```
#include <iostream>
#include <iomanip>
#include <cstdlib>
#include <sys/time.h>

using namespace std;

#include "tree.h"

int        main();

int main()
{
        Tree<int>                       tree;

  // single rotations
  cerr << "inserting: " << setw(2) <<  6 << " ";  tree.insert(6);  cerr << endl;
  cerr << "inserting: " << setw(2) <<  4 << " ";  tree.insert(4);  cerr << endl;
  cerr << "inserting: " << setw(2) <<  2 << " ";  tree.insert(2);  cerr << endl;
  cerr << "inserting: " << setw(2) <<  8 << " ";  tree.insert(8);  cerr << endl;
  cerr << "inserting: " << setw(2) << 10 << " "; tree.insert(10);  cerr << endl;
  cerr << "inserting: " << setw(2) << 12 << " "; tree.insert(12);  cerr << endl;
  cerr << "inserting: " << setw(2) << 14 << " "; tree.insert(14);  cerr << endl;

  // double rotations
  cerr << "inserting: " << setw(2) << 32 << " "; tree.insert(32);  cerr << endl;
  cerr << "inserting: " << setw(2) << 30 << " "; tree.insert(30);  cerr << endl;
  cerr << "inserting: " << setw(2) << 28 << " "; tree.insert(28);  cerr << endl;

  // set up for deletion
  cerr << "inserting: " << setw(2) <<  3 << " ";  tree.insert(3);  cerr << endl;
  cerr << "inserting: " << setw(2) << 31 << " "; tree.insert(31);  cerr << endl;

  // inorder tree traversal
  cout << "Tree: " << endl << tree;
}
```

```
#include <iostream>

#include "tree.h"

template <typename T>
std::ostream& operator<<(std::ostream& s, const Tree<T>& rhs)
{
  if(rhs.empty()) s << "empty" << std::endl;
  else rhs.inorder(s,rhs.root);

  return s;
}

template <typename T>
void Tree<T>::inorder(std::ostream& s, Node* const &t) const
{
  if(t != NULL) {
    inorder(s,t->left);
    s << t->data << "(" << t->height << ") " << std::endl;
    inorder(s,t->right);
  }
}

template <typename T>
Tree<T>::Tree()
{
  // this constructor creates an emply Tree object
  root = NULL;
}

template <typename T>
const Tree<T>& Tree<T>::operator=(const Tree<T>& rhs)
{
  if(this != &rhs) {       // standard alias test
    clear();
    root = clone(rhs.root);
  }

  return *this;
}

template <typename T>
typename Tree<T>::Node* Tree<T>::clone(Node* t) const
{
  if(t == NULL) return NULL;

  return new Node(t->data,clone(t->left),clone(t->right),t->height);
}

template <typename T>
void Tree<T>::clear(Node* &t)
{
  if(t != NULL) {
    clear(t->left);
    clear(t->right);
    delete t;
  }
  t = NULL;
}

template <typename T>
typename Tree<T>::Node* Tree<T>::min(Node* t) const
{
  return t->left == NULL ? t : min(t->left);
}

template <typename T>
typename Tree<T>::Node* Tree<T>::max(Node* t) const
{
  return t->right == NULL ? t : max(t->right);
}

template <typename T>
bool Tree<T>::contains(const T& x, Node* t) const
{
  if(t == NULL) return false;
  else if(x < t->data) return contains(x,t->left);
  else if(x > t->data) return contains(x,t->right);
  else return true;
}

template <typename T>
void Tree<T>::insert( const T& x, Node* & t)
{
  if(t == NULL) t = new Node(x,NULL,NULL);
  else if(x < t->data) {
    insert(x,t->left);
    if(height(t->left) - height(t->right) == 2)
      if(x < t->left->data) {
        std::cerr << "rotate_left";
        rotate_left(t);
      } else {
        std::cerr << "double_rotate_left";
        double_left(t);
      }
  } else if(x > t->data) {
    insert(x,t->right);
    if(height(t->right) - height(t->left) == 2)
      if(x > t->right->data) {
        std::cerr << "rotate_right";
        rotate_right(t);
      } else {
        std::cerr << "double_rotate_right";
        double_right(t);
      }
  }
  else; // duplicate, do nothing
  // height of current node is 1 + max height of children
  t->height = max(height(t->left),height(t->right)) + 1;
}

template <typename T>
void Tree<T>::rotate_left(Node* & k2)
{
  // rotate tree node with left child
  // for AVL trees, this is single rotation for case 1
  // update heights then set new root
  Node* k1 = k2->left;
```

```
  k2->left = k1->right;
  k1->right = k2;
  k2->height = max(height(k2->left),height(k2->right)) + 1;
  k1->height = max(height(k1->left),k2->height) + 1;
  k2 = k1;
}

template <typename T>
void Tree<T>::rotate_right(Node* & k1)
{
  // rotate tree node with right child
  // for AVL trees, this is single rotation for case 4
  // update heights then set new root
  Node* k2 = k1->right;
  k1->right = k2->left;
  k2->left = k1;
  k1->height = max(height(k1->left),height(k1->right)) + 1;
  k2->height = max(height(k2->right),k1->height) + 1;
  k1 = k2;
}

template <typename T>
void Tree<T>::double_left(Node* & k3)
{
  // double rotate tree node: first left child
  // with its right child, then node k3 with new left child
  // for AVL trees, this is double rotation for case 2
  rotate_right(k3->left);
  rotate_left(k3);
}

template <typename T>
void Tree<T>::double_right(Node* & k3)
{
  // double rotate tree node: first right child
  // with its left child, then node k3 with new right child
  // for AVL trees, this is double rotation for case 3
  rotate_left(k3->right);
  rotate_right(k3);
}

template <typename T>
void Tree<T>::erase(const T& x, Node* & t)
{
  if(t == NULL) return; // item not found, do nothing

  if(x < t->data)
    erase(x,t->left);
  else if(x > t->data)
    erase(x,t->right);
  else if(t->left != NULL && t->right != NULL) // two children
  {
    t->data = min(t->right)->data;
    erase(t->data,t->right);
  } else {
       Node* old = t;
    t = (t->left != NULL) ? t->left : t->right;
    delete old;
  }
}

////////////////////////// specializations ////////////////////////////////
template class Tree<int>;
template std::ostream& operator<<(std::ostream&, const Tree<int>&);

template class Tree<float>;
template std::ostream& operator<<(std::ostream&, const Tree<float>&);

template class Tree<double>;
template std::ostream& operator<<(std::ostream&, const Tree<double>&);
```