Data Structures

(Feodor F. Dragan)
Department of Mathematics and Computer Science
Kent State University

Fall, 2000

CHAPTER 19
Binary Search Trees

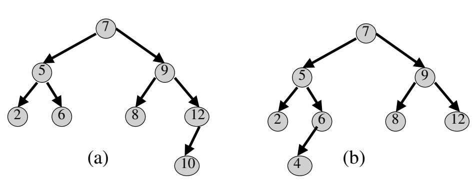
Contents

We will learn:

- The basic binary search tree;
- A method for adding order statistics (i.e., the *findKth* operation);
- Three different ways to eliminate the *O(N)* worst case (namely, the *AVL tree*, *red-black tree*, and *AA tree*);
- Implementation of the STL *set* data structure; and
- Use of the *B-tree* to search a large database quickly.

Binary Search Trees: definition

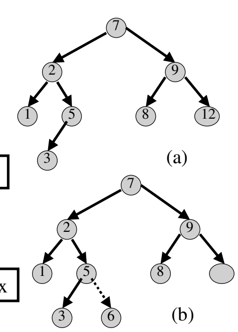
- For large amounts of input, the linear access time of linked lists is prohibitive. Hence, **Binary Search Trees (**the running time for insertion, find and deletion will be O(log N) on average).
- In general case, we search for an item (or element) by its **key**.
- **Definition**: For any node in the *binary search tree*, all smaller keyed nodes are in the left subtree and all larger keyed nodes are in the right subtree. Duplicates are not allowed.



Two binary trees: (a) a search tree; (b) not a search tree.

Binary Search Trees: operations

- find element x;
- remove element x;
- insert element x;
- find minimum element;
- find maximum element;
- remove minimum element.



find element x

A find operation is performed by repeatedly branching either left or right, depending on the result of a comparison.

insert element x

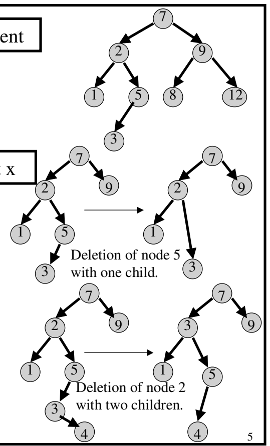
Try to find. Then insert x at the point at which the unsuccessful search terminated.

(a) before (b) after the insertion of 6.

find min/max element

The *findMin* operation is performed by following left nodes as long as there is a left child. The *findMax* operation is similar.

remove element x



- The *remove* operation is difficult because nonleaf nodes hold the tree together and we do not want to disconnect the tree. (remove a leaf is easy)
- If a node has one child, it can be removed by having its parent bypass it. (The root is a special case since it does not have a parent.)
- *removeMin, removeMax* are of these cases.
- A node with two children is replaced by using the smallest item in the right subtree. Then another node is removed.

C++ Implementation

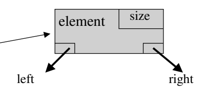
```
template <class Comparable>
class BinaryNode
{
   Comparable  element;
   BinaryNode *left;
   BinaryNode *right;
   int size;

   BinaryNode( const Comparable & theElement, BinaryNode *lt,
            BinaryNode *rt, int sz = 1 )
     : element( theElement ), left( lt ), right( rt ), size( sz ) { }

   friend class BinarySearchTree<Comparable>;
};
```

The *BinaryNode* class for the binary search tree.

a node in a binary search tree is




```
// BinarySearchTree class
// CONSTRUCTION: with no parameters or another BinarySearchTree.
// ******************PUBLIC OPERATIONS*******
// void insert( x ) --> Insert x
// void remove( x )  --> Remove x
// void removeMin( )  --> Remove smallest item
 // Comparable find( x )  --> Return item that matches x
// Comparable findMin( ) --> Return smallest item
// Comparable findMax( )  --> Return largest item
// bool isEmpty( )  --> Return true if empty; else false
// void makeEmpty( ) --> Remove all items
// ******************ERRORS*******************
// Exceptions are thrown by insert, remove, and removeMin if warranted

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

   Cref<Comparable> findMin( ) const;
   Cref<Comparable> findMax( ) const;
   Cref<Comparable> find( const Comparable & x ) const;
   bool isEmpty( ) const;
```

The *BinarySearchTree* class interface.

A *cref* object is returned so that we can distinguish between successful and unsuccessful searches. Functions for *cref* class are *isNull( )* and *get( )* (see page 169 of textbook).


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

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

   typedef BinaryNode<Comparable> Node;

 protected:
   Node *root;

   Cref<Comparable> elementAt( Node *t ) const;
   virtual void insert( const Comparable & x, Node * & t ) const;
   virtual void remove( const Comparable & x, Node * & t ) const;
   virtual void removeMin( Node * & t ) const;
   Node * findMin( Node *t ) const;
   Node * findMax( Node *t ) const;
   Node * find( const Comparable & x, Node *t ) const;
   void makeEmpty( Node * & t ) const;
   Node * clone( Node *t ) const;
};
```

The *BinarySearchTree* class interface (cont.).

The *root* pointer points at the root of the tree, which is *NULL* if the tree is empty.

difference

- The public class functions will call hidden private routines.

The *find operation for binary search trees.*

```
// Internal method to find an item in a subtree.
// x is item to search for.  t is the node that roots the tree.
// Return node containing the matched item.
template <class Comparable>
BinaryNode<Comparable> * BinarySearchTree<Comparable>::
find( const Comparable & x, Node *t ) const
{   while( t != NULL )
        if( x < t->element ) t = t->left;
        else if( t->element < x ) t = t->right;
            else  return t;    // Match
    return NULL;        // Not found
}
```

Because of call by value, the actual argument (*root*) is not changed.

The *elementAt* member function.

```
// Internal method to wrap the element field in node t inside a Cref object.
template <class Comparable>
Cref<Comparable> BinarySearchTree<Comparable>::elementAt( Node *t ) const
{ return t == NULL ? Cref<Comparable>( ) : Cref<Comparable>( t->element ); }
```

The *find* public member function.

```
// Find item x in the tree.
// Return the matching item wrapped in a Cref object.
template <class Comparable>
Cref<Comparable> BinarySearchTree<Comparable>::
find( const Comparable & x ) const
{ return elementAt( find( x, root ) ); }
```


findMin, findMax methods.

```
// Internal method to find the smallest item in a subtree t.
// Return node containing the smallest item.
template <class Comparable>
BinaryNode<Comparable> * BinarySearchTree<Comparable>::findMin( Node *t ) const
{ if( t != NULL )
      while( t->left != NULL )  t = t->left;
  return t; }

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

The recursive *insert* for the *BinarySearchTree* class.

```
// 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. Throw DuplicateItemException if x is already in t.
template <class Comparable>
void BinarySearchTree<Comparable>::insert( const Comparable & x, Node * & t ) const
{  if( t == NULL ) t = new Node( x, NULL, NULL );
   else if( x < t->element ) insert( x, t->left );
        else if( t->element < x ) insert( x, t->right );
             else throw DuplicateItemException( );
}
```

For *insert*, we must pass the tree **t** by reference to effect a change.

The *remove* for the *BinarySearchTree* class.

```
// 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. Throw ItemNotFoundException is x is not in t.
template <class Comparable>
void BinarySearchTree<Comparable>::remove( const Comparable & x, Node * & t ) const
{  if( t == NULL ) throw ItemNotFoundException( );
   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;
               removeMin( t->right ); }      // Remove minimum
           else                       // One or zero children
             { BinaryNode<Comparable> *oldNode = t;
               t = ( t->left != NULL ) ? t->left : t->right;  // Reroot t
               delete oldNode;    }           // delete old root
}
```

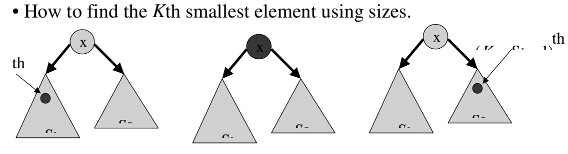
Passing **t** by reference also works in the *remove* routines. In effect we maintain the parent in the recursion stack.

The *removeMin* for the *BinarySearchTree* class.

```
// Internal method to remove minimum item from a subtree.
// t is the node that roots the tree.
// Set the new root. Throw UnderflowException if t is empty.
template <class Comparable>
void BinarySearchTree<Comparable>::removeMin( Node * & t ) const
{  if( t == NULL ) throw UnderflowException( );
   else if( t->left != NULL ) removeMin( t->left );
        else { Node *tmp = t;  t = t->right;  delete tmp;   }
}
```

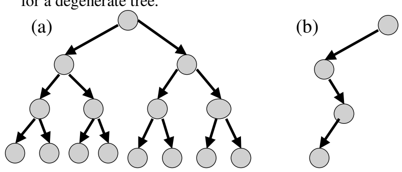

Order Statistics (*findKth*)

- Sometimes, we also have to be able to access the *K*th smallest element, for an arbitrary *K* provided as a parameter.
- We can implement *findKth* by maintaining the size of each node as we update the tree.
  - During *insert*, each node on the path to the insertion point gains one node in its subtree. Thus the size of each node increases by 1, and the inserted node has size 1.
  - In *removeMin,* each node on the path to the minimum loses one node in its subtree. Thus the size of each node decreases by 1.
  - During a *remove*, all nodes on the path to the node that physically removed also lose one node in their subtrees.
- How to find the *K*th smallest element using sizes.

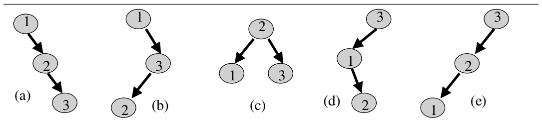


Analysis of Binary Search Tree Operations

- The cost of an operation is proportional to the depth of the last accessed node.
- The cost is logarithmic for a well-balanced tree, but it could be as bad as linear for a degenerate tree.



(a) The balanced search tree has a depth of ~log N; (b) The unbalanced tree has a depth of ~N.

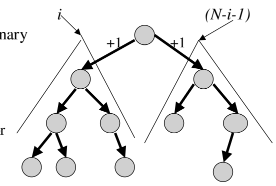


Binary search tree that can result from inserting a permutation 1,2,3; The balanced tree (c) is twice as likely to result as any of the others.

The cost of a successful search on average

- The **internal path length** is used to measure the cost of a successful search.
  - The internal path length of a binary search tree is the sum of depths of its nodes.
  - The average node depth is the internal path length divided by the number of nodes in the tree.

Theorem: The internal path length of a binary search tree on N nodes is O(N log N) on average.



Let ***D(N)*** be the average internal path length for trees of N nodes.

$$\left.\begin{aligned} &D(N) = (2/N)\left(\sum_{i=0}^{N-1} D(i)\right) + N - 1 \\ &D(1) = 0 \end{aligned}\right\} \Rightarrow O(N \log N)$$

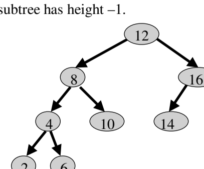
Hence, O(log N) on average for one operation.

Balanced Binary Search Trees

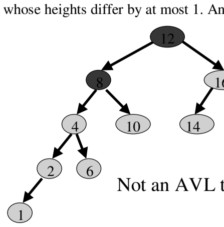
- Has an added structure property to guarantee logarithmic depth in the worst case.
- Updates are slower, but accesses are faster.

AVL trees

- The AVL tree was the first balanced binary search tree. (Adelson-Velskii and Landis, 1962)
- Every node in AVL tree has subtrees whose heights differ by at most 1. An empty subtree has height –1.



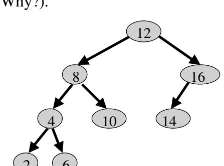
An AVL tree.



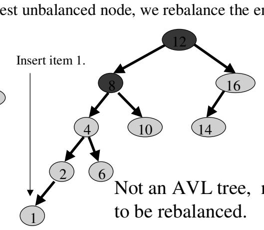
Not an AVL tree.

AVL trees (cont.)

- The depth of a typical node in an AVL tree is very close to the optimal log N.
- Hence, all searching operations in an AVL tree have logarithmic worst case bounds.
- An update in an AVL tree could destroy the balance. It must then be rebalanced before the operation can be considered complete.
- Only nodes on the path from the root to the insertion point can have their balances altered.
- If we fix the balance at the deepest unbalanced node, we rebalance the entire tree (Why?).



An AVL tree.



Not an AVL tree, needs to be rebalanced.

AVL trees: rebalancing.

There are four cases that we might have to fix; two are mirror images of the other two (1~4, 2~3). Let X be the node to be rebalanced.

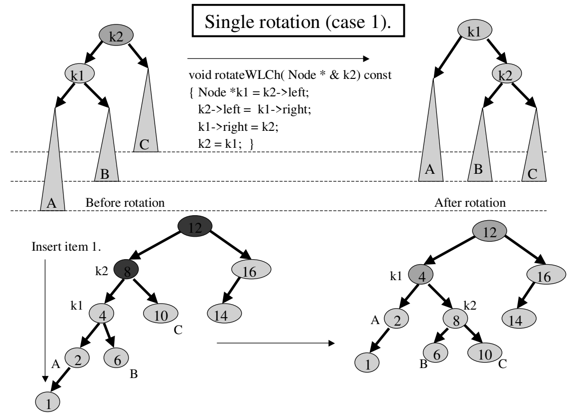
1. An insertion in the left subtree of the left child of X,
2. An insertion in the right subtree of the left child of X,
3. An insertion in the left subtree of the right child of X,
4. An insertion in the right subtree of the right child of X.

Balance is restored by tree rotations: a **single rotation** for left-left and right-right cases; a **double rotation** for left-right and right-left cases.

Single rotation.

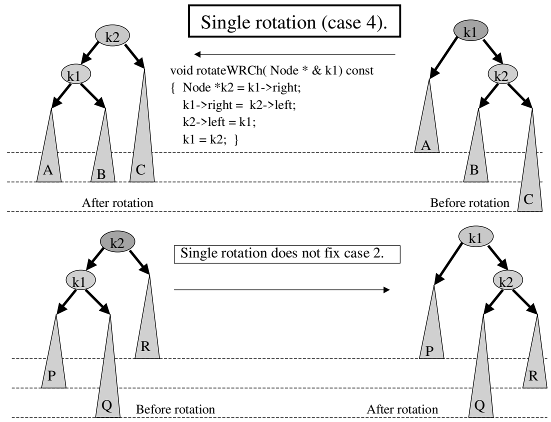
- A single rotation handles the cases 1 and 4.
- We rotate between a node and its child.
- The result is a binary search tree that satisfies the AVL property.

Single rotation (case 1).



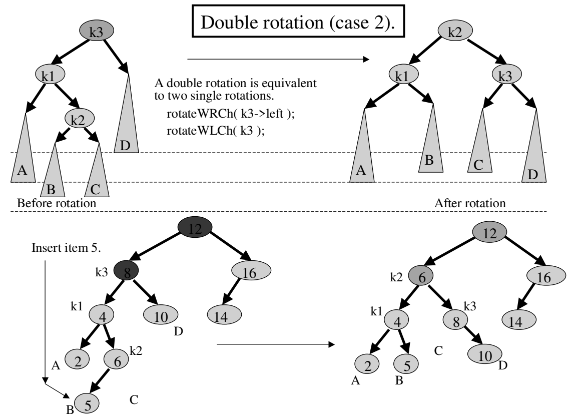

```
void rotateWLCh( Node * & k2) const
{ Node *k1 = k2->left;
  k2->left =  k1->right;
  k1->right = k2;
  k2 = k1;  }
```


Single rotation (case 4).




```
void rotateWRCh( Node * & k1) const
{  Node *k2 = k1->right;
   k1->right =  k2->left;
   k2->left = k1;
   k1 = k2;  }
```


Double rotation (case 2).



A double rotation is equivalent to two single rotations.

```
rotateWRCh( k3->left );
rotateWLCh( k3 );
```

Double rotation (case 3).

```
rotateWLCh( k1->right );
rotateWRCh( k1 );
```