Closed Lists and Related Data Structures

**Closed Lists**

- A closed list is implemented as an open list in a box.
- The list is not shared from the outside, so it can be mutated.
- Outside access is through a mutable reference called the "head".

**An Empty Closed List**

**Closed List Usage**

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

- **Queue**
  - remembers elements in order of entry, i.e. first-in element is first-out (FIFO)
  - normally implemented with another reference (to tail) in addition to head

**Stack Abstraction**

- 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)**
Stack Implementation (pop)

BEFORE
![Stack diagram before pop]

pop():
- head
- no longer used
- return value

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
  \[ s \rightarrow \text{something} \quad s = t; \quad s \rightarrow \text{something} \]
  \[ t \rightarrow \text{something else} \quad \text{something} \]

Figurative Code for Push/Pop

- \( s.push(\text{Object } x): \)
  \[ s.head = \text{new Cell}(x, s.head); \]

- \( s.pop(): \)
  \[ \text{Object top} = s.head.value; \]
  \[ s.head = s.head.next; \]
  \[ \text{return top; } \]

Push \( \rightarrow \) Shove

- Define \textit{shove} to be an operation that
  adds the contents of an entire open
  list to a stack, with the first item in
  the list the last item to be added.
  How would this be coded?

Queue Abstraction

- \( \text{Queue } r = \text{new Queue}(); \)
- \( r.enqueue(\text{"a"}); \)
- \( r.enqueue(\text{"b"}); \)
- \( r.enqueue(\text{"c"}); \)
- \( \text{value} = r.dequeue(); // \text{value will be "a"} \)
- \( \text{value} = r.dequeue(); // \text{value will be "b"} \)
- \( \text{value} = r.dequeue(); // \text{value will be "c"} \)

Queue Implementation

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

![Queue diagram with head and 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?

**Exercise**
- Write the code for enqueue and dequeue.

**Related Topics**
- Lists of lists: No problem in Polya, 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.

**Doubly-Linked Lists**
- An implementation concept
- Could use to implement double-ended queues ("deques", not to be confused with "dequeue").

**Deque ("deck") Abstraction**
- void pushFront(Object)
- Object popFront()
- void pushBack(Object)
- Object popBack()
- boolean isEmpty()

**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 done.
- Applications?
Doubly-Linked Lists as an Implementation Concept

- In the implementation (as opposed to an appropriate abstraction), we realize that the list is composed of cells.
- Cells make it easy to talk about various operations.

Cells make it easy to talk about various operations:

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

Possible Abstractions for Doubly-Linked Lists

- The problem is that Cell, which is convenient for implementation, does not make an attractive abstraction.
- A preferable view is to think in terms of a list Iterator or Cursor, which maintains a 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.

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
  - `boolean hasPrevious()` tells whether there is a previous element
  - `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);
    
    Example (cont'd)
```
Example (cont'd)

```java
while( it.hasNext() ) {
    System.out.println(ll);
    ll = it.next();
}
```

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

Java Interface Abstractions

Interface Abstractions in Java

- A principal abstraction mechanism in Java is the formal concept of interface.
- An interface is like a class, except that it only declares methods, it does not implement them.
- A given class may implement the interface by giving concrete methods for each of the ones declared in the interface.
- The class asserts that it implements the interface. The compiler checks that this is the case.

Use of an Iterator

```java
public static void main(String arg[])
{
    Polylit L = Polylit.list("a", "b", "c");
    PolylitIterator It = new PolylitIterator(L);
    while( It.hasNext() ) {
        System.out.println(It.next());
    }
}
```

Interface vs. Implementation

Interface Iterator
```
boolean hasNext();
Object next();
void remove();
```

Implementing Interface
```
import java.util.Iterator;
import polya.*;
class Polylist implements Iterator{
    Polylit theList;
    public Polylist(Polylit theList){theList = theList;}
    boolean hasNext(){return theList.nonEmpty();}
    Object next(){
        Object result = theList.first();
        theList = theList.rest();
        return result;
    }
    void remove(){throw new UnsupportedOperationException();}
}
```

Not good style actually, we are forced into it if using standard Iterator.

Additional Aspects of Interface

- Any number of distinct classes can implement a given interface.
- An interface is a type, just as a class is.
- When a variable's type is that of an interface, a variable of any implementing class type may be used.
Implication of third point

The same code can be used for any type of Iterator.

```java
void IteratorPrinter(Iterator It) {
    while (It.hasNext()) {
        System.out.println(It.next());
    }
}

IteratorPrinter(new PolylistIterator(...));
IteratorPrinter(new ArrayIterator(...));
```

Reemphasis: Power of Interface

- The interface abstraction is powerful, because it does not require the user to know which implementation is being used.
- The user of a method that specifies an interface argument can thus pass an object of any class that implements the interface.

Value of Interfaces

- Interfaces force the provider of a service to give a clear specification of what the service is, independent of implementing the service.

Interface Examples with some Implementations (see Java API)

- List interface
  - LinkedList
  - Vector, ArrayList (array-based implementations)
- Set interface
  - HashSet
- SortedSet interface
  - TreeSet
- Comparable interface
  - Character, Double, Long, String, BigInteger, Date, etc.

More Interface Examples with some Implementations

- ListIterator interface
  - listIterator() returned by a LinkedList
- Enumeration interface: read-only version of Iterator
  - elements() returned by a Vector, HashTable, or Polylist
  - StringTokenizer
- Cloneable interface
  - Many classes