Lecture 5

LinkedList

Last time:

We looked at ArrayList:

1. It grows dynamically and there is a way to shrink its length.
2. But, the underlying data structure of ArrayList class is still an array. That means **"immutable length"** and **"no holes allowed!"** This causes an issue of **copying many elements** when inserting elements (while dynamically resizing the array when necessary). Also, when deleting elements, it is necessary to **shift many elements**.
3. The policy of increasing the length by 50% when the array is full makes us to think a little more in terms of memory usage and running time of add method (amortized constant time).
4. Once again, removing an element is expensive because it is necessary to shift elements down every time the method is called unless it is the last element.

Is there any way we can overcome these disadvantages?

1. Is there any way we can use as much memory as it needs and can expand to fill all of the available memory?
2. Is there any way we do not need to shift elements when there is deletion or insertion?

My situation:

1. I do not want to waste any memory that I am not really utilizing and cannot allow the worst-case latency that ArrayList has.
2. Whenever I need to access the elements, I need to **access them sequentially anyway**.

Step 1. A brief look at LinkedList in the Collections Framework

LinkedList is one of the two general-purpose List implementations. In addition to some general and common methods, LinkedList in Java also offers a few methods that work with the elements at the beginning and at the end of the list as follows.

```
addFirst(element: Object) : void
addLast(element: Object) : void
getFirst() : Object
getLast() : Object
removeFirst() : Object
removeLast() : Object
```

Step 2. Conceptual View

It is a linear data structure where each element has its data and a reference to the next node. Let's call it node.
The entry point is called the head of the list. (Head is not a separate node but simply the reference to the first node.)
The last node has a reference to null.

Advantages
As we can see, a linked list is a dynamic data structure.

Disadvantages
1. It does not (cannot) allow random access to individual elements.
2. It also requires each node to use extra memory to have a reference to the next node.

Types

1. Singly linked list
2. Doubly linked list
3. Circular linked list

Example questions

With the singly linked list above, write the output for the following cases. (The list is restored to its initial state before each line executes.)

a) ____________ head.next.next.next.data;
b) ____________ head.next.next.next.next.data;

With the doubly linked list above, write the output for the following cases. (The list is restored to its initial state before each line executes.)

a) _______________ head.next.next.next.data;
b) _______________ head.next.next.next.prev.prev.data;
c) _______________ tail.prev.prev.next.data;

Step 3. Implementation View

Node class

Java programming language allows you to define a class within another class, called a nested class.

```
class OuterClass {
      ....
      class NestedClass {
           ....
      }
}
```

There are two types of nested classes: static and non-static. Non-static nested classes (also called inner classes) have access to other members of the enclosing class, even if they are declared as private. However, *static nested classes do not have access to non-static members of the enclosing class*.

Why do we use nested classes?

- You can group classes that are only used in one place.
- It also increases encapsulation.

For the purpose of the LinkedList class that we will implement in this section, we will use the following static nested Node class.

```
____________ _____________ class Node<AnyType> {
    private AnyType data;  // data
    private Node<AnyType> next;  // reference to the next node

    // constructor with data and next node
    Node(AnyType d, Node<AnyType> n) {
        data = d;
        next = n;
    }
}
```

Skeleton: LinkedList class (Singly)

```
public class LinkedList<AnyType> {
    _______________ Node<AnyType> head;

    //  Constructs an empty list.
    public LinkedList() {
        head = null;
    }

    //  TODO implements all of the methods below.

}
```

addFirst

```
// Inserts a new item at the beginning of this list.
public void addFirst(______________________ item) {
     __________ = new Node<AnyType>(item, __________);
}
```

Traverse to the last node

```
Node<AnyType> tmp = head;
while (________________ != ___________) tmp = tmp.______________;
```

addLast

```
/**
 * Inserts a new item to the end of the list.
 */
public void addLast(__________________ item) {
    // if the list empty
    if (__________________ == _________________) {
        ________________________________;
        return;
    }
    // traverse to find the last element
    Node<AnyType> tmp = head;
    while (_____________________ != _____________________) {
        tmp = tmp._________;
    }

    // finally, add the new element into the list
    tmp.__________ = new Node<AnyType>(item, _______________);
}
```

insertAfter

```
/**
 * Finds a node containing “key” and insert a new item after it.
 */
public void insertAfter(AnyType key, AnyType item) {
    // find the location first with the given key
    Node<AnyType> tmp = head;
    while (________________ && _______________________________) {
        tmp = tmp.next;
    }

    // as long as the key is in the list
    if (tmp != _________________) {
        Node<AnyTpye> toBeInserted = new Node<AnyType>(item,
_________________________);
        tmp.next = toBeInserted;
    }
}
```

insertBefore (trickier)

```
/**
 * Finds a node containing “key” and insert a new item before it.
 * @param key a key to be found to add a new element
 * @param item a data to be added into the list
 */
public void insertBefore(AnyType key, AnyType item) {
    // if the list is empty
    if (________________ == _________________) {
        return;
    }
    // if head has the key
    if (__________________________________________________) {
        ____________________________________;
        return;
    }

    /*
     * key is not in the head
     * Needs to keep track of previous node of current node
     */
    Node<AnyType> prev = ____________;
    Node<AnyType> cur = head;
    while (______ != ______ && ______________.equals(________)) {
        prev = ______________;
        cur = _______________;
    }

    // Found it, then add new node into next of the previous
    if (__________________ != null) {
        ________.next = new Node<AnyType>(item, _______________);
    }
}
```

remove (trickier)

```
/**
 * Removes the first occurrence of a key from the list.
 * @param key a key to be deleted
 */
public void remove(AnyType key) {
    // if the list is empty
    if (_____________ == ______________) {
        return;
    }

    // if the key is found from the head element
    if (head.data.equals(key)) {
        head = head.____________;
        return;
    }

    Node<AnyType> prev = _____________;
    Node<AnyType> cur = head;

    while (_________ != _________ && !cur.data.equals(_______)) {
        prev = ______________________;
        cur = _______________________;
    }

    // as long as key is found
    if (___________ != null) {
        ______________________ = _______________________;
    }
}
```

Iterator

As we use ArrayList and/or LinkedList in Java, we can iterate through elements as follows.

```
import java.util.*;

public class ListIterationDemo {

    public static void main(String[] args) {
        // Create a new list
        List<Integer> numbers = new LinkedList<Integer>();

        numbers.add(1);
        numbers.add(2);
        numbers.add(3);
        numbers.add(4);
        numbers.add(5);

        // get an Iterator object for numbers using iterator()
        Iterator<Integer> itr = numbers.iterator();

        // iterate using hasNext() and next() methods of Iterator
        System.out.println("Iterating through elements");
        while (itr.hasNext()) {
            System.out.println(itr.next());
        }

    }
}
```

How is this possible?

Iterator is to provide an access to a group of data that is private. In Java, an iterator is an object. To iterate over the dataset, the data structure should implement `Iterable` interface and it also requires a class that implements the `Iterator` interface. It is typically done by having a private inner class (not-static nested class).

Modifications to the LinkedList class:

```
import java.util.*;

public class LinkedList<AnyType> implements Iterable<AnyType> {
    private Node<AnyType> head;

    /**
     * Constructs an empty list.
     */
    public LinkedList() {
        head = null;
    }

    …

    /**
     * Iterator implementation that returns iterator object.
     */
    @Override
    public Iterator<AnyType> iterator() {
        return new LinkedListIterator();
    }

    /**
     * Inner class for LinkedListIterator
     * that implements Iterator interface.
     */
    private class LinkedListIterator implements Iterator<AnyType>
    {
        private Node<AnyType> nextNode;

        LinkedListIterator() {
            nextNode = ________________________;
        }

        @Override
        public boolean hasNext() {
            return nextNode != ________________;
        }
        @Override
        public AnyType next() {
            // when there is no next element
            if (!_______________________) {
                throw new NoSuchElementException();
            }

            AnyType result = nextNode.data;
            // Set the nextNode (move one step forward)
            ____________________ = __________________________;
            return result;
        }

    }
}
```

Revisiting running time complexity:
addFirst:

insertBefore or insertAfter:

delete:

search:

Comparison with arrays (ArrayList):