Overview of Lists

What type of lists are we concerned with?

The particular types of lists we are concerned with are linked lists and array lists (C++ refers to array lists as “vectors” which are not to be confused with our coordinate vectors).

Didn’t we already discuss linked lists with the Stack and Queue units?

Although Stacks and Queues can be created using linked list implementations, these implementations of linked lists are custom to the nature of these data structures.

In a general-purpose linked list, we are able to add and remove from any arbitrary node. With Stacks and Queues, we have a single add point and a single remove point.

The benefit of using a general-purpose linked list over an array is that we get arbitrary add and remove without creating a new data structure and copying over all of the elements. What we give up (or trade off) is that we no longer get constant time (O(1)) access of elements or items within the data structure.

Therefore, this type of linked list we are going to discuss is a list that allows us to get/add/remove from any node in the entire list.

What about array lists? How are these any better than a general-purpose linked list?

A general-purpose linked list will still require a worse case of O(n) time to access an item inside of it due to the fact that we must follow the references (pointers) in the nodes. Assuming the item we are trying to access is the last one in the list (worst case in a single-ended linked list), this would be n operations.

In Java, an array list is implemented behind-the-scenes using an array (resizeable) of type List. This allows constant time access to each item inside of the array.

That being said, an array list is still not a perfect data structure. It still requires resizing (up or down) with add and remove. This works similar to a Stack using an array implementation with dynamic sizing.

The biggest benefit of array lists is in how easy they are for the user to use and how flexible they are for a wide variety of tasks. However, for situations where you must have optimal performance for a given thing (item access, add or remove, etc.), you might be better of using a different data structure that is tailored for the dominant need of your algorithm.

Common Operations in a Linked List

What functionality do Linked Lists support in Java (the API version)?

The API for Linked Lists in Java is quite robust. But the following are the most important methods supported:

- add (index, item)
- clear ()
- boolean contains (item) // Returns whether the list contains this item
- item get (index)
- item peek () // Head only
- remove (index) // Remove an arbitrary item from the list (based on index)
- size ()

Example of using a Java API Linked List

What is an example of how to use the Java Linked List class in a general-purpose way?

```
package Main;

import java.util.LinkedList;

public class listtest{

	public static void main(String[] args){

		LinkedList<Integer> lst = new LinkedList<Integer>(); // Create Linked List Object

		for(int i = 0; i < 10; i++){

			lst.add(0, i); // Add 0 - 9 to list

		}

		lst.remove(0); // Remove head of linked list

		for(int i = 0; i < lst.size(); i++)

			System.out.print(lst.get(i) + " "); // Print items in list from head to tail

		System.out.println("\nList contains number 9: " + lst.contains(9));

		lst.clear(); // Remove everything from the list

		System.out.println(lst.peek()); // This should be null

	}

}
```

Creating Linked Lists from Scratch

What if we wanted to write a general-purpose Linked List class from scratch?

We should support, at least, the bare minimum methods as given in the Java version. Those seven methods can allow us to do most anything we need with regard to the general-purpose linked list. We already wrote Stack and Queue implementations using specialized linked list access, so we have no need to redo that work again.

```
package Main;

public class Llist{

	// Fields

	private Node head;

	private int n;

	private class Node{

		String item;

		Node next;

	}

	// Constructor

	public Llist(){

		head = null;

		n = 0;

	}

	// Methods

	public void add(int index, String item){

		// Create node to add

		Node X = new Node();

		X.item = item;

		X.next = null;

		// Check for strange use cases...

		if(head == null){

			if(index != 0) return; // Not a valid index relative to current list

			head = X; // Set our new node as head

			n++;

			return;

		}else if(head != null && index == 0){

			X.next = head;
```

```
            head = X;
            n++;
            return;
        }
        // Otherwise, traverse the linked list for index position
        if(index > n) return; // Cannot insert at an index larger than n
        Node current = head;
        Node previous = null;
        int i = 0;
        while (i < index) {
            previous = current;
            current = current.next;
            if (current == null) break;
            i++;
        }
        X.next = current;
        previous.next = X;
        n++;
    }
    public boolean contains(String item){
        for(int i = 0; i < n; i++){
            if(get(i).equals(item))
                return true;
        }
        return false; // For now (default)
    }
    public String get(int index){
        if(index > n-1) return "Invalid Index!";
        if(index == 0) return head.item;
        Node cur = head;
        int i = 0;
        // Traverse to find correct node to "get" item from...
        while(i < index){
            cur = cur.next;
```

```
            i++;
        }
        return cur.item;
    }
    public String peek(){
        if(head != null) return head.item;
        return "Null"; // Safe return!
    }
    public void remove(int index){
        // Strange cases first...
        if(index > n-1) return; // Cannot remove from index that doesn't exist!
        if(head != null && index == 0){
            head = head.next;
            n--;
            return;
        }
        // Otherwise, traverse the linked list for the index of node to remove...
        Node current = head;
        Node previous = null;
        int i = 0;
        while (i < index) {
            previous = current;
            current = current.next;
            if (current == null) break;
            i++;
        }
        previous.next = current.next;
        n--;
    }
    public int size(){
        return n;
    }
    public void clear(){
        head = null;
```

```
            n = 0;

        }

}
```