```
#ifndef PAIRS_H
#define PAIRS_H

class pairs_t {
        public:
        // constructors (overloaded)
        pairs_t(float ix=0.0, char ic='a') : \
                        x(ix), c(ic)                    { };

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

        // destructors (default ok)
        //~pairs_t()                                    { };

        // 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 pairs_t&);
friend  std::ostream& operator<<(std::ostream& s, const pairs_t& rhs);
friend  std::ostream& operator<<(std::ostream& s, pairs_t *rhs)
                                                { return(s << (*rhs)); }

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

        // operators
        bool            operator<(const pairs_t& rhs) const;
        bool            operator>(const pairs_t& rhs) const;

        // member functions
        char            getc()                  { return c; }
        float           getx()                  { return x; }

        // private: only available to this class
        private:
        float           x;
        char            c;
};

#endif
```

```
#ifndef TREE_H
#define TREE_H

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

template <typename T>
class tree_t {

  private:
  struct node_t    // an all public class with data only, no member ftns
  {
       T        data;
       node_t   *left;
       node_t   *right;

       node_t(const T& d = T(), node_t *l = NULL, node_t *r = NULL) : \
              data(d), left(l), right(r) \
                                       { };
  };

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

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

  // destructors
  ~tree_t()                            { 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_t&);
  friend
  std::ostream& operator<< <>(std::ostream& s, const tree_t& rhs);
  friend
  std::ostream& operator<<(std::ostream& s, tree_t *rhs)
                                       { return(s << (*rhs)); }
  void inorder(std::ostream& s, node_t* const &t) const;

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

  // 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_t   *root;

  bool     contains(const T&, node_t* ) const;
  void     insert(const T& x, node_t* &t);
```

```
  void     erase(const T& x, node_t* &t);
  void     clear(node_t* &);
  node_t*  min(node_t* ) const;
  node_t*  max(node_t* ) const;
  node_t*  clone(node_t* ) const;

};

#endif
```