Arrays

- **Advantages**
  - easy to use, initialize, visualize
  - sequential storage of sequential data

- **Disadvantages**
  - if we exceed the array's capacity, we have to copy all data to a new array before adding more data
  - inserting and deleting data in the middle of an array is a pain!
  - requires shifting data up or down

Linked lists

- Sometimes, we want to store lists of data so that list members are only aware of their immediate “neighbors”
  - non-sequentially
- These are called *linked lists*
- Applications:
  - stacks and queues
  - firewalls/packet filters
  - advanced languages (Lisp)

Features of linked lists

- Each item is an object composed of 2 parts:
  - data
  - links to one or more items
- Items are not stored sequentially in memory
  - when a new list item is created, only the links are updated
  - no copying or shifting
- Items = “nodes”
Types of linked lists

- Single linked lists
- Double linked lists
- Circular lists
- Skip lists

Single linked lists

- Most frequently used type of linked list
- Each node contains
  - data (an object of any type)
  - link to the next node in the list
- Refer to any member of the list from the first item on the list

Example: Alphabetical list

```java
public class NameNode {
    public NameObject nameObj;  // stores first and last names
    public NameNode next;  // pointer to next item in list
    public NameNode(NameObject name) {
        this(name, null);  // create first item in list
    }
    public NameNode(NameObject name, NameNode n) {
        nameObj = name;
        next = n;
    }
}
```

Example: NameObject

```java
public class NameObject {
    public String firstName;
    public String lastName;
    public NameObject(String first, String last) {
        firstName = first;
        lastName = last;
    }
    public boolean isEqual(NameObject name) {
        if (firstName.equals(name.firstName) &&
            lastName.equals(name.lastName))
            return true;
        return false;
    }
}
```
Example: Creating the list

public class ListOfNames {
    public static void main(String[] args) {
        NameNode n1 = new NameNode(new NameObject("Drew", "Bledsoe"));
        n1.next = new NameNode(new NameObject("Brett", "Favre"));
        n1.next.next = new NameNode(new NameObject("Patrick", "Ramsey"));
        n1.next.next.next = new NameNode(new NameObject("Vinny", "Testaverde"));
    }
}

Notes

- All new nodes that are created are done through the first node, n1
- Problem: referring to nodes far down the list is awkward
  - n1.next.next.next.....
  - one solution: use references to head and tail of list (see next example)
- We've only used the first constructor so far because we've only been adding nodes to the end of the list

Example: Linked list w/ head and tail references

- ListOfNames2.java

What is the complexity of...

- adding a node to the head of the list? O(1)
- adding a node to the tail of the list? O(1)
- deleting a node from the head of the list? O(1)
- deleting a node from the tail of the list? O(n)
- deleting an arbitrary node from the list? O(n)
- determining if a node is in the list? O(n)
Generalizing linked lists

- We don't want to have to redefine our linked list structure each time we have another type of linked list
- Solution: write linked list and node classes so that the “data” field is of type `Object`
  - polymorphism again!
- Define comparison operations, etc, within the “data” object's class (like we did for `NameObject`)
- See Figure 3.9 (pg 85 in Drozdek) for an example