Lecture 5

LinkedList

Last time:

We saw ArrayList:

1. It can grow dynamically and there is a way for to shrink its length.
2. But, the underline 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 (dynamically resizing the array when necessary) and an issue of **shifting many elements** when deleting elements from the underline array (each time).
3. 50% increasing policy makes us to think a little more in terms of memory usage.
4. Once again, it is expensive to remove an element because it is necessary to shift many 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 a deletion or insertion?

My situation:

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

Step 1. A brief look at LinkedList class in Java Collections

LinkedList is one of the two general-purpose List implementations. In addition to some general methods, LinkeList in Java offers a few methods that work with the elements at the beginning and 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 is a separate object.
Each element (let's call it a node) of a list has its data and a reference to the next 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 direct access to the individual elements.
2. It also requires extra 4 bytes (on a 32bit machine) to store a reference to the next node.

Theoretical comparison between ArrayList and LinkedList in terms of memory usage (Elements are all null and on x32 system).

Number of Elements	ArrayList	LinkedList
0	36	20
1	36	44
2	44	60
3	44	84
4	52	100
5	52	124
6	60	140
7	60	164
8	68	180
9	68	204
10	76	220

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 categories 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 private. However, *static nested classes do not have access to other 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 use in this section, we will use the following Node class as a static nested class.

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

     // constructor with data and next node
      public Node(AnyType data, Node<AnyType> next) {
            this.data = data;
            this.next = next;
      }
}
```

Skeleton: LinkedList class (Singly)

```
import java.util.*;

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

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

       //  TODO implements all of the methods below.

}
```

addFirst

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

Traverse

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

addLast

```
// Inserts a new node to the end of the list.
public void addLast(AnyType item) {
     // if the list empty
     if(__________ == ____________) _________________________;
     else {
           // 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

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

     // as long as the key is in the list
     if(tmp != _________) {
          Node<AnyTpye> toBeInserted = new
Node<AnyType>(toInsert, _________________);

          tmp.next = toBeInserted;
     }
}
```

insertBefore (trickier)

```
/**
 * Find a node containing “key” and insert a new node before it.
 * @param key a key to be found to add a new element
 * @param toInsert a data to be added into the list
 */
public void insertBefore(AnyType key, AnyType toInsert) {
       // if the list is empty
       if(__________ == __________ ) return;
       // if head has the key
       if(head.data.equals(________)) {
           _________________;
           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(______ != ______ && __________.data.equals(______)) {
           prev = _____________;
           cur = _________________;
       }

       // Found it, then add new node into next of the previous
       if(__________ != null)
           ______.next = new Node<AnyType>(toInsert, _________);
}
```

remove (trickier)

```
/**
 * remove 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(head == __________) {
             throw new RuntimeException(“cannot delete”);
       }

       // 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 = ______________________;
       }

       // if key is not found
       if(___________ == null) {
           throw new RuntimeException(“cannot delete”);
       }

       // finally remove the node
       ____________________ = _____________________;
}
```

Iterator

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

```
import java.util.*;

public class ListIterationDemo {

    public static void main(String[] args) {
        // Create an arrayList
        List<Integer> arrayList = new ArrayList<Integer>();

        // Add elements to arrayList
        arrayList.add(1);
        arrayList.add(2);
        arrayList.add(3);
        arrayList.add(4);
        arrayList.add(5);

        // get an Iterator object for arrayList using iterator()
        Iterator itr = arrayList.iterator();

        // iterate using hasNext() and next() methods of Iterator
        System.out.println("Iterating through arraylist
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.

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;
    }

    // implements all the other methods here.

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

    /**
     * inner class for LinkedListIterator
     * that implements Iterator inteface
     */
    private class LinkedListIterator implements Iterator<AnyType>
    {
        private Node<AnyType> nextNode;
        public LinkedListIterator() {
            nextNode = _________________;
        }

        // implementaion of hasNext()
        @Override
        public boolean hasNext() {
            return nextNode != _________________;
        }
```

```
        // implementation of next()
        @Override
        public AnyType next() {
            // when there is no next element
            if (!__________________) {
                throw new NoSuchElementException();
            }

            AnyType result = nextNode.data;
            // Set the nextNode
            ____________________ = _________________________;
            return result;
        }

        // do not want to removal happens
        @Override
        public void remove() {
            throw new UnsupportedOperationException();
        }
    }
}
```

Revisit the running time complexity:
addFirst:

insertBefore or insertAfter:

delete:

search:

Comparison with arrays: