LINKED DATA STRUCTURES

Linked Lists

A **linked list** is a structure in which objects refer to the same kind of object, and where:

- the objects, called **nodes**, are linked in a linear sequence.
- we keep a reference to the first node of the list (called the "front" or "head").

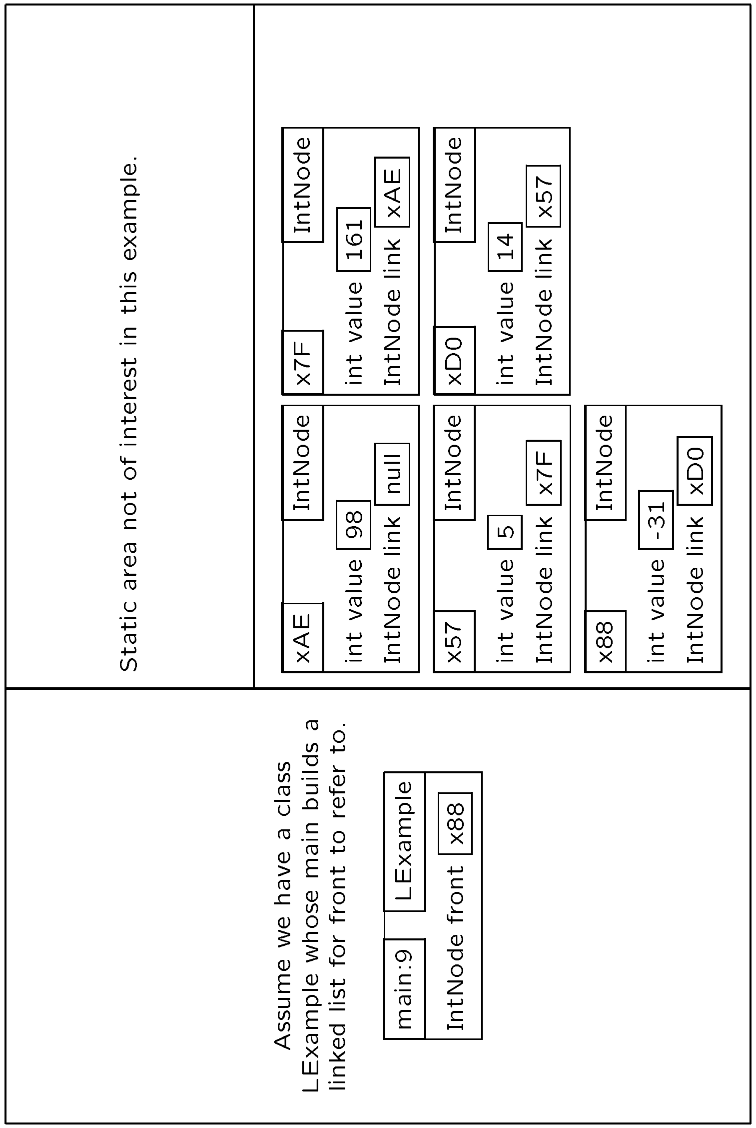
The nodes are used to store data, for example here is a class for nodes in a linked list of `int`s:

```
public class IntNode {
    public int value;
    public IntNode link;
    public IntNode(int v) { value = v; }
}
```

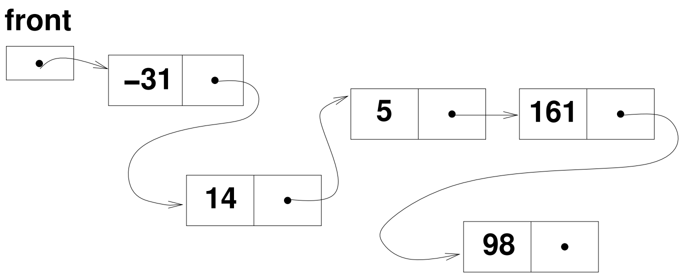
Note: having public instance variables is not as bad here as it might first look. Why?

Drawing a linked list

The next slide shows in detail how memory might look for a linked list containing 5 nodes.

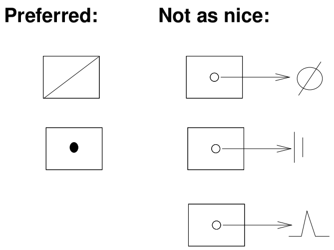


A simpler picture



When drawing such diagrams, always be careful to 'anchor' the references clearly, i.e., to show where they are stored.

Note that there are several conventions for drawing a null reference, including:



Tracing

It's easy for linked structures to get all tangled up, so you will have to develop some new debugging skills for working with them.

When writing, debugging, or understanding code with linked structures, it is extremely useful to trace by hand, using diagrams.

Exercise: Trace this code.

```
public class Example {
    public static void main(String[] args) {
        // Make a variable to refer to the list,
        // and put one element into the list.
        IntNode front = new IntNode(95);

        // Put another element into the list
        IntNode temp = new IntNode(104);
        temp.link = front;
        front = temp;

        // We can chain together dots:
        System.out.println(front.link.value); // 95
    }
}
```


Questions and Exercises

- What happens if we repeat the 3 lines "`Put another` ..." several times, using a different integer each time?
- Suppose you want to write a general method that inserts a new element at the front of a linked list of `IntNode`s, for use in the example code. What header would you use? Have you considered the case where the list is empty?
- Write your insert method.
- Write a general method that prints out the contents of a linked list of `IntNode`s.
- Write a test driver for your two methods and use it to test them.
- Think up other methods to practice using linked lists.

Hints for working with linked structures

- Trace your code by hand, using pictures, as you write it.
- If you are ever unsure of what some code does to references, make up fake memory addresses for the references and treat them like ordinary integers.
- Every time you write a line like this:

  ```
  blah.bloop
  ```

  to follow a reference, be sure the reference (in this case `blah`) is not `null`.
  Either (1) use an `if`, or (2) be sure that the reference cannot possibly be `null` in that context, and assert that fact in a comment.

Linked Lists vs Arrays

Arrays are contiguous

- In an array, the elements have to be in a contiguous (connected and sequential) portion of memory.
- Memory immediately next to the array may already be in use for something else.
- So programming languages don't generally provide for arrays that can grow and shrink in place.

Question: Doesn't Java's `Vector` do just that?

Linked lists are not contiguous

- In a linked list, the reference in each node says where to find the next one.
- So the nodes can be all over memory.

Implications

- With an array, we must choose a size once and for all. (This can waste memory, or we could run out of spaces).
- With a linked list, we can add and remove elements at will. (Each one does take up a tiny bit more space for its link field, and it may take time to 'get to' the insertion / removal spot).
- With an array, we can immediately access the n^{th} element.
- With a linked list, we cannot. We have to follow n pointers.

Neither arrays nor linked lists are best; it depends on

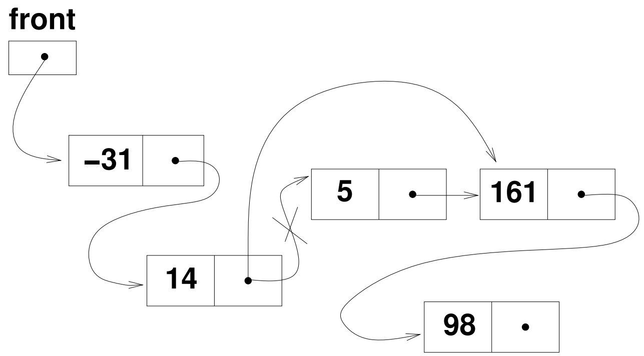
- what you're going to do most (e.g., search or update?), and
- what your priorities are (e.g., time or space?).

Java also takes time managing the location of objects in memory. It has to find an empty range of memory when we construct a new object. And it looks for unreferenced objects, to reuse their memory. But this is outside the scope of our course.

Deletion from a Linked List

Question: If the list is sorted, should we use binary search to find the element to delete?

Unlike an array, there is no need to shift any elements up to fill the "hole" left by deletion. We just unlink the unwanted element.



A Deletion Method

Implementing the queue ADT

```
/** A node in a linked list. */
class ListNode {
    public Object value;
    public ListNode link;

    /** A ListNode containing o. */
    public ListNode(Object o) {
        value = o;
    }
}

/** Note: elements are compared by equals(Object). */
interface ExtendedQueue extends Queue {
    /** Return whether I contain o. */
    public abstract boolean contains(Object o);

    /** Remove first (starting from the front)
      * occurence of o from me if it's there. */
    public abstract void delete(Object o);
}

public class LinkedQueue implements ExtendedQueue {
    /** Front node in my linked list. */
    private ListNode front;

    // Methods go here. See next slide for delete.
}
```

Method details

```java
/** Remove first (starting from the front)
  * occurence of o from me if it's there. */
public void delete(Object o) {

    ListNode previous = null;
    ListNode current = front;

    // Follow links until we fall off, or until we
    // find the node to delete.
    while (current != null &&
           !current.value.equals(o)) {
        previous = current;
        current = current.link;
    }

    // If we didn't fall off, we found it.
    // Update either the front of the list or
    // the previous link.
    if (current != null) {
        if (current == front) {
            front = current.link;
        } else {
            previous.link = current.link;
        }
    }
}
```


Questions:

- Why did we initialize `previous`?
- When implementing `Queue` with a linked list, should we keep a pointer to the end of the linked list? What about when implementing `Stack`?

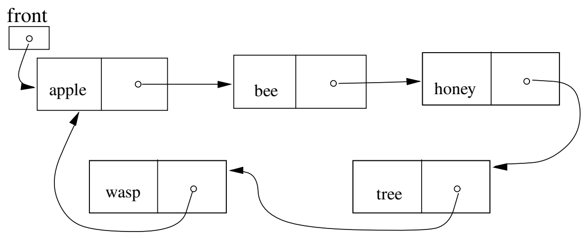
Exercises:

- Devise a thorough set of test cases for `delete`.
- Modify `delete` so that it handles `null` elements.
- In `delete`, `previous` is used to pass information from one iteration to the next: it 'remembers' information from the previous iteration. Write `delete` using only one local variable, by 'looking ahead'.
- Implement all `Queue` methods in `LinkedQueue`.
- Trace the code by hand for each method.

Other Linked Data Structures

References can be used to create many different data structures.

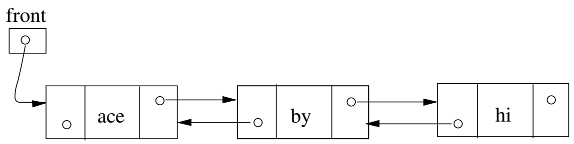
Circularly-linked lists



The last element contains a reference to the first element, rather than `null`.

Doubly-linked lists

Each element keeps both a frontwards and a backwards reference.



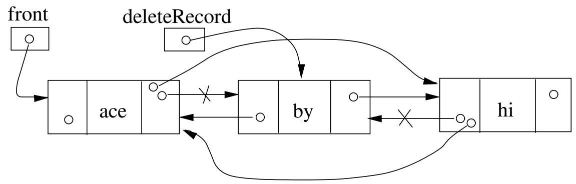

```java
/** A node for a doubly linked list of Strings. */
class DoubleNode {
    public String value;
    public DoubleNode next, prev;

    /** A DoubleNode containing v. */
    public DoubleNode(String v) {
        value = v;
    }
}
```

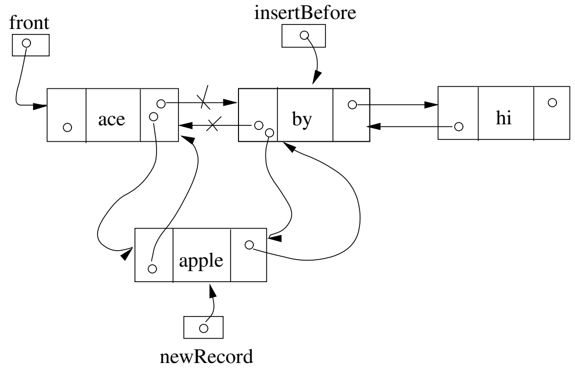
This could be useful for a list presented on screen, so we can easily write code to scroll up and down.

When traversing a doubly-linked list to prepare for insertion or deletion, there is no need to use both a current and previous reference: each node has a built-in reference to the previous element.

Deletion with a doubly-linked list:



Insertion with a doubly-linked list:



Exercise: Write complete insertion and deletion methods for a sorted doubly-linked list.

```
/** Insert a new node containing s into the appropriate
  * position in the sorted, doubly-linked list whose
  * first node is front.
  * Return the (possibly new) first node. */
public static DoubleNode insert(DoubleNode front,
                                String s)

/** Delete s, if it exists, from the doubly-linked list
  * whose first node is front.
  * Return the (possibly new) first node. */
public static DoubleNode delete(DoubleNode front,
                                String s)
```

Questions:

- The above comments aren't clear about the behaviour in certain circumstances. What are those circumstances?
- Why do we make these methods `static`?

Exercise: Think about the pros and cons of using a doubly-linked list of `Comparable`s instead of `String`s.