PA1 Late / Results → June 3rd @ 1/gm

PA2 released

Exam 1
↳ Fri wd 3
↳ ~2 hours → online → Fri 8am / Sun 8am] 48 hours

What is a Linked List?

A Linked List is a data structure that implements a List ADT, where elements in the list may appear anywhere in memory, but are "linked" together in a particular order using references or pointers.

```
class Node {
  String value;
  Node next;
  public Node(String value, Node next) {
    this.value = value;
    this.next = next;
  }
}

// Somewhere else in the code.  still inside Node class (can access next)
Node n1 = new Node("banana", null);
Node n2 = new Node("apple", null);
n1.next = n2;
```


Draw the memory model diagram for this code.

Linked Lists are implemented with Node objects

The Node forms the structure of the list.

It contains:

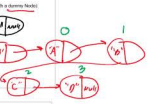
- A reference to the data stored at that position in the list
- A reference to the next node in the list
- Optionally (for a doubly linked list) a reference to the previous node in the list.

The Linked List itself usually contains only a reference to the first node in the list (head), and sometimes a reference to the last node (tail). It also might store the list's size

Assume that a linked list contains the following elements in this order: "A", "B", "C", "D". Draw the memory model of the linked list. Label the index of each element.

Without a sentinel node (root starts with null):

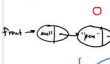
With a sentinel node (root starts with a dummy Node):




```
public class LinkedStringList implements StringList {
  Node front;
  int size;

  public LinkedStringList() {
    this.front = new Node(null, null);

  }

  /* Add an element at the beginning of the list */
  public void prepend(String s) {
    Node newFront = new Node(s, this.front.next);
    this.front.next = newFront;
    this.size += 1;
  }

  /* Add an element at the end of the list */
  public void add(String s) {
    Node curNode = front;
    while (curNode.next != null) {
      curNode = curNode.next;
    }
    curNode.next = new Node (s, null);
    size += 1;
  }

  /* Add an element at the specified index */
  public void insert(int index, String s) {
    Node curNode = front;
    int count = 0;
    while (count < index) {
      curNode = curNode.next;
      count += 1;
    }
    Node node = new Node(s, curNode.next);
    curNode.next = node;
    size += 1;
  }
  /* Remove the element at the specified index */
  public void remove(int index) {
    Node curNode = front;
    for (int i=0; i<index; i++) {
      curNode = curNode.next;
    }
    curNode.next = curNode.next.next;
    size -= 1;
  }
}
```

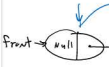
Draw the linked list for the following method calls:

prepend("banana"),
prepend("apple"),
prepend("pear"),



Draw the linked list for the following method calls:

add("banana"),
add("apple"),
add("pear"),



Draw the linked list for the following method calls:

insert(0, "A"),
insert(0, "B"),
insert(0, "C"),
insert(1, "D"),

Draw the linked list for the following method calls:

add("C"),
add("D"),
add("E"),
add("X"),

remove(1),
remove(2),

no dummy node
front → null

special case

```
add(E)
  if (front == null) {
    front = new Node( )
  }
  else {
    //...
  }
}
```

Invariant

Java Generics

```
public interface CollectionsIn<E> extends IterableE {
```

What does the <E> mean in the above code?

(a) That this collection can only be used with objects of a built-in Java type called E
(b) That an object reference that implements Collection can be instantiated to work with (almost) any object type
That a single collection can hold objects of different types

object/reference types
does not work w/ primitives
no int
use Integer

Java Generics: use parameterized types in class definitions

```
public class RecentRemembrancer<T> {
  private ArrayList<T> elementS;

  public RecentRemembrancer() {

    elementS = new ArrayList<T>();
  }
  public T add(T element) {
    elementS.add(element);
    return element;
  }
  public int getNumElements() {
    return elementS.size();
  }

  public T getLastElement() {
    return elementS.get(elementS.size()-1);
  }
}
```

What is the type parameter for the RecentRemembrancer class?

Complete the implementation of the RecentRemembrancer class.

Complete the following main method to create an instance of rr for integers and rr2 for strings.

```
public static void main(String[] args) {
  RecentRemembrancer<Integer> rr = new RecentRemembrancer<>();
  RecentRemembrancer<String> rr2 = new RecentRemembrancer<>();

  rr.add(1);
  rr.add(2);
  rr2.add("Three");
  System.out.println(rr.getNumElements() + "times added");
  System.out.println("Last elem was " + rr.getLastElement());
}
```

int

rr.add(i), int Value i);

What gets printed?

2 elems added
↳ last elem was 2

The type parameter can be used to stand for a type (to be specified later anywhere in this class (and its inner classes))

You are not allowed to use Generics as follows:

In creating an object of that type:

```
new T() // error
```

In creating an array with elements of that type:

```
new T[10] // error
```

As an argument to instanceof

```
someref instanceof T // error
```

Note: To ensure that certain methods can be called, we can constrain the generic type to be subclass of an interface or class

```
public class MyGenericA <E extends Comparable> {...}
```

E

compare To

Generics : https://docs.oracle.com/javase/tutorial/java/generics/erasure.html

Important for data structures in general
public class MyGenericB {
  //order that uses E
}

Pros of using generics
- Avoid type casting (i.e. limit runtime errors)

Before Java 5

```
ArrayList list = new ArrayList();// a list of objects
list.add("ping");
list.add(new Integer(32));

Integer data = (Integer) list.get(0);
```

Cons of using generics
- Type erasure

Type erasure during compile time:
- Compiler checks if generic type is used properly. Then replaces them with Object
- Runtime doesn't have different generic types

```
MyList<String> set1 = new MyList<String>();
MyList<Integer> set2 = new MyList<Integer>();
```

Compile time:

```
MyList<String> set1 = new MyList<String>();
```

Runtime:

```
MyList<Object> set1 = new MyList<Object>();
```

E[] arr = (E[]) new Object[5]

Convert Node and LinkedStringList to be a generic using List interface

```
public interface ListElement {
  /* Add an element at the end of the list */
  void addElement(E e);
  /* Get the element at the given index */
  Element get(int index);
  /* Get the number of elements in the list */
  int size();

}

class Node {
  String value;
  Node next;
  public Node(String value, Node next) {
    this.value = value;
    this.next = next;
  }
}

LL <E>front;
LinkElement

public class LinkedStringList implements ListElement {
  Node front;
  int size;

  public LinkedStringList() {
    this.front = new Node(null, null);
    this.size = 0;
  }
  public Element get(int index) {
    Node temp = this.front.next;
    for (int i = 0; i < index; i += 1) {
      temp = temp.next;
    }
    return temp.value;
  }

  public int size() {
    return this.size;
  }
  public void addElement(E e) {
    Node temp = this.front;
    while (temp.next != null) {
      temp = temp.next;
    }
    temp.next = new Node(e, null);
    this.size += 1;
  }
}
```

front
E
Node
Element
Element
Node<Element>

Exceptions

What happens if an invalid index is passed to get()?

Null pointer

Modify get() to throw an exception if the index is invalid

```
public String get(int index) {
  Node temp = this.front.next;
  for (int i = 0; i < index; i += 1) {
    temp = temp.next;
  }
  return temp.value;
}
```

if (index < 0 || index >= size) {
  throw new IndexOutOfBoundsException();
  new IllegalArgumentException();
}

JUnit - test that an exception is thrown

```
@Test(expected = IndexOutOfBoundsException.class)
```

Test fails if no IOOBE exception is thrown

Write a test to verify get() throws an exception with an invalid index