Problem with arrays: fixed size

- Sequence structures grow and shrink as put() and get() are done.
- If we use an array, we must create an array big enough to hold maximum number of elements that could ever exist at one time in structure (or copy elements from small arrays to bigger arrays).
- Can we use "dynamic" data structures that grow on demand?
- One solution: use linked lists and trees.
ListCell class

class ListCell {
    protected Object datum;
    protected ListCell next;

    public ListCell(Object o, ListCell n){
        datum = o;
        next = n;
    }

    //this is sometimes called the "car" method
    public Object getDatum() {
        return datum;
    }

    //this is sometimes called the "cdr" method
    public ListCell getNext(){
        return next;
    }
}

Examples of list manipulation:

Building a list

1 = new ListCell(new Integer(24),null);

1 24

p = new ListCell(t, new ListCell(s,null));

p = new ListCell(new ListCell(s,p));

Another way:

p = new ListCell(e,null);

Accessing list elements

Lists are not random-access data structures. To access an element, you must traverse list to the appropriate cell and then access the data.

Getting first element in list: p.getDatum()
Getting second element in list: p.getNext().getDatum()
Getting third element: p.getNext().getNext().getDatum()

Writing to elements can be done the same way:

p.setDatum(new Integer(53)); //sets first element of list to 53
p.getNext().setDatum(new Integer(53)); //set second element to 53
p.getNext().getNext().setDatum(new Integer(53)); //set third element

Make sure you understand all this code.
Linear search on unsorted list

```java
public static boolean search(ListCell l, Object o) {
    ListCell current = l;
    while (current != null) {
        Comparable d = (Comparable)(current.getDatum());
        if (d.compareTo(o) == 0)
            return true;
        else
            current = current.getNext();
    }
    return false;
}

System.out.println(search(p, "Dolly"); //should print true
```

Recursive method for inserting into cell:

Let us use the notation \([f, n]\) to denote a list cell whose

- datum is \(f\)
- next is \(n\).

Pseudo-code for insert:

```java
Type of insert: object x ListCell -> ListCell

English:
if ListCell argument is null, return ListCell with o and null
otherwise,
    let ListCell argument be \([f, n]\),
    if (o < f) return ListCell with o and \([f, n]\),
    otherwise return ListCell with f and ListCell returned by recursive call to insert with o and n

insert o, null = [o, null]
insert o, [f, n] = [o, [f, n]] if o < f
= [f, insert(o, n)] if o >= f
```

Let us look at Java code for this method.

Complexity of linear search on list: \(O(n)\).

Can you do binary search on a list of Comparables? No.
Not in \(O(\log(n))\) time anyway - a list is not a random-access data structure.

Inserting an object into a sorted list:

Locate cell just before insertion point, and make the change.

```
//Insert object 7 into sorted list 0-24-78-87-99
ListCell before = l.getNext(); //locate
before.setNext(new ListCell(7, before.getNext())); //update
```

```java
Inserting an object into a sorted list:
Locate cell just before insertion point, and make the change.
```

```
ListCell before = l.getNext(); //locate
before.setNext(new ListCell(7, before.getNext())); //update
```
Some list methods are easier to write iteratively.

Another simple list method: reversing a list

```java
public static ListCell reverse(ListCell l) {
    ListCell newList = null;
    ListCell oldListCursor = l;
    while (oldListCursor != null) {  
        newList = new ListCell(oldListCursor.getDatum(), newList);
        oldListCursor = oldListCursor.getNext();  
    }
    return newList;
}
```

```
//intuition: think of ‘reversing’ a pile of coins
```

```
//
```

// Iterative code: pseudo-code
locate before and victim;
before.setNext(victim.getNext());
```

// Recursive code: pseudo-code
\[
\text{delete}(o, \text{null}) = \text{null} \\
\text{delete}(o, [f,n]) = \begin{cases} 
    [f,n] & \text{if } (o < f) \\
    n & \text{if } (o \text{ equals } f) \\
    [f, \text{delete}(o,n)] & \text{if } (o > f)
\end{cases}
\]

Exercise: write code to insert and delete objects from a sorted DLL.

- In general, it is easier to work with doubly-linked lists (DLL) than regular lists.
- For example, reversing a DLL can be done very simply by swapping the previous and next references in every cell! (Write the code yourself)
- Trade-off: doubly-linked lists require more heap space than singly-linked lists.

Doubly-linked lists:

In some applications, it is convenient to have a list cell which has a reference to both its successor and predecessor in the list. Here is the declaration of a doubly-linked list cell.

```java
class DLLCell extends ListCell{
    protected ListCell previous;

    ....
}
```
Trees

Summary of lists

- Lists can grow and shrink as needed.
- Insertion and deletion from lists:
  - recursive code is easy
  - iterative code: keep track of previous list cell.
- Doubly-linked lists are easier to work with if you are willing to pay the price of extra heap storage.
- Price: no random access, so only linear search.

Some Trees

(i) General Tree

(ii) Binary tree: every node has at most two children

(iii) Not a tree

(iv) Special case of binary tree: list-like tree!
Cell for building binary trees:

// declaration of tree cell
class TreeCell {
    protected Object obj;
    protected TreeCell left;
    protected TreeCell right;

    public TreeCell(Object i) {
        obj = i;  // left and right are null by default
    }

    public TreeCell(Object i, TreeCell l, TreeCell r) {
        obj = i;
        left = l;
        right = r;
    }

    public void setObject(Object o) {
        this.obj = o;
    }

    public Object getObject() {
        return obj;
    }

    public void setLeft(TreeCell t) {
        this.left = t;
    }

    public TreeCell getLeft() {
        return left;
    }

    public void setRight(TreeCell t) {
        this.right = t;
    }

    public TreeCell getRight() {
        return right;
    }

    public String toString() {
        String lString = "";
        String rString = "";
        if (left != null) {
            lString = left.toString();
        }
        if (right != null) {
            rString = right.toString();
        }
        return lString + " " + ((Integer)obj.intValue() + " " + rString;
    } // build the second tree in previous slide

    TreeCell temp = new TreeCell(new Integer(7),
        new TreeCell(new Integer(2),
            new TreeCell(new Integer(6)));
    TreeCell myTree = new TreeCell(new Integer(6), temp, null);

In some applications, it is useful to have trees in which nodes point back to their parents (analog of doubly linked lists):

class DLTreeCell extends TreeCell {
    protected TreeCell parent;
    // getter and setter methods
}

This lets you walk from a leaf up to the root.