Closed Lists and Related Data Structures

Open vs. Closed Lists

- Two list models are described in the text:
  - **Open lists:**
    - Elements and sublists can be shared
    - Mutation of lists is discouraged
    - Use without side-effects, functional programming
    - Mathematically elegant
  - **Closed lists:**
    - Sharing generally not done
    - Mutation of lists is ok, because they are encapsulated
    - Use with side-effects, object-oriented programming
    - Mathematically more cumbersome
  - Closed lists can be built by wrapping open lists

Purpose of Closed Lists

- A closed list is used for its **identity and state** as an object, rather than purely for its **value** as a sequence.
- Several "clients" can access the same closed list; modifications made by one will be felt by all.
- In some cases, this is the desired behavior.
- More space-efficient, for some applications, due to in-place modification.

Closed List Implementation

- A closed list can be viewed as a "list in a box".
- Cells in the list are typically **not shared** from the outside, so they can be **mutated at will**.
- Outside access is through a mutable reference called the **"head"**.

```
import OpenList;

class ClosedList
{
  private OpenList head;
  OpenList() {}
  ...
  . . . other stuff . . .
}
```

An Empty Closed List
How to Add and Remove Elements?

- To add, must specify:
  - Item to be added
  - To where it should be added

- To remove, must specify
  - From where it should be removed

Some Typical Choices

- Always add and remove from the head.
- Always add and remove from the tail.
- Add to the tail, remove from the head.
- Add and remove at random places (how to identify where?)

Common Closed List Usages

- **Stack**
  - remove elements in reverse order of entry, i.e. last-in element is first-out ("LIFO")

- **Queue**
  - remove elements in order of entry, i.e. first-in element is first-out ("FIFO")

Stack Abstraction

```java
Stack s = new Stack();
s.push("a");
s.push("b");
s.push("c");
value = s.pop(); // value will be "c"
value = s.pop(); // value will be "b"
value = s.pop(); // value will be "a"
```

Stack Implementation (push)

BEFORE

```
head ← a ← b ← c ← d
```

push(x):

```
new Cell
```

AFTER

```
head ← a ← b ← c ← d
```

Stack Implementation (pop)

BEFORE

```
head ← a ← b ← c ← d
```

pop():

```
head
```

AFTER

```
head
```

return value

no longer used
Reading Code containing References and Pointers

- Suppose s and t are references.
- Read the assignment statement
  \[ s = t; \]
  as "make s point to where t points".
- To see why, consider

Queue Abstraction

```
Queue r = new Queue();
r.enqueue("a");
r.enqueue("b");
r.enqueue("c");
value = r.dequeue(); // value will be "a"
value = r.dequeue(); // value will be "b"
value = r.dequeue(); // value will be "c"
```

Queue Implementation

- For a queue, we usually add another reference, to the last element, for convenience.
  This element is called the tail.

Enqueue/Dequeue

- enqueue adds a new element to one end of the internal open list.
- dequeue removes an element and returns it.
- But which end is used for which?

Related Topics

- Lists of lists: No problem with OpenList, or in any framework in which lists contain Objects and are objects.
- Otherwise, need to define special type of list, tailored to the type of element being listed.

Figurative Code for Push/Pop

- \( \text{s.push(} \text{Object } x) \):
  \[
  \text{s.head } = \text{new Cell}(x, \text{s.head});
  \]
- \( \text{s.pop()} \):
  \[
  \text{Object top } = \text{s.head.value};
  \text{s.head } = \text{s.head.next};
  \text{return top;}
  \]
**Doubly-Linked Lists**

- An implementation concept
- Could use to implement double-ended queue abstraction:
  - “deque” (pron. “deck”) not to be confused with “dequeue”

```
head W E I N D tail
```

**Deque Abstraction**

- void `enqueueFront(Object)`
- Object `dequeueFront()`
- void `enqueueBack(Object)`
- Object `dequeueBack()`
- boolean `isEmpty()`

**PriorityQueue**

- Elements must implement interface `Comparable`
- `dequeue` always removes the smallest with respect to `compareTo` method

**General Doubly-Linked Lists**

- Extend usage in Deque by allowing insertion and removal at arbitrary points
- Can access the object before any object, as well as after, unlike singly-linked lists.
- Disadvantages:
  - More storage is used for the extra pointer per cell.
  - Sharing is extremely tricky; better not to share.
- Applications?
Cells make it easy to talk about various operations:

- `void insertAfter(cell, newCell)`
- `void insertBefore(cell, newCell)`
- `void remove(cell)`
- `Cell getNext()`
- `Cell getPrevious()`

### Doubly-Linked Lists as an Implementation Concept

- A problem is that `Cell` is an implementation concept, one that does not make an attractive abstraction or user interface.
- A preferable view is to think in terms of a list iterator (or cursor), which maintains an abstract position within a list and can move backward or forward.
- The iterator determines an insertion point for a new value, or point before/after a value is removed.

### Possible Abstractions for Doubly-Linked Lists

- A problem is that `Cell` is an implementation concept, one that does not make an attractive abstraction or user interface.

### Example: ListIterator

(in `java.util`)

- If `L` is a List, then `L.listIterator()` returns a ListIterator positioned at the start of the list.
- For a ListIterator:
  - `Object next()` returns the next element, if any
  - `boolean hasNext()` tells whether there is a next element
  - `Object previous()` returns the previous element, if any
  - `void set(Object)` sets the value at the current position
  - `void remove()` removes the value at the current position
  - `void add(Object)` adds a value at the current position

### Example, part 1

(complete source file)

```java
import java.util.*;

class TestListIterator {
    public static void main(String[] arg) {
        LinkedList ll = new LinkedList();    // create a LinkedList
        ll.add("north"); // add some elements
        ll.add("east");
        ll.add("south");
        ll.add("west");
        System.out.println(ll);
        ll.add(1, "northeast"); // add at position 1 of ll
        ll.addLast("northwest");
        System.out.println(ll);
        ListIterator it = ll.listIterator(); // get a new iterator for ll
        it.next(); // move the iterator
        it.next();
        it.next(); // add another element
        System.out.println(ll);
        while( it.hasNext() )   // move to end
            { it.next(); }
        it.previous(); // move back
        it.previous(); // add another element
        System.out.println(ll);
        ll.set("southeast"); // set the value at the current position
        System.out.println(ll);
    }
}
```

### Example (cont'd)

```java
while( it.hasNext() )   // move to start
    { it.previous(); }
while( it.hasNext() )   // move to end
    { it.next(); }
while( it.hasNext() )   // move to start
    { it.previous(); }
while( it.hasNext() )   // move to end
    { it.next(); }
it.remove(); // remove element
System.out.println(ll);
ll.add("northeast"); // insert
System.out.println(ll);
}
```

### Example (cont'd)

Final output:
```
[north, east, southeast, south, southwest, west, northwest]  
[north, northeast, east, southeast, south, southwest, west, northwest]  
[north, northeast, east, southeast, south, southwest, west, northwest]  
```