Binary Trees

- General Trees
- Rooted M-ary Trees
- Binary Tree ADT
- Binary Tree Representations
  - Array Representation
  - Example: Heaps
  - Linked Implementation
  - Linked Example

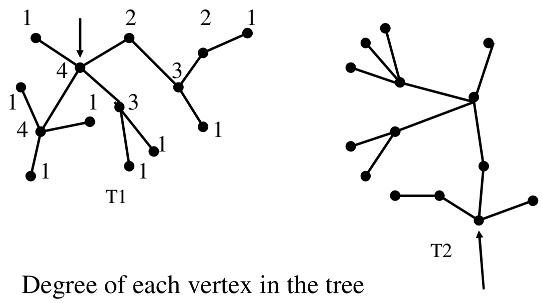
Binary Trees (continued)

- Recursive Trees
- Tree Traversals
- Binary Op-Trees
- Binary Search Trees

General Trees

- <u>Definition: General Trees</u>
- ***A (general) tree $T=(V,E')$ is defined on graph $G=(V,E)$***
- ***Where $E' \subseteq E$.***
- ***If the size of $V = n$, then $|E'| = n-1$.***
- Note: T is **connected** and **acyclic**.

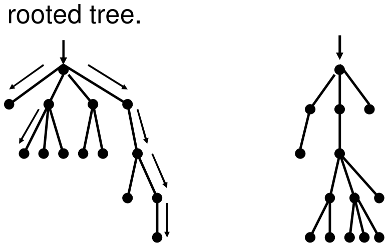
- T1 and T2 are the same tree



Degree of each vertex in the tree

Rooted Trees

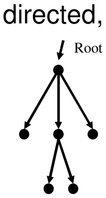
- If we "pick up" the tree by a specific vertex, and "hang it up" , we get a rooted tree.



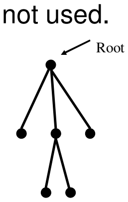
from T1 fromT2

5

- The vertex we "pick up" becomes the root. In a rooted tree, all edges are directed, but the arrows are not used.



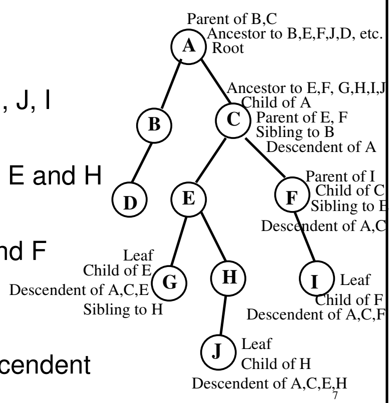
is shown as



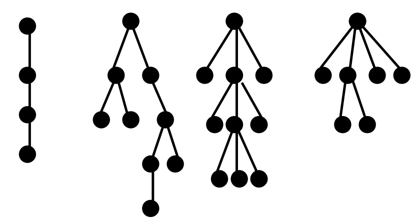
- The root has in-degree zero (Source)
- The leaves have out-degree zero. (Sink)

6

- Root - A
- Leaves - D, G, J, I
- Parent/Child - E and H
- Siblings - E and F
- Ancestor/Descendent
  C and H

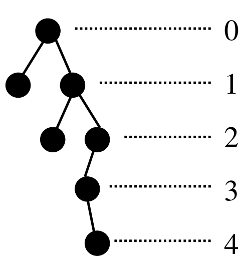


- An m-ary tree is a rooted tree, with m or fewer children per vertex
- (Each vertex has out-degree $\leq$ m)



1- ary unary | 2-ary binary | 3-ary tertiary | 4-ary

- The level of a vertex is it's distance from the root. The root is at level 0
- (Note - some define the root at level 1).

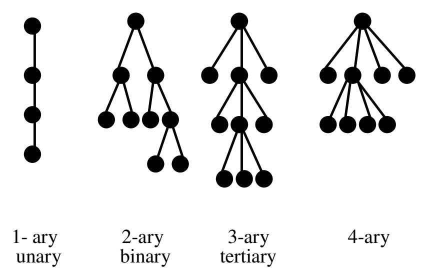


- The <u>height</u> of an m-ary tree is the highest level attained by a leaf

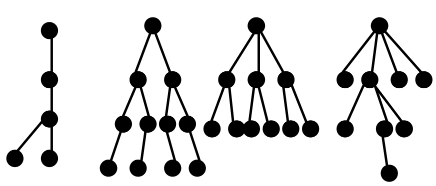
OR

- The <u>height</u> of an m-ary tree is length of the longest path from the root to a leaf.
- Height of tree in last slide is 4 (even though leaves are at different levels).

- A <u>regular m-ary tree</u> has exactly 0 or m children per vertex

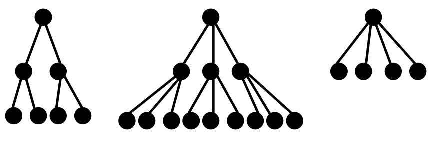


- A <u>regular m-ary tree</u> has exactly 0 or m children per vertex



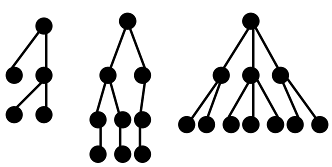
- These trees are not regular

A complete m-ary tree of height k is regular and has all the leaves at same level k.



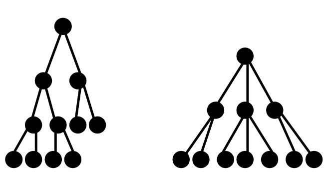
1- ary unary | 2-ary binary | 3-ary tertiary | 4-ary

A complete m-ary tree is regular and has all the leaves at the same level.



These trees are not complete

An almost complete tree has missing nodes in the highest numbered level. Some sources require the missing node to be at the right hand end of the "bottom" level.



These trees are almost complete

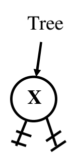
- A complete binary tree of height k has

k	leaves	total nodes
0	1	1
1	2	3
2	4	7
3	8	15
4	16	31
k	2^k	$2^{k+1} - 1$

- exactly 2^k leaves and 2^{k+1} -1 total nodes.
- This is easily proved using induction.

Binary Tree ADT

- Constructor - Initialize empty tree
- MakeTree - Create tree containing 1 value as root with no children
- SetLeft - attach value as left child
- SetRight - attach value as right child
- Display Tree - optional
- TraverseTree - move though tree in orderly fashion
- Search Tree - Look for specified value



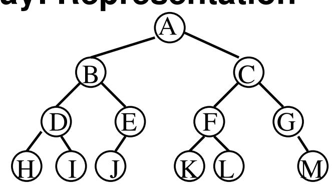
- DeleteNode - Deletes specified node
- Data section:
  - Reference to tree
  - NumNodes //in tree - optional
  - Height //of tree - optional
- Plus actual data in tree

Binary Tree Representation

- Sequential Array Representation
- Linked Representation

Sequential Array: Representation

- Tree is <u>complete</u> or <u>almost complete</u>

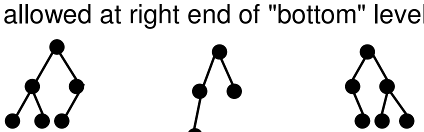


1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	C	D	E	F	G	H	I	J	-	K	L	-	M

- For node at location **i**:
- Parent is at **i/2** .
- Children are at **2i** and **2i+1**.
- What do B, D, F, H, J, K have in common?

<u>Heaps</u>

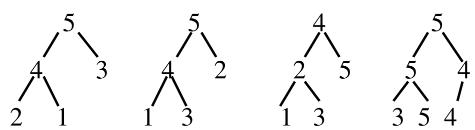
- This representation of trees is often used to represent a special type of binary tree called a <u>heap</u>.
- We use the strictest sense of "almost complete" trees: missing values only allowed at right end of "bottom" level.



Yes Yes No

A **max heap** is defined as a binary tree such that no child is larger than their parent.

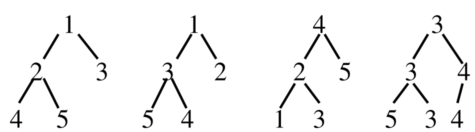
- Note - root contains the largest value.



Yes Yes No Yes

A **min heap** is defined as a binary tree such that no child is smaller than their parent.

- Note - root contains the smallest value.



Yes Yes No Yes

- Heaps are used by operating systems to manage free space.
- Deaps (double ended heaps) and Min-Max heaps are heap variations discussed in Horowitz and Sahni.
- Fibonacci heaps are discussed in Cormen, et. al.

Priority Queues

- Heaps are an alternate way to represent Priority Queues.
- Given Priority Queue as a Max Heap, Insert a Value
- Delete from Priority Queue

Linked Implementation

- Trees are easily and naturally represented using dynamic allocation.
- Use node specified as we did for doubly linked lists.

Left	Data	Right

Linked Representation: Building Tree

Problem: Maintain a list of integers in ascending order

14 15 4 9 7 18 3 5 16 4 20 17 9 14 5

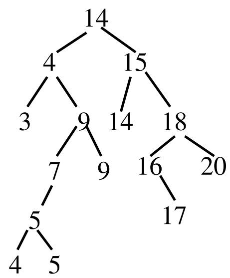
Method:

Store 1 value in list as root

Other values are stored in left subtree if $\leq$ Root

Other values are stored in right subtree if $\geq$ Root

14 15 4 9 7 18 3 5 16 4 20 17 9 14 5

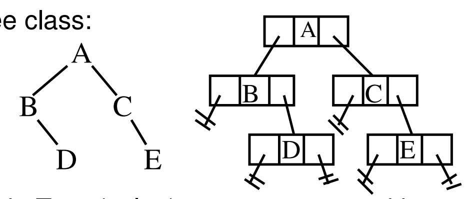


- This method builds a general tree - not complete or regular. It is easily done with Maketree, Setleft, Setright.

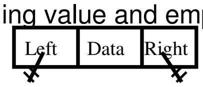
- Notice that everything in the left subtree is smaller the root and everything in the right subtree is larger than the root.

Linked Implementation

- A binary tree can be viewed as a multi-linked list. Methods we will need in are tree class:



- MakeTree (value) - creates a tree with a root containing value and empty children



Linked Implementation

- SetLeft (value, parent) calls MakeTree with value and attaches the resulting tree as the left child of "parent." It is a error if "parent" already has a left child.
- Set Right (value, parent) - Calls MakeTree with value and attaches the resulting tree as the right child of "parent". It is an error if "parent" already has a right child.

Binary Tree Code

A simple Linked implementation

```
class TreeNode {

    DataType Data;   //any appropriate type
    TreeNode Left, Right;
}
```

Note: default constructor is TreeNode(). You could define methods like GetData and SetData if desired.


```
public class TreeClass {
    private TreeNode Tree;    //reference to root
    private TreeNode Here ; //reference to node
                        // in Tree where action occurs
    private TreeNode Parent_of_Here;
            //maintained as a matter of efficiency

public void TreeClass() {            //constructor
    Tree = null;
    Here = null;
    Parent_of_Here = null;
}
```

```
public TreeNode MakeTree(DataType item) {
                                        //constructor
    TreeNode Temp = new TreeNode;
    Temp.Data = item;
    Temp.Left = null;
    Temp.Right = null;
    return Temp;
}

public boolean TreeEmpty() {
        return (Tree == null);
}
```



```
public void SetRight(DataType item) {
                //add right child to existing node P

    TreeNode P = Here;
    if (P == null) "error handling"    //P must exist
    else if (P.Right != null) "error handling"
                        //It is an error for P to have an
        else {                      //existing right child
            TreeNode Temp = MakeTree (item);
            P.Right = Temp;
        }
}
```



```
public void SetLeft(DataType item) {
                    //add left child to existing node P
    TreeNode P = Here;
    if (P == null) "error handling"    //P must exist
    else if (P.Left != null) "error handling"
                        //It is an error for P to have an
    else {                          //existing left child
        TreeNode Temp = MakeTree (item);
        P.Left = Temp;
    }
} //Note: It would be easy to rewrite SetLeft and SetRight to
  // move an existing child down a level if desired.
```

```
public void TreeCopy (TreeNode Tree) {
    //a copy constructor...takes the existing
      tree Tree and makes a copy of it to
      initiate a new Tree - may require
      modifying interface or ADT
    ... exercise for the student...
}

public DataType TreeDelete
    // to be discussed later
}  //end TreeDelete
```

37

```
public void TreeSearch (DataType Item) {
                                  //iterative solution

//Searches tree for value Item.  Result is
   returned in the class variable Here.  Here is
   set to null if the tree is empty or the item is not
   present, otherwise Here is set to point to the
   node containing item

   Here = Tree;       //Start at root of existing tree
   Parent_of_Here = null;
```

38

```
   if !(Here == null) {  //look thru non-empty tree
      while ((Here != null) && (Here.Data != Item)) {
         Parent_of_Here = Here;
         if (Item < Here.Data) Here = Here.Left
         else Here = Here.Right;
      }  //end while
      if (Here == null) Parent_of_Here = null;
                                     //item not in tree
   }  // end if Here != null
}  //end TreeSearch
```

39

```
public void TreeInsert
                (DataType Item, TreeNode Root) {
                                        //recursive solution

   if (Root == null) Root = MakeTree(Item);
                                    //insert at root of Tree
   else if (Item < Root.Data) TreeInsert (Item,Root.Left)
   else TreeInsert (Item, Tree.Right);

}  //end TreeInsert
```



```
public void TreeInsert (DataType Item) {
                                        //iterative solution

   TreeNode Parent;              //trailing pointer
   TreeNode Curr;                        //Current ptr
   TreeNode Temp = MakeTree(Item);
                              //set up node for insertion
   Curr = Tree;      //Start at root of existing tree
```



```
    if (Curr == null) Curr = Temp;
                                  //insert at root of Tree
    else {
        while (Curr != null) {          //move Curr through
            Parent = Curr;        //tree to identify insert pt
            if (Item < Curr.Data) Curr = Curr.Left
            else Curr = Curr.Right);
        }
        if (Item < Parent.Data) Parent.Left  = Temp
        else Parent.Right = Temp;
    }
 }  //end TreeInsert
 } //end TreeClass
```

Suppose a user had an instance T of TreeClass and he wanted to insert Bob as the right child of Alice. Assume the variables **BobsName** and **AlicesName** have been correctly defined. The insertion could take place as follows:

TreeSearch (**AlicesName**); //Set **Here** to Alice

SetRight (**BobsName**); //Will return an error if Alice //has existing right child or if Alice is not in the tree.

The use of the class variable **Here** allows the user to pass around a pointer without having direct knowledge of it.

Recursive Binary Trees

- A tree is a collection of three things
  - a root
  - a left subtree (LST)
  - a right subtree (RST)
- The LST and RST are also trees and may be empty.

Recursive Binary Trees

- The recursive approach is natural. Many tree algorithms are recursive.
- Moving through this tree simply requires moving through or "visiting" each of the three parts. Visiting means performing whatever action is needed for the desired application:
  – writing out the contents
  – performing a calculation
  – updating contents
  – etc..

Recursive Binary Trees

There are six possible combinations.

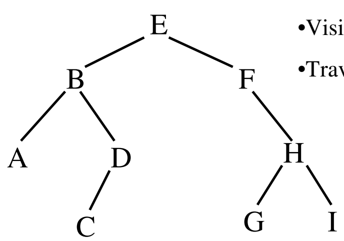
ROOT	LST	RST	~~ROOT~~	~~RST~~	~~LST~~
LST	ROOT	RST	~~RST~~	~~ROOT~~	~~LST~~
LST	RST	ROOT	~~RST~~	~~LST~~	~~ROOT~~

- The left child is “Before” the right child.
- We will always "visit" the LST before the RST so use the three on the left.
  - Preorder traversal
  - Inorder Traversal
  - Postorder traversal

46

Tree Traversal - INORDER

- Traverse LST
- Visit Root
- Traverse RST

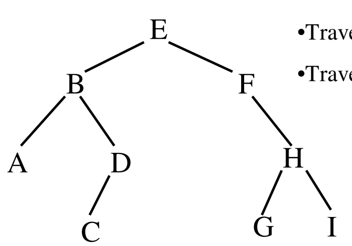


A B C D E F G H I

47

Tree Traversal - PREORDER

- Visit Root
- Traverse LST
- Traverse RST

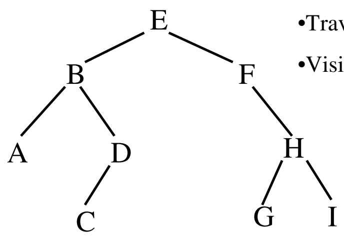


E B A D C F H G I

48

Tree Traversal - POSTORDER

- Traverse LST
- Traverse RST
- Visit Root



A C D B G I H F E

Tree Traversals

```
public void Traverse (TreeNode T) {
        if (T != null) {
                Traverse ( T.left)       //visit left subtree
        ***     "visit" T.data    //e.g. write out contents
                Traverse ( T.right)   //visit right subtree
        }
}
```


Traversing a binary tree

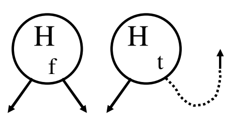
- Recursively
- Interatively
  - –Simulate Recursion / Improve
  - –Parent Pointers
  - –Threaded Trees.

Parent Pointer Example

- This works for any transversal order
- Flexible - you can move about the tree at will.
- Extra space for parent pointer is allocated and maintained in each node.

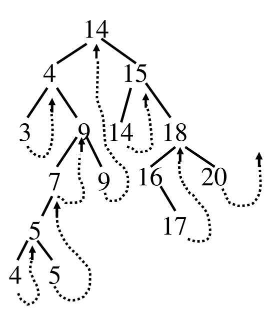
Threaded Trees

- <u>Right-In-Treaded</u> trees
- Only works for inorder transversal
- Uses space allocated & not being used.
- Eliminates some of redundancy of recursive traversals.



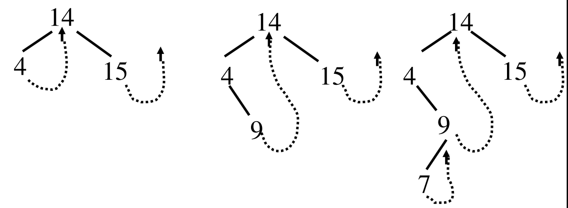
Left	Data	T	Right

Threaded Trees



Still get sorted list with inorder traversal

Threaded Trees - Example



Thread always points to Inorder successor

Modify SetLeft, SetRight, MakeTree


```
public void SetLeft(DataType item) {
                          //add left child to existing node P
    TreeNode P = Here;
    if (P == null) "error handling"        //P must exist
    else if (P.Left != null) "error handling"
                    //P can't have an existing left child
    else {
        TreeNode Temp = MakeTree (item);
        P.Left = Temp;
        Temp.Right = P;           //set thread to parent
        Temp.Rthread = true;
    }
}
```

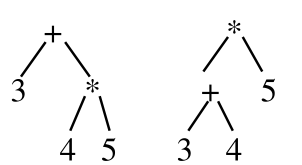
```
public void SetRight(DataType item) {
                        //add right child to existing node P
    TreeNode P = Here;
    if (P == null) "error handling"            //P must exist
    else if (!P.RThread) "error handling"
                        //P can't have an existing right child
      else {
         TreeNode Temp = MakeTree (item);
         Temp.Right = P.Right //copy IOS from parent
         Temp.RThread = true;
         P.Right = Temp;
         P.RThread = false;
       }
}
```


Threaded Trees

- Trees can be right-threaded for preorder and postorder transversals.
- Trees can also be left threaded or both
- Reference: Horowitz and Sahni

Example: Binary Op-trees

Recall:	3+4*5=23	(3+4)*5=35	Infix
	+3*45	34+5*	Postfix
	345*+	*+345	Prefix

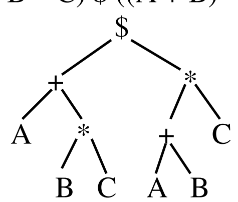


Example: Binary Op-trees

- Binary Op-Trees are heterogeneous binary trees with operands as the leaves and operators as non-leaves.
  - The Operator with least precedence is the root of the tree.
  - Traversing the tree in preorder gives the prefix expression
  - Traversing the tree in postorder gives the postfix expression

Example: Binary Op-trees

(A + B * C) $ ((A + B) * C)



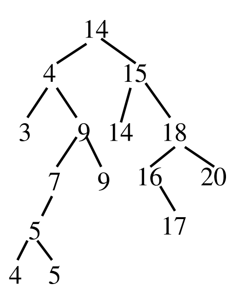
Preorder: $ + A * B C * + A B C

Inorder: A + B * C $ A + B * C

Postorder: A B C * + A B + C * $

Binary Trees: Application

Recall the general tree built earlier:



An inorder transversal gives sorted list because of way tree constructed

Binary Search Trees

- The "Search Tree" characteristic defines that everything in the LST is ≤ the root of the subtree. And everything in the RST is ≥ the root of the subtree.

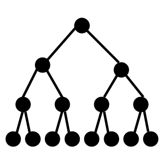
- This is a recursive characteristic.
- If a tree has this characteristic, then an Inorder Traversal will yield the values in monotonically increasing order.

Binary Search Trees

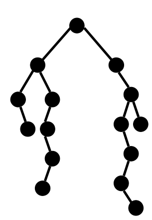
- Previous method for building a tree resulted in a general tree with no special attributes.
- Creating and traversing it is not especially efficient.
- Can we build a tree in such a way that it still has the search tree characteristic, but is more efficient to build and access?

Simple Binary Search Trees

- Such a tree would be short and bushy for the number of nodes contained. It would be "balanced" (an intuitive concept).



versus

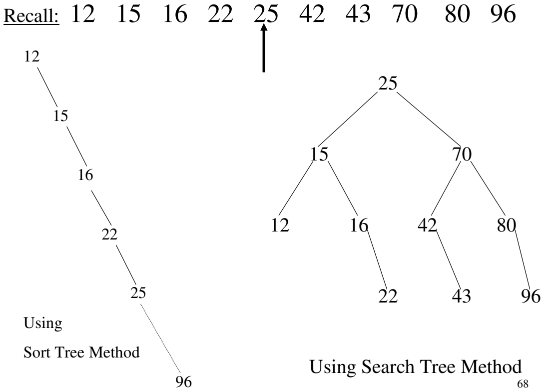


Simple Search Trees

Not a Binary Sort Tree - built differently.

- Start with a sorted file
- Use the middle item as the root.
- The left half becomes the left subtree
- The right half becomes right subtree
- This is recursive.

Recall: 12 15 16 22 25 42 43 70 80 96



Using Sort Tree Method

Using Search Tree Method

Sort Tree vs. Search Tree

Sort Tree	**Search Tree**
Uses random data	Uses sorted data
Root is first item	Root is middle item
Builds general tree	Builds balanced tree
Easy for sorting	Efficient for searching
Do inorder traversal to get sorted list	Do inorder traversal to get sorted list
Can use for searching	Can use for searching

Simple Binary Search Trees

- Casual concept of balance
- Relaxed in maintenance.
- No real effort to retain the balance
- As insertions and deletions are performed, the tree degrades

70

Simple Binary Search Trees

- When performance becomes unacceptable, rebuild tree
- Or do at designated intervals
- Do inorder transversal to get sorted list
- Schedule at time of low system usage.

71

Simple Binary Search Trees

To Insert:

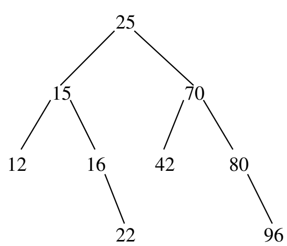
- Use same insertion scheme as Binary Tree Sort
- As insertions are performed, the tree degrades because no effort is made to retain balance
  - less balanced
  - less bushy
  - more general in it's structure
  - more time consuming to use

72

Simple Binary Search Trees

- To delete there are three cases:
  - (1) No children - just delete node
  - (2) 1 child - replace value with child and delete child
  - (3) 2 children - replace with inorder successor and delete IOS
- When performance is poor (or at designated times) the tree is rebuilt

Binary Search Tree: Deletion

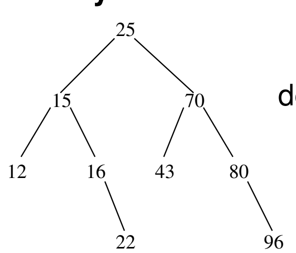


Using Search Tree Method

Tree with 43 deleted

Case I - no children

Binary Search Tree: Deletion

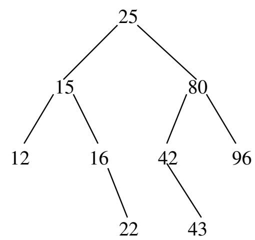


Using Search Tree Method

Tree with 42 deleted - replace with child

Case II - one child

Binary Search Tree: Deletion

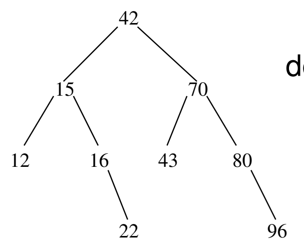


Tree with 70 deleted - replace with 80

Case III - two children

Using Search Tree Method

Binary Search Tree: Deletion



Tree with 25 deleted -replace with 42

Case III - two children

Using Search Tree Method

Simple Binary Search Tree

- This is a simple strategy that gives reasonable results.
- Stay tuned for more complex schemes
  - More work
  - Better results