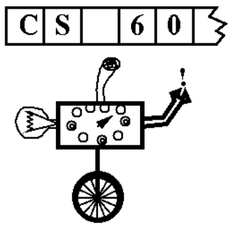
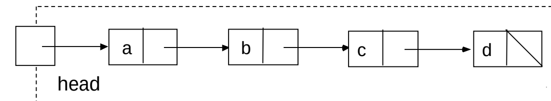


Linked Lists and Interfaces

October 8, 2001

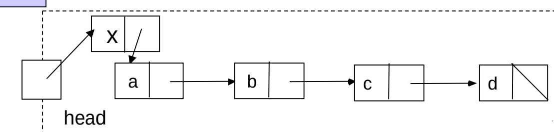
Stack Implementation: push

BEFORE



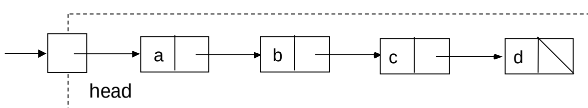
`push(x):`

AFTER

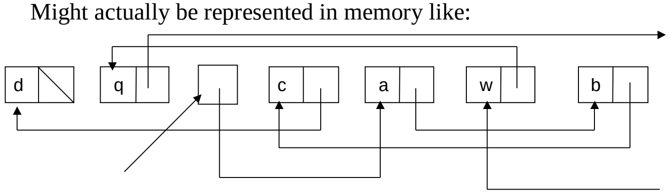


These Diagrams are *Abstractions*

Remember, a diagram like:

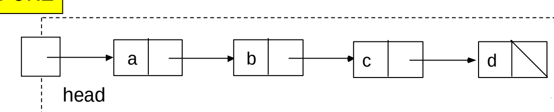


Might actually be represented in memory like:



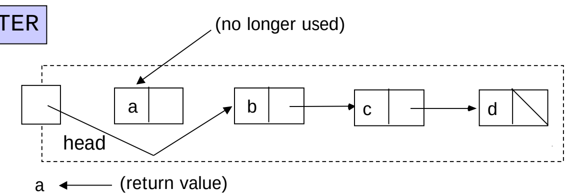
Stack Implementation: pop

BEFORE



`pop():`

AFTER



Code Excerpts for Push/Pop

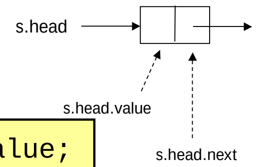
- `s.push(Object x):`

```
s.head = new Cell(x, s.head);
```

Notation for the head value of stack s.

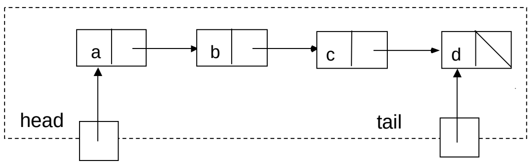
- `s.pop():`

```
Object top = s.head.value;
s.head = s.head.next;
return top;
```



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

- Sketch the code for enqueue and dequeue.

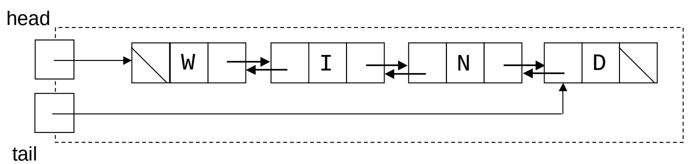
Deque ("deck") Abstraction

```
void    pushFront(Object)
Object  popFront()
void    pushBack(Object)
Object  popBack()
boolean isEmpty()
```

- A deque should not be confused with a "dequeue".

Doubly-Linked Lists

- An **implementation** concept
- Could use to implement double-ended queues



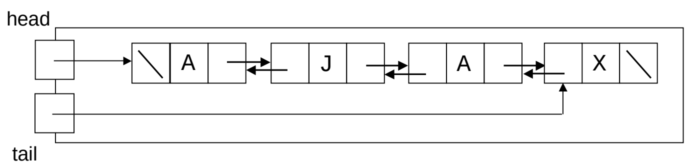
Observation on Doubly-Linked Lists

- Could extend deques 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

- Cells make it easy to talk about various operations:

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



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 interface` (in `java.util`)

- Idea:
  - List implementations can provide a way of creating a `ListIterator` positioned at the start of the list.
  - E.g., `LinkedList` class has a `listIterator()` method
- The `ListIterator` interface includes:
  - `Object next()` returns the next element, if any
  - `boolean hasNext()` whether there is a next element
  - `Object previous()` returns the previous element, if any
  - `boolean hasPrevious()` whether there is a previous element
  - `void set(Object)` sets the value at the current position
  - `void remove()` removes value at the current position (last value returned by `next`/`previous`)
  - `void add(Object)` adds a value at the current position (before next element)

Example

```
import java.util.*;

class TestListIterator
{
public static void main(String arg[])
  {
  LinkedList ll = new LinkedList();  // create a LinkedList
                                     // (doubly-linked, actually)

  ll.add("northeast");               // add some elements
  ll.add("east");                    //  (Each is put at the end
  ll.add("south");                   //   of the list)
  ll.add("west");
  System.out.println(ll);

  ll.add(1, "northwest");            // add at position 1 of ll
  ll.addFirst("north");
  System.out.println(ll);
```

output so far:

```
[northeast, east, south, west]
[north, northeast, east, south, west, northwest]
```

Example (cont'd)

```
ListIterator it = ll.listIterator();  // get a new iterator for ll
it.next();                            // move the iterator
it.next();                            //   (Also returns & ignores
it.next();                            //    list elements)
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)

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

  it.next();
  it.next();
  it.remove();                         // 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]
```

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.

Interface vs. Implementation

```
interface Iterator
{
boolean hasNext();
Object next();
void remove();
}
```

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

class PolylistIterator implements Iterator
{
private Polylist theList;

public PolylistIterator(Polylist theList)// constructor
    {
    this.theList = theList;
    }

public boolean hasNext()
    {
    return theList.nonEmpty();
    }

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

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

    PolylistIterator It = new PolylistIterator(L);

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

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.

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

Interface/Implementation Examples

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

More Examples

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