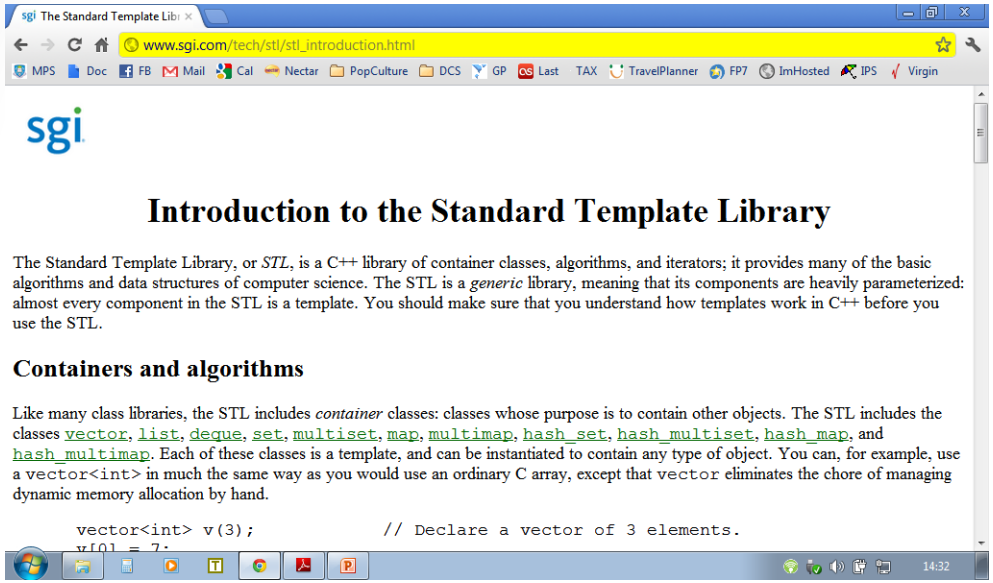
ADTs and their Applications

COM2001 Advanced Programming Techniques

C++ Standard Template Library



STL - An influential C++ software library based on

- Templates
- Containers
- Algorithms

We split the ADT into two parts. The STL container gives sorts and syntax. Generic algorithms give the semantics.

Java Containers

```
public interface Collection<E> extends Iterable<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    // optional
    boolean add(E element);
    // optional
    boolean remove(Object element);
    Iterator<E> iterator();

    // Bulk operations
    boolean containsAll(Collection<?> c);
    // optional
    boolean addAll(Collection<? extends E> c);
    // optional
    boolean removeAll(Collection<?> c);
    // optional
    boolean retainAll(Collection<?> c);
    // optional
    void clear();

    // Array operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

Interface

- Sorts
- Syntax

Instances

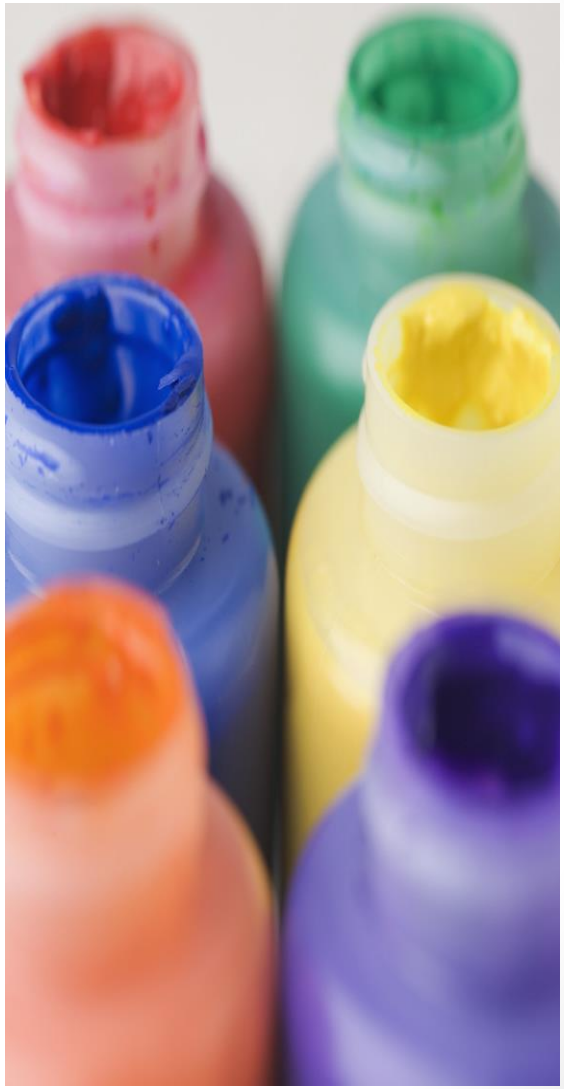
- Semantics

http://docs.oracle.com/javase/tutorial/collections/interfaces/collection.html

Containers

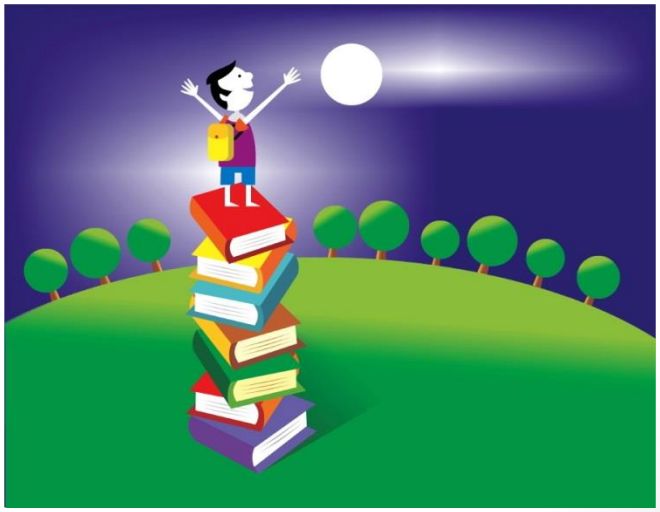
There are many types of container, including

- Stack
- Queue
  - Priority Queue
  - Deque
- List, Array
- Tuple
- Set, Multiset
- Tree, Ordered Tree, Heap



Stack

- LIFO (Last-in, First-out)
- You can interrogate one entry immediately
- All others require you to iterate through the container



Queue

- FIFO (First-in, First-out)
- You can interrogate one entry immediately
- All others require you to iterate through the container

Priority Queue

Just like a queue, except priorities are taken into account.

Operating System: Tasks with higher priority are dealt with before those with lower priority

Airlines: Customers with expensive seats board before those in the cheaper seats.

users.encs.concordia.ca

http://www.gotravel24.com/theme/feature-focus/art-queue-jumping

Deque

- **Deque = D**ouble-**E**nded **QUE**ue
- Also called a **head-tail linked list**
- A cross between a stack and a queue
- Three versions
  - Add and delete at both ends
  - Add at one end, delete at both
  - Add at both ends, delete at one
- Can access objects at either end quickly
- Other items require iteration

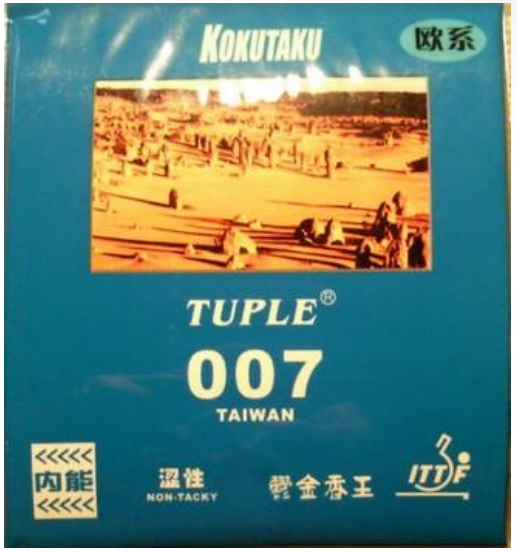
List

- Ordered structure
- Data access
  - To see a value you **traverse** the list, visiting each value in turn
  - Traversal leaves the list intact (in a stack you have to **remove** recent entries to access older ones)
- All entries should normally be of the same type
  - Some languages let you use mixed types

Array

- Ordered structure
- Random access
  - All elements can be accessed equally quickly
- Fixed (finite) capacity
  - Doesn't need to be full
- All entries of the same type
- Can be regarded as a partial function *arr*: [1..*n*] -> *Type*
- Some languages allow *arr(n)* to be defined even if *arr(n-1)* isn't

Tuple



- Like an array, except
  - Must always be full. A tuple defined to contain n elements can never contain fewer than this.
  - Different elements can have different types
  - The function that accesses the n'th entry is called the n'th **projection**

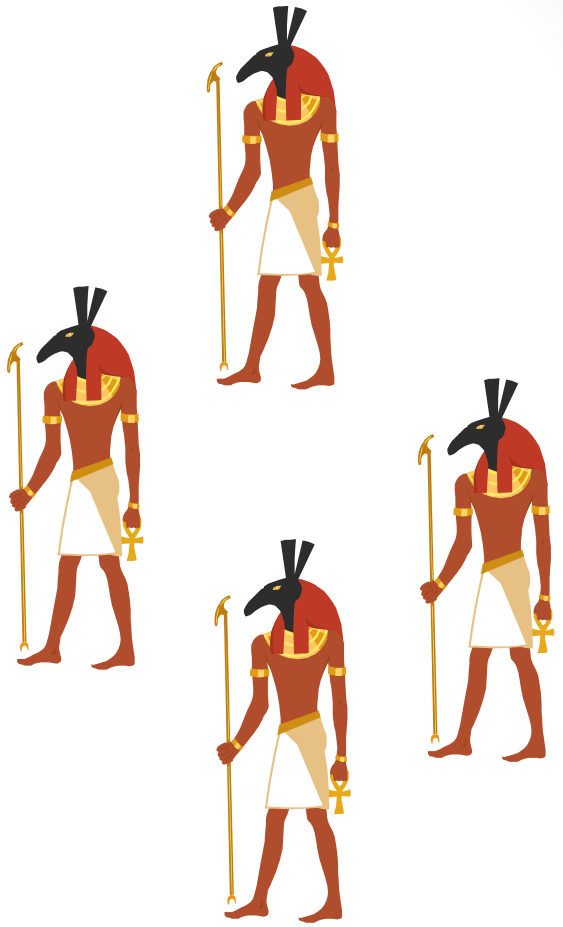
http://ooakforum.com/viewtopic.php?f=10&t=16681

Set

- An unordered structure
  - **Extensionality**: Two sets are equal if they contain the same elements
  - "A = B" means "for all relevant x, x is in A if and only if x is in B"
- No entry can occur more than once
  - Technically we can't actually tell how many times an entry occurs
- Defined by its characteristic function $\mathrm{X}_S: Universe \rightarrow \{0,1\}$ where $\mathrm{X}_S(x) = 1 \leftrightarrow x \in S$

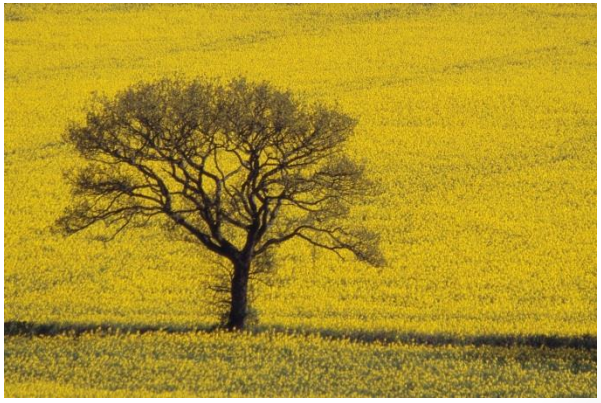
Multiset (bag)

- Like a set, except that multiple copies of each entry can be present
- Two multisets are equal provided they contain the same numbers of each element
- Can be defined by a function $\mathrm{m}: Universe \rightarrow Nat$ where $m(x)$ is a number called the **multiplicity** of x



Tree

- Partially-ordered structure
  - Any two nodes have a common ancestor
  - The root is an ancestor of every node
- Like a list, a tree has to be traversed
- If each node has at most n subtrees, it is an **n-ary tree**
  - n=2: Binary tree
  - n=3: Ternary tree
- A 1-ary tree is a **list**



Ordered Tree

- A **binary** tree in which the arrangement of entries reflects their own intrinsic ordering
  - All entries in the left subtree of a node are smaller than or equal to its own value
  - All entries in the right subtree are greater

http://www.barcham.co.uk

Heap

- A tree in which the value at a node is always greater than or equal to the values in its subtrees
- Some times we have them the other way up: the node is always smaller than or equal to the values in its subtrees

http://en.wikipedia.org/wiki/File:Cairns_on_Chandrashila_Peak,_Tungnath,_Uttarakhand.jpg

Using heaps

Later we will see how to use heaps to implement priority queues and do fast sorting

$$list \rightarrow heap \rightarrow sorted\ list$$

Complete development example

FROM ADT TO APPLICATION

The goal: using ordered trees

```
data Tree a = EmptyT | Node (Tree a) a (Tree a)
```

Typical node in a tree

Node left x right

Ordered Tree

- Every value in **left** is smaller than or equal to **x**
- Every value in **right** is bigger than **x**

What's the underlying ADT?

Tree: syntax

$$
\begin{aligned}
createT &: \rightarrow Tree\langle Entry\rangle \\
insertT &: Entry \rightarrow Tree\langle Entry\rangle \rightarrow Tree\langle Entry\rangle \\
mergeT &: Tree\langle Entry\rangle \rightarrow Tree\langle Entry\rangle \rightarrow Tree\langle Entry\rangle \\
removeT &: Entry \rightarrow Tree\langle Entry\rangle \rightarrow Tree\langle Entry\rangle \\
emptyT &: Tree\langle Entry\rangle \rightarrow Boolean \\
leftT &: Tree\langle Entry\rangle \rightarrow Tree\langle Entry\rangle \cup Msg\langle Tree\langle Entry\rangle\rangle \\
rightT &: Tree\langle Entry\rangle \rightarrow Tree\langle Entry\rangle \cup Msg\langle Tree\langle Entry\rangle\rangle \\
rootT &: Tree\langle Entry\rangle \rightarrow Entry \cup Msg\langle Entry\rangle \\
flattenT &: Tree\langle Entry\rangle \rightarrow List\langle Entry\rangle
\end{aligned}
$$

Note: *Entry* has to have an ordering defined on if *Tree<Entry>* is to be ordered.

Tree constructors

createT : → Tree<Entry>

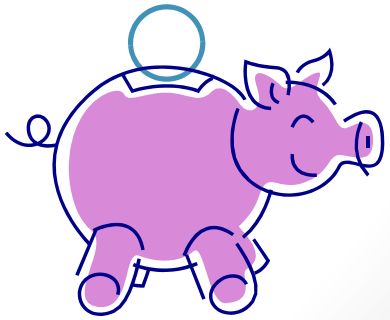
- creates a new, empty, tree

insertT : Entry → Tree<Entry> → Tree<Entry>

- inserts an entry into a tree, and returns the new tree.

mergeT : Tree<Entry> → Tree<Entry> → Tree<Entry>

- merges two trees and returns the result.
- Can be defined by recursively calling insertT on the elements of the first tree, so no separate semantics will be given



removeT, emptyT

removeT : Entry → Tree<Entry> → Tree<Entry>

- removes an entry from a tree and returns the resulting tree; if the entry isn't present in the tree to start with, returns the original tree unchanged

emptyT : Tree<Entry> → Boolean

- returns True if the tree is empty, and False otherwise

leftT, rightT, rootT

leftT : Tree<Entry> → Tree<Entry> ∪ Msg<Tree<Entry>>

- returns the left subtree of the tree; if the tree is empty, returns an error message.

rightT : Tree<Entry> → Tree<Entry> ∪ Msg<Tree<Entry>>

- returns the right subtree of the tree; if the tree is empty, returns an error message.

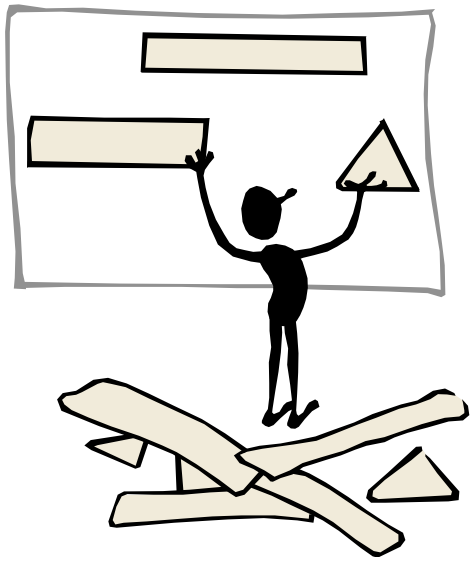
- rootT : Tree<Entry> → Entry ∪ Msg<Entry>
- returns the value at the node of the tree; if the tree is empty, returns an error message.

flattenT

flattenT : Tree<Entry> → List<Entry>

- returns the entries from the tree in a list.

This assumes List<Entry> is also defined. If we change the syntax of List<Entry>, we will have to change the semantics of Tree<Entry> to match.



Tree: semantics

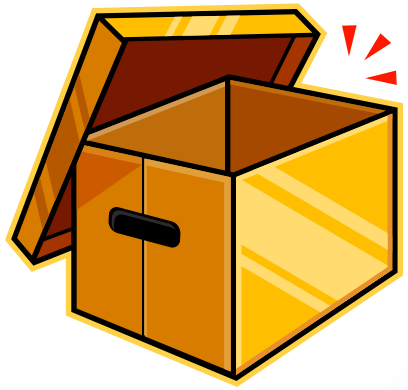
```
mergeT (createT) t = t
mergeT (insertT x s) t = insertT x (mergeT s t )
mergeT (mergeT s t ) u = mergeT s (mergeT t u)
```

As observed above, mergeT can be defined in terms of insertT, so we'll ignore it when defining the semantics

removeT

removeT : Entry → Tree<Entry> → Tree<Entry>
removes an entry from a tree and returns the resulting tree; if the entry isn't present in the tree to start with, returns the original tree unchanged

```
removeT y (createT) = createT
removeT y (insertT x t ) =
        if x == y
        then t
        else insertT x (removeT y t )
```



emptyT

emptyT : Tree<Entry> → Boolean
returns True if the tree is empty, and False otherwise

emptyT (createT) = True

emptyT (insertT x t) = False

I'm assuming that True and False have been declared as part of the syntax of Bool

leftT

leftT : Tree<Entry> → Tree<Entry> ∪ Msg<Tree<Entry>>
returns the left subtree of the tree; if the tree is empty, returns an error message.

```
leftT (createT)
      = Msg "Tree.leftT: tree is empty"
leftT (insertT x t ) =
      if x <= rootT t
      then insertT x (leftT t )
      else leftT t
```

rightT

rightT: Tree<Entry> → Tree<Entry> ∪ Msg<Tree<Entry>>
returns the right subtree of the tree; if the tree is empty, returns an error message.

```
rightT (createT)
      = Msg "Tree.rightT: tree is empty"
rightT (insertT x t ) =
      if x <= rootT t
      then rightT t
      else insertT x (rightT t )
```

rootT

rootT : Tree<Entry> → Entry ∪ Msg<Entry>
returns the value at the node of the tree; if the tree is empty, returns an error message.

```
rootT (createT)
      = Msg "Tree.rootT: tree is empty"
rootT (insertT x createT) = x
rootT (insertT x t ) = rootT t
```

flattenT

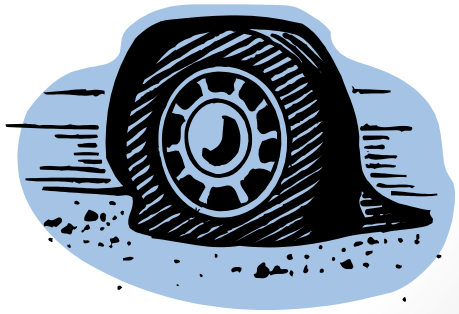
flattenT : Tree<Entry> → List<Entry>
returns the entries from the tree in a list.

flattenT (createT) = createL

flattenT (insertT x t)

= sortL (pushL x (flattenT t))

This assumes that the syntax of List<Entry> includes createL, sortL and pushL



Representation: Designing the data structure

Tree: data storage options

- We need to implement a container for the data
- The only "natural" container in Haskell is the **list**
- There **is** a simple way to store binary trees in a list

```
root  -->  arr[0]
If   node  --> arr[n]  then
       left child     -> arr[2n+1]
       right child  ->  arr[2n+2]
```

For completeness, we'll implement our own Tree container type as an algebraic data type

What data structure should we use?

As well as the data, we need to store information relating to 5 observers:

- emptyT
- leftT
- rightT
- rootT
- flattenT

```
-- Data structure
data Tree a = Tree {
      empty :: Bool,
      left :: Tree a,
      right :: Tree a,
      root :: a,
      flatten :: [a]
} deriving (Eq, Show)
```

We'll look at this stage in more detail next week.

Tree: options

```
-- Data structure
data Tree a = Tree {
        empty :: Bool,
        left :: Tree a,
        right :: Tree a,
        root :: a
        } deriving (Eq, Show)
flatten :: Tree a -> [a]
```

Rather than make observers like flatten members of the type, we can define them as functions instead.

Reduces redundancy

Tree: options

Another Option

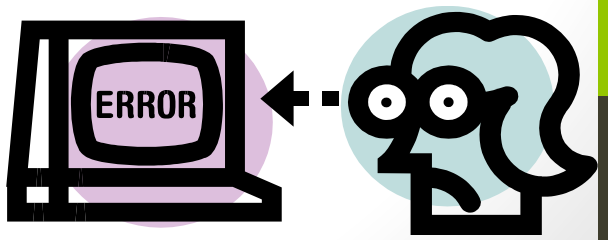
Implement Tree's constructors as Haskell constructors

```
-- Data structure
data BTree a
      = EmptyBT
      | InsertBT a (BTree a)
      | MergeBT (BTree a) (BTree a)
      deriving (Eq, Show)
```

Tree: today's choice

```
data Tree a
    = EmptyT
    | Node {
        leftT :: Tree a,
        rootT :: a,
        rightT :: Tree a
      }
```

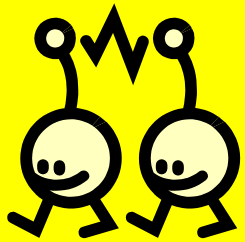
The observers leftT, rootT and rightT are only defined here for trees of the form Node l x r, so we will automatically get an error condition if they are applied to EmptyT



Implementation
1. Instance declarations

Tree: class membership

```
-- This bit is needed to make the code work with lists
instance (Eq a) => Eq (Tree a) where
      s == t = (flattenT s) == (flattenT t)
```



```
-- This bit is used to simplify testing. It's not in the ADT.
instance (Show a) => Show (Tree a) where
      show EmptyT = "*"
      show (Node EmptyT x EmptyT) = show x
      show (Node l x r) = "{" ++ (show l) ++ "} "
                  ++ (show x) ++ " {" ++ (show r) ++ "}"
```

Implementation: constructors

Note the use of foldr. We repeatedly insert the elements of flattenT s into t

```
-- constructors
createT :: Tree a
createT = EmptyT

mergeT :: Ord a => Tree a -> Tree a -> Tree a
mergeT EmptyT t = t
mergeT s t = foldr insertT t (flattenT s)
```

Tree: making it ordered

The key factor in deciding whether a tree is ordered or not is the way new entries are inserted. Notice that the entry type has to belong to Ord.

```
insertT :: Ord a => a -> Tree a -> Tree a
insertT x EmptyT = Node EmptyT x EmptyT
insertT x (Node l y r)
        | x > y   = Node l y (insertT x r)
        | x <= y = Node (insertT x l) y r
```

Tree: observers

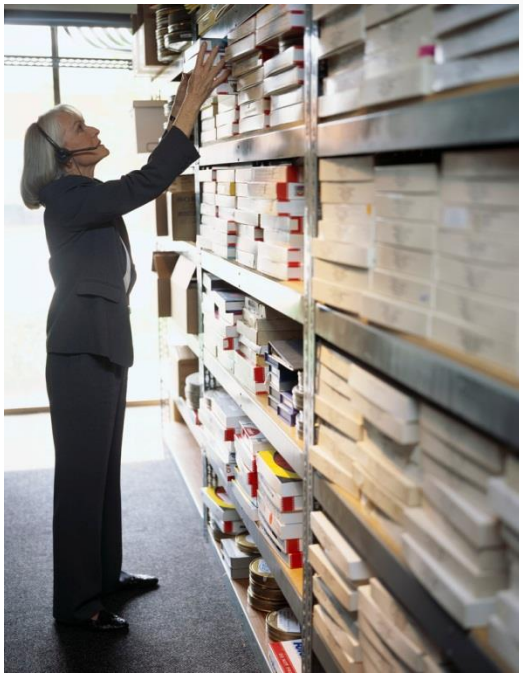
Only two of these remain to be implemented

```
--observers

emptyT :: Tree a -> Bool
emptyT EmptyT = True
emptyT _ = False

flattenT :: Tree a -> [a]
flattenT EmptyT = []
flattenT (Node l x r) = (flattenT l) ++ [x] ++ (flattenT r)
```

Tree: mutators

There is only one of these

```
-- mutator
removeT :: Ord a => a -> Tree a -> Tree a
removeT _ EmptyT = EmptyT
removeT x (Node l y r)
      | x < y     = Node (removeT x l) y r
      | x > y     = Node l y (removeT x r)
      | x == y    = mergeT l r
```



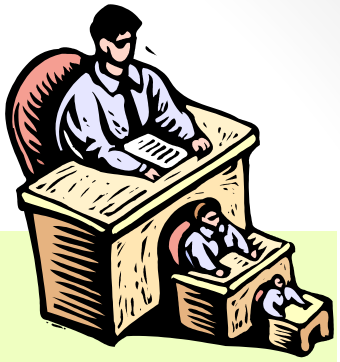
Applications

TREE-SORT VS INSERTION-SORT

Insertion sort

```
isort :: Ord a => [a] -> [a]
isort [] = []
isort (x:xs) = insertL x (isort xs)

insertL :: Ord a => a -> [a] -> [a]
insertL x [] = [x]
insertL x l@(y:ys)
      | x <= y = x:l
      | otherwise = y: insertL x ys
```

Tree sort



- Given a list
- Insert the entries one at a time into an ordered tree
- Flatten the tree to get an ordered version of the list

We seem to be doing some extra work here, because we need to construct the ordered tree.
How do these two approaches compare?

Tree sort vs. insertion sort

```
Main>     treesort [10,9..1] ---> (    642 reductions,    1160 cells)
Main>        isort [10,9..1] ---> (    723 reductions,    1135 cells)

Main>   treesort [100,99..1] ---> (  37407 reductions,   69426 cells)
Main>      isort [100,99..1] ---> (  51753 reductions,   78221 cells)

Main> treesort [1000,999..1] ---> (3523557 reductions, 6544477 cells)
Main>    isort [1000,999..1] ---> (5017053 reductions, 7532472 cells)
```

Tree sort is a bit faster than insertion sort for these examples, but the speed-up seems to be essentially linear at best. It's not particularly impressive.



HEAPS, HEAPSORT, PRIORITY QUEUES

Heaps

- A heap is just a tree in which the value at a node is always smaller than the values in its left and right subtrees.
- The only real difference between a heap and an ordered tree is the way entries are inserted.

```
insertH :: a -> Heap a -> Heap a
insertH x EmptyH = NodeH EmptyH x EmptyH
insertH x h@(NodeH l y r)
        | x < y = NodeH h x EmptyH
        | otherwise = NodeH (insert x l) y r
```

Flattening a heap

- Finding the smallest entry in a heap is easy
- It's always the value at the root of the tree

```
flattenH :: Heap a -> Tree a
flattenH EmptyH = []
flattenH (Node l x r)
        = x : flattenH (mergeH l r)
```

It's not all easy. We have to merge the left and right subtrees on each iteration of this recursive definition, and this involves additional work.

Heap sort

As before, we can sort a list of items

- Insert the items into a heap, one at a time
- Flatten the heap to get an ordered version of the list

How good is heap sort?

Much more impressive!

```
Main> heapsort     [10,9..1] ---> (    351 reductions,     608 cells)
Main> treesort     [10,9..1] ---> (    642 reductions,    1160 cells)
Main>    isort     [10,9..1] ---> (    723 reductions,    1135 cells)

Main> heapsort   [100,99..1] ---> (   3051 reductions,    5469 cells)
Main> treesort   [100,99..1] ---> (  37407 reductions,   69426 cells)
Main>    isort   [100,99..1] ---> (  51753 reductions,   78221 cells)

Main> heapsort [1000,999..1] ---> (  30051 reductions,   54970 cells)
Main> treesort [1000,999..1] ---> (3523557 reductions, 6544477 cells)
Main>    isort [1000,999..1] ---> (5017053 reductions, 7532472 cells)
```

Exercise: Heap sort appears to be dramatically faster than insertion sort. How much faster? Can you work out their worst-case complexities?

Heaps and Priority Queues

- We can use heaps to implement a priority queue
- Use a heap where the **biggest** item is always at the top
- As tasks arrive, add them to the heap
- High priority tasks always move above low priority tasks
- When we extract tasks from the heap, we'll do so in order of priority

Implementation

type PQueue e p = Heap (PQEntry e p)

```
data PQEntry e p = PQEntry {
      entry :: e,
      priority :: p
} deriving (Show)
```

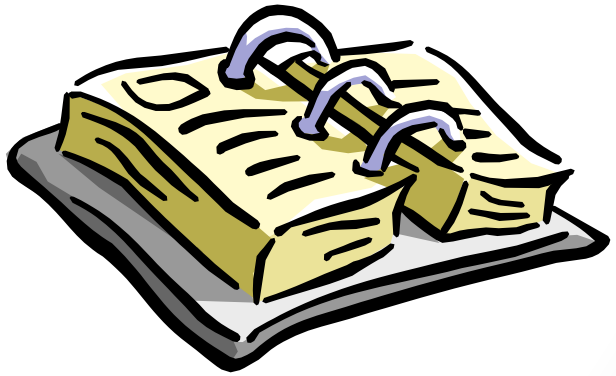
We need to define a data type representing tasks-with-priorities

```
instance (Eq p) => Eq (PQEntry e p) where
      (PQEntry _ y) == (PQEntry _ y') = (y == y')

instance (Ord p) => Ord (PQEntry e p) where
      (PQEntry _ y) <= (PQEntry _ y') = (y <= y')
```

Example: A scheduler

```
type ProcName = String
type Priority = Int
type Process = PQEntry ProcName Priority
type Scheduler = PQueue ProcName Priority
```

This is all of the required code for the implementation. All of the actual work has been done already.



Scheduling in action

```
proc1 = PQEntry "Explorer" 9 :: Process
proc2 = PQEntry "Windows" 4 :: Process
proc3 = PQEntry "Notepad" 9 :: Process
scheduler :: Scheduler
scheduler
        = insertH proc3 (insertH proc2 (insertH proc1 createH))
```

```
Main> flattenH scheduler --->
   [    PQEntry{entry="Explorer", priority=9},
        PQEntry{entry="Notepad", priority=9},
        PQEntry{entry="Windows", priority=4}   ]
```

Summary

- **STL**: ADTs and algorithms
- Containers
- Implementing **Tree a**
  - Lots of options
- Tree sort vs insertion sort
- **Heaps**
  - A tree with a less stringent ordering
- **Heap sort** vs insertion sort
- Using heaps to implement **priority queues**
- **A simple scheduler**