Data Structure Implementation:
LinkedList & Queue

CSE 017 | Prof Ziltz | Spring 23

This Week's Plan

- **Implementation of Lists**
  - ArrayList
  - **LinkedList**
- Implementation Stack
- **Implementation of Queue**
- Implementation PriorityQueue

Student Learning Outcomes

By the end of this week, you should be able to:

- Implement the List using an array
- Implement the List using a LinkedList (nodes)
- Implement the Stack using an arraylist
- Implement the Queue using a LinkedList
- Implement the Priority Queue using an arraylist
- Analyze the complexity of the operations on all the data structures

Why care about Data Structure Implementation?

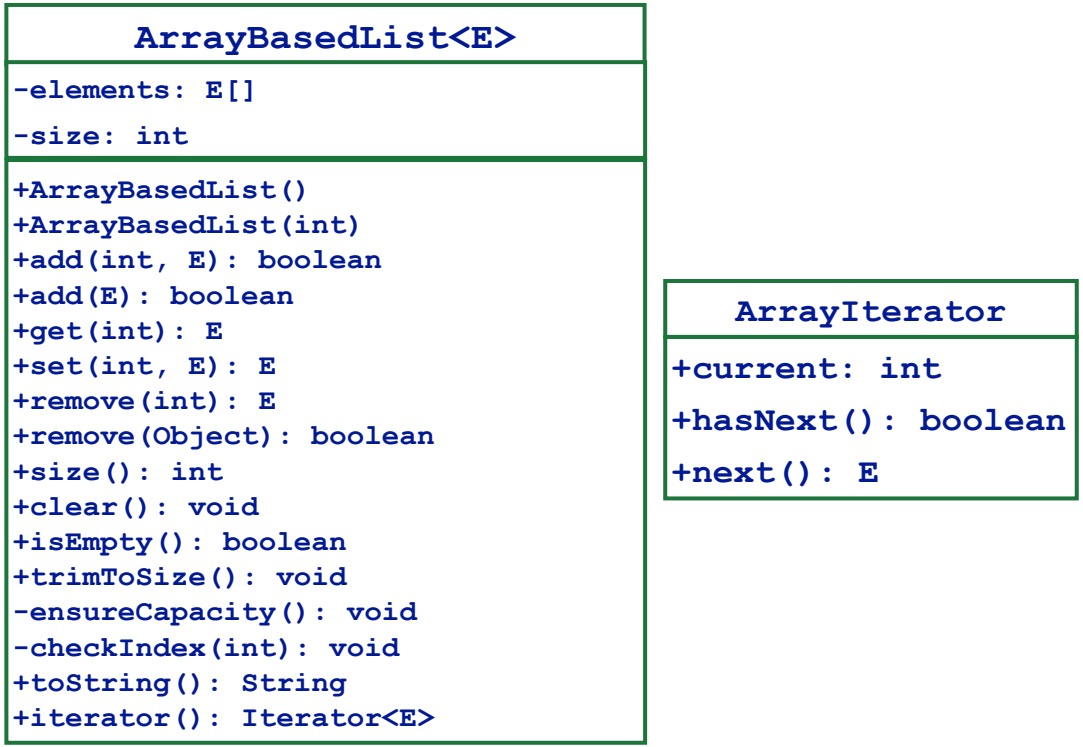
- **Available Data Structures in Java API**
  - List, Stack, Queue (using list), PriorityQueue
- **How are they implemented?**
- **How to create new data structures?**
- **Become a data structure designer rather than a data structure user**

Data Structure: List

- **Implemented in Java.util.Collection**
- **Store data in some order**
- **Common operations on List**
  - Retrieve an element from the list
  - Add a new element into the list
  - Remove an element from the list
  - Get the number of elements in the list
- **Implementation Options:**
  - Array Based List - `ArrayList<E>`
    - Fixed array size when the list is constructed
    - New larger array created when the current array is full
  - Linked List - `LinkedList<E>`
    - Size not fixed
    - Nodes are created when an item is added
    - Nodes are linked together to form the list

ArrayBasedList

Implementing our own Array-based List (instead of using that from Java Collection)



Analyzing the ArrayBasedList

What is the complexity of the operations in the ArrayBasedList?

Method	Complexity	Method	Complexity
`ArrayList()`	**O(1)**	`iterator()`	**O(1)**
`ArrayList(int)`	**O(1)**	`trimToSize`	**O(n)**
`size()`	**O(1)**	`ensureCapacity`	**O(n)**
`checkIndex()`	**O(1)**	`add(int, E)`	**O(n)**
`get(int)`	**O(1)**	`remove(int)`	**O(n)**
`set(int, E)`	**O(1)**	`toString()`	**O(n)**
`isEmpty()`	**O(1)**	`add(E)`	**O(1) - O(n)**
`clear()`	**O(1)**		

Analyzing the Stack

Stack is implemented using an array based list with access only at the end of the list (next available space)

What is the complexity of the operations in a Stack?

Method	Complexity
`Stack<>()`	O(1)
`peek()`	O(1)
`pop()`	O(1)
`push()`	O(1)/O(n)
`size()`	O(1)
`isEmpty()`	O(1)
`toString()`	O(n)

Priority Queue implementation (using ArrayList)

PriorityQueue<E>
-list: ArrayList<E> -comparator: Comparator<E>
+PriorityQueue() +PriorityQueue(Comparator<E>) +offer(E): void +poll(): E +peek(): E +size(): int +clear(): void +isEmpty(): boolean +toString(): String

Analyzing a PriorityQueue

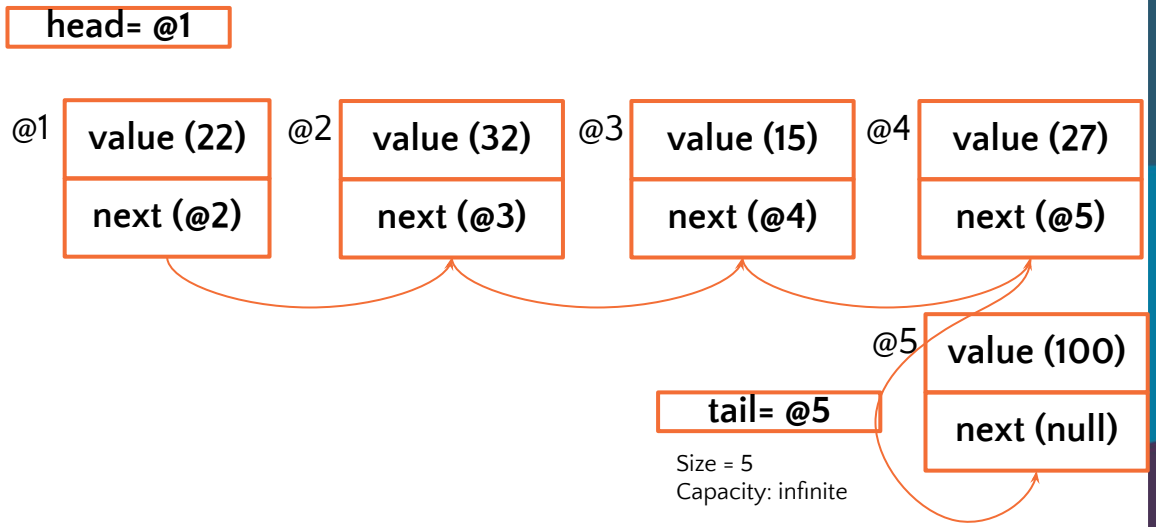
What is the complexity of the operations in a PriorityQueue?

Method	Complexity
`Queue<>()`	**O(1)**
`offer(E)`	**O(n)**
`poll()`	**O(n)**
`peek()`	**O(1)**
`size()`	**O(1)**
`clear()`	**O(1)**
`isEmpty()`	**O(1)**
`toString()`	**O(n)**

LinkedLists: Alt to ArrayList

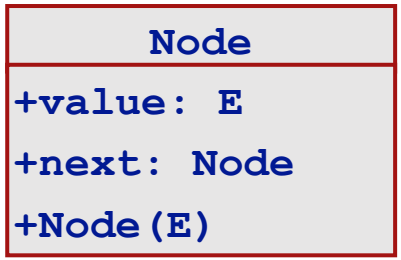
Node:

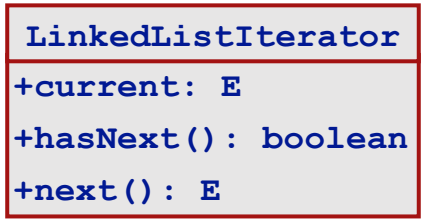
value	Value of the node (data stored @ that spot in the list)
next	Reference (pointer) to the next node

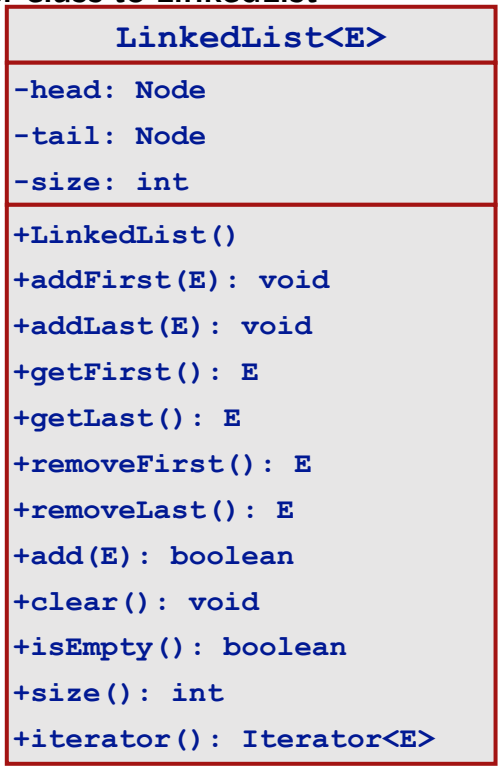


LinkedList: Node-Based List

- Implemented using linked nodes
- Class node is an inner class to LinkedList







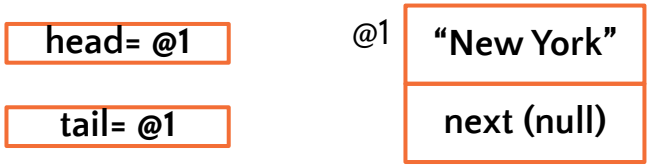
LinkedList

Creating the list -- no nodes yet

```
Node head = null;
Node tail = null; size =0;
```

Adding the first element to an empty list (no nodes)

```
// Adding the first element
head = new Node("New York");
tail = head; size++;
```



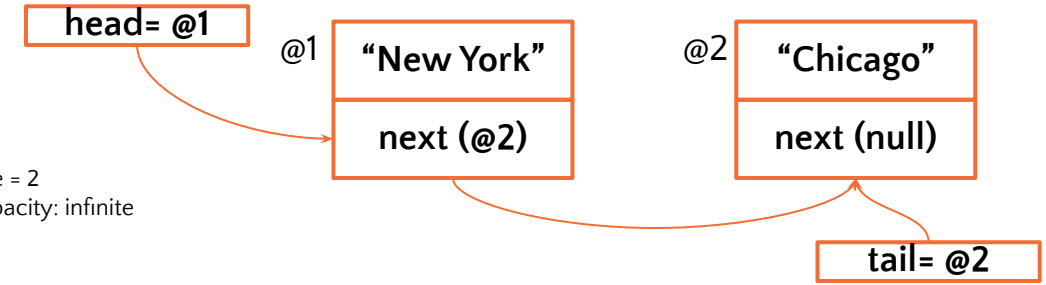
LinkedList

Adding an element to the end of the list (addLast())

```
tail.next = new Node("Chicago");
```

Changing tail to the new end of the list

```
tail = tail.next;
size++;
```



Size = 2
Capacity: infinite

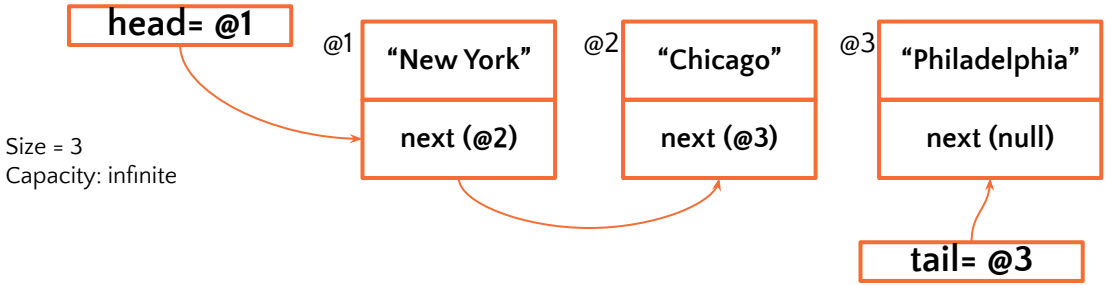
LinkedList

Adding an element to the end of the list (addLast())

```
tail.next = new Node("Philadelphia");
```

Changing tail to the new end of the list

```
tail = tail.next;
size++;
```

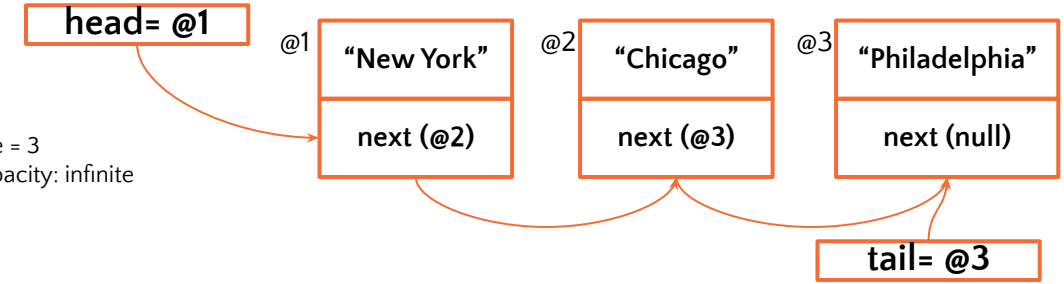


Size = 3
Capacity: infinite

LinkedList

Traversing the list

```
Node node = head;
while(node != null){
   System.out.println(node.value);
   node = node.next;
}
```



Size = 3
Capacity: infinite

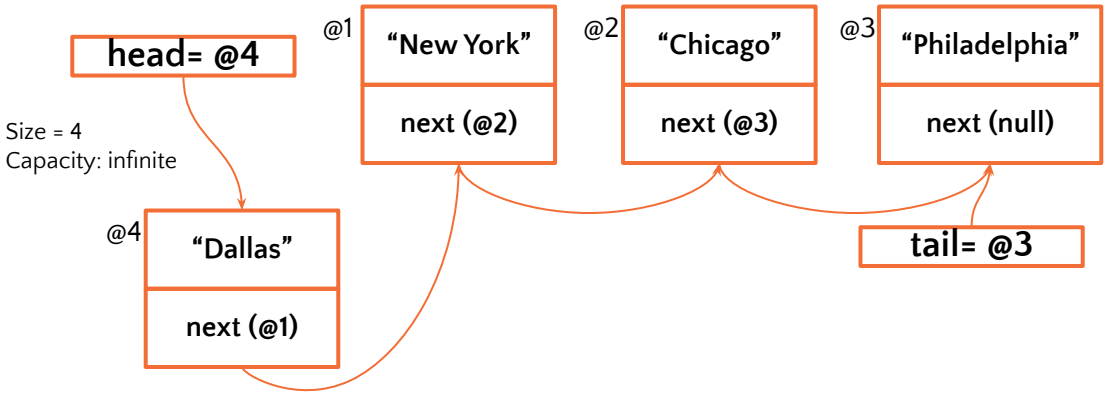
LinkedList

Adding an element to the beginning of the list (addFirst())

```
Node newNode = new Node("Dallas");
newNode.next = head;
```

Changing head to the new beginning of the list

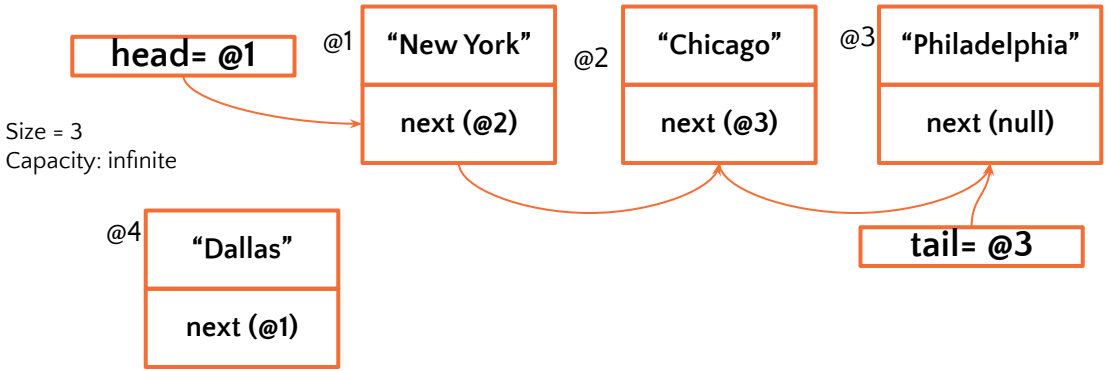
```
head = newNode;
size++;
```



LinkedList

Removing the element at the head -- removeFirst()

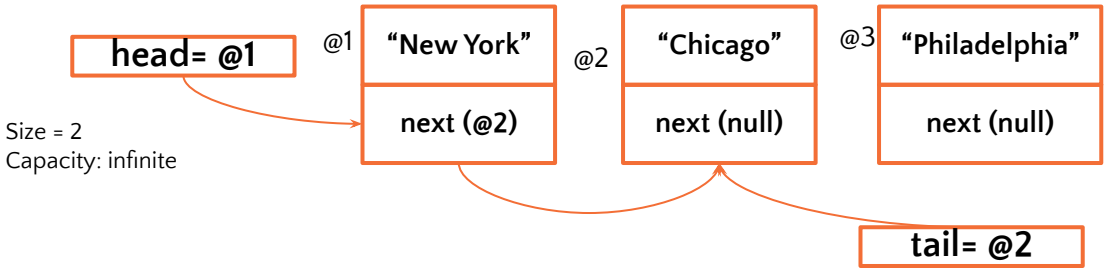
```
head= head.next;
size --;
```



LinkedList

Removing the element at the tail -- removeLast()

```
Previous of tail(@2).next= null;
      tail = node@2; size--;
```



Implementing LinkedList<E>

```
public class LinkedList<E> {
    // Data members
    private Node head, tail;
    int size;
    // Inner class Node
    private class Node {
        E value;
        Node next;

        Node(E initialValue) {
            value = initialValue;
            next = null;
        }
    }
    // Constructor
    public LinkedList() {
        head = tail = null;
        size = 0;
    }
```

```
    // Adding an item to the list
    public boolean addFirst(E item) {
        Node newNode = new Node(item);
        if (head == null) {
            head = tail = newNode;
        } else {
            newNode.next = head;
            head = newNode;
        }
        size++;
        return true;
    }
    public boolean addLast(E item) {
        Node newNode = new Node(item);
        if (head == null) {
            head = tail = newNode;
        } else {
            tail.next = newNode;
            tail = newNode;
        }
        size++;
        return true;
    }
    public boolean add(E item) {
        return addFirst(item); }
```

Implementing LinkedList<E>

```
// Retrieving an item from the list
public E getFirst() {
    if (head == null)
        throw new NoSuchElementException();
    return head.value;}
public E getLast() {
    if (head == null)
        throw new NoSuchElementException();
    return tail.value;
}
// Removing an item from the list
public boolean removeFirst() {
    if (head == null)
        throw new NoSuchElementException();
    head = head.next;
    if (head == null)
        tail = null;
    size--;
    return true;
}
```

```
public boolean removeLast() {
    if (head == null)
        throw new NoSuchElementException();
    if (size == 1)
        return removeFirst();
    Node current = head;
    Node previous = null;
    while (current.next != null) {
        previous = current;
        current = current.next;
    }
    previous.next = null;
    tail = previous;
    size--;
    return true;
}
```

Implementing LinkedList<E>

```
// toString() method
public String toString() {
    String output = "[";
    Node node = head;
    while (node != null) {
        output += node.value + " ";
        node = node.next;
    }
    output += "]";
    return output;
}
// clear, check if empty, and size
public void clear() {
    head = tail = null;
    size = 0;
}
public boolean isEmpty() {
    return (size == 0);
}
public int size() {
    return size;
}
```

```
// Generating an iterator for the list
   public Iterator<E> iterator() {
       return new LinkedListIterator();
   }
   private class LinkedListIterator implements Iterator<E> {
       private Node current = head;
       public boolean hasNext() {
           return (current != null);
       }
       public E next() {
           if (current == null)
               throw new NoSuchElementException();
           E value = current.value;
           current = current.next;
           return value;
       }
   }
```

Testing LinkedList<E>

```
import java.util.Iterator;
public class Test2 {
   public static void main(String[] args) {
       LinkedList<String> cityList;
       cityList = new LinkedList<>();
       cityList.addFirst("Boston");
       cityList.addFirst("Philadelphia");
       cityList.addFirst("San Francisco");
       cityList.addFirst("Washington");
       cityList.addFirst("Portland");
       System.out.println(cityList.toString());
       Iterator<String> LLIterator = cityList.iterator();
       System.out.print("LinkedList (iterator): ");
       while (LLIterator.hasNext()) {
           System.out.print(LLIterator.next() + " ");
       }
   }
}
```

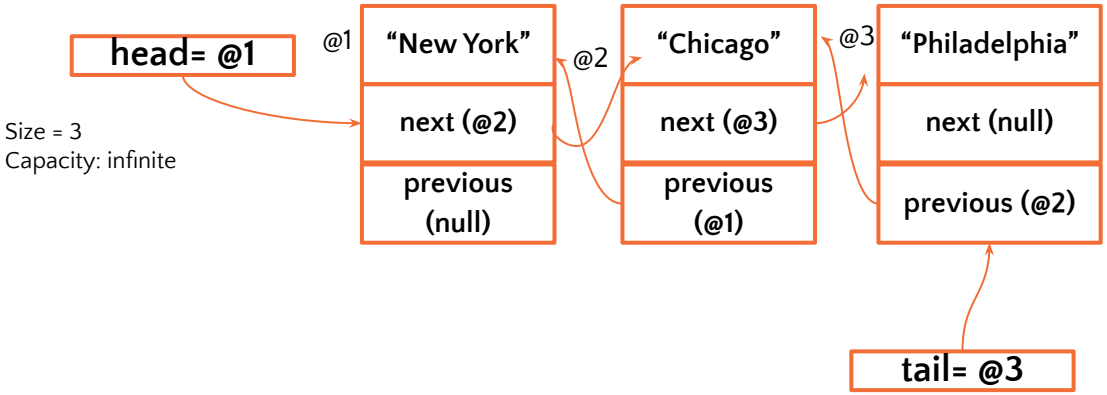
Analyzing the LinkedList

What is the complexity of the operations in the LinkedList?

Method	Complexity	Method	Complexity
`LinkedList()`	**O(1)**	`addFirst()`	**O(1)**
`size()`	**O(1)**	`addLast()`	**O(1)**
`clear()`	**O(1)**	`add(E)`	**O(1)**
`isEmpty()`	**O(1)**	`removeFirst()`	**O(1)**
`iterator()`	**O(1)**	`removeLast()`	**O(n)**
`getFirst()`	**O(1)**	`toString()`	**O(n)**
`getLast()`	**O(1)**		

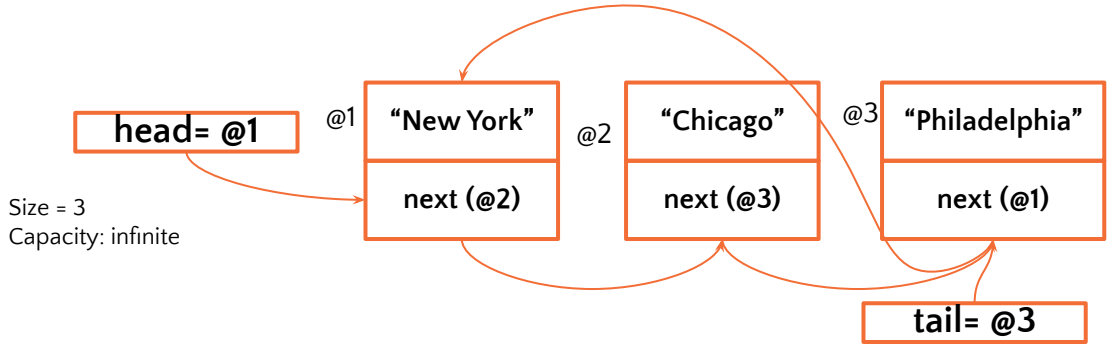
Variations of LinkedLists

- **Doubly Linked List**
  - Every node is linked to the next and the previous elements
  - Improves the performance of removeLast (from O(n) to O(1))



Variations of LinkedLists

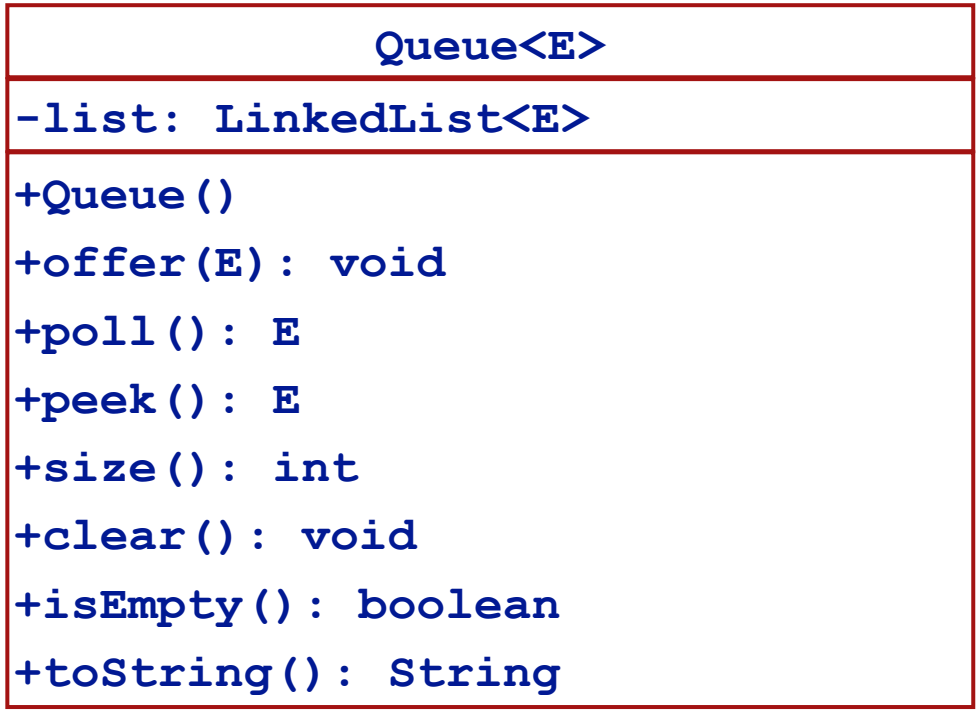
- **Circular Linked List**
  - Last element is linked back to the first element



Queue

- **Queue is implemented using a linked list with access at the head and the tail**
  - Remove from head
  - Add to tail

Queue implementation (using LinkedList)



Queue implementation (using LinkedList)

```
public class Queue<E> {
   private LinkedList<E> list;
   public Queue() {
       list = new LinkedList<>(); }
   public void offer(E item) {
       list.addLast(item);
   }
   public E poll() {
       E value = list.getFirst();
       list.removeFirst();
       return value;
   }
   public E peek() {
       return list.getFirst();
   }
   public String toString() { return "Queue: " + list.toString(); }
   public int size() { return list.size();}
   public void clear() { list.clear(); }
   public boolean isEmpty() { return list.size() == 0; }
}
```

Queue implementation (using LinkedList)

```
public class Test4 {
   public static void main(String[] args) {
       Queue<String> cityQueue = new Queue<>();
       cityQueue.offer("New York");
       cityQueue.offer("San Diego");
       cityQueue.offer("Atlanta");
       cityQueue.offer("Baltimore");
       cityQueue.offer("Pittsburg");
       System.out.println("City Queue (toString): " +
               cityQueue.toString());
       System.out.print("City Queue (poll): ");
       while (!cityQueue.isEmpty())
           System.out.print(cityQueue.poll() + " ");
   }
}
```

Analyzing a Queue

What is the complexity of the operations in a Queue?

Method	Complexity
`Queue<>()`	**O(1)**
`offer(E)`	**O(1)**
`poll()`	**O(1)**
`peek()`	**O(1)**
`size()`	**O(1)**
`clear()`	**O(1)**
`isEmpty()`	**O(1)**
`toString()`	**O(n)**

Quick Summary

- **Implementing Data Structures**
- **Lists: ordered set of data**
  - Array based list
  - linked List
- **Stack**
  - implemented using ArrayList
- **Queues**
  - Queue and PriorityQueue using LinkedList and ArrayList (respectively)
- **Complexity of data structure operations**