Part 6.
Trees (1)

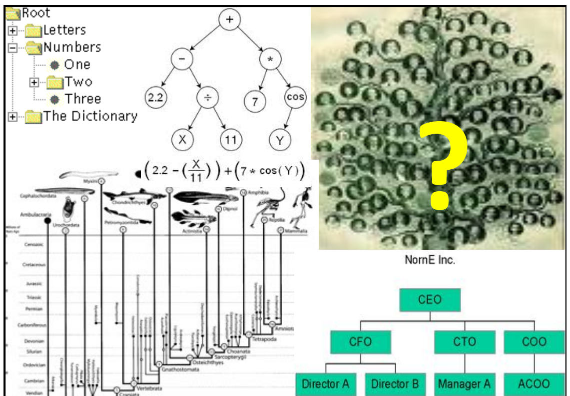
CS 200 Algorithms and Data Structures

1

Outline

- **Terminology**
- Binary Tree
  - Basic operations
  - Traversals of a Binary Tree
  - Representations of a Binary Tree
  - Implementations
- Binary Search Tree
  - Algorithms
  - Implementations
  - Efficiency
  - TreeSort
- General Tree

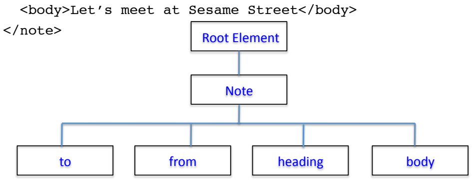
2



3

Data Representation

```
<?xml version="1.0" encoding="ISO-8859-1"?>
 <note>
   <to>Elmo</to>
  <from>Zoe</from>
  <heading>Reminder</heading>
  <body>Let's meet at Sesame Street</body>
</note>
```

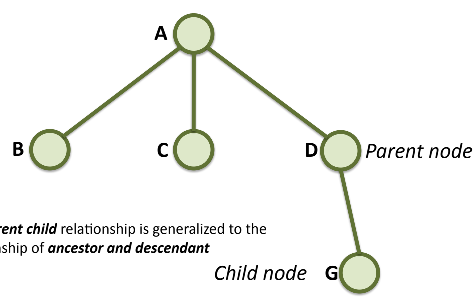


4

- Represent ***hierarchical* relationships**

5

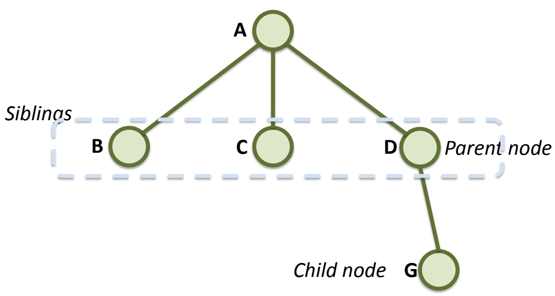
Terminology



The ***parent child*** relationship is generalized to the relationship of ***ancestor and descendant***

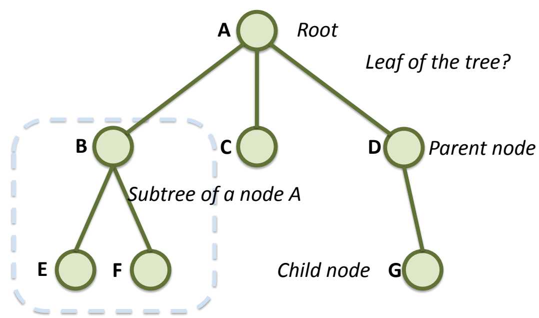
6

Terminology



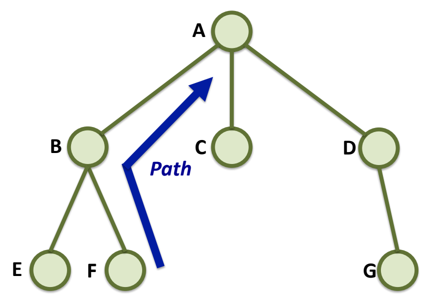
7

Terminology



8

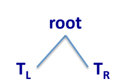
Terminology



9

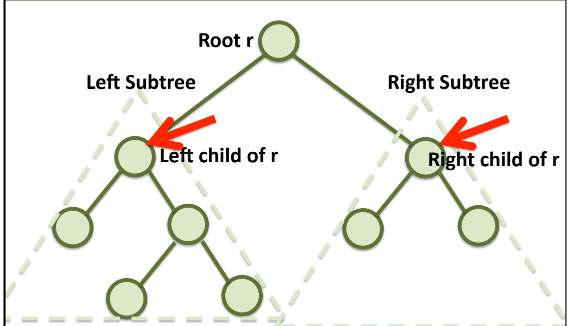
Binary Tree

- Set T of nodes such that either,
  - T is empty, or
  - T is partitioned into three disjoint subsets
    - A single node *r*, ***root***
    - Two possibly empty sets that are binary trees, called ***left*** and ***right subtrees*** of *r*
- Each node in a binary tree has no more than two children.



10

Binary Tree



11

General Tree

- General Tree T is a set of one or more nodes such that T is partitioned into disjoint subsets:
  - A single node *r*, the ***root***
  - Sets that are general trees, called ***subtrees of r***

12

Degree, depth, and Height

- ***Degree***
  - Degree of a node: number of children
  - Degree of a tree: maximum degree of nodes.
- ***Depth***: length of path from node to root.
- ***Height*** of a tree
  - The number of nodes on the longest path from the root to a leaf.

13

Height of a Tree

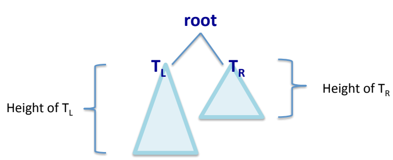
- If T is empty, its height is 0.
- If T is not empty, its height is equal to the maximum level of its nodes.

14

Height of a Binary Tree

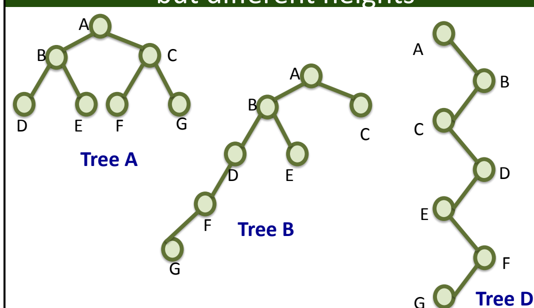
- If T is empty, its height is 0.
- If T is a non empty binary tree,

$$height(T) = 1 + max\{height(TL), height(TR)\}$$



15

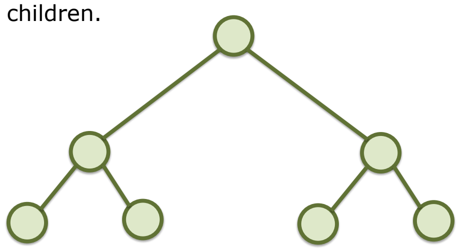
Binary trees with same nodes but different heights



16

Full Binary Tree

- A full binary tree is a tree in which every node other than the leaves has two children.



17

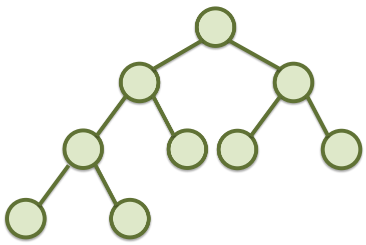
Definition of Full Binary Tree

- If T is empty, T is a full binary tree of height 0.
- If T is not empty and has height h > 0, T is a full binary tree if its root's subtrees are both full binary trees of height h -1.

18

Complete Binary Tree

- Is a binary tree in which every level, *except possibly the last*, is completely filled, and all nodes are as far left as possible.



19

Formal Definition of a Complete Binary Tree

- A binary tree T of height h is complete if,
  - All nodes at level h-2 and above have two children each, and
  - When a node at level h-1 has children, all nodes to its left at the same level have two children each, and
  - When a node at level h-1 has one child, it is a left child.
- Full binary tree is complete.

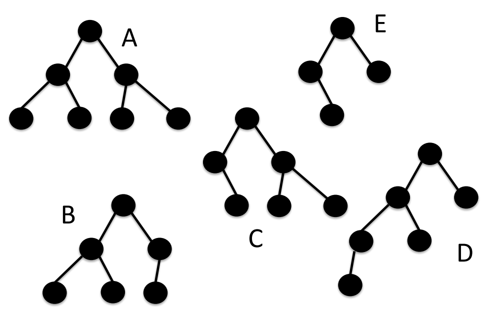
20

Balanced Binary Tree

- A binary tree is ***height balanced***, or ***balanced***, if the height of any node's right subtree differs from the height of the node's left subtree by **no more than 1**
  - The height of the two subtrees of every node never differ by more than 1

21

Full? Complete? Balanced? Binary tree?



22

Outline

- Terminology
- **Binary Tree**
  - Basic operations
  - Traversals of a Binary Tree
  - Representations of a Binary Tree
  - Implementations
- Binary Search Tree
  - Algorithms
  - Implementations
  - Efficiency
  - TreeSort
- General Tree

23

Operations of the Binary Tree

- Add and remove node and subtrees
- Retrieve and set the data in the root
- Determine whether the tree is empty

24

General operations

```
Root
Left subtree
Right subtree

creteBinaryTree()
makeEmpty()
isEmpty()
getRootItem()
setRootItem()
attachLeft()
attachRight()
attachLeftSubtree()
attachRightSubtree()
detachLeftSubtree()
detachRightSubtree()
getLeftSubtree()
getRightSubtree()
```

25

Example

```
tree1.setRootItem("F")
tree1.attachLeft("G")

tree2.setRootItem("D")
tree2.attachLeftSubtree(tree1)

tree3.setRootItem("B")
tree3.attachLeftSubtree(tree2)
tree3.attachRight("E")

tree4.setRootItem("C")

binTree.createBinaryTree("A", tree3, tree4)
```

26

Traversal Algorithms

- The traversal of a tree is the process of "visiting" every node of the tree
  - Display a portion of the data in the node.
  - Process the data in the node
- Because a tree is not linear, there are many ways that this can be done.

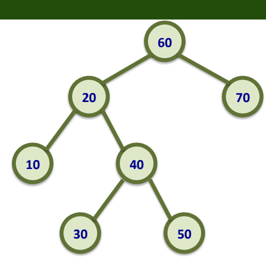
27

Breadth-first traversal

- Breadth-first processes the tree **level by level** starting at the root and handling all the nodes at a particular level from **left to right**.

28

Breadth-first traversal



60 – 20 – 70 – 10 – 40 – 30 – 50

29

Depth-first traversals

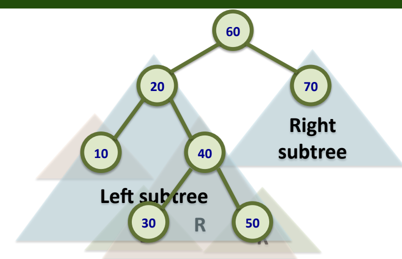
- Three choices of when to visit the root *r*.
  - These methods are recursive methods for tree traversal.
  1. **Before** it traverses both of *r*'s subtrees
  2. After it has traversed *r*'s **left** subtree (before it traverses *r*'s right subtree)
  3. After it has traversed **both** of *r*'s subtrees
- **Preorder, inorder,** and **postorder**

30

Depth First: Preorder traversal

- ***Preorder traversal*** processes the information at the root, followed by the entire left subtree and concluding with the entire right subtree.

Depth First: Preorder traversal

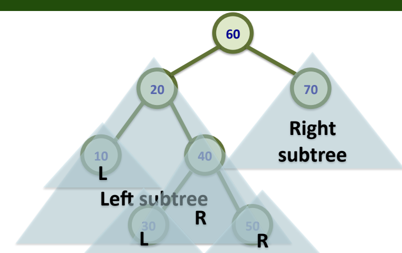


60 – 20 – 10 – 40 – 30 – 50 – 70

Depth First: Inorder traversal

- ***Inorder traversal*** processes all the information in the left subtree before processing the root.
- It finishes by processing all the information in the right subtree.

Depth First: Inorder traversal

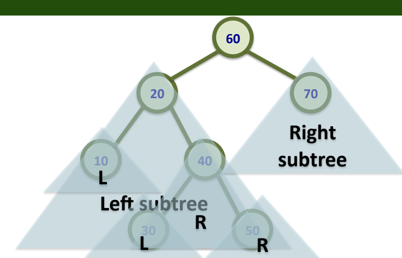


10 – 20 – 30 – 40 – 50 – 60 – 70

Depth First: Postorder traversal

- **Postorder traversal** processes the left subtree, then the right subtree and finishes by processing the root.

Depth First: Postorder traversal



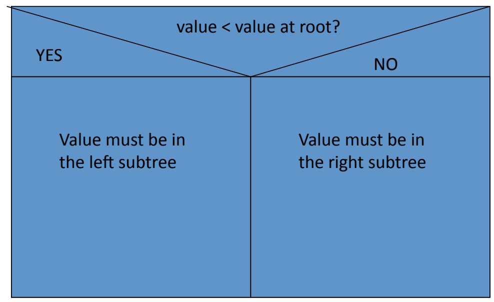
10 – 30 – 50 – 40 – 20 – 70 – 60

Use case: postorder traversal

- When destroying a tree, we can't delete the root node until we have destroyed the left and right subtree
  - So a postorder traversal makes sense.

37

Use case: Preorder/Inorder traversal (Creating and Browsing of Binary Sorting)



38

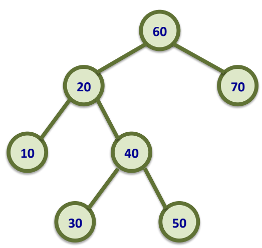
Example of Binary sorting

Create Tree

60	20	10	40	70	50	30

Traversal Tree

10	20	30	40	50	60	70



39

Preorder algorithm

```
preorder (in binTree:BinaryTree)
  if (binTree is not empty){
     display the data in the root of binTree
     preorder(Left subtree of binTree's root)
     preorder(Right subtree of binTree's root)
  }
```

40

Inorder algorithm

```
inorder (in binTree:BinaryTree)
  if (binTree is not empty){
     inorder(Left subtree of binTree's root)
     display the data in the root of binTree
     inorder(Right subtree of binTree's root)
  }
```

41

Postorder algorithm

```
postorder (in binTree:BinaryTree)
  if (binTree is not empty){
     postorder(Left subtree of binTree's root)
     postorder(Right subtree of binTree's root)
     display the data in the root of binTree
  }
```

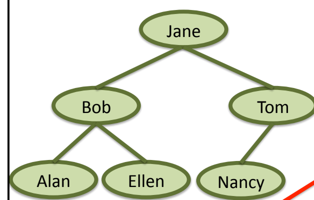
42

Analysis of Tree Traversal

- n visits occur for a tree of n nodes.
  - $O(n)$

43

An array based representation (1/2)



A binary tree of names

Free list: Array – based Linked List

root 0

free 3

index	item	leftChild	rightChild
0	Jane	1	2
1	Bob	3	4
2	Tom	5	-1
3	Alan	-1	-1
4	Ellen	-1	-1
5	Nancy	-1	-1
6	?	-1	7
7	?	-1	8
8	?	-1	9
	.	.	.
	.	.	.
	.	.	.
	.	.	.

44

An array based representation (2/2)

0	1	2	3	4	5

```
public class TreeNode<T>{
    private T item;
    private int leftChild;
    private int rightChild;
    …
    public TreeNode(){
    }
    public int getItem(){
       return item;
    }
    public int getLeftChild(){
       return leftChild;
    }
    public int getRightChild(){
        return rightChild;
    } … setters
}
```

45

An array based representation (2/2)

```
public class BinaryTreeArrayBased<T> {
    protected final int MAX_NODES = 100;
    protected ArrayList<TreeNode<T>> tree;
    protected int root;
    protected int free; //index of next unused array location
    …

public BinaryTreeArrayBased<T> () {
    tree  = new ArrayList<TreeNode<T>>()’
}

public creatTree(TreeNode<T> _root){
    root = 0;
    tree.set(0, _root);
    free++;
}
```

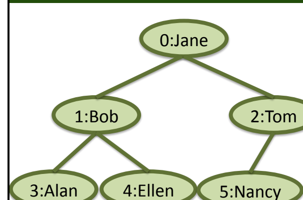
46

An array based representation (2/2)

```
public TreeNode<T> getRootItem(){
    return tree.get(root);
}

public TreeNode<T> getRight(){
    return tree.get(root.getRightChild());
}

public TreeNode<T> getLeft(){
    return tree.get(root.getLeftChild());
}

public void makeEmpty(){
    how?
}

More methods..
```

47

Complete Binary Tree



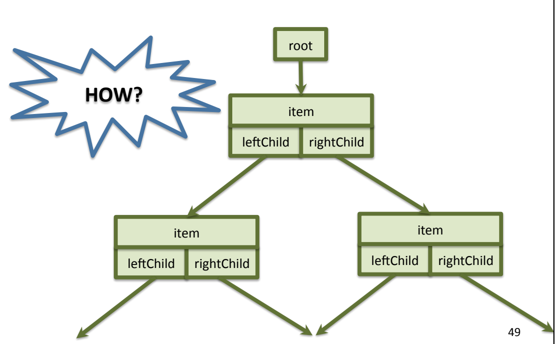
Level-by-level numbering of a complete binary tree

index	item
0	Jane
1	Bob
2	Tom
3	Alan
4	Ellen
5	Karen
6	
7	

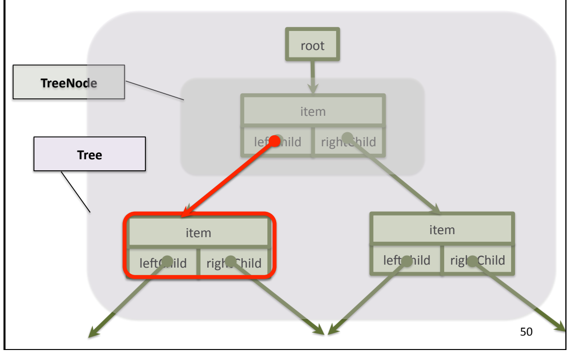
If the binary tree is complete and remains complete

A memory-efficient array-based implementation can be used

A reference-based representation



A reference-based representation



A reference based representation (1/6)

Step 1. TreeNode

```
public TreeNode<T> {
    T item;
    TreeNode<T> leftChild;
    TreeNode<T> rightChild;

    public TreeNode(T newItem){
      item = newItem;
     leftChild = null;
     rightChild = null;
    }

    public TreeNode(T newItem, TreeNode<T> left, TreeNode<T>
right){
        item = newItem;
        leftChild = left;
        rightChild = right;
    }
}
```

A reference based representation (2/6)

Step 2. Tree (BinaryTree)

```
public class BinaryTree<T> {

    public BinaryTree(){
    }
    public BinaryTree(T rootItem, BinaryTree<T>
leftTree,BinaryTree<T> rightTree){
      root = new TreeNode<T> (rootItem, null, null);
      attachLeftSubtree(leftTree);
      attachRightSubtree(rightTree);
    }

    public void setRootItem(T newItem){
        if(root!=null){
            root.ite = newItem;
        }
        else {
            root = new TreeNode<T>(newItem, null, null);
        }
    }
```

A reference based representation (3/6)

```
public void attachLeft(T newItem){
    if (!isEmpty()&&root.leftChild == null) {
        root.leftChild = new TreeNode<T>(newItem, null, null);
    }
}

public void attachRight(T newItem){
    if (!isEmpty()&&root.leftChild == null) {
        root.rightChild = new TreeNode<T>(newItem, null,
null);
    }
}
```

A reference based representation (4/6)

```
public void attachLeftSubtree(BinaryTree<T> leftTree)
    throws TreeException{
    if (isEmpty()) {
        throw new TreeException("TreeException:Empty tree.");
    }
    else if (root.leftChild != null){
        throw new TreeException("TreeException: cannot
overwrite left subtree.");
    }
    else{
        root.leftChild = leftTree.root;
        leftTree.makeEmpty();
    }
}

Public void attachRightSubtree(…
similar to attachLeftSubtree()
}
```

A reference based representation (5/6)

```
public BinaryTree<T> detachLeftSubtree(BinaryTree<T>
leftTree)
    throws TreeException{
    if (isEmpty()) {
        throw new TreeException("TreeException:Empty tree.");
    }
    else{
        BinaryTree<T> leftTree;
        leftTree = new BinaryTree<T>(root.leftChild);
        root.leftChild = null;
        return leftTree;
    }
}

Public void detachRightSubtree(…
similar to detachLeftSubtree()
}
```

A reference based representation (6/6)

```
Public class TreeException extends RuntimeException{
    public TreeException (String s){
        super(s)
    }
}
```

56

Tree Traversals Using an Iterator

- Iterator interface
  - `next()`, `hasNext()`, and `remove()`

1. Use a queue to order the nodes according to the type of traversal.
2. Initialize iterator by type (pre, post or in) and enqueue all nodes, in order, necessary for traversal
3. dequeue in the next operation

57

What is Java `Iterator`?

- An iterator allows going over all the element of the collection in sequence
- Unlike Enumeration, iterator allows the caller to remove element from the underlying collection
  - java.util.Iterator
    - boolean hasNext()
    - Object next()
    - void remove()
  - Java.util.Enumeration
    - Boolean hasMoreElement()
    - Object nextElement()

58

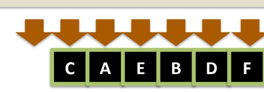
How to use Java `Iterator`?

```
import java.util.*;
class IteratorDemo {
  public static void main(String args[]) {
    // create an array list
    ArrayList al = new ArrayList();
    // add elements to the array list
    al.add("C"); al.add("A"); al.add("E");
    al.add("B"); al.add("D"); al.add("F");
```

59

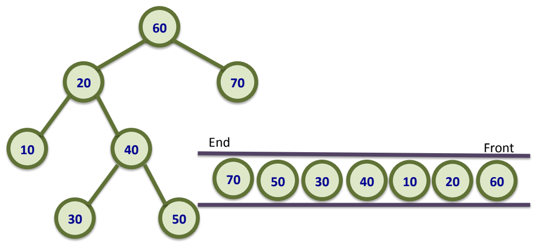
How to use Java `Iterator`?

```
// use iterator to display contents of al
Iterator itr = al.iterator();
  while(itr.hasNext()) {
     Object element = itr.next();
     System.out.print(element + " ");
  }
}
```



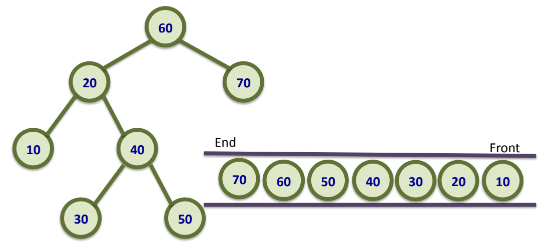
60

Using TreeIterator for the Preorder



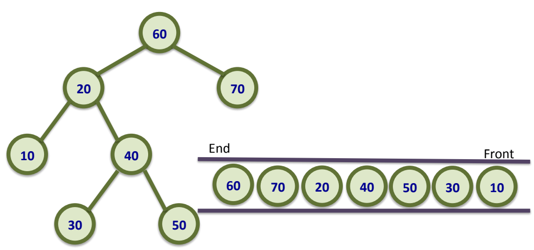
61

Using TreeIterator for the Inorder



62

Using TreeIterator for the Postorder



63

TreeIterator

```
import java.util.LinkedList;
public class TreeIterator<T> implements java.util.Iterator<T> {

   private BinaryTreeBasis<T> binTree;
   private TreeNode<T> currentNode;
   private LinkedList <TreeNode<T>> queue;
   public TreeIterator(BinaryTreeBasis<T> bTree) {}

   public boo
   public T                                        tException{}

   public voi
   public void
   public void

   private voi
   private void
   private void
}
```

```
binTree = bTree;
return !queue.isEmpty();
currentNode = queue.remove();
queue.clear();
queue.clear();  \
if (treeNode != null) {
        postorder(treeNode.getLeft());
        postorder(treeNode.getRight());
        queue.add(treeNode);
} // end if
```

Outline

- Terminology
- Binary Tree
  - Basic operations
  - Traversals of a Binary Tree
  - Representations of a Binary Tree
  - Implementations
- **Binary Search Tree**
  - Algorithms
  - Implementations
  - Efficiency
  - TreeSort
- General Tree

65

Binary Search Trees

Find the record for the person whose ID number is 1212121212.

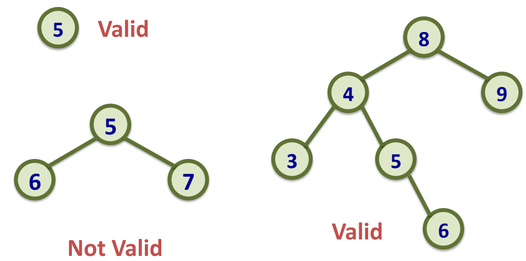
66

Binary Search Trees

- A binary tree T is a binary search tree if for every node n in T:
  - n's value is greater than all values in its left subtree T_L
  - n's value is less than all values in its right subtree T_R
  - T_R and T_L are binary search trees

67

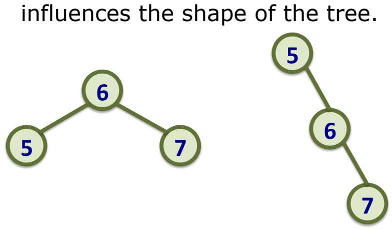
Is this a Binary Search Tree?



68

Organizing Items in BST

- BST uses **inorder** traversal.
- The sequence of adding and removing nodes influences the shape of the tree.



69

BST operations (1/2)

- Insert a new item
  - `insert(in newItem:TreeItemType)`
- Delete the item with a given search key
  - `delete(in searchKey: KeyType) throws TreeException`
- Retrieve the item with a given search key
  - `retrieve(in searchKey:KeyType):TreeItemType`

70

Example of nodes in BST

```
public class Person {
  private String id;
  private String phoneNo;
  private String address;
  public Person( String _id, String _phoneNo,
                 String _address){
     id = _id;
     phoneNo = _phoneNo;
  .
  .
  .
  }
  //other methods appear here
}
```

71

Search in BST

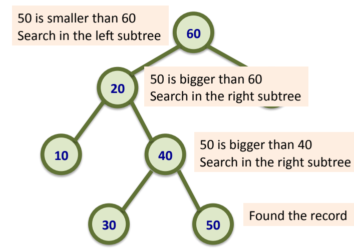
- BST is a recursive algorithm.
- Search key should be unique.
- Using BST's properties.
  - If search key == search key of root's item
    - The record is found
  - If search key < search key of root's item
    - Search in the left subtree → **recursive method**
  - If search key > search key of root's item
    - Search in the right subtree → **recursive method**

72

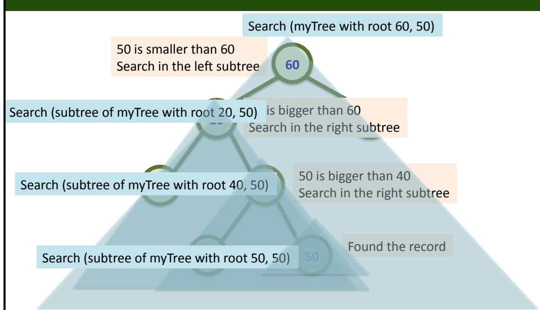
Pseudocode

```
search(in bst:BinarySearchTree, in searchKey:KeyType)
  if (bst is empty){
     The desired record is not found
  }
  else if (searchKey == searchKey of root's item){
     The desired record is found
  }
  else if (searchKey < search key of root's item) {
     search (Left subtree of bst, searchKey)
  }
  else {
     search(Right subtree of bst, searchKey)
  }
```


Example: Find the person with ID 50

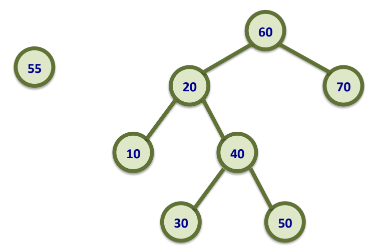


Example: Find 50 (using recursive calls)



Insertion in BST (1/2)

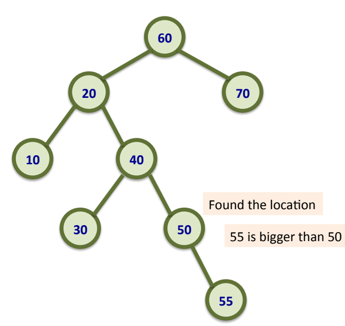
- You want to insert a record for a person
  - Person's ID number is 55.



Insertion in BST (2/2)

- Use search to determine where in the tree to insert a new ID, 55.
- New item will be inserted as a new leaf.
- Different orders of insertion can get a different binary search tree.

Example: Insert a person with ID 55

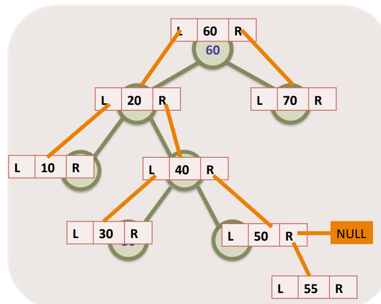


Pseudocode (high-level)

```
insertItem (in treeNode:TreeNode,
            in newItem:TreeItemType)
  Let parentNode be the parent of the empty subtree at
  which search terminates when it seeks newItem's search
  key

  if (search terminated at parentNode's left subtree){
      Set leftChild of parentNode to reference newItem
  }
  else{
      Set rightChild of parentNode to reference newItem
  }
```

79

Reference to the new item

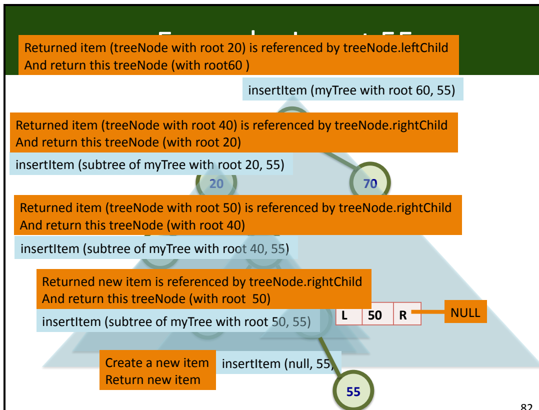


80

Pseudocode

```
insertItem (in treeNode:TreeNode,
              in newItem:TreeItemType)
  if (treeNode is null){
      create a new node and let treeNode reference it
      create a new node with newItem as the data portion
      set the reference in the new node to null
  }
  else if (newItem.getKey() < treeNode.item.getKey()){
      treeNode.leftChild =
                insertItem(treeNode.leftChild, newItem)
  }
  else {
      treeNode.rightChild =
                insertItem(treeNode.rightChild, newItem)
  }
  return treeNode
```

81



82

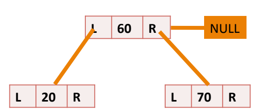
Deletion in BST

- Emptying a tree Vs. Deleting a node from a tree?
- Use search algorithm to locate the item with the specified search key.
- Three cases to consider:
  1. N is a leaf
  2. N has only one child
  3. N has two children

83

Case 1. N is a leaf

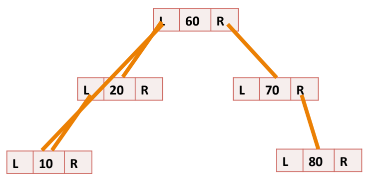
- Set the reference in its parent to null.



84

Case 2. N has only one child (left child)

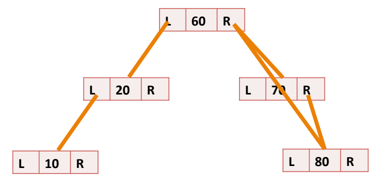
- Let N's parent adopt N's child.



85

Case 2. N has only one child (right child)

- Let N's parent adopt N's child.



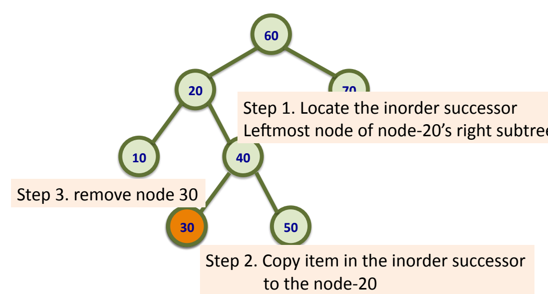
86

Case 3. N has two children

1. Find the inorder successor of N's search key.
   - The node whose search key comes immediately after N's search key
   - The inorder successor is in the leftmost node in N's right subtree.
2. Copy the item of the inorder successor, M to the deleting node N.
3. Remove the node M from the tree.

87

Remove 20



88

Retrieval in BST

```
retrieveItem(in treeNode:TreeNode,
                 in searchKey:KeyType): TreeItemType
   if(treeNode ==null){
        treeItem = null
   }
   else if (searchKey< treeNode.item.getKey()){
        treeItem =
            retrieveItem(treeNode.leftChild, searchKey)
   }
   else{
        treeItem =
            retrieveItem(treeNode.rightChild, searchKey)
   }
   return treeItem
```

89

Does the inorder traversal of a BST visit its nodes in sorted-key order?

Theorem 6-1

The inorder traversal of a binary search tree *T* will visit its nodes in sorted search-key order.

Proof

90

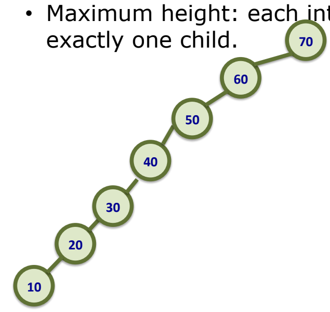
The Efficiency of BST Operations

- Searching, inserting, deleting, and retrieving data
- Compare the specified value `searchKey` to the search keys in the nodes along a path through the tree.
- The maximum number of comparisons that these operations can require is equal to the height of the binary search tree.

91

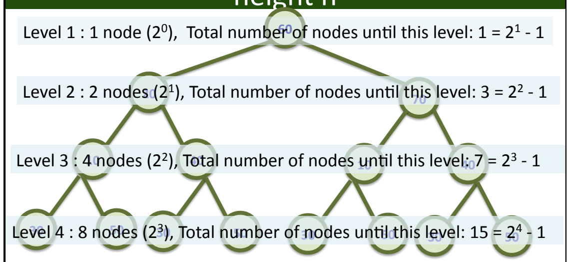
The maximum and minimum heights of a BST

- Maximum height: each internal node has exactly one child.



92

Counting the nodes in a full binary tree of height h



Level 1 : 1 node (2^0), Total number of nodes until this level: $1 = 2^1 - 1$

Level 2 : 2 nodes (2^1), Total number of nodes until this level: $3 = 2^2 - 1$

Level 3 : 4 nodes (2^2), Total number of nodes until this level: $7 = 2^3 - 1$

Level 4 : 8 nodes (2^3), Total number of nodes until this level: $15 = 2^4 - 1$

Level h : 2^{h-1} nodes, Total number of nodes until this level: $2^h - 1$

93

Theorem 6-2

The maximum number of nodes that a binary tree on the level $h > 0$ can have is 2^{h-1} nodes

Proof

94

Theorem 6-3

A full binary tree of height $h \geq 0$ has $2^h - 1$ nodes

Proof

95

Theorem 6-4

The minimum height of a binary tree with n nodes is $\lceil \log(n+1) \rceil$

Proof

96

- Complete trees and full trees with n nodes have heights of $\lceil \log(n+1) \rceil$
- The height of an n-node BST ranges from $\lceil \log(n+1) \rceil$ to n
- Insertion in search-key order will need maximum-height BST.
  – Adding the node to the leaf node.

97

Treesort

- Sort an array of records efficiently into a search-key order.
- Each insertion into a BST requires $O(\log n)$ operations in the average case, and $O(n)$ for the worst case.
- n insertions requires $O(n \log n)$ operations in the average case and $O(n^2)$ in the worst case.
- Sorted items in the tree are traversed and copied to the array: $O(n)$
- Average case: $O(n \log n) + O(n) = O(n \log n)$
- Worst case: $O(n^2) + O(n) = O(n^2)$

98

Storing a BST in a file

- Storing and restoring
- Original shape vs. balanced shape?
  – There is NO difference in the ADT operations.
  – Efficient operations are assured if the binary search tree is balanced.

99

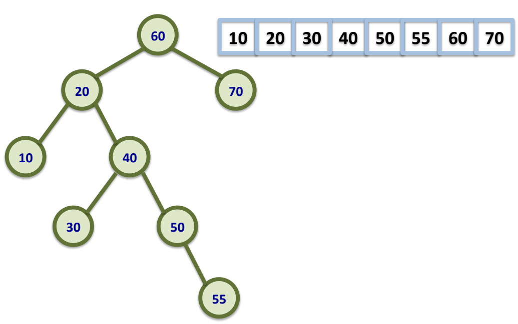
How to guarantee a restored tree of minimum height

- Create complete binary search tree with **n** nodes

```
readTree(in inputFile:FileType,
         in n:integer):Treenode

  if (n > 0 ){
     treeNode = reference to new node
                with null child references
     Set treeNode's left child to
          readTree(inputFile, n/2)
      Read Item from file into treeNode's item
      Set treeNode's right child to
          readFull(inputFile, (n-1)/2)
   }
   return treeNode
```

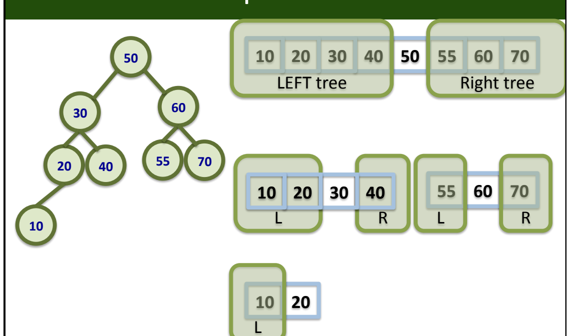
100

Example: store



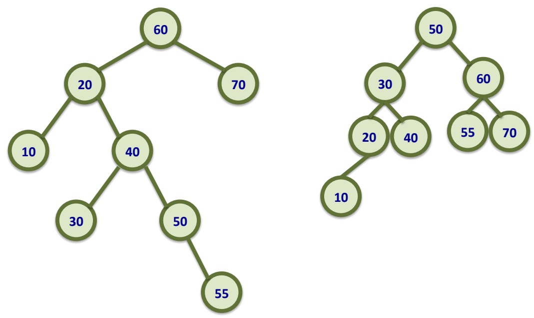
101

Example: restore



102

Comparison



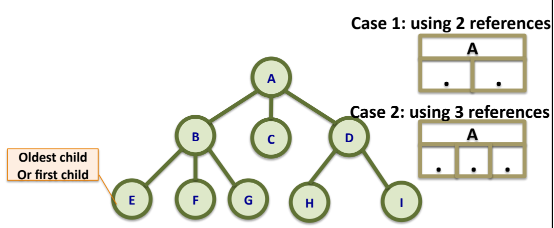
103

n-ary General tree

- Tree with no more than n children.
- How can we implement it?

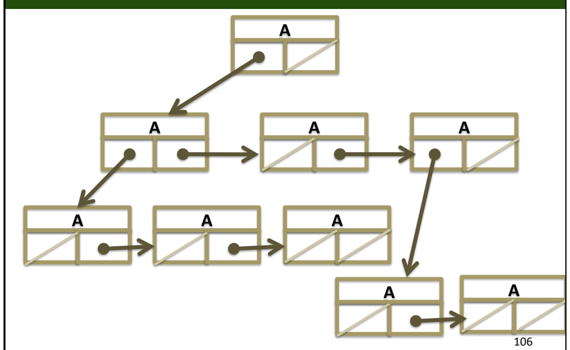
104

n = 3



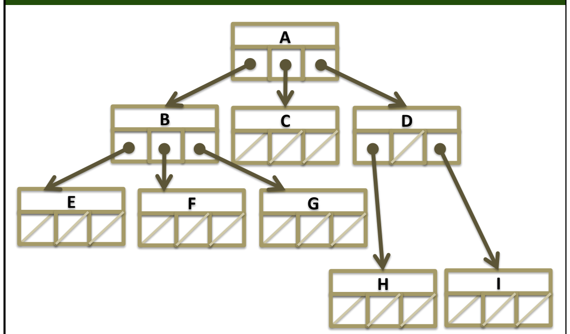
105

Case 1: Using 2 references



106

Case 2: Using 3 references



107