Programming with Data Structures

CMPSCI 187

Spring 2016

- **Please find a seat**
  - **Try to sit close to the center (the room will be pretty full!)**
- **Turn off or silence your mobile phone**
- **Turn off your other internet-enabled devices**

Topics

- Implement List ADT using Links (References)
- The `find` Method
- `RefUnsortedList` and `RefSortedList`

Implementing List ADT using Links

- We've seen array-based and link-based implementations of StringLogs, stacks, and queues.

- We will now cover link-based implementation of the List ADT. To avoid confusions, we use the names **`RefUnsortedList`** and **`RefSortedList`**, which really mean **`LinkedUnsortedList`** and **`LinkedSortedList`.**

- Unlike the other classes, we now need to be able to add or remove elements from the *middle* of a linked structure, which is more complicated.

Implementing List ADT using Links

- We will not cover `RefIndexedList`, because accessing the i-th element in a linked structure (where i is an index), requires O(N) running cost, which is not very efficient.

- In contrast, accessing the i-th element in an array is just O(1).

- So if we need indexed list, it's better to use an array-based implementation.

Implementing List ADT using Links

- Similar to the array-based implementation, the link-based implementation also implements the `ListInterface`, and this is regardless of whether the listed is unsorted or sorted, since both use the same set of methods.
- And remember the assumptions for lists we gave in the last lecture:
  - duplicate elements are allowed.
  - no null elements.
  - methods return values to indicate success / failure (as opposed to throw exceptions).
  - equals and compareTo are consistent.

The `RefUnsortedList` Class

- As before we use `LLNode<T>` objects to store elements. The first (head) node is stored in variable `list`.

- An empty list has `list == null`.

- The code we present here, particularly for the `find` method, will be different from the textbook, in order to avoid passing return values through instance variables (which is in general a bad approach).

RefUnsortedList

```
public class RefUnsortedList<T>
             implements ListInterface<T>
{
    protected int numElements;
    protected LLNode<T> list; // head of list

    public RefUnsortedList( ) {
        numElements= 0;
        list = null;
    }
```

The `find` Method

- The find method traverses the linked structure to search for the element being requested. So how do we indicate the result of the search?

- If an element is found, we need a pointer to its node. So we will have the **`find`** method return a **`LLNode<T>`** object.

- If the element is not found, we return a **`null`**. Therefore a non-**`null`** return value means element is found, and a **`null`** return value means the element is not found.

Version 1 of `find`

```
protected LLNode<T> find (T target) {
  LLNode<T> curr = list;
  while (curr != null) {
    if (curr.getInfo().equals(target)) {
      break;
    } else {
      curr = curr.getLink();
    }
  }
  return curr;
}
```

Version 2 of `find`

```
protected LLNode<T> find (T target) {
   LLNode<T> curr = list;
   while (curr != null &&
          !curr.getInfo().equals(target)) {
      curr = curr.getLink();
   }
   return curr;
}
```

Clicker Question #1

```
while (!curr.getInfo().equals(target) &&
        curr != null) {
  curr = curr.getLink();
}
```

In the above code, we have switched the order of the two statements around && in while loop condition. What would happen? (**Hint**: consider how && is evaluated)

(a) nothing would change

(b) in some cases it throws `NullPointerException`

(c) `curr` will always be `null` after the loop ends.

(d) `curr` will always be non-`null` after the loop ends.

Answer on next slide

Clicker Question #1

```
while (!curr.getInfo().equals(target) &&
        curr != null) {
  curr = curr.getLink();
}
```

In the above code, we have switched the order of the two statements around && in while loop condition. What would happen? (**Hint**: consider how && is evaluated)

(a) nothing would change

(b) in some cases it throws `NullPointerException`

(c) `curr` will always be `null` after the loop ends.

(d) `curr` will always be non-`null` after the loop ends.

Revisiting the `find` Method

- Shortly we will see how to remove a node from the list. This time we need to be able to remove any node (including head and last node) from the list.

- To remove a node (say **`curr`**), we need to re-wire the link by setting the previous node's link point to **`curr`**'s next node.

- In order to make use of the **`find`** method, it not only needs to return the current (i.e. found) node, but also the previous node (i.e. curr's predecessor). How to do so?

Revisiting the `find` Method

- How can we have a Java method return two or more values?

- One common approach is to create a class that contains any number of return values you need, and have the method return an object of that type.

- If the return values are all of the same type, you can also return an array that contains all the return values:

```
public int[] foo() {
   .. .. ..
   return new int[] {value1, value2, value3};
}
```

```
private class findVal<T> {
  public LLNode<T> curr, prev;
  public findVal(LLNode<T> c, LLNode<T> p) {
    curr = c; prev = p;
  }
}
protected findVal<T> find (T target) {
  LLNode<T> curr = list, prev = null;
  while (curr != null) {
    if (curr.getInfo().equals(target) {
      break;
    } else {
      prev = curr;
      curr = curr.getLink();
    }
  }
  return new findVal(curr, prev);
}
```

Clicker Question #2

```
protected findVal<T> find (T target) {
  LLNode<T> curr = list, prev = null;
  while (curr != null) {
    if (curr.getInfo().equals(target) {
      break;
    } else {
      prev = curr;
      curr = curr.getLink();
  }
    return new findVal(curr, prev);
  }
```

Suppose the first (head) node contains the element we are searching for. What's the value of **`prev`** returned by the **`find`** method?

`(a) list` **`(b) curr`**

`(c) null` **`(d) curr.getLink();`**

Answer on next slide

Clicker Question #2

```
protected findVal<T> find (T target) {
  LLNode<T> curr = list, prev = null;
  while (curr != null) {
    if (curr.getInfo().equals(target) {
      break;
    } else {
      prev = curr;
      curr = curr.getLink();
    }
  }
```

Suppose the first (head) node contains the element we are searching for. What's the value of **`prev`** returned by the **`find`** method?

(a) `list`

(b) `curr`

(c) `null`

(d) `curr.getLink();`

The `remove` Method

```
public boolean remove (T element) {
 boolean found = false;
  findVal<T> fval = find(element);
  if (fval.curr != null) {
    found = true;
    if (fval.curr == list) // node to remove is head
      list = list.getLink();
    else
      fval.prev.setLink(fval.curr.getLink());
    numElements--;
  }
  return found;
}
```

Clicker Question #3

```
LLNode<T> next = curr.getLink(); // assume non-null
curr.setInfo(next.getInfo());
curr.setLink(next.getLink());
```

Assume **`curr`** points to a node in the middle of a linked list (i.e. not the head nor the last node). What's the effect of the above code?

(a) it removes the current node from the list

(b) it removes the node following curr from the list

(c) it removes the node prior to curr from the list

(d) it has no effect on the linked list

Answer on next slide

Clicker Question #3

```
LLNode<T> next = curr.getLink(); // assume non-null
curr.setInfo(next.getInfo());
curr.setLink(next.getLink());
```

Assume **`curr`** points to a node in the middle of a linked list (i.e. not the head nor the last node). What's the effect of the above code?

(a) it removes the current node from the list

(b) it removes the node following curr from the list

(c) it removes the node prior to curr from the list

(d) it has no effect on the linked list

The `RefSortedList` Class

- If we have our sorted list class extend the unsorted list class, what do we need to change?

- Actually not very much. The **`contains`**, **`remove`**, and **`get`** methods all depend on the **`find`** method for their behavior. What they do after the element is found does not change at all.

The `RefSortedList` Class

- And for that matter, the `find` method of `RefUnsortedList` works perfectly well on sorted lists.

- The one thing we *could* change is that in an unsuccessful search, we could give up once we find an element in the list greater than the target, since we couldn't have the target come after that point.

- This is no savings at all in the worst case (the method is still O(n)) but will save us some time in some cases.

The `RefSortedList` Class

```
public class RefSortedList <T extends Comparable<T>>
                            extends RefUnsortedList<T>
                            implements ListInterface<T>
{
    public RefSortedList() {
        super();
    }
```

- The **`add`** method, however, definitely needs to be overridden, and we'll do that in a bit.

- First, though, we should notice that the header to this class has an unusual feature.

A Variation on Generics

```
public class RefSortedList <T extends Comparable<T>>
                           extends RefUnsortedList<T>
                           implements ListInterface<T>
```

- **`RefSortedList`** is a generic class, but we can only define **`RefSortedList<T>`** in the cases where **`T`** implements the interface **`Comparable<T>`**, meaning that **`T`** must be a class that has a **`compareTo`** method that takes a parameter of type **`T`**.

A Variation on Generics

```
public class RefSortedList <T extends Comparable<T>>
                           extends RefUnsortedList<T>
                           implements ListInterface<T>
```

- We can add an **`extends`** clause to the type variable in the declaration of a generic class, which makes it only valid when the promise of that clause is fulfilled.

- We say “extends” rather than “implements” because **`T`** could be anywhere below **`Comparable<T>`** in the inheritance hierarchy, not just directly under it.

Methods of `RefSortedList`

- To add to a sorted list, we skip **past all elements that are less than the new element**, until either we reach the end or we find the right element to put the new element in front of.
- Unfortunately we can't make use of the **`find`** method because we are not searching for a specific element, instead, we are looking for the first element that's no longer smaller than the new element to be added.
- So we will basically replicate the **`find`** method, with the necessary changes described above.

```
public void add (T element) {
    LLNode<T> curr = list, prev = null;
    while (curr != null) {
        T curr_elem = curr.getInfo();
        if (curr_elem.compareTo(element) < 0) {
            prev = curr;
            curr = curr.getLink();
        } else break;
    }
    LLNode<T> newNode = new LLNode<T>(element);
    if (prev == null) {  // adding before head
        newNode.setLink(list);  list = newNode;
    } else {
        newNode.setLink(curr);
        prev.setLink(newNode);
    }
    numElements++;
}
```