### Preliminaries

#### An array-based implementation

<table>
<thead>
<tr>
<th>List</th>
</tr>
</thead>
<tbody>
<tr>
<td>- items</td>
</tr>
<tr>
<td>+createList()</td>
</tr>
<tr>
<td>+isEmpty():boolean(query)</td>
</tr>
<tr>
<td>+size():integer(query)</td>
</tr>
<tr>
<td>+add(index:integer, item: ListItemType)</td>
</tr>
<tr>
<td>+remove(in index:integer)</td>
</tr>
<tr>
<td>+removeAll()</td>
</tr>
<tr>
<td>+get(index):ListItemType(query)</td>
</tr>
</tbody>
</table>
An array-based implementation

- Has a fixed size
- Data must be shifted during insertions and deletions

Array indexes

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>k-1</th>
<th>k</th>
<th>MAX_LIST-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td>Eve</td>
<td>Ann</td>
<td>Jessie</td>
<td>....</td>
<td>Jack</td>
<td>Lisa</td>
</tr>
<tr>
<td>numItems</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>K-1</td>
<td>k</td>
<td>MAX_LIST-1</td>
</tr>
</tbody>
</table>

ADT list positions

- Search an item at a given position
  - Store a list's kth item in items[k-1].

- Insert a new items at a given position
  - Shift array elements to insert an item

- Delete an item from the list
  - Blank it out, leads to gaps:
    - numItems-1 is no longer the index of the last item in the array
    - Items are spread out, method get might have to look at every cell of the array even when only a few items are present.
    - List could appear full, even when fewer then MAX_LIST items are present
  - Shift array elements to delete an item
PRELIMINARIES

(a) A linked list of integers; b) insertion; c) deletion

OBJECT REFERENCES

- **Object References**
  - **A reference variable**
    - Contains the location of an object
    - Example
      - Integer intRef;
      - intRef = new Integer(5);
  - As a data field of a class
    - Has the default value `null`
  - A local reference variable to a method
    - Does not have a default value
Object References

- When one reference variable is assigned to another reference variable, both references then refer to the same object
  - Integer p, q;
  - p = new Integer(6);
  - q = p;

- A reference variable that no longer references any object is marked for garbage collection
Object References

- An array of objects
  - Is actually an array of references to the objects

- Example
  - Integer[] scores = new Integer[30];

- Instantiating Integer objects for each array reference
  - scores[0] = new Integer(7);
  - scores[1] = new Integer(9); // and so on …

Object References

- Equality operators (== and !)
  - Compare the values of the reference variables, not the objects that they reference
    - Mynumber x = new Mynumber(9);
    - Mynumber y = new Mynumber(9);
    - z = x;
    - x == y?
    - x == z?

- Object Comparison (page 222, chapter 4)
  - Compares objects field by field
Object References

- When an object is passed to a method as an argument, the reference to the object is copied to the method’s formal parameter
  - The use of the new operator with a formal parameter in a method can produce unexpected results

```java
public void changeNumber(Mynumber n){
    n=new Mynumber(5);
}

Mynumber x=new Mynumber(9);
changeMynumber(x);
System.out.println(x);
```

- The ADT implementations and data structures that use Java references are said to be reference based.

Resizable Arrays

- The number of references in a Java array is of fixed size

```java
final int MAX_SIZE=50;
double[ ] myArray=new double[MAX_SIZE];
```

- Resizable array
  - An array that grows and shrinks as the program executes
  - An illusion that is created by using an allocate and copy strategy with fixed-size arrays

```java
double[ ] newArray=new double[capacity];
For(int i=0;i<myArray.length;i++){
    newArray[i]=myArray[i];
}
myArray=newArray;
```

- java.util.Vector & java.util.ArrayList
Linked list

- Contains nodes that are linked to one another
- A node contains both data and a link to the next item
- Access is package-private

```java
package List;
class Node {
    Object item;
    Node next;
    // constructors, accessors,
    Node(Object newItem){
        Item=newItem;
        Next=null;
    }
    Node(Object newItem, Node nextNode){
        Item=newItem;
        Next=nextNode;
    }
} // end class Node
```

Example

- Node n = new Node(new Integer(6));
- Node first = new Node(new Integer(9), n);

```java
Node n = new Node(new Integer(6));
Node first = new Node(new Integer(9), n);
```
Linked list

- What is the value of the data field next in the last node?
  - Set the field to `null`

- Nothing so far references the beginning of the linked list
  - Have an additional reference variable whose sole purpose is to locate the first node in the linked list `head`
  - `head` always exists even at times when there are no nodes in the linked list
  - `head` reference variable can be assigned `null` without first using `new`
Displaying the Contents of a Linked List

- **curr** reference variable
  - References the current node
  - Initially references the first node: `Node curr = head;`

- **To display the data portion of the current node**
  - `System.out.println(curr.item)`

- **To advance the current position to the next node**
  - `curr = curr.next;`

```java
for (Node curr = head; curr != null; curr = curr.next) {
    System.out.println(curr.item);
} // end for
```

**List traversal**
- Sequentially visits each node in the linked list until it reaches the end of the list
Inserting a Node into a Specified Position of a Linked List

Three steps to insert a new node into a linked list:

1. Determine the point of insertion
   - `curr` and `prev` reference variables
2. Create a new node and store the new data in it
   - `newNode = new Node(item);`
3. Connect the new node to the linked list by changing references
   - `newNode.next = curr;`
   - `prev.next = newNode;`

---

Inserting a Node into a Specified Position of a Linked List

- `insert` a node at the beginning of a linked list is a special case
  - `newNode.next = head;`
  - `head = newNode;`
Determining `curr` and `prev`:

```java
for (prev = null, curr = head; (curr != null) && (newValue.compareTo(curr.item) > 0);
    prev = curr, curr = curr.next) {
}
```

Deleting a Specified Node from a Linked List:

- Deleting the first node is a special case
  - head = head.next;
Deleting a Specified Node from a Linked List

Three steps to delete a node from a linked list

- Locate the node that you want to delete
  - curr and prev
- Disconnect this node from the linked list by changing references
  - prev.next = curr.next;
- Return the node to the system
  - curr.next = null;
  - curr = null;

A reference-based implementation of the ADT list

- Does not shift items during insertions and deletions
- Does not impose a fixed maximum length on the list
Comparing Array-Based and Referenced-Based Implementations

- **Size**
  - Array-based
    - Fixed size
      - Can you predict the maximum number of items in the ADT?
      - Will an array waste storage?
    - Resizable array
      - Increasing the size of a resizable array can waste storage and time
  - Reference-based
    - Do not have a fixed size
    - Do not need to predict the maximum size of the list
    - Will not waste storage

- **Storage requirements**
  - Reference-based
    - Requires more storage
    - An item explicitly references the next item in the list

- **Access time**
  - Array-based
    - Constant access time
  - Reference-based
    - The time to access the i\textsuperscript{th} node depends on i
Comparing Array-Based and Referenced-Based Implementations

- **Insertion and deletions**
  - **Array-based**
    - Require you to shift the data
  - **Reference-based**
    - Do not require you to shift the data
    - Require a list traversal

Passing a Linked List to a Method

- A method with access to a linked list’s head reference has access to the entire list
- When head is an actual argument to a method, its value is copied into the corresponding formal parameter
Processing Linked Lists Recursively

Traversal
- Recursive strategy to display a list
  - Write the first node of the list
  - Write the list minus its first node
- Recursive strategies to display a list backward
  - writeListBackward strategy
    - Write the last node of the list
    - Write the list minus its last node backward
  - writeListBackward2 strategy
    - Write the list minus its first node backward
    - Write the first node of the list

Insertion
- Recursive view of a sorted linked list
  - The linked list that head references is a sorted linked list
  - If head is null (the empty list is a sorted linked list)
  - Or
  - head.next is null (a list with a single node is a sorted linked list)
  - Or
  - head.item < head.next.item,
  - and head.next references a sorted linked list

The recursive methods should not be public
- They require the linked list's head reference as an argument
Processing Linked Lists Recursively

Insertion

```java
private static Node insertRecursive(Node headNode, java.lang.Comparable newItem) {
    if ((headNode == null) || (newItem.compareTo(headNode.item) < 0)) {
        Node newNode = new Node(newItem, headNode);
        headNode = newNode;
    } else {
        Node nextNode = insertRecursive(headNode.next, newItem);
        headNode.next = nextNode;
    }
    return headNode;
}
```

head = insertRecursive(head, newItem);

Variations of the Linked List
Variations of the linked list

- **Tail references**
  - Remembers where the end of the linked list is
  - To add a node to the end of a linked list

```java
Tail.next = new Node(request, null);
Tail = Tail.next;
```

- **Circular linked list**
  - Last node references the first node
  - Every node has a successor
    - Access the list through any node

```java
//Display the data
//list references its last node
if(list!=null){
    Node first=list.next;
    Node curr=first;
    do{
        System.out.println(curr.item);
        curr=curr.next;
    }while(curr!=first);
}
```
Variations of the linked list

- **Dummy head nodes**
  - Always present, even when the linked list is empty
  - Insertion and deletion algorithms initialize `prev` to reference the dummy head node, rather than null

\[
\text{Prev.next} = \text{curr.next};
\]

- **Doubly linked list**
  - Each node references both its predecessor and its successor
Variations of the linked list

- **Circular Doubly linked list**
  - preceding reference of the dummy head node references the last node
  - next reference of the last node references the dummy head node
  - Eliminates special cases for insertions and deletions
  - Dummy head nodes are useful in doubly linked lists

\[
\text{prev}=\text{curr}.\text{preceeding};
\]

Application

- **Application: Maintaining an Inventory**
  - **Inventory of a list of movies**
    - Movie title
    - Have value: number of DVD currently in stock
    - Want value: number of DVDs should be in stock
    - Wait list: list of names of people waiting for the title if it is sold out
Java provides classes that implement many of the more commonly used ADTs

- **Java collections framework**
  - A unified architecture for representing and manipulating collections
  - Include
    - Interfaces or ADTs representing collections
    - Implementations or concrete implementation of collection interfaces
    - Algorithms or methods that perform useful computations, e.g. sorting searching

- **JCF List interface**
  - **Java.util.List**
  - Classes implement List interface
    - **LinkedList**, **ArrayList**, and **Vector**
Read through Chapter 5