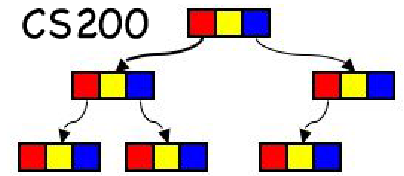
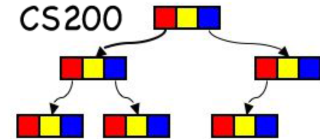
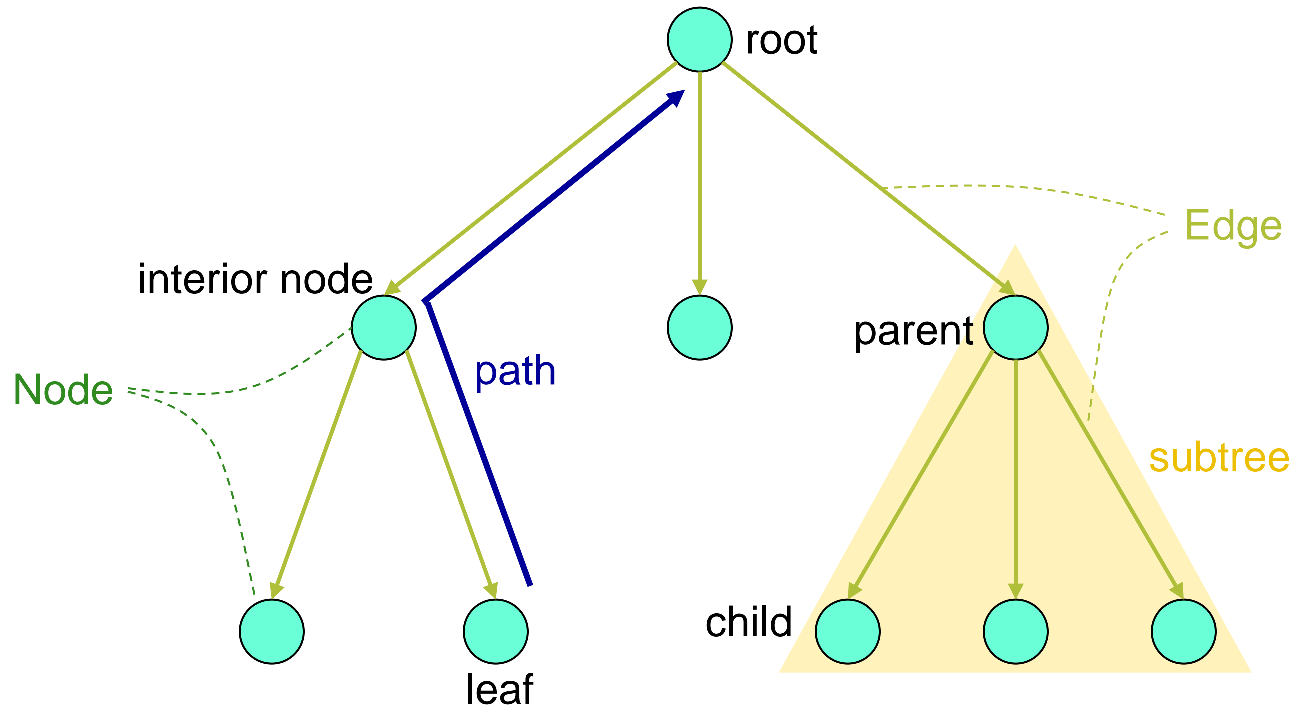


CS200: Trees

Walls Ch. 11

Tree Terminology





Degree?

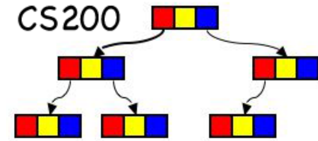
Depth/Level?

Height?

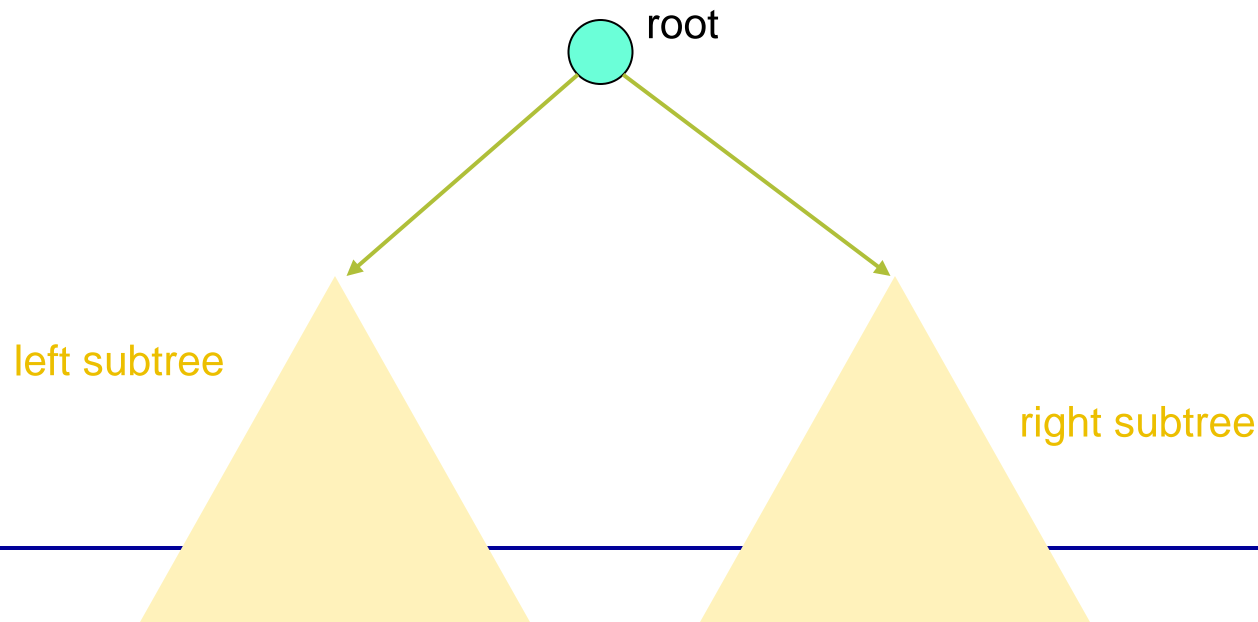
The parent child relationship is generalized to the relationship of ancestor and descendant

All the defs are in page 525 of the textbook

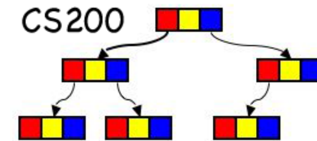
Binary Trees



- 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



Tree Terminology

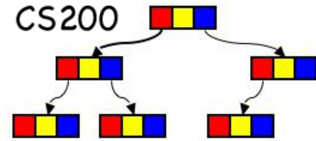


- Level/depth of a node n in a tree T
  - If n is the root of T, it is at level 1
  - If n is not the root of T, its level is 1 greater than the level of its parent
- Height of a tree T defined in terms of the levels of its nodes
  - If T is empty, its height is 0
  - If T is not empty, its height is equal to the maximum level of its nodes or alternatively:

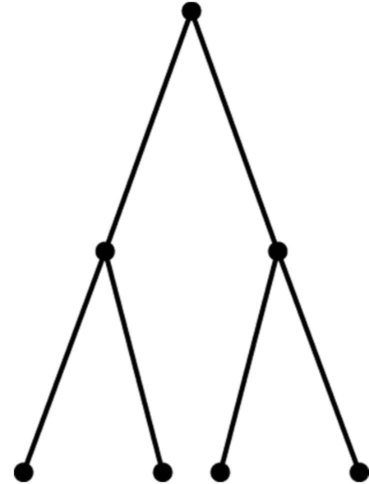
    $height(T) = 1 + max(height(T_L), height(T_R))$

Can also be defined in terms of path lengths (# of nodes on a path from the root)

Trees - more definitions

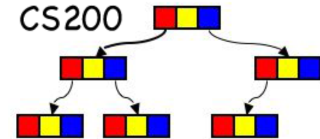


- m-ary tree
  - Every internal vertex has no more than m children.
  - Our main focus will be binary trees
- Full m-ary tree
  - all interior nodes have m children
- Perfect m-ary tree
  - Full m-ary tree where all leaves are at the same level

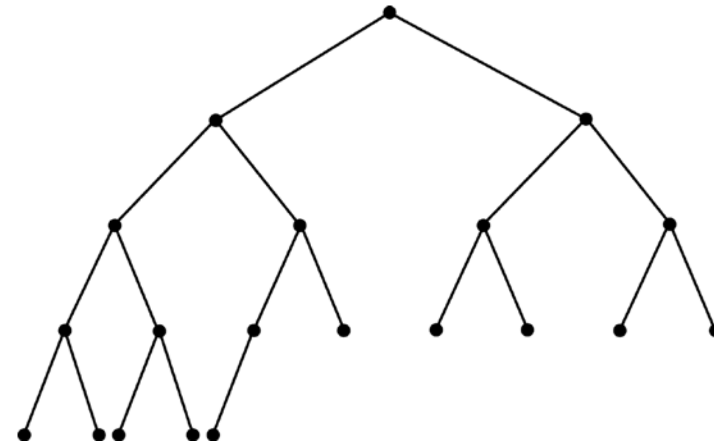


- Perfect binary tree
  - number of leaf nodes: 2^{h-1}
  - total number of nodes: $2^h - 1$
  - Recurrence relations for the # of leaf nodes and total # of nodes?

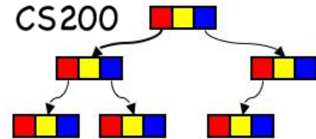
More definitions



- 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

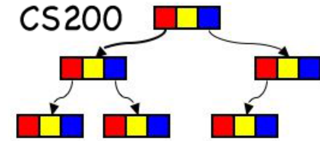


More definitions

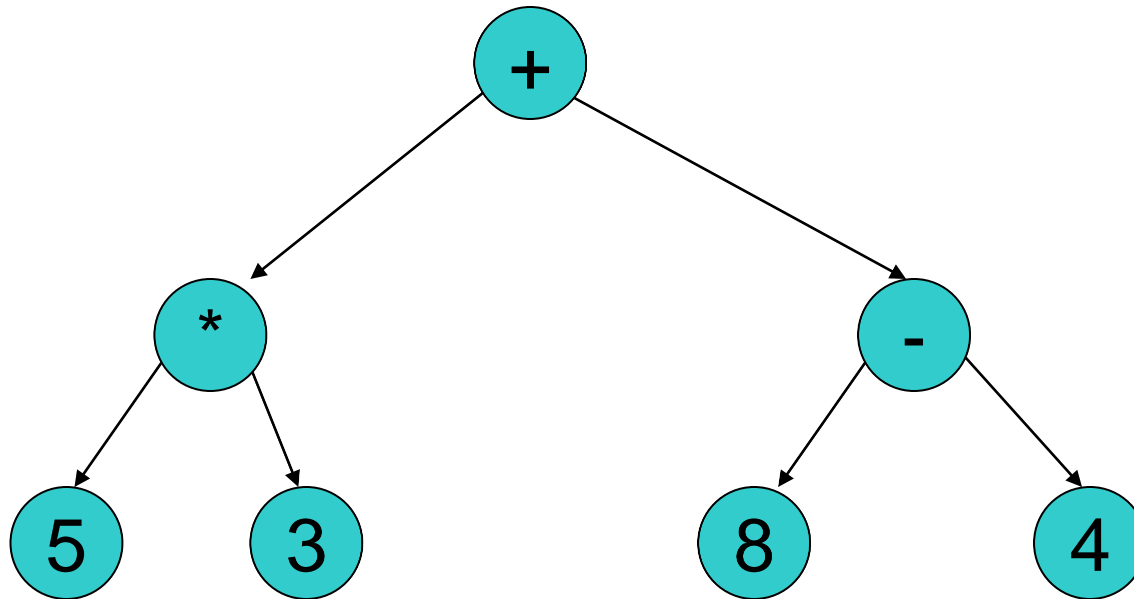


- balanced tree
  - Height of any node's right subtree differs from left subtree by 0 or 1
- A complete tree is balanced

Applications - Expression Trees

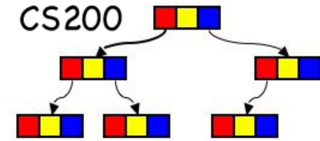


unambigously represent infix expressions



(5 * 3) + (8 - 4)

Applications - Parse Trees



Used in compilers to check syntax

assignment statement

identifier = *expression* ;

x

expression + *expression*

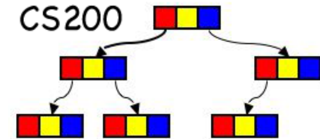
identifier

number

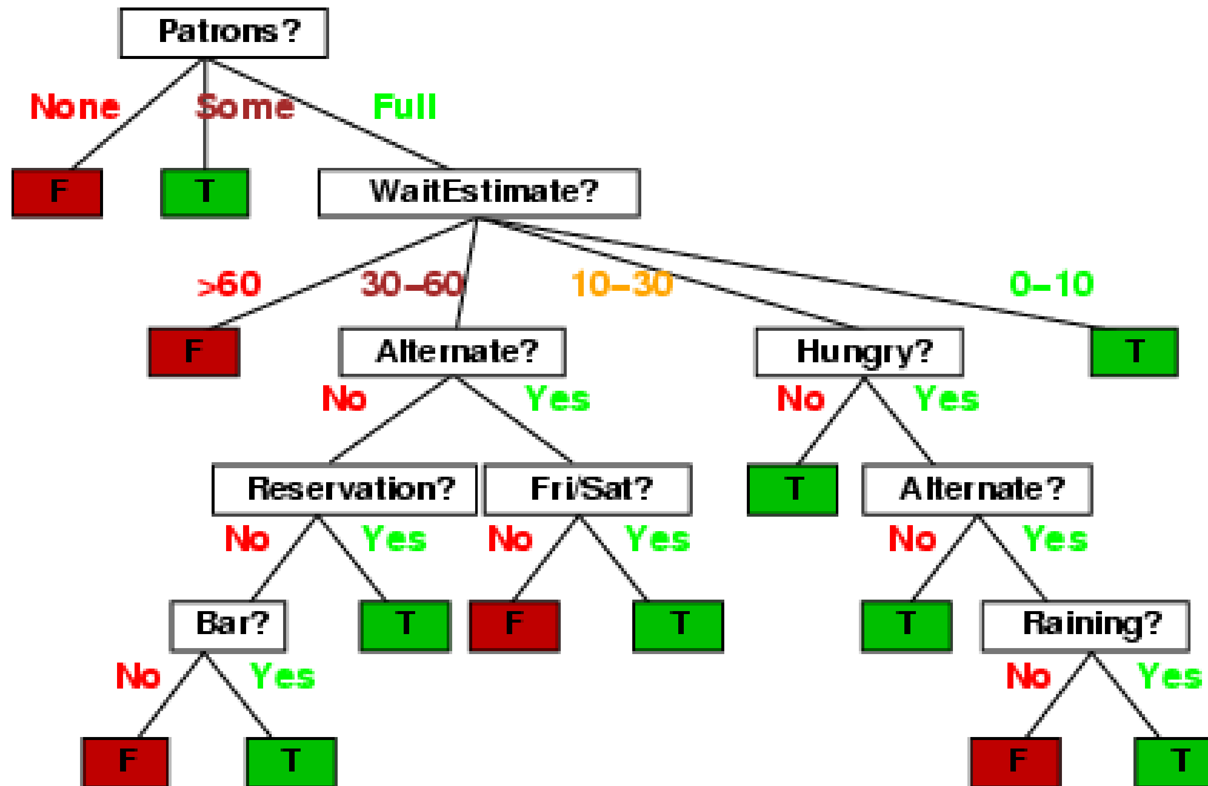
y

1

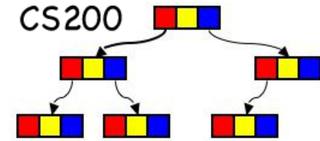
Decision trees



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

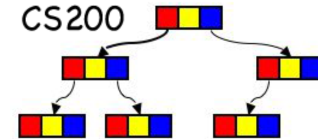


Binary Tree ADT



- Create
  - createBinaryTree()
  - createBinaryTree(in rootItem:TreeItemType)
- Add/Modify
  - setRootItem(in rootItem:TreeItemType) throws UnsupportedOperationException
- Remove
  - makeEmpty()
- Ask
  - isEmpty():boolean {query}
  - getRootItem():TreeItemType throws TreeException {query}

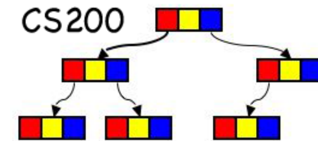
Extensions to Binary Tree ADT

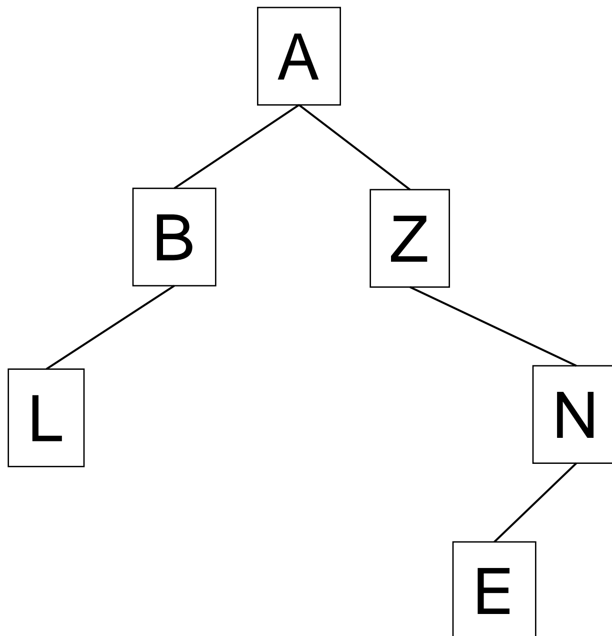


- Create
  - createBinaryTree(in rootItem:TreeItemType, in leftTree:BinaryTree, in rightTree: BinaryTree)
- Add/Modify
  - attachLeft(in newItem:TreeItemType) throws TreeException
  - attachLeftSubtree(in leftTree:BinaryTree) throws TreeException
- Remove
  - detachLeftSubtree():BinaryTree throws TreeException
- Ask
  - getLeftSubtree():BinaryTree throws TreeException

  Show example of how to use the ADT

Binary Tree: Array Implementation





0	A	1	2
1	B	5	-1
2	Z	-1	3
3	N	4	-1
4	E	-1	-1
5	L	-1	-1
6	?	-1	7
7	?	-1	8

Root

0

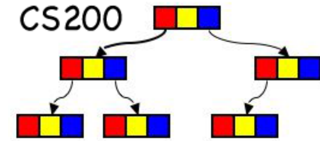
Free

6

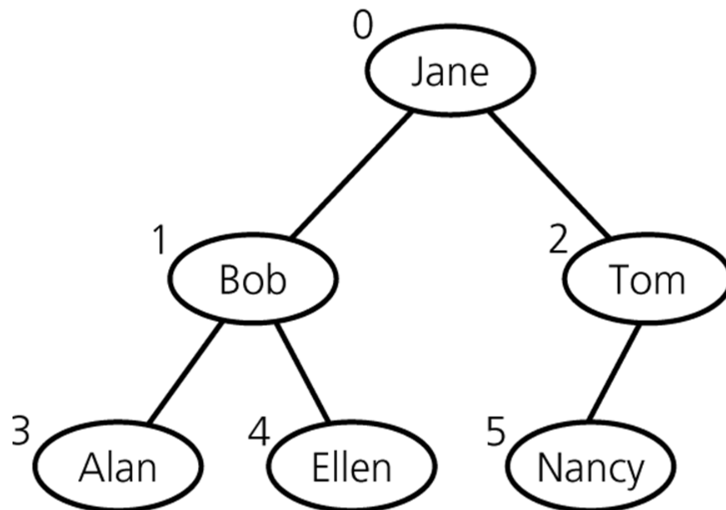
Need to maintain a list of free positions

How do we update the free positions when deleting a node?

Complete Binary Tree



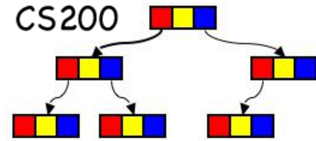
- If the binary tree is complete and remains complete
  - A memory-efficient array-based implementation can be used



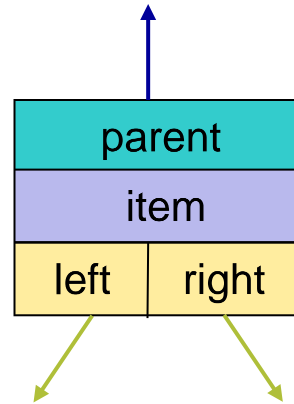
0	Jane
1	Bob
2	Tom
3	Alan
4	Ellen
5	Nancy
6	
7	

Indices of left/right child and parent node?

Reference Implementation

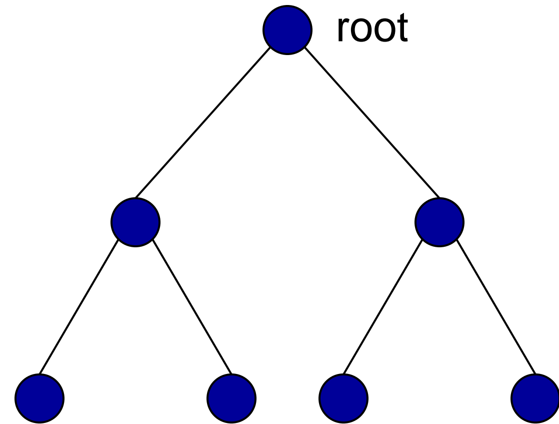


- TreeNode
  - item
  - left child
  - right child
  - parent (optional)

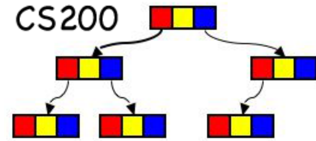


What if a node does not have a left child?

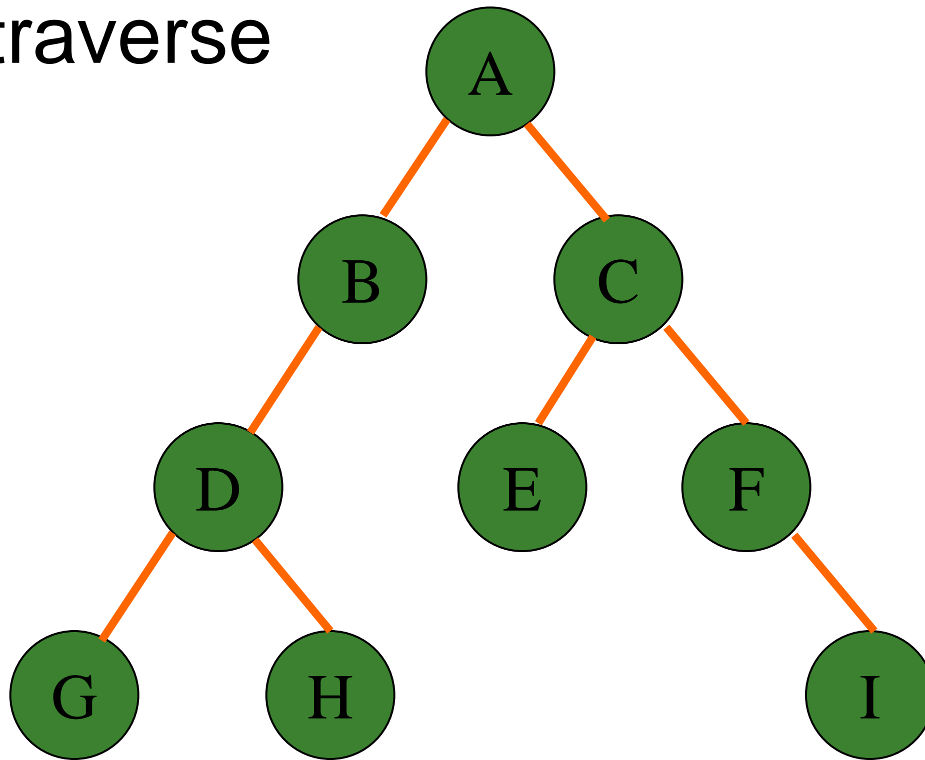
- Tree
  - root
  - size (optional)



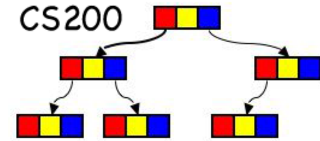
Traversing a binary tree



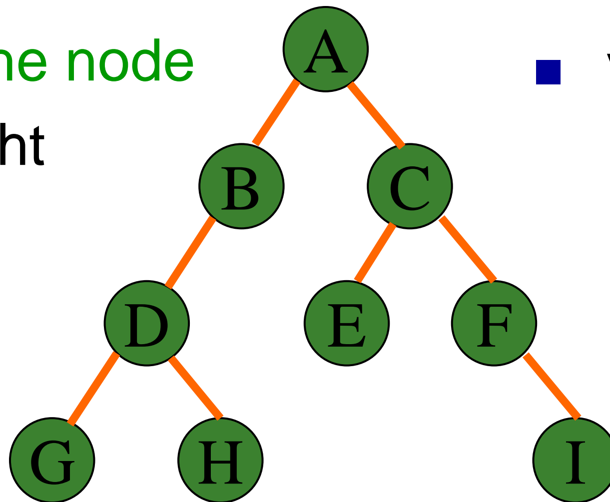
- How to traverse a tree?



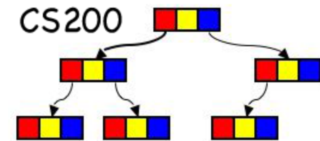
Traversing a Binary Tree

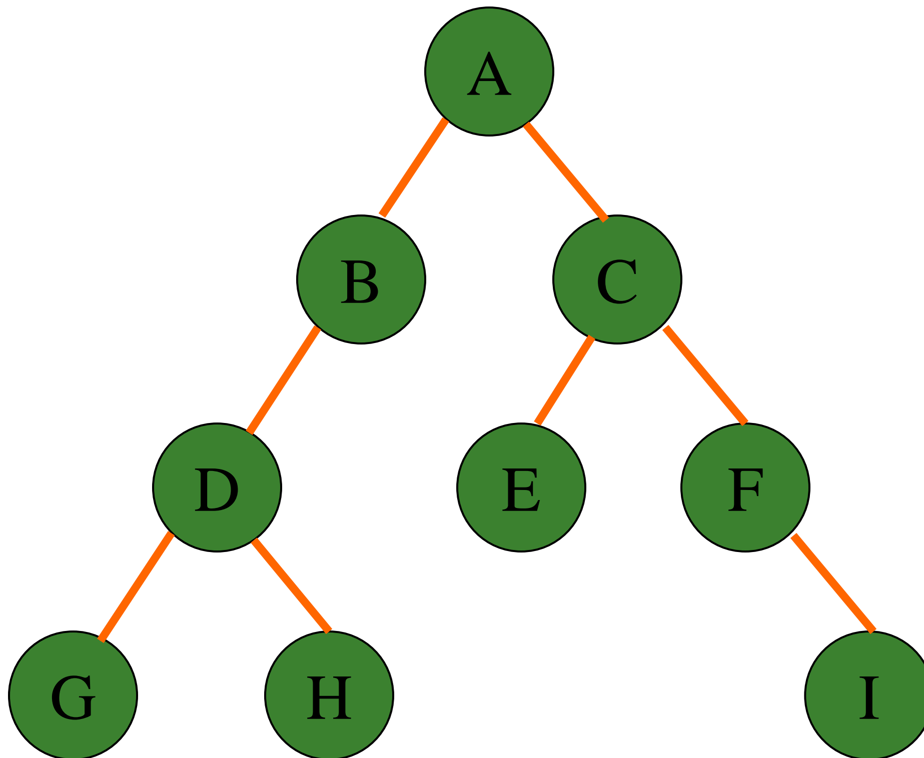


- Pre order
  - visit the node
  - go left
  - go right
- In order
  - go left
  - visit the node
  - go right
- Post order
  - go left
  - go right
  - visit the node
- Level order / breadth first
  - for d = 0 to height
    - visit nodes at level d



Traversal Examples





Pre order

A B D G H C E F I

In order

G D H B A E C F I

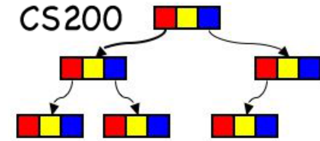
Post order

G H D B E I F C A

Level order

A B C D E F G H I

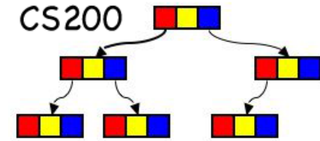
Traversal Implementation



- recursive implementation of preorder
  - The steps:
    - visit node
    - preorder(left child)
    - preorder(right child)
  - base case?
- What changes need to be made for in-order, post-order?

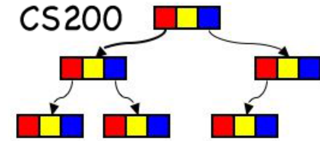
```
void preorder(TreeNode<T>
                      node) {
  doSomething(node);
  preorder(node.getLeft());
  preorder(node.getRight());
}
```

Implementing Traversal with Iterators



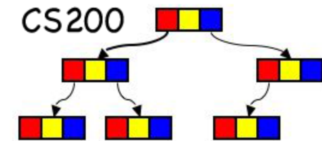
- Use a queue to order the nodes according to the type of traversal.
- Initialize iterator by type (pre, post or in) and enqueue all nodes in order necessary for traversal
- dequeue in next operation

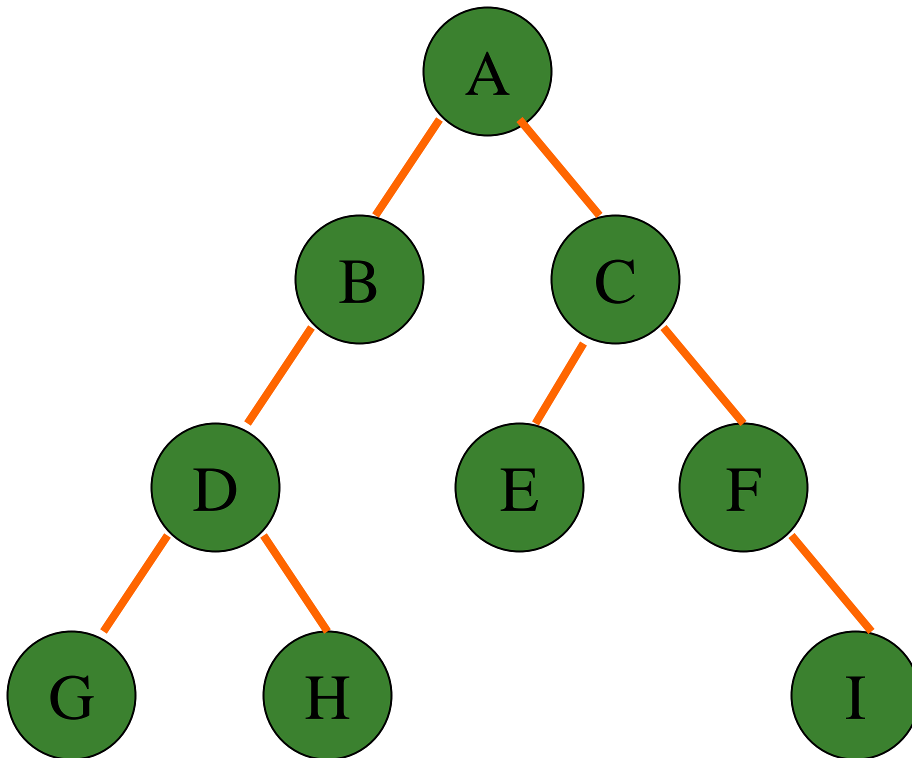
LevelOrder Algorithm



- 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

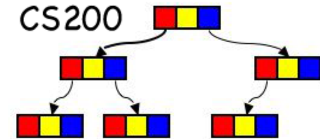
LevelOrder





	Queue	Output
Init	[A]	-
Step 1	[B,C]	A
Step 2	[C,D]	A B
Step 3	[D,E,F]	A B C
Step 4	[E,F,G,H]	A B C D
Step 5	[F,G,H]	A B C D E
Step 6	[G,H,I]	A B C D E F
Step 7	[H,I]	A B C D E F G
Step 8	[I]	A B C D E F G H
Step 9	[ ]	A B C D E F G H I

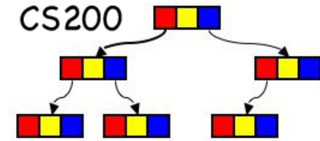
Categories of Data Structures



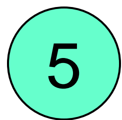
- Position-oriented data structures: access is by position.
- Value-oriented structures: access is by value.

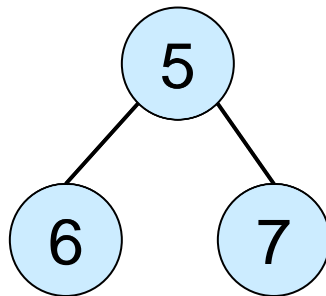
- Examples?

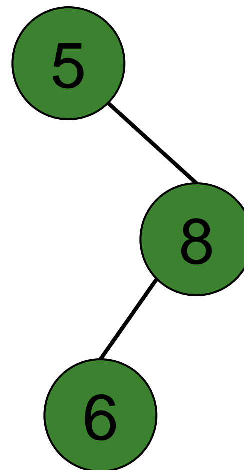
Binary Search Trees

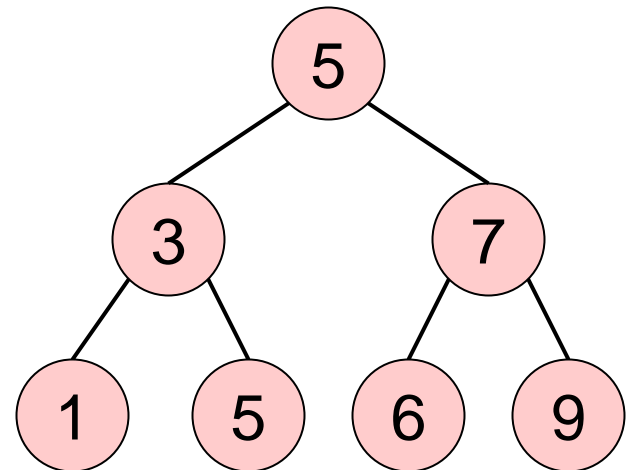


- **Definition**: A binary tree T is a binary search tree if for every node *n* in T:
  - *n*'s value is greater than all values in its left subtree T_L
  - *n*'s value is less than all values in its right subtree T_R
  - T_R and T_L are binary search trees



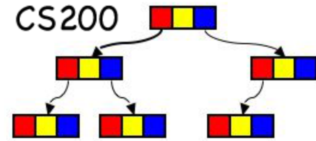




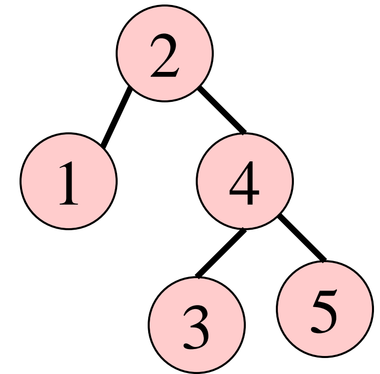


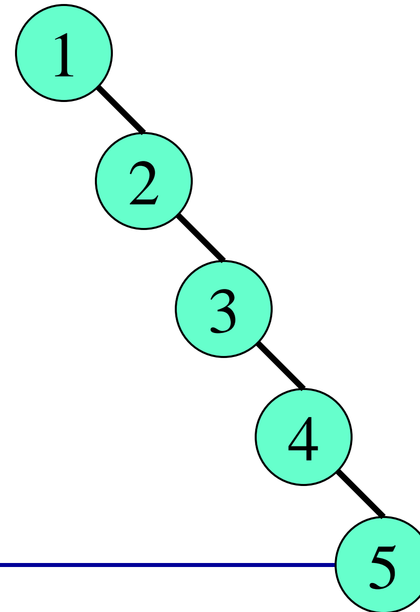
Valid Not Valid Valid ?

BST

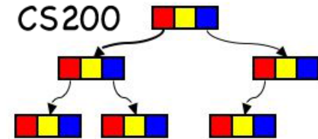


- Organization
  - the sequence of adding and removing influences the shape of the tree
- Search / Retrieval
  - Using *inorder traversal*
  - On a search key





BST ADT



insert(in newIterm:TreeItemType)

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

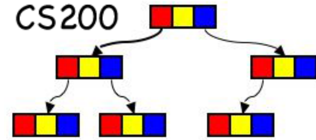
delete(in searchKey: KeyType) throws TreeException

- 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.

UML



BinaryTreeBasis
root
createBinaryTree() *makeEmpty()* *isEmpty()* *getRootItem()* *setRootItem()*

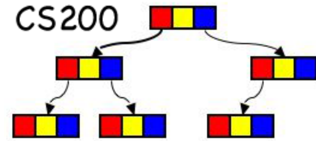
TreeIterator
binary tree
hasNext() *next()* *setPreorder()* *setInorder()* *setPostorder()*

BinaryTree
left subtree *right subtree*
attachLeft() *attachRight()* *attachLeftSubtree()* *attachRightSubtree()* *detachLeftSubtree()* *detachRightSubtree()* *getLeftSubtree()* *getRightSubtree()*

BinarySearchTree
left subtree *right subtree*
insert() *delete()* *retrieve()*

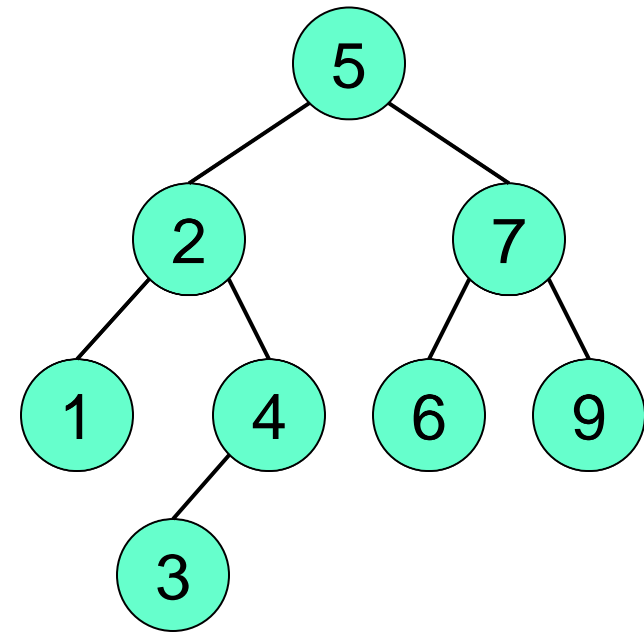
Why doesn't BinarySearchTree extend Binary Tree?

BST - Search



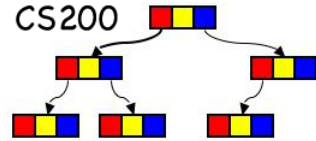
compare value with node

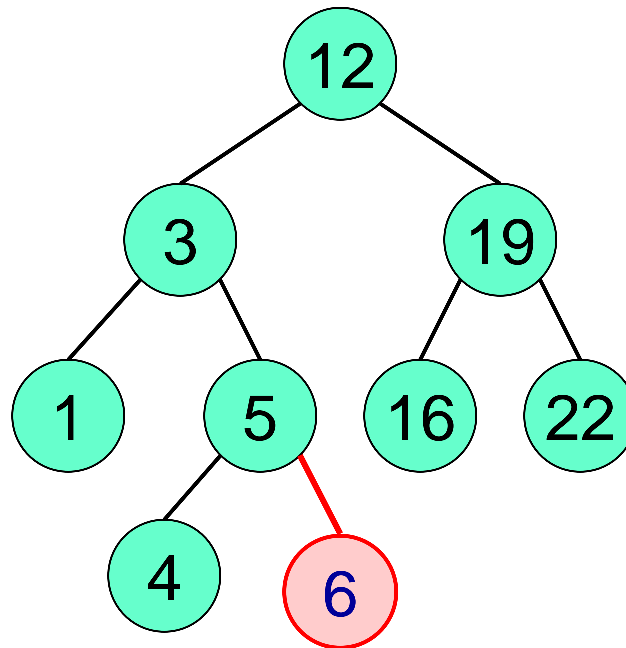
- empty: not found
- == : found
- < : search in the left sub-tree
- > : search in the right sub-tree



Locate 4 in the BST !

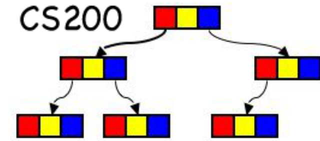
BST – Insert



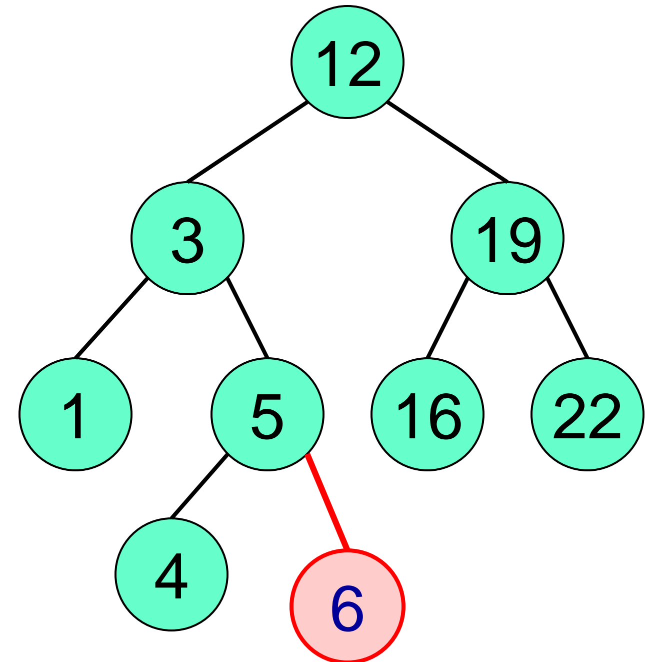


Add 6

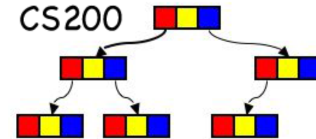
BST – Insert



- 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



Inserting an item



insertItem(in treeNode:TreeNode, in newItem:TreeItemType)

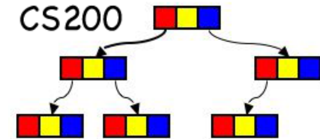
// Inserts newItem into the binary search tree of which

//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
}
```

Inserting an item

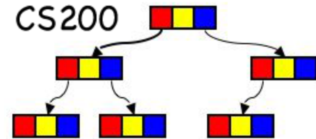


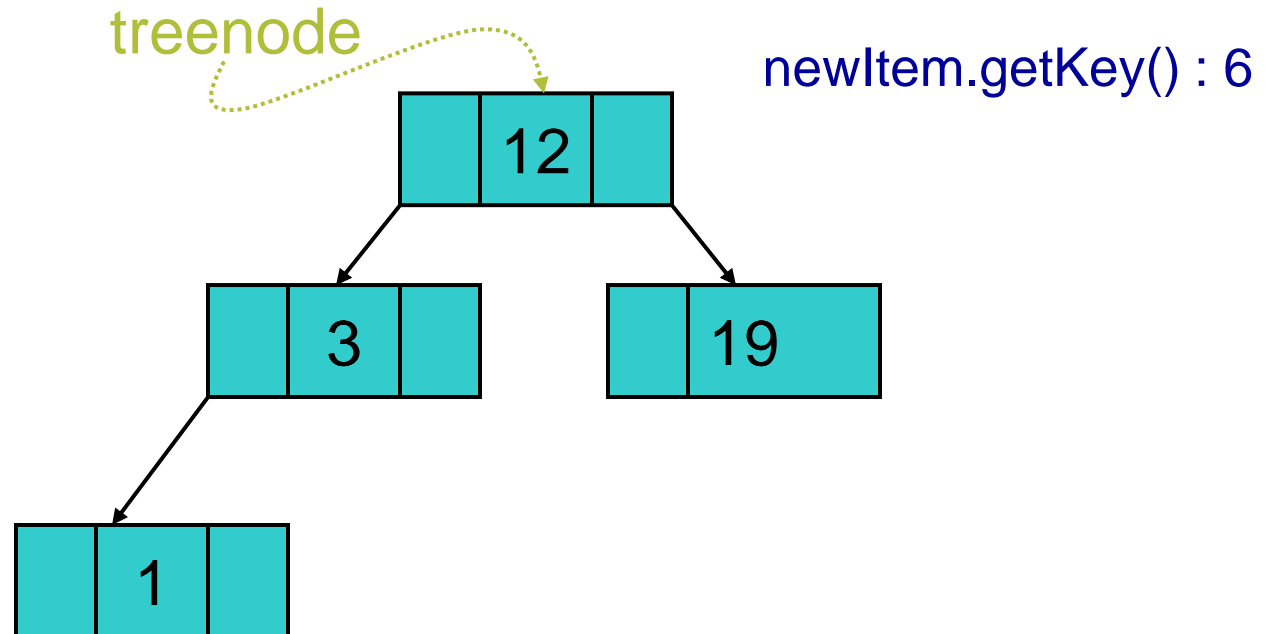

```
insertItem(in treeNode:TreeNode, in newItem:TreeItemType)
   // Inserts newItem into the binary search tree of which
   // treeNode is the root
   if (treeNode is null) {
         create new node with newItem as data
         return new node }
   else if (newItem.getKey() < treeNode.getItem().getKey())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

         treeNode.setRight(insertItem(treeNode.getRight(),new
   Item))
```

How is insertItem used in the code?

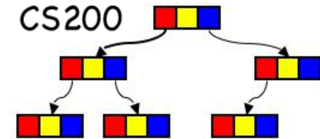
BST – Insert

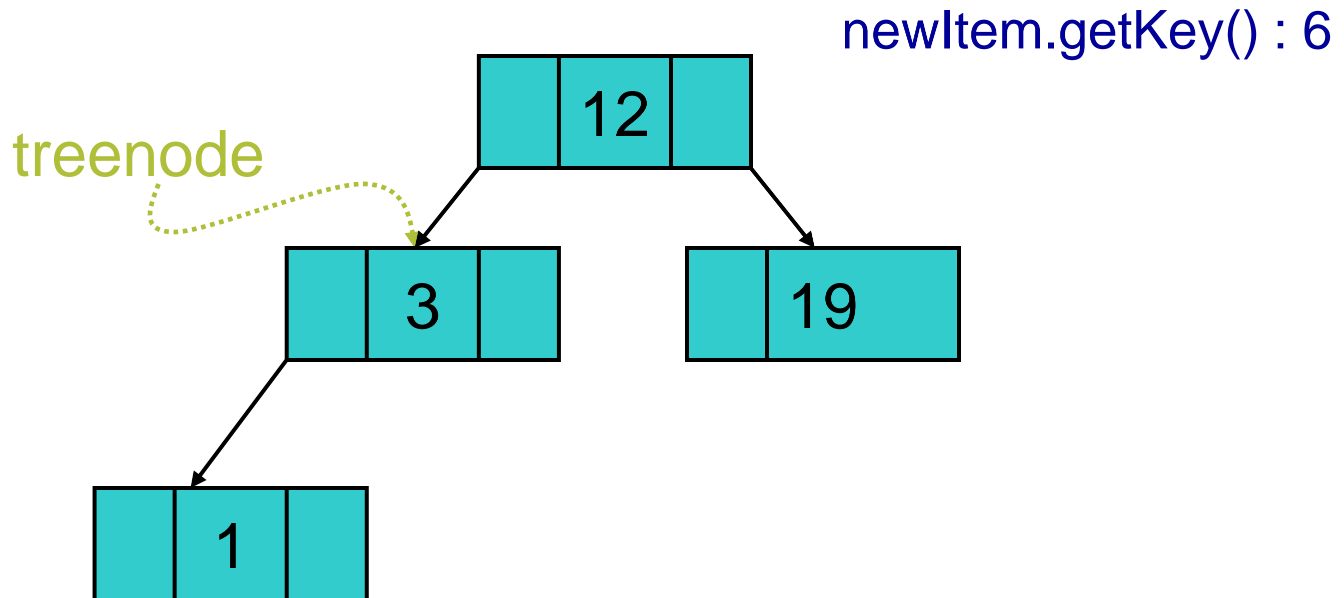





```
if (newItem.getKey() < treeNode.getItem().getKey()) {
        treeNode.setLeft(insertItem(treeNode.getLeft(), newItem)
```

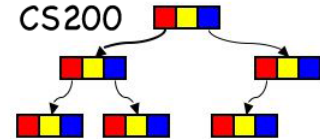
BST – Insert

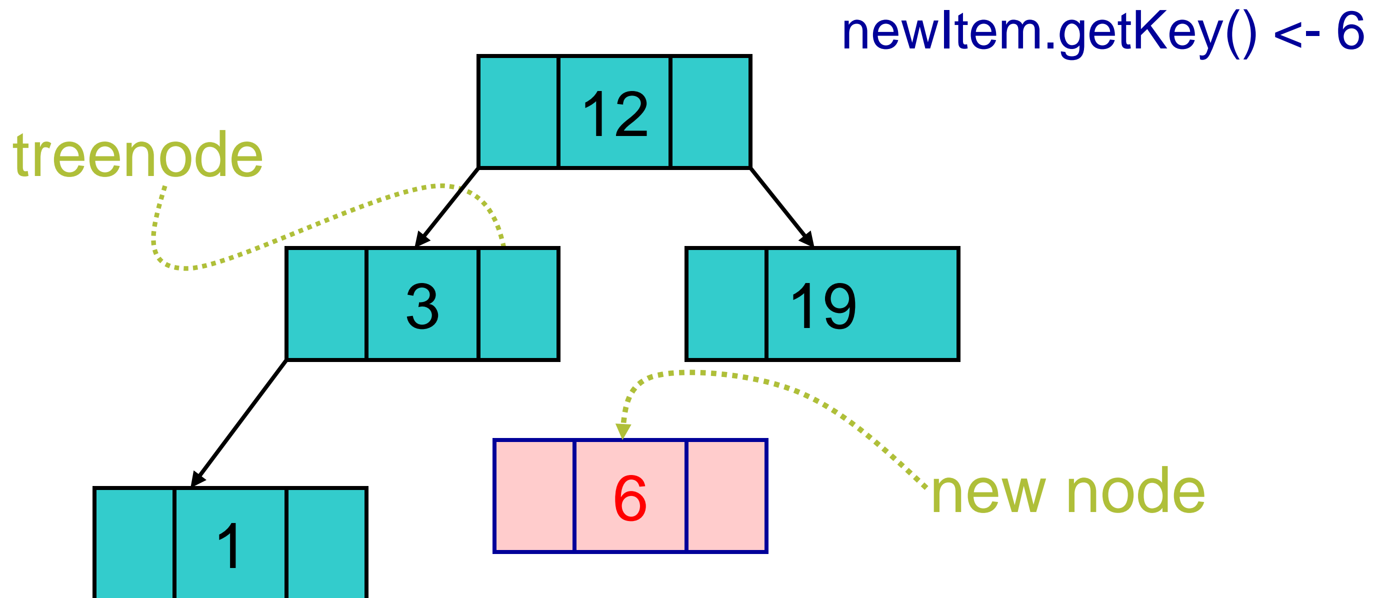





```
else {
        treeNode.setRight(insertItem(treeNode.getRight(),newItem))
```

BST – Insert



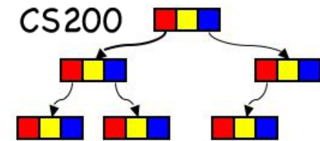


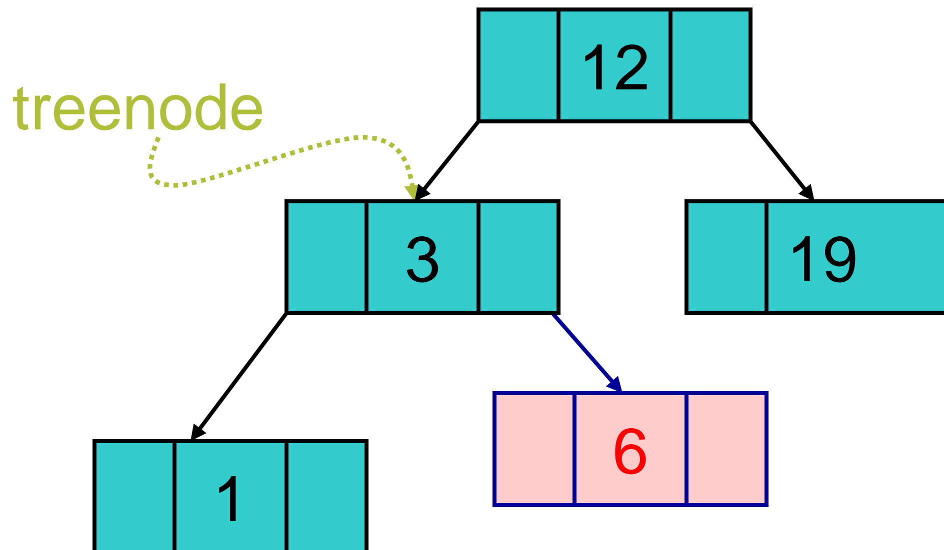
if (treeNode is null) {

create new node with newItem as data

return new node

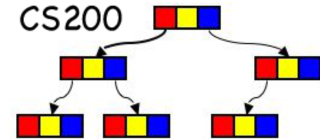
BST – Insert





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

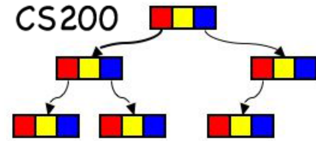
Delete: Cases to Consider



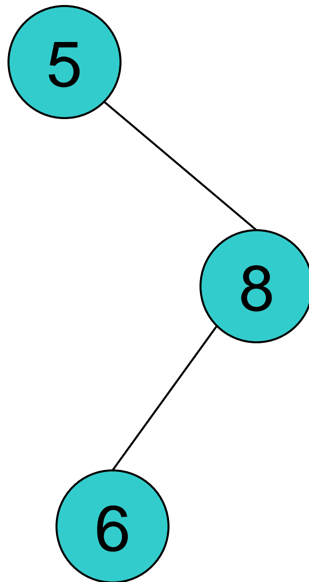
- Delete something that is not there
  - Throw exception
- Delete a leaf
  - Easy, just set link from parent to null
- Delete a node with one child
- Delete a node with two children

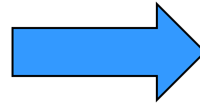
Delete

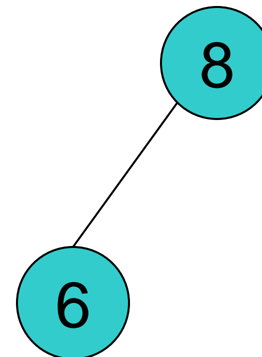
Case 1: one child



delete(5)

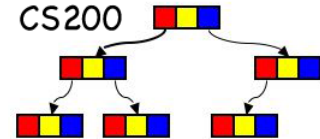






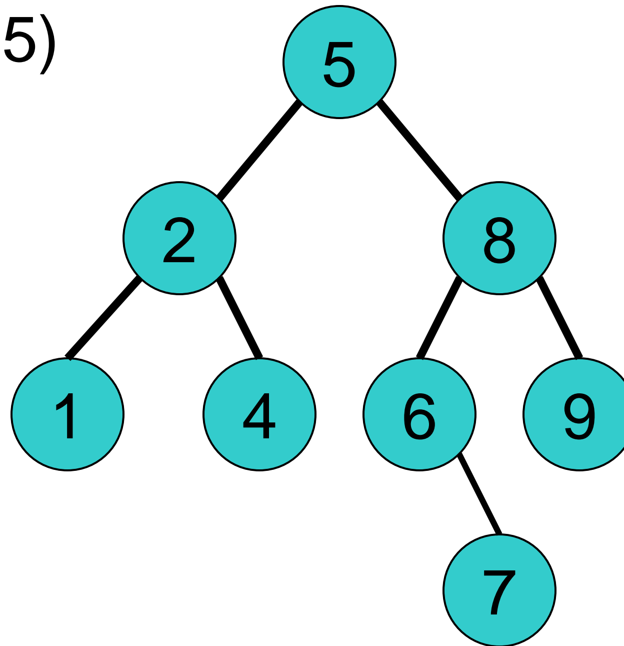
Other child becomes root

Delete



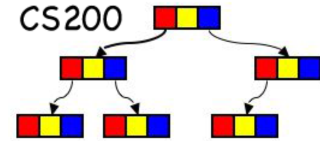
Case 2: two children

delete(5)



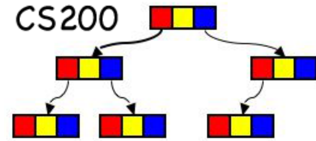
Strategy: replace node with a node that is easier to remove!

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!

Delete

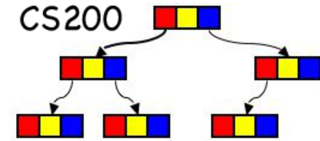


Case 2: two children

delete(5)

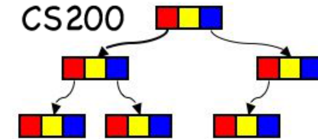
Replace root with its leftmost right descendant and replace that node with its right child, if necessary (an easy delete case).
That node is the inorder successor of the root

Delete Pseudo Code I



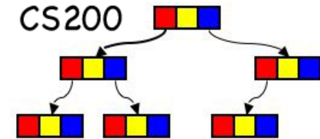

```
deleteItem(in rootNode:TreeNode, in searchKey:KeyType): TreeNode
    if (rootNode is null){ throw TreeException}
    else if (searchKey equals key in rootNode item) { //found it
            newRoot = deleteNode(rootNode, searchKey)
            return newRoot }
    else if (searchKey < key in rootNode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

Delete Pseudo Code II



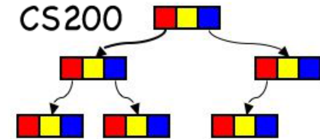

```
deleteNode(in treeNode:TreeNode):TreeNode
    // deletes the item in the node referenced by treeNode
    // returns root of resulting subtree
    if (treeNode is leaf) { return null }
    else if (treeNode has only 1 child c) {
            if (c is left child) { return treeNode.getLeft() }
     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

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
    }
```

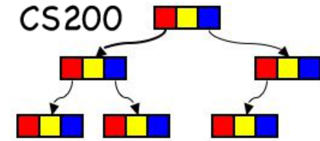
Complexity of BST Operations



	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

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: $O(n^2)$