# The Set ADT

Based on BinarySearch Tree after Chapter 4 Weiss DSAAC++

```
#ifndef BINARY_SEARCH_TREE_H
#define BINARY_SEARCH_TREE_H

#include <cassert>
#include <algorithm>
#include "Vector.h"
#include "Stack.h"
#include "Random.h"
using namespace std;

// Set ADT implemented exactly like BinaryTree
// Set iterator added to step through the stored
// data items im depth-first fashion;

template <typename C>
class Set
{
private:

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

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

public:

  class iterator
  {
  public:
    iterator() : current(nullptr) {}

    C & operator *()
    {
      return current->element;
    }
```

```
  iterator& operator++()
  {
      if (current == nullptr)
          return *this;

      if (current->right != nullptr)
      {
          current = current->right;
          while (current->left != nullptr)
          {
              antes.push(current);
              current = current->left;
          }
      }
      else
      {
          if (!antes.empty())
          {
              current = antes.top();
              antes.pop();
          }
          else
              current = nullptr;
      }
      return *this;
  }

  iterator operator++(int)
  {
    iterator old = *this;
    ++(*this);
    return old;
  }

  bool operator ==(const iterator & rhs) const
  {
    return current == rhs.current;
  }

  bool operator !=(const iterator & rhs) const
  {
    return !(*this == rhs);
  }

private:
```

```
    BinaryNode * current;
    Stack<Vector<BinaryNode* > > antes;

    iterator(BinaryNode* p, Stack<Vector<BinaryNode*> > st)
         : current{p}, antes{st}
    {}

    friend class Set<C>;
  }; // end class iterator


public:
    Set( ) : root{ nullptr }
    { }

    Set( const Set & rhs ) : root{ nullptr }
    {
        root = clone( rhs.root );
    }

    Set( Set && rhs ) : root{ rhs.root }
    {
        rhs.root = nullptr;
    }

    ~Set( )
    {
        makeEmpty( );
    }

    iterator begin()
    {
        BinaryNode* lmost = root;
        Stack<Vector<BinaryNode* > > nstack;

        while (lmost->left != nullptr)
        {
            nstack.push(lmost);
            lmost = lmost->left;
        }

        return iterator(lmost, nstack);
    }
```

```
iterator end()
{
    Stack<Vector<BinaryNode* > > emptystack;
    return iterator(nullptr, emptystack);
}

Set & operator=( const Set & rhs )
{
    Set copy = rhs;
    std::swap( *this, copy );
    return *this;
}

const C & findMin( ) const
{
    assert(!isEmpty());
    return findMin( root )->element;
}

const C & findMax( ) const
{
  assert(!isEmpty());
  return findMax( root )->element;
}

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

bool isEmpty( ) const
{
    return root == nullptr;
}


void printTree( ostream & out = cout ) const
{
    if( isEmpty( ) )
        out << "Empty tree" << endl;
    else
        printTree( root, out );
}


void printInternal(){ printInternal(root,0); }
```

```
  void makeEmpty( ){ makeEmpty( root ); }

  void insert( const C & x )
  {
      insert( x, root );
  }

  void remove( const C & x )
  {
      remove( x, root );
  }

private:


  BinaryNode *root;


  /**
   * 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 C & x, BinaryNode * & t )
  {
      if( t == nullptr )
          t = new BinaryNode{ x, nullptr, nullptr };
      else if( x < t->element )
          insert( x, t->left );
      else if( t->element < x )
          insert( x, t->right );
      else
          ;  // Duplicate; do nothing
  }




   /**
```

```
 * 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 C & x, BinaryNode * & t )
{

  if( t == nullptr )
  {
    cout << "nothing to remove" << endl;
        return;   // Item not found; do nothing
  }

  if( x < t->element )
    remove( x, t->left );
  else if( t->element < x )
    remove( x, t->right );

  else
  {// t->element == x

    if( t->left != nullptr && t->right != nullptr )
    {
      t->element = findMin( t->right )->element;
      remove( t->element, t->right );
    }
  else if (t->left != nullptr)
    {
      BinaryNode *oldNode = t;
      t = t->left;
      delete oldNode;
    }
  else if (t->right != nullptr)
    {
      BinaryNode *oldNode = t;
      t = t->right;
      delete oldNode;
    }
  else
    {
      assert(t->left == nullptr &&
         t->right == nullptr);

      BinaryNode *oldNode = t
      delete oldNode;
```

```
      t = nullptr;
    }
    }
  return;
}

/**
 * Internal method to find the smallest item in a subtree t.
 * Return node containing the smallest item.
 */
BinaryNode * findMin( BinaryNode *t ) const
{
    if( t == nullptr )
        return nullptr;
    if( t->left == nullptr )
        return t;
    return findMin( t->left );
}

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


/**
 * 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 C & x, BinaryNode *t ) const
{
    if( t == nullptr )
        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
}

void makeEmpty( BinaryNode * & t )
{
    if( t != nullptr )
    {
        makeEmpty( t->left );
        makeEmpty( t->right );
        delete t;
    }
    t = nullptr;
}


void printTree( BinaryNode *t, ostream & out ) const
{
    if( t != nullptr )
    {
        printTree( t->left, out );
        out << t->element << endl;
        printTree( t->right, out );
    }
}

void printInternal(BinaryNode* t, int offset) // KV's
{
  for(int i = 1; i <= offset; i++)
     cout << "..";

  if (t == nullptr)
  {
     cout << "@" << endl;
     return;
  }

  cout <<  t->element << endl;

  printInternal(t->left, offset + 1);
  printInternal(t->right, offset + 1);
}




BinaryNode * clone( BinaryNode *t ) const
```

```
    {
        if( t == nullptr )
            return nullptr;
        else
            return new BinaryNode{ t->element,
                                   clone( t->left ),
                                   clone( t->right ) };
    }
};

#endif
```