```
#ifndef TREE_H
#define TREE_H

// forward declcarations
template <typename T> class Tree;
template <typename T> std::ostream& operator<<(std::ostream&, const Tree<T>&);

template <typename T>
class Tree {
        private:
        struct Node        // an all public class with data only, no member ftns
        {
                T        data;
                Node     *left;
                Node     *right;
                int      height;

                Node(const T& d = T(), Node *l = NULL,
                                       Node *r = NULL, int h = 0) : \
                        data(d), left(l), right(r), height(h) \
                                                { };
        };

        public:
        // constructors (overloaded)
        Tree();

        // copy constructor
//      Tree(const Tree& rhs);

        // destructors
        ~Tree()                                 { clear(); }

        // friends -- note the extra <> telling the compiler to instantiate
        // a templated version of the operator<< -- <T> is also legal, i.e.,
        // friend std::ostream& operator<< <T>(std::ostream& s, const Tree&);
friend  std::ostream& operator<< <>(std::ostream& s, const Tree& rhs);
friend  std::ostream& operator<<(std::ostream& s, Tree *rhs)
                                                { return(s << (*rhs)); }
        void inorder(std::ostream& s, Node* const &t) const;

        // assignment operator
        const Tree&        operator=(const Tree&);

        // operators

        // members
        bool            empty() const
                        { return root == NULL ? true : false;}
        bool            contains(const T& x) const { return contains(x,root); }
        void            insert(const T& x)         { insert(x, root); }
        void            erase(const T& x)          { erase(x, root); }
        void            clear()                    { clear(root); }
        const T&        min() const
                        { if(!empty()) return(min(root)->data); }
        const T&        max() const
                        { if(!empty()) return(max(root)->data); }

        // private: only available to this class
        private:
        Node            *root;

        bool            contains(const T&, Node* ) const;
        void            insert(const T& x, Node* &t);
        void            erase(const T& x, Node* &t);
        void            clear(Node* &);
        Node*           min(Node* ) const;
        Node*           max(Node* ) const;
        Node*           clone(Node* ) const;
        int             height(Node* t)            { return t == NULL ? -1 :
                                                                t->height; }
        int             max(int a,int b)           { return a > b ? a : b; }
        void            rotate_left(Node* &t);
        void            rotate_right(Node* &t);
        void            double_left(Node* &t);
        void            double_right(Node* &t);

};

#endif
```

Lazy AVL