Closed Lists
and
Related Data Structures
Purpose of Closed Lists

- A closed list is used for its identity 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 seen by all.
- In some cases, this is the desired behavior.
- In some cases, more space-efficient due to in-place modification.
Closed List Implementation

- A closed list can be implemented as an “open list in a box”.
- Cells in the list are typically not shared from the outside, so they can be mutated.
- Outside access is through a mutable reference called the “head”.

```
+-----+     +-----+     +-----+     +-----+
|     | →   |     | →   |     | →   | →   |
+-----+     +-----+     +-----+     +-----+
|     |   |     |   |     |   |     |   |
+-----+     +-----+     +-----+     +-----+
  head  a   b   c   d
```

mutable items
An Empty Closed List

The empty open list.
Possible 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”)

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)

Before:

```
head
```

\textbf{push}(x) :

```
head
```

After:

```
```
Stack Implementation (pop)

BEFORE

pop():

AFTER
Suppose s and t are references.

Read the assignment statement

\[ s = t; \]

as “make s point to where t points”.

To see why, consider

\[ s = t; \]

something

s

t

something else

something

something else
### Figurative Code for Push/Pop

- **s.push(Object x):**
  
  ```java
  s.head = new Cell(x, s.head);
  ```

  Java notation for the head value of stack s.

- **s.pop():**
  
  ```java
  Object top = s.head.value;
  s.head = s.head.next;
  return top;
  ```
Push -> Shove

- Define **shove** to be an operation that adds the contents of an entire *open* list to a stack, with the first item in the list being the last item to be added.

- How would this be coded?
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?
Homework

- Write the code for enqueue and dequeue.
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.
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 ()

return value types
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

- A 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
import java.util.*; // contains LinkedList class

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);
    }
}

output so far:
[north, east, south, west]
[north, northeast, east, south, west, northwest]
Example (cont'd)

```java
ListIterator it = ll.listIterator();  // get a new iterator for ll
it.next();                            // move the iterator
it.next();
it.next();
it.next();
it.add("southeast");                  // add another element
System.out.println(ll);

while( it.hasNext() )                 // move to end
{
  it.next();
}

it.previous();                        // move back
it.previous();
it.add("southwest");                  // add another element
System.out.println(ll);
```

additional output:

```
[north, northeast, east, southeast, south, west, northwest]
[north, northeast, east, southeast, south, southwest, west, northwest]
```
Example (cont'd)

```java
while( it.hasPrevious() )             // move to start
    {
        it.previous();
    }

it.next();
it.next();
it.next();                          // remove element
System.out.println(ll);

it.add("northeast");                  // insert
System.out.println(ll);
```

```
final output:
[north, east, southeast, south, southwest, west, northwest]
[north, northeast, east, southeast, south, southwest, west, northwest]
```
Java Interface Abstractions
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 definition must assert that it implements the interface. The compiler checks that this is the case.
interface Iterator
{
    boolean hasNext();
    Object next();
    void remove();
}

import java.util.Iterator;
import polya.*;

class OpenListIterator implements Iterator{
    private OpenList theList;
    public OpenListIterator(OpenList theList)// constructor
    {
        this.theList = theList;
    }
    public boolean hasNext()
    {
        return !theList.isEmpty();
    }
    public Object next()
    {
        Object result = theList.first();
        theList = theList.rest();
        return result;
    }
    public void remove()
    {
        throw new UnsupportedOperationException();
    }

Not good style actually; we are forced into it if using standard Iterator.
Use of an Iterator

```java
public static void main(String arg[]) {
    OpenList L = OpenList.list("a", "b", "c");

    OpenListIterator It = new OpenListIterator(L);

    while( It.hasNext() ) {
        System.out.println(It.next());
    }
}
```
Important Aspects of 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.

- Any number of distinct classes can implement a given interface.
Implication of second 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 OpenListIterator(...));
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: List
  • Implementations: LinkedList, Vector, ArrayList

• Set
  • Interface: Set
  • Implementations: HashSet, 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 OpenList
  - StringTokenizer

- Cloneable interface
  - Many classes