Binary Search Trees (BST)

- Binary Tree Property:

  No node can have more than two children.

- Binary Search Tree Property:

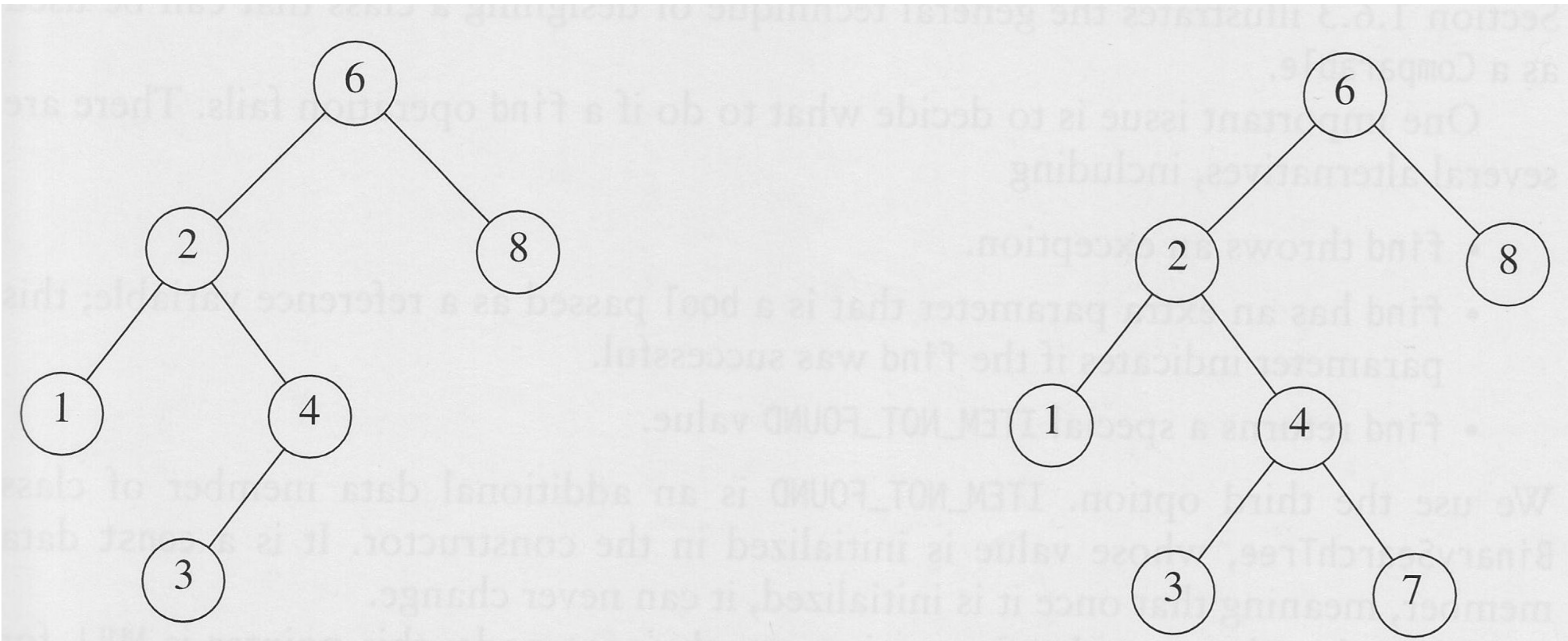
  For every node X in the tree:

  (i) the values of all the items in its *left* subtree are *smaller* than the item in X, and

  (ii) the values of all items in its *right* subtree are *larger* than the item in X.

- All elements in the tree can be ordered in some consistent manner.

Examples



BST　　　　　　　　　　　　Binary Tree

BinarySearchTree Class

```
template <typename Comparable>
class BinarySearchTree
{
    public:
            BinarySearchTree( );
            BinarySearchTree( const BinarySearchTree & rhs );
            ~BinarySearchTree( );

            const Comparable & findMin( ) const;
            const Comparable & findMax( ) const;
            bool contains( const Comparable & x ) const;
            bool isEmpty( ) const;
            void printTree( ) const;

            void makeEmpty( );
            void insert( const Comparable & x );
            void remove( const Comparable & x );

            const BinarySearchTree & operator=( const BinarySearchTree & rhs );
```

BinarySearchTree Class

```
private:
        struct BinaryNode
        {
          Comparable element;
          BinaryNode *left;
          BinaryNode *right;

          BinaryNode( const Comparable & theElement,
                      BinaryNode *lt, BinaryNode *rt )
               : element( theElement ), left( lt ), right( rt ) { }
        };
```

BinarySearchTree Class

```
private:
    BinaryNode *root;

    void insert( const Comparable & x, BinaryNode * & t );
    void remove( const Comparable & x, BinaryNode * & t );
                //pointer variable passed using call by reference

    BinaryNode * findMin( BinaryNode *t ) const;
    BinaryNode * findMax( BinaryNode *t ) const;
    bool contains( const Comparable & x, BinaryNode *t ) const;

    void makeEmpty( BinaryNode * & t );
    void printTree( BinaryNode *t ) const;
    BinaryNode * clone( BinaryNode *t ) const;
};
```

contains Method (public)

```
// Returns true if x is found in the tree.

bool contains( const Comparable & x ) const
{
   return ( contains( x, root ) );
}
```

contains Method (internal)

```
/**
* Internal method to find an item in a subtree.
* x is item to search for.
* t is the node that roots the tree.
*/

bool contains( const Comparable & x, BinaryNode *t ) const
{
    if( t == NULL )
            return NULL;
    else if( x < t->element )
            return contains( x, t->left );
    else if( t->element < x )
            return contains( x, t->right );      in average T(N) = O(log N)
    else
            return true; // Match
}
```

findMin Method (internal)

```
/**
* Internal method to find the smallest item in a subtree t.
* Return node containing the smallest item.
*/

BinaryNode * findMin( BinaryNode *t ) const
{
   if( t == NULL )
            return NULL;
   if( t->left == NULL )
            return t;
   return findMin( t->left );                T(N) =?
}
```

findMax Method (internal)

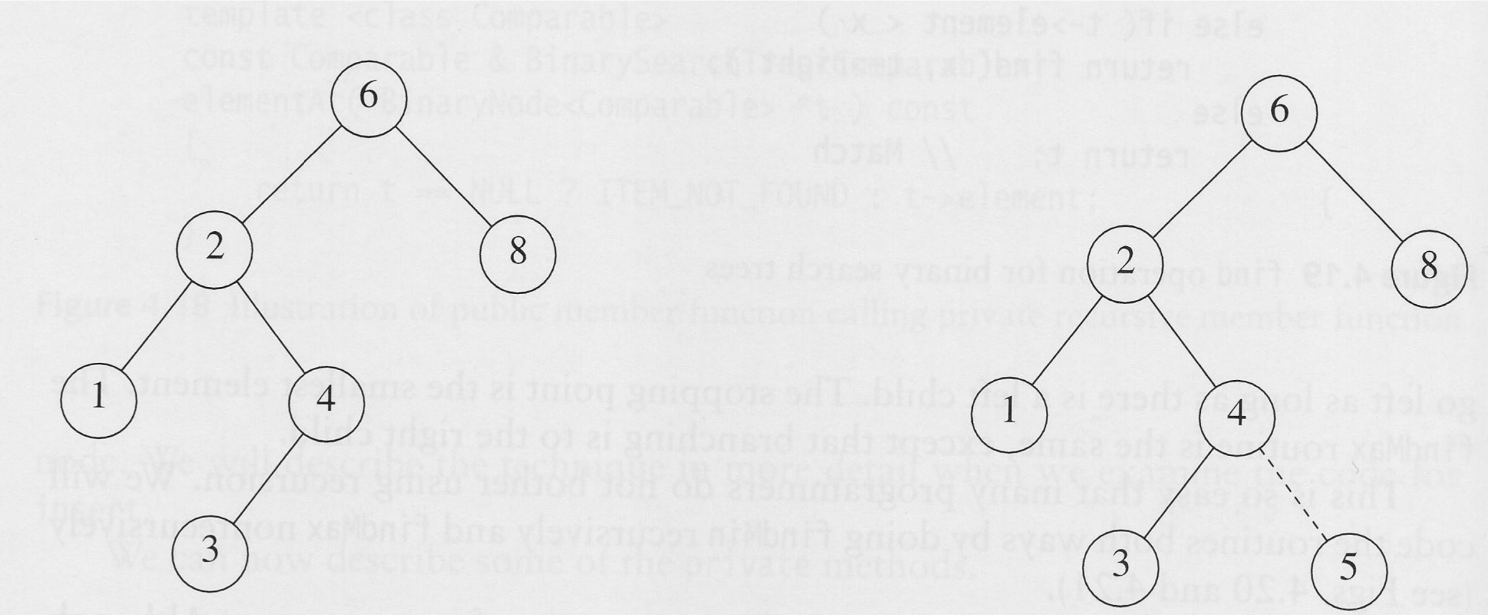
```
/**
* Internal method to find the largest item in a subtree t.
* Return node containing the largest item.
*/
BinaryNode * findMax( BinaryNode *t ) const
{
    if( t != NULL )
            while( t->right != NULL )
                    t = t->right;
    return t;
}
```

T(N) = ?

It is safe to change t because we are working with a copy of the pointer

Insert Operation

Insert 5



insert Method

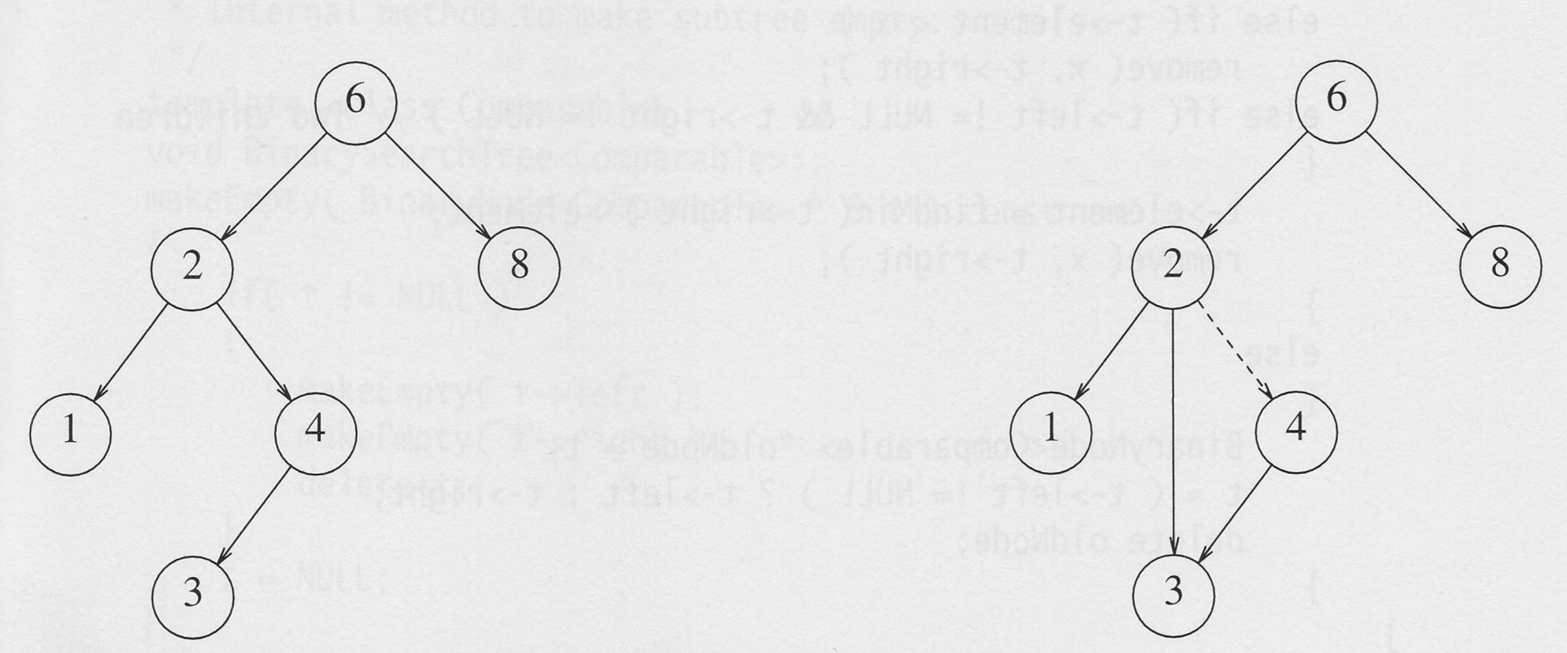
```
/**
* Internal method to insert into a subtree.
* x is the item to insert.
* t is the node that roots the tree.
* Set the new root.
*/

void insert( const Comparable & x, BinaryNode * & t ) const
{
    if( t == NULL )
            t = new BinaryNode( x, NULL, NULL );
    else if( x < t->element )
            insert( x, t->left );
    else if( t->element < x )
            insert( x, t->right );
    else ; // Duplicate; do nothing
}
```

T(N) = ?

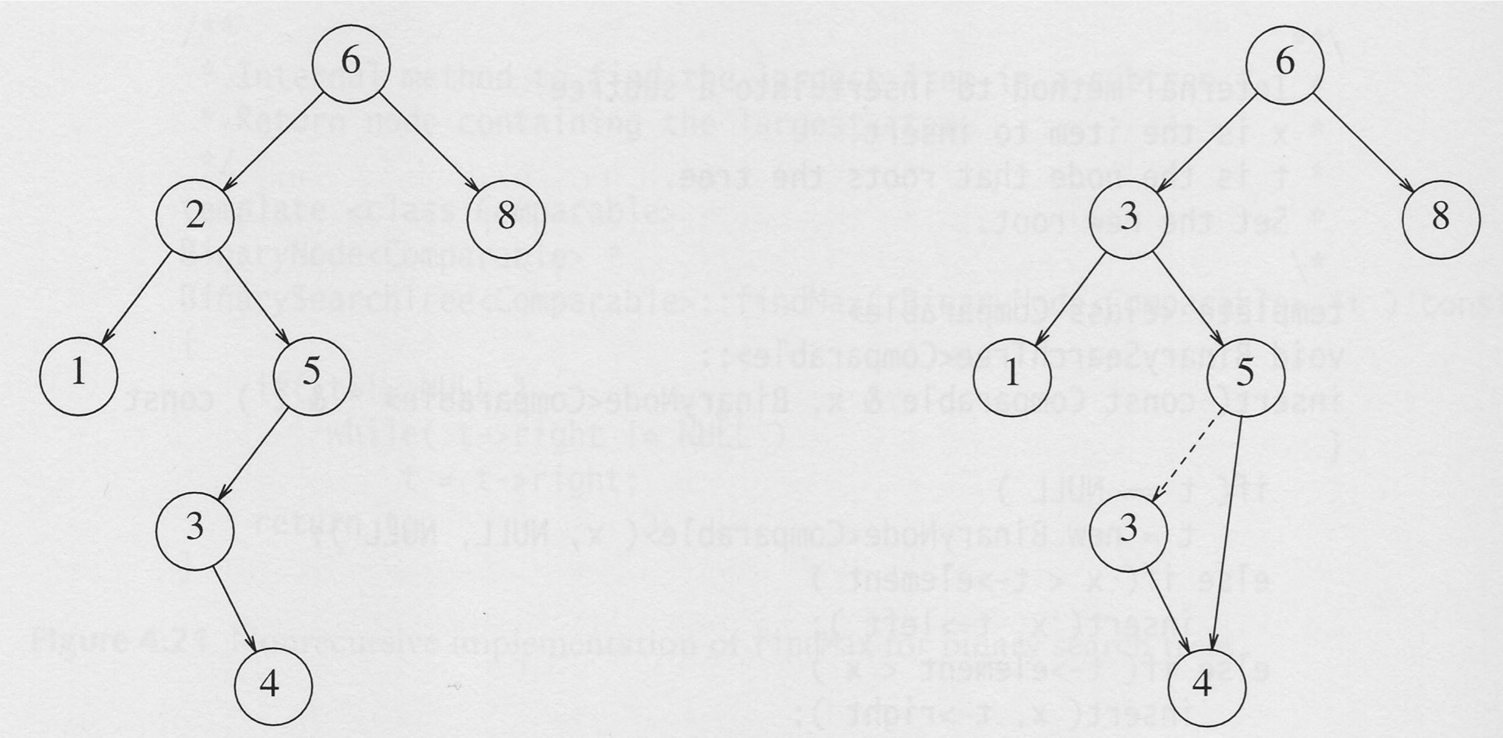
remove Operation

Delete a node with one child => remove 4



remove Operation

Delete a node with two children => remove 2



remove Method

```
/**
* Internal method to remove from a subtree.
* x is the item to remove; t is the node that roots the tree. Set the new root.
*/

void remove( const Comparable & x, BinaryNode * & t ) const
{
    if( t == NULL )
            return; // Item not found; do nothing
    if( x < t->element )
            remove( x, t->left );
    else if( t->element < x )
            remove( x, t->right );
    else if( t->left != NULL && t->right != NULL ) // Two children
    {
            t->element = findMin( t->right )->element;
            remove( t->element, t->right );
    }
    else
    {
            BinaryNode *oldNode = t;
            t = ( t->left != NULL ) ? t->left : t->right;
            delete oldNode;
    }
}
```

T(N) = ?

Destructor and makeEmpty

```
/**
* Destructor for the tree.
*/
~BinarySearchTree( )
{
    makeEmpty( );
}

/**
* Internal method to make subtree empty.
*/
void makeEmpty( BinaryNode * & t ) const
{
    if( t != NULL )
    {
            makeEmpty( t->left );
            makeEmpty( t->right );
            delete t;
    }
    t = NULL;
}
```

T(N) = ?

operator= and clone

```
/** * Deep copy. */
const BinarySearchTree & operator=( const BinarySearchTree & rhs )
{
    if( this != &rhs )
    {
            makeEmpty( );
            root = clone( rhs.root );
    }
    return *this;
}

/** * Internal method to clone subtree. */
BinaryNode * clone( BinaryNode * t ) const
{
    if( t == NULL )
            return NULL;

    return new BinaryNode( t->element,
                    clone( t->left ), clone( t->right ) );
}
```

Average Case Analysis

- $T(N) = O(d)$ for all operations except MakeEmpty and operator=, where d = depth of the node containing the accessed item.
- Average analysis:
  - Q: What is the average depth over all nodes in a tree?
- Assumption:
  - all insertion sequences are equally likely.
- Internal path length: $D(N)$ = sum of the depths of all nodes in the tree
- Average depth = $D(N)/N$
- $D(N) = ?$

Average Case Analysis

- D(N) = internal path length of a tree T of N nodes

  D(1) = 0, D(0) = 0
- T consists of:
  - An i-node left subtree => D(i) + i
  - An (N-i-1)-node right subtree => D(N-i-1) + (N-i-1)
- D(N) = D(i) + D(N-i-1) + N -1
- All subtree sizes are equally likely

$$D(N)=\frac{2}{N}\left[\sum_{j=0}^{N-1} D(j)\right]+N-1$$

Average Case Analysis

- $ND(N) - (N-1)D(N-1) = 2D(N-1) + 2(N-1)$
- $ND(N) = (N+1)D(N-1) + 2(N-1)$
- $D(N)/(N+1) = D(N-1)/N + 2(N-1)/(N(N+1))$
- $D(N)/(N+1) = D(1)/2 + 2(\sum_2^N (i-1)/(i(i+1)))$
- $D(N)/(N+1) = 2 \sum_2^N (2/(i+1) - 1/i)$
- $D(N)/(N+1) = 2(1/(N+1) - ½ + \sum_2^N (1/(i+1))$
- $D(N) = O(N \log N)$

=> Average depth of a node = O(log N)