One way to handle a collection of elements is the array, or its richer relative, the Vector. In an array/vector the elements are stored contiguously in memory, and each element is referred to by its position: The i'th element is $a[i]$.

Another way to organize a collection of elements is a linked list, or in short list. A list is a collection of nodes; Each node stores an element, and a link to another node.

The simplest and most common type of list is one where each node stores the link to the next node in the list.

class Node {
    Object element;
    Node next;
}

A Node contains a reference to itself; it is a self-referential structure.

If each element know the element that comes after it, then all we need to know in order to traverse the list is the head of the list.

class List {
    Node head;
}

Conceptually, we think of lists as implementing a sequence of elements: the head of the list is the first element; if we follow the link from the head, this takes us to a node that we consider to be second; and so on.

Convention: when a link points to NULL, that is the end of the list.

Note: a node can point back to the first node in the list, making the list circular. We’ll come back to this.

1. IMPLEMENTING A NODE

class Node{
    Object element;
    Node next;

    //create a node storing object o and set its next link to NULL
    public Node(Object o) {
        element = o;
        next = NULL;
    }

    // create a a new node storing element o and set the next link to node n
    public Node(Object o, Node n) {
        element = o;
        next = n;
    }
}
2. **Lecture: Recursion**

next = n;

//getters
public Object element() {
    return element;
}

public Node next() {
    return next;
}

//setters
public void setElement(Object newel) {
    element = newel;
}
public void setNext(Node n) {
    next = n;
}

2. **UNDERSTANDING CLASS NODE**

Example:

Node n1 = new Node(10);

Node n2 = new Node(20, n1);

Node n3 = new Node(5, n2);

The nodes form a (linear) list if we chain them in a proper way.

Note that insertion at the front of the list is easy.

3. **LIST METHODS**

What we expect from a list:

—constructor
—insert
—delete
—isEmpty
—size

To get to an item in the list, we need to navigate to it, following the links.

To insert a node at an arbitrary position, we need to navigate there (or the node after which we want to insert needs to be given).

To delete a node from a list we need to re-link its previous node to its next node.

However, insertions and deletions of nodes can be done in $O(1)$ time at the head of the list.

csci210: Data Structures  Spring 2008.
Lecture: Lists

What we expect from a list in $O(1)$ time:

- constructor
- insert: at front
- delete: at front
- isEmpty
- size

4. IMPLEMENTING A LIST

class List
   Node head; int count;
   //create an empty list public List();
   //return how many elements in the list public int size();
   //return true if list is empty public boolean isEmpty();
   //insert this value at the head of the list public void insertAtHead(Object value);
   //delete the first value in the list and return it public Object removeAtHead();
   Analysis: all operations above take $O(1)$ time.

5. MORE OPERATIONS ON A LIST

A list can implement all operations that an array/vector can, just that some will be slower.

   //linear search: return true if the list contains a node that
   //stores this value
   //analysis: $O(n)$, where n is the size of the list
   public boolean contains(Object value)
   {
      return false; // Assume list contains only integers for simplicity
   }

   //remove the node with this value
   //analysis: $O(n)$
   public Object remove(Object e);
8. EXTENDING A LIST

Fast insertions at the end: keep tail.
Fast deletions: doubly linked list.
Avoid insert/delete checking-for-NULL cases: Circular lists. Dummy-head lists.