

## CS61B Lecture #18

### Administrative:

- Handouts: Lecture notes, HW #6 (there is no HW #5).

### Today:

- Array vs. linked: tradeoffs
- Sentinels
- Specialized sequences: stacks, queues, dequeues
- Circular buffering
- Recursion and stacks
- Adapters

Readings for Today: *Data Structures*, Chapter 4

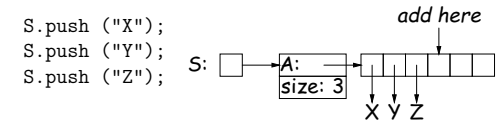
Readings for Next Topic: *Data Structures*, Chapter 5

## Arrays and Links

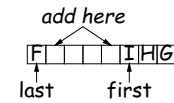
- Two main ways to represent a sequence: array and linked list
- In Java Library: ArrayList and Vector vs. LinkedList.
- Array:
  - Advantages: compact, fast ( $\Theta(1)$ ) random access (indexing).
  - Disadvantages: insertion, deletion can be slow ( $\Theta(N)$ )
- Linked list:
  - Advantages: insertion, 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 #14).
  - Growth at one end really easy; classical stack implementation:



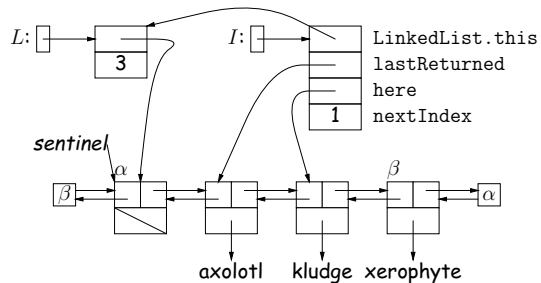
- To allow growth at either end, use *circular buffering*:



- Random access still fast.

## Linking

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



```
L = new LinkedList();
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 by ensuring that the first and last item of a list are not special.
- Example: for representation of last slide, here's deletion:

```
// To delete list node pointed to by p:
p.next.prev = p.prev;
p.prev.next = p.next;
```

If there are sentinels, this works for *all* list elements, even first and last.

## 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.

## Stacks and Recursion

- Stacks of particular interest for close relation to *recursion*.
- Recursion, in fact, implemented using "runtime stack."
- Can convert any recursive algorithm to stack-based:
  - Calls become "push current state, set parameters to new values, and loop."
  - Not important for speed, but interesting.
- Example (pseudo-code, with a little cheating):

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

11	10	7	8	9
12	3	6	14	15
13	2	5		16
0	1	4		

Last modified: Fri Oct 12 10:20:01 2001

CS61B: Lecture #18 7

## Design Choices: Specializing by Extension

- The standard `java.util.Stack` type is an *extension* of `Vector` (a kind of `List`).
- Thus, has both the operations of a stack, and those of a list.
- Not really a terribly pure way to realize a subset of an interface.
- Could have outlawed non-stack operations of `Stack`, but no particular point to that:

```
public Object get (int k)
{ throw new UnsupportedOperationException (); }
public Object peek () {
    return super.get (size ()-1);
}
```

Last modified: Fri Oct 12 10:20:01 2001

CS61B: Lecture #18 8

## Design Choices: Adapters

- Or, could use field to hold a representation:

```
class ArrayStack {
    private List repl = new ArrayList ();
    void push (Object x) { repl.add (x); }
    ...
}
```

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

```
class StackAdapter { // NAME CHANGED FROM LECTURE
    private List repl;
    ArrayAdapter (List repl) { this.repl = repl; }
    void push (Object x) { repl.add (x); }
    ...
}
```

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

Last modified: Fri Oct 12 10:20:01 2001

CS61B: Lecture #18 9