Computer Science II
Handout 10

ArrayLists – example 6

- Create a method that reverses the contents of an ArrayList<String>

```
public ArrayList<String> reverse(ArrayList<String> a) {
    ArrayList<String> rev = new ArrayList<String>();




    return rev;
}
```

ArrayLists – example 6

- Create a method that reverses the contents of *the same* ArrayList<String> parameter passed in

```
public ArrayList<String> reverse(ArrayList<String> a) {
     String copy;









     return a;
}
```

ArrayLists – example 6

- What would this method look like if it had return type **void**?

```
public void reverse(ArrayList<String> a) {
    String copy;









}
```

ArrayLists summary

- An ArrayList is a simple data structure that allows storing multiple Object references that are somehow related
  - It functions more-or-less like an array, except with no hassles managing size
  - The data is always stored sequentially, which is sometimes useful

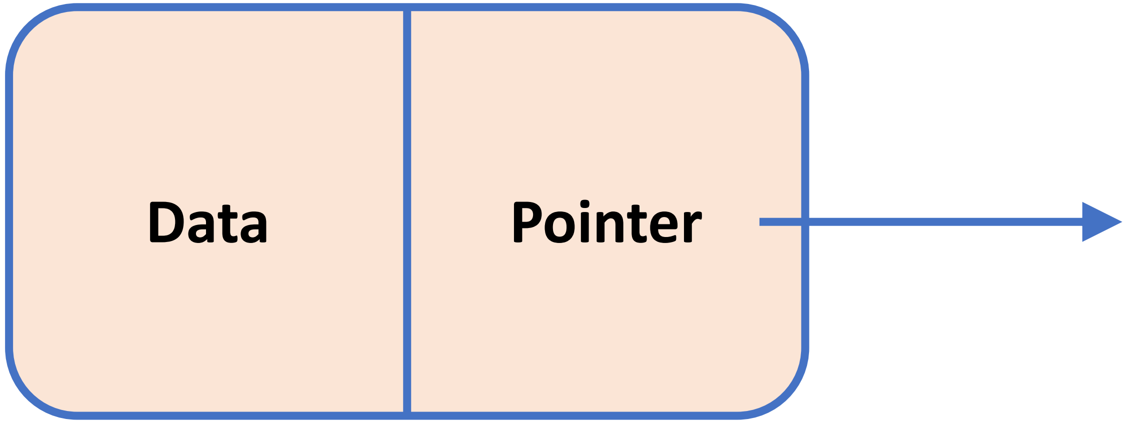
- Examples of when an ArrayList could be used:
  - Storing ingredients needed for a recipe
  - Storing several Rectangle shapes that will be displayed together
  - Storing the names of all Dalhousie students

- What happens when an element of these ArrayLists needs to be removed?

Linked Lists

- Linked Lists are another data structure
  - They are another way of arranging and accessing data

- Linked Lists are made up of “nodes” that contain both data and a pointer to the next node



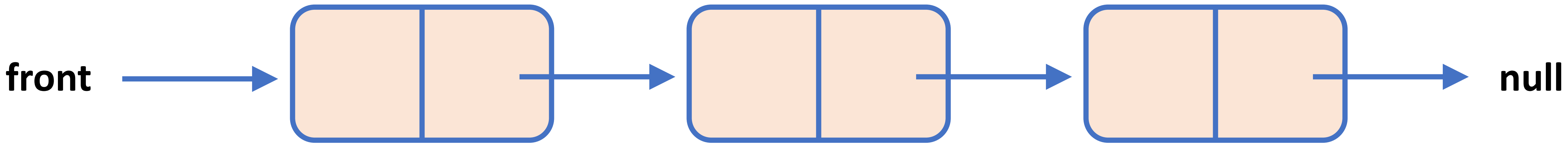
Linked Lists – nodes

- What would a Node class look like, based on this description?
  - For now, let's say our data will be a simple String value

```
public class Node {



        public Node(                              ) {


        }





}
```

Linked Lists

- The *first* or *front* node of a Linked List is indicated directly
  - This tells us where the Linked List starts
  - All other nodes are only referenced by their preceding node
- The final node will have a null pointer
- Data is then accessed by traversing nodes in the list



Linked Lists

- What would a (basic) LinkedList class look like, based on this description?
  - Assume we are using the Node class we already described

```
public class LinkedList {




    public LinkedList(                    ) {



    }

    // Other methods ...


}
```

Linked Lists

So why use Linked Lists?

- Inserting and removing elements never requires “extra” work
  - Elements do not need to be shifted, like in ArrayLists
- No initial capacity needs to be declared
- Nothing “extra” is needed for dynamically changing the number of elements
  - ArrayLists hid this extra work, but it was still there!

Linked Lists – expanding on the class

- The Java standard library already has a LinkedList class (for arbitrary Objects)
  - But we implement our own version (for Strings) to better understand the data structure!

Linked Lists – expanding on the class

- We will implement methods to do the following for our LinkedList class:

1. Add a new Node with given data (String) to the front of the List
2. Determine if the List is empty
3. A default (no-args) constructor
4. Clear the List of all elements
5. Get the data (String) of front Node (reference)
6. Get the front Node (reference)
7. Return a String representation of the full List
8. Get the size of the List
9. Remove the front Node
10. Add a new Node with given data (String) to the end of the List
11. Remove the last Node
12. Get the index for a given String (or -1 if not found)
13. Add a new Node at a given index
14. Remove the node at a given index
15. Get the Node at a given index (reference)

Linked Lists – Method 1

1. Add a new Node with given data (String) to the front of the List

- Within class LinkedList, the first method should:
  - Take a String parameter
  - Create a new instance of class Node using the String data
  - Set the **`next`** equal to the current **`front`** of the List
  - Update the LinkedList with its new **`front`** value

```
public void addToFront(String d) {
    Node n = new Node(d, front);
    front = n;
}
```

Linked Lists – Method 1

1. Add a new Node with given data (String) to the front of the List

This helps to highlight some *invariants* in our classes:

- The value of **`front`** inside a **`LinkedList`** instance always refers to the first element of the List
  - Note that the LinkedList instance does *not* store any references to any other Nodes!
  - The data is *not* within the LinkedList instance, but rather within each individual Node
- The value of **`next`** inside each **`Node`** instance always refers to the next element in a List
  - Note that each Node instance is ignorant of every other Node *except* for its successor
  - Each Node instance is also ignorant of the overall List to which it belongs

Linked Lists – Method 2

2. Determine if the List is empty

There are other *invariants* at work that decide how this method should work

- When is a LinkedList instance empty (i.e., it has no Nodes)?
  - What condition is there on the "internal state" of a LinkedList or one of its Nodes?

```
public boolean isEmpty() {



}
```

Linked Lists – "Method" 3

3. A default (no-args) constructor

A no-args constructor should create an empty LinkedList – we now know what that looks like

```
public LinkedList() {


}
```

Linked Lists – Method 4

4. Clear the List of all elements

A cleared List should be an empty List

- It is *not* the responsibility of the LinkedList to clear data from individual Nodes!

```
public void clear() {



}
```

Linked Lists – Method 5

5. Get the data (String) of front Node (reference)

A LinkedList instance always has direct access to the **`front`** Node, but no others (directly)

```
public String getFrontData() {


}
```

Linked Lists – Method 6

6. Get the front Node (reference)

A LinkedList instance always has direct access to the **`front`** Node, but no others (directly)

```
public Node getFrontNode() {


}
```

Linked Lists – Method 7

7. Return a String representation of the full List

Accessing every element means “walking” through each Node successively

- Start with the **front** Node
- How do we get the Node after **front**?
- How do we tell when we have reached the last Node?

```
public String toString() {
    String ts = "[";




    return ts + "]";
}
```

Linked Lists – Method 8

8. Get the size of the List

Accessing every element means “walking” through each Node successively

- Start with the **`front`** Node
- How do we get the Node after **`front`**?
- How do we tell when we have reached the last Node?

```
public int size() {
    int count = 0;





    return count;
}
```

Linked Lists – Method 9

9. Remove the front Node

A LinkedList instance always has direct access to the **`front`** Node

- What does it mean to remove the **`front`** Node from an empty List?

```
public void removeFront() {
    if(!isEmpty())




}
```

Linked Lists – Method 10

10. Add a new Node with given data (String) to the end of the List

How does a LinkedList instance find its last Node?

- What if the LinkedList is empty?

```
public void addToEnd(String d) {
    Node n = new Node(d, null);
    if(isEmpty())

    else {




    }
}
```

Linked Lists – Method 11

11. Remove the last Node

How does a LinkedList instance find its last Node?

- What if the LinkedList is empty?
- What if the LinkedList only has one Node?

```
public void removeLast() {
       if(!isEmpty()) {









       }
}
```

Linked Lists – Method 12

12. Get the index for a given String (or -1 if not found)

How should a LinkedList search to make sure every Node is considered?

```
public int contains(String d) {
        Node cur = front;
        boolean found = false;
        int index = -1;







        if(!found)
                index = -1;
        return index;
}
```

Linked Lists – Method 13

13. Add a new Node with given data (String) at a given index

What should be the range of valid index values?

```
public void add(int index, String d) {
       if(                                     ) {
              if(index == 0)

              else {




              }
       }
       //else { System.out.println("Index out of bounds!"); }
}
```

Linked Lists – Method 14

14. Remove the node at a given index

What should be the range of valid index values?

```
public void remove(int index) {
        if(                                             ) {
                if(index == 0)

                else if(index == size()-1)

                else {




                }
        }
        //else { System.out.println("Index out of bounds!"); }
}
```

Linked Lists – Method 15

15. Get the Node at a given index (reference)

What should be the range of valid index values?

```
public Node getNode(int index) {
      Node cur = null;
      if(                                         ) {
            if(index == 0)

            else {



            }
      } // else { System.out.println("Index out of bounds!"); }
      return cur;
}
```