CS61B Lecture #18

- **Administrative:**
  - Initial test run of Project #1 tonight.
  - No homework due Wednesday, but there *will* be a lab devoted to test review.
- **Today:**
  - Array vs. linked: tradeoffs
  - Sentinels
  - Specialized sequences: stacks, queues, deques
  - Circular buffering
  - Recursion and stacks
  - Adapters
- **Readings for Today:** *DS(IJ),* Chapter 4;
- **Readings for Next Topic:** *DS(IJ),* Chapter 5;

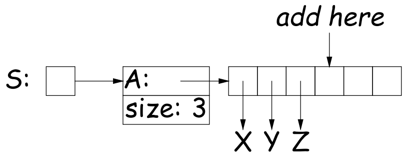
Arrays and Links

- Two main ways to represent a sequence: array and linked list
- In Java Library: `ArrayList` and `Vector` vs. `LinkedList`.
- Array:
  - Advantages: compact, fast ($\Theta(1)$) *random access* (indexing).
  - Disadvantages: insertion, deletion can be slow ($\Theta(N)$)
- Linked list:
  - Advantages: insertion, deletion fast once position found.
  - Disadvantages: space (link overhead), random access slow.

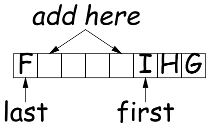
Implementing with Arrays

- Biggest problem using arrays is insertion/deletion in the *middle* of a list (must shove things over).
- Adding/deleting from ends can be made fast:
  - Double array size to grow; amortized cost constant (Lecture #15).
  - Growth at one end really easy; classical stack implementation:

```
S.push ("X");
S.push ("Y");
S.push ("Z");
```



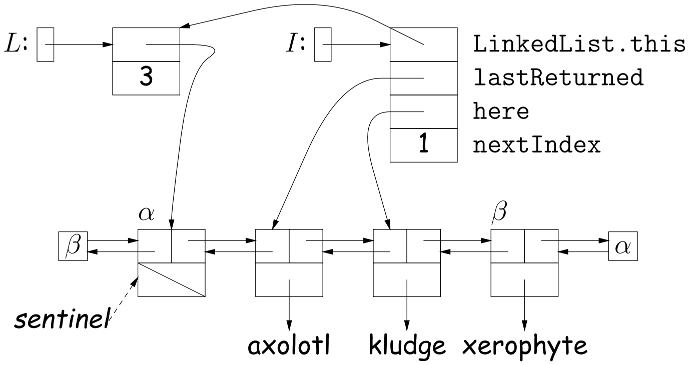
  - To allow growth at either end, use *circular buffering:*



  - Random access still fast.

Linking

- Essentials of linking should now be familiar
- Used in Java `LinkedList`. One possible representation:

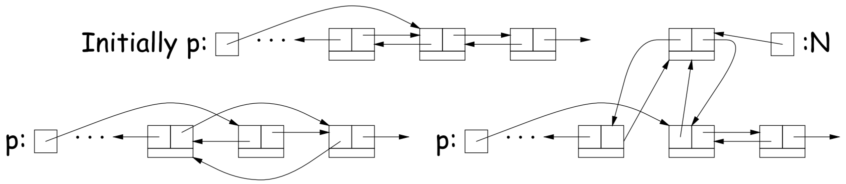


```
L = new LinkedList<String>();
L.add("axolotl");
L.add("kludge");
L.add("xerophyte");
I = L.listIterator();
I.next();
```

Clever trick: Sentinels

- A *sentinel* is a dummy object containing no useful data except links.
- Used to eliminate special cases and to provide a fixed object to point to in order to access a data structure.
- Avoids special cases ('**if**' statements) by ensuring that the first and last item of a list always have (non-null) nodes—possibly sentinels—before and after them:
-

```
// To delete list node at p:    // To add new node N before p:
p.next.prev = p.prev;           N.prev = p.prev; N.next = p;
p.prev.next = p.next;           p.prev.next = N;
                                p.prev = N;
```



Specialization

- Traditional special cases of general list:
  - **Stack**: Add and delete from one end (LIFO).
  - **Queue**: Add at end, delete from front (FIFO).
  - **Dequeue**: Add or delete at either end.
- All of these easily representable by either array (with circular buffering for queue or deque) or linked list.
- Java has the `List` types, which can act like any of these (although with non-traditional names for some of the operations).
- Also has `java.util.Stack`, a subtype of `List`, which gives traditional names ("push", "pop") to its operations. There is, however, no "stack" interface.

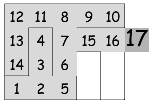
Stacks and Recursion

- Stacks related to *recursion*. In fact, can convert any recursive algorithm to stack-based (however, generally no great performance benefit):
  - Calls become "push current variables and parameters, set parameters to new values, and loop."
  - Return becomes "pop to restore variables and parameters."

```
findExit(start):
  if isExit(start)
    FOUND
  else if (! isCrumb(start))
    leave crumb at start;
    for each square, x,
      adjacent to start:
        if legalPlace(x)
          findExit(x)
```

```
findExit(start):
  S = new empty stack;
  push start on S;
  while S not empty:
    pop S into start;
    if isExit(start)
      FOUND
    else if (! isCrumb(start))
      leave crumb at start;
      for each square, x,
        adjacent to start (in reverse):
          if legalPlace(x)
            push x on S
```

Call: findExit(0)
Exit: 16



12	11	8	9	10	
13	4	7	15	16	17
14	3	6			
1	2	5			

Design Choices: Extension, Delegation, Adaptation

- The standard `java.util.Stack` type *extends* `Vector`:

```
class Stack<Item> extends Vector<Item> { void push (Item x) { add (x); } ... }
```

- Could instead have *delegated* to a field:

```
class ArrayStack<Item> {
   private ArrayList<Item> repl = new ArrayList<Item> ();
   void push (Item x) { repl.add (x); } ...
}
```

- Or, could generalize, and define an *adapter:* a class used to make objects of one kind behave as another:

```
public class StackAdapter<Item> {
   private List repl;
   /** A stack that uses REPL for its storage. */
   public StackAdapter (List<Item> repl) { this.repl = repl; }
   public void push (Item x) { repl.add (x); } ...
}

class ArrayStack<Item> extends StackAdapter<Item> {
  ArrayStack () { super (new ArrayList<Item> ()); }
}
```