

Topic 15

The Binary Search Tree ADT

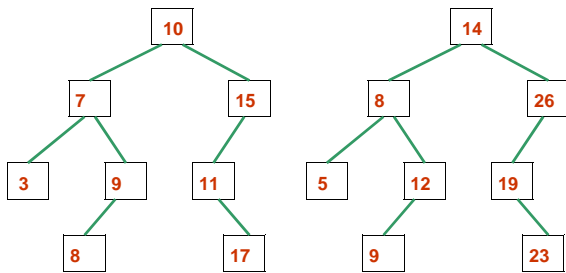
11-1

Binary Search Tree

- A **binary search tree (BST)** is a binary tree with an **ordering** property of its elements, such that the data in any internal node is
 - **Greater than** the data in any node in its left subtree
 - **Less than** the data in any node in its right subtree
- **Note:** this definition does not allow duplicates; some definitions do, in which case we could say "**less than or equal to**"

11-2

Examples: are these Binary Search Trees?



11-3

Discussion

- Observations:
 - What is in the leftmost node?
 - What is in the rightmost node?

11-4

BST Operations

- A binary search tree is a special case of a binary tree
 - So, it has all the operations of a binary tree
- It also has **operations specific to a BST**:
 - **add** an element (requires that the BST property be maintained)
 - **remove** an element (requires that the BST property be maintained)
 - **remove the maximum** element
 - **remove the minimum** element

11-5

Searching in a BST

- Why is it called a binary **search** tree?
 - Data is stored in such a way, that it can be more **efficiently** found than in an ordinary binary tree

11-6

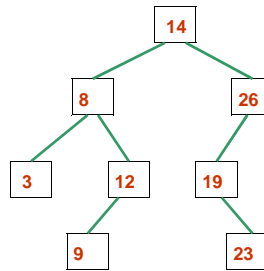
Searching in a BST

Algorithm to search for an item in a BST

- Compare data item to the root of the (sub)tree
- If data item = data at root, found
- If data item < data at root, go to the left; if there is no left child, data item is not in tree
- If data item > data at root, go to the right; if there is no right child, data item is not in tree

11-7

Search Operation – a Recursive Algorithm



To search for a value k ; returns *true* if found or *false* if not found

If the tree is empty, return *false*.

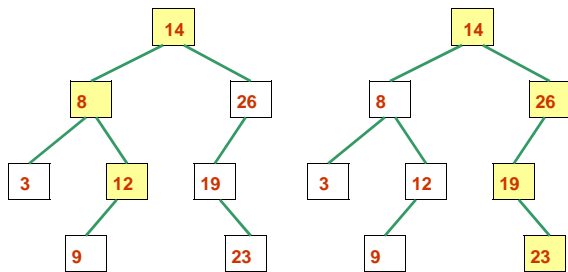
If $k ==$ value at root
return *true*: we're done.

If $k <$ value at root
return result from search for k in the left subtree

Else
return result from search for k in the right subtree.

11-8

Search Operation



Search for 13: visited nodes are coloured yellow; return false when node containing 12 has no right child

Search for 22: return false when node containing 23 has no left child

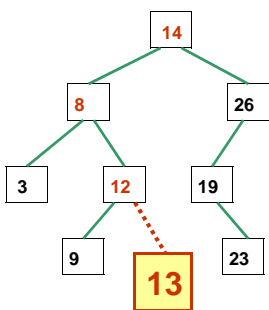
11-9

BST Operations: add

- To **add** an item to a BST:
 - Follow the algorithm for searching, until there is no child
 - Insert at that point
- So, new node will be added as a leaf
- (We are assuming no duplicates allowed)

11-10

Add Operation



To insert 13:
Same nodes are visited as when **searching** for 13.
Instead of returning **false** when the node containing 12 has no right child, build the new node, attach it as the right child of the node containing 12, and return **true**.

11-11

Add Operation – an Algorithm

To insert a value k into a tree, returning *true* if successful and *false* if not

Build a new node for k .

If tree is empty
add new node as root node, return *true*.

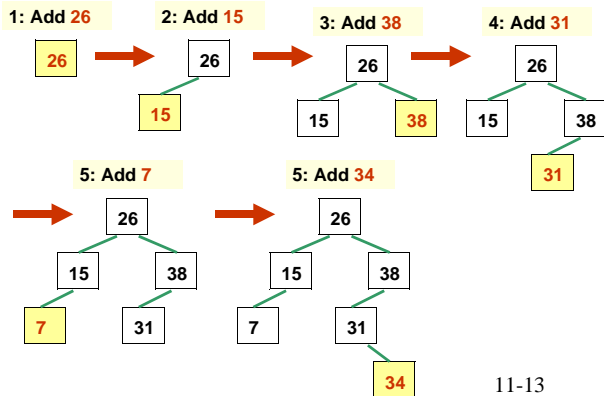
If $k ==$ value at root
return *false* (no duplicates allowed).

If $k <$ value at root
If root has no left child
add new node as left child of root, return *true*
Else insert k into left subtree of root.

If $k >$ value at root
If root has no right child
add new node as right child of root, return *true*
Else insert k into the right subtree of root.

11-12

Example: Adding Elements to a BST

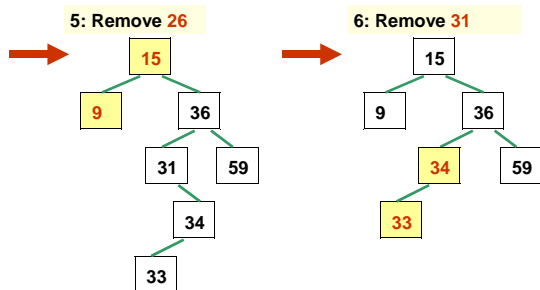
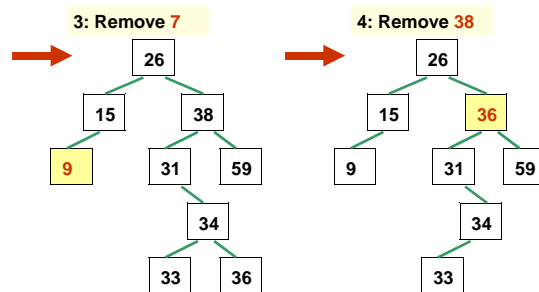
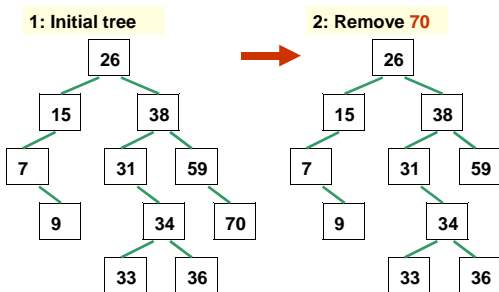


BST Operations: Remove

- **Case 1:** value to be removed is in a *leaf node*
 - Node can be removed and the tree needs no further rearrangement
- **Case 2:** value to be removed is in an *interior node*
 - Why can't we just change the link from its parent node to a successor node?
 - We can replace the node with its *inorder predecessor* (or successor)
 - Complex, and we will not implement this

11-14

Example: Removing BST Elements

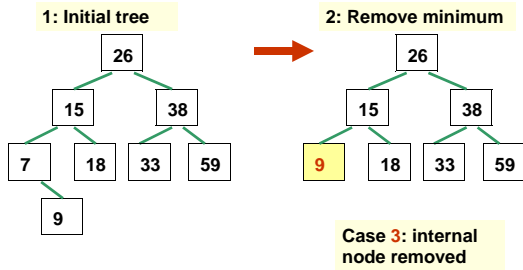


BST Operations: Remove Minimum

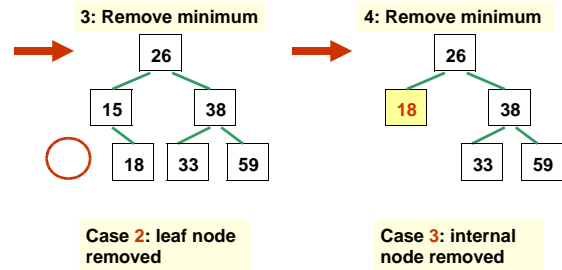
- Recall that *leftmost node* contains the minimum element
- Three cases:
 - 1) *root has no left child* (so, root is minimum)
 - its right child becomes the root
 - 2) *leftmost node is a leaf*
 - set its parent's left child to null
 - 3) *leftmost node is internal*
 - the right child of the node to be removed becomes the parent's left child

11-18

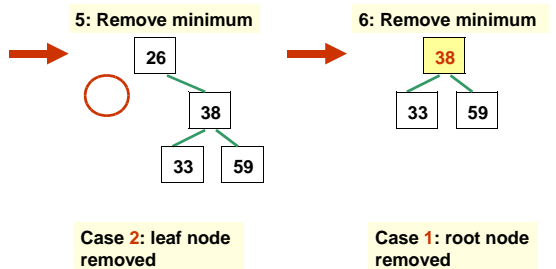
Example: Removing Minimum BST Element



11-19



11-20



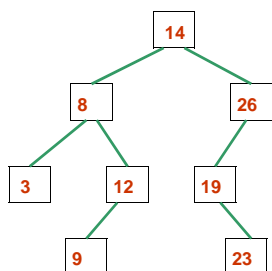
11-21

Binary Search Tree Traversals

- Consider the traversals of a binary search tree: preorder, inorder, postorder, level-order
- Try the traversals on the example on the next page
 - Is there anything special about the *order of the data* in the BST, for each traversal?
- **Question:** what if we wanted to visit the nodes in *descending* order?

11-22

Binary Search Tree Traversals



Try these traversals:

- preorder
- inorder
- postorder
- level-order

11-23

Binary Search Tree ADT

- A BST is just a binary tree with the ordering property imposed on all nodes in the tree
- So, we can define the **BinarySearchTreeADT** interface as an *extension* of the **BinaryTreeADT** interface

11-24

```

public interface BinarySearchTreeADT<T> extends
    BinaryTreeADT<T> {
    public void addElement (T element);

    public T removeElement (T targetElement);

    public void removeAllOccurrences (T targetElement);

    public T removeMin( );

    public T removeMax( );

    public T findMin( );

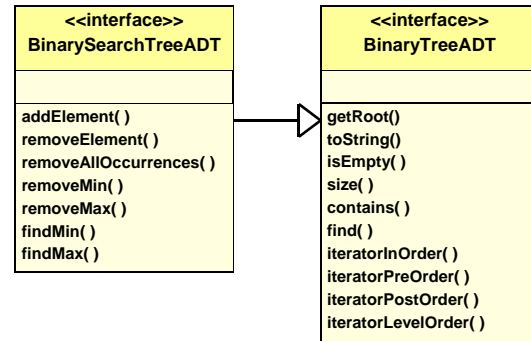
    public T findMax( );
}

```

The BinarySearchTreeADT interface

11-25

UML Description of BinarySearchTreeADT



11-26

Implementing BSTs using Links

- See *LinkedBinarySearchTree.java*
 - Constructors: use `super()`
 - **addElement** method
 - (does not implement our recursive algorithm of p.12; also, allows duplicates)
 - note the use of **Comparable**: so that we can use `compareTo` method to know where to add the new node
 - **removeMin** method
 - essentially implements our algorithm of p. 18

11-27

Implementing BSTs using Links

- The special thing about a Binary Search Tree is that **finding a specific element is efficient!**
 - So, *LinkedBinarySearchTree* has a **find** method that **overrides** the `find` method of the parent class *LinkedBinaryTree*
 - It only has to search the appropriate side of the tree
 - It uses a recursive helper method `findAgain`
 - Note that it does not have a **contains** method that overrides the `contains` of *LinkedBinaryTree* – why not?
 - It doesn't need to, because `contains` just calls `find`

11-28

Using Binary Search Trees: Implementing Ordered Lists

- A BST can be used to provide *efficient* implementations of other collections!
- We will examine an implementation of an **Ordered List ADT** as a *binary search tree*
- Our implementation is called *BinarySearchTreeList.java* (naming convention same as before: this is a BST implementation of a List)

11-29

Using BST to Implement Ordered List

- *BinarySearchTreeList* **implements** **OrderedListADT**
 - Which extends **ListADT**
 - So it also implements ListADT
 - So, what operations do we need to *implement*?
 - `add`
 - `removeFirst`, `removeLast`, `remove`, `first`, `last`, `contains`, `isEmpty`, `size`, `iterator`, `toString`
 - But, for which operations do we actually need to write code? ...

11-30

Using BST to Implement Ordered List

- **BinarySearchTreeList** *extends* our binary search tree class **LinkedBinarySearchTree**
 - Which extends **LinkedBinaryTree**
 - So, what operations have we *inherited* ?
 - addElement, removeElement, removeMin, removeMax, findMin, findMax, find
 - getRoot, isEmpty, size, contains, find, toString, iteratorInOrder, iteratorPreOrder, iteratorPostOrder, iteratorLevelOrder

11-31

Discussion

- First, let's consider some of the methods of the **List ADT** that we do *not* need to write code for:
 - **contains** method: we can just *use* the one from the **LinkedBinaryTree** class
 - What about the methods
 - **isEmpty**
 - **size**
 - **toString**

11-32

Discussion

- To implement the following methods of the **OrderedListADT**, we can *call* the appropriate methods of the **LinkedBinarySearchTree** class (*fill in the missing ones*)
 - **add** call addElement
 - **removeFirst** removeMin
 - **removeLast**
 - **remove**
 - **first**
 - **last**
 - **iterator**

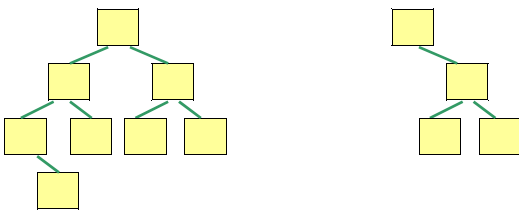
11-33

Balanced Trees

- **Our definition:** a **balanced tree** has the property that, for any node in the tree, the height of its left and right subtrees can *differ by at most 1*
 - Note that conventionally the height of an empty subtree is **-1**

11-34

Balanced Trees



Which of these trees is a balanced tree?

11-35

Analysis of BST Implementation

- We will now compare the **linked list** implementation of an ordered list with its **BST** implementation, making the following important assumptions:
 - The BST is a **balanced** tree
 - The maximum level of any node is **$\log_2(n)$** , where **n** is the number of elements stored in the tree

11-36

Analysis of Ordered List Implementations: Linked List vs. Balanced BST

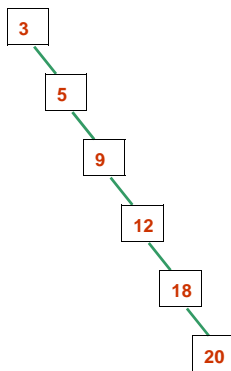
Operation	LinkedList	BinarySearchTreeList
removeFirst	$O(1)$	$O(\log_2 n)$
removeLast	$O(n)$	$O(\log_2 n)$
remove	$O(n)$	$O(\log_2 n)$ <small>*but may cause tree to become unbalanced</small>
first	$O(1)$	$O(\log_2 n)$
last	$O(n)$	$O(\log_2 n)$
contains	$O(n)$	$O(\log_2 n)$
isEmpty	$O(1)$	$O(1)$
size	$O(1)$	$O(1)$
add	$O(n)$	$O(\log_2 n)$ *

Discussion

- Why is our balance assumption so important?
 - Look at what happens if we insert the following numbers in this order without rebalancing the tree:

3 5 9 12 18 20

11-38



11-39

Degenerate Binary Trees

- The resulting tree is called a **degenerate** binary tree
 - Note that it looks more like a linked list than a tree!
 - But it is actually less efficient than a linked list (**Why?**)

11-40

Degenerate Binary Trees

- Degenerate BSTs are far less efficient than balanced BSTs
 - Consider the worst case time complexity for the **add** operation:
 - $O(n)$ for degenerate tree
 - $O(\log_2 n)$ for balanced tree

11-41

Balancing Binary Trees

- There are many approaches to balancing binary trees
 - But they will not be discussed in this course ...

11-42