Lecture No.14
Deletion in Binary Search Tree

CC-213 Data Structures
Department of Computer Science
University of the Punjab

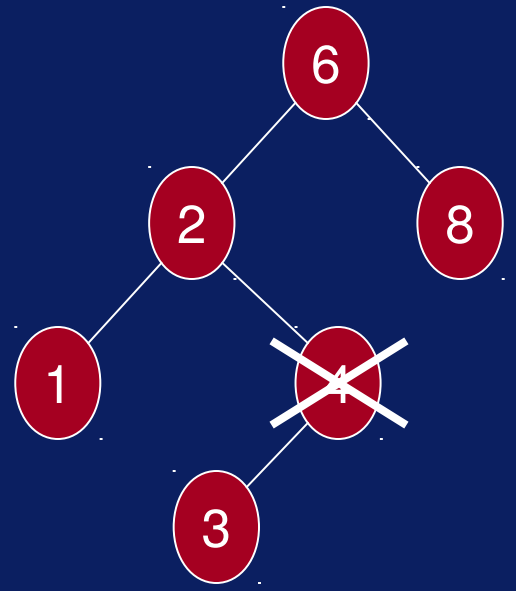
Slides modified very slightly from the late Dr. Sohail Aslam's lectures at VU

Deleting a node in BST

- As is common with many data structures, the hardest operation is deletion.
- Once we have found the node to be deleted, we need to consider several possibilities.
- If the node is a *leaf*, it can be deleted immediately.

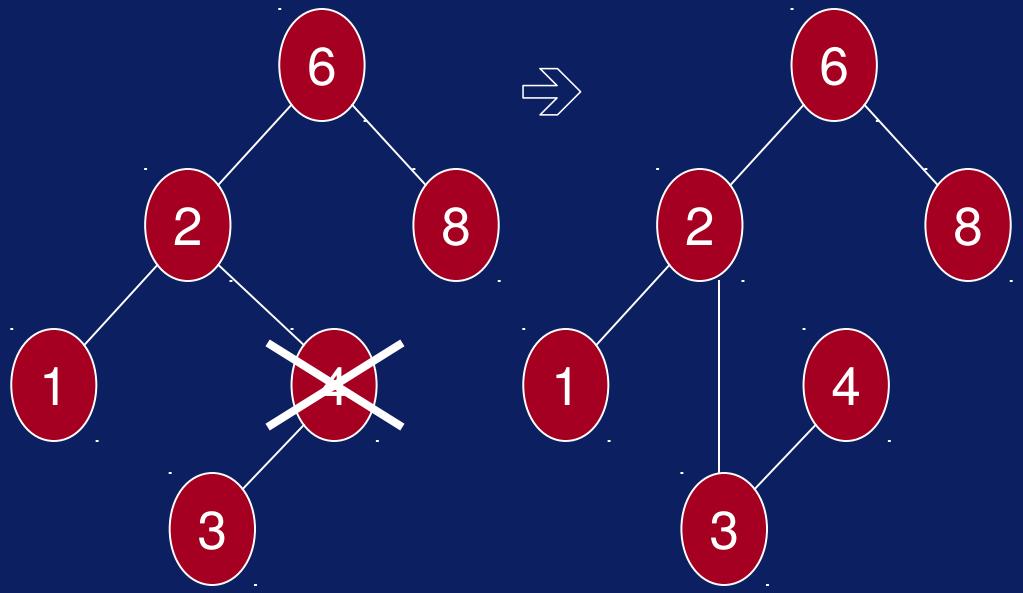
Deleting a node in BST

- If the node has one child, the node can be deleted after its parent adjusts a pointer to bypass the node and connect to inorder successor.



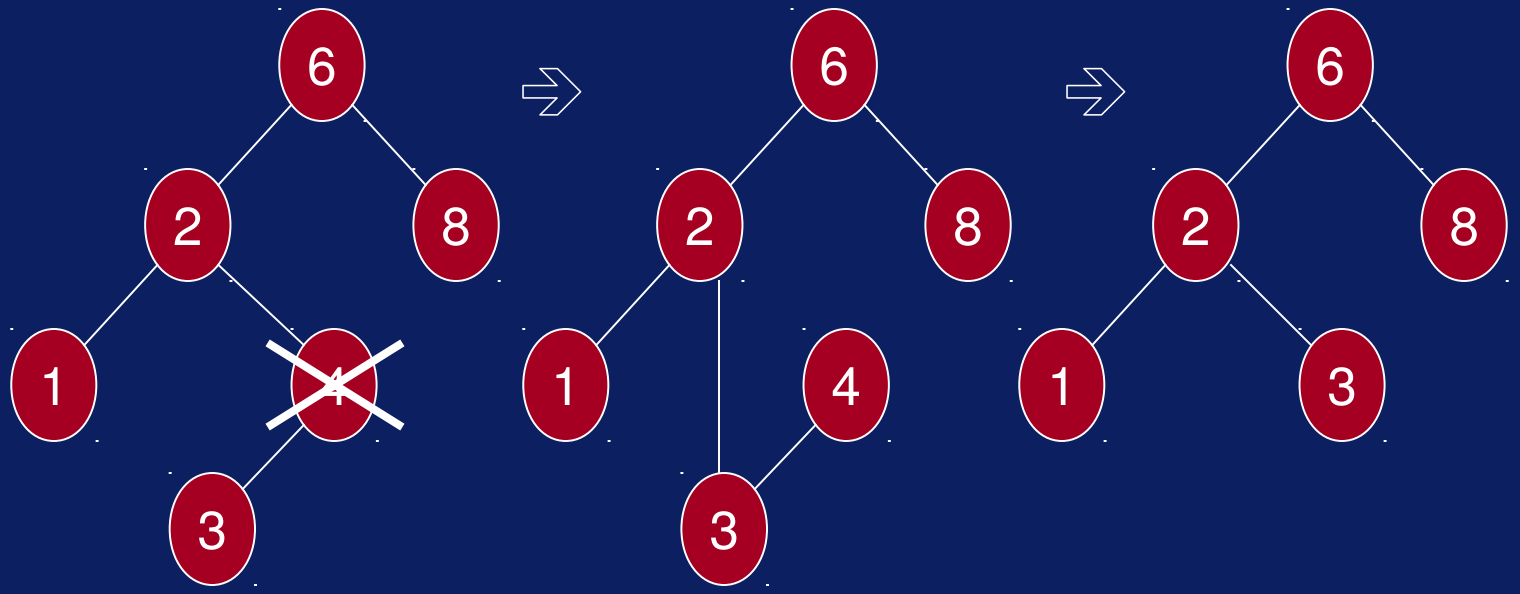
Deleting a node in BST

- The inorder traversal order has to be maintained after the delete.



Deleting a node in BST

- The inorder traversal order has to be maintained after the delete.

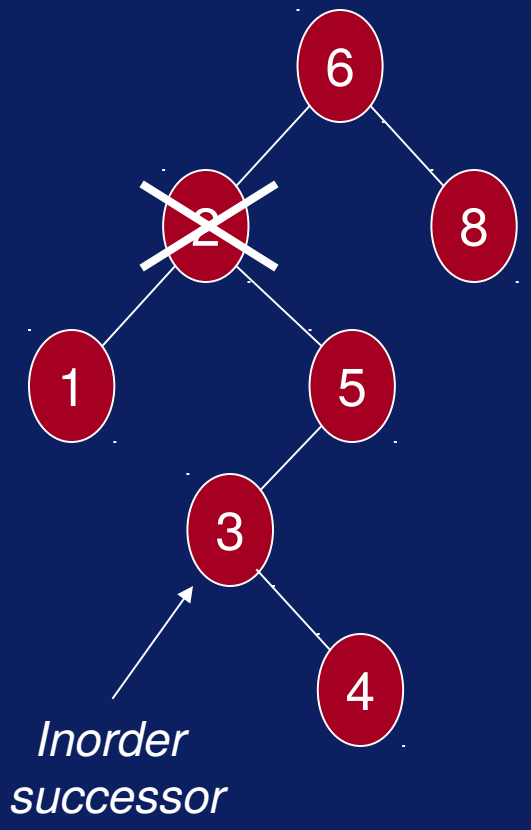


Deleting a node in BST

- The complicated case is when the node to be deleted has both left and right subtrees.
- The strategy is to replace the data of this node with the smallest data of the right subtree and recursively delete that node.

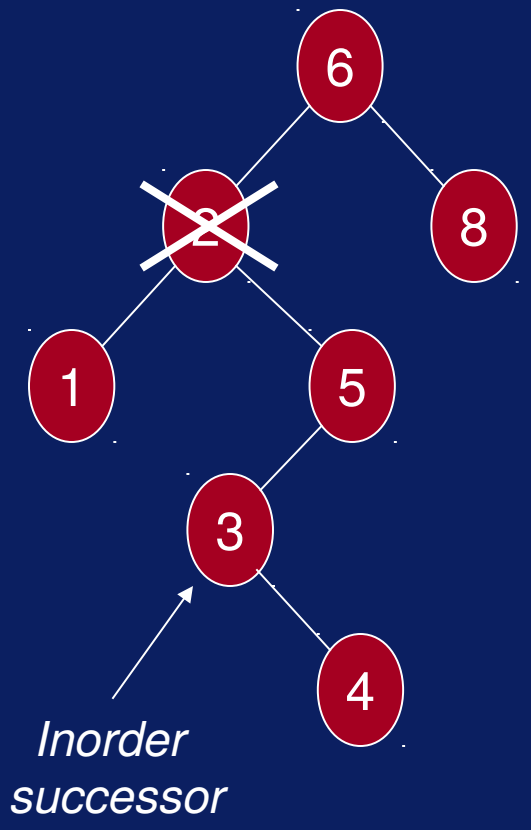
Deleting a node in BST

Delete(2): locate inorder successor



Deleting a node in BST

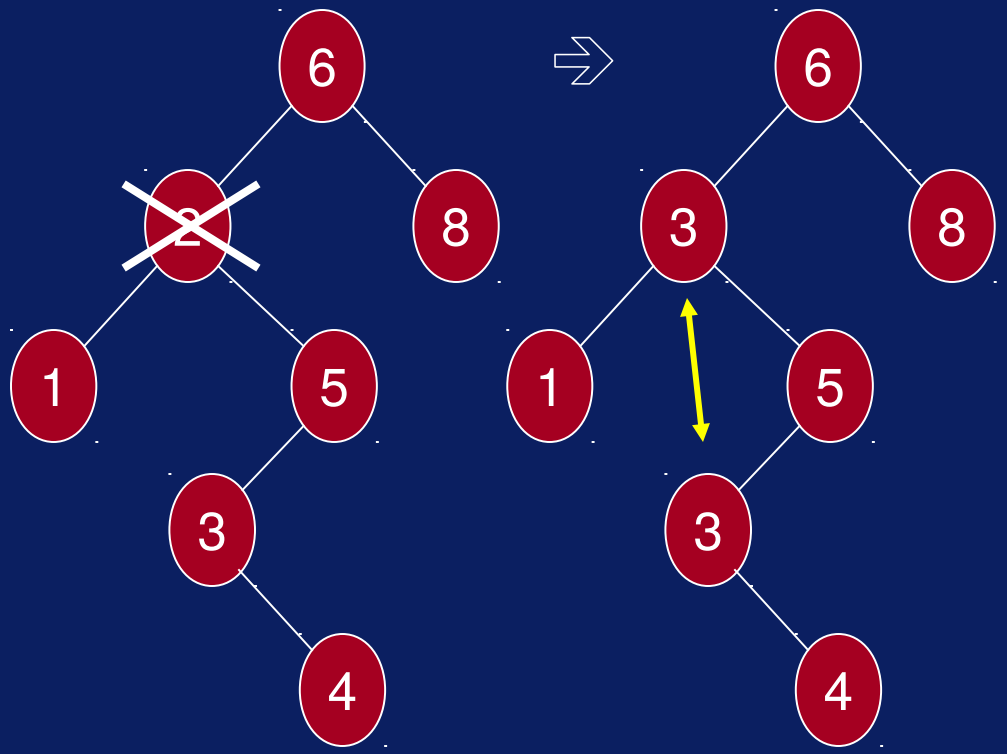
Delete(2): locate inorder successor



- Inorder successor will be the left-most node in the right subtree of 2.
- The inorder successor will not have a left child because if it did, that child would be the left-most node.

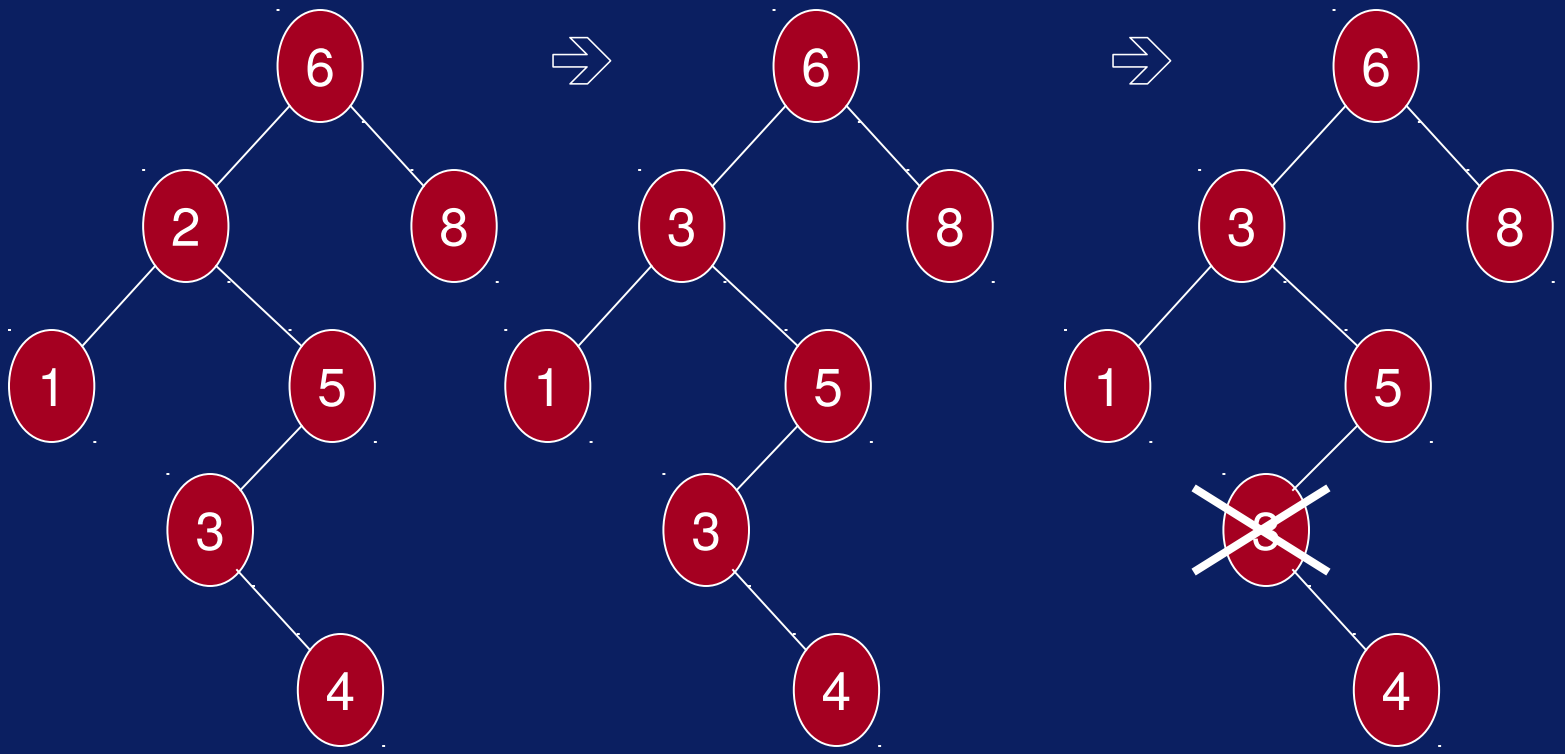
Deleting a node in BST

Delete(2): copy data from inorder successor



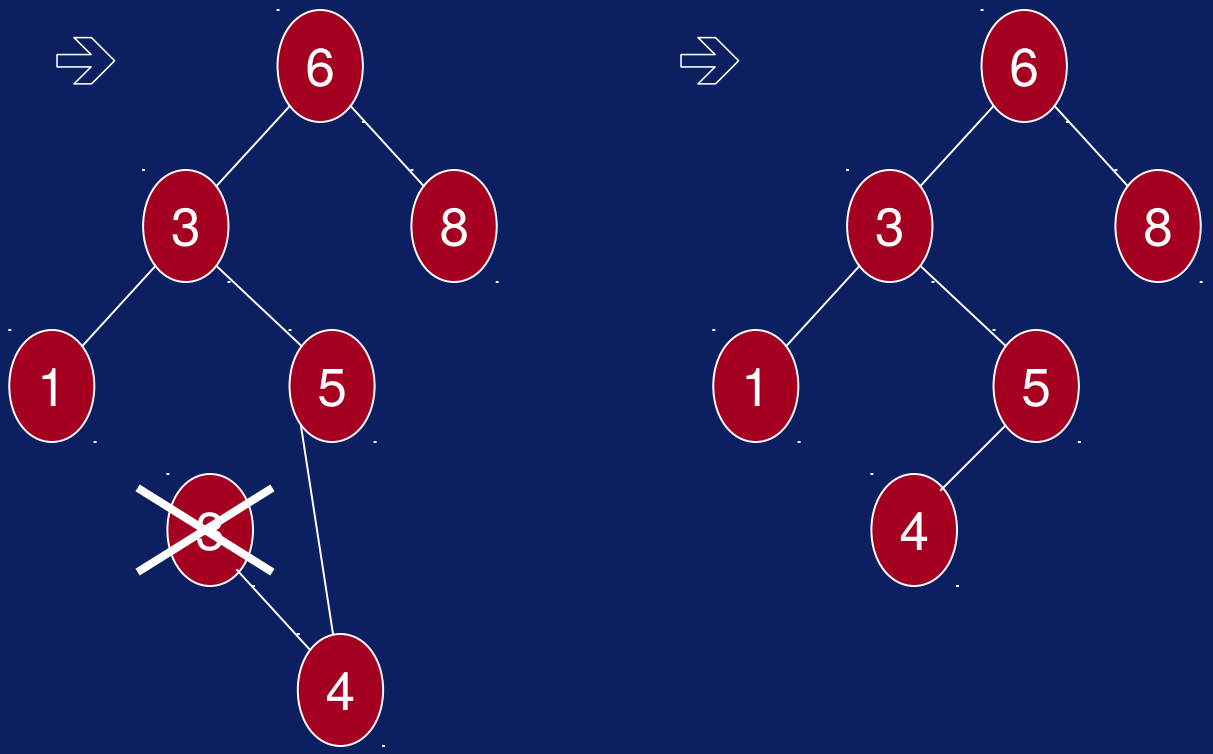
Deleting a node in BST

Delete(2): remove the inorder successor



Deleting a node in BST

Delete(2)



C++ code for delete

- ‘delete’ is C++ keyword. We will call our deleteNode routine remove.
- Here is the C++ code for remove.

C++ code for delete

```
TreeNode<int>* remove(TreeNode<int>* tree, int info)
{
    TreeNode<int>* t;
    int cmp = info - *(tree->getInfo());
    if( cmp < 0 ){
        t = remove(tree->getLeft(), info);
        tree->setLeft( t );
    }
    else if( cmp > 0 ){
        t = remove(tree->getRight(), info);
        tree->setRight( t );
    }
```

C++ code for delete

```
TreeNode<int>* remove(TreeNode<int>* tree, int info)
{
    TreeNode<int>* t;
    int cmp = info - *(tree->getInfo());
    if( cmp < 0 ){
        t = remove(tree->getLeft(), info);
        tree->setLeft( t );
    }
    else if( cmp > 0 ){
        t = remove(tree->getRight(), info);
        tree->setRight( t );
    }
```

C++ code for delete

```
TreeNode<int>* remove(TreeNode<int>* tree, int info)
{
    TreeNode<int>* t;
    int cmp = info - *(tree->getInfo());
    if( cmp < 0 ){
        t = remove(tree->getLeft(), info);
        tree->setLeft( t );
    }
    else if( cmp > 0 ){
        t = remove(tree->getRight(), info);
        tree->setRight( t );
    }
```

C++ code for delete

```
TreeNode<int>* remove(TreeNode<int>* tree, int info)
{
    TreeNode<int>* t;
    int cmp = info - *(tree->getInfo());
    if( cmp < 0 ){
        t = remove(tree->getLeft(), info);
        tree->setLeft( t );
    }
    else if( cmp > 0 ){
        t = remove(tree->getRight(), info);
        tree->setRight( t );
    }
```

C++ code for delete

```
//two children, replace with inorder successor
else if(tree->getLeft() != NULL
        && tree->getRight() != NULL ){
    TreeNode<int>* minNode;
    minNode = findMin(tree->getRight());
    tree->setInfo( minNode->getInfo() );
    t = remove(tree->getRight(),
               *(minNode->getInfo()));
    tree->setRight( t );
}
```

C++ code for delete

```
//two children, replace with inorder successor
else if(tree->getLeft() != NULL
        && tree->getRight() != NULL ){
    TreeNode<int>* minNode;
    minNode = findMin(tree->getRight());
    tree->setInfo( minNode->getInfo() );
    t = remove(tree->getRight(),
               *(minNode->getInfo()));
    tree->setRight( t );
}
```

C++ code for delete

```
//two children, replace with inorder successor
else if(tree->getLeft() != NULL
        && tree->getRight() != NULL ){
    TreeNode<int>* minNode;
    minNode = findMin(tree->getRight());
    tree->setInfo( minNode->getInfo() );
    t = remove(tree->getRight(),
               *(minNode->getInfo()));
    tree->setRight( t );
}
```

C++ code for delete

```
//two children, replace with inorder successor
else if(tree->getLeft() != NULL
        && tree->getRight() != NULL ){
    TreeNode<int>* minNode;
    minNode = findMin(tree->getRight());
    tree->setInfo( minNode->getInfo() );
    t = remove(tree->getRight(),
               *(minNode->getInfo()));
    tree->setRight( t );
}
```

C++ code for delete

```
//two children, replace with inorder successor
else if(tree->getLeft() != NULL
        && tree->getRight() != NULL ){
    TreeNode<int>* minNode;
    minNode = findMin(tree->getRight());
    tree->setInfo( minNode->getInfo() );
    t = remove(tree->getRight(),
               *(minNode->getInfo()));
    tree->setRight( t );
}
```

C++ code for delete

```
    else {  // case 1
        TreeNode<int>* nodeToDelete = tree;
        if( tree->getLeft() == NULL ) //will handle 0 children
            tree = tree->getRight();
        else if( tree->getRight() == NULL )
            tree = tree->getLeft();
        else tree = NULL;

        delete nodeToDelete;
    }
    return tree;
}
```

C++ code for delete

```
    else {  // case 1
        TreeNode<int>* nodeToDelete = tree;
        if( tree->getLeft() == NULL ) //will handle 0 children
            tree = tree->getRight();
        else if( tree->getRight() == NULL )
            tree = tree->getLeft();
        else tree = NULL;

        delete nodeToDelete;
    }
    return tree;
}
```

C++ code for delete

```
    else {  // case 1
        TreeNode<int>* nodeToDelete = tree;
        if( tree->getLeft() == NULL ) //will handle 0 children
            tree = tree->getRight();
        else if( tree->getRight() == NULL )
            tree = tree->getLeft();
        else tree = NULL;

        delete nodeToDelete;
    }
    return tree;
}
```

C++ code for delete

```
TreeNode<int>* findMin(TreeNode<int>* tree)
{
    if( tree == NULL )
        return NULL;
    if( tree->getLeft() == NULL )
        return tree; // this is it.
    return findMin( tree->getLeft() );
}
```

C++ code for delete

```
TreeNode<int>* findMin(TreeNode<int>* tree)
{
    if( tree == NULL )
        return NULL;
    if( tree->getLeft() == NULL )
        return tree; // this is it.
    return findMin( tree->getLeft() );
}
```

C++ code for delete

```
TreeNode<int>* findMin(TreeNode<int>* tree)
{
    if( tree == NULL )
        return NULL;
    if( tree->getLeft() == NULL )
        return tree; // this is it.
    return findMin( tree->getLeft() );
}
```

BinarySearchTree.h

Let us design the BinarySearchTree class (factory).

BinarySearchTree.h

```
#ifndef _BINARY_SEARCH_TREE_H_
#define _BINARY_SEARCH_TREE_H_
#include <iostream.h>       // For NULL

// Binary node and forward declaration
template <class EType>
class BinarySearchTree;
```

BinarySearchTree.h

```
#ifndef _BINARY_SEARCH_TREE_H_
#define _BINARY_SEARCH_TREE_H_
#include <iostream.h>       // For NULL

// Binary node and forward declaration
template <class EType>
class BinarySearchTree;
```

BinarySearchTree.h

```
template <class EType>
class BinaryNode
{
    EType element;
    BinaryNode *left;
    BinaryNode *right;

    BinaryNode( const EType & theElement,
                BinaryNode *lt, BinaryNode *rt )
      : element( theElement ), left( lt ),
        right( rt ) { }
    friend class BinarySearchTree<EType>;
};
```

BinarySearchTree.h

```
template <class EType>
class BinaryNode
{
    EType element;
    BinaryNode *left;
    BinaryNode *right;

    BinaryNode( const EType & theElement,
                BinaryNode *lt, BinaryNode *rt )
      : element( theElement ), left( lt ),
        right( rt ) { }
    friend class BinarySearchTree<EType>;
};
```

BinarySearchTree.h

```
template <class EType>
class BinaryNode
{
    EType element;
    BinaryNode *left;
    BinaryNode *right;

    BinaryNode( const EType & theElement,
                BinaryNode *lt, BinaryNode *rt )
      : element( theElement ), left( lt ),
        right( rt ) { }
    friend class BinarySearchTree<EType>;
};
```

BinarySearchTree.h

```
template <class EType>
class BinaryNode
{
    EType element;
    BinaryNode *left;
    BinaryNode *right;

    BinaryNode( const EType & theElement,
                BinaryNode *lt, BinaryNode *rt )
      : element( theElement ), left( lt ),
        right( rt ) { }
    friend class BinarySearchTree<EType>;
};
```

BinarySearchTree.h

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

    const EType& findMin( ) const;
    const EType& findMax( ) const;
    const EType& find( const EType & x ) const;
    bool isEmpty( ) const;
    void printInorder( ) const;
```

BinarySearchTree.h

```
void insert( const EType& x );
void remove( const EType& x );

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

BinarySearchTree.h

```
private:
    BinaryNode<EType>* root;
    // ITEM_NOT_FOUND object used to signal failed finds
    const EType ITEM_NOT_FOUND;

    const EType& elementAt( BinaryNode<EType>* t );
    void insert(const EType& x, BinaryNode<EType>* & t);
    void remove(const EType& x, BinaryNode<EType>* & t);
    BinaryNode<EType>* findMin(BinaryNode<EType>* t);
    BinaryNode<EType>* findMax(BinaryNode<EType>* t);
    BinaryNode<EType>* find(const EType& x,
                            BinaryNode<EType>* t );
    void makeEmpty(BinaryNode<EType>* & t);
    void printInorder(BinaryNode<EType>* t);
};
#endif
```