The Binary Search Tree

Section 4.3

Trees 3
Binary Search Tree
Binary Search Tree
- Also known as Totally Ordered Tree
Binary Search Tree

- Also known as Totally Ordered Tree

- Definition: A binary tree $\mathbb{B}$ is called a binary search tree iff:
Binary Search Tree

- Also known as Totally Ordered Tree

- Definition: A binary tree $B$ is called a binary search tree iff:
  - There is an order relation $\leq$ defined for the vertices of $B$
Binary Search Tree

- Also known as Totally Ordered Tree

- Definition: A binary tree \( \mathcal{B} \) is called a **binary search tree** iff:
  - There is an order relation \( \preceq \) defined for the vertices of \( \mathcal{B} \)
  - For any vertex \( v \), and any descendant \( u \) of \( v.\text{left} \), \( u \preceq v \)
Binary Search Tree

- Also known as Totally Ordered Tree

- Definition: A binary tree \( B \) is called a binary search tree iff:
  - There is an order relation \( \leq \) defined for the vertices of \( B \)
  - For any vertex \( v \), and any descendant \( u \) of \( v \).left, \( u \leq v \)
  - For any vertex \( v \), and any descendant \( w \) of \( v \).right, \( v \leq w \)
Binary Search Tree

Which one is NOT a BST?
Binary Search Tree
Binary Search Tree

- Consequences:
Binary Search Tree

- Consequences:
  - The smallest element in a binary search tree (BST) is the "left-most" node
Binary Search Tree

- Consequences:
  - The smallest element in a binary search tree (BST) is the “left-most” node
  - The largest element in a BST is the “right-most” node
Binary Search Tree

- Consequences:
  - The smallest element in a binary search tree (BST) is the “left-most” node
  - The largest element in a BST is the “right-most” node
  - Inorder traversal of a BST encounters nodes in increasing order
Binary Search using BST
Binary Search using BST

- Assumes nodes are organized in a totally ordered binary tree
Binary Search using BST

- Assumes nodes are organized in a totally ordered binary tree
  - Begin at root node
Binary Search using BST

- Assumes nodes are organized in a totally ordered binary tree
  - Begin at root node
  - Descend using comparison to make left/right decision
Binary Search using BST

- Assumes nodes are organized in a totally ordered binary tree
  - Begin at root node
  - Descend using comparison to make left/right decision
    - if (search_value < node_value)
      - "go to the left child"
Binary Search using BST

- Assumes nodes are organized in a totally ordered binary tree
  - Begin at root node
  - Descend using comparison to make left/right decision
    - if (search_value < node_value)
      “go to the left child”
    - else if (search_value > node_value)
      “go to the right child”
Binary Search using BST

- Assumes nodes are organized in a totally ordered binary tree
  - Begin at root node
  - Descend using comparison to make left/right decision
    - if (search_value < node_value)
      "go to the left child"
    - else if (search_value > node_value)
      "go to the right child"
    - else return true // success
Binary Search using BST

- Assumes nodes are organized in a totally ordered binary tree
  - Begin at root node
  - Descend using comparison to make left/right decision
    - if (search_value < node_value)
      - "go to the left child"
    - else if (search_value > node_value)
      - "go to the right child"
    - else return true // success
  - Until descending move is impossible
Binary Search using BST

- Assumes nodes are organized in a totally ordered binary tree
  - Begin at root node
  - Descend using comparison to make left/right decision
    - if (search_value < node_value)
      "go to the left child"
    - else if (search_value > node_value)
      "go to the right child"
    - else return true // success
  - Until descending move is impossible
  - Return false (failure)
BST Class Template

```cpp
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);
```
**BST Class Template (contd.)**

```cpp
private:

    struct BinaryNode
    {
        Compareable element;
        BinaryNode *left;
        BinaryNode *right;

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

    BinaryNode *root;

    void insert( const Compareable & x, BinaryNode * & t ) const;
    void remove( const Compareable & x, BinaryNode * & t ) const;
    BinaryNode * findMin( BinaryNode *t ) const;
    BinaryNode * findMax( BinaryNode *t ) const;
    bool contains( const Compareable & x, BinaryNode * t ) const;
    void makeEmpty( BinaryNode * & t );
    void printTree( BinaryNode *t ) const;
    BinaryNode * clone( BinaryNode *t ) const;
```

*Pointer passed by reference (why?)*

*Internal functions used in recursive calls*
BST: Public members calling private recursive functions

```cpp
/**
 * Returns true if x is found in the tree.
 */
bool contains( const Comparable & x ) const
{
    return contains( x, root );
}

/**
 * Insert x into the tree; duplicates are ignored.
 */
void insert( const Comparable & x )
{
    insert( x, root );
}

/**
 * Remove x from the tree. Nothing is done if x is not found.
 */
void remove( const Comparable & x )
{
    remove( x, root );
}
```
BST: Searching for an element

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

bool contains( const Comparable & x, BinaryNode * t ) const
{
    if( t == NULL )
        return false;
    else if( x < t->element )
        return contains( x, t->left );
    else if( t->element < x )
        return contains( x, t->right );
    else
        return true;  // Match
}
```
BST: Search using function objects

```cpp
template <typename Object, typename Comparator=less<Object> >
class BinarySearchTree {
    public:
        // Same methods, with Object replacing Comparable
    private:
        BinaryNode *root;
        Comparator isLessThan;
        // Same methods, with Object replacing Comparable
        
        /**
         * Internal method to test if an item is in a subtree.
         * x is item to search for.
         * t is the node that roots the subtree.
         */
        bool contains( const Object &x, BinaryNode *t ) const
        {
            if( t == NULL )
                return false;
            else if( isLessThan(x, t->element) )
                return contains( x, t->left );
            else if( isLessThan(t->element, x) )
                return contains( x, t->right );
            else
                return true;  // Match
        }
};
```
BST: Find the smallest element

```c
/**
 * 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 );
}
```

Tail recursion
BST: Find the biggest element

```c
/*
 * 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;
}
```
BST: Insertion

Before insertion

After insertion
BST: Insertion (contd.)

/*
 * Internal method to insert into a subtree.
 * x is the item to insert.
 * t is the node that roots the subtree.
 * Set the new root of the subtree.
 */
void insert( const Comparable & x, BinaryNode * & t )
{
    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
}
BST: Deletion

Deleting a node with one child

Before `delete(4)`
BST: Deletion

Deleting a node with one child

Before delete(4)

After delete(4)
BST: Deletion

Deleting a node with one child

Before delete(4)

After delete(4)

Deletion Strategy: Bypass the node being deleted
BST: Deletion (contd.)

Deleting a node with two children

Before `delete(2)`
BST: Deletion (contd.)

Before delete(2)

Deleting a node with two children

After delete(2)
BST: Deletion (contd.)

Deleting a node with two children

Before \texttt{delete}(2)

After \texttt{delete}(2)

Deletion Strategy: Replace the node with smallest node in the right subtree
/**
 * Internal method to remove from a subtree.
 * x is the item to remove.
 * t is the node that roots the subtree.
 * Set the new root of the subtree.
 */

void remove( const Comparable & x, BinaryNode * & t )
{
    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;
    }
}
BST: Lazy Deletion

- Another deletion strategy
  - Don’t delete!
  - Just mark the node as deleted.
  - Wastes space
  - But useful if deletions are rare or space is not a concern.
BST: Destructor

```c
/**
 * Destructor for the tree
 */
~BinarySearchTree()
{
    makeEmpty();
}
/**
 * Internal method to make subtree empty.
 */
void makeEmpty( BinaryNode * & t )
{
    if( t != NULL )
    {
        makeEmpty( t->left );
        makeEmpty( t->right );
        delete t;
    }
    t = NULL;
}
```
BST: Assignment Operator

```cpp
/**
 * 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 ) );
}
```
BST: Insertion Bias
BST: Insertion Bias

- Start with an empty tree.
BST: Insertion Bias

- Start with an empty tree.
- Insert elements in sorted order
BST: Insertion Bias

- Start with an empty tree.
- Insert elements in sorted order
- What tree do you get?
BST: Insertion Bias

- Start with an empty tree.
- Insert elements in sorted order.
- What tree do you get?
- How do you fix it?
BST: Deletion Bias

After large number of alternating insertions and deletions

Why this bias? How do you fix it?