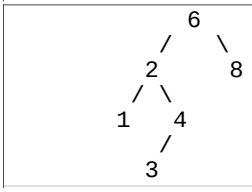
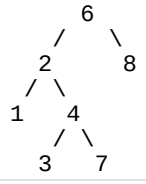
Lecture 8
10/10/2003
Binary Search Trees

Binary Search Trees

Definitions

A *binary tree* is a tree in which each node has at most two children. A *binary search tree* is a binar that for all nodes n_i, the following two properties hold:

1. All elements in the left subtree of *i* are smaller than the element in node *i*
2. All elements in the right subtree of *i* are greater than the element in node *i*

This is a binary search tree:	This is not:
	
	Everything would be ok, except that 7 cannot be in the left sub-tree of 6

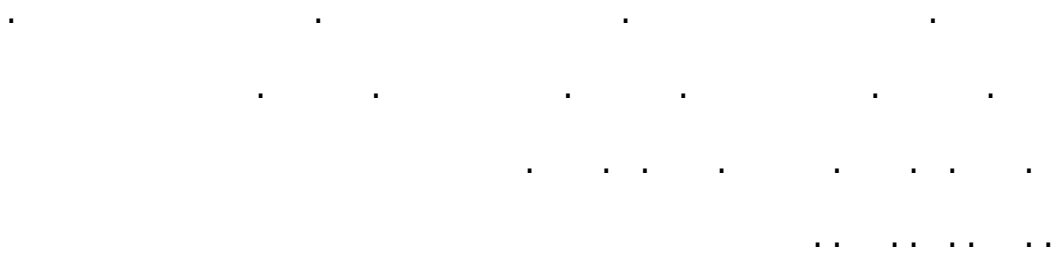
The generic tree traversals apply here.

Complexity

Before we begin to think about algorithms on binary search trees, let's think a little bit about the of a BST. When talking about BST we often talk about the number of nodes (*n*) and the height of the

Recall that the height of a tree is the height of its root, which is the length of the longest path from

Lets consider the following set of perfectly balanced BSTs.



What is the relationship between the number of nodes and the height of the tree?

n	height
1	1
3	2
7	3

n	height
15	4
n	log(*n*+1)

This is a "best-case" scenario, *i.e.*, a perfectly balanced tree. What is the "worst-case" scenario, th a tree with *n* nodes, what is the largest possible height? (It's *n*.)

So, if we need to traverse a tree from top to bottom, the best we can hope for is O(log *n*) anc possible case is O(*n*), *i.e.*, a list.

This analysis will become important as we begin implementing algorithms on BSTs. To simplify will assume a well-balanced tree and later we will discuss how to guarantee a balancing property.

Representation

First, how might we represent a binary tree? We could of course use the previous tree data str given this simplified structure, it might make more sense to use something simpler and more compact

```
class BinaryNode {
    private Object element;
    private BinaryNode left;
    private BinaryNode right;
}

class Tree {
    private BinaryNode root;
}
```

Some functions we might like to write on a BST are not terribly different than those on a list:

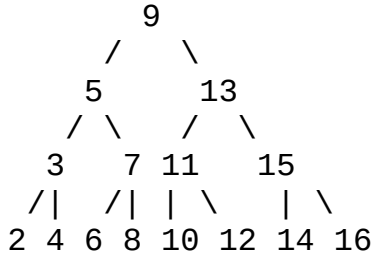
- Find
- Insert
- Remove
- Child (analogous to next)
- Parent (analogous to previous)

Note that `Find`K^{th} doesn't really make sense here since we don't have a linear ordering of our data choose to implement it with respect to one of our tree traversals.

Operations

Find

Do we have to search the entire tree to find an element in a BST? No, the inherent ordering can be Consider the following tree:



If we are looking for the element 8 in this tree, we naturally start at the root. We immediately know 8<9, that we need not search the entire right subtree. We then notice that 8>5 so we can ignore 5's l At node 7, we go right and find 8. Note that in this tree of 15 elements, we only had to make four co *i.e.*, log (*n*+1) comparisons.

That is, `Find` runs in O(log *n*)! But, this assumes a well-balanced tree. What is the worst possible we don't have a balanced tree?

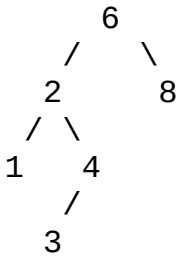
```
2 - 3 - 4 - 5 - 6 - ...
```

O(*n*) - that is, a tree reduces to a list if everyone has exactly one child.

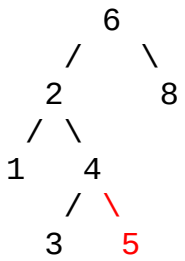
So we can only really say that find is O(log *n*) if the tree is well-balanced. We will see shortly that efficient way to guarantee this with an ordinary BST, but we will be able to guarantee balance advance types of trees.

Insert

We have yet to describe how to build a tree. Unlike a list, we can't simply insert an element into a position in a tree as we need to make sure to not violate the ordering property of BST. For exampl inserting 5 into the following tree:



Where should it go? Well, without displacing any of the existing elements, there is only one place child of node 4 - why?



Let's sketch the code:

```
BinaryNode Insert( x, T ) {
    if( T == null )
        T = new BinaryNode(x, null, null);
    else if( x < T.element )
        T.left = Insert( x, T.left );
    else
        T.right = Insert( x, T.right );

    return( T );
}
```

Note that new element is always inserted as leaf.

Example 1

Let *T* be the tree:

```
     6
   /   \
  2     8
```

```
Insert( 4, T ) → T.left = Insert( 4, T.left )
Insert( 4, T.left ) → T.left.right = Insert( 4, T.left.right )
Insert( 4, T.left.right ) → T.left.right = new BinaryNode(4)
```

Delete

This is a little tricky; if we delete a node from the middle of tree, we potentially lose all the childı other hand, if we delete a leaf, this is not a problem. So let's consider three possible cases for delet BST.

Case 1. Node to be deleted has no children. Easy: just delete it. In our example tree, `Delete`(5) – = `null`.

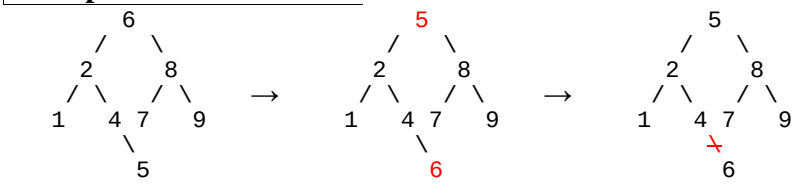
Case 2. Node to be deleted has only one child. This is also easy, connect parent of node to be dele *e.g.*, `Delete`(4) → 2.`right` = 5.

Case 3. Node to be deleted has two children. This is a bit trickier. For example, if we wanted t what do we do? We need to promote someone to take over the parent position. Which one? We options:

- the max of the left subtree - 5
- the min of the right subtree - 7

Why only these two options? The new parent must be larger than everything on the left, but sr everything on the right. What do we know about the min/max of the right/left subtree? They must i.e., case 1 applies. So case 3 is a two-step operation: swap the node to be deleted with one of the tv then try to delete it again.

Example 2: Delete node 6.



Java™ Code

Let's look at the Java™ code that you may use for homework #3.

```
public interface Tree
{
    public List preorder();
    public List postorder();
    public List inorder();
    public void Insert(Object x);
```

```
    public void Remove(Object x);
}


public interface BST extends Tree
{
    public void Insert(Comparable x);
    public void Remove(Comparable x);
}


public class BinarySearchTree implements BST
{
    private BinaryNode root;

    public BinarySearchTree() { root = null; }

    public void Insert( Object x ) { return; /* silently fail */ }
    public void Remove( Object x ) { return; /* silently fail */ }

    public void Insert( Comparable x ) { root = this.insert( x, root
    public void Remove( Comparable x ) { root = this.remove( x, root
    public Comparable Find( Comparable x ) {
        BinaryNode rv = find( x, root );
        if(rv == null)
            return null;
        else
            return find(x, root).element;
    }

    private BinaryNode insert( Comparable x, BinaryNode t )
    {
        if( t == null )
            t = new BinaryNode(x);
        else if( x.compareTo( t.element ) < 0 )
            t.leftChild = insert( x, t.leftChild );
        else if( x.compareTo( t.element ) > 0 )
            t.rightChild = insert( x, t.rightChild);
        else
            ;  // x.compareTo(t.element) == 0, so x is already in the tree.
        return ( t );
    }


    private BinaryNode remove( Comparable x, BinaryNode t )
    {
        if( t == null )
            return t;
        else if( x.compareTo( t.element ) < 0 )
            t.leftChild = remove( x, t.leftChild );
        else if( x.compareTo( t.element ) > 0 )
            t.rightChild = remove( x, t.rightChild );
        else if( t.leftChild != null && t.rightChild != null )
        {
            // Arbitrary choice. Could also swap with
```

```
            // findMax( t.leftChild )
            t.element = findMin( t.rightChild ).element;
            t.rightChild = remove( t.element, t.rightChild );
        }
        else
            t = ( t.leftChild != null ) ? t.leftChild : t.rightChild;

        return t;
    }

    private BinaryNode findMin( BinaryNode t )
    {
        if ( t == null )
            return null;
        else
        {
            while ( t.leftChild != null )
                t = t.leftChild;
            return t;
        }
    }

    private BinaryNode find( Comparable x, BinaryNode t)
    {
        if ( t ==  null )
            return null;
        else if ( x.compareTo( t.element ) < 0 )
            return find( x, t.leftChild );
        else if ( x.compareTo( t.element ) > 0 )
            return find( x, t.rightChild );
        else // Found it
            return t;
    }

    public boolean isEmpty()
    {
        return (root == null);
    }



   /* Not the ideal way to do this,
    * but we can use our Queue data structure
    * to enqueue elements as we traverse
    * the tree, the return the internal linked
    * list that is built.
    */
    public List inorder()
    {
        return inorder(root, new QueueList() ).getList();
    }

    public List preorder()
    {
```

```
        return preorder(root, new QueueList() ).getList();
    }

    public List postorder()
    {
        return postorder(root, new QueueList() ).getList();
    }

    private QueueList inorder( BinaryNode t, QueueList q )
    {
        if ( t !=  null )
        {
            q = inorder( t.leftChild, q );
            q.Enqueue( t.element );
            q = inorder( t.rightChild, q );
        }
        return q;
    }

    private QueueList preorder( BinaryNode t, QueueList q )
    {
        if ( t != null )
        {
            q.Enqueue( t.element );
            q = preorder( t.leftChild, q );
            q = preorder( t.rightChild, q );
        }
        return q;
    }

    private QueueList postorder( BinaryNode t, QueueList q )
    {
        if ( t != null )
        {
            q = postorder( t.leftChild, q );
            q = postorder( t.rightChild, q );
            q.Enqueue( t.element );
        }
        return q;
    }


    public static void main(String[] args)
    {
        BinarySearchTree bst = new BinarySearchTree();

        bst.Insert(new Integer(5));
        bst.Insert(new Integer(2));
        bst.Insert(new Integer(8));
        bst.Insert(new Integer(3));
        bst.Insert(new Integer(7));
        bst.Insert(new Integer(9));
        bst.Insert(new Integer(6));
        bst.Insert(new Integer(1));
```

```
        bst.Insert(new Integer(4));

        bst.Remove(new Integer(4));
        bst.Remove(new Integer(7));
        bst.Remove(new Integer(8));

        LinkedList l = (LinkedList)(bst.preorder());
        LinkedListItr p = l.first();
        int i;

        System.out.print("Preorder: ");
        while(!p.isPastEnd())
        {
            i = ((Integer)p.retrieve()).intValue();
            System.out.print(i + " ");
            p.advance();
        }
        System.out.println();

        l = (LinkedList)(bst.postorder());
        p = l.first();
        System.out.print("Postorder: ");
        while(!p.isPastEnd())
        {
            i = ((Integer)p.retrieve()).intValue();
            System.out.print(i + " ");
            p.advance();
        }
        System.out.println();

        l = (LinkedList)(bst.inorder());
        p = l.first();
        System.out.print("Inorder: ");
        while(!p.isPastEnd())
        {
            i = ((Integer)p.retrieve()).intValue();
            System.out.print(i + " ");
            p.advance();
        }
        System.out.println();


    }

}


class BinaryNode
{
    Comparable element;
    BinaryNode leftChild;
    BinaryNode rightChild;

    public BinaryNode(Comparable x) {
        this(x, null, null);
```

```
        }

        public BinaryNode(Comparable x, BinaryNode l, BinaryNode r)
        {
            element = x;
            leftChild = l;
            rightChild = r;
        }
    }
```