CITS2200 Data Structures and Algorithms

Topic 5

Lists

- Why lists?
- List windows
- Specification
- Block representation
- Singly linked representation
- Performance comparisons

Reading: Weiss, Chapter 17.

1. Introduction

Queues and stacks are restrictive — they can only access one position within the data structure ("first" in queue, "top" of stack)

In some applications we want to access a sequence at many different positions:

eg. Text editor — sequence of characters, read/insert/delete at any point
eg. Bibliography — sequence of bibliographic entries
eg. Manipulation of polynomials
eg. List of addresses
⋮

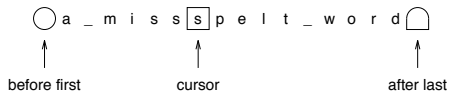
In this section, we introduce the List ADT which generalises stacks and queues.

2. List Windows

We will use the word "window" to refer to a specific position in the list:

- maintain a distinction from "reference" or "index" which are specific implementations
- maintain a distinction from "cursor" which is most commonly used as an application of a window in editing

May be several windows, eg. . .



Our List ADT will provide explicit operations for handling windows.

The following specification assumes that *w* is a Window object, defined in a separate class. Different window objects will be needed for different List implementations

⇒ a List class and a companion Window class will be developed together.

Note: A window class is generally not good software engineering practice as there is no coupling between the *List* and the *window*. Instead, modern ADTs specify list operations in terms of iterators.

3. List Specification

☐ **Constructors**

1. *List()*: Initialises an empty list with two associated window positions, *before first* and *after last*.

☐ **Checkers**

2. *isEmpty()*: Returns *true* if the list is empty.
3. *isBeforeFirst(w)*: True if *w* is over the before first position.
4. *isAfterLast(w)*: True if *w* is over the after last position.

☐ **Manipulators**

5. *beforeFirst(w)*: Initialises *w* to the before first position.
6. *afterLast(w)*: Initialises *w* to the after last position.
7. *next(w)*: Throws an exception if *w* is over the after last position, otherwise moves *w* to the next window position.

8. *previous(w)*: Throws an exception if *w* over is the before first position, otherwise moves *w* to the previous window position.
9. *insertAfter(e,w)*: Throws an exception if *w* is over the after last position, otherwise an extra element *e* is added to the list after *w*.
10. *insertBefore(e,w)*: Throws an exception if *w* is over the before first position, otherwise an extra element *e* is added to the list before *w*.
11. *examine(w)*: Throws an exception if *w* is in the before first or after last position, otherwise returns the element under *w*.
12. *replace(e,w)*: Throws an exception if *w* is in the before first or after last position, otherwise replaces the element under *w* with *e* and returns the old element.
13. *delete(w)*: Throws an exception if *w* is in the before first or after last position, otherwise deletes and returns the element under *w*, and places *w* over the next element.

3.1 Simplifying Assumptions

Allowing multiple windows can introduce problems. Consider the following use of the List ADT:

```
Window w1 = new Window();
Window w2 = new Window();

beforeFirst(w1);            // Initialise first window
next(w1);                   // Place over first element
beforeFirst(w2);            // Initialise second window
next(w2);                   // Place over first element
delete(w1);                 // Delete first element
```

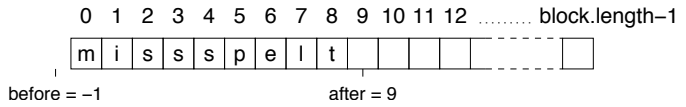
Our specification doesn't say what happens to the second window!

For now, we will assume only one window will be used at a single time.

4. Block Representation

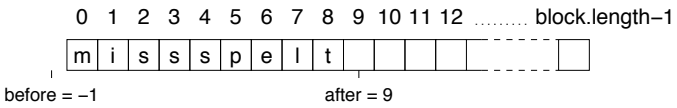
List is defined on a block (array)...

```
public class ListBlock {

  private Object[] block;   // Holds general objects
  private int before;       // An index to the before first position
  private int after;        // iAn ndex to the after last position
```



Constructor

```
public ListBlock (int size) {
  block = new Object[size];
  before = -1;
  after = 0;
}
```



Windows

Some ADTs we have created have implicit windows — eg Queue has a "window" to the first item.

There was a fixed number of these, and they were built into the ADT implementation — eg a member variable `first` held an index to the block holding the queue.

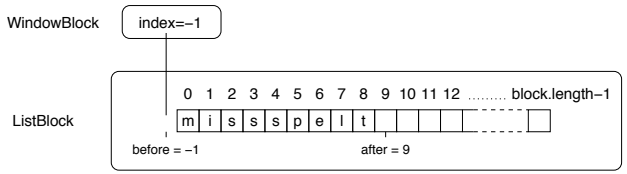
For lists, the user needs to be able to create arbitrarily many windows $\Rightarrow$ we define these as separate objects.

For the block representation, they just hold an index...

```
public class WindowBlock {
  public int index;
  public WindowBlock () {}
}
```

The index is then initialised by a call to *beforeFirst* or *afterLast*.

```
public void beforeFirst (WindowBlock w) {w.index = before;}
```



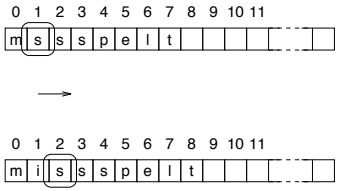
`next` and `previous` simply increment or decrement the window position...

```
public void next (WindowBlock w) throws OutOfBounds {
  if (!isAfterLast(w))  w.index++;
  else
    throw new OutOfBounds("Calling next after list end.");
}
```

`examine` and `replace` are simple array assignments/lookups.

Insertion and deletion may require moving many elements
⇒ worst-case performance — *linear* in size of block

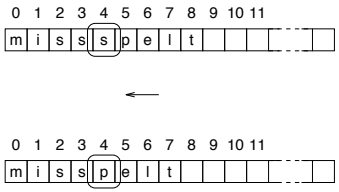
eg. `insertBefore`



From an 'abstract' point of view, the window hasn't moved — it's still over the same element. However, the 'physical' location has changed.

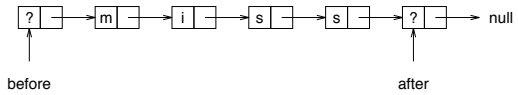

```
public void insertBefore (Object e, WindowBlock w) throws
OutOfBounds, Overflow {
  if (!isFull()) {
    if (!isBeforeFirst(w)) {
      for (int i = after-1; i >= w.index; i--)
        block[i+1] = block[i];
      after++;
      block[w.index] = e;
      w.index++;
    }
    else throw new OutOfBounds ("Inserting before start.");
  }
  else throw new Overflow("Inserting in full list.");
}
```


eg. `delete`



The window has moved from an 'abstract' point of view, although the 'physical' location is the same.

5. Singly Linked Representation



Uses two *sentinel* cells for before first and after last:

- *previous* and *next* always well-defined, even from first or last element
- Constant time implementation for *beforeFirst* and *afterLast*

Empty list just has two sentinel cells...

```
public class ListLinked {

  private Link before;
  private Link after;

  public ListLinked () {
    after = new Link(null, null);
    before = new Link(null, after);
  }

  public boolean isEmpty () {return before.successor == after;}
```


Windows

```
public class WindowLinked {
  public Link link;
  public WindowLinked () {link = null;}
}
```

eg.

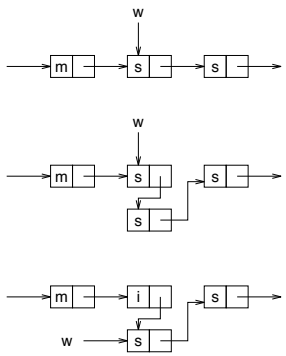
```
public void beforeFirst (WindowLinked w) {w.link = before;}

public void next (WindowLinked w) throws OutOfBounds {
  if (!isAfterLast(w))  w.link = w.link.successor;
  else
    throw new OutOfBounds("Calling next after list end.");
}
```

Why don't we just use a `Link` here?

`insertBefore` and `delete`

Problem — need *previous* cell! To find this takes linear rather than constant time.

One solution: insert *after* and swap items around




```
public void insertBefore (Object e, WindowLinked w) throws
OutOfBounds {
  if (!isBeforeFirst(w)) {
    w.link.successor = new Link(w.link.item, w.link.successor);
    if (isAfterLast(w))  after = w.link.successor;
    w.link.item = e;
    w.link = w.link.successor;
  }
  else throw new OutOfBounds ("inserting before start of list");
}
```

Alternative solution: define window value to be the link to the cell previous to the cell in the window — Exercise.

5.1 Implementing previous

To find the previous element in a singly linked list we must start at the first sentinel cell and traverse the list to the current window, while storing the previous position...

```
public void previous (WindowLinked w) throws
OutOfBounds {
  if (!isBeforeFirst(w)) {
    Link current = before.successor;
    Link previous = before;
    while (current != w.link) {
      previous = current;
      current = current.successor;
    }
    w.link = previous;
  }
  else throw new OutOfBounds ("Calling previous before start of list.");
}
```

This is called *link coupling* — linear in size of list!

Note: We have assumed (as in previous methods) that the window passed is a valid window to *this* List.

In this case if this is not true, Java will throw an exception when the `while` loop reaches the end of the list.

6. Performance Comparisons

Operation	Block	Singly linked
List	1	1
isEmpty	1	1
isBeforeFirst	1	1
isAfterLast	1	1
beforeFirst	1	1
afterLast	1	1
next	1	1
previous	1	n
insertAfter	n	1
insertBefore	n	1
examine	1	1
replace	1	1
delete	n	1

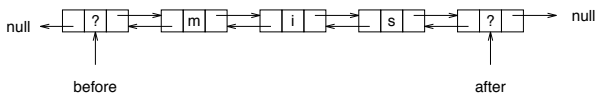
In addition to a *fixed maximum length*, the block representation takes *linear time for insertions and deletions*.

The singly linked representation wins on all accounts except *previous*, which we address in the next sections...

7. Doubly Linked Lists

Singly linked list: *previous* is linear in worst case — may have to search through the whole list to find the previous window position.

One solution — keep references in both directions!



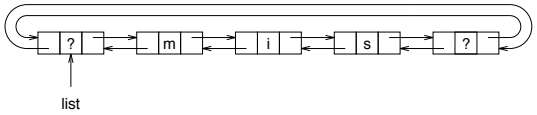
Called a *doubly linked list*.

Advantage: *previous* is similar to *next* — easy to program and constant time.

Disadvantage: extra storage required in each cell, more references to update.

8. Circularly Linked Lists

The doubly linked list has two wasted pointers. If we link these round to the other end...



Called a *circularly linked list*.

Advantages: (over doubly linked)

- Only need a reference to the first sentinel cell.
- Elegant!

Redefine Link

```
public class LinkDouble {
  public Object item;
  public LinkDouble successor;
  public LinkDouble predecessor;         // extra cell
```

Redefine List

```
public class ListLinkedCircular {
  private LinkDouble list;              // just one reference
```


Code for previous

```
public void previous (WindowLinked w) throws
OutOfBounds {
  if (!isBeforeFirst(w))  w.link = w.link.predecessor;
  else throw
    new OutOfBounds("calling previous before start of list ");
}
```

Cf. previous *previous*!

9. Performance — List

Operation	Block	Singly linked	Doubly linked
List	1	1	1
isEmpty	1	1	1
isBeforeFirst	1	1	1
isAfterLast	1	1	1
beforeFirst	1	1	1
afterLast	1	1	1
next	1	1	1
previous	1	n	1
insertAfter	n	1	1
insertBefore	n	1	1
examine	1	1	1
replace	1	1	1
delete	n	1	1

We see that the doubly linked representation has superior performance. This needs to be weighed against the additional space overheads.

Rough rule

- *previous* commonly used $\Rightarrow$ doubly (circularly) linked
- *previous* never or rarely used $\Rightarrow$ singly linked

10. Summary

- Lists generalise stacks and queues by enabling insertion, examination, and deletion at any point in the sequence.
- Insertion, examination, and deletion are achieved using *windows* on the list.
- Explicit window manipulation is included in the specification of our List ADT.
- A block representation restricts the list size and results in linear time performance for insertions and deletions.

- A singly linked representation allows arbitrary size lists, and offers constant time performance in all operations except *previous*.
- *Doubly (and Circularly) Linked Lists* have constant time performance on all operations but needs extra space