Thursday, September 24, 2009
3:29 PM

Some oddities in ASG2 (binheap):

```
... binheap<T>::clear()
{
   empty();   // ??
}
```

just a query, "is the binheap empty()?"
not asking "make the binheap empty"

```
....binheap<T>::deckey(int p, T& delta)
{
   arr[p] -= delta;   // OK
   ~~buildheap();~~   // WTF??
}
```

percolateUP()

Thursday, September 24, 2009
3:38 PM

Tree cont'd

- insert(), erase(), min(), max(), contains()
- clear() ⇝ have to unlink nodes & delete them. (free up memory)
  - can't just call empty() (don't make sense)

- once thing to notice:
  almost all public member functions have private counterpart

e.g.)
```
clear();              ——— called by "user"
clear(Node *& n);     — called by class itself
```

Node is private data member

class

reason & we don't want "the public" messing with it

read right-to-left: reference to a pointer

⇒ we can alter the ptr, change what it's pointing to

(in C# would be ptr to a ptr, NodeData)

```
class Tree {
  :
private:
Node * root;
bool contains(const T& Node*) const;
```

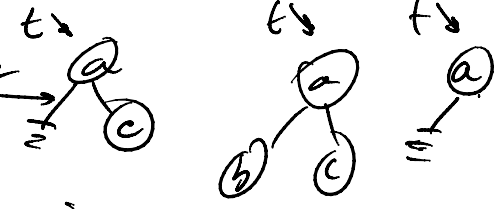
```
Node ...
bool contains (const T&, Node*) const;
void insert (const T&, Node*);
" erase ( "  " );
void clear ( Node*& ); // (ref to a pointer)
Node* min (Node*);
Node* max (Node*);
Node* clone (Node*);
public:
bool empty () const { return root == NULL ?
                            true : false; }
bool contains (const T& x) const
                { return contains (x, root); }
// similarly for insert, erase, clear()
const T& min() const
{ if (!empty()) return (min(root)->data); }
const T& max() const { max(root)->data); }
}
```

need an access point to the tree

Thursday, September 24, 2009
3:57 PM

```
template <typename T>
typename Tree<T>::Node*  Tree<T>::min(Node* t) const
{
  if(t == NULL) return NULL; // shouldn't happen;
                             // never call this routine
                             // if tree is empty

  if(t->left == NULL) return t;
  return min(t->left);
}
```



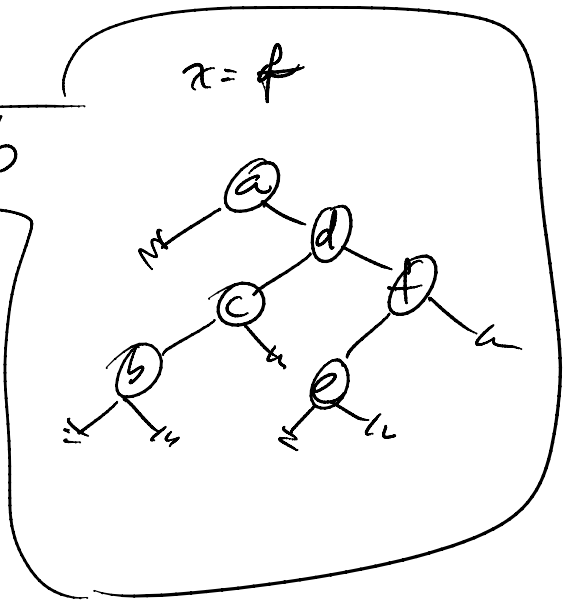
can reduce this to one line

```
return t->left == NULL ? t : min(t->left);
```

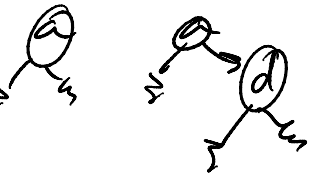
Thursday, September 24, 2009
4:07 PM

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

a d f c e b

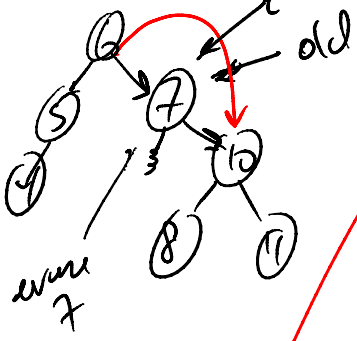


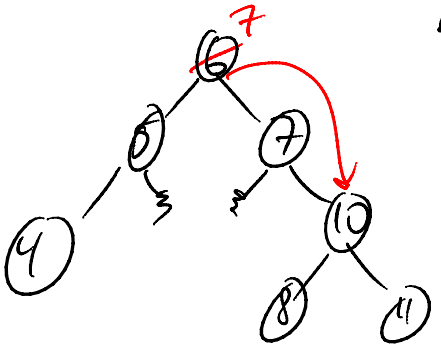

```
insert(T& x, Node *& t)
{
   if(t==NULL) t = New Node(x, NULL, NULL);
   else if(x<t->data) insert(x, t->left)
   else if(x>t->data) insert(x, t->right)
   else; // duplicate, do nothing
}
```



Thursday, September 24, 2009
4:16 PM

```
erase (const T& x, Node *& t)
{
  // similar to find — need to locate node
  // containing x
  if (t == NULL) return;
  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)
    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; // free up memory
  }
```





erase 6

if t->data (type T) was a huge object, we just called T::operator=(T&) which could be inefficient (deep copy) ⟹ 2 ways to improve this:

a) type T should be a pointer.
→ users of the Tree<> should

⇒ users of the Tree<> should use Tree<Object *>
Ok, but puts burden on Tree user (responsibility lies with user)

b) "rewire" the tree — change tree pointers around (like doubly linked list)
+
keep tree balanced ⇒ AVL trees

Thursday, September 24, 2009
4:31 PM

Tree<int> tree;

- what about printing? std::cout << tree << std::endl;

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

const ref to ptr