the root. For each node x encountered, check if heights of left(x) and right(x) differ by at most 1.
- If yes, proceed to parent(x). If not, restructure by doing either a single rotation or a double rotation [next slide].
- For insertion, once we perform a rotation at a node x, we won't need to perform any rotation at any ancestor of x.

- Let x be the node at which left(x) and right(x) differ by more than 1
- Assume that the height of x is h+3
- There are 4 cases
  - Height of left(x) is h+2 (i.e. height of right(x) is h)
    - Height of left(left(x)) is h+1 ⇒ single rotate with left child
    - Height of right(left(x)) is h+1 ⇒ double rotate with left child
  - Height of right(x) is h+2 (i.e. height of left(x) is h)
    - Height of right(right(x)) is h+1 ⇒ single rotate with right child
    - Height of left(right(x)) is h+1 ⇒ double rotate with right child
- **Note:** Our test conditions for the 4 cases are different from the code shown in the textbook. These conditions allow a uniform treatment between insertion and deletion.

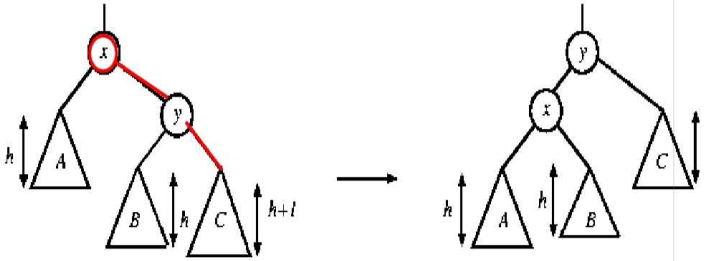
Single rotation

- The new key is inserted in the subtree A.
- The AVL-property is violated at x
  - height of left(x) is h+2
  - height of right(x) is h.



Rotate with left child

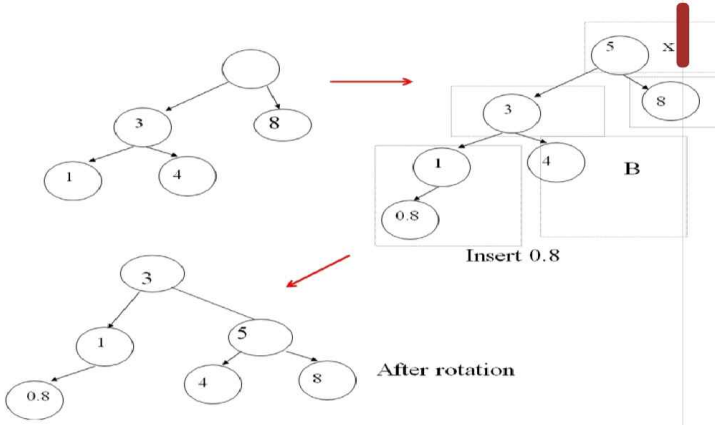
The new key is inserted in the subtree C.
The AVL-property is violated at x.



Rotate with right child

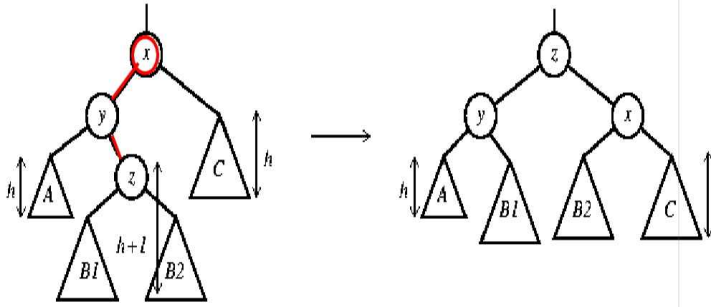
Single rotation takes O (1) time.
Insertion takes O (log N) time.

Example AVL Tree:



Double rotation

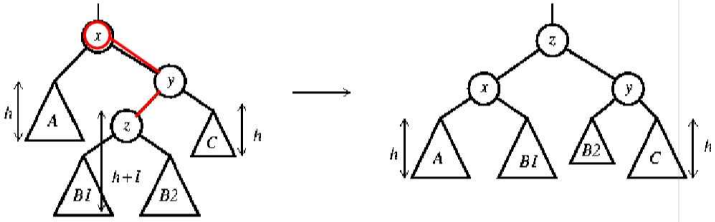
The new key is inserted in the subtree B1 or B2.
The AVL-property is violated at x.
x-y-z forms a zig-zag shape



Double rotate with left child

Also called left-right rotate.

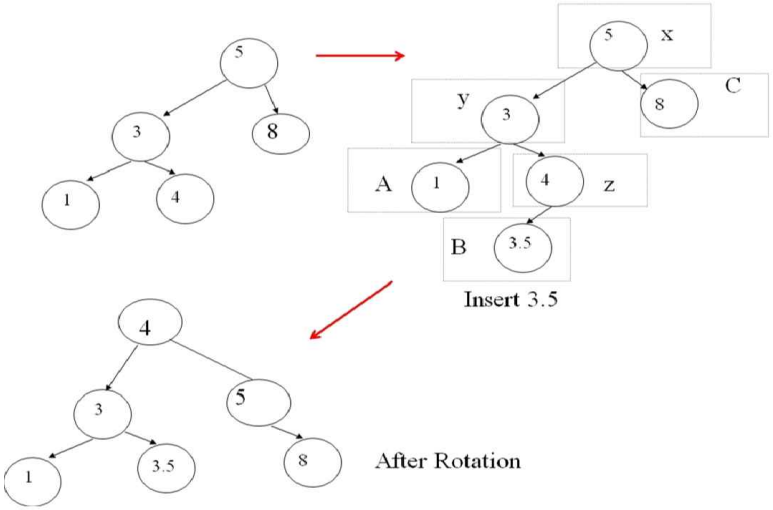
The new key is inserted in the sub tree B1 or B2.
The AVL-property is violated at x.



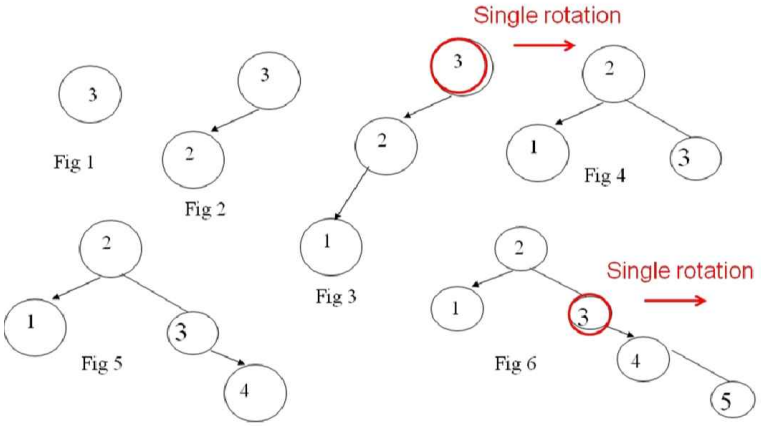
Double rotate with right child

also called right-left rotate.

AVL Tree:



An Extended Example:
Insert 3,2,1,4,5,6,7, 16,15,14



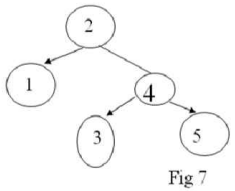


Fig 7

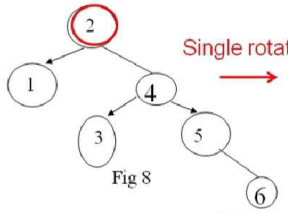


Fig 8

Single rotation

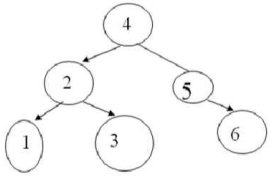


Fig 9

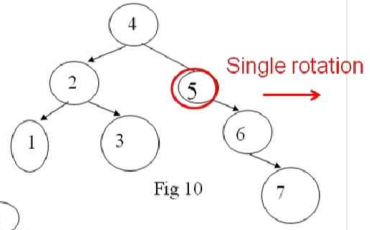


Fig 10

Single rotation

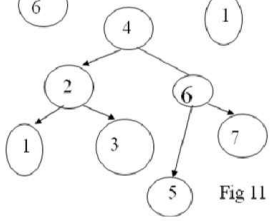


Fig 11

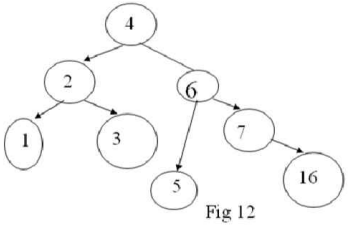


Fig 12

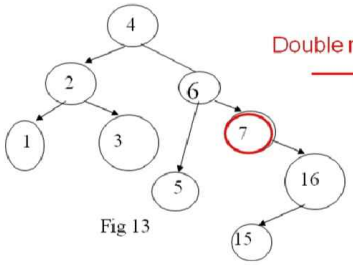


Fig 13

Double rotation

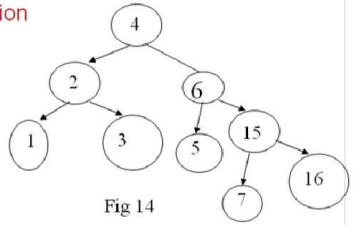


Fig 14

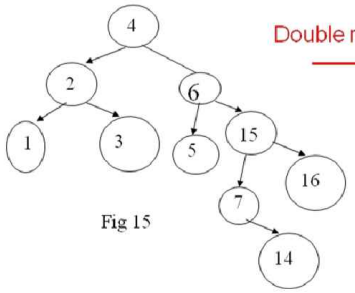


Fig 15

Double rotation

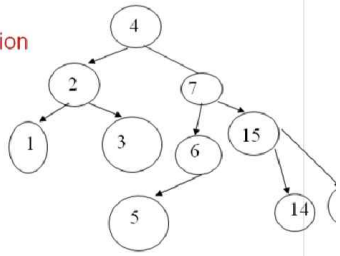


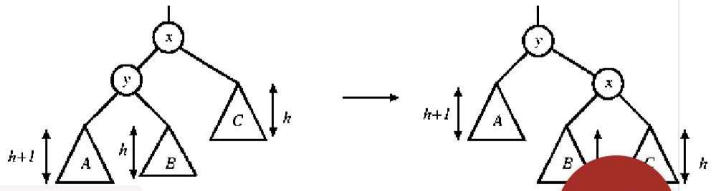
Fig 16

Deletion:

- Delete a node x as in ordinary binary search tree. Note that the last node deleted is leaf.
- Then trace the path from the new leaf towards the root.
- For each node x encountered, check if heights of left(x) and right(x) differ by at mos 1. If yes, proceed to parent(x). If not, perform an appropriate rotation at x. Th e a 4 cases as in the case of insertion.
- For deletion, after we perform a rotation at x, we may have to perform a rotation some ancestor of x. Thus, we must continue to trace the path until we reach the root.
- On closer examination: the single rotations for deletion can be divided into 4 cas (instead of 2 cases)
  - Two cases for rotate with left child
  - Two cases for rotate with right child

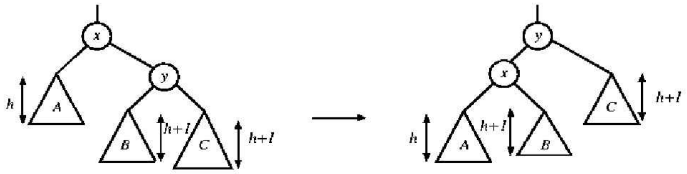
Single rotations in deletion

In both figures, a node is deleted in sub tree C, causing the height to drop to h. The height y is h+2. When the height of sub tree A is h+1, the height of B can be h or h+1. Fortunatel the same single rotation can correct both cases.



Rotate with left child

In both figures, a node is deleted in sub tree A, causing the height to drop to h. The height of y is h+2. When the height of sub tree C is h+1, the height of B can be h or h+1. A single rotation can correct both cases.



rotate with right child

Rotations in deletion

- There are 4 cases for single rotations, but we do not need to distinguish among them.
- There are exactly two cases for double rotations (as in the case of insertion)
- Therefore, we can reuse exactly the same procedure for insertion to determine which rotation to perform

Hashing

Hashing is function that maps each key to a location in memory. A key's location does not depend on other elements, and does not change after insertion unlike a sorted list. A good hash function should be easy to compute. With such a hash function, the dictionary operations can be implemented in O(1) time.

- Static Hashing
  - File Organization
  - Properties of the Hash Function
  - Bucket Overflow
  - Indices
- Dynamic Hashing
  - Underlying Data Structure
  - Querying and Updating
- Comparisons
  - Other types of hashing
  - Ordered Indexing vs. Hashing

Static Hashing

- Hashing provides a means for accessing data without the use of an index structure.

- Hashing provides a means for accessing data without the use of an index structure.
- Data is addressed on disk by computing a function on a search key instead.

Organization

- A bucket in a hash file is unit of storage (typically a disk block) that can hold one or more records.
- The hash function, h, is a function from the set of all search-keys, K, to the set of all bucket addresses, B.
- Insertion, deletion, and lookup are done in constant time.

Querying and Updates

- To insert a record into the structure compute the hash value $h(K_i)$, and place the record in the bucket address returned.
- For lookup operations, compute the hash value as above and search each record in the bucket for the specific record.
- To delete simply lookup and remove.

Properties of the Hash Function

- The distribution should be uniform.
  - An ideal hash function should assign the same number of records in each bucket.
- The distribution should be random.
  - Regardless of the actual search-keys, the each bucket has the same number of records on average
  - Hash values should not depend on any ordering or the search-keys

Bucket Overflow

- How does bucket overflow occur?
  - Not enough buckets to handle data
  - A few buckets have considerably more records then others. This is referred to as skew.
    - Multiple records have the same hash value
    - Non-uniform hash function distribution.

Solutions

- Provide more buckets then are needed.
- Overflow chaining
  - If a bucket is full, link another bucket to it. Repeat as necessary.
  - The system must then check overflow buckets for querying and updates. This is known as closed hashing.

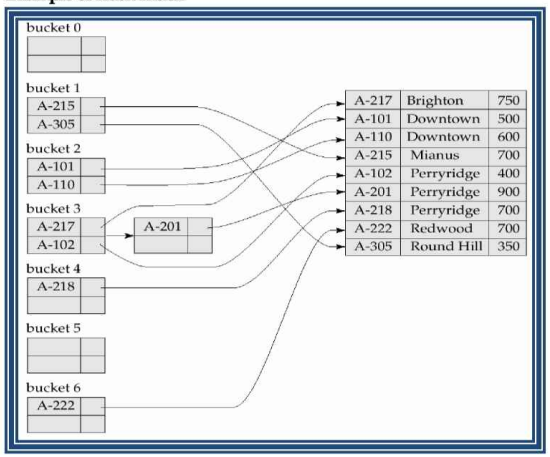
Alternatives

- Open hashing
  - The number of buckets is fixed
  - Overflow is handled by using the next bucket in cyclic order that has space.
    - This is known as linear probing.
- Compute more hash functions.
- Note: Closed hashing is preferred in database systems.

Indices

- A hash index organizes the search keys, with their pointers, into a hash file.
- Hash indices never primary even though they provide direct access.

Example of Hash Index



1.7 HEAP SORTING

This type of sorting starts by building a *heap* from the initial array. Since the maximum element of the array is stored at the root, data [0], it can be put into its correct final position by exchanging it with data [n-1].

If we then "discard" node *n-1* from the heap, we observe that data [0..(n-2)] can easily be made into a heap. The children of the root remain heaps, but the new root element may violate the heap property. All that is needed to restore the heap property, however, is one call

to reheapify_down(), which leaves a heap indata[0..(n-2)]. The heapsort algorithm then repeats this process for the heap of size *n-2* down to a heap of size 1.

```
heapsort(vector<Item>& v)
build_heap(v); // create a heap vector corresponding to v
for(i=v.size()-1; i>=1; i--)
swap(heap[0], heap[i]);
heap-size = heap-size - 1;
reheapify_down();

buildheap(vector<Item>& v)
for(i=0; i<v.size(); i++)
insert(v[i]);
```

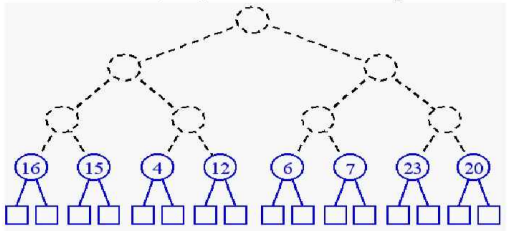
A more efficient buildheap() procedure will require a slight modification to the reheapify_down() routine to take as an argument the index *i* of a node. So we would call reheapify_down(i) to reheapify down from node *i*, and reheapify_down(0) to reheapify down from the root.

```
buildheap(vector<Item> &v)
heap-size = v.size();
for(i=v.size()/2 - 1; i >= 0; i--)
reheapify_down(i);  // reheapify from node i
```

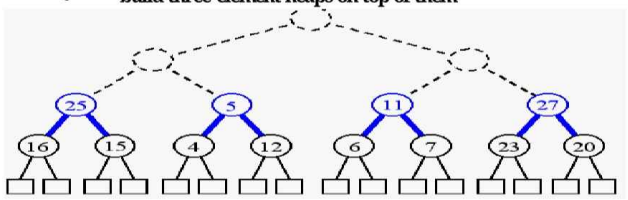
- **Step 1: Build a heap**
- **Step 2: removeMin( )**

Recall: Building a Heap:

- **build (n + 1)/2 trivial one-element heaps**

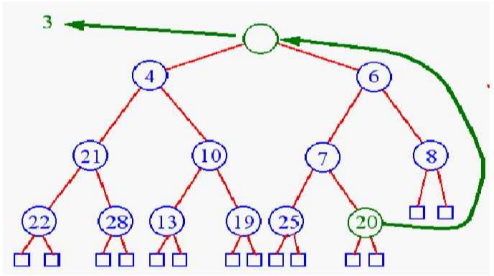


- build three-element heaps on top of them



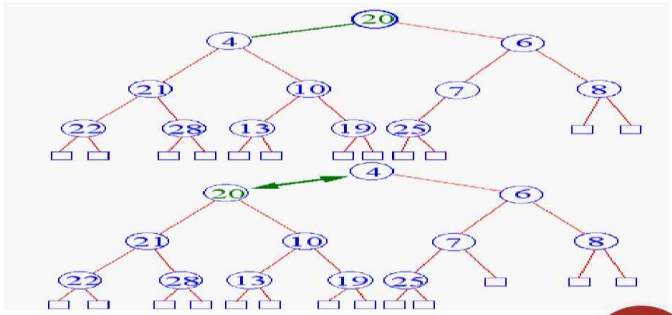
Recall: Heap Removal:

- Remove element from priority queues?
  - Remove Min( )



Recall: Heap Removal

Begin downheap



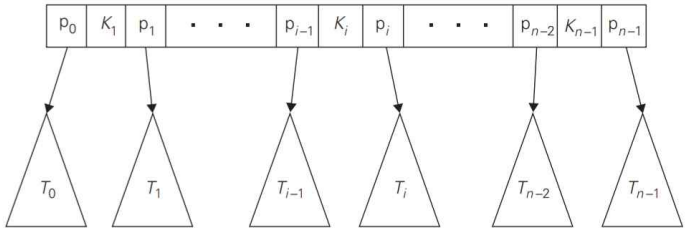


FIGURE 7.7 Parental node of a B-tree.

4 B-Trees

The idea of using extra space to facilitate faster access to a given data set is particularly important if the data set in question contains a very large number of records that need to be stored on a disk. A principal device in organizing such data sets is an ***index***, which provides some information about the location of records with indicated key values. For data sets of structured records (as opposed to "unstructured" data such as text, images, sound, and video), the most important index organization is the ***B-tree***, introduced by R. Bayer and E. McGreight [Bay72]. It extends the idea of the 2-3 tree (see Section 6.3) by permitting more than a single key in the same node of a search tree.

In the B-tree version we consider here, all data records (or record keys) are stored at the leaves, in increasing order of the keys. The parental nodes are used for indexing. Specifically, each parental node contains $n - 1$ ordered keys $K_1 < \cdots < K_{n-1}$ assumed, for the sake of simplicity, to be distinct. The keys are interposed with n pointers to the node's children so that all the keys in subtree T_0 are smaller than K_1, all the keys in subtree T_1 are greater than or equal to K_1 and smaller than K_2 with K_1 being equal to the smallest key in T_1, and so on, through the last subtree T_{n-1} whose keys are greater than or equal to K_{n-1} with K_{n-1} being equal to the smallest key in T_{n-1} (see Figure 7.7).[4]

In addition, a B-tree of order $m \geq 2$ must satisfy the following structural properties:

- The root is either a leaf or has between 2 and m children.
- Each node, except for the root and the leaves, has between $\lceil m/2 \rceil$ and m children (and hence between $\lceil m/2 \rceil - 1$ and $m - 1$ keys).
- The tree is (perfectly) balanced, i.e., all its leaves are at the same level.

4. The node depicted in Figure 7.7 is called the ***n-node***. Thus, all the nodes in a classic binary search tree are 2-nodes; a 2-3 tree introduced in Section 6.3 comprises 2-nodes and 3-nodes.

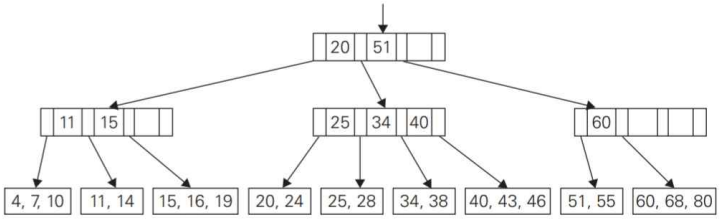


FIGURE 7.8 Example of a B-tree of order 4.

An example of a B-tree of order 4 is given in Figure 7.8.

Searching in a B-tree is very similar to searching in the binary search tree, and even more so in the 2-3 tree. Starting with the root, we follow a chain of pointers to the leaf that may contain the search key. Then we search for the search key among the keys of that leaf. Note that since keys are stored in sorted order, at both parental nodes and leaves, we can use binary search if the number of keys at a node is large enough to make it worthwhile.

It is not the number of key comparisons, however, that we should be concerned about in a typical application of this data structure. When used for storing a large data file on a disk, the nodes of a B-tree normally correspond to the disk pages. Since the time needed to access a disk page is typically several orders of magnitude larger than the time needed to compare keys in the fast computer memory, it is the number of disk accesses that becomes the principal indicator of the efficiency of this and similar data structures.

How many nodes of a B-tree do we need to access during a search for a record with a given key value? This number is, obviously, equal to the height of the tree plus 1. To estimate the height, let us find the smallest number of keys a B-tree of order m and positive height h can have. The root of the tree will contain at least one key. Level 1 will have at least two nodes with at least $\lceil m/2 \rceil - 1$ keys in each of them, for the total minimum number of keys $2(\lceil m/2 \rceil - 1)$. Level 2 will have at least $2\lceil m/2 \rceil$ nodes (the children of the nodes on level 1) with at least $\lceil m/2 \rceil - 1$ in each of them, for the total minimum number of keys $2\lceil m/2 \rceil(\lceil m/2 \rceil - 1)$. In general, the nodes of level i, $1 \le i \le h - 1$, will contain at least $2\lceil m/2 \rceil^{i-1}(\lceil m/2 \rceil - 1)$ keys. Finally, level h, the leaf level, will have at least $2\lceil m/2 \rceil^{h-1}$ nodes with at least one key in each. Thus, for any B-tree of order m with n nodes and height $h > 0$, we have the following inequality:

$$n \ge 1 + \sum_{i=1}^{h-1} 2\lceil m/2 \rceil^{i-1}(\lceil m/2 \rceil - 1) + 2\lceil m/2 \rceil^{h-1}.$$

After a series of standard simplifications (see Problem 2 in this section's exercises), this inequality reduces to

$$n \geq 4\lceil m/2 \rceil^{h-1} - 1,$$

which, in turn, yields the following upper bound on the height h of the B-tree of order m with n nodes:

$$h \leq \lfloor \log_{\lceil m/2 \rceil} \frac{n+1}{4} \rfloor + 1. \quad \textbf{(7.7)}$$

Inequality (7.7) immediately implies that searching in a B-tree is a $O(\log n)$ operation. But it is important to ascertain here not just the efficiency class but the actual number of disk accesses implied by this formula. The following table contains the values of the right-hand-side estimates for a file of 100 million records and a few typical values of the tree's order m:

order m	50	100	250
h's upper bound	6	5	4

Keep in mind that the table's entries are upper estimates for the number of disk accesses. In actual applications, this number rarely exceeds 3, with the B-tree's root and sometimes first-level nodes stored in the fast memory to minimize the number of disk accesses.

The operations of insertion and deletion are less straightforward than searching, but both can also be done in $O(\log n)$ time. Here we outline an insertion algorithm only; a deletion algorithm can be found in the references (e.g., [Aho83], [Cor09]).

The most straightforward algorithm for inserting a new record into a B-tree is quite similar to the algorithm for insertion into a 2-3 tree outlined in Section 6.3. First, we apply the search procedure to the new record's key K to find the appropriate leaf for the new record. If there is room for the record in that leaf, we place it there (in an appropriate position so that the keys remain sorted) and we are done. If there is no room for the record, the leaf is split in half by sending the second half of the records to a new node. After that, the smallest key K' in the new node and the pointer to it are inserted into the old leaf's parent (immediately after the key and pointer to the old leaf). This recursive procedure may percolate up to the tree's root. If the root is already full too, a new root is created with the two halves of the old root's keys split between two children of the new root. As an example, Figure 7.9 shows the result of inserting 65 into the B-tree in Figure 7.8 under the restriction that the leaves cannot contain more than three items.

You should be aware that there are other algorithms for implementing insertions into a B-tree. For example, to avoid the possibility of recursive node splits, we can split full nodes encountered in searching for an appropriate leaf for the new record. Another possibility is to avoid some node splits by moving a key to the node's sibling. For example, inserting 65 into the B-tree in Figure 7.8 can be done by moving 60, the smallest key of the full leaf, to its sibling with keys 51 and 55, and replacing the key value of their parent by 65, the new smallest value in

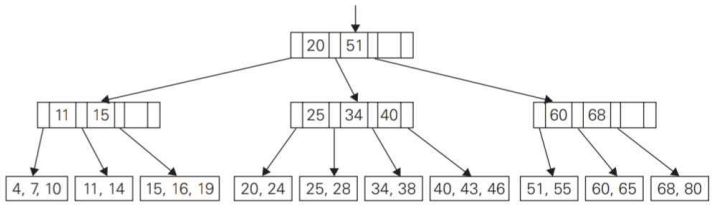


FIGURE 7.9 B-tree obtained after inserting 65 into the B-tree in Figure 7.8.

the second child. This modification tends to save some space at the expense of a slightly more complicated algorithm.

A B-tree does not have to be always associated with the indexing of a large file, and it can be considered as one of several search tree varieties. As with other types of search trees—such as binary search trees, AVL trees, and 2-3 trees—a B-tree can be constructed by successive insertions of data records into the initially empty tree. (The empty tree is considered to be a B-tree, too.) When all keys reside in the leaves and the upper levels are organized as a B-tree comprising an index, the entire structure is usually called, in fact, a ***B*$^+$*-tree***.

Exercises 7.4

1. Give examples of using an index in real-life applications that do not involve computers.
2. **a.** Prove the equality

$$1 + \sum_{i=1}^{h-1} 2\lceil m/2 \rceil^{i-1}(\lceil m/2 \rceil - 1) + 2\lceil m/2 \rceil^{h-1} = 4\lceil m/2 \rceil^{h-1} - 1,$$

   which was used in the derivation of upper bound (7.7) for the height of a B-tree.

   b. Complete the derivation of inequality (7.7).
3. Find the minimum order of the B-tree that guarantees that the number of disk accesses in searching in a file of 100 million records does not exceed 3. Assume that the root's page is stored in main memory.
4. Draw the B-tree obtained after inserting 30 and then 31 in the B-tree in Figure 7.8. Assume that a leaf cannot contain more than three items.
5. Outline an algorithm for finding the largest key in a B-tree.
6. **a.** A ***top-down 2-3-4 tree*** is a B-tree of order 4 with the following modification of the *insert* operation: Whenever a search for a leaf for a new key