Remember Sorting_Machine?

Isn't it a beauty! Go ahead . . . kick the tires!

Change_To_Extraction_Phase

In    Out

Sorting_Machine Continued...

2: Push the button

1: Items go in here

3: Sorted items come out here
**Sorting Machine Continued…**

- **Type**
  - (inserting: boolean
    contents: multiset of Item
  )
- **Initial value**
  - (true, {})

---

**A Math Type: Multiset**

- *Multisets* are just like sets except that duplicates are allowed.
- In *set* theory:
  
  \[ \{2, 18, 2, 36\} = \{2, 18, 36\} \text{ and } |\{2, 18, 2, 36\}| = 3 \]

- In *multiset* theory:
  
  \[ \{2, 18, 2, 36\} \neq \{2, 18, 36\} \text{ and } |\{2, 18, 2, 36\}| = 4 \]
Sorting Machine Continued...

- Operations
  - m.Insert (x)
  - m.Change_To_Extraction_Phas e ( )
  - m.Remove_First (x)
  - m.Remove_Any (x)
  - m.Is_In_Extraction_Phas e ( )
  - m.Size ( )

Sorting With Sorting Machine
Sorting Algorithms

- Selection Sort
- Insertion Sort
- Mergesort
- Quicksort
- Heapsort
- Tree Sort
- ...

A Sort Procedure

```c
procedure Sort ( alters Queue_Of_Item& q );
/*! ensures q is permutation of #q and IS_ORDERED (q) */
```
**Math Definition**

```plaintext
math definition IS_ORDERED (s: string of Item)
    : boolean is
    for all u, v: Item
        where (<u>*<v> is substring of s)
        (ARE_IN_ORDER (u, v))
```

**An Old Math Definition**

```plaintext
math definition ARE_IN_ORDER (x: Item, y: Item)
    : boolean satisfies restriction
    for all x, y, z: Item
        (ARE_IN_ORDER (x, x) and
         (ARE_IN_ORDER (x, y) or ARE_IN_ORDER (y, x)) and
         (if (ARE_IN_ORDER (x, y) and ARE_IN_ORDER (y, z))
          then ARE_IN_ORDER (x, z)))
```
Selection Sort

procedure Remove_Min ( alters Queue_Of_Item& q, produces Item& x):
  /*!
   requires
   q /= empty_string
   ensures
   (q * <x>) is permutation of #q and
   for all y: Item
     where (y is in elements (q))
     (ARE_IN_ORDER (x, y))
  */

Selection Sort: You Give It A Try

procedure Sort ( alters Queue_Of_Item& q )
{
{
}
**Insertion Sort**

**You Give It A Try**

```cpp
procedure Insert_In_Order (
    alters Queue_Of_Item& q,
    consumes Item& x
); /*!

    requires
    IS_ORDERED (q)

    ensures
    q is permutation of (#q * <#x>) and
    IS_ORDERED (q)

    */

procedure Sort (
    alters Queue_Of_Item& q
)
{
}
}
Quicksort

procedure Partition (  
   consumes Queue_Of_Item& q,  
   preserves Item& p,  
   produces Queue_Of_Item& q1,  
   produces Queue_Of_Item& q2  
);  
/*!
   ensures  
   q1 * q2 is permutation of #q and  
   for all x: Item where (x is in elements (q1))  
       (ARE_IN_ORDER (x, p)) and  
   for all x: Item where (x is in elements (q2))  
       (not ARE_IN_ORDER (x, p))  
/*!

Quicksort Continued...

procedure Combine (  
   produces Queue_Of_Item& q,  
   consumes Item& p,  
   consumes Queue_Of_Item& q1,  
   consumes Queue_Of_Item& q2  
);  
/*!
   ensures  
   q = #q1 * <#p> * #q2  
/*!
**Quicksort: You Give It A Try**

```plaintext
procedure Sort ( alters Queue_Of_Item& q )
{

}
```

**Quicksort Continued...**

```plaintext
procedure Partition ( consumes Queue_Of_Item& q,
                     preserves Item& p,
                     produces Queue_Of_Item& q1,
                     produces Queue_Of_Item& q2 )
{

}
```
Quicksort Continued...

```plaintext
procedure Combine (  
    produces Queue_Of_Item& q,  
    consumes Item& p,  
    consumes Queue_Of_Item& q1,  
    consumes Queue_Of_Item& q2  
)  
{

}  

```  

Mergesort

```plaintext
procedure Split (  
    consumes Queue_Of_Item& q,  
    produces Queue_Of_Item& q1,  
    produces Queue_Of_Item& q2  
);  
/*!  
    ensures  
    q1 * q2 is permutation of #q and  
    |q2| <= |q1| <= |q2| + 1  
*/
```
Mergesort Continued...

procedure Merge (  
   consumes Queue_Of_Item& q1,  
   consumes Queue_Of_Item& q2,  
   produces Queue_Of_Item& q
);
/*!
   requires
      IS_ORDERED(q1) and IS_ORDERED(q2)
   ensures
      q is permutation of #q1 * #q2 and
      IS_ORDERED(q)
/*!

Mergesort: You Give It A Try

procedure Sort (  
   alters Queue_Of_Item& q
    )
{
}
Sorted Machine: Inside Story

Are you ready to look under the hood of this baby?

Change To Extraction Phase

In  Out

Inside Story Continued...

- As an example, assume the representation for Sorted Machine has two fields:
  - contents_rep of type Queue_Of_Item
  - inserting_rep of type Boolean
- What are some possible implementations?
Let's Procrastinate — the Students’ Choice

<table>
<thead>
<tr>
<th>operation</th>
<th>what happens?</th>
</tr>
</thead>
<tbody>
<tr>
<td>m.Insert (x)</td>
<td></td>
</tr>
<tr>
<td>m.Change_To_</td>
<td></td>
</tr>
<tr>
<td>Extraction_Phase ()</td>
<td></td>
</tr>
<tr>
<td>m.Remove_First (x)</td>
<td></td>
</tr>
<tr>
<td>m.Remove_Any (x)</td>
<td></td>
</tr>
</tbody>
</table>

How About Eager Beavers

<table>
<thead>
<tr>
<th>operation</th>
<th>what happens?</th>
</tr>
</thead>
<tbody>
<tr>
<td>m.Insert (x)</td>
<td></td>
</tr>
<tr>
<td>m.Change_To_</td>
<td></td>
</tr>
<tr>
<td>Extraction_Phase ()</td>
<td></td>
</tr>
<tr>
<td>m.Remove_First (x)</td>
<td></td>
</tr>
<tr>
<td>m.Remove_Any (x)</td>
<td></td>
</tr>
</tbody>
</table>
Are There Other Possibilities?

<table>
<thead>
<tr>
<th>operation</th>
<th>what happens?</th>
</tr>
</thead>
<tbody>
<tr>
<td>m.Insert (x)</td>
<td></td>
</tr>
<tr>
<td>m.Change_to_ExtractionPhase ()</td>
<td></td>
</tr>
<tr>
<td>m.Remove_First (x)</td>
<td></td>
</tr>
<tr>
<td>m.Remove_Any (x)</td>
<td></td>
</tr>
</tbody>
</table>

A Heapsort Implementation

- An ARE_IN_ORDER Heap is a special kind of binary tree:
  - shape property: the tree is complete
  - ordering property: for each item x in the tree, ARE_IN_ORDER (x, child) holds for each child of x
**Complete Binary Trees**

- All levels of the tree are completely filled up except possibly the bottom level. Any "holes" in the bottom level must appear to the right of all existing items at that level.

---

**Heap Ordering Property**

- For each item $x$ in the tree:
  
  $x$

  $y$

  $z$

  $\text{ARE\_IN\_ORDER}\ (x, y)$ and $\text{ARE\_IN\_ORDER}\ (x, z)$

  $\text{ARE\_IN\_ORDER}\ (x, z)$

  $\text{ARE\_IN\_ORDER}\ (x, y)$
**Examples**

![Example Trees]

Item is Integer

\[ \text{ARE}_{-} \text{IN}_{-} \text{ORDER} \ (x, y) \equiv x \leq y \]

**Heapsort**

- Build an \( \text{ARE}_{-} \text{IN}_{-} \text{ORDER} \) heap with the items to be sorted using the specified \( \text{ARE}_{-} \text{IN}_{-} \text{ORDER} \)
- Implement Remove First so that, after removing the first item in the ordering from the heap, it restores the heap properties
How Will Remove_First Work?

Sift_Root_Down

```plaintext
procedure Sift_Root_Down (  
    alters "binary tree" t  
);  
/*@!

    requires  
        [t is a complete tree and  
         both left and right subtrees of t  
         are heaps]  

    ensures  
        [t is a heap and  
         t contains exactly the items in #t]  
/*!
```
procedure Heapify ( alters "binary tree" t ); /*!
  requires
    [t is a complete tree]
  ensures
    [t is a heap and
    t contains exactly the items in #t]
  */

procedure_body Heapify ( alters "binary tree" t )
{
}

Turning a Complete Tree
Into a Heap

Turning a Complete Tree
Into a Heap Continued...
Mapping Complete Binary Tree Positions Into Array Locations

Let's number the tree positions (top-bottom, left-right)

Which array corresponds to the tree?

Remember Array Operations?

- a.Set_Bounds (lower, upper)
- a[i] --accessor operation
- a.Lower_Bound ()
- a.Upper_Bound ()
**Insertion-Phase Container?**

- What is the effect of `a.Set_Bounds (lower, upper)`?
- At what point will the `Sorting_Machine` be ready to set the array bounds?

---

**Swapping/Comparing Two Elements in an Array**

- Given:
  ```
  object Array_Of_Item a;
  object Integer i, j;
  ```
- What's wrong with:
  - `a[i] &= a[j]` and
  - `Item_Are_In_Order::Are_In_Order (a[i], a[j])`?
- They violate the repeated argument rule
- `a[i]` and `a[j]` are references to parts of `a`'s representation: the parts could be the same (see `Partial_Map_Kernel_3`)
Swapping Two Array Elements

- Use `a.Exchange_At(i, j)` instead of `a[i] &= a[j]`
- `Exchange_At` is an Array extension
- It requires that `a.lb ≤ i, j ≤ a.ub`
- See AT/Array/Exchange_At.h in the RESOLVE_Catalog for complete specs

Comparing Two Array Elements

- Use `a.Are_In_Order_At(i, j)` instead of `Item_Are_In_Order::Are_In_Order(a[i], a[j])`
- `Are_In_Order_At` is an Array extension
- It requires that `a.lb ≤ i, j ≤ a.ub`
- See AT/Array/Are_In_Order_At.h in the RESOLVE_Catalog for complete specs