Sairam INSTITUTIONS

Sri SAI RAM ENGINEERING COLLEGE
INSTITUTE OF TECHNOLOGY
West Tambaram, Chennai - 44

YEAR	SEM
II	III

CS8391

DATA STRUCTURES (COMMON TO CSE & IT)

UNIT No. 3

NON LINEAR DATA STRUCTURES - TREES

QUESTION BANK

Version: 1.XX

4 QUALITY EDUCATION

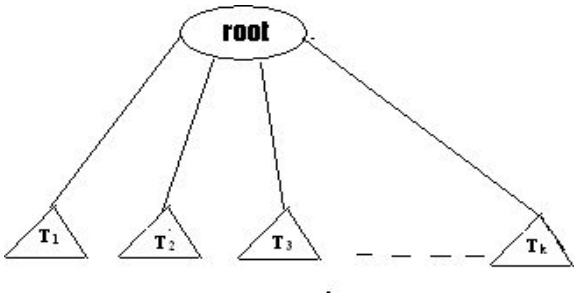
NBA NATIONAL BOARD OF ACCREDITATION | NAAC (A+) | nirf

Unit III

Part A

1. Define Tree.

A Tree is a collection of one or more nodes with a distinct node called the root , while remaining nodes are partitioned as T1 ,T2, ..,Tk , K≥ 0 each of which are sub trees, the edges of T1,T2,…,Tk are connected the root.



2. Give some applications of Trees.

- Implementing the file system of several operating systems.
- Evaluation of arithmetic expression.
- Set representation.
- Gaming/Decision making problems.

3. Define node, degree, siblings, depth/height, level.

Node: A node is an item of information with branches to other items.

Degree: The number of subtrees of a node is called is degree.

Siblings: The children of the same parent is said to be siblings.

Level: The level of a node is defined recursively by assuming the level of the root to be one and if a node is at level l, then its children at level l+1.

Depth/Height: The depth/height of a tree is defined to be the level of a node which is maximum.

4.Define a path in a tree.

A path in a tree is a sequence of distinct nodes in which successive nodes are connected by edges in the tree.

5.Define terminal nodes in a tree.

A node which has no children is called a terminal node.It is also referred as a leaf node.

These nodes have a degree as zero.

6. Define nonterminal nodes in a tree

All intermediate nodes that traverse the given tree from its root node to the terminal nodes are referred as terminal nodes.

7.Define a Binary Tree.

A Binary Tree is a tree, which has nodes either empty or not more than two child nodes, each of which may be a leaf node.

8.Define a full binary tree.

A full binary tree, is a tree in which all the leaves are on the same level and every non- leaf node has exactly two children.

9.Define a complete binary tree.

A complete binary tree is a tree in which every non-leaf node has exactly two children not necessarily to be on the same level.

10.Define a right-skewed binary tree.

A right-skewed binary tree is a tree, which has only right child nodes.

11.State the properties of a Binary Tree.

- Maximum No. of nodes on level n of a binary tree is 2^(n-1),where n>=1.
- Maximum No. of nodes in a Binary tree of height is 2^(n-1),where n>=1.
- For any non-empty tree, nl=nd+1 where nl is the number of leaf nodes and nd is the no. of nodes of degree 2.

12.What are the different ways of representing a Binary Tree?

- Linear Representation using Arrays.
- Linked Representation using Pointers.

13.State the merits of linear representation of binary trees.

- Store methods is easy and can be easily implemented in arrays.
- When the location of the parent/child node is known, other one can be determined easily.
- It requires static memory allocation so it is easily implemented in all programming languages.

14.State the DISADVANTAGES of linear representation of binary trees.

- Insertions and deletions are tougher.
- Processing consumes excess of time.
- Slow data movements up and down the array.

15.Define Traversal.

Traversal is an operation which can be performed on a binary tree is visiting

all the nodes exactly once.

Inorder: traversing the LST, visiting the root and finally traversing the RST.
Preorder: visiting root, traversing LST and finally traversing RST.
Post- order: traversing LST, then RST and finally visiting root.

16.What are the tasks performed while traversing a binary tree?

- Visiting a node
- Traverse the left structure
- Traverse the right structure.

17.What are the tasks performed during preorder traversal?

- Process the root node
- Traverse the left subtree
- Traverse the right subtree. Ex : +AB

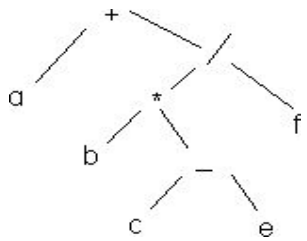
18.What are the tasks performed during inorder traversal?

- Traverse the left subtree
- Process the root node
- Traverse the right subtree. Ex : A+B

19.What are the tasks performed during postorder traversal?

- Traverse the left subtree
- Traverse the right subtree.
- Process the root node. Ex : AB+

20.Give the pre & postfix form of the expression (a + ((b*(c-e))/f).



21.Define a Binary Search Tree.

A Binary Search Tree is a special binary tree,which is either empty or if it is

empty it should satisfy the conditions given below:

- Every node has a value and no two nodes should have the same value(Values should be distinct).
- The value in any left subtree is less than the value of its parent node.
- The value in any right subtree is greater than the value of its parent node.

23.What do u mean by General trees?

General Tree is a tree with nodes having any number of children.

24.Define Forest.

A forest is a collection on N(N>0) disjoint tree or group of trees are called forest. If the root is removed from the tree that tree becomes a forest.

25.Define balanced search tree.

Balanced search tree have the structure of binary tree and obey binary search tree properties with that it always maintains the height as O(log n) by means of a special kind of rotations. Eg. AVL, Splay, B-tree.

26.Define AVL tree.

An empty tree is height balanced. If T is a non-empty binary tree with T_L and T_R as Its left and right subtrees, then T is height balanced if

1. T_L and T_R

are height balanced.

2. $| h_L - h_R | \leq 1$.

Where hl and hr are the height of T_L and T_R respectively.

27.What are the drawbacks of AVL trees?

The drawbacks of AVL trees are

- ❖ Frequent rotations
- ❖ The need to maintain balances for the tree's nodes
- ❖ Overall complexity, especially of the deletion operation.

28.What is a heap?

A heap is a partially ordered data structure, and can be defined as a binary tree assigned to its nodes, one key per node, provided the following two conditions

are met

- ❖ The tree's shape requirement-The binary tree is essentially complete, that is all the leaves are full except possibly the last level, where only some rightmost leaves will be missing.

- ❖ The parental dominance requirement-The key at each node is greater than or equal to the keys of its children

29.What is the main use of heap?

Heaps are especially suitable for implementing priority queues. Priority queue is a set of items with orderable characteristic called an item's priority, with the following operations

- ❖ Finding an item with the highest priority
- ❖ Deleting an item with highest priority
- ❖ Adding a new item to the set

30.Give three properties of heaps?

The properties of heap are

- ❖ There exists exactly one essentially complete binary tree with 'n' nodes. Its height is equal to $\log_2 n$
- ❖ The root of the heap is always the largest element
- ❖ A node of a heap considered with all its descendants is also a heap

31.Give the main property of a heap that is implemented as an array.

A heap can be implemented as an array by recording its elements in the top-down, left-to- right fashion. It is convenient to store the heap's elements in positions 1 through n of such an array. In such a representation

- ❖ The parental node keys will be in the first n/2 positions of the array, while the leaf keys will occupy the last n/2 positions
- ❖ The children of a key in the array's parental position 'i' ($1 \leq i \leq n/2$) will be in positions 2i and 2i+1and correspondingly, the parent of the key in position 'i' ($2 \leq i \leq n$) will be in position i/2.

32.Define B-tree?

A B-tree of order m in an m-way search tree that is either empty or is of height ≥ 1 and

1. The root node has at least 2 children
2. All nodes other than the root node and failure nodes have at least m/2 children.
3. All failure nodes are at same level.

33.Define Priority Queue?

Priority queue is a specialized data structure in which the elements are organized according to the priorities of some key value.

34.Define Binary Heap?

A Binary Heap is a complete binary tree in which the key value of any node must be lesser than its children is called min heap. If the key value of any node is greater than its children is called max heap. Binary heap is also called as partially ordered

tree.

35.Explain array implementation of Binary Heap.

For any element in the array position 'i', the left child is at the position '2i', the right child is at the position '2i+1' and parent is at the position 'i/2'.

36.Define Max-heap.

Maxheap: A heap in which the parent has a larger key that the child's key values then it is called Maxheap.

37.Explain AVL rotation.

Manipulation of tree pointers is centered at the pivot node to bring the tree back into height balance. The visual effect of this pointer manipulation so to rotate the sub tree whose root is the pivot node. This operation is referred as AVL rotation.

38.What are the different type of Rotation in AVL Tree?

Two types of rotation are
1.single rotation
2.double rotation.

PART-B

1.With suitable examples, explain binary tree traversal algorithms.

Binary tree traversal is a method for visiting all the nodes in the tree exactly once. There are three types of tree traversal techniques, namely

1. Inorder Traversal
2. Preorder Traversal
3. Postorder Traversal

1. Inorder Traversal

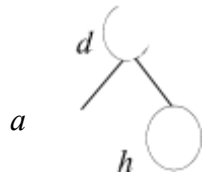
The inorder traversal of a binary tree is performed as,

(i) Traverse the left subtree in order
(ii) Visit the root
(iii) Traverse the right subtree in order

```
voidinorder (Tree T)
{
if(T!=NULL)
{
        inorder(
        T->left)
        ;
```

```
            printf("
            %d",T-
            >data);
            inorder(
            T->righ
            t);
        }
    }
```

Example:



Output: a d h

1. Preorder Traversal:

The preorder traversal of a binary tree is performed as,

(i) Visit the root
(ii) Traverse the left subtree in order
(iii) Traverse the right subtree in order

```
void preorder(Tree T)
{
    if(T!=NULL)
    {
            printf("
            %d",T-
            >data);
            preorde
            r(T->le
            ft);
            preorde
            r(T->ri
            ght);
    }
}
```

Output: d a h

2. Postorder Traversal

The Post order traversal is performed as :

(i) Traverse the left subtree in order

(ii) Traverse the right subtree in order
(iii)Visit the root void

```
postorder(Tree T)
{
    if(T!=NULL)
    {postorder(T->left);
    postorder(T->right);
    printf("%d", T->data);
    }
}
```

Output: a h d

2.What are expression tree? Represent the following expression using tree: (a–b) / ((c*d)+e). Comment on the result that you get when this tree is traversed in
Preorder, Inorder
and Postorder. <u>Ans:</u>
Expression Tree:

Expression tree is a tree in which the leaf nodes are operands and the interior nodes are operators.

Represent the expression into Tree:

The expression (a–b) / ((c*d)+e) is converted to postfix form a b – c d * e + /. This postfix expression is converted into the following tree.

Inorder Traversal

- Traverse the left subtree in order Visit the root
- Traverse the right subtree in order

```
voidinorder (Tree T)
{
    if(T!=NULL)
    {
        inorder(T->left); printf("%d",T->data);
        inorder(T->right);
    }
}
```

Output:
a-b/c*d+e

Preorder Traversal:

- Visit the root
- Traverse the left subtree in order
- Traverse the right

subtree in order void
preorder(Tree T)

```
{
    if(T!=NULL)
    {
            printf("%d", T->data);
            preorder(T->left);
            preorder(T->right);
    }

}
```

Output: /-ab+*cde

Postorder Traversal

- Traverse the left subtree in order
- Traverse the right subtree in order
- V

isit the
root void
postorder
(Tree T)

```
{
    if(T!=NULL)
    {
            postorder(T->left);
            postorder(T->right);
            printf("%d",T->data);
    }
}
```

Output: /-ab+*cde

3.Write an algorithm to insert an item into a binary search tree. Insertion of Binary Search Tree:

To insert the element X into the tree, the steps are,

1. Check with the root node T.
2. If it is less than the root,
   Traverse the left subtree recursively until it reaches the T-> left equals to NULL, then X is placed in T->left.
3. If X is greater than the root
   Traverse the right subtree recursively until it reaches the T->right equals to NULL. Then X is place in T->right.

Routine for insertion

```
searchtree insert(int x, searchtree t)
{
if (t==NULL)
   {
        t=malloc (sizeof (structtreenode));
        if (t != NULL)
        {
        t->element = x; t->left = NULL;
        t->right = NULL;
        }
   }
 else if (x < t->element)
        t->left =
        insert(x,
        t->left); else
        if (x >
        t->element)
        t->right   =
 insert(x,t->right); return
 t;
 }
```

4.Explain the possible AVL rotation with algorithm and example .

An AVL tree is a special type of binary tree that is always "partially" balanced. The criteria that is used to determine the "level" of "balanced-ness" is the difference between the heights of subtrees of a root in the tree. The "height" of tree is the "number of levels" in the tree. Or to be more formal, the height of a tree is defined as follows:

The height of a tree with no elements is 0 The height of a tree with 1 element is 1

The height of a tree with > 1 element is equal to 1 + the height of its tallest subtree.

An AVL tree is a binary tree in which the difference between the height of the right

and left subtrees (or the root node) is never more than one.

The idea behind maintaining the "AVL-ness" of an AVL tree is that whenever we insert or delete an item, if we have "violated" the "AVL-ness" of the tree in anyway, we must then restore it by performing a set of manipulations (called "rotations") on the tree. These rotations come in two flavors: single rotations and double rotations (and each flavor has its corresponding "left" and "right" versions).

An example of a single rotation is as follows: Suppose I have a tree that looks like this:

```
  c
 /
b
```

Now I insert the item "a" and get the resulting binary tree:

```
   c
  /
 b
/
a
```

Now, this resulting tree violates the "AVL criteria", the left subtree has a height of 2 but the right subtree has a height of 0 so the difference in the two heights is "2" (which is greater than 1). SO what we do is perform a "single rotation" (or RR for a single right rotation, or LL for a single left rotation) on the tree (by rotating the "c" element down clockwise to the right) to transform it into the following tree:

```
  b
 / \
a c
```

This tree is now balanced.

An example of a "double rotation" (or RL for a double right rotation, or LR for a double left rotation) is the following: Suppose I have a tree that looks like this:

```
a
 \
  c
```

Now I insert the item "b" and get the resulting binary tree:

```
a
 \
  c
 /
b
```

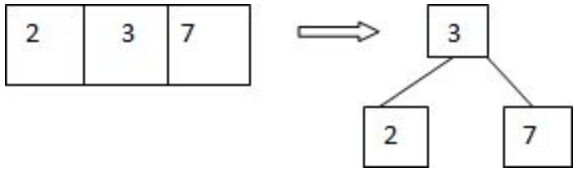
This resulting tree also violates the "AVL criteria" so we fix it by first rotating "c" down to the right (so we get "a-b-c"), and then rotating "a" down to the left so that the tree is transformed into this:

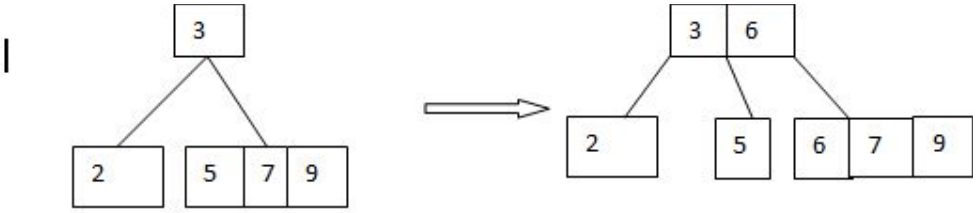
```
  b
 / \
```

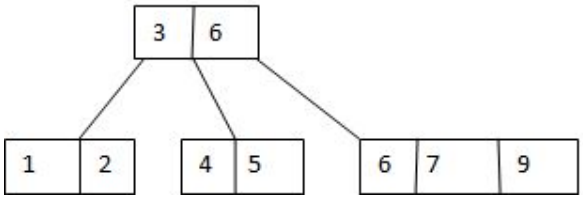
a c

In order to detect when a "violation" of the AVL criteria occurs we need to have each node keep track of the difference in height between its right and left subtrees. We call this "difference" the "balance" factor and define it to be the height of the right subtree minus the height of the left subtree of a tree. So as long as the "balance" factor of each node is never >1 or <-1 we have an AVL tree. As soon as the balance factor of a node becomes 2 (or -2) we need to perform one or more rotations to ensure that the resultant tree satisfies the AVL criteria.

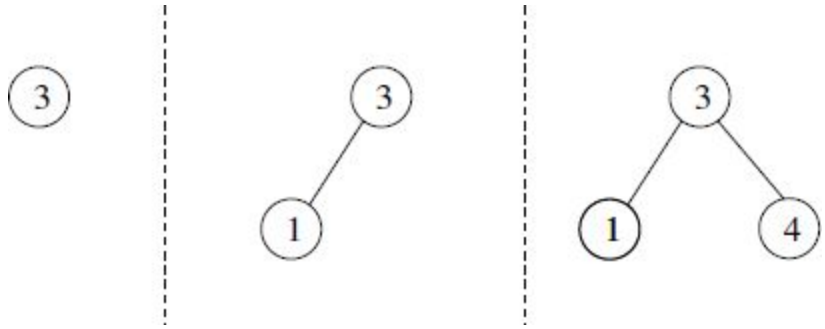
5.Construct a B-Tree with order m=3 for the key values 2,3,7,9,5,6,4,8,1 and delete the values 4 and 6.Show tree in performing all operations.

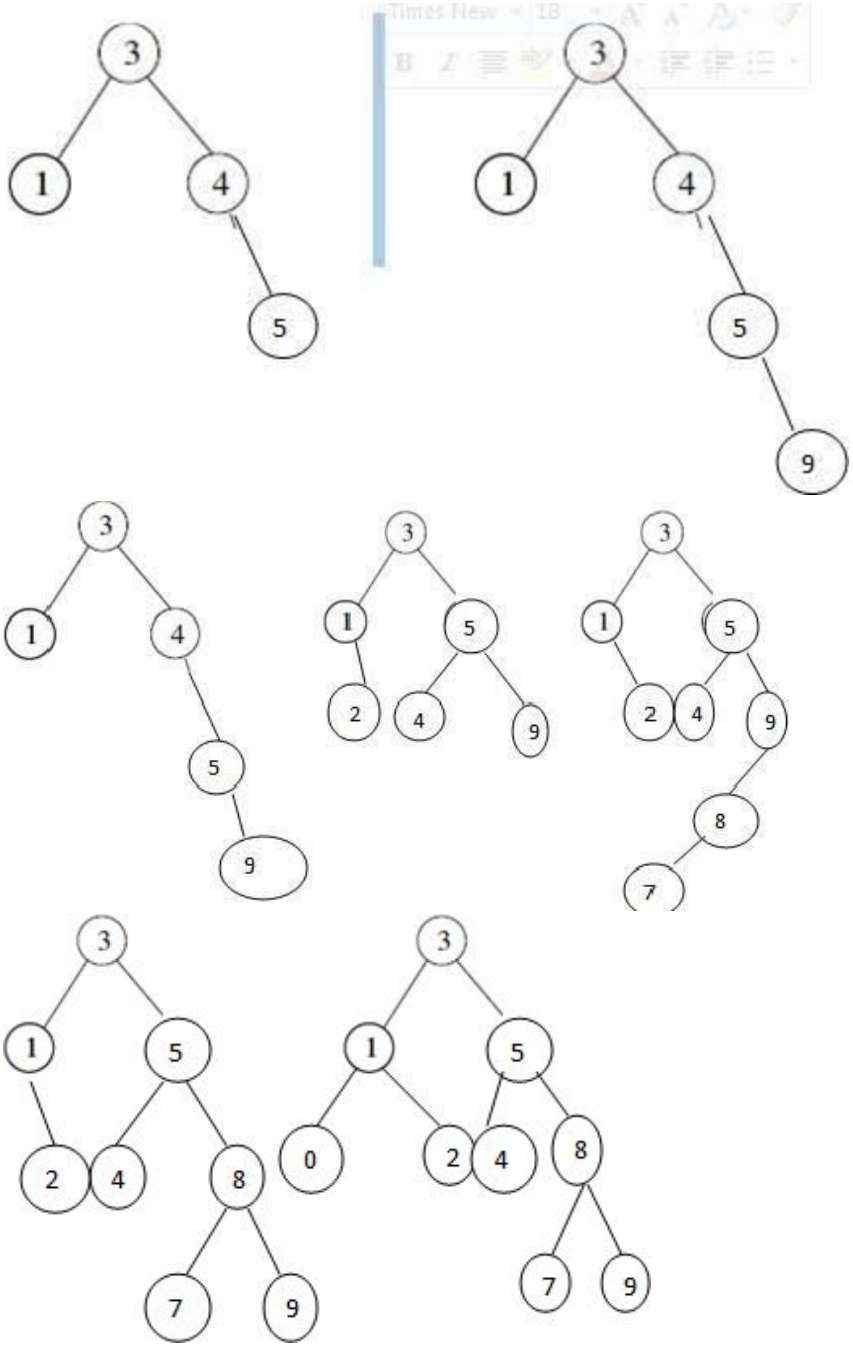






6.Construct an AVL tree with the values 3,1,4,5,9,2,8,7,0 into an initially empty tree.Write the code for inserting into an AVL tree.

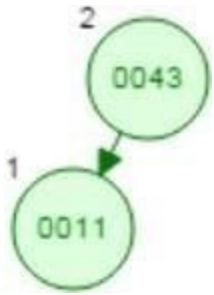




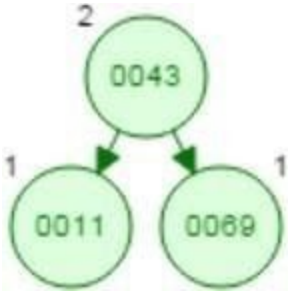
7.**Show the result of inserting 43,11,69,72 and 30 into an initially empty AVL tree. Show the results of deleting the nodes 11 and 72 one after the other of the constructed tree.**

Inserting 43

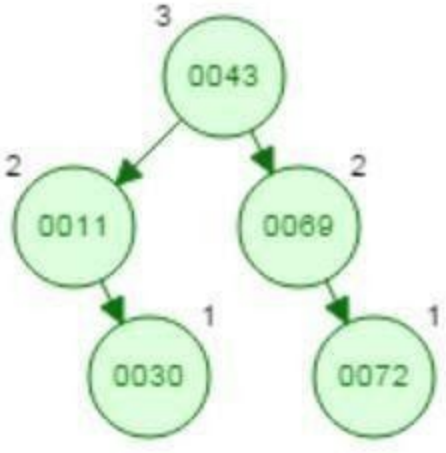
Inserting 11



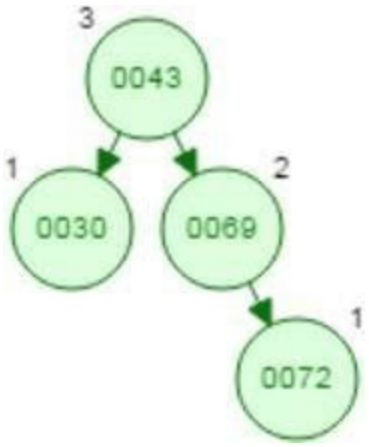
Inserting 69



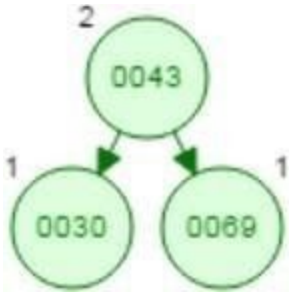
Similarly Inserting 72 and 30



Deleting node 11



Deleting node 72



8. Explain In Detail About AVL TREES.

AVL - Good but not Perfect Balance

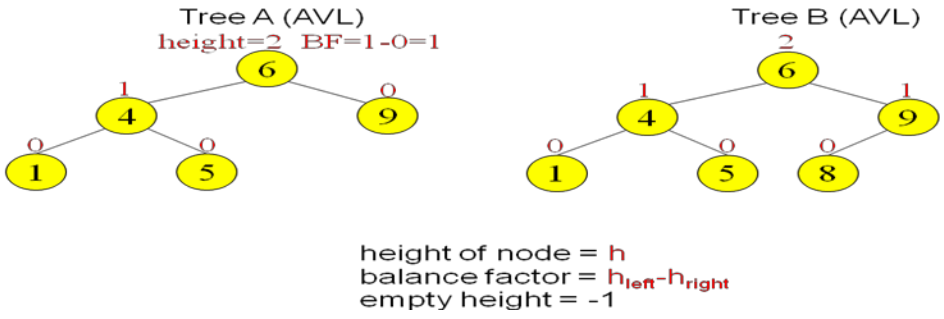
- AVL trees are height-balanced binary search trees
- Balance factor of a node
  - › height(left subtree) - height(right subtree)

An AVL tree has balance factor calculated at every node

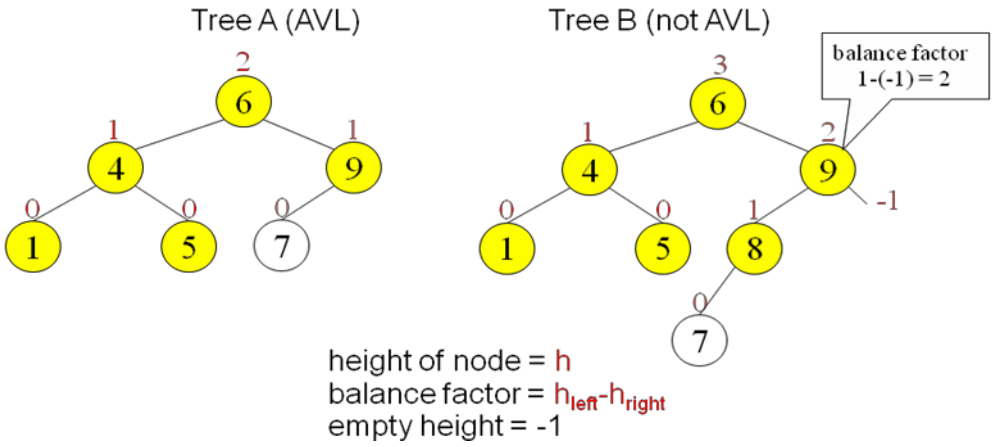
- › For every node, heights of left and right subtree can differ by no more than 1
- › Store current

heights in each node Node

Heights



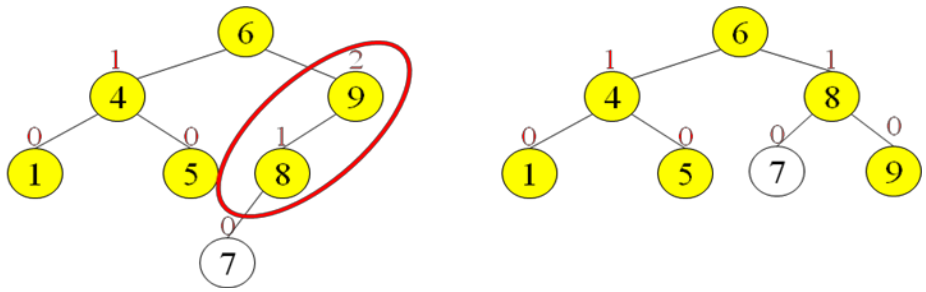
Node Heights after Insert 7



Insert and Rotation in AVL Trees

- Insert operation may cause balance factor to become 2 or –2 for some node
  - › only nodes on the path from insertion point to root node have possibly changed in height
  - › So after the Insert, go back up to the root node by node, updating heights
  - › If a new balance factor (the difference h_{left}-h_{right}) is 2 or –2, adjust tree by *rotation*

around the node Single Rotation in an AVL Tree



Insertions in AVL Trees

Let the node that needs rebalancing be α.

There are 4 cases:

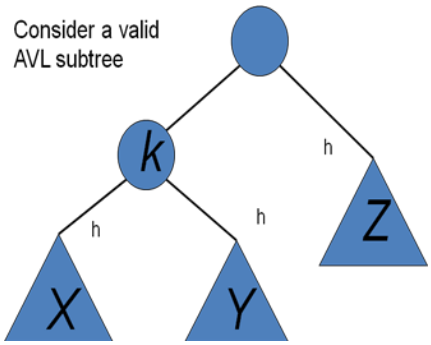
Outside Cases (require single rotation) :

1. Insertion into left subtree of left child of α.
2. Insertion into right subtree of right child of α. Inside Cases

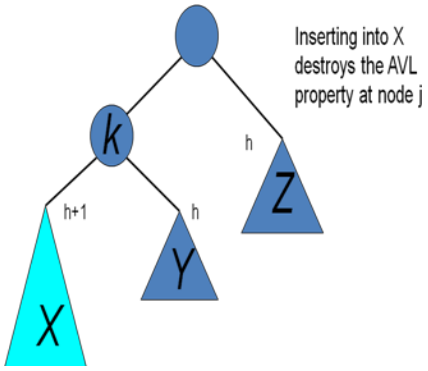
(require double rotation) :

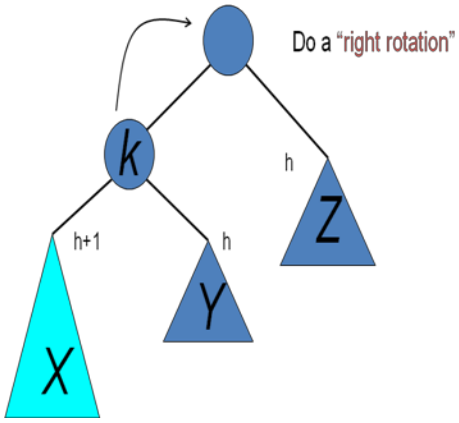
3. Insertion into right subtree of left child of α.
4. Insertion into left subtree of right child of α.

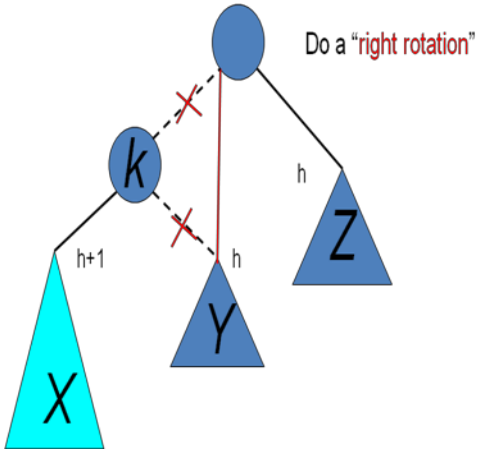
The rebalancing is performed through four separate rotation algorithms. AVL Insertion: Outside Case

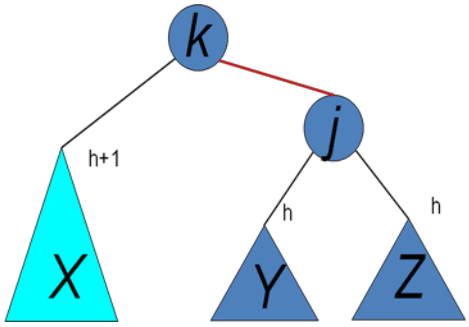


AVL Insertion:









8. Explain about B trees?

A B-tree is a tree data structure that keeps data sorted and allows searches, insertions, and deletions in logarithmic amortized time. Unlike self-balancing binary search trees, it is optimized for systems that read and write large blocks of data. It is most commonly used in database and file systems.

Important properties of a B-tree:

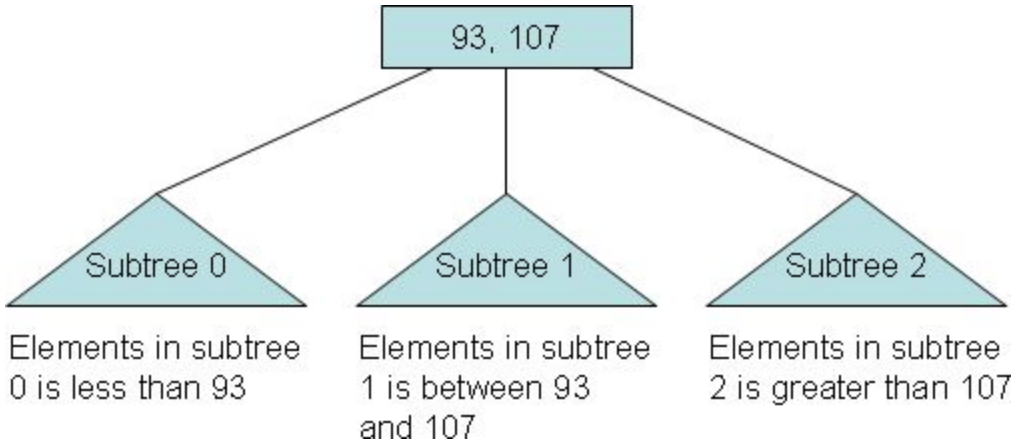
B-tree nodes have many more than two children.

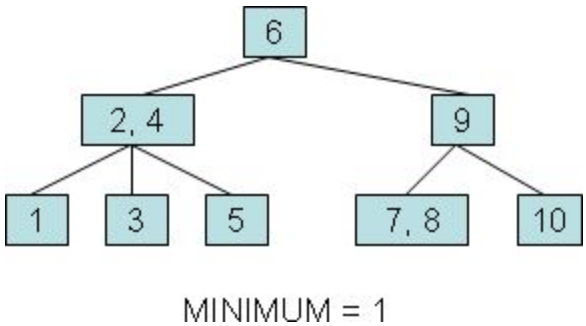
- A B-tree node may contain more than just a single element.

The set formulation of the B-tree rules: Every B-tree depends on a positive constant integer called MINIMUM, which is used to determine how many elements are held in a single node.

- **Rule 1**: The root can have as few as one element (or even no elements if it also has no children); every other node has at least MINIMUM elements.
- **Rule 2**: The maximum number of elements in a node is twice the value of MINIMUM.
- **Rule 3**: The elements of each B-tree node are stored in a partially filled array, sorted from the smallest element (at index 0) to the largest element (at the final used position of the array).
- **Rule 4**: The number of subtrees below a nonleaf node is always one more than the number of elements in the node.
  - Subtree 0, subtree 1, ...
- **Rule 5**: For any nonleaf node:
  1. An element at index *i* is greater than all the elements in subtree number *i* of the node, and
  2. An element at index *i* is less than all the elements in subtree number *i* + 1 of the node.

Rule 6: Every leaf in a B-tree has the same depth. Thus it ensures that a B-tree avoids the problem of a unbalanced tree





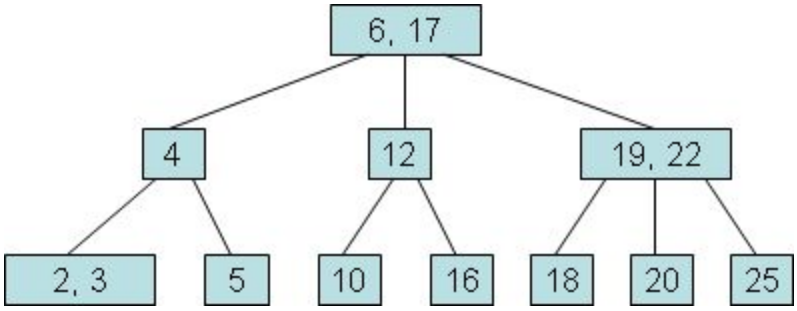
Remember that "**Every child of a node is also the root of a smaller B-tree**".

Searching for a Target in a Set

The pseudo code:

1. Make a local variable, *i*, equal to the first index such that data[*i*] >= target. If there is no such in then set *i* equal to dataCount, indicating that none of the elements is greater than or equal to target.
2. if (we found the target at data[*i*]) return true;
   else if (the root has no children) return false;
   else return subset[i].contains(target);

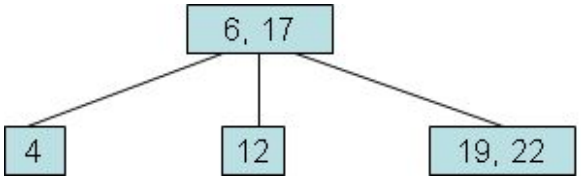
See the following example, try to search for 10.

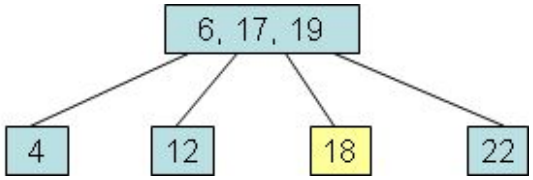


Adding an Element to a B-Tree

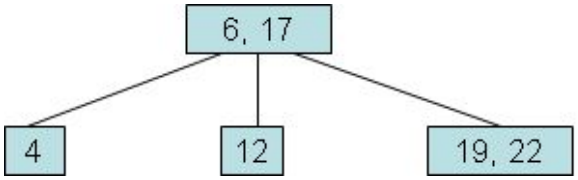
It is easier to add a new element to a B-tree if we relax one of the B-tree rules.

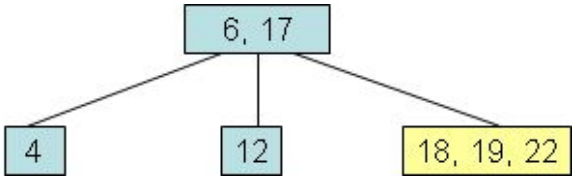
Loose addition allows the root node of the B-tree to have MAXIMUM + 1 elements. For exampl suppose we want to add 18 to the tree:





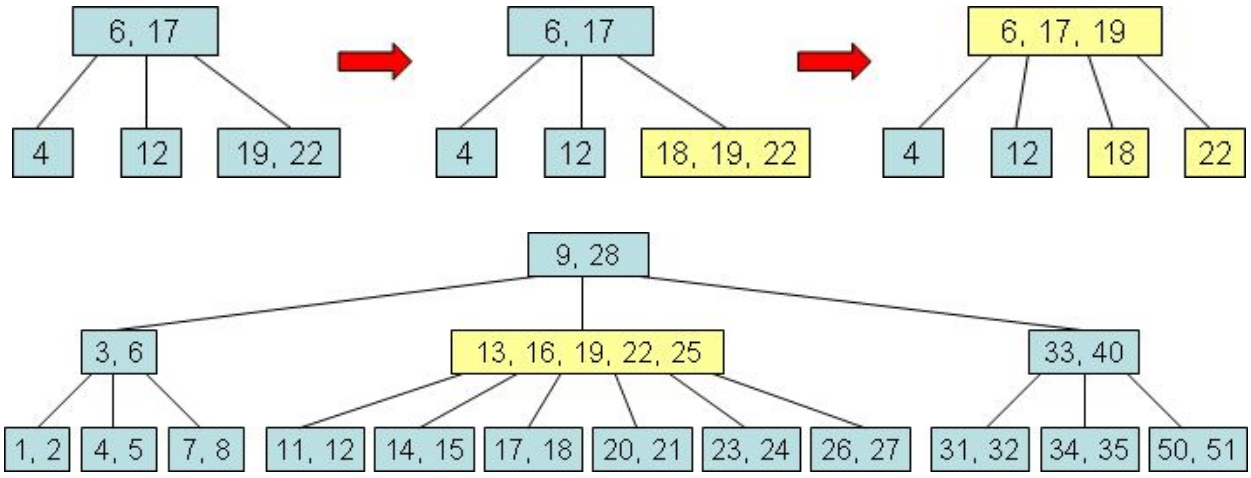
The above result is an illegal B-tree. Our plan is to perform a loose addition first, and then fix the root's problem.





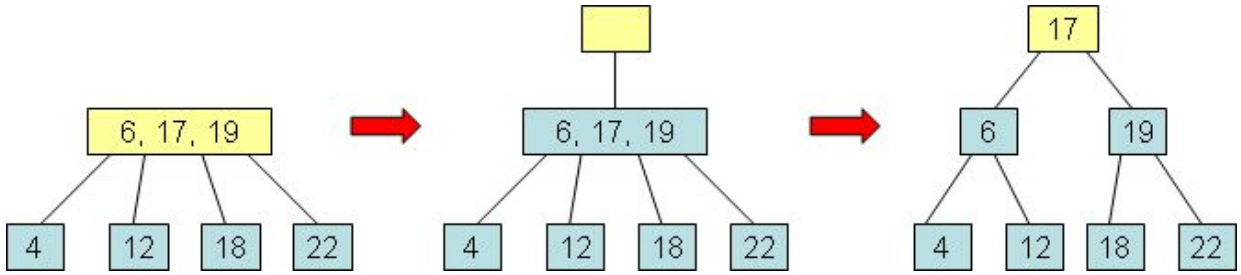
Fixing a Child with an Excess Element:

- To fix a child with MAXIMIM + 1 elements, the child node is split into two nodes that each contain MINIMUM elements. This leaves one extra element, which is passed up to the parent.
- It is always the *middle* element of the split node that moves upward.
- The parent of the split node gains one additional child and one additional element.
- The children of the split node have been equally distributed between the two smaller nodes.



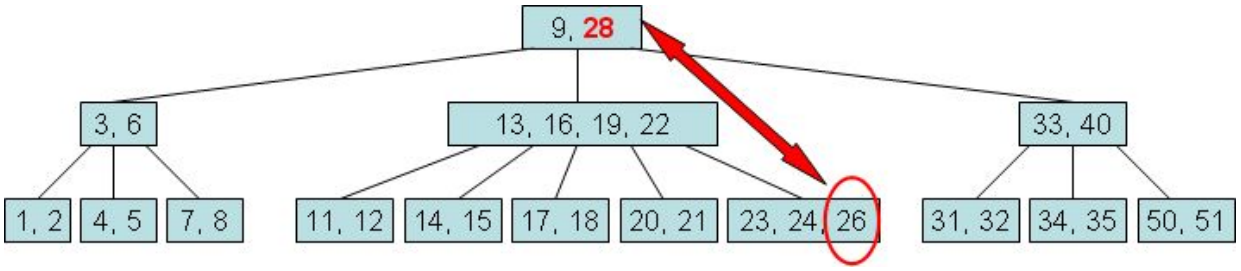
Fixing the Root with an Excess Element:

- Create a new root.
- fixExcess(0).



Removing an Element from a B-Tree

Loose removal rule: *Loose removal allows to leave a root that has one element too few*



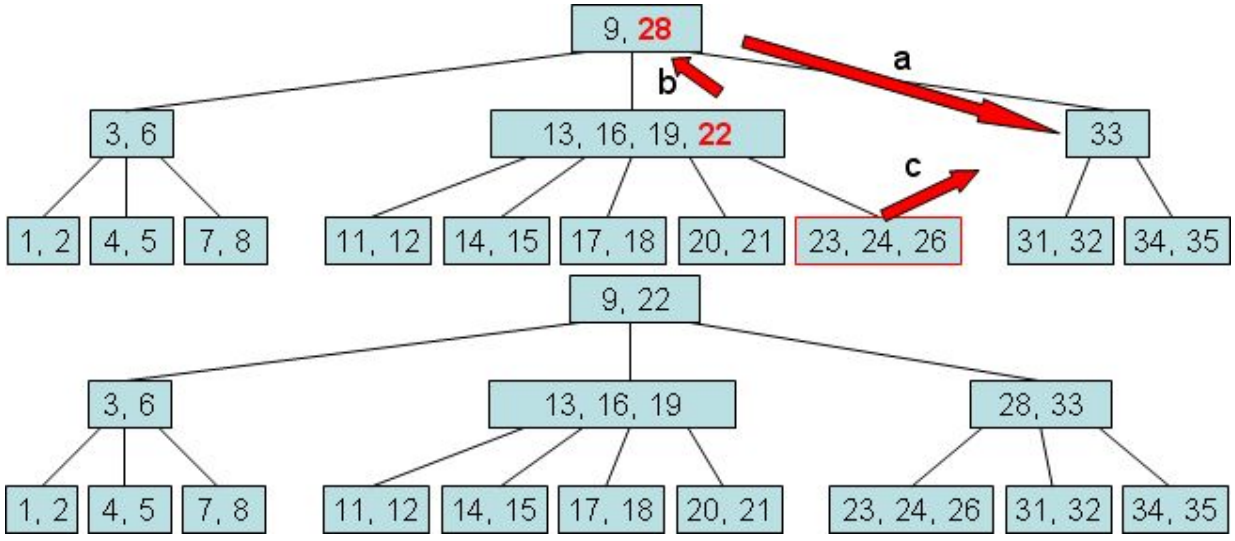
Fixing Shortage in a Child:

When fixShortage(*i*) is activated, we know that subset[*i*] has MINIMUM - 1 elements. There are four cases that we need to consider:

Case 1: Transfer an extra element from subset[*i*-1]. Suppose subset[*i*-1] has more than the MINIMUM number of elements.

Transfer data[*i*-1] down to the front of subset[*i*].data.

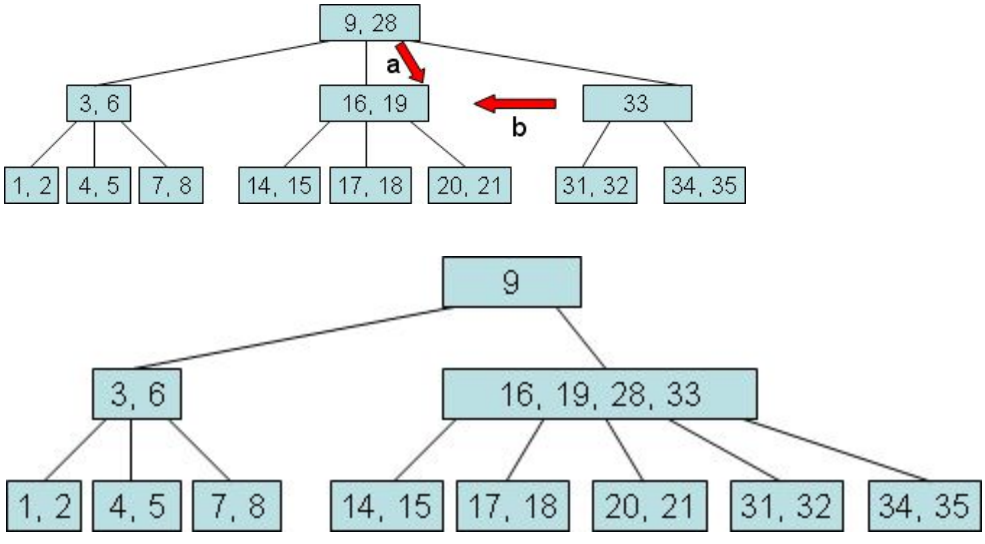
a. Transfer the final element of subset[*i*-1].data up to replace data[*i*-1].
b. If subset[*i*-1] has children, transfer the final child of subset[*i*-1] over to the front of subset[*i*].



Case 2: Transfer an extra element from subset[*i*+1]. Suppose subset[*i*+1] has more than the MINIMUM number of elements.

Case 3: Combine subset[*i*] with subset[*i*-1]. Suppose subset[*i*-1] has only MINIMUM elements.

a. Transfer data[*i*-1] down to the end of subset[*i*-1].data.
b. Transfer all the elements and children from subset[*i*] to the end of subset[*i*-1].
c. Disconnect the node subset[*i*] from the B-tree by shifting subset[*i*+1], subset[*i*+2] and so on leftward.

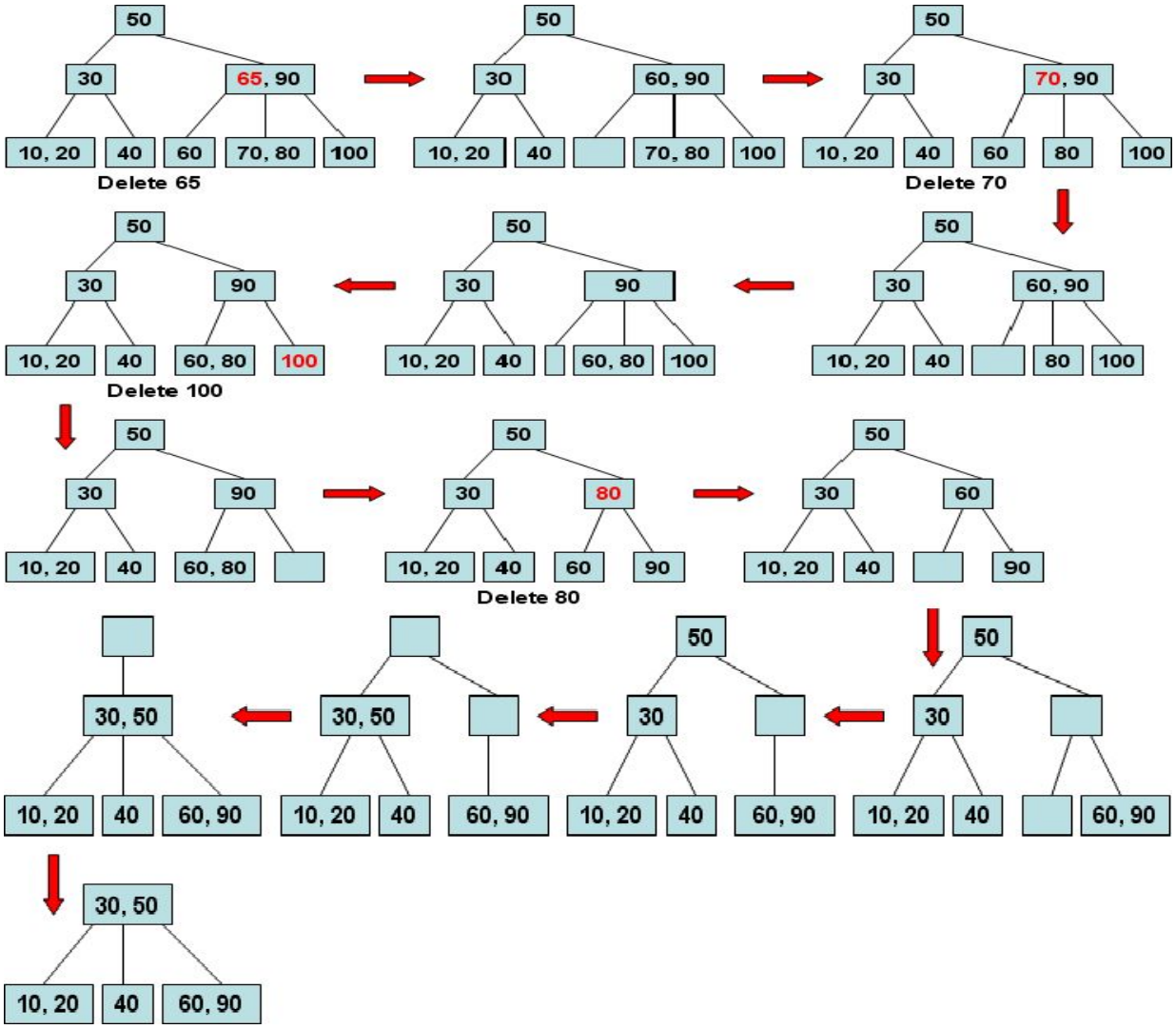


Case 4: Combine subset[*i*] with subset[*i*+1]. Suppose subset[*i*+1] has only MINIMUM elements.

We may need to continue activating fixShortage() until the B-tree rules are satisfied.

Removing the Biggest Element from a B-Tree:

A more concrete example for node deletion:



A 2-3 tree is a type of B-tree where every node with children (internal node) has either two children and one data element (2-nodes) or three children and two data elements (3-node). Leaf nodes have no children and one or two data elements.

9.Explain in detail about Binomial Heap.

A Binomial heap is implemented as a collection of binomial trees (compare with a binary heap, which has a shape of a single binary tree). A **binomial tree** is defined recursively:

- A binomial tree of order 0 is a single node
- A binomial tree of order *k* has a root node whose children are roots of binomial trees of orders *k*−1, *k*−2, ..., 2, 1, 0 (in this order).
- A binomial tree of order *k* has 2^k nodes, height *k*.
- Because of its unique structure, a binomial tree of order *k* can be constructed from two trees of order *k*−1 trivially by attaching one of them as the leftmost child of root of the other one. This feature is central to the *merge* operation of a binomial heap, which is its major advantage over other conventional heaps.
- The name comes from the shape: a binomial tree of order n has $\binom{n}{d}$ nodes at depth d

Structure of a binomial heap

A binomial heap is implemented as a set of binomial trees that satisfy the *binomial heap properties*:

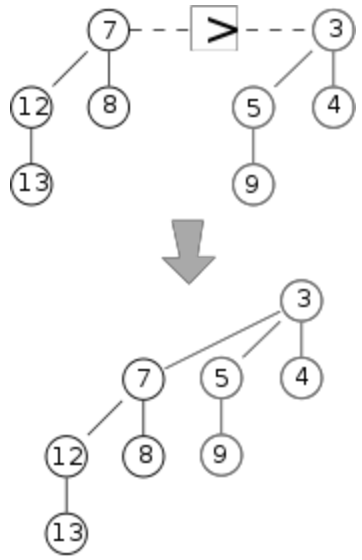
- Each binomial tree in a heap obeys the *minimum-heap property*: the key of a node is greater than or equal to the key of its parent.
- There can only be either *one* or *zero* binomial trees for each order, including zero order.

The **first property** ensures that the root of each binomial tree contains the smallest key in the tree, which applies to the entire heap.
The **second property** implies that a binomial heap with *n* nodes consists of at most log *n* + 1 binomial trees. In fact, the number and orders of these trees are uniquely determined by the number of nodes *n*: each binomial tree corresponds to one digit in the binary representation of number *n*. For example number 13 is 1101 in binary, $2^3 + 2^2 + 2^0$, and thus a binomial heap with 13 nodes will consist of three binomial trees of orders 3, 2, and 0 (see figure below).

Implementation

Because no operation requires random access to the root nodes of the binomial trees, the roots of the binomial trees can be stored in a linked list, ordered by increasing order of the tree.

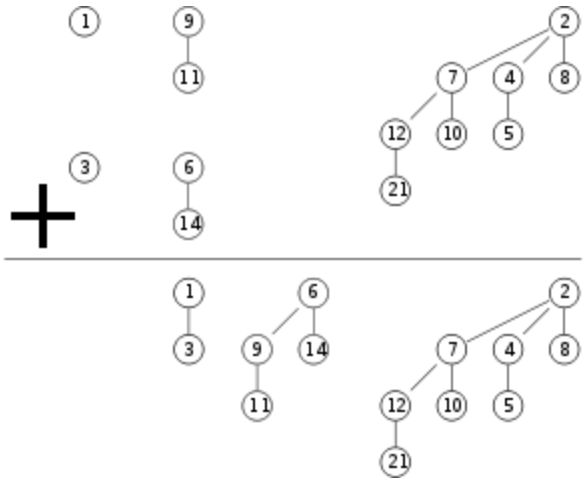
Merge



To merge two binomial trees of the same order, first compare the root key. Since 7>3, the black tree on the left(with root node 7) is attached to the grey tree on the right(with root node as a subtree. The result is a tree of order 3.

As mentioned above, the simplest and most important operation is the merging of two binomial trees of the same order within two binomial heaps. Due to the structure of binomial trees, they can be merged trivially. As their root node is the smallest element within the tree, by comparing the two keys, the smaller of them is the minimum key, and becomes the new root node. Then the other tree becomes a subtree of the combined tree. This operation is basic to the complete merging of two binomial heaps.

```
function mergeTree(p, q)
    if p.root.key <= q.root.key
        return p.addSubTree(q)
  else
     return q.addSubTree(p)
```



This shows the merger of two binomial heaps. This is accomplished by merging two binomial trees of the same order one by one. If the resulting merged tree has the same order as one binomial tree in one of the two heaps, then those two are merged again.

The operation of **merging** two heaps is perhaps the most interesting and can be used as a subroutine in most other operations. The lists of roots of both heaps are traversed simultaneously, similarly as in the merge algorithm.

If only one of the heaps contains a tree of order *j*, this tree is moved to the merged heap. If both heaps contain a tree of order *j*, the two trees are merged to one tree of order *j*+1 so that the minimum-heap property is satisfied. Note that it may later be necessary to merge this tree with some other tree of order *j*+1 present in one of the heaps. In the course of the algorithm, we need to examine at most three trees of any order (two from the two heaps we merge and one composed of two smaller trees).

Because each binomial tree in a binomial heap corresponds to a bit in the binary representation of its size, there is an analogy between the merging of two heaps and the binary addition of the *sizes* of the two heaps, from right-to-left. Whenever a carry occurs during addition, this corresponds to a merging of two binomial trees during the merge.

Each tree has order at most log *n* and therefore the running time is $O(\log n)$.

```
function merge(p, q)
  while not (p.end() and q.end())
    tree = mergeTree(p.currentTree(), q.currentTree())

    if not heap.currentTree().empty()
      tree = mergeTree(tree, heap.currentTree())

    heap.addTree(tree) heap.next();
    p.next(); q.next()
```

Insert

Inserting a new element to a heap can be done by simply creating a new heap containing only this element and then merging it with the original heap. Due to the merge, insert takes O(log *n*) time,however it has an *amortized* time of O(1) (i.e. constant).

Find minimum

To find the **minimum** element of the heap, find the minimum among the roots of the binomial trees. This can again be done easily in $O(\log n)$ time, as there are just $O(\log n)$ trees and hence roots to examine.

By using a pointer to the binomial tree that contains the minimum element, the time for this operation can be reduced to $O(1)$. The pointer must be updated when performing any operation other than Find minimum. This can be done in $O(\log n)$ without raising the running time of any operation.

Delete minimum

To **delete the minimum element** from the heap, first find this element, remove it from its binomial tree, and obtain a list of its subtrees. Then transform this list of subtrees into a separate binomial heap by reordering them from smallest to largest order. Then merge this heap with the original heap. Since each tree has at most log *n* children, creating this new heap is $O(\log n)$. Merging heaps is $O(\log n)$, so the entire delete minimum operation is $O(\log n)$.

```
function deleteMin(heap) min
  = heap.trees().first()
  for each current in heap.trees()
    if current.root < min then min = current
  for each tree in min.subTrees()
```

```
    tmp.addTree(tree)
  heap.removeTree(min)
  merge(heap, tmp)
```

Decrease key

After **decreasing** the key of an element, it may become smaller than the key of its parent, violating the minimum-heap property. If this is the case, exchange the element with its parent, and possibly also with its grandparent, and so on, until the minimum-heap property is no longer violated. Each binomial tree has height at most log *n*, so this takes *O*(log *n*) time.

Delete

To **delete** an element from the heap, decrease its key to negative infinity (that is, some value lower than any element in the heap) and then delete the minimum in the heap.

Performance

All of the following operations work in O(log *n*) time on a binomial heap with *n* elements:

- Insert a new element to the heap
- Find the element with minimum key
- Delete the element with minimum key from the heap
- Decrease key of a given element
- Delete given element from the heap
- Merge two given heaps to one heap

Finding the element with minimum key can also be done in *O*(1) by using an additional pointer to the minimum.

Application

- Discrete event simulation
- Priority queues

10.Explain about Fibonacci Heap

A **Fibonacci heap** is a heap data structure consisting of a collection of trees. It has a better amortized running time than a binomial heap

A Fibonacci heap is a collection of trees satisfying the minimum-heap property, that is, the key of a child is always greater than or equal to the key of the parent. This implies that the minimum key is always at the root of one of the trees. Compared with binomial heaps, the structure of a Fibonacci heap is more flexible. The trees do not have a prescribed shape and in the extreme case the heap can have every element in a separate tree. This flexibility allows some operations to be executed in a "lazy" manner, postponing the work for later operations. For example merging heaps is done simply by

concatenating the two lists of trees, and operation *decrease key* sometimes cuts a node from its parent and forms a new tree.

However at some point some order needs to be introduced to the heap to achieve the desired running time. In particular, degrees of nodes (here degree means the number of children) are kept quite low: every node has degree at most $O(\log n)$ and the size of a subtree rooted in a node of degree k is at least F_{k+2}, where F_k is the kth Fibonacci number. This is achieved by the rule that we can cut at most one child of each non-root node. When a second child is cut, the node itself needs to be cut from its parent and becomes the root of a new tree (see Proof of degree bounds, below). The number of trees is decreased in the operation *delete minimum*, where trees are linked together.

As a result of a relaxed structure, some operations can take a long time while others are done very quickly. For the amortized running time analysis we use the potential method, in that we pretend that very fast operations take a little bit longer than they actually do. This additional time is then later combined and subtracted from the actual running time of slow operations. The amount of time saved for later use is measured at any given moment by a potential function. The potential of a Fibonacci heap is given by

Potential = $t + 2m$

where t is the number of trees in the Fibonacci heap, and m is the number of marked nodes. A node is marked if at least one of its children was cut since this node was made a child of another node (all roots are unmarked). The amortized time for an operation is given by the sum of the actual time and c times the difference in potential, where c is a constant (chosen to match the constant factors in the O notation for the actual time).

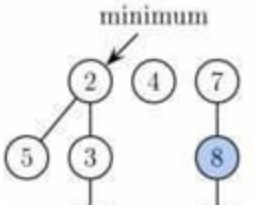
Thus, the root of each tree in a heap has one unit of time stored. This unit of time can be used later to link this tree with another tree at amortized time 0. Also, each marked node has two units of time stored. One can be used to cut the node from its parent. If this happens, the node becomes a root and the second unit of time will remain stored in it as in any other root.

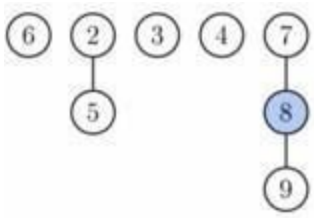
Implementation of operations

To allow fast deletion and concatenation, the roots of all trees are linked using a circular, doubly linked list. The children of each node are also linked using such a list. For each node, we maintain its number of children and whether the node is marked. Moreover we maintain a pointer to the root containing the minimum key.

Operation **find minimum** is now trivial because we keep the pointer to the node containing it. It does not change the potential of the heap, therefore both actual and amortized cost is constant. As mentioned above, **merge** is implemented simply by concatenating the lists of tree roots of the two heaps. This can be done in constant time and the potential does not change, leading again to constant amortized time.

Operation **insert** works by creating a new heap with one element and doing merge. This takes constant time, and the potential increases by one, because the number of trees increases. The amortized cost is thus still constant.





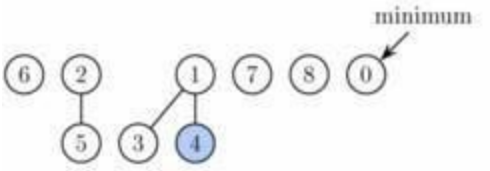
Fibonacci heap from Figure 1 after first phase of extract minimum. Node with key 1 (the minimum) was deleted and its children were added as separate trees.

Operation **extract minimum** (same as *delete minimum*) operates in three phases. First we take the root containing the minimum element and remove it. Its children will become roots of new trees. If the number of children was d, it takes time $O(d)$ to process all new roots and the potential increases by $d-1$. Therefore the amortized running time of this phase is $O(d) = O(\log n)$.

Fibonacci heap from Figure 1 after extract minimum is completed. First, nodes 3 and 6 are linked together. Then the result is linked with tree rooted at node 2. Finally, the new minimum is found.

However to complete the extract minimum operation, we need to update the pointer to the root with minimum key. Unfortunately there may be up to n roots we need to check. In the second phase we therefore decrease the number of roots by successively linking together roots of the same degree. When two roots u and v have the same degree, we make one of them a child of the other so that the one with the smaller key remains the root. Its degree will increase by one. This is repeated until every root has a different degree. To find trees of the same degree efficiently we use an array of length $O(\log n)$ in which we keep a pointer to one root of each degree. When a second root is found of the same degree, the two are linked and the array is updated. The actual running time is $O(\log n + m)$ where m is the number of roots at the beginning of the second phase. At the end we will have at most $O(\log n)$ roots (because each has a different degree). Therefore the difference in the potential function from before this phase to after it is: $O(\log n) - m$, and the amortized running time is then at most $O(\log n + m) + c(O(\log n) - m)$. With a sufficiently large choice of c, this simplifies to $O(\log n)$.

In the third phase we check each of the remaining roots and find the minimum. This takes $O(\log n)$ time and the potential does not change. The overall amortized running time of extract minimum is therefore $O(\log n)$.



Fibonacci heap from Figure 1 after decreasing key of node 9 to 0. This node as well as its two marked ancestors are cut from the tree rooted at 1 and placed as new roots.

Operation **decrease key** will take the node, decrease the key and if the heap property becomes violated (the new key is smaller than the key of the parent), the node is cut from its parent. If the parent is not a root, it is marked. If it has been marked already, it is cut as well and its parent is marked. We continue upwards until we reach either the root or an unmarked node. In the process we create some number, say k, of new trees. Each of these new trees except possibly the first one was marked originally but as a root it will become unmarked. One node can become marked. Therefore the number of marked nodes changes by $-(k - 1) + 1 = -k + 2$. Combining these 2 changes, the potential changes

by $2(-k + 2) + k = -k + 4$. The actual time to perform the cutting was $O(k)$, therefore (again with a sufficiently large choice of c) the amortized running time is constant.

Finally, operation **delete** can be implemented simply by decreasing the key of the element to be deleted to minus infinity, thus turning it into the minimum of the whole heap. Then we call extract minimum to remove it. The amortized running time of this operation is $O(\log n)$.