DATA STRUCTURE
TREES
UNIT II

 U2. 1

Learning Objectives

Trees

- General Tree
- Binary trees
- AVL Trees
- Binary Tree Applications
- Threaded Tree
- B Trees
- B* Trees
- B+ Trees

 U2. 2

TREE

An Hierarchical Collection of Nodes

 U2. 3

Objectives

- Linear Lists and Trees
- Basic Terminology
- General Tree
- Binary Tree
- Traversal Methods
- Basic Operations
- Stack Based Traversal

Linear Lists And Trees

- Linear lists are useful for serially ordered data.
  - **(e_0, e_1, e_2, …, e_{n-1})**
  - Days of week.
  - Months in a year.
  - Students in a class.
- Trees are useful for hierarchically ordered data.
  - Employees of a corporation.
    - ✓President, vice presidents, managers, and so on.
  - Classes

Hierarchical Data And Trees

- The element at the top of the hierarchy is the root.
- Elements next in the hierarchy are the children of the root.
- Elements next in the hierarchy are the grandchildren of the root, and so on.
- Elements that have no children are leaves.

Definition

- A tree ***t*** is a finite nonempty set of elements called nodes.
- One of these elements is called the root.
- The remaining elements, if any, are partitioned into trees, which are called the sub-trees of the tree ***t***.

Basic Terminology

Node (Vertex, Element):
- Main component of a tree structure
- Stores **N** data and **(N-1)** links to other nodes.

Parent (Father)
- Immediate predecessor of a node

Child (Son):
- Immediate successor of a node

Sibling
- Nodes having the same parent

Link (Edge, Branch):
- A pointer to a node in a tree.
- Means of traversing the tree

Contd...

Root:
- Specially designated node that has no parent.
- Node at the topmost level in the tree hierarchy

Leaf (Terminal node, External node):
- A node that has no child nodes
- At the bottommost level in the tree hierarchy

Level:
- Rank of hierarchy of a node.
- Root is said to be at level 0.
- If a node is at level ***l*** then its child nodes are level ***l+1***

Contd...

Path: (from n_l to n_k)

- Sequence of nodes n_l, $n_{l+1,}$ n_{l+2} ... n_k such that n_i is parent of n_{i+1} for 1< i <= k
- Length of this path is the number of links traversed in this path.
- Exactly one path from root to each node
- Depth of n_i is the length of unique path from root to n_i
- Height of n_i is the length of longest path from n_i to a leaf.

Height of a tree:

- Height of a tree equal to the depth of its deepest leaf which in turn is equal to the depth of the tree.

If there is a path from n_i to n_j then n_i is an **ancestor** of n_j and n_j is a **descendant** of n_i.

Contd...

Forest:

- A set of disjoint trees

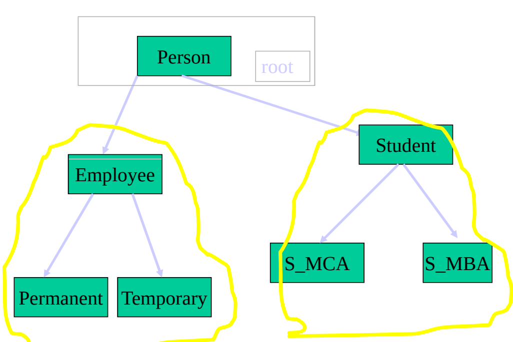
Free Tree

- Has no node designated as a root
- *A connected acyclic graph*

Ordered Tree:

- Child nodes are ordered from youngest to oldest

Visualization



Binary Tree

- Finite collection of elements / nodes that is either empty or partitioned into **three disjoint** subsets.
- The first subset is termed as the root element.
- The remaining elements (if any) are partitioned into two binary trees.
- These are called the left and right sub-trees of the binary tree.

Properties of Binary Trees

- If a binary tree contains *m* nodes at level *l*; it can contain at most *2m* nodes at level *l+1.*
- Maximum number of nodes at level l is 2^l.
- The total number of nodes in a complete binary tree= *sum of number of nodes at each level between 0 and d (depth)*
  - = 2^0 + 2^1 + 2^2 + … + 2^d= 2^(d+1) –1

Contd...

- The sub-trees of a binary tree are ordered;
- Are different when viewed as binary trees.
- Are the same when viewed as trees.

Contd...

$H_{max} = N$

$H_{min} = [Log_2N] + 1$

$N_{min} = H$

$N_{max} = 2^H - 1$

$B = H_L - H_R$ (Balance Factor)

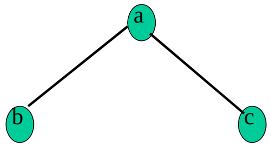
Binary Tree Traversal Methods

- In a traversal of a binary tree, each element of the binary tree is visited exactly once.
- During the visit of an element, all action (display, evaluate the operator, etc.) with respect to this element is taken.

Binary Tree Traversal Methods

- Preorder
- Inorder
- Postorder
- Level order

Preorder Example (visit = print)

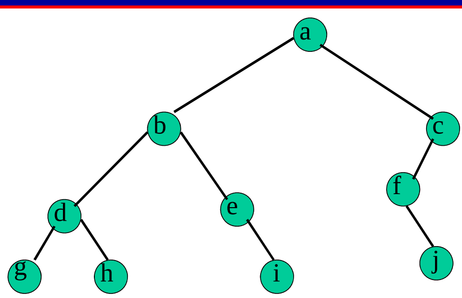


a b c

Preorder Traversal

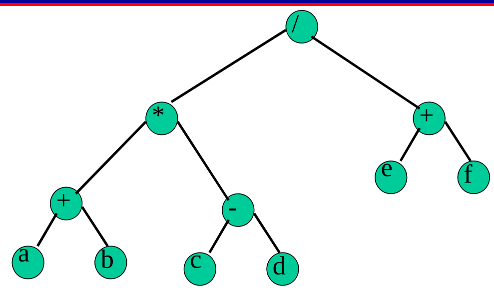
```
void preOrder(BinaryTreeNode t)
{
    if(t != null)
    {
      visit(t);
      preOrder(t.leftChild);
      preOrder(t.rightChild);
    }
}
```


Preorder Example (visit = print)



a b d g h e i c f j

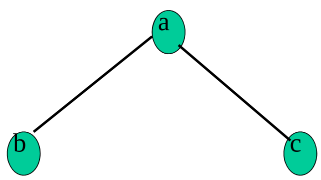
Preorder of Expression Tree



/ * + a b - c d + e f

Gives prefix form of expression!

Inorder Example (visit = print)

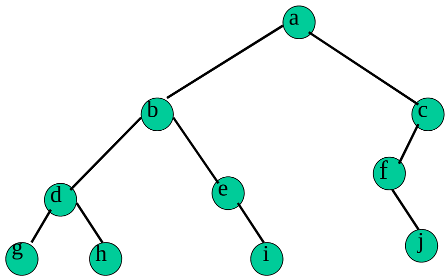


b a c

Inorder Traversal

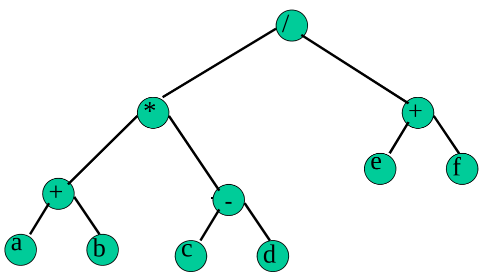
```
void inOrder(BinaryTreeNode t)
{
   if (t != null)
   {
     inOrder(t.leftChild);
     visit(t);
     inOrder(t.rightChild);
   }
}
```

Inorder Example (visit = print)



g d h b e i a f j c

Inorder of Expression Tree

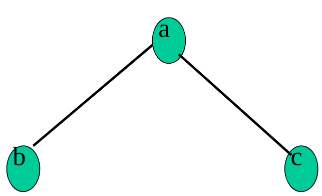


a + b * c - d / e + f

Gives infix form of expression (sans parentheses)!

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh

Postorder Example (visit = print)



b c a

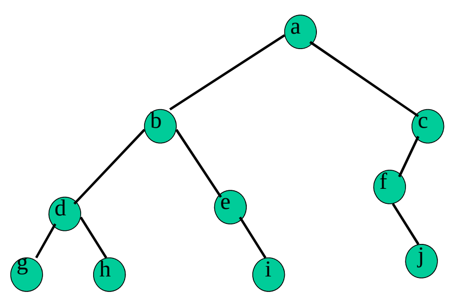
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh

© Bharati Vidyapeeth’s Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh

Postorder Traversal

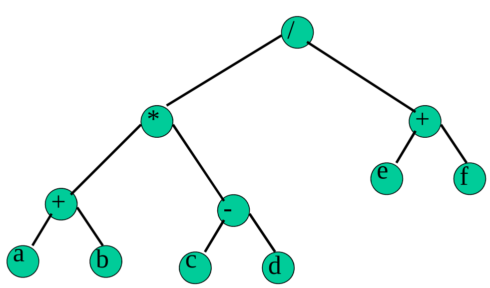
```
Void postOrder(BinaryTreeNode t)
{
if (t != null)
    {
      postOrder(t.leftChild);
      postOrder(t.rightChild);
      visit(t);
    }
}
```


Postorder Example (visit = print)



g h d i e b j f c a

Postorder of Expression Tree



a b + c d - * e f + /

Gives postfix form of expression!

Traversal Applications

- Make a clone.
- Determine height.
- Determine number of nodes.

Make a Clone Using Traversal

```
TreePtr Clone (srcTree, destTree)
   If srcTree Is Not NULL
      clone the root in the visit step.
      Set destTree:Left to Clone(srcTree:Left,
        destTree:Left)
      Set destTree:Right to Clone(srcTree:Right,
        destTree:Right)
```


Determine Height

```
If tree is NULL
   height = -1
Else
   Height= Max (Height of Left Subtree, Height
     of Right Subtree) +1
```

Determine Number of Nodes

```
If tree is NULL
    nodeCount = 0
Else
    nodeCount = Count (LeftSubtree) +
      Count (RightSubtree) +1
```


Searching Using Traversal

Search for **data** in the tree **N**
  If (if **N** is **NULL** or **N** contains **data**)
    Return **N**
  Else If (**data** greater than contents of **N**)
    Return the results of Searching the N:right
  Else If (**data** lesser than contents of **N**)
    Return the results of Searching the N:left

Make Empty

Make the tree N Empty
  If **N** is not Empty
    Make the left branch of **N** empty
    Make the right branch of **N** empty
    Free **N**

Inserting a Node

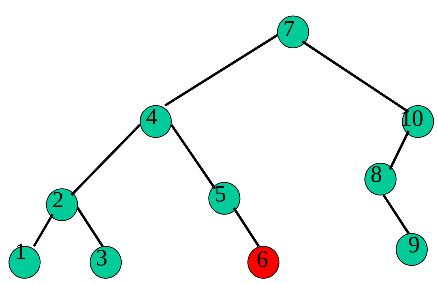
```
Insert (TPtr, data )
   If TPtr is NULL
      Set TPtr= Create a Node with contents data
   Else If data is greater than contents of TPtr
      Set TPtr:Right= Insert (TPtr:Right, data)
   Else If data is lesser than contents of TPtr
      Set TPtr:Left= Insert (TPtr:Left, data)
   Return TPtr
```


Deleting a Node

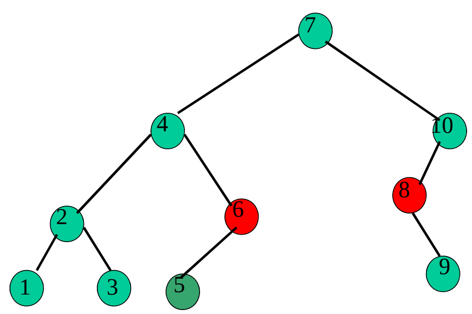
Three possible cases

- **CASE I** The node to be deleted has no child nodes
  - ✓Free the node
  - ✓Set the appropriate pointer of *father* NULL
- **CASE II** The node to be deleted has one child node
  - ✓Adjust the pointer of parent to bypass the node to be deleted
  - ✓Free the node
- **CASE III** The node to be deleted has two child nodes
  - ✓Replace the data in the *node to be deleted* with the *smallest data in the right subtree*
  - ✓Recursively delete the replacing node from the right subtree

Case I

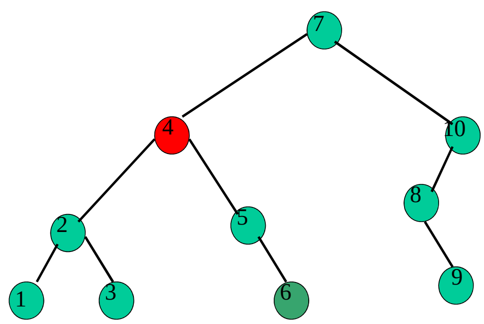


Case II



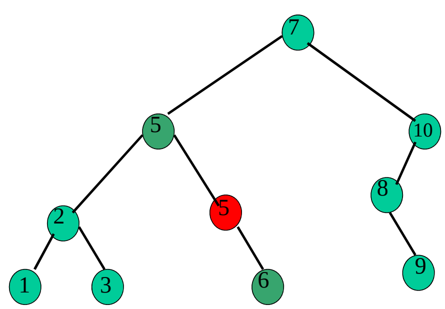
 U2. 40

Case III



 U2. 41

Contd...



 U2. 42

Deleting a Node

Three possible cases
- The node to be deleted has no child nodes
  - ✓Free the node
  - ✓Set the appropriate pointer of *father* NULL
- The node to be deleted has one child node
  - ✓Adjust the pointer of parent to bypass the node to be deleted
  - ✓Free the node
- The node to be deleted has two child nodes
  - ✓Replace the data in the *node to be deleted* with the *smallest data in the right subtree*
  - ✓Recursively delete the replacing node from the right subtree

 U2. 43

Deleting a Node: Recursive Delete

```
TreePtr DeleteNode ( TreePtr T, int Data)
    TreePtr tempPtr;
    If T is not NULL
      If Data is lesser than T:Data
       Set T:Left to DeleteNode (T:Left, Data)
      Else If Data is greater than T:Data
       Set T:Right to DeleteNode (T:Right, Data)
      Else              // node found
       If (T:Left AND T:Right)
           Set tempPtr to FindMin (T:Right)
           Set T:Data to tempPtr:Data
           Set T:Right to DeleteNode (T:Right, T:Data)
```

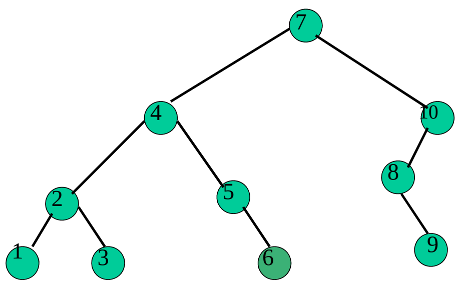
 U2. 44

Recursive Delete: Contd.

```
       Else
           Set tempPtr to T
           If T:Left is NULL
               Set T to T:Right
           Else
               Set T to T:Left
           Free tempPtr
    Return T
```

 U2. 45

Case I



 U2. 46

Find Min

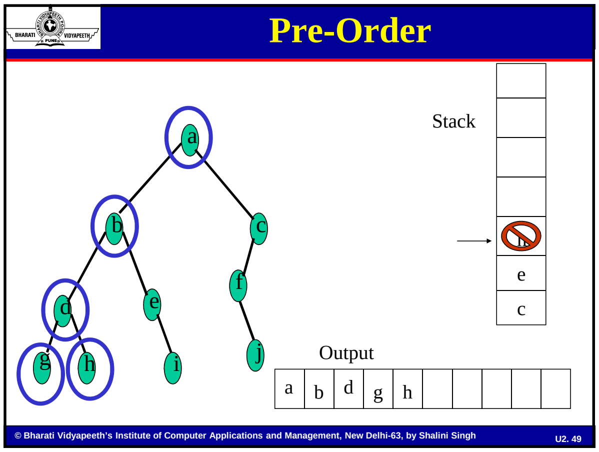
```
Position FindMin( SearchTree T)
{
   If( T == NULL)
        return NULL;
   else
        if(T->Left == NULL)
              return T;
        else
              return FindMin(T->Left);
}
```

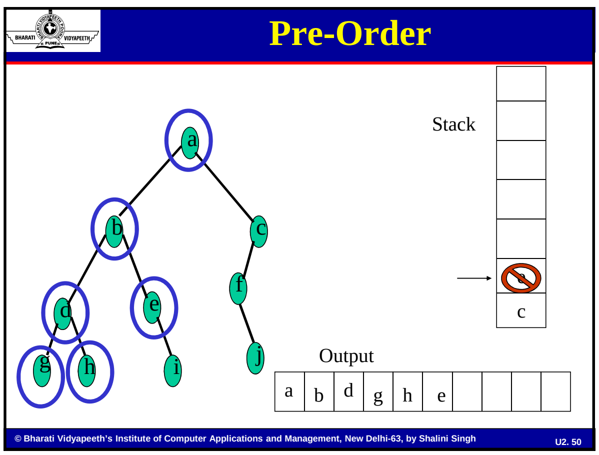
 U2. 47

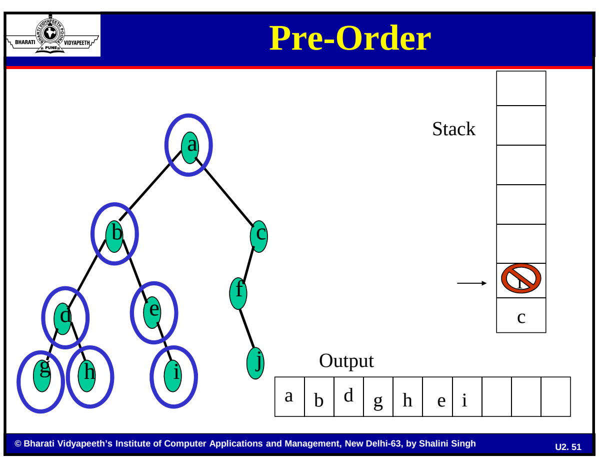
Stack Based Traversal (Pre-order)

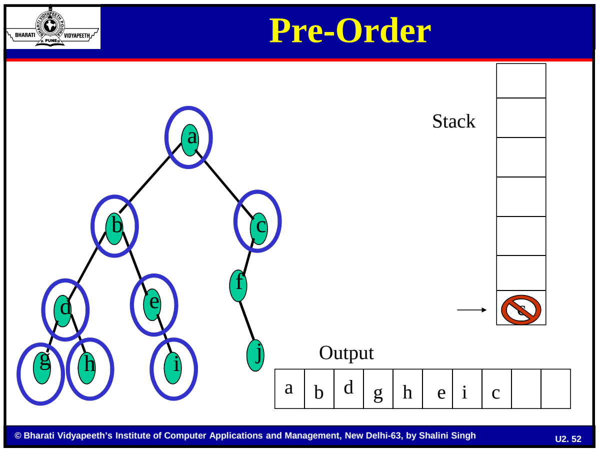
Set **N** to **Root**
While (**N**)
   Process (**N**)
   If **N:Right**
      Push **N:Right** on **Stack**
   Set **N** to **N:Left**
   If **N** is **NULL**
      Pop **Stack** and place the result in **N**
         Done if **Stack** Pop Fails because of Empty Stack

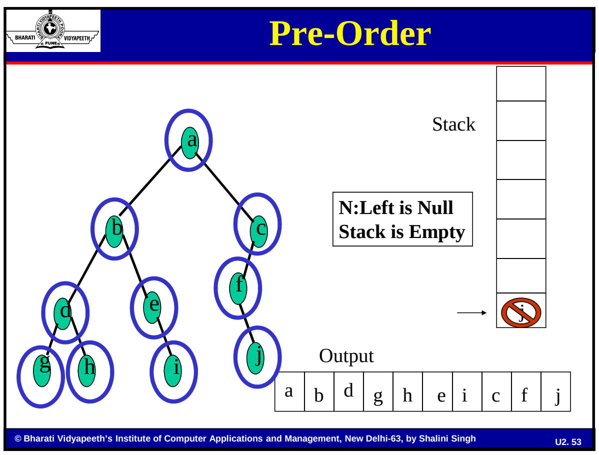
 U2. 48


Pre-Order
Stack
e
c
a b c d e f g h i j
Output
a b d g h
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh
U2. 49


Pre-Order
Stack
c
a b c d e f g h i j
Output
a b d g h e
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh
U2. 50


Pre-Order
Stack
c
a b c d e f g h i j
Output
a b d g h e i
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh
U2. 51

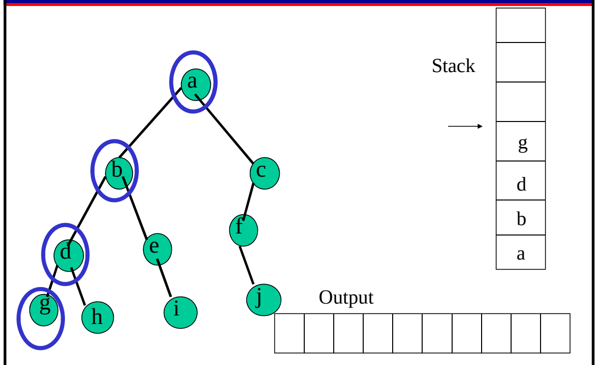




Stack Based Traversal (In-order)

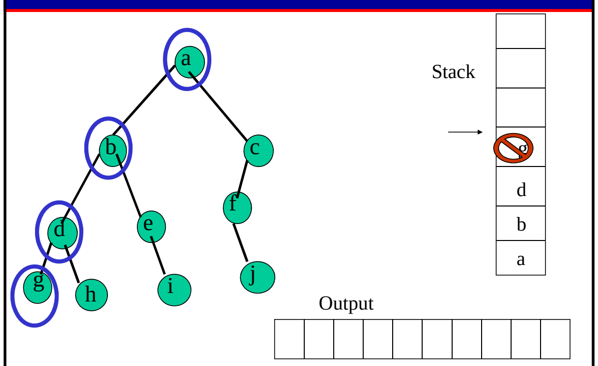
```
Set N to Root
While (N)
    While (N)
        Push N on Stack
        Set N to N:Left
    Do
        Set N to Pop Stack
        Done if Pop Fails because of Empty Stack
        Process (N)
        If N:Right is Not NULL
            Set N to N:Right
            Break out of Loop
    While (N is Not NULL)
```

In-Order



 U2. 55

In-Order



 U2. 56

In-Order

To be continued in the same fashion…

 U2. 57

Stack Based Traversal (Post-order)

- Will require to run the push-pop sequence twice.
  - Once when the left branch has not been processed
  - Once after the left branch has been processed
- Thus there is a need to distinguish between the two states
- Can do so by encapsulating the Node in another structure that hold a *flag (to indicate state)* and a pointer to the Tree Node
  - Say **sN** is a variable of such a structure

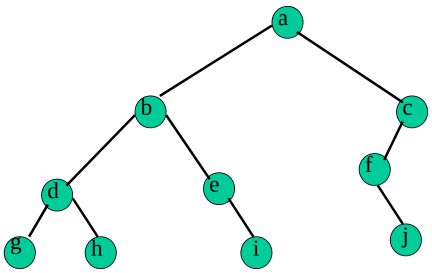
Stack Based Traversal (Post-order)

```
struct postOrderTrav
{
   int flag;
   struct TreeNode *N;
};

struct postOrderTrav sN;
Set N to Root
```


Contd...

```
While not Done
    While N is Not NULL
      Set sN:N to N
      Set sN:Flag to 1
      Push sN on Stack
      Set N to N:Left

    Do
        Pop Stack and place the result in sN
                Done if Pop Fails
        If sN:Flag is 1
                Set sN:Flag to 0
                Push sN on Stack
                Set N to N:Right
                Step out of loop
        Else
          Process (N)
    While (N) End While
```

Level-Order Example (visit = print)

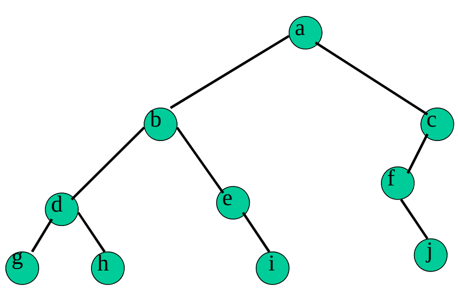


a b c d e f g h i j

Level Order

Set **N** to **Root**
Insert **N** into the **Queue**
While **Queue** is not **Empty**
  Perform *Dequeue* on **Queue** and place the result in **N**
  If **N** is **Not NULL**
    Process (**N**)
    Insert **N:Left** into the **Queue**
    Insert **N:Right** into the **Queue**

Level-Order Example (visit = print)



a b c d e f g h i j

What we Learned

- ✓ Linear Lists and Trees
- ✓ Basic Terminology
- ✓ Binary Tree
- ✓ Traversal Methods
- ✓ Basic Operations
- ✓ Stack Based Traversal

AVL Trees

discovered by

G.M. Adelson-Velskii and E.M. Landis

Objectives

- BST
- Balanced Trees
- Unbalanced Trees
- AVL Trees
- AVL vs. Balanced Trees
- Basic Operations & Implementation

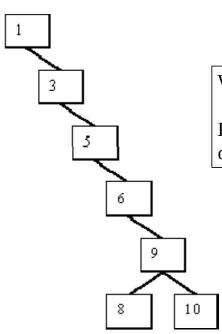
BST

- Searching in a BST is most efficient when the tree is balanced
- In other words, a balanced tree gives the best search/insert times for a given number of nodes
- If a tree is ***balanced*** then it implies there is no binary tree of lesser height that can have the same number of nodes as the tree itself.
- A tree is said to be ***perfectly balanced*** if the following rule applies for all the nodes of the tree
- *height of left subtree = height of right subtree*

Balanced Trees

- Since the number of nodes in a full binary tree of height h is given by the expression $2^{h+1} - 1$
- A binary tree of n elements is balanced if:
  $2^h - 1 < n \leq 2^{h+1} - 1$
- So, finding an element, inserting, and removing in a balanced tree containing n elements are O(log n) operations
- Unfortunately, binary search trees can become unbalanced and, in the worst case, these operations then become O(n)

Unbalanced Tree: Example



Worst case search!!!

Requires -------- operations

AVL Trees

- AVL trees are named after two Russian mathematicians **G.M. Adelson-Velskii** and **E.M. Landis**, who discovered them in 1962.
- An AVL tree is
  - A binary search tree
  - In which the **heights of the left and right *subtrees* of the root differ by at most 1**, and
  - In which the left and right subtrees of the root are again AVL trees
- *Some adjustments might be needed after insertion / deletion to maintain AVL properties.*

 U2. 70

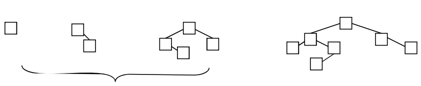
AVL vs Balanced

- An AVL tree is a type of binary search tree
  - which is ***nearly as good as*** a balanced tree for time complexity of the operations, and
  - whose structure we can maintain as insertions and deletions proceed
- i.e. an AVL tree may or (in certain cases) may not be balanced
- But, the structure is such that it supports efficient search / insert operations.

 U2. 71

AVL Trees: Examples

AVL



AVL and balanced

Legal AVL, but not balanced

Non-Legal AVL

 U2. 72

Implementing AVL Trees

- To implement algorithms for inserting and deleting from an AVL tree, we associate a **balance factor** with each node
- This is difference between heights of left and right subtrees **(Note that the difference cannot have a magnitude greater than 1).**
- Allowed values for the *balance factor* are:
  - **0 =>** Height (L_T) = Height (R_T)**,**
  - **1 =>** Height (L_T) > Height (R_T) or
  - **-1 =>** Height (L_T) < Height (R_T)

  depending on whether the left subtree of the node has height greater than, less than or equal to that of the right subtree

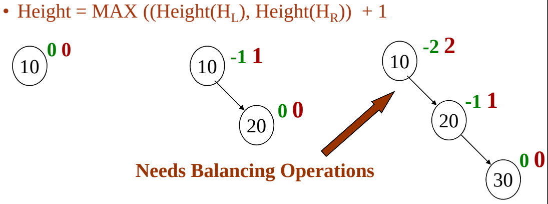
Height of a NULL tree is -1 by definition

Inserting into an AVL Tree

- Though the basic insertion algorithm is similar to that of a binary search tree; as nodes are randomly added or removed the ***balance factor*** can get disturbed.
- Balance factor will get disturbed when
  - the new node is added to a subtree of the root that is **higher** than the other subtree **and**
  - the new node increases the height of the subtree
- In this case we have to carry out some ***balancing operations*** in the neighborhood of the root

AVL Tree: Insertion

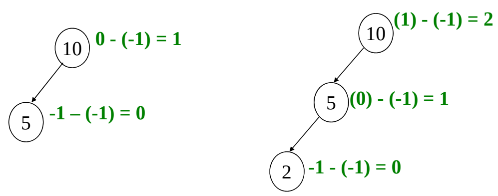
- Consider the insert sequence: 10, 20, 30
- As nodes are added the change in the balance factor is evident
- Addition of 30 leads to violation of AVL property
- balance factor = $H_L - H_R$
- Height = MAX ((Height(H_L), Height(H_R)) + 1



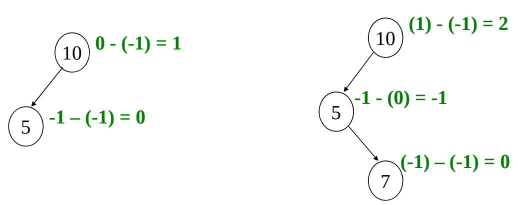
Balanced -> Unbalanced

- Unless keys appear in the perfect order, imbalance is bound to occur.
- These imbalance conditions can be reduced to one of four reference cases, listed as:
  1. Addition in left subtree of left child (**Left-Left Imbalance**).
  2. Addition in right subtree of left child (**Left-Right Imbalance**).
  3. Addition in right subtree of right child (**Right-Right Imbalance**).
  4. Addition in left subtree of right child (**Right-Left Imbalance**).

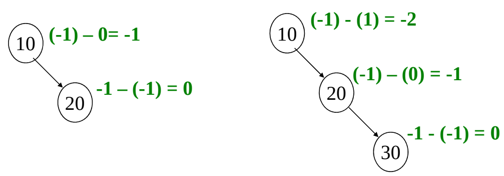
Case – I: Left-Left Imbalance



Case – II: Left-Right Imbalance

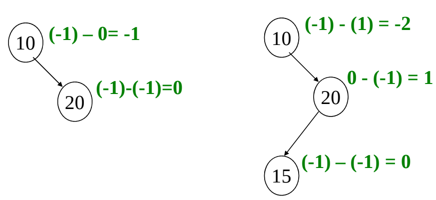


Case – III: Right-Right Imbalance



 U2. 79

Case – IV: Right-Left Imbalance



 U2. 80

Generalization

Careful inspection reveals that

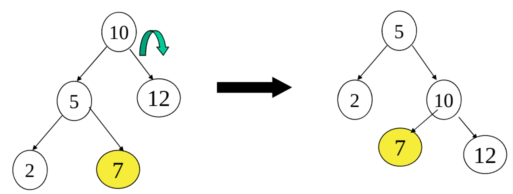
- Case - I and Case – III are mirror images of each other
- Similarly, Case - II and Case – IV are mirror images of each other

Thus, the process for restoring balance can be reduced to two generalized cases…

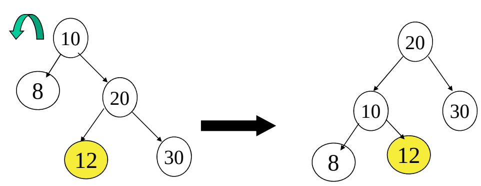
- Straight line addition (I & III)
- The so-called *Dog-leg* pattern (II & IV)

 U2. 81

Restoring Balance: Case - I

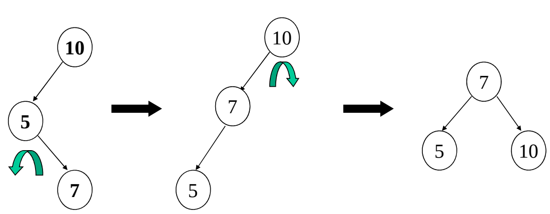


Restoring Balance: Case - II



Thus, a single rotation, clockwise / anticlockwise restores balance in case of *straight line addition*

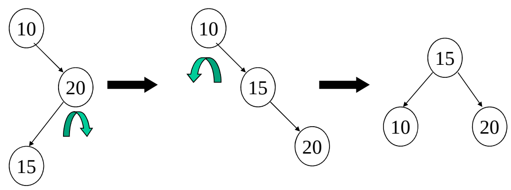
Restoring Balance: Case III



Strategy:

1. Transfer extra weight in left branch
2. Rotate clockwise

Restoring Balance: Case IV



Strategy:

1. **Transfer extra weight in right branch**
2. **Rotate anti-clockwise**

Restoring Balance

Balance, in either of the cases can be restored by means of a process termed ***rotation***

- In cases I & III balance is restored by means of ***single rotation***
- Whereas, in cases II & IV ***double rotation*** is used for the same purpose.

Principles

- Imbalance will occur only on the path from root to the inserted node.
  - As only these nodes have their subtrees altered
- Rebalancing needs to be done at the ***deepest unbalanced node***

Left-Left Imbalance: Before Rotation

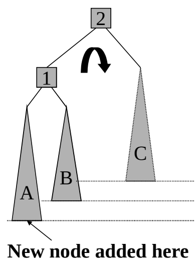
B and C have the same height

A is one level higher

Therefore

- Make 1 the new root
- 2 its right child; and
- B and C the subtrees of 2

Note the levels



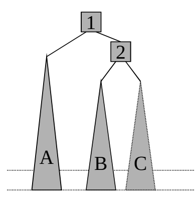
(Solve right-right imbalance, by symmetry)

 U2. 88

Left-Left Imbalance: After Rotation

Result: A more balanced and legal AVL tree

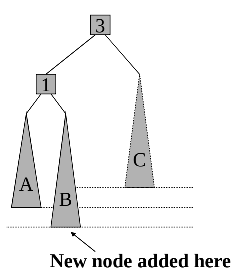
Note the levels



 U2. 89

Left-Right Imbalance: Before Rotation

- Can't use the left-left balance trick
- Because now it's the *middle subtree*, i.e. it is B, that's too deep.
- So we consider what's inside B...

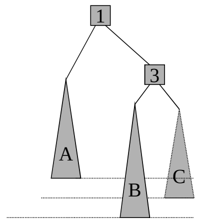


(Solve right-left imbalance by symmetry)

 U2. 90

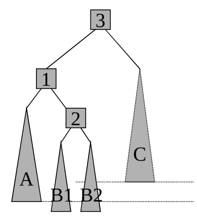
LR Imbalance: Inadequacy of Single Rotation

- Let's try to apply single rotation in case of Left-right Imbalance:
- Steps
  - Make 1 the new root
  - 3 it's right child
  - B & C the subtrees of 3



Left-Right Imbalance: Double Rotation

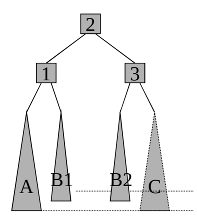
- B will have two subtrees containing at least one item (just added)
- We do not know which is too deep - set them both to 0.5 levels below subtree A ***so that either can be adjusted half a level up / down***



Left-Right Imbalance: Double Rotation

- Neither 1 nor 3 worked as root node so make 2 the root
- Rearrange the subtrees in the correct order
- No matter how deep B1 or B2 (+/- 0.5 levels) we get a legal AVL tree again

Equivalent to:
SingleRotateForRight (1)
SingleRotateForLeft (3)



Approach towards Solution

- ***Spot the where imbalance has occurred***:
  - you'll need to make sure you're updating your balance factors, all the way from each new node to root.
- ***Spot which re-balancing operation to use***:
  - from the imbalanced node,
  - draw subtree triangles left and right.
  - Then break the deeper subtree itself into two subtrees.
  - The location of the deeper sub-subtree will tell you which rule to use (see diagrams on previous slides)

 U2. 94

Example

- *Example question*:
- Sketch the various stages involved in inserting the following keys, in the order given, into an AVL tree:

342, 206, 444, 523, 607, 301, 142, 183, 102, 157, 149

 U2. 95

Algorithm: Insert

```
AVLTree AVLInsert ( ElementType X, AVLTree T)
BEGIN
   IF ( T IS NULL) THEN
        Allocate Memory to T
        T:Data <- X
        T:Height <- 0
        T:Left <- T:Height <- NULL
   ELSEIF ( X < T:Data )
        T:Left <- AVLInsert (X, T:Left);
        IF ( Height(T:Left) – Height(T:Right) = 2) THEN
                IF ( X < T:Left:Data )
                        T <- SingleRotateLL (T)
                ELSE
                        T <- DoubleRotateLR (T)
        ENDIF
```

 U2. 96

Algorithm: Insert Contd.

```
ELSEIF ( X > T:Data )
    T:Right <- AVLInsert (X, T:Right);
    IF ( Height(T:Right) – Height(T:Left) = 2) THEN
        IF ( X > T:Right:Data )
            T <- SingleRotateRR (T)
        ELSE
            T <- DoubleRotateRL (T)
ENDIF
ENDIF
T:Height= MAX ( Height (T:Left), Height
        (T:Right) ) + 1;
return T;
END
```


Algorithm: Height

```
INT Height ( AVLTree T)
BEGIN
  IF ( T IS NULL)
     RETURN –1;
  ENDIF
  RETURN T:Height
END
```


Algorithm: SingleRotateForLeft

```
AVLTREE SingleRotateForLeft (AVLTree T)
BEGIN
  AVLTree temp;
  temp <- T:Left;
  T:Left <- temp:Right;
  temp:Right <- T;

  T:Height <- MAX (Height(T:Left), Height(T:Right)) + 1;
  //adjust height of temp;

  return temp; //the new root
END
```

Algorithm: DoubleRotateForLeft

```
AVLTree DoubleRotateForLeft (AVLTree T)
BEGIN
  T:Left <- SingleRotateForRight ( T:Left );
  return SingleRotateForLeft ( T );
END
```


Algorithm: SingleRotateForRight

```
AVLTREE SingleRotateForRight(AVLTree T)
BEGIN
  AVLTree temp;
  temp <- T:Right;
  T:Right <- temp:Left;
  temp:Left <- T;

  T:Height <- MAX (Height(T:Left), Height(T:Right)) + 1;
  temp:Height <- MAX (Height(temp:Left), Height(temp:Right)) +
  1;
  //adjust height of temp;
  return temp; //the new root
END
```


Algorithm: DoubleRotateForRight

```
AVLTree DoubleRotateForRight(AVLTree T)
BEGIN
T:Right <- SingleRotateForLeft (T:Right );
  return SingleRotateForRight ( T );
END
```


What we Learned

- ✓ BST
- ✓ Balanced Trees
- ✓ Unbalanced Trees
- ✓ AVL Trees
- ✓ AVL vs. Balanced Trees
- ✓ Basic Operations & Implementation

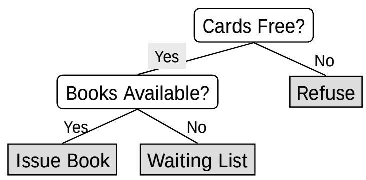
BINARY TREES: APPLICATIONS

Objectives

- Decision Trees
- Binary Expression Trees
- Evaluation of Arithmetic Operations
- Building Expression Trees

Decision Trees

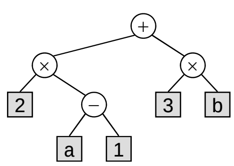
- Binary tree associated with a decision process
  - internal nodes: questions with yes/no answer
  - external nodes: decisions
- Example: Library decision



 U2. 106

Binary Expression Trees

- Binary tree associated with an arithmetic expression
  - internal nodes: operators
  - external nodes: operands
- Example: arithmetic expression tree for the expression
  - $(2 \times (a - 1) + (3 \times b))$



 U2. 107

Characteristics

A special kind of binary tree in which:

1. Each **leaf node** contains a single operand
2. Each **nonleaf node** contains a single binary operator
3. The left and right subtrees of an operator node represent **subexpressions** that must be evaluated **before** applying the operator at the root of the subtree.

 U2. 108

Inductive Definition

- A single atomic operand is represented by a single node binary tree
- Consider expressions **E** and **F** that are represented by binary trees **S** and **T**, respectively, and op is a binary operator, then
  - **(E op F)** is represented by the binary tree **U** consisting a new root labeled **op** with left child **S** and right child **T**.
- Consider an expression **E,** represented by tree **S** and **op** is a prefix unary operator, then
  - **(op E)** is represented by the binary tree **T** consisting a new root labeled **op** with empty left child and right child **S**.

Significance of Level (Depth)

- Level of a node indicates its order of evaluation
- ***Parentheses are not required to indicate precedence***.
- **Operations at *lesser depth* of the tree are evaluated later** than those below them.
- The operation at the root is always the last operation performed.

Implementation

- **TreeNode:Info** can contain either of the following:
  - Operator
  - Operand
- Thus the member Info must be capable of holding either of the two members

enum

```
struct Info
{
   OpType whichType;
   union ------
   {
      char operater ;
      int operand ;
   }
};
```

Printing Arithmetic Expressions

Application of tree traversals:

- Preorder: yields Prefix
- Inorder: yields Infix
- Postorder: yields Postfix

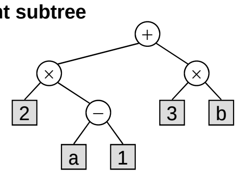
Inorder: Infix

Specialization of an inorder traversal

- print operand or operator when visiting node
- **print "(" before traversing left subtree**
- **print ")" after traversing right subtree**

Why

```
Algorithm printExpression(v)
   if isInternal (v)
      print("(")
      inOrder (leftChild (v))
   print(v.element ())
   if isInternal (v)
      inOrder (rightChild (v))
      print (")")
```



$((2 \times (a - 1)) + (3 \times b))$

Evaluate Arithmetic Expression

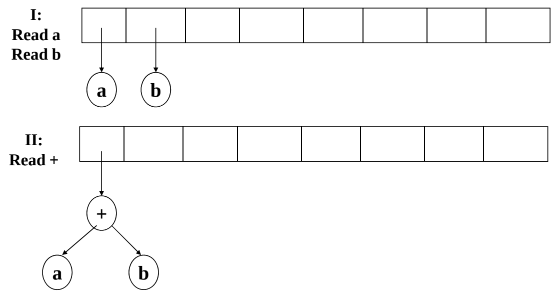
Specialization of a postorder traversal

- recursive method returning the value of a subtree
- when visiting an internal node, combine the values of the subtrees

```
Algorithm evalExpr(v)
   if isExternal (v)
      return v.element ()
   else
      x ← evalExpr(leftChild (v))
      y ← evalExpr(rightChild (v))
      ◊ ← operator stored at v
      return x ◊ y
```

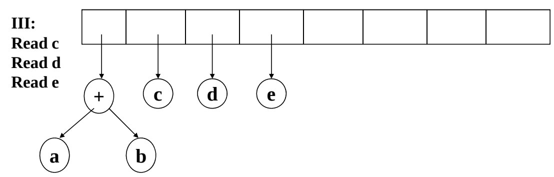
Building Expression Trees

a b + c d e + * *



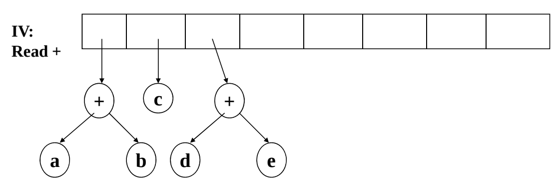
Building Expression Tree

a b + c d e + * *



Building Expression Tree

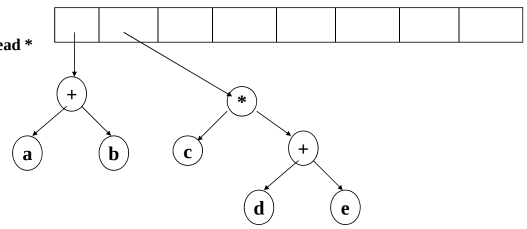
a b + c d e + * *



Building Expression Tree

a b + c d e + * *

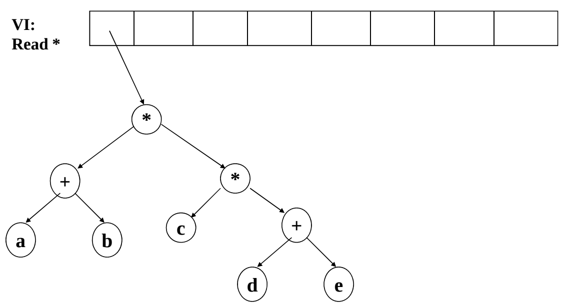
V:
Read *



Building Expression Tree

a b + c d e + * *

VI:
Read *



Building Expression Trees

```
While hasMoreSymbols (Expression)
Symbol= readNextSymbol (Expression)
If isOperand (Symbol)
    Node= CreateNode (Symbol)
    Push (Stk, Node)
ElseIf isOperator (Symbol)
    T1= Pop (Stk)
    T2= Pop (Stk)
    Node= CreateNode (Symbol)
    Node:Left= T1
    Node:Right= T2
    Push (Stk, Node)
Endif
```

What we Learned

- ✓ Decision Trees
- ✓ Binary Expression Trees
- ✓ Evaluation of Arithmetic Operations
- ✓ Building Expression Trees

Threaded Trees

Objectives

- Binary Tree Traversal
- Thread as Solution
- Threaded Tree Traversal
- Basic Operations

Binary Tree: Traversal

Traversal:
- Recursive
- Non-recursive

Both procedures use a stack to store information about nodes.

Problem:
- Some **additional time** has to be spent to maintain the stack
- Some **more space** has to be set aside for the stack itself.
- In the worst case, the stack may hold information about **almost every node** of the tree
- A serious concern for very large trees.

Solution: Threads

- In order to improve efficiency:
  - **The stack is *incorporated* as part of the tree**.
  - This is done by using ***threads*** in a given node.
  - **Threads are pointers to the predecessor and successor of the node according to a certain sequence**, and
  - The trees whose nodes use threads are called **threaded trees**.
- Requirements:
  - Four pointer fields
    - ✓Two for children,
    - ✓Two for predecessor and successor
  - would be needed for each node in the tree, *which again takes up valuable space*.

Utilizing Space

- Binary trees have a lot of wasted space
  - All the leaf nodes each have 2 null pointers
- We can use these pointers to help us in tree traversals by setting them up as threads.
- We make the pointers point to the next / previous node in a traversal
- Thus each pointer can act as an actual ***link*** or a ***thread***, so we keep a ***boolean*** for each pointer that tells us about the usage.

Explanation

Thus the problem is solved by **overloading** existing pointer fields.

- **An operator is called overloaded if it can *have different meanings*;**
- **The * operator in C is a good example, since it can be used as the multiplication operator or for dereferencing pointers.**
- **In threaded trees, left or right pointers are pointers to children, but they can also be used as pointers to predecessors and successors, thereby overloaded with meaning.**

 U2. 127

Threads: Usage

- For an overloaded operator, **context** is always a disambiguating factor: **Example!!!**
- In threaded trees, however, a new field has to be used to indicate the current meaning of the pointers.
- The left pointer is either a pointer to the **left child or to the predecessor**.
- Analogously, the right pointer will point either to the **right child or to the successor**.
- The meaning of predecessor and successor differs depending on the sequence under scrutiny.

 U2. 128

Variations

Depending upon traversal:

- In-threaded
- Pre-threaded
- Post-threaded

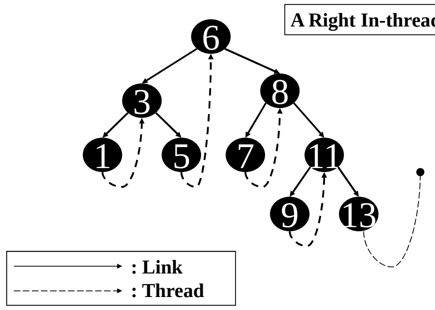
Depending upon threading technique:

- Left <trav type> threaded
- Right <trav type> threaded
- <trav type> threaded

 U2. 129

Threaded Tree Example

A Right In-threaded Tree

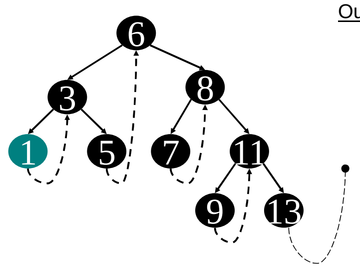


Traversal: Right In-threaded Tree

- Start at the leftmost node in the tree,
  - print it,
  - and follow its right child
- If you follow a thread to the right, output the node and continue to its right
- If you follow a link to the right, go to the leftmost node, print it, and continue

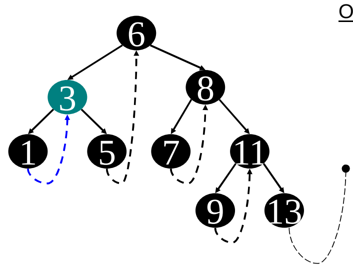
Traversal: Right In-threaded Tree

Output
1



Start at leftmost node, print it

Traversal: Right In-threaded Tree

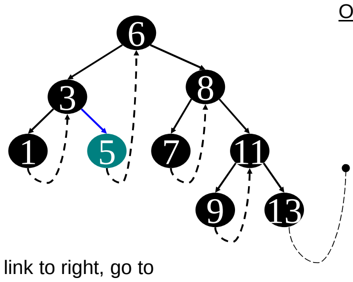


Output
1
3

Follow thread to right, print node

 U2. 133

Traversal: Right In-threaded Tree

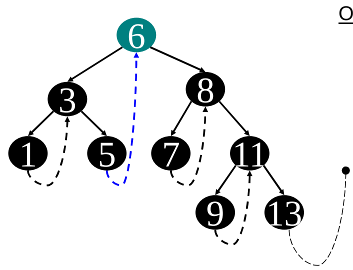


Output
1
3
5

Follow link to right, go to leftmost node and print

 U2. 134

Traversal: Right In-threaded Tree

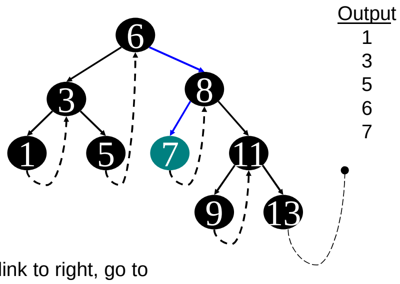


Output
1
3
5
6

Follow thread to right, print node

 U2. 135

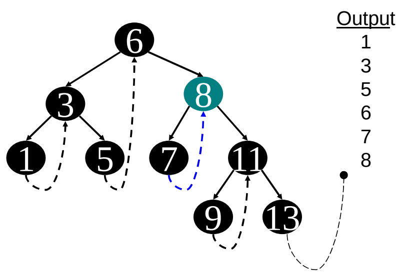
Traversal: Right In-threaded Tree



Follow link to right, go to leftmost node and print

 U2. 136

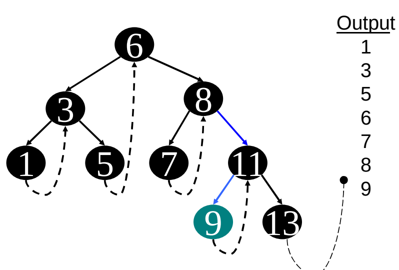
Traversal: Right In-threaded Tree



Follow thread to right, print node

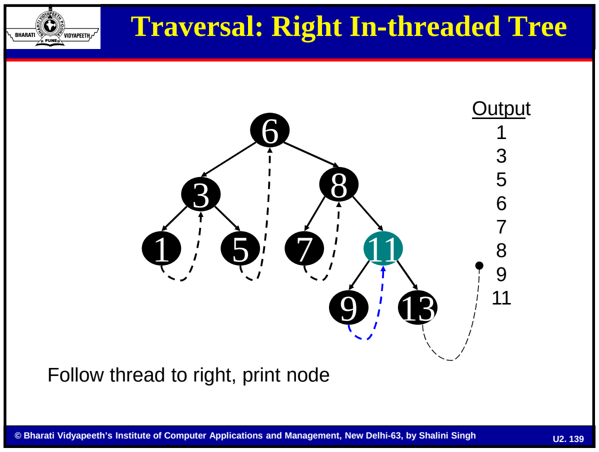
 U2. 137

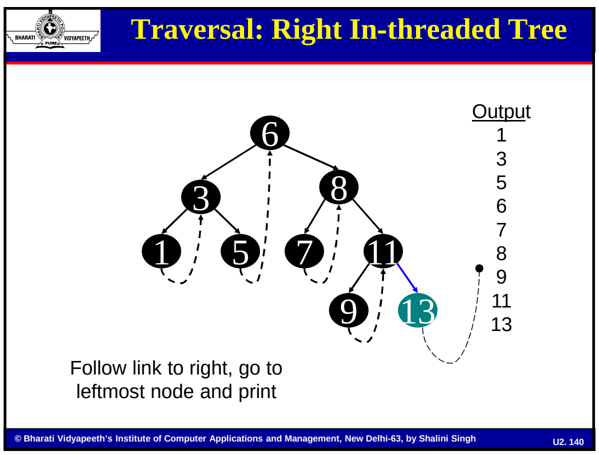
Traversal: Right In-threaded Tree

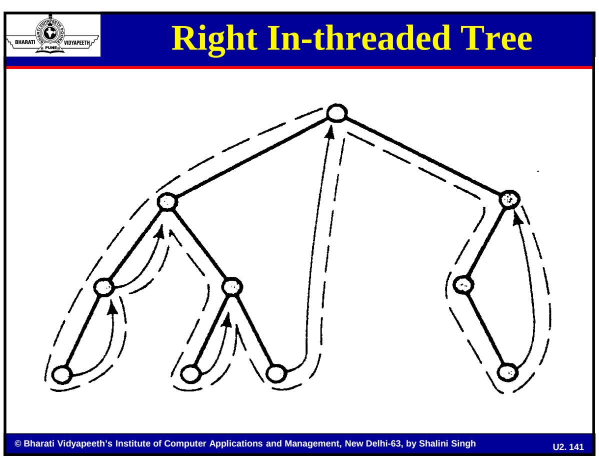


Follow link to right, go to leftmost node and print

 U2. 138


Traversal: Right In-threaded Tree
Output
1
3
5
6
7
8
9
11
6
3
8
1
5
7
11
9
13
Follow thread to right, print node
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh
U2. 139


Traversal: Right In-threaded Tree
Output
1
3
5
6
7
8
9
11
13
6
3
8
1
5
7
11
9
13
Follow link to right, go to
leftmost node and print
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh
U2. 140


Right In-threaded Tree
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh
U2. 141

Traversal: Alternate Approach

```
Ptr <- LeftmostChild (Root)
While (Ptr)
Do
  Process (Ptr)
  Ptr  <- InorderSuccessor (Ptr)
Done
```

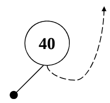

Traversal: Alternate Approach

```
InorderSuccessor ( Node )
Begin
  Succ <- Node:Right
  If ! Rthread
     While ( Succ:Left )
          Succe <- Succ:Left
  Return Succ
End
```


Adding a Node

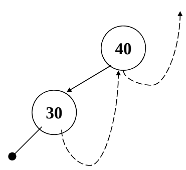
- Adding a node to a threaded tree requires handling of two special cases
  - Adding a left child
  - Adding a right child
  - **Why!!!**
- Consider adding the following data set
  - 40, 30, 25, 45, 35, 50, 27

Insertion Example



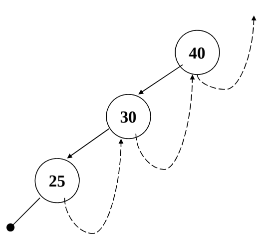
After: 40

Insertion Example



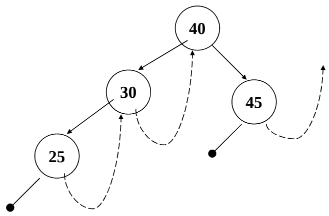
After: 40, 30

Insertion Example



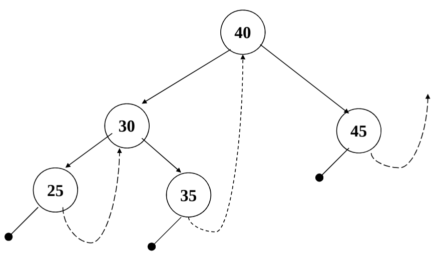
After: 40, 30, 25

Insertion Example



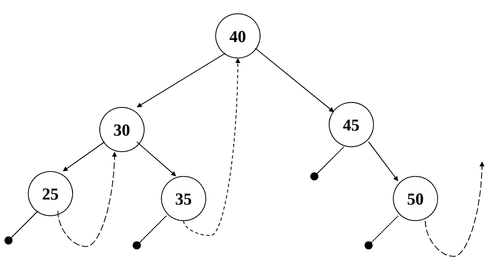
After: 40, 30, 25, 45

Insertion Example



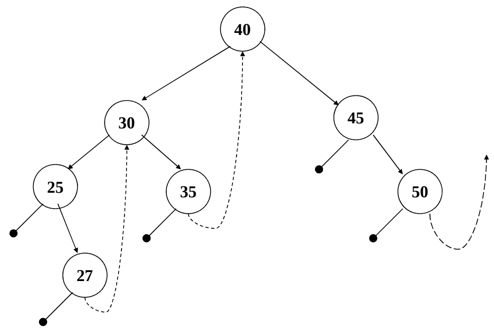
After: 40, 30, 25, 45, 35

Insertion Example



After: 40, 30, 25, 45, 35, 50

Insertion Example



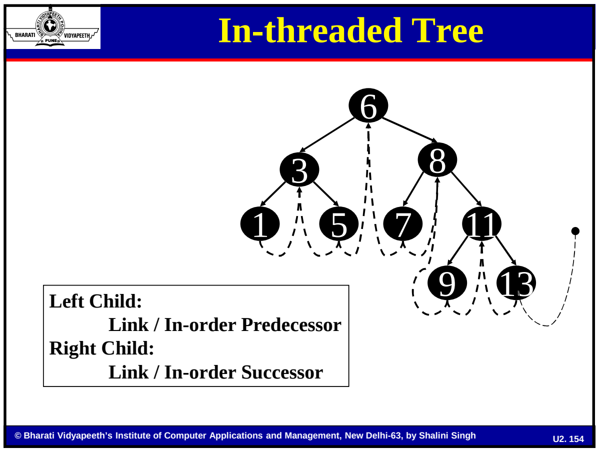
After: 40, 30, 25, 45, 35, 50, 27

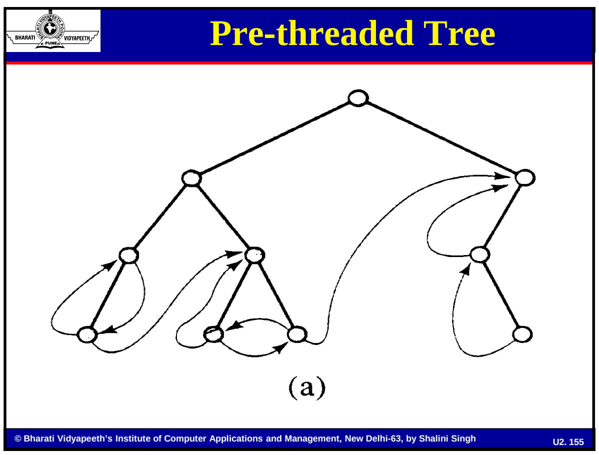
Adding Left Child

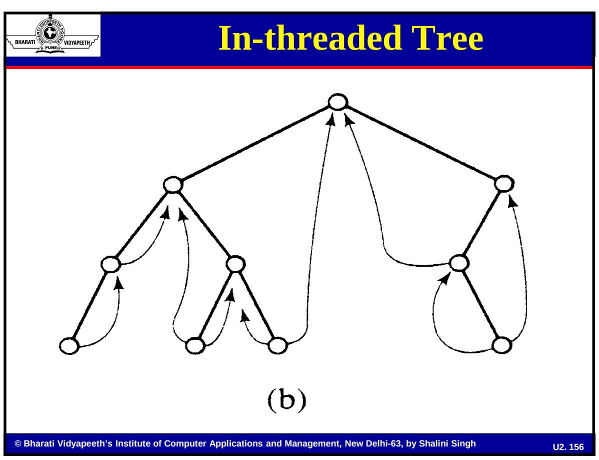
```
AddLeft ( Father, Data )
Begin
  // Allocate Memory & Set Data
  NewNode:Left <- NULL

  NewNode:RThread <- ------- //!!!
  NewNode:Right <- ------- //!!!

  Father:Left <- NewNode
End
```


Adding Right Child

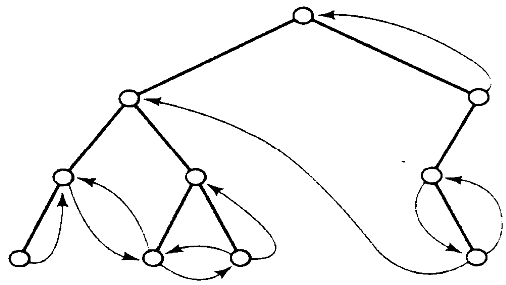
```
AddRight ( Father, Data )
Begin
  // Allocate Memory & Set Data
  NewNode:Left <- NULL

  NewNode:Rthread <- ------- //!!!
  NewNode:Right <- ------- //!!!

  Father:Right <- NewNode
  Father:Rthread <- ------- //!!!
End
```


In-threaded Tree
6
3
8
1
5
7
11
9
13
Left Child:
Link / In-order Predecessor
Right Child:
Link / In-order Successor
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh
U2. 154


Pre-threaded Tree
(a)
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh
U2. 155


In-threaded Tree
(b)
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh
U2. 156

Post-threaded Tree



(c)

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U2. 157

What we Learned

- ✓ Binary Tree Traversal
- ✓ Thread as Solution
- ✓ Threaded Tree Traversal
- ✓ Basic Operations

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U2. 158

B-Trees

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U2. 159

Objectives

- B-Tree
- Motivations
- Basic Operations
- B* Tree

Motivation for B-Trees

- So far we have assumed that we can store an entire data structure in main memory
- What if we have so much data that it doesn't fit?
- We will have to use disk storage but when this happens our time complexity fails
- The problem is that Big-Oh analysis **assumes that all operations take roughly equal time**
- **This is not the case when disk access is involved**

Motivation (cont.)

Assume that a disk spins at 3600 RPM
- ➢ In 1 minute it makes 3600 revolutions,
- ➢ One revolution occurs in 1/60 of a second, or 16.7ms
- ➢ On an average a disk access (half way round the disk) will take 8ms

Comparing with CPU instructions:
- ▪ 120 disk accesses a second ⇔ 10^6 instructions
- ▪ In other words, one disk access takes about the same time as 10,000 instructions

It is worth executing lots of instructions to avoid a disk access!!!

Motivation Contd...

- Assume that we use an AVL tree to create a database having a record count of the order of 10^6
- We still end up with a **very** deep tree with lots of different disk accesses;
  - $\log_2 1,000,000$ is about 20,
  - so this takes about 0.2 seconds (if there is only one user of the program)
- We know we can't improve on the log *n* for a binary tree
- **Solution –**
  - Use more branches and thus *lesser height*!
  - As branching increases, depth decreases

B - Trees

Definition: A balanced search tree in which

- Every node has between m/2 and m children,
  - where m>1 is a fixed integer.
  - m is the order.
- The root may have as few as 2 children.

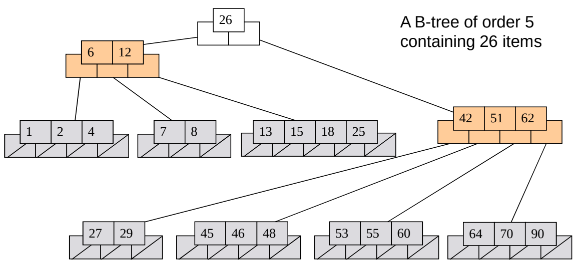
Definition of a B-tree

A B-tree of order *m* is an ***m*-way** tree (i.e., a tree where each node may have up to *m* children) in which:

1. The number of keys in each non-leaf node is one less than the number of its children and these keys partition the keys in the children in the fashion of a search tree
2. All leaves are on the same level
3. All non-leaf nodes except the root have at least $\lceil m / 2 \rceil$ children
4. The root is either a leaf node, or it has from two to *m* children
5. A node contains no more than $m – 1$ keys

The number *m* should always be odd

An Example B-Tree



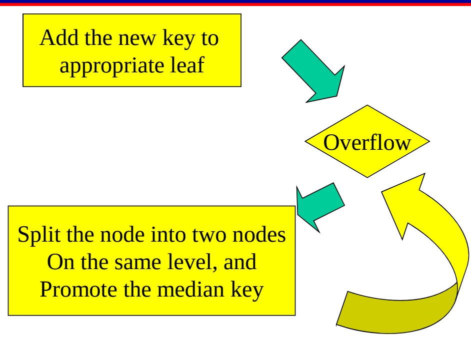
A B-tree of order 5 containing 26 items

Note that all the leaves are at the same level

 U2. 166

Insertion in a B-Tree

1, 12,
8,2,25,
6,14,
28, 17,
7, 52,
16, 48,
68, 3,
26,
29,53,
55, 45



 U2. 167

Constructing a B-tree

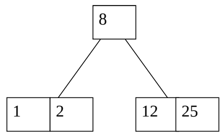
- Suppose we start with an empty B-tree and keys arrive in the following order:
  1 12 8 2 25 6 14 28 17 7 52 16 48 68 3 26 29 53 55 45
- We want to construct a B-tree of order 5
- The first four items go into the root:

1	2	8	12

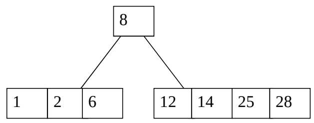
- To put the fifth item in the root would violate condition 5
- Therefore, when 25 arrives, pick the middle key to make a new root

 U2. 168

Constructing a B-tree (contd.)



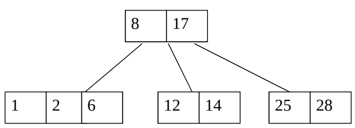
6, 14, 28 get added to the leaf nodes:



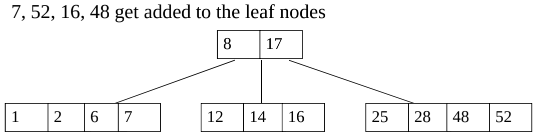
 U2. 169

Constructing a B-tree (contd.)

Adding 17 to the right leaf node would over-fill it, so we take the middle key, promote it (to the root) and split the leaf



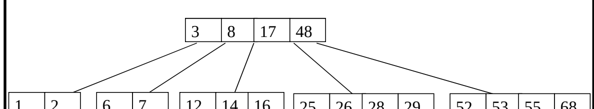
7, 52, 16, 48 get added to the leaf nodes



 U2. 170

Constructing a B-tree (contd.)

- Adding 68 causes us to split the right most leaf, promoting 48 to the root
- Adding 3 causes us to split the left most leaf, promoting 3 to the root; 26, 29, 53, 55 then go into the leaves

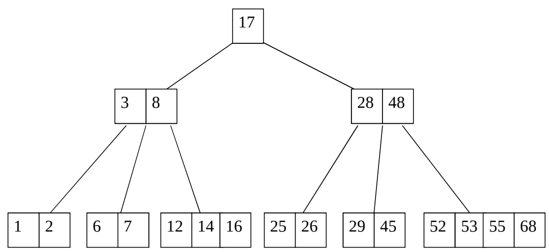


- Adding 45 causes a split of | 25 | 26 | 28 | 29 |

  And promoting 28 to the root then causes the root to split

 U2. 171

Constructing a B-tree (contd.)



 U2. 172

Inserting into a B-Tree

- Attempt to insert the new key into a leaf
- If this would result in that leaf becoming too big
  - split the leaf into two,
  - promoting the middle key to the leaf's parent
- If this would result in the parent becoming too big,
  - split the parent into two
  - promoting the middle key
- This strategy might have to be repeated all the way to the top
- If necessary, the root is split in two and the middle key is promoted to a new root, making the tree one level higher

 U2. 173

Algorithm Insertion in a B- Tree

Insert newEntry in the appropriate Leaf
currentNode = leaf
While(currentNode overflow)
  split the currentNode into two nodes on the same level;
  Promote median key up to the parent of currentNode
  currentNode := Parent of currentNode

 U2. 174

Exercise in Inserting a B-Tree

Insert the following keys to a 5-way B-tree:
3, 7, 9, 23, 45, 1, 5, 14, 25, 24, 13, 11, 8, 19, 4, 31, 35, 56

 U2. 175

Removal from a B-tree

During insertion, the key always goes *into* a *leaf*.
For deletion we wish to remove *from* a leaf.
There are three possible ways we can do this:

1. If the key is already in a leaf node
   - removing not leads to underflow condition
     - simply remove the key to be deleted.
2. If the key is *not* in a leaf
   - delete the key and promote the predecessor or successor key to the non-leaf deleted key's position.

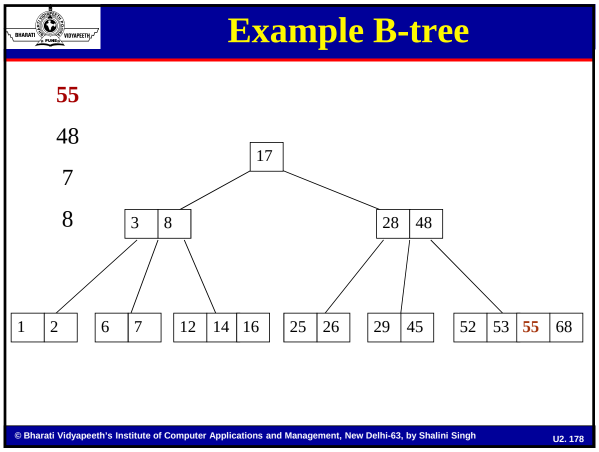
 U2. 176

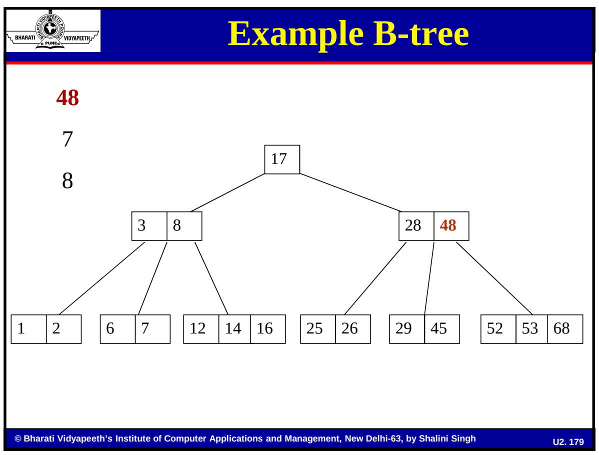
Removal from a B-tree (2)

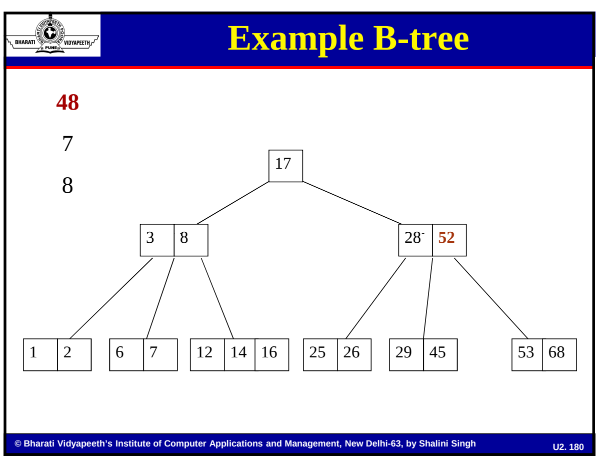
3. If (1) or (2) lead to a leaf node in underflow condition

- **look at the siblings immediately adjacent to the leaf in question**
- if one of them has more than the min' number of keys
  - ✓ promote one of its keys to the parent and take the parent key into our lacking leaf
- if neither of them has more than the min' number of keys
  - ✓ the lacking leaf and one of its neighbours can be combined with their shared parent (the opposite of promoting a key)
  - ✓ the new leaf will have the correct number of keys
  - ✓ if this step leave the parent with too few keys
    - repeat the process up to the root itself, if required

 U2. 177


Example B-tree
55
48
7
8
17
3 8
28 48
1 2
6 7
12 14 16
25 26
29 45
52 53 55 68
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh
U2. 178

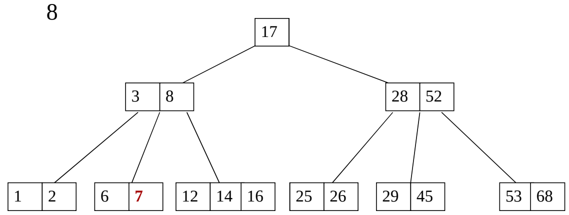

Example B-tree
48
7
8
17
3 8
28 48
1 2
6 7
12 14 16
25 26
29 45
52 53 68
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh
U2. 179


Example B-tree
48
7
8
17
3 8
28 52
1 2
6 7
12 14 16
25 26
29 45
53 68
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh
U2. 180

Example B-tree

7

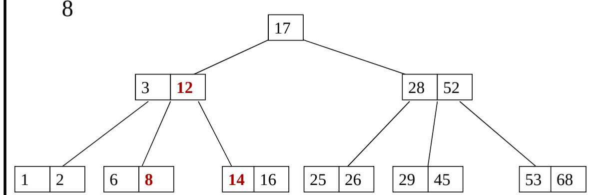
8



Example B-tree

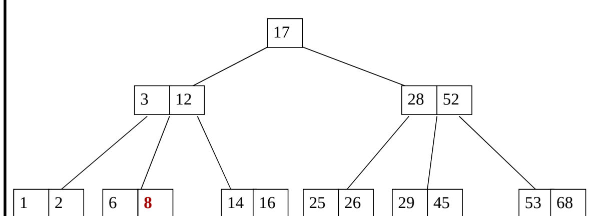
7

8



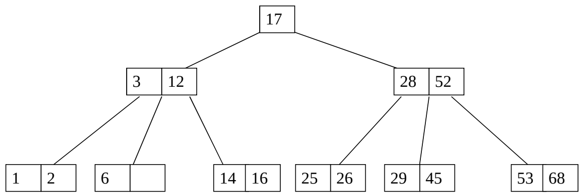
Example B-tree

8



Example B-tree

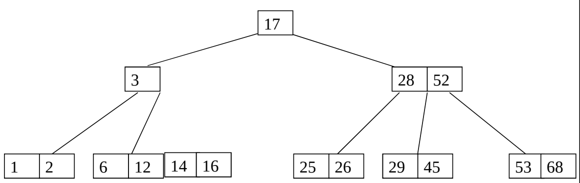
8



 U2. 184

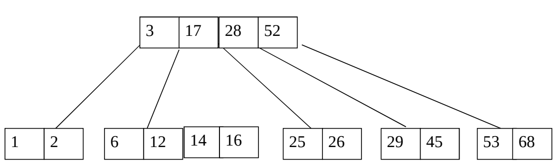
Example B-tree

8



 U2. 185

Example B-tree



 U2. 186

Algorithm Deletion from a B Tree

If (the entry to remove is not in a leaf)
  swap it with its successor
  currentNode = Leaf
While(currentNode underflow)
  if (Immediate sibling have more than min Key)
    redistribute entries from an immediate sibling into currentNode via the parent Node;
  else
    merge currentNode with a sibling and one entry from Parent;
  currentNode = Parent of currentNode;

Exercise in Removal from a B-Tree

Given 5-way B-tree created by these data (last exercise):

3, 7, 9, 23, 45, 1, 5, 14, 25, 24, 13, 11, 8, 19, 4, 31, 35, 56

Add these further keys: 2, 6,12

Delete these keys: 4, 5, 7, 3, 14

Motivation for B-Trees

- So far we have assumed that we can store an entire data structure in main memory
- What if we have so much data that it doesn't fit?
- We will have to use disk storage but when this happens our time complexity fails
- The problem is that Big-Oh analysis **assumes that all operations take roughly equal time**
- **This is not the case when disk access is involved**

Motivation Contd...

- Assume that a disk spins at 3600 RPM
  - In 1 minute it makes 3600 revolutions,
  - One revolution occurs in 1/60 of a second, or 16.7ms
  - On an average a disk access (half way round the disk) will take 8ms
- Comparing with CPU instructions:
  - 120 disk accesses a second ⇔ 10^6 instructions
  - In other words, one disk access takes about the same time as 10,000 instructions
- It is worth executing lots of instructions to avoid a disk access!!!

Motivation Contd...

- Assume that we use an AVL tree to create a database having a record count of the order of 10^6
- We still end up with a **very** deep tree with lots of different disk accesses;
  - $\log_2$ 1,000,000 is about 20,
  - so this takes about 0.2 seconds (if there is only one user of the program)
- We know we can't improve on the log n for a binary tree
- **Solution –**
  - Use more branches and thus *lesser height*!
  - As branching increases, depth decreases

Analysis of B-Trees

The maximum number of items in a B-tree of order m and height h:

root	$m - 1$
level 1	$m(m - 1)$
level 2	$m^2(m - 1)$
. . .	
level h	$m^h(m - 1)$

So, the total number of items is

$$(1 + m + m^2 + m^3 + \ldots + m^h)(m - 1) =$$

$$[(m^{h+1} - 1)/ (m - 1)]\ (m - 1) = \mathbf{m^{h+1} - 1}$$

When $m = 5$ and $h = 2$ this gives $5^3 - 1 = 124$

Analysis of B-Trees

The maximum number of search for n items is: F = n+1

in a B-tree of order *m* :

Level 1	=	1 node
Level 2	=	2 node
Level 3	=	$2\ (m/2 + 1)$
Level 4	=	$2\ (m/2 + 1)^2$
Level (h+1)	=	$2\ (m/2 + 1)^{h-1}$
$2\ (m/2 + 1)^{h-1}$ <=	n+1	
$(m/2 + 1)^{h-1}$	<=	½(n+1)
h-1	<=	$\log_{(m/2+1)}$ ½(n+1)
h	<=	1+ $\log_{(m/2+1)}$ ½(n+1)

Analysis of B-Trees

in a B-tree of order *m =200 and n= 1000000* :

h	<=	1+ $\log_{(m/2+1)}$ ½(n+1)
h	<=	1+ $\log_{(101+1)}$ ½(*1000001*)
h	<= 3.85	

Reasons for using B-Trees

- When searching tables held on disc, the cost of each disc transfer is high but doesn't depend much on the amount of data transferred, especially if consecutive items are transferred
  - If we use a B-tree of order 101, say, we can transfer each node in one disc read operation
  - A B-tree of order 101 and height 3 can hold $101^4 - 1$ items (approximately 100 million) and any item can be accessed with 3 disc reads (assuming we hold the root in memory)

Contd...

If we take m = 3, we get a 2-3 tree, in which non-leaf nodes have two or three children (i.e., one or two keys)

- B-Trees are always balanced
- (since the leaves are all at the same level),
- so 2-3 trees make a good type of balanced tree

Comparing Trees

- Binary trees
  - Can become *unbalanced* and *lose* their good time complexity (big O)
  - AVL trees are strict binary trees that *overcome the balance problem*
  - Heaps remain balanced but only *prioritise* (not order) the keys
- Multi-way trees
  - B-Trees can be *m*-way, they can have any (odd) number of children
  - One B-Tree, the 2-3 (or 3-way) B-Tree, *approximates* a permanently balanced binary tree, exchanging the AVL tree's balancing operations for insertion and (more complex) deletion operations

B* Trees

- Requires non-root nodes to be at least 2/3 full instead of 1/2.
- To maintain this, instead of immediately splitting up a node when it gets full, its keys are shared with the node next to it.
- When both are full, then the two of them are split into three.

What we Learned

- ✓ B-Tree
- ✓ Motivations
- ✓ Basic Operations
- ✓ B* Tree

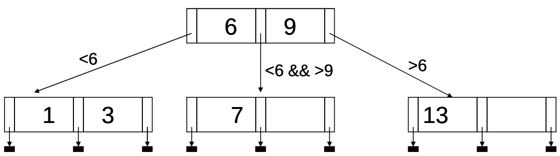
B+ Trees

Objectives

- B+ Tree
- Problem with B Tree
- Solution as B+ Tree
- Basic Operations

Problem With B Trees

Accessing keys from B-tree in sorted order **Requires Backtracking**



Solution: B+ Trees

B+ Trees:

- Facilitate Sequential Operations
- String all leaf nodes together
- Replicate **keys** from non-leaf nodes to make sure each key appears at leaf level.

Properties of B-trees

B Trees

- Are multi-way trees i.e. each node contains a set of keys and pointers.
- Contain only data pages.
- Are dynamic i.e., the height of the tree grows and contracts as records are added and deleted.

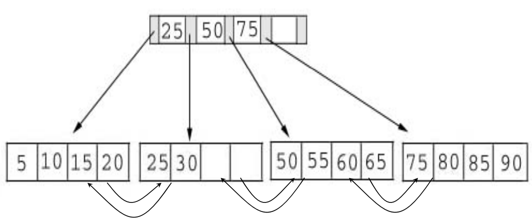
B+ Trees

- Combines features of ISAM and B Trees.
- It contains **index pages** and **data pages**.
- The data pages always appear as leaf nodes in the tree.
- The root node and intermediate nodes are always index pages.
- B+ trees grow and contract like their B Tree counterparts
- The **index pages** are constructed through the process of inserting and deleting records and the contents and the number of index pages reflects the growth and shrinkage in height.

 U2. 205

An Example B+ Tree

Order: 5



Data Pages

 U2. 206

B+ Tree

- The key value determines a record's placement in a B+ tree.
- The leaf pages are maintained in sequential order
- AND a doubly linked list connects each leaf page with its sibling page(s).
- ***This doubly linked list speeds data movement as the pages grow and contract.***

 U2. 207

B+ Tree: Adding Records

Cases to consider while adding records:

	Leaf Page	Index Page
Case I	NOT FULL	NOT FULL
Case II	NOT FULL	FULL
Case III	FULL	NOT FULL
Case IV	FULL	FULL

B+ Tree: Adding Records

Case I & II:

- Place the record in sorted position in the appropriate leaf page

Case III:

1. Split the leaf page
2. Place Middle Key in the index page in sorted order.
3. Left leaf page contains records with keys below the middle key.
4. Right leaf page contains records with keys equal to or greater than the middle key.

B+ Tree: Adding Records

Case IV:

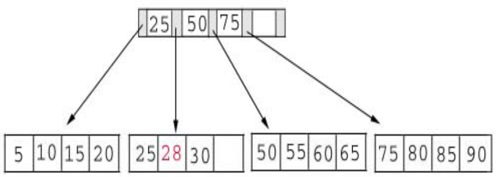
1. Split the leaf page.
2. Records with keys < middle key go to the left leaf page.
3. Records with keys >= middle key go to the right leaf page.

4. Split the index page.
5. Keys < middle key go to the left index page.
6. Keys > middle key go to the right index page.
7. The middle key goes to the next (higher level) index.

IF the next level index page is full, continue splitting the index pages.

Insert Algorithm: Case I

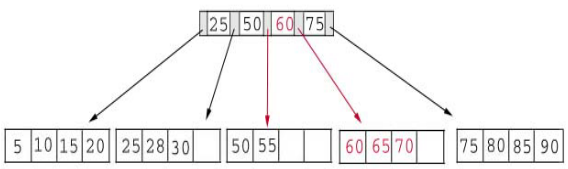
- Inserting a record into a leaf page that is not full
- E.g. insert a record with a key value of 28 into the B+ tree



 U2. 211

Insert Algorithm: Case III

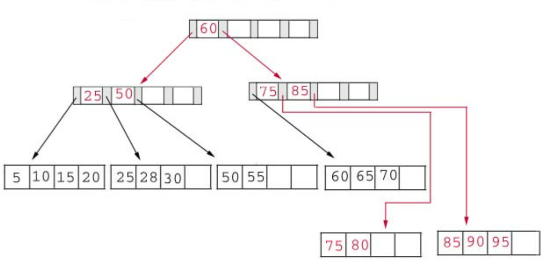
- **Adding a record when the leaf page is full but the index page is not**
- **E.g. insert a record with a key value of 70**



 U2. 212

Insert Algorithm: Case IV

- Both the leaf page and the index page are full
- E.g. Add a record containing a key value of 95



 U2. 213

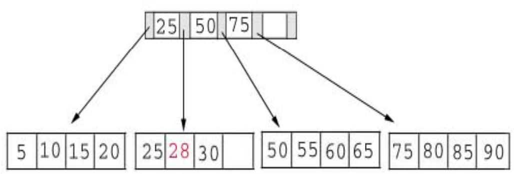
Rotation

- B+ trees can incorporate rotation to reduce the number of page splits.
- A rotation occurs when a leaf page is full, but one of its sibling pages is not full.
- Rather than splitting the leaf page, we move a record to its sibling, adjusting the indices as necessary.

 U2. 214

Rotation Example

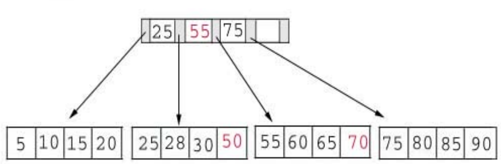
Before addition of record with key 70



 U2. 215

Rotation Example

- Using rotation we shift the record with the lowest key to its sibling.
- Since this key appeared in the index page we also modify the index page.



 U2. 216

Deleting Keys from a B+ tree

- Like Insertion we must consider three scenarios when we delete a record from a B+ tree.
- Each scenario causes a different action in the delete algorithm.

	Leaf Page	Index Page
Case I	Not Below Fill Factor	Not Below Fill Factor
Case II	Not Below Fill Factor	Below Fill Factor
Case III	Below Fill Factor	Not Below Fill Factor
Case IV	Below Fill Factor	Below Fill Factor

Deletion

Case I & II:

- Delete the record from the leaf page. Arrange keys in ascending order to fill void.
- If the key of the deleted record appears in the index page, use the next key to replace it.

Case III:

- Combine the leaf page and its sibling / Shift data from sibling.
- Change the index page to reflect the change.

Deletion

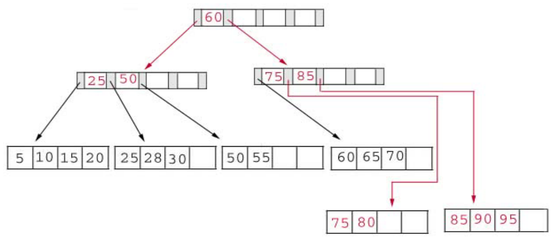
Case IV

1. Combine the leaf page and its sibling.
2. Adjust the index page to reflect the change.
3. Combine the index page with its sibling.

Continue combining index pages until you reach a page with the correct fill factor or you reach the root page.

B+ tree: Deletion Illustrations

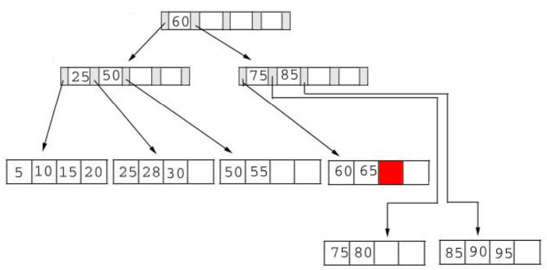
Consider the B+ tree after we added 95 as a key



 U2. 220

Delete - Case I

Delete 70 from the B+ Tree



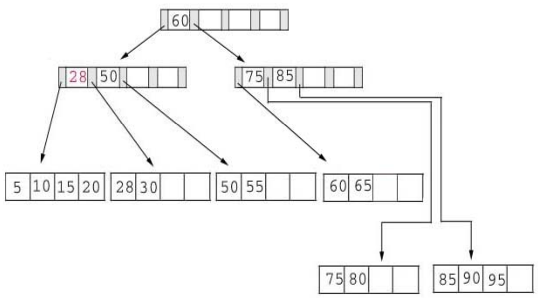
 U2. 221

Delete – Case I B

- Delete the record containing 25 from the B+ tree.
- This record is found in the leaf node containing 25, 28, and 30.
- The fill factor will be 50% after the deletion; however, 25 appears in the index page.
- Thus, when we delete 25 we must replace it with 28 in the index page.

 U2. 222

Delete – Case I B

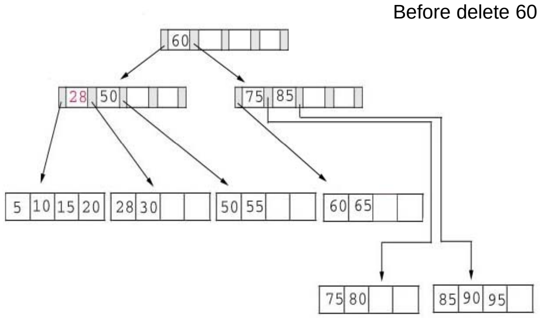


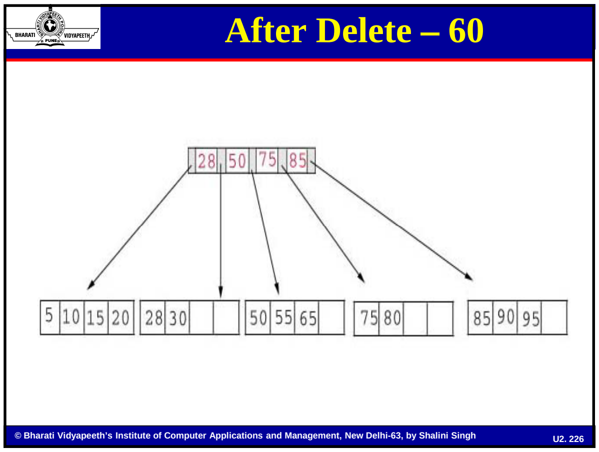
Delete – Case IV

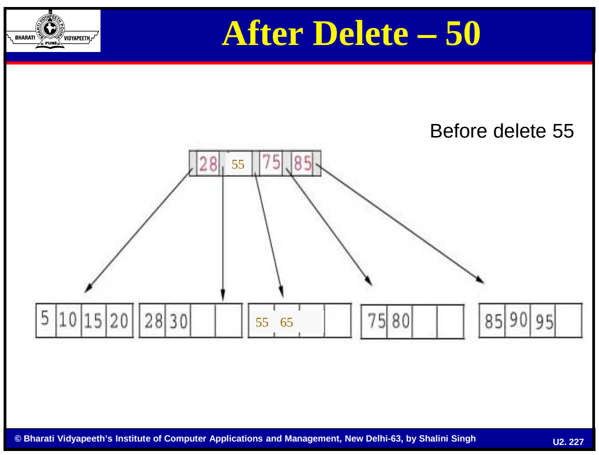
- Delete 60 from the B+ tree. Points to consider:
- The leaf page containing 60 (60 65) will be below the fill factor after the deletion. Thus, we must combine leaf pages.
- With recombined pages, we must readjust the index pages to reflect the change.

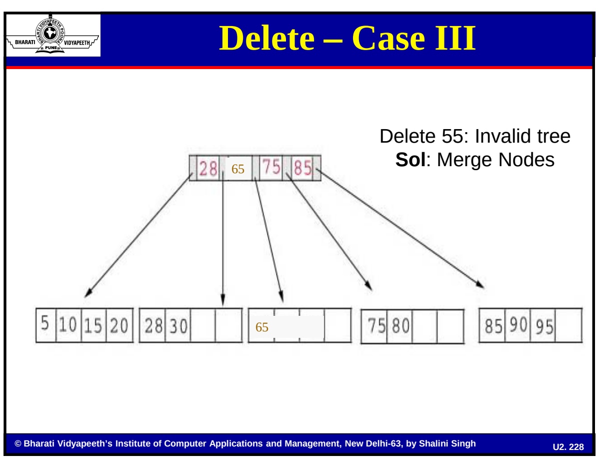
Delete : Case IV

Before delete 60




After Delete – 60
28 50 75 85
5 10 15 20
28 30
50 55 65
75 80
85 90 95
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh
U2. 226


After Delete – 50
Before delete 55
28 55 75 85
5 10 15 20
28 30
55 65
75 80
85 90 95
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh
U2. 227


Delete – Case III
Delete 55: Invalid tree
Sol: Merge Nodes
28 65 75 85
5 10 15 20
28 30
65
75 80
85 90 95
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh
U2. 228

What we Learned

- ✓ B+ Tree
- ✓ Problem with B Tree
- ✓ Solution as B+ Tree
- ✓ Basic Operations

Review Questions (Objective)

1. How many different trees are possible with 10 nodes ?
2. How many null branches are there in a binary tree with 20 nodes?
3. There are 8, 15, 13, 14 nodes in 4 different trees. Which of them could have formed a full binary tree?
4. In an AVL tree, at what condition the balancing is to be done?
5. Define full tree and complete tree.

Review Questions (Objective)

6. What is a binary heap?
7. Where is the biggest data value found in a Max-heap?
8. How can you get sorted data from a BST?
9. Differentiate Index Pages and Data pages in B+ Tree.
10. Which traversal order would you prefer to:
    1. Clone a BST?
    2. Destroy a BST

    Why?

Review Questions (Short Type)

1. What is binary tree. Discuss its properties. Give algorithm to traverse the tree in
   - inorder
   - preorder
   - post order
2. How is a binary tree different from a binary search tree?
3. Write a C function to insert an element into an AVL tree.
4. Define Threaded Binary tree. Write an algorithm for preorder traversal of threaded binary tree without a stack.
5. Define Binary trees. How it can be represented in the memory?

 U2. 232

Review Questions (Short Type)

6. What are the ways of traversing a binary tree? Describe them.
7. Give algorithm to insert a value in a Binary Search Tree.
8. Differentiate between balanced trees and AVL trees with example.
9. What is the maximum total number of nodes in a tree that has N levels? Note that the root is level (zero).
10. Define Game Tree. Write the significance of internal and external nodes of game tree.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U2. 233

Review Questions (Short Type)

11. Draw an expression tree for the expression : A * B - (C + D) * (P / Q).
12. Draw the binary tree with threads to indicate the post order traversal for the expression A - B + C * E/F.
13. Draw the B-tree of order 3 created by inserting the following data arriving in sequence - 92 24 6 7 11 8 22 4 5 16 19 20 78.
14. Draw the B-tree when the data values are deleted in the same order.
15. Draw a B+ tree for the problem sequence given above.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U2. 234

Review Questions (Long Type)

1. A binary tree T has 9 nodes .The inorder and preorder traversals of T yield the following sequence of nodes
   - Inorder : E A C K F H D B G
   - Preorder : F A E K C D H G B

   Draw the tree T

- Consider the algebraic expression E=(2x+y)(5a-b)^3. Draw the Tree T which correspond to expression E.

- Draw all the non similar trees T where:
  - T is a binary tree with 3 nodes
  - T is a 2-tree with 4 external nodes.

4. Write the algorithms of searching in a Binary Search Trees.

Review Questions (Long Type)

5. Give the algorithm of deletions in the binary search trees. Explain with example. Suppose the following list of letters is inserted in order into an empty binary search tree: J,R,D,G,T,E,M,H,P,A,F,Q

   a) Find the final tree T

   b) Find inorder traversal of T

6. What do u mean by tree traversal? Explain the different tree traversals giving suitable examples.

7. Discuss a generalized case of AVL imbalance that requires a **Double Rotation** in any case to restore balance. Also write the C Functions that perform the different rotations.

Review Questions (Long Type)

8. Write a procedure which deletes all the terminal nodes from a binary tree.

9. Write short notes on
   8. Complete binary tree
   9. Weight of a tree
   10. Binary search tree
   11. Heap

10. What do you mean by height balanced tree? How an height balanced tree is different from a binary search tree? What do you mean by rebalancing of height balanced tree. Explain with example.

References

- E. Horowitz and S. Sahani, “Fundamentals of Data Structures in C”, 2nd Edition, Universities Press, 2008.
- Mark Allen Weiss, “Data Structures and Algorithm Analysis in C”, 2nd Edition Addison-Wesley, 1997.
- Schaum’s Outline Series, “Data Structure”, TMH, Special Indian Ed., Seventeenth Reprint, 2009.
- Y. Langsam et. al., “Data Structures using C and C++”, PHI, 1999.
- Mary E. S. Loomes, “Data Management and File Structure”, PHI, 2nd Ed., 1989.
- http://datastructures.itgo.com/graphs/transclosure.htm

 U2. 238