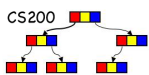



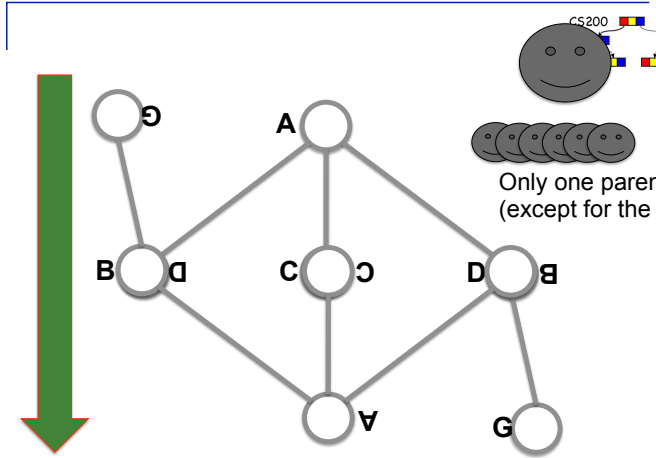
CS200 

CS200: Trees

Rosen Ch. 10.1 & 10.3
Walls Ch. 11

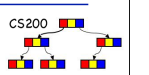
CS200 - Trees 1

CS200 

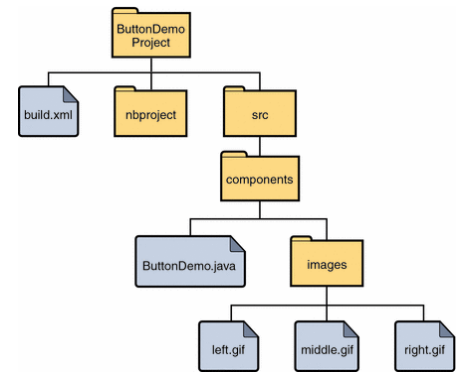


Tree grows top to bottom!

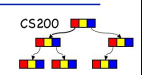
CS200 - Trees 2

CS200 

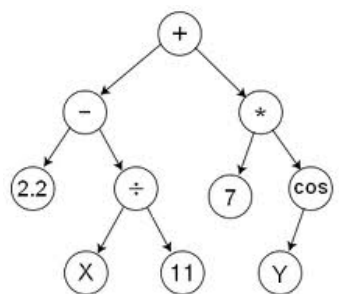
Applications – File System



CS200 - Trees 3

CS200 

Applications – Expression Tree



$$\left(2.2 - \left(\frac{X}{11} \right) \right) + (7 * \cos(Y))$$

CS200 - Trees 4

Applications - Parse Trees

Used in compilers to check syntax

CS200 - Trees

5

Decision trees

- Example: a tree for deciding whether to wait for a table at a restaurant

CS200 - Trees

6

Question : Can we model this with a Tree Data Structure?

CS200 - Trees

8

Tree Terminology

Degree?
Depth/Level?
Height?

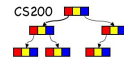
The parent child relationship is generalized to the relationship of ancestor and descendant

All the defs are in page 525 of the textbook

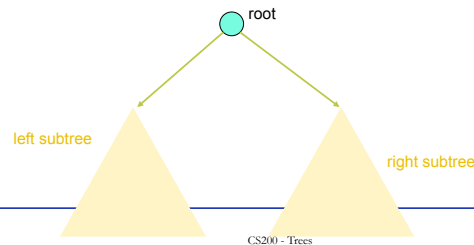
CS200 - Trees

8

Binary Trees

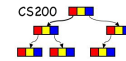


- A **binary tree** is a set T of nodes such that either
 - T is empty, or
 - T is partitioned into three disjoint subsets:
 - A single node r , the root
 - Two possibly empty sets that are binary trees, called left and right subtrees of r



9

Tree Terminology



- **Level/depth** of a node n in a tree T
 - If n is the root of T , it is at level 1
 - If n is not the root of T , its level is 1 greater than the level of its parent

CS200 - Trees

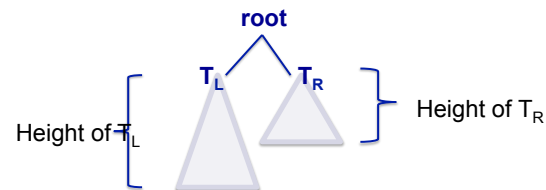
10

Height of a Binary Tree



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

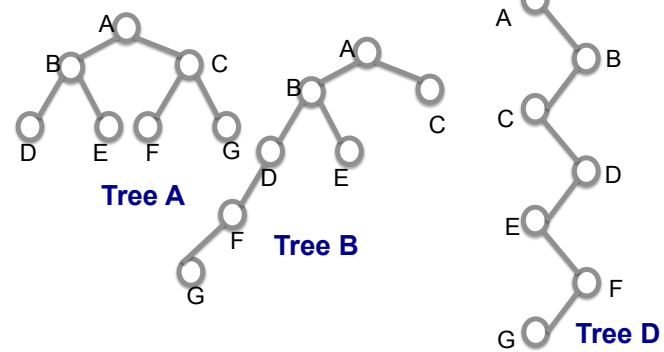
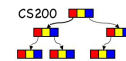
$$height(T) = 1 + \max\{height(T_L), height(T_R)\}$$



CS200 - Trees

11

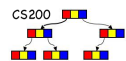
Binary trees with same nodes but different heights



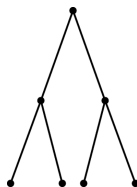
CS200 - Trees

12

Trees - more definitions



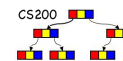
- m-ary tree
 - Every internal vertex has no more than m children.
 - Our main focus will be binary trees
- Full m-ary tree
 - all interior nodes have m children
- Perfect m-ary tree
 - Full m-ary tree where all leaves are at the same level
- Perfect binary tree
 - number of leaf nodes: 2^{h-1}
 - total number of nodes: $2^h - 1$
 - Recurrence relations for the # of leaf nodes and total # of nodes?



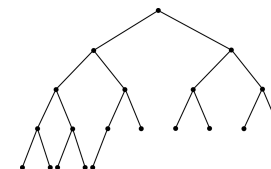
CS200 - Trees

13

More definitions



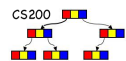
- Complete binary tree of height h
 - zero or more rightmost leaves not present at level h
- A binary tree T of height h is complete if
 - All nodes at level h - 2 and above have two children each, and
 - When a node at level h - 1 has children, all nodes to its left at the same level have two children each, and
 - When a node at level h - 1 has one child, it is a left child



CS200 - Trees

14

More definitions

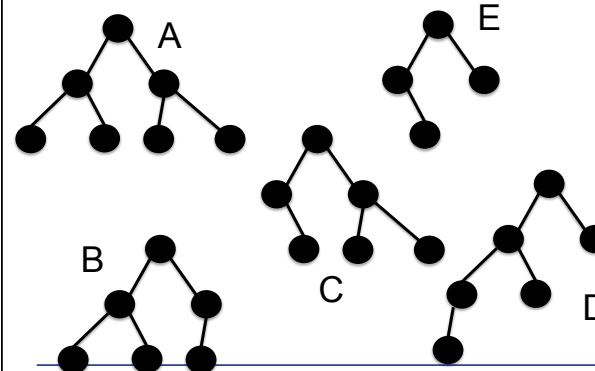
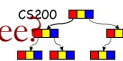


- balanced tree
 - Height of any node's right subtree differs from left subtree by 0 or 1
- A complete tree is balanced

CS200 - Trees

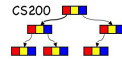
15

Full? Complete? Balanced? Binary trees



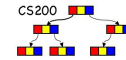
CS200 - Trees

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

General operations

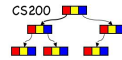


```

Root
Left subtree
Right subtree

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

Example



```

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

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

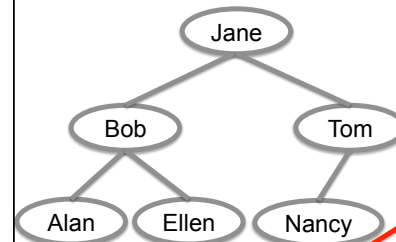
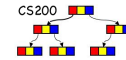
tree3.setRootItem("B")
tree3.attachLeftSubtree(tree2)
tree3.attachRight("E")

tree4.setRootItem("C")

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



Array based representation



A binary tree of names

root 0

Free list: Array - based Linked List

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	-1
7	?	-1	-1
8	?	-1	-1
	.	.	.
	.	.	.
	.	.	.

Array based representation

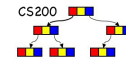


0 1 2 3 4 5

```
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
}
```

CS200 - Trees

Array based representation



```
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++;
    }
}
```

CS200 - Trees

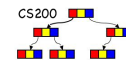
An array based representation



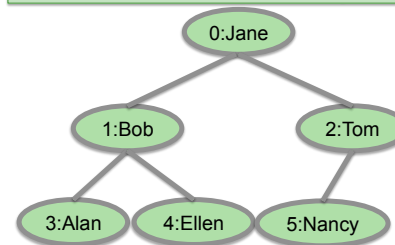
```
public TreeNode<T> getRootItem(){
    return tree.get(root);
}
public TreeNode<T> getRight(){
    return tree.get(root.getRightChild());
}
public TreeNode<T> getLeft(){
    return tree.get(root.getLeftChild());
}
public void makeEmpty(){
    how?
}
More methods..
```

CS200 - Trees

Complete Binary Tree



Level-by-level numbering of a complete binary tree

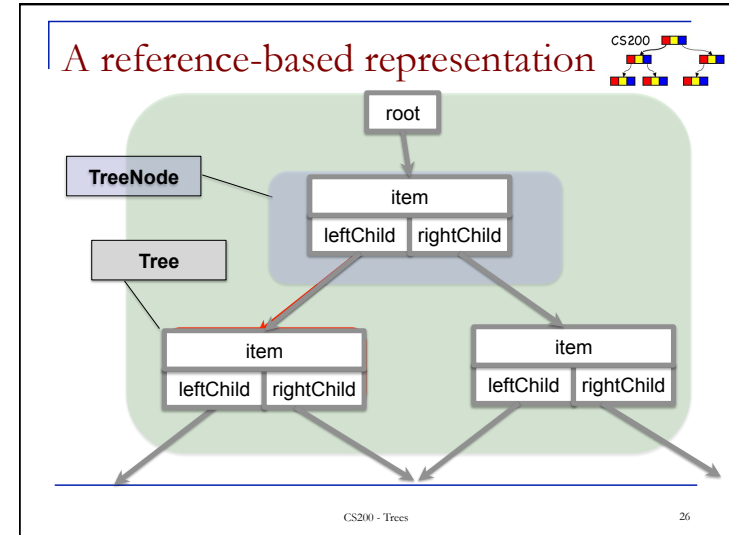
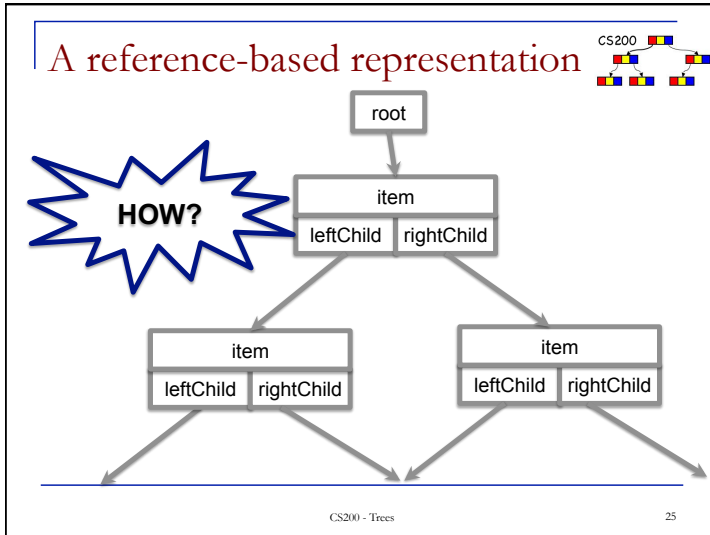


index	item
0	Jane
1	Bob
2	Tom
3	Alan
4	Ellen
5	Karen
6	
7	

If the binary tree is complete, an array-based implementation can be memory-efficient.

CS200 - Trees

24



Reference based: Node

```

public class 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;
    }
}

```

Step 1. TreeNode

CS200

27

Reference based: Tree

```

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.item = newItem;
        }
        else {
            root = new TreeNode<T>(newItem, null, null);
        }
    }
}

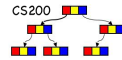
```

Step 2. Tree (BinaryTree)

CS200

28

Reference based: Add Child



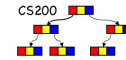
```
public void attachLeft(T newItem){
    if (!isEmpty() && root.leftChild == null) {
        root.leftChild = new TreeNode<T>(newItem, null,
            null);
    }
}

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

CS200 - Trees

29

Reference based: Add Subtree



```
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();
    }
}
```

CS200 - Trees

30

Reference based: Remove Subtree

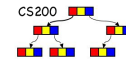


```
public BinaryTree<T> detachLeftSubtree(BinaryTree<T>
    leftTree) throws TreeException{

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

CS200 - Trees

Traversal Algorithms

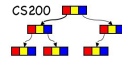


- The traversal of a tree is the process of “visiting” every node of the tree
 - Display a portion of the data in the node.
 - Process the data in the node
- Because a tree is not linear, there are many ways that this can be done.

CS200 - Trees

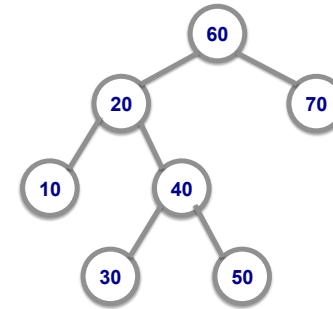
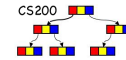
32

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**.

Breadth-first traversal



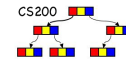
60 – 20 – 70 – 10 – 40 – 30 – 50

Depth-first traversals



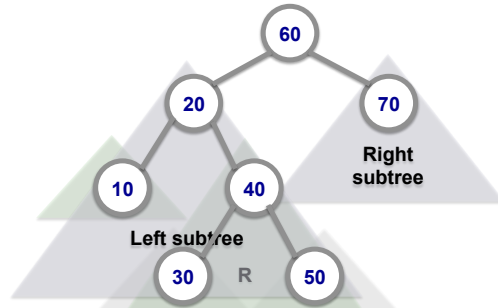
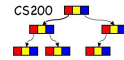
- Three choices of when to visit the root r .
 1. **Before** it traverses both of r 's subtrees
 2. After it has traversed r 's **left** subtree (before it traverses r 's right subtree)
 3. After it has traversed **both** of r 's subtrees
- **Preorder, inorder, and postorder**

Depth First: Preorder traversal



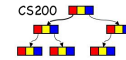
- **Preorder traversal** processes the information at the root, followed by the entire left subtree and concluding with the entire right subtree.

Depth First: Preorder traversal



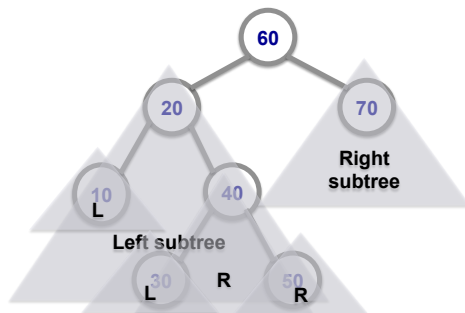
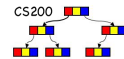
60 – 20 – 10 – 40 – 30 – 50 – 70

Depth First: Inorder traversal



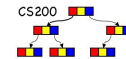
- **Inorder traversal** processes all the information in the left subtree before processing the root.
- It finishes by processing all the information in the right subtree.

Depth First: Inorder traversal



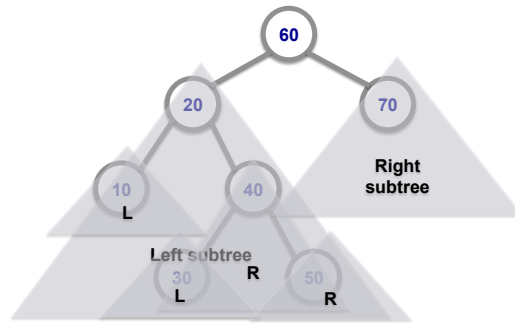
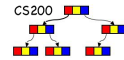
10 – 20 – 30 – 40 – 50 – 60 – 70

Depth First: Postorder traversal



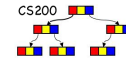
- **Postorder traversal** processes the left subtree, then the right subtree and finishes by processing the root.

Depth First: Postorder traversal



10 – 30 – 50 – 40 – 20 – 70 – 60

Preorder algorithm



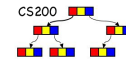
```
preorder (in binTree:BinaryTree)
  if (binTree is not empty){
    display the data in the root of binTree
    preorder(Left subtree of binTree's root)
    preorder(Right subtree of binTree's root)
  }
```

Implementing Traversal with Iterators



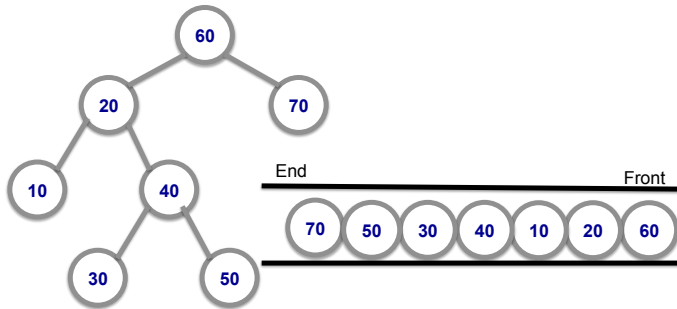
- Use a queue to order the nodes according to the type of traversal.
- Initialize iterator by type (pre, post or in) and enqueue all nodes in order necessary for traversal
- dequeue in **next** operation

What is Java Iterator?



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

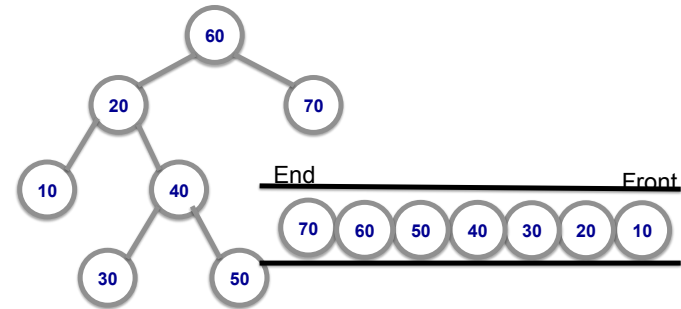
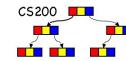
Using TreeIterator for Preorder



CS200 - Trees

45

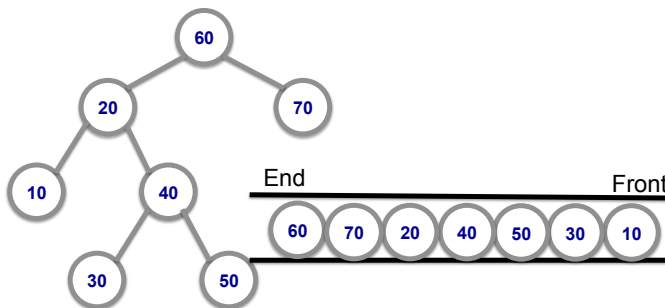
Using TreeIterator for Inorder



CS200 - Trees

46

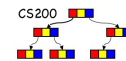
Using TreeIterator for Postorder



CS200 - Trees

47

LevelOrder Algorithm



- Use a *queue* to track unvisited nodes
- For each node that is dequeued,
 - enqueue each of its children
 - until queue empty
- Also called: breadth first traversal

CS200 - Trees

48

LevelOrder

Queue	Output
Init [A]	-
Step 1 [B,C]	A
Step 2 [C,D]	AB
Step 3 [D,E,F]	ABC
Step 4 [E,F,G,H]	ABCD
Step 5 [F,G,H]	ABCDE
Step 6 [G,H,I]	ABCDEF
Step 7 [H,I]	ABCDEFG
Step 8 [I]	ABCDEFGH
Step 9 []	ABCDEFGHI

CS200 - Trees 49

Categories of Data Structures

- Position-oriented data structures: access is by position.
- Value-oriented structures: access is by value.
- Examples?

CS200 - Trees 50

Binary Search Trees

- **Definition:** 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

CS200 - Trees 51

Clicker Q

Which are binary search tree(s)?

- Tree A only
- Tree A and B
- Tree A, B and C
- Tree A and C

CS200 - Trees 52

BST

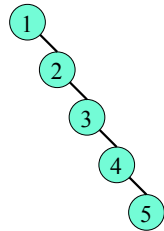
■ Organization

- the sequence of adding and removing influences the shape of the tree

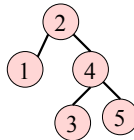
■ Search / Retrieval

- Using *inorder traversal*
- On a search key

1, 2, 3, 4, 5



2, 1, 4, 5, 3



CS200 - Trees

53

BST Methods

`insert(in newItem:TreeItemType)`

- inserts `newItem` into a BST whose items have distinct search keys that differ from `newItem`'s

`delete(in searchKey:KeyType)` throws `TreeException`

- Deletes the item whose search key equals `searchKey`. If none exists, the operation fails.

`retrieve(in searchKey:KeyType):TreeItemType`

- Returns the item whose search key equals `searchKey`. Returns null if not found.

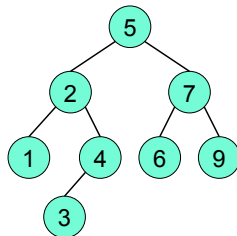
CS200 - Trees

54

BST - Search

compare value with node

- empty: not found
- == : found
- < : search in the left sub-tree
- > : search in the right sub-tree

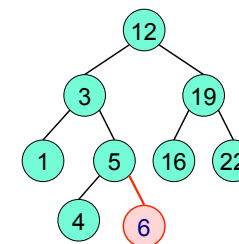


Locate 4 in the BST !

CS200 - Trees

55

BST - Insert

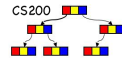


Add 6

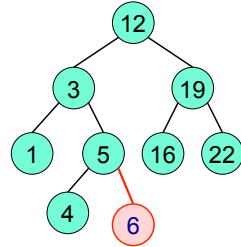
CS200 - Trees

56

BST – Insert



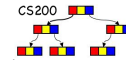
- Always add as a leaf – in the position where the search method would look for it
- Find leaf location
 - < : add to the left sub-tree
 - > : add to the right sub-tree
- Special Cases:
 - already there
 - empty tree



CS200 - Trees

57

Inserting an item



```
insertItem(in treeNode:TreeNode, in newItem:TreeItemType)
// Inserts newItem into the binary search tree of which
//treeNode is the root
```

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
}
```

CS200 - Trees

58

Inserting an item



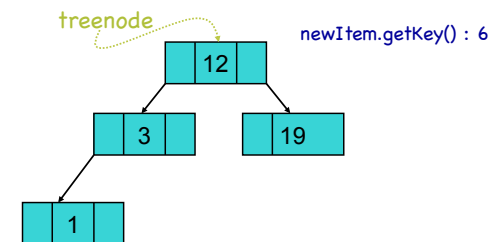
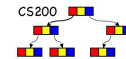
```
insertItem(in treeNode:TreeNode, in newItem:TreeItemType)
// Inserts newItem into the binary search tree of which
// treeNode is the root
if (treeNode is null) {
    create new node with newItem as data
    return new node }
else if (newItem.getKey() < treeNode.getItem().getKey()) {
    treeNode.setLeft(insertItem(treeNode.getLeft(), newItem))
    return treeNode }
else {
    treeNode.setRight(insertItem(treeNode.getRight(),newItem))
    return treeNode }
```

How is insertItem used in the code?

CS200 - Trees

59

BST – Insert

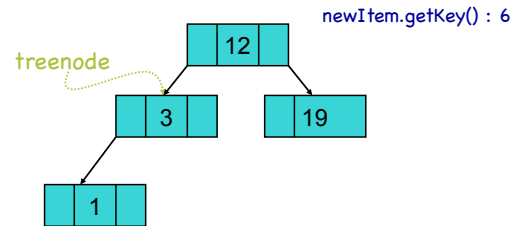
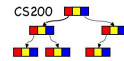


```
if (newItem.getKey() < treeNode.getItem().getKey()) {
    treeNode.setLeft(insertItem(treeNode.getLeft(), newItem))
```

CS200 - Trees

60

BST – Insert

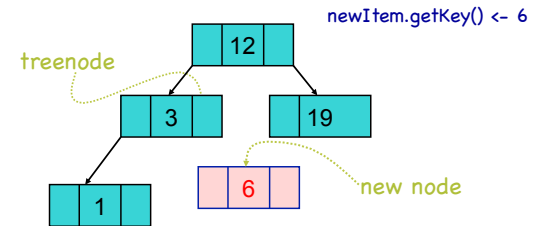
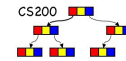


```
else {  
    treeNode.setRight(insertItem(treeNode.getRight(),newItem))  
}
```

CS200 - Trees

61

BST – Insert

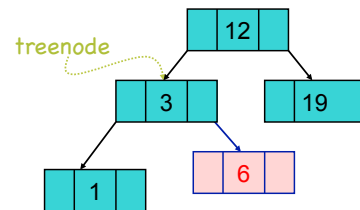
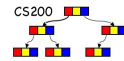


```
if (treeNode is null) {  
    create new node with newItem as data  
    return new node  
}
```

CS200 - Trees

62

BST – Insert



```
treeNode.setRight(insertItem(treeNode.getRight(),newItem))  
return treeNode
```

CS200 - Trees

63

Delete: Cases to Consider



- Delete something that is not there
 - Throw exception
- Delete a leaf
 - Easy, just set link from parent to null
- Delete a node with one child
- Delete a node with two children

CS200 - Trees

64

Delete
Case 1: one child

delete(5)

Other child becomes root

CS200 - Trees 65

Delete
Case 2: two children

delete(5)

Strategy: replace node with a node that is easier to remove!

CS200 - Trees 66

Digression: inorder traversal of BST

- In order:
 - go left
 - visit the node
 - go right
- The keys of an inorder traversal of a BST are in sorted order!

CS200 - Trees 67

Delete
Case 2: two children

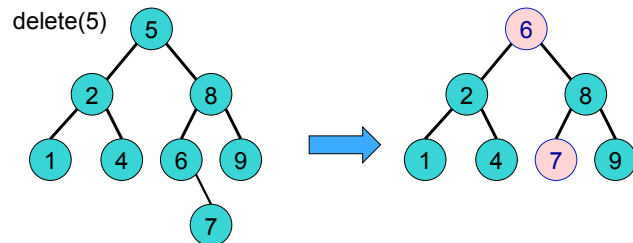
delete(5)

Replace root with its leftmost right descendant and replace that node with its right child, if necessary (an easy delete case).
That node is the inorder successor of the root

CS200 - Trees 68

Delete

Case 2: two children



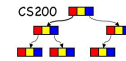
Replace root with its leftmost right descendant and replace that node with its right child, if necessary (an easy delete case). That node is the inorder successor of the root

CS200 - Trees

69

Delete

Case 2: 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.

CS200 - Trees

70

Delete Pseudo Code I

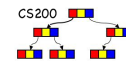


```
deleteItem(in rootNode:TreeNode, in searchKey:KeyType): TreeNode
if (rootNode is null){ throw TreeException}
else if (searchKey equals key in rootNode item) { //found it
    newRoot = deleteNode(rootNode, searchKey) ← remove it
    return newRoot }
else if (searchKey < key in rootNode item) { //search left
    newLeft = deleteItem(rootNode.getLeft(), searchKey)
    rootNode.setLeft(newLeft) ← repair links to child nodes
    return rootNode }
else { // search right
    newRight = deleteItem(rootNode.getRight(), searchKey)
    rootNode.setRight(newRight) ← repair links to child nodes
    return rootNode }
```

CS200 - Trees

71

Delete Pseudo Code II

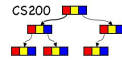


```
deleteNode(in treeNode:TreeNode):TreeNode
// deletes the item in the node referenced by treeNode
// returns root of resulting subtree
if (treeNode is leaf) { return null }
else if (treeNode has only 1 child c) { ← Case 1: replace root w/child
    if (c is left child) { return treeNode.getLeft() }
    else { return treeNode.getRight() } }
else { ← Case 2: replace root w/leftmost child on right
    // find leftmost child on right as replacement
    replacementItem = findLeftMost(treeNode.getRight()) // grab it
    replacementRChild = deleteLeftmost(treeNode.getRight())
    Set treeNode's item to replacementItem
    Set treeNode's right child to replacementRChild
    return treeNode }
```

CS200 - Trees

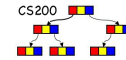
72

Delete Pseudo Code III



```
deleteLeftmost(in treeNode:TreeNode):TreeNode
// Deletes the node that is the leftmost descendant of the tree rooted at treeNode
// Returns subtree of deleted node
if (treeNode.getLeft() is null) // found the node to delete
    { return treeNode.getRight() }
else { // still replacing left nodes
    replacementLChild = deleteLeftmost(treeNode.getLeft())
    treeNode.setLeft(replacementLChild)
    return treeNode
}
```

Complexity of BST Operations



	Average	Worst
search	$O(\log n)$	$O(n)$
insert	$O(\log n)$	$O(n)$
delete	$O(\log n)$	$O(n)$

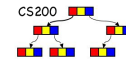
Compare with a sorted list

Properties of Trees



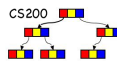
1. An undirected tree has a unique simple path between any two of its vertices.
2. A tree with n vertices has $n-1$ edges.
3. A full m -ary tree with i internal vertices contains $n=mi+1$ vertices.
4. A full m -ary tree with
 - a) n vertices has $i=(n-1)/m$ internal vertices and $l=\lfloor(m-1)n+1\rfloor/m$ leaves,
 - b) i internal vertices has $n=mi+1$ vertices and $l=(m-1)i+1$ leaves,
 - c) l leaves has $n=(ml-1)/(m-1)$ vertices and $i=(l-1)/(m-1)$ internal vertices.
5. There are at most m^h leaves in an m -ary tree of height h .

Tree Sort



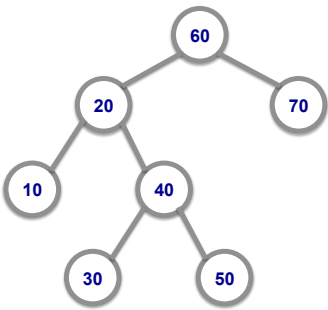
- Uses the binary search tree ADT to sort an array of records according to search-key
- Efficiency
 - Average case: $O(n * \log n)$
 - Worst case: $O(n^2)$

Example of Binary sorting

CS200 

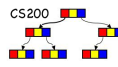
Create Tree
60 20 10 40 70 50 30

Traversal Tree
10 20 30 40 50 60 70



CS200 - Trees 77

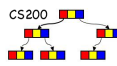
n -ary General tree

CS200 

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

CS200 - Trees 78

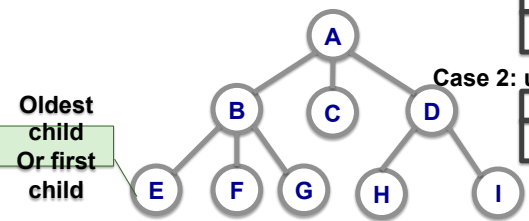
$n = 3$

CS200 

Case 1: using 2 referer

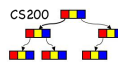
Case 2: using 3 referer

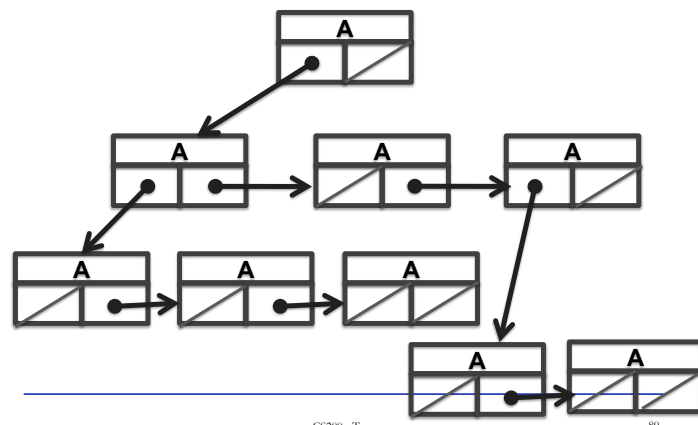
Oldest child
Or first child



CS200 - Trees 79

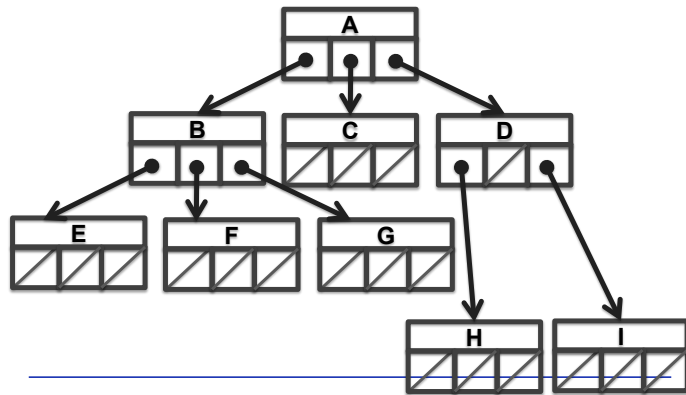
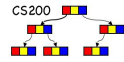
Case 1: Using 2 references

CS200 



CS200 - Trees 80

Case 2: Using 3 references



CS200 - Trees

81