

More Trees

- recall our previous treatment of trees resulted in an efficient but complex version of insert
- try rewriting `insert` but using a functional approach

```
/*  
 * insert - place, i, into the tree and return  
 *         the tree.  
 */  
tree tree::insert (int i);
```

More Trees

- axioms

- ```
insert (i, empty ()) -> cons (i, empty (), empty ())

insert (i, cons (j, l, r)) ->
 if (i = j) then cons (i, l, r)
 else if (i < j) then cons (j, insert (i, l), r)
 else cons (j, l, insert (i, r))
```

## More Trees

- the first axiom says if the tree is empty then create a node with  $i$  as the root and
  - the left branch as an empty tree and
  - the right branch as an empty tree

## More Trees

- the second axiom says
  
- if we have a non-empty tree into which we want to place,  $i$ 
  - and the tree has a left subtree,  $l$ , right subtree,  $r$  and node value  $j$   
then we
  
- create a new tree if  $i == j$
  
- if ( $i < j$ )
  - create a tree with  $j$  at the root the left branch is the result of `insert (i, l)` and right branch,  $r$ .
  
- otherwise create a tree with  $j$  at the root, its left branch,  $l$  and the right branch the result of creating a tree with  $i$  and subtree  $r$

## More Trees

- another way to view the last axiom is to consider
- if ( $i < j$ ) then we leave the right branch alone and add  $i$  to the left branch
- if ( $i > j$ ) then we leave the left branch alone and add  $i$  to the right branch

## Implement insert using these axioms

- firstly we need some helper methods:

```
tree tree::left (void);
tree tree::right (void);
```

## Implement insert using these axioms

```
tree tree::insert (int i)
{
 if (is_empty ())
 return cons (i, empty (), empty ());
 else
 return insert_non_empty (i);
}
```

## Implement insert using these axioms

```
tree tree::insert_non_empty (int i)
{
 if (i == root ())
 return cons (i, left (), right ());
 else if (i < root ())
 return cons (root (), left ().insert (i), right ());
 else
 return cons (root (), left (), right ().insert (i));
}
```



## Consider a removal method

- again axioms:

- ```
remove (i, empty ()) -> empty ()

remove (i, cons (i, l, empty ())) -> l
remove (i, cons (i, empty (), r)) -> r

# item at root
remove(i, cons (i, l, r)) ->
  cons (min (r), l, remove (min (r), r))

# not at root
remove(i, cons (j, l, r)) ->
  if (i < j) then cons (j, remove (i, l), r)
  else cons (j, l, remove (i, r))
```

minv and maxv

- the smallest and largest values in the tree

```
minv (cons (i, l, r) -> if is_empty (l) then i  
                        else minv (l)  
  
maxv (cons (i, l, r) -> if is_empty (r) then i  
                        else maxv (r))
```

code for minv

examples/c++/trees/int/tree.cc

```
/*  
 * minv - return the smallest value in the tree.  
 */  
  
int tree::minv (void)  
{  
    if (left ().is_empty ())  
        return root ();  
    else  
        return left ().minv ();  
}
```

code for maxv

[examples/c++/trees/int/tree.cc](#)

```
/*  
 * maxv - return the largest value in the tree.  
 */  
  
int tree::maxv (void)  
{  
    if (right ().is_empty ())  
        return root ();  
    else  
        return right ().maxv ();  
}
```

remove

examples/c++/trees/int/tree.cc

```
tree tree::remove (int i)
{
    if (is_empty ())
        return empty ();
    if ((root () == i) && right ().is_empty ())
        return left ();
    if ((root () == i) && left ().is_empty ())
        return right ();
    if (root () == i)
        return cons (right ().minv (),
                    left (),
                    right().remove (right ().minv ()));

    if (i < root ())
        return cons (root (), left ().remove (i), right ());
    else
        return cons (root (), left (), right ().remove (i));
}
```

Conclusion

- we have seen how functional code can be easier to formulate an algorithm.
- it does have a penalty, memory consumption
 - `cons`, `left`, and `right` all duplicate trees
- examine the code to `cons` and see the requirement to have garbage collection