• **List**
  o Data structure which stores a fixed-size sequential collection of elements of the same type.
  o We've already seen two ways that you can store data in lists in Java. What are they? [array or ArrayList].
    o An array is nice but rather low level. It has no methods other than those inherited from Object.
    o An ArrayList is much smarter and more flexible.
  o Lists are such an important way to store data that there are actually several more built-in list classes we can use.

ArrayList is an implementation of List based on arrays. LinkedList is an implementation of List based on nodes.

• **Limitations of ArrayLists:**
  o When list is full need to copy whole array into a new array
  o Part of the array might be unused and so wasted space
  o Inserting into and deleting from ArrayLists requires moving potentially large amounts of data.

A **linked list** is a data structure that can store a collection of items. Like arrays, linked lists are used to store several objects of the same type. However, linked lists differ from arrays in the following ways:

- linked lists are dynamically sized; they grow and shrink as nodes are added and removed and they do not require more memory than there are objects currently in the collection.
- the nodes of a linked list need not be stored contiguously in memory.

Each element of the list is referred to as a **node**. Each node contains an **Object** that represents the data stored in the node and a reference to the **next** node in the list. The head is the first node in the list. The last node in the list refers to **null**.
Example 1

head = head.next;

Example 2

head = head.next.next;

Linked List Implementation

• Class for Nodes

Note: In Java you are allowed to define a class (say, B) inside of another class (say, A). The class A is called the outer class, and the class B is called the inner class. The purpose of inner classes is purely to be used internally as helper classes. Here is the LinkedList class with the inner Node class

class MyLinkedList {
    <fields>
    <methods>

    private class Node {

        Node next; // reference to the next node in the chain, or null if there isn't one
        Object data; // data carried by this node. could be of any type you need.
    }
}
Let's create the inner class Node. Let’s add constructors, getters and setters for both variables.

Linked List Operations

• Arbitrary Insertion/Deletion

To insert in an arbitrary position in a linked list, you must find the node that comes before the node you wish to insert (previous) as shown below:

- **Add First**

The method creates a node and prepends it at the beginning of the list.

- Make a new node with the right data, and with its next field pointing to the old head of the list
- Update the head-of-list pointer to point to the new node
• Add Last

The method appends the node to the end of the list. This requires traversing, but make sure you stop at the last node.

• Add at specific location

• Delete at specific location
Generic Classes

Suppose you have a class, such as a linked list, that can be made to work with any object. Suppose further that when someone uses that class, they want to specify what type of object it should work with. Then you want a generic class.

A generic class has a type variable that allows you to write your class for any type and allows other code to instantiate objects in your class with a specific type.

For example, the ArrayList class can take any object. But if you want an ArrayList of Strings, you can tell the compiler that this ArrayList object will hold only Strings.

This make writing the rest of the code simpler because
- You won’t need to do as much type casting.
- Instead of doing run-time checks on the type of the object, the compiler can do it for you.

In the project, your linked list class is a general class in that the type of data it stores can be any object. When you write

```java
public class MyLinkedList<T> implements Iterable<T>
```

you are telling Java that your LinkedList will have some aspect of it that depend on a type to be specified by the creator of the object. T is a variable that indicates that type.

Note: It is possible to make methods and interfaces generic, too.

(See [https://docs.oracle.com/javase/tutorial/java/generics/types.html](https://docs.oracle.com/javase/tutorial/java/generics/types.html) for a review on Generic Types)

Iterators

toString() method of MyLinkedList list the data in our Linked List.

```java
public String toString() {
    String output = " ";
    if (head != null) {
        Node current = head;
        while (current != null) {
            output += "[" + current.getData() + "]";
            current = current.next;
        }
    }
    return output;
}(You can also write an equivalent do..while loop)
Note that it uses `current = current.next` just like for loops use i++. We can actually rewrite this to use a for loop:

```java
public String toString() {
    String output = " ";

    for (Node current = head; current != null; current = current.next)
    {
        output += current.data + " ";
    }

    return output;
}
```

This code is fine. However, it uses the Nodes directly. What if the user wants to loop through the data in the list? We need some way of letting the user loop through a list. One way would be to use our method `get(int index)`.

Problem: What if the list has length 1 million? This will take way too long. We need a way of stepping through the linked list without dealing directly with Node. We can do this with Iterators!

Let’s make our MyLinkedList class a generic class: `public class MyLinkedList<T>

First, we change our class definition as:

```java
public class MyLinkedList<T> implements Iterable<T>
```

Then, we implement our own Iterator subclass

```java
private class LLIterator implements Iterator<T>
```

Once we have an iterator for the class, we can use it explicitly

```java
for (Iterator<T> iter = list.iterator();
    iter.hasNext(); )
{
    T item = iter.next();
    // 1 - can call methods of element
    // 2 - can use iter.remove() to remove the current element from the list
    ...
}
```

Simplify using the ForEach loop

```java
for (T item : list) {
    T item = iter.next();

    // 1 - can call methods of element
```
The Iterator interface contains the following methods:

- **AnyType next()** - returns the next element in the container
- **boolean hasNext()** - checks if there is a next element
- **void remove()** - (optional operation). removes the element returned by next()

**Summary**

For our linked list classes (both the in-class and project classes), there are iterators with this relationship to the class:

- The LinkedList implements Iterable, which means it has a method that creates an Iterator object.
- The LinkedList has an inner class that implements Iterator. This inner class has hasNext, next, and remove methods.

And because the linked lists have iterators, it is possible to use the ForEach loop structure.

**Doubly Linked Lists**

Doubly linked lists are linked lists where each node maintains a reference to the next node in the list and a reference to the previous node in the list.

You can move forward and backward in the list.

Some real-life applications?

Pros?

Cons?

However, maintaining two pointers for each node adds overhead with respect to memory used and maintenance of the references.
Note: Header and trailer are sentinel nodes. A sentinel node is a specifically designated node used with linked lists and trees as a traversal path terminator. This type of node does not hold or reference any data managed by the data structure.

Each node in a doubly linked list contains three fields: the data, and two pointers: prev and next.

Our class definition: `public class MyDoublyLinkedList<T> implements Iterable<T>

```java
private class Node {
    private T data;
    private Node next;
    private Node prev;

    private Node (T data) { //constructor
        this.data = data;
        this.next = null;
        this.prev = null;
    }
}
```

Example

- Original Doubly Linked List

- Delete head node

- Delete middle node
• Delete last node

Let’s implement these functions:
  • public T removeFirst ()
  • public T removeLast ()
  • public T get(int index)
  • public void add(T item)

Disclaimer: Notes adapted from previous CS 231 lecture materials at Colby College.