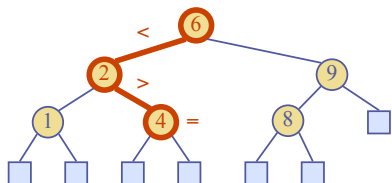


Binary Search Trees



Binary Search Trees

1

Fast Operations

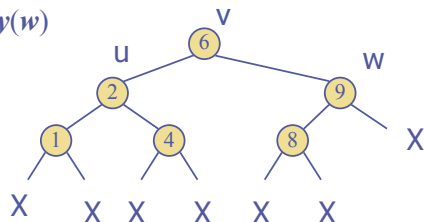
- ◆ What if we could do search, insert, and remove in $O(\log n)$?
 - $\log 1,048,576 = 20$

Binary Search Trees

2

Binary Search Tree (§9.1)

- ◆ A binary search tree is a binary tree storing keys (or key-element pairs) satisfying the following property:
 - Let u , v , and w be three nodes such that u is in the left subtree of v and w is in the right subtree of v . We have $key(u) \leq key(v) \leq key(w)$

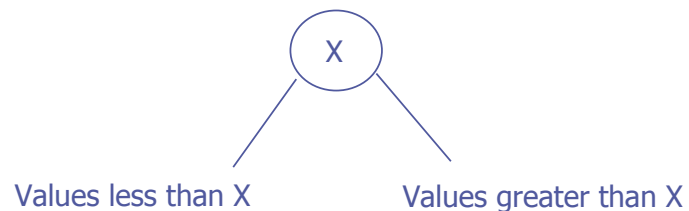


Binary Search Trees

3

Binary Search Tree (§9.1)

- ◆ Property - give a node with a value X , all the values of nodes in the left subtree are smaller than X and all the values of the nodes in the right subtree are larger than X



Binary Search Trees

4

Binary Search Tree (§9.1)

- ◆ External nodes do not store items? (Book says this)
 - Not a good way to look at the tree
 - External nodes store items and children point to NULL
- ◆ An inorder traversal of a binary search trees visits the keys in increasing order

BST Operations

- ◆ makeFromEmpty - initialize a new tree
- ◆ isEmpty - return true if empty, false if not
- ◆ search - return pointer to node in which key is found, otherwise return NULL
- ◆ findMin - return smallest node value
- ◆ findMax - return largest node value

BST Operations

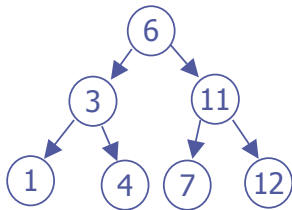
- ◆ insert - insert a new node into the tree maintaining BST property. All inserts are done at a leaf
- ◆ remove - remove a node from the tree maintaining BST property.
- ◆ display - print a tree in an order traversal

Array Implementation of a BST

- ◆ A BST can be implemented with an array
- ◆ Given a node i
 - $\text{parent}(i) = (i - 1)/2$
 - ◆ If $i = 0$, then no parent since root
 - $\text{leftChild}(i) = 2i+1$
 - ◆ If $2i+1 \leq N$, otherwise no child
 - $\text{rightChild}(i) = 2i+2$
 - ◆ If $2i+2 \leq N$, otherwise no child

Array Implementation of a BST

0	1	2	3	4	5	6	7
6	3	11	1	4	7	12	

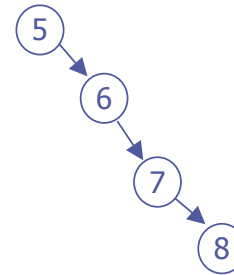


Binary Search Trees

9

Array Implementation of a BST

◆ In class exercise - show the array for the following tree



Binary Search Trees

10

Linked Implementation of a BST

◆ Array disadvantages

- Wasted space
- Not enough space

◆ Linked implementation

- Similar to linked list - size can grow and shrink easily during runtime

Node class

itemtype item

Node left*

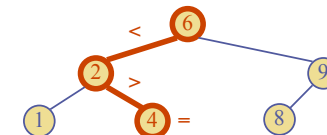
Node right*

Binary Search Trees

11

Search (§9.1.1)

- ◆ To search for a key k , we trace a downward path starting at the root
- ◆ The next node visited depends on the outcome of the comparison of k with the key of the current node
- ◆ If we reach a leaf, the key is not found and we return a null position
- ◆ Example: `find(4)`



Binary Search Trees

12

Search

Recursive implementation of search

```
Node* search ( Node* nodePtr, itemtype key )
    if ( nodePtr == NULL )
        return NULL
    else if ( nodePtr->item == key )
        return nodePtr
    else if ( nodePtr->item > key )
        search( nodePtr->left, key )
    else
        search( nodePtr->right, key )
```

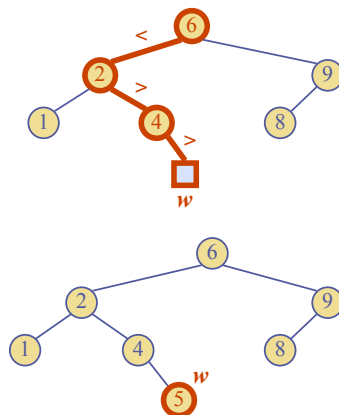
Inorder Traversal

Recursive implementation of inorder traversal

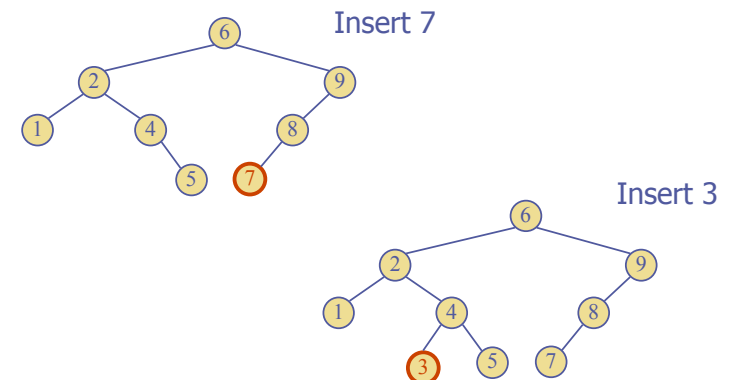
```
void inorder( node* nodePtr )
    if ( nodePtr != NULL )
        inorder( nodePtr->left )
        print node
        inorder( nodePtr->right )
```

Insertion (§9.1.2)

- ◆ To perform operation `insertItem(k, o)`, we search for key `k`
- ◆ Assume `k` is not already in the tree, and let `w` be the leaf reached by the search
- ◆ We insert `k` at node `w`
- ◆ Example: insert 5



Insertion



Insertion

◆ In class exercise - create a BST by inserted the following integers in the given order

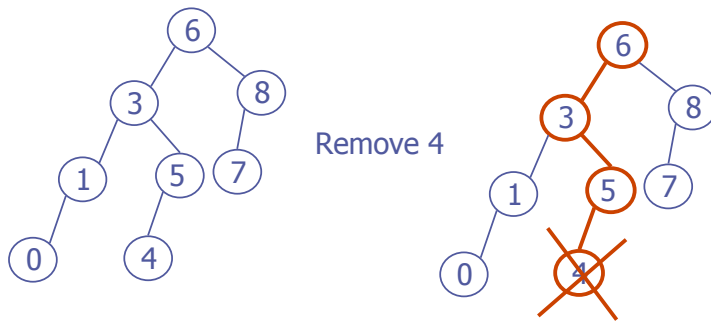
- 6 8 3 5 1 0 7 4

Deletion

◆ Traverse tree and search for node to remove

- Five possible situations
 - ◆ Item not found
 - ◆ Removing a leaf
 - ◆ Removing a node with one child - right only
 - ◆ Removing a node with one child - left only
 - ◆ Removing a node with two children

Deletion - Removing a leaf



Deletion - Removing a node with children

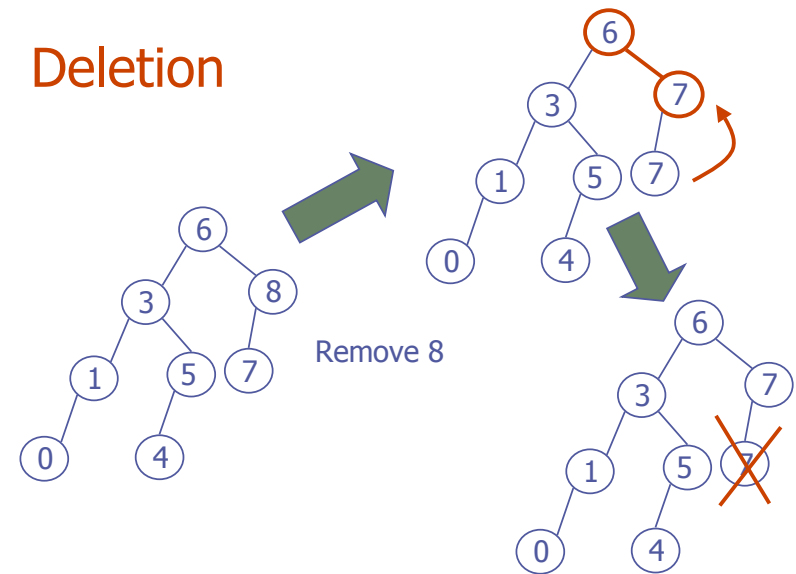
◆ Otherwise the node has children - find replacement node

- If the left child exists
 - ◆ Replace node information with the **largest** value smaller than the value to remove
 - findMax(leftChild)
- Else there is a right child
 - ◆ Replace node information with the **smallest** value larger than value to remove
 - findMin(rightChild)

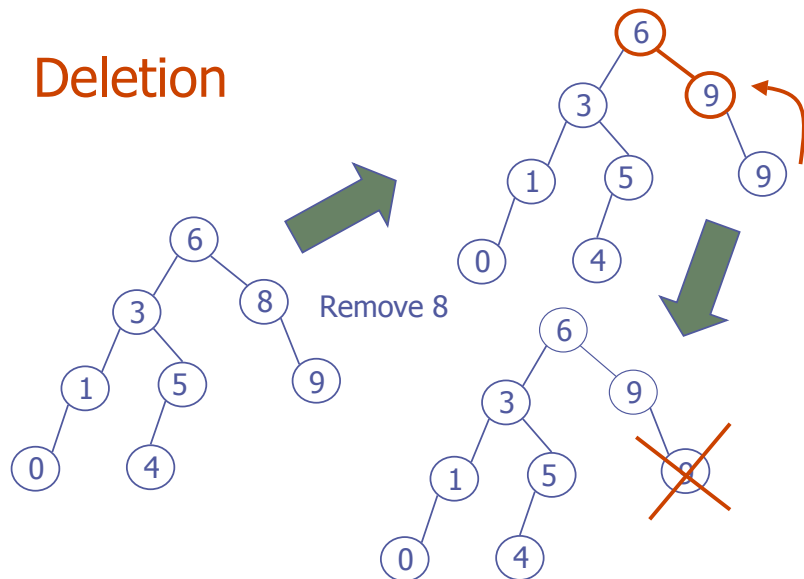
Deletion - Removing a node with children (continued)

- ◆ Splice out replacement node (call remove recursively)
- ◆ Just copy in info of replacement node over the value to remove (overload = or a copy constructor if necessary)
- ◆ Delete replacement node if leaf

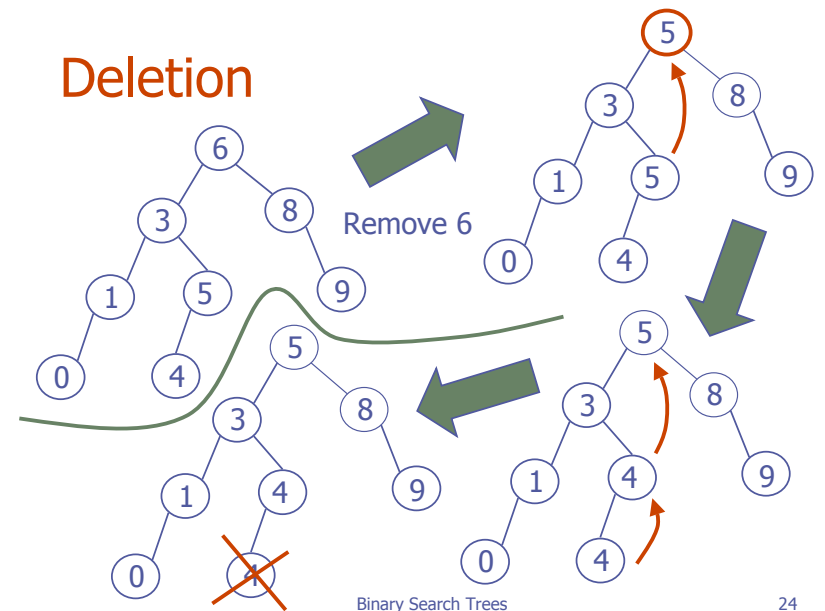
Deletion



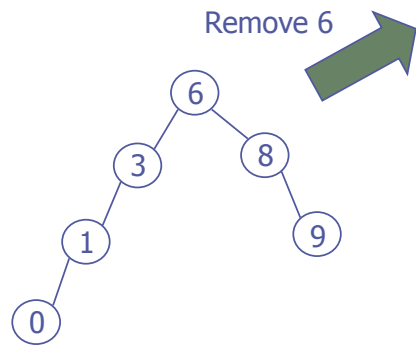
Deletion



Deletion

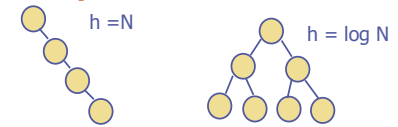


Deletion



Analysis of BST Operations

◆ In class exercise



	Worst Case	Average Case
empty		
search		
findMin		
findMax		
insert		
remove		
display		
makeFromEmpty		

Analysis of BST Operations

- ◆ Given a random ordering of insertions and deletions, the height of the tree will be quite close to $\log n$
- ◆ We will learn later how to ensure the average case running times are also the worst case running times

Treesort

- ◆ Uses a BST to sort records efficiently
 - Use makeFromEmpty
 - ◆ Read in elements and insert in that order into a BST
 - Traverse inorder to read out nodes in ascending order
- ◆ Runtime
 - Average case - $O(N \log N)$
 - Worst case - $O(N^2)$