# ITI 1121. Introduction to Computing II *

Marcel Turcotte
School of Information Technology and Engineering

Version of March 26, 2011

## Abstract

- Recursive list processing (part II)

*These lecture notes are meant to be looked at on a computer screen. Do not print them unless it is necessary.

# “Head + tail”

```
if ( ... ) { // base case

   calculate results

} else { // general case

  // pre-processing

  s = method( p.next ); // recursion

  // post-processing

}
```

# E get( int index )

Let's build the recursive method, this requires augmenting the signature of the method **E get( int index )** with an additional parameter representing the current element (**p**).

# E get( int index )

Let's build the recursive method, this requires augmenting the signature of the method **E get( int index )** with an additional parameter representing the current element (**p**).

```
private E get( Node<E> p, int index );
```

# E get( int index )

Let's build the recursive method, this requires augmenting the signature of the method **E get( int index )** with an additional parameter representing the current element (**p**).

```
private E get( Node<E> p, int index );
```

If **index** represents the position of the element with respect to the list starting at **p**, what is the position of the element in the list starting at **p.next**?

# E get( int index )

Let's build the recursive method, this requires augmenting the signature of the method **E get( int index )** with an additional parameter representing the current element (**p**).

```
private E get( Node<E> p, int index );
```

If **index** represents the position of the element with respect to the list starting at **p**, what is the position of the element in the list starting at **p.next**? Yes, it's **index-1**!

# E get( int index )

Let's build the recursive method, this requires augmenting the signature of the method **E get( int index )** with an additional parameter representing the current element (**p**).

```
private E get( Node<E> p, int index );
```

If **index** represents the position of the element with respect to the list starting at **p**, what is the position of the element in the list starting at **p.next**? Yes, it's **index-1**!

What is the base case?

# E get( int index )

Let's build the recursive method, this requires augmenting the signature of the method **E get( int index )** with an additional parameter representing the current element (**p**).

```
private E get( Node<E> p, int index );
```

If **index** represents the position of the element with respect to the list starting at **p**, what is the position of the element in the list starting at **p.next**? Yes, it's **index-1**!

What is the base case?

If the **index** is zero then simply returns **p.value**.

## private E get( Node<E> p, int index )

```
private E get( Node<E> p, int index  ) {

    if ( index == 0 ) {

       return p.value;

    }

    return get( p.next, index-1 );
}
```

## private E get( Node<E> p, int index )

```
private E get( Node<E> p, int index  ) {

    if ( index == 0 ) {

       return p.value;

    }

    return get( p.next, index-1 );
}
```

What would occur if the initial value of **index** was larger than the total number of elements in list?

# private E get( Node<E> p, int index )

```
private E get( Node<E> p, int index  ) {

    if ( index == 0 ) {

       return p.value;

    }

    return get( p.next, index-1 );
}
```

What would occur if the initial value of **index** was larger than the total number of elements in list? **index** $>$ **0** and **p == null**.

# private E get( Node<E> p, int index )

```
private E get( Node<E> p, int index ) {

    if ( p == null ) {
       throw new IndexOutOfBoundsException();
    }

    if ( index == 0 ) {
       return p.value;
    }

    return get( p.next, index - 1 );
}
```

# public E get( int index )

```
public E get( int index ) {
    if ( index < 0 ) {
       throw new IndexOutOfBoundsException();
    }
    return get( first, index );
}
private E get( Node<E> p, int index ) {
    if ( p == null ) {
       throw new IndexOutOfBoundsException();
    }
    if ( index == 0 ) {
       return p.value;
    }
    return get( p.next, index-1 );
}
```

# int indexOf( E obj )

The methods **indexOf** returns the position of the first (left most) occurrence of **obj** in the list, the first element of the list is the position 0, the method returns -1 if the element is not found in the list.

# int indexOf( E obj )

The methods **indexOf** returns the position of the first (left most) occurrence of **obj** in the list, the first element of the list is the position 0, the method returns -1 if the element is not found in the list.

Following the "head + tail" strategy, the general case will involve a recursive call for the tail of the list:

```
int s = indexOf( p.next, o );
```

# int indexOf( E obj )

The methods **indexOf** returns the position of the first (left most) occurrence of **obj** in the list, the first element of the list is the position 0, the method returns -1 if the element is not found in the list.

Following the "head + tail" strategy, the general case will involve a recursive call for the tail of the list:

```
int s = indexOf( p.next, o );
```

What does **s** represent?

# int indexOf( E obj )

The methods **indexOf** returns the position of the first (left most) occurrence of **obj** in the list, the first element of the list is the position 0, the method returns -1 if the element is not found in the list.

Following the "head + tail" strategy, the general case will involve a recursive call for the tail of the list:

```
int s = indexOf( p.next, o );
```

What does **s** represent?

It's the position of **o** in the list starting at **p.next**.

# int indexOf( E obj )

The methods **indexOf** returns the position of the first (left most) occurrence of **obj** in the list, the first element of the list is the position 0, the method returns -1 if the element is not found in the list.

Following the "head + tail" strategy, the general case will involve a recursive call for the tail of the list:

```
int s = indexOf( p.next, o );
```

What does **s** represent?

It's the position of **o** in the list starting at **p.next**.

What is the position of **o** with respect to the list starting at **p**?

# int indexOf( E obj )

The methods **indexOf** returns the position of the first (left most) occurrence of **obj** in the list, the first element of the list is the position 0, the method returns -1 if the element is not found in the list.

Following the "head + tail" strategy, the general case will involve a recursive call for the tail of the list:

```
int s = indexOf( p.next, o );
```

What does **s** represent?

It's the position of **o** in the list starting at **p.next**.

What is the position of **o** with respect to the list starting at **p**? `1 + s`.

**int indexOf( Node<E> p, E obj )**

## int indexOf( Node<E> p, E obj )

Combining **s** (the result of the recursive call) and the current node.

# int indexOf( Node<E> p, E obj )

Combining **s** (the result of the recursive call) and the current node.

```
if ( p.value.equals( o ) ) {
    result = 0;
} else if ( result == -1 ) {
    result = s;
} else {
    result = 1 + s;
}
```

# int indexOf( Node<E> p, E obj )

What is the base case?

# int indexOf( Node<E> p, E obj )

What is the base case?

The smallest list is the empty list, it does not contain the value we're looking for, we should return the special value -1, to indicate that a match could not be found.

# int indexOf( Node<E> p, E obj )

What is the base case?

The smallest list is the empty list, it does not contain the value we're looking for, we should return the special value -1, to indicate that a match could not be found.

```
if ( p == null ) {
    return -1;
}
```

# int indexOf( Node<E> p, E obj )

```
private int indexOf( Node<E> p, E o ) {

    if ( p == null ) {
        return -1;
    }

    int result = indexOf( p.next, o);

    if ( p.value.equals( o ) ) {
        return 0;
    }

    if ( result == -1 ) {
        return result;
    }
    return result + 1;
}
```

## int indexOf( Node<E> p, E obj )

Is this working?

## int indexOf( Node<E> p, E obj )

Is this working? Yes, but it's inefficient.

## int indexOf( Node<E> p, E obj )

Is this working? Yes, but it's inefficient. Why?

## int indexOf( Node<E> p, E obj )

The recursion should stop as soon as the first occurrence has been found.

# int indexOf( Node<E> p, E obj )

The recursion should stop as soon as the first occurrence has been found. Okay, but how?

# int indexOf( Node<E> p, E obj )

The recursion should stop as soon as the first occurrence has been found. Okay, but how?

```
private int indexOf( Node<E> p, E o ) {

    if ( p == null ) { // Base case
        return -1;
    }
    if ( p.value.equals( o ) ) { // Base case
        return 0;
    }
    // General case
    int result = indexOf( p.next, o );
    if ( result == -1 ) {
        return result;
    }
    return result + 1;
}
```

**p.value.equals( o )** is a base case!

# int indexOf( Node<E> p, E obj )

Observe how "declarative" the method is.

```
private int indexOf( Node<E> p, E o ) {

    if ( p == null ) { // Base case
        return -1;
    }
    if ( p.value.equals( o ) ) { //Base case
        return 0;
    }
    // General case
    int result = indexOf( p.next, o );
    if ( result == -1 ) {
        return result;
    }
    return result + 1;
}
```

## int indexOfLast( E obj, Node<E> p )

The methods **indexOfLast** returns the position of the last (right most) occurrence of **obj** in the list, the first element of the list is the position 0, the method returns -1 if the element is not found in the list.

# int indexOfLast( E obj, Node<E> p )

The methods **indexOfLast** returns the position of the last (right most) occurrence of **obj** in the list, the first element of the list is the position 0, the method returns -1 if the element is not found in the list.

What are the necessary changes?

# int indexOfLast( E obj, Node<E> p )

The methods **indexOfLast** returns the position of the last (right most) occurrence of **obj** in the list, the first element of the list is the position 0, the method returns -1 if the element is not found in the list.

What are the necessary changes?

First, can **p.value.equals( o )** be a base case?

# int indexOfLast( E obj, Node<E> p )

The methods **indexOfLast** returns the position of the last (right most) occurrence of **obj** in the list, the first element of the list is the position 0, the method returns -1 if the element is not found in the list.

What are the necessary changes?

First, can **p.value.equals( o )** be a base case? No, the recursion must be allowed to go all the way to the end.

Second, how to combine the result of **indexOfLast(obj, p.next)** and the current node?

# int indexOfLast( Node<E> p, E obj )

```
public int indexOfLast( E obj ) {
    return indexOfLast( first, obj );
}
private int indexOfLast( Node<E> p, E obj ) {
    if (p == null) {
        return -1;
    }
    int result = indexOfLast( p.next, obj );
    if ( result > -1 ) {
        return result + 1;
    } else if ( obj.equals( p.value ) ) {
        return 0;
    } else {
        return -1;
    }
}
```

# boolean contains( E o )

The method contains returns **true** if the list contains the element **o**, i.e. there is a node such that **value.equals( o )**.

The auxiliary will initiates the search from the first node.

```
public boolean contains( E o ) {
    return contains( o, first );
}
```

# boolean contains( Node<E> p, E o )

The signature of the recursive methods will be:

```
private boolean contains( Node<E> p, E o );
```

Let's apply the head and tail strategy.

The empty list has to be part of the base case, if list is empty it cannot contain the object, contains should return **false**:

```
if ( p == null ) {
    return false;
}
```

The strategy suggests calling **contains** for the tail:

```
boolean result = contains( o, p.next );
```

# boolean contains( E o )

**Contains** is similar to **indexOf**, the method should stop as soon as the first occurrence has been found.

```
private boolean contains( Node<E> p, E o ) {
    if ( p == null ) {
        return false;
    }
    if ( p.value.equals( o ) ) {
        return true;
    }
    return contains( o, p.next );
}
```

# More complex patterns

The methods considered this far used one element at a time but this does not need to be.

# More complex patterns

The methods considered this far used one element at a time but this does not need to be.

Let's consider the method **isIncreasing**. It returns **true** if each element of the list is **equal to** or **greater than** its predecessor, and **false** otherwise.

# More complex patterns

The methods considered this far used one element at a time but this does not need to be.

Let's consider the method **isIncreasing**. It returns **true** if each element of the list is **equal to** or **greater than** its predecessor, and **false** otherwise.

To solve this problem, let's scan the list and return **false** as soon a consecutive pair of elements has been found such that the predecessor is greater than its successor, if the end is reached this means the list is increasing.

# boolean isIncreasing()

```
public boolean isIncreasing() {
    return isIncreasing( first );
}
```

# boolean isIncreasing( Node<E> p )

**Base case(s)**: shortest valid list(s)?

# boolean isIncreasing( Node<E> p )

**Base case(s)**: shortest valid list(s)? The empty list and the singleton list are valid and increasing.

# boolean isIncreasing( Node<E> p )

**Base case(s)**: shortest valid list(s)? The empty list and the singleton list are valid and increasing.

```
if ( ( p == null ) || ( p.next == null ) ) {
    return true;
}
```

# boolean isIncreasing( Node<E> p )

**General case**: 1) should the strategy be to make a recursive call the combine this result or 2) consider the current element(s) then make a recursive call (if necessary)?

# boolean isIncreasing( Node<E> p )

**General case**: 1) should the strategy be to make a recursive call the combine this result or 2) consider the current element(s) then make a recursive call (if necessary)?

```
if ( p.value.compareTo( p.next.value ) > 0) {
    return false;
} else {
    return isIncreasing( p.next );
}
```

# boolean isIncreasing( Node<E> p )

```
private boolean isIncreasing( Node<E> p ) {

    if ( ( p == null ) || ( p.next == null ) ) {
        return true;
    }

    if ( p.value.compareTo( p.next.value ) > 0 ) {
        return false;
    }

    return isIncreasing( p.next );
}
```

# Pitfall!

```
private boolean isIncreasing( Node<E> p ) {

    if ( ( p == null ) || ( p.next == null ) ) {
        return true;
    }

    if ( p.value.compareTo( p.next.value ) > 0 ) {
        return false;
    }

    return isIncreasing( p.next.next );
}
```

# Exercises

For a singly linked list implement the following methods recursively.

**void addLast( E o )**; adds an element at the last position of a list.

**boolean eq( OrderedList<E> other )**; compares all the elements of **this** list to the elements of the **other** list; the lists are not necessarily of the same length!

# Modifying the structure of the list

We now consider methods that are modifying the structure of the list.

# Modifying the structure of the list

We now consider methods that are modifying the structure of the list.

For methods such as **indexOf** and **contains**, the consequence of unnecessary recursive calls was inefficiency.

# Modifying the structure of the list

We now consider methods that are modifying the structure of the list.

For methods such as **indexOf** and **contains**, the consequence of unnecessary recursive calls was inefficiency.

However, when the methods are allowed to change the structure of the list, such as **remove** below, the consequences of unnecessary recursive calls are severe.

# public void remove( E o )

**void remove( E o )**; removes the first (left most) occurrence of an object.

# public void remove( E o )

**void remove( E o )**; removes the first (left most) occurrence of an object.

Outline a general strategy.

# public void remove( E o )

**void remove( E o )**; removes the first (left most) occurrence of an object.

Outline a general strategy.

- Traversing the list;
- Once the element has been found, remove it.

# public void remove( E o )

**void remove( E o )**; removes the first (left most) occurrence of an object.

Outline a general strategy.

- Traversing the list;
- Once the element has been found, remove it.

Difficulty?

# public void remove( E o )

**void remove( E o )**; removes the first (left most) occurrence of an object.

Outline a general strategy.

- Traversing the list;
- Once the element has been found, remove it.

Difficulty?

You remember that for a singly linked list, we shouldn't be stopping on the node to be removed since the variable **next** of the previous node needs to be changed **and** singly nodes of a singly linked list do not have a **previous** reference.

# public void remove( E o )

**void remove( E o )**; removes the first (left most) occurrence of an object.

Outline a general strategy.

- Traversing the list;
- Once the element has been found, remove it.

Difficulty?

You remember that for a singly linked list, we shouldn't be stopping on the node to be removed since the variable **next** of the previous node needs to be changed **and** singly nodes of a singly linked list do not have a **previous** reference.

How about removing the first node?

# public void remove( E o )

**void remove( E o )**; removes the first (left most) occurrence of an object.

Outline a general strategy.

- Traversing the list;
- Once the element has been found, remove it.

Difficulty?

You remember that for a singly linked list, we shouldn't be stopping on the node to be removed since the variable **next** of the previous node needs to be changed **and** singly nodes of a singly linked list do not have a **previous** reference.

How about removing the first node? Yes, this is a special case, the variable **first** has to be changed.

# public void remove( E o )

First, consider the **public non-recursive** method.

# public void remove( E o )

First, consider the **public non-recursive** method.

What are the pre-conditions?

# public void remove( E o )

First, consider the **public non-recursive** method.

What are the pre-conditions? The list should not be empty.

# public void remove( E o )

First, consider the **public non-recursive** method.

What are the pre-conditions? The list should not be empty.

When changing the structure of the list, the **public non-recursive** method often has a **special** case.

# public void remove( E o )

First, consider the **public non-recursive** method.

What are the pre-conditions? The list should not be empty.

When changing the structure of the list, the **public non-recursive** method often has a **special** case. What is it?

# public void remove( E o )

First, consider the **public non-recursive** method.

What are the pre-conditions? The list should not be empty.

When changing the structure of the list, the **public non-recursive** method often has a **special** case. What is it?

```
public void remove( E o ) {
    if ( first == null ) {
        throw new NoSuchElementException();
    }
    if ( first.value.equals( o ) ) {
        first = first.next;
    } else {
        remove( first, o );
    }
}
```

Exercise: "scrubbing the memory".

## **private void remove( Node<E> p, E o )**

Remark: for the first call to **remove( Node<E> p, E o )**, we know that **p.value.equals( o )** is **false**.

## **private void remove( Node<E> p, E o )**

Remark: for the first call to **remove( Node<E> p, E o )**, we know that **p.value.equals( o )** is **false**. Indeed, **p == first** and the case **first.value.equals( o )** has been processed by the **public non-recursive** method **remove**.

## private void remove( Node<E> p, E o )

Remark: for the first call to **remove( Node<E> p, E o )**, we know that **p.value.equals( o )** is **false**. Indeed, **p == first** and the case **first.value.equals( o )** has been processed by the **public non-recursive** method **remove**.

We'll keep that property here too, the method **remove( Node<E> p, E o )** will look for an occurrence of **o** at the position **p.next**, if the object is found then remove it, otherwise keep going.

The recursive method **remove** knows that the current element has been checked.

## private void remove( Node<E> p, E o )

What is the base case?

## **private void remove( Node<E> p, E o )**

What is the base case? Singleton (**p.next == null**).

## **private void remove( Node<E> p, E o )**

What is the base case? Singleton (**p.next == null**). What should be done?

## **private void remove( Node<E> p, E o )**

What is the base case? Singleton (**p.next == null**). What should be done? Nothing.

General case: 1) make a recursive call then post-processing or 2) pre-processing then recursive call (only if necessary).

## **private void remove( Node<E> p, E o )**

What is the base case? Singleton (**p.next == null**). What should be done? Nothing.

General case: 1) make a recursive call then post-processing or 2) pre-processing then recursive call (only if necessary).

Since the method should be removing the leftmost occurrence, the second strategy should be applied.

# **private void remove( Node<E> p, E o )**

What is the base case? Singleton (**p.next == null**). What should be done? Nothing.

General case: 1) make a recursive call then post-processing or 2) pre-processing then recursive call (only if necessary).

Since the method should be removing the leftmost occurrence, the second strategy should be applied.

Outline the pre-processing that should be done.

# **private void remove( Node<E> p, E o )**

What is the base case? Singleton (**p.next == null**). What should be done? Nothing.

General case: 1) make a recursive call then post-processing or 2) pre-processing then recursive call (only if necessary).

Since the method should be removing the leftmost occurrence, the second strategy should be applied.

Outline the pre-processing that should be done.

If **o** is found at the next position remove it otherwise move forward (make a recursive call).

# private void remove( Node<E> p, E o )

```
private void remove( Node<E> p, E o ) {
    if ( p.next == null ) {
        throw new NoSuchElementException();
    }
    // General case
    if ( p.next.value.equals( o ) ) {
        p.next = p.next.next;
    } else {
        remove( p.next, o );
    }
}
```

# public void remove( E o )

```
public void remove( E o ) {
    if ( first == null ) {
        throw new NoSuchElementException();
    }
    if ( first.value.equals( o ) ) {
        first = first.next;
    } else {
        remove( first, o );
    }
}
private void remove( Node<E> p, E o ) {
    if ( p.next == null ) {
        throw new NoSuchElementException();
    }
    if ( p.next.value.equals( o ) ) {
        p.next = p.next.next;
    } else {
        remove( p.next, o );
    }
}
```

# Exercises

**void add( E c )**; adds the element while preserving the natural order of the elements.

**void removeLast()**; removes the last element of a list.

**void removeLast( E o )**; removes the last occurrence of **o** (this is actually trickier than it seems).

**void removeAll( E o )**; removes all the occurrences of **o**.

**void remove( int pos )**; remove the element found at position **pos**.

# LinkedList<E> subList( int fromIndex, int toIndex )

We now consider methods that are returning a list of values.

# **LinkedList<E> subList( int fromIndex, int toIndex )**

We now consider methods that are returning a list of values.

In particular, the method **LinkedList<E> subList( int fromIndex, int toIndex )** returns a new **LinkedList<E>** that contains the elements found in between positions **fromIndex** and **toIndex** of the original **LinkedList<E>** (the elements are not removed from the original list).

# **LinkedList<E> subList( int fromIndex, int toIndex )**

We now consider methods that are returning a list of values.

In particular, the method **LinkedList<E> subList( int fromIndex, int toIndex )** returns a new **LinkedList<E>** that contains the elements found in between positions **fromIndex** and **toIndex** of the original **LinkedList<E>** (the elements are not removed from the original list).

One of the main issues is to determine a strategy for building the list of results.

# **LinkedList<E> subList( int fromIndex, int toIndex )**

We now consider methods that are returning a list of values.

In particular, the method **LinkedList<E> subList( int fromIndex, int toIndex )** returns a new **LinkedList<E>** that contains the elements found in between positions **fromIndex** and **toIndex** of the original **LinkedList<E>** (the elements are not removed from the original list).

One of the main issues is to determine a strategy for building the list of results. Suggestions?

# **LinkedList<E> subList( int fromIndex, int toIndex )**

We now consider methods that are returning a list of values.

In particular, the method **LinkedList<E> subList( int fromIndex, int toIndex )** returns a new **LinkedList<E>** that contains the elements found in between positions **fromIndex** and **toIndex** of the original **LinkedList<E>** (the elements are not removed from the original list).

One of the main issues is to determine a strategy for building the list of results. Suggestions?

I will be proposing two strategies, for each them we have to know what is the current position, we'll the approach developed for the method **E get( int index )**.

# **LinkedList<E> subList( int fromIndex, int toIndex )**

We now consider methods that are returning a list of values.

In particular, the method **LinkedList<E> subList( int fromIndex, int toIndex )** returns a new **LinkedList<E>** that contains the elements found in between positions **fromIndex** and **toIndex** of the original **LinkedList<E>** (the elements are not removed from the original list).

One of the main issues is to determine a strategy for building the list of results. Suggestions?

I will be proposing two strategies, for each them we have to know what is the current position, we'll the approach developed for the method **E get( int index )**.

**Strategy 1**: traverse the list until the end, when the end is reached return an empty list, following the recursive call, add the current value to the list of result, but only if its position belongs to the interval;

# **LinkedList<E> subList( int fromIndex, int toIndex )**

We now consider methods that are returning a list of values.

In particular, the method **LinkedList<E> subList( int fromIndex, int toIndex )** returns a new **LinkedList<E>** that contains the elements found in between positions **fromIndex** and **toIndex** of the original **LinkedList<E>** (the elements are not removed from the original list).

One of the main issues is to determine a strategy for building the list of results. Suggestions?

I will be proposing two strategies, for each them we have to know what is the current position, we'll the approach developed for the method **E get( int index )**.

**Strategy 1**: traverse the list until the end, when the end is reached return an empty list, following the recursive call, add the current value to the list of result, but only if its position belongs to the interval;

**Strategy 2**: create an empty list to store the values, as the method traverses the list, the elements are added at the end of the list.

# Strategy 1

Recursive calls are traversing the list from **head** to **tail** (from **left** to **right**), the recursion can be stopped upon reaching the **toIndex**.

Base case:

```
LinkedList<E> result;

if ( index == toIndex ) {
   result = new LinkedList<E>();
   result.addFirst( p.value );
}
```

# Strategy 1

**General case**:

```
result = subList( p.next, fromIndex, toIndex, index + 1 );
```

# Strategy 1

**General case**:

```
result = subList( p.next, fromIndex, toIndex, index + 1 );
```

What is **result**?

# Strategy 1

**General case**:

```
result = subList( p.next, fromIndex, toIndex, index + 1 );
```

What is **result**? What's the next step?

# Strategy 1

**General case**:

```
result = subList( p.next, fromIndex, toIndex, index + 1 );
```

What is **result**? What's the next step?

```
if ( index > fromIndex ) {
   result.addFirst( p.value );
}
```

# Strategy 1

```
private LinkedList<E> subList( Node<E> p, int fromIndex, int toIndex, int index ) {
    LinkedList<E> result;

    if ( index == toIndex ) {
        result = new LinkedList<E>();
        result.addFirst( p.value );
    } else {
        result = subList( p.next, fromIndex, toIndex, index + 1 );
        if ( index >= fromIndex ) {
           result.addFirst( p.value );
        }
    }
    return result;
```

Even this more complex method fits the "head+tail" pattern nicely!

# Strategy 1

```
public LinkedList<E> subList( int fromIndex, int toIndex ) {
    return subList( first, fromIndex, toIndex, 0 );
}
```

Handling the pre-conditions (illegal range of values) is left as an exercise.

# Strategy 2

In strategy 2, the list used to store the results is created first and filled with elements as the recursive method proceeds.

```
public LinkedList<E> subList( int fromIndex, int toIndex ) {
    LinkedList<E> result = new LinkedList<E>();
    subList( first, result, fromIndex, toIndex, 0 );
    return result;
}
```

# Strategy 2

**Base case:**

```
if ( index == toIndex ) {
   result.addLast( p.value );
}
```

# Strategy 2

**Base case:**

```
if ( index == toIndex ) {
   result.addLast( p.value );
}
```

**result.addLast( p.value )** ou **result.addFirst( p.value )**?

# Strategy 2

**General case**:

```
if ( index >= fromIndex ) {
   result.addLast( p.value );
}
subList( p.next, result, fromIndex, toIndex, index + 1 );
```

# Strategy 2

```
private void subList( Node<E> p, LinkedList r, int f, int t, int i ) {

    if ( i == t ) {
        r.addLast( p.value );
    } else {
        if ( i >= f ) {
           r.addLast( p.value );
        }
        subList( p.next, r, f, t, i + 1 );
    }
```

Even this more complex method fits the "head+tail" pattern nicely!

Where **f = fromIndex**, **t = toIndex**, **i = index**, **r = result**.

# **LinkedList<E> filterLessThan( E c )**

**LinkedList<E> filterLessThan( E c )** returns in a **LinkedList<E>** all the elements that are less than **c**.

# **LinkedList<E> filterLessThan( E c )**

**LinkedList<E> filterLessThan( E c )** returns in a **LinkedList<E>** all the elements that are less than **c**.

Let's outline our strategy.

# **LinkedList<E> filterLessThan( E c )**

**LinkedList<E> filterLessThan( E c )** returns in a **LinkedList<E>** all the elements that are less than **c**.

Let's outline our strategy.

(Base case:) When to stop?

# **LinkedList<E> filterLessThan( E c )**

**LinkedList<E> filterLessThan( E c )** returns in a **LinkedList<E>** all the elements that are less than **c**.

Let's outline our strategy.

(Base case:) When to stop? At the end of the list, all the elements need to be visited.

# **LinkedList<E> filterLessThan( E c )**

**LinkedList<E> filterLessThan( E c )** returns in a **LinkedList<E>** all the elements that are less than **c**.

Let's outline our strategy.

(Base case:) When to stop? At the end of the list, all the elements need to be visited.

(General case:) What is **result**?

```
result = filterLessThan( c, p.next );
```

# **LinkedList<E> filterLessThan( E c )**

**LinkedList<E> filterLessThan( E c )** returns in a **LinkedList<E>** all the elements that are less than **c**.

Let's outline our strategy.

(Base case:) When to stop? At the end of the list, all the elements need to be visited.

(General case:) What is **result**?

```
result = filterLessThan( c, p.next );
```

All the elements that are less than **c** in the list that starts at **p.next**.

# **LinkedList<E> filterLessThan( E c )**

**LinkedList<E> filterLessThan( E c )** returns in a **LinkedList<E>** all the elements that are less than **c**.

Let's outline our strategy.

(Base case:) When to stop? At the end of the list, all the elements need to be visited.

(General case:) What is **result**?

```
result = filterLessThan( c, p.next );
```

All the elements that are less than **c** in the list that starts at **p.next**.

What is the next step?

# **LinkedList<E> filterLessThan( E c )**

**LinkedList<E> filterLessThan( E c )** returns in a **LinkedList<E>** all the elements that are less than **c**.

Let's outline our strategy.

(Base case:) When to stop? At the end of the list, all the elements need to be visited.

(General case:) What is **result**?

```
result = filterLessThan( c, p.next );
```

All the elements that are less than **c** in the list that starts at **p.next**.

What is the next step? Adding the current element (value) to the list if the value is less than **c**.

# LinkedList<E> filterLessThan( Node<E> p, LinkedList<E> result, E c )

```
private void filterLessThan( Node<E> p, LinkedList<E> result, E c ) {
  if ( p == null ) {
      return;
  }
  filterLessThan( p.next, result, c );
  if ( p.value.compareTo( c ) < 0 )
      result.addFirst( p.value );
}
```

# LinkedList<E> filterLessThan( E c )

```
public LinkedList<E> filterLessThan( E c ) {
  LinkedList<E> result = new LinkedList<E>();
  filterLessThan( first, result, c );
  return result;
}
private void filterLessThan( Node<E> p, LinkedList<E> result, E c ) {
  if ( p == null ) {
      return;
  }
  filterLessThan( p.next, result, c );
  if ( p.value.compareTo( c ) < 0 ) {
      result.addFirst( p.value );
  }
}
```

# LinkedList<E> filterLessThan( E c ) (take 2)

```
public LinkedList<E> filterLessThan( E c ) {
  return filterLessThan( first, c );
}
private LinkedList<E> filterLessThan( Node<E> p, E c ) {
  LinkedList<E> result;
  if ( p == null ) {
      result = new LinkedList<E>();
  } else {
      result = filterLessThan( p.next, c );
      if ( p.value.compareTo( c ) < 0 ) {
          result.addFirst( p.value );
      }
  }
  return result;
}
```

# LinkedList<E> filter( Predicate<E> f )

```
private LinkedList<E> filterLessThan( Node<E> p, E c ) {
  LinkedList<E> result;
  if ( p == null ) {
      result = new LinkedList<E>();
  } else {
      result = filterLessThan( p.next, c );
      if ( p.value.compareTo( c ) < 0 ) {
          result.addFirst( p.value );
      }
  }
  return result;
}
```

What needs to be changed to implement **filterGreaterThan**?

# LinkedList<E> filter( Predicate<E> f )

```
private LinkedList<E> filterLessThan( Node<E> p, E c ) {
  LinkedList<E> result;
  if ( p == null ) {
      result = new LinkedList<E>();
  } else {
      result = filterLessThan( p.next, c );
      if ( p.value.compareTo( c ) < 0 ) {
          result.addFirst( p.value );
      }
  }
  return result;
}
```

What needs to be changed to implement **filterGreaterThan**? How about **filterGenderMale**?

# LinkedList<E> filter( Predicate<E> f )

```
private LinkedList<E> filterLessThan( Node<E> p, E c ) {
  LinkedList<E> result;
  if ( p == null ) {
      result = new LinkedList<E>();
  } else {
      result = filterLessThan( p.next, c );
      if ( p.value.compareTo( c ) < 0 ) {
          result.addFirst( p.value );
      }
  }
  return result;
}
```

What needs to be changed to implement **filterGreaterThan**? How about **filterGenderMale**? How about . . . ?

# Predicate

```
public interface Predicate<E> {
    public abstract boolean evaluate( E arg );
}
```

# IsNegative

```
public class IsNegative implements Predicate<Integer> {
    public boolean evaluate( Integer arg ) {
        return arg.intValue() < 0;
    }
}
```

# LinkedList<E> filter( Predicate<E> f )

```
public LinkedList<E> filter( Predicate<E> f ) {
  return filter( first, f );
}
private LinkedList<E> filter( Node<E> p, Predicate f ) {
  LinkedList<E> result;
  if ( p == null ) {
      result = new LinkedList<E>();
  } else {
      result = filter( p.next, f );
      if ( f.evaluate( p.value ) ) {
          result.addFirst( p.value );
      }
  }
  return result;
}
```

# IsNegative

```
l2 = l1.filter( new IsNegative() );
```

# IsPositive (anonymous)

```
l2 = l1.filter( new Predicate<Integer>() {
    public boolean evaluate( Integer arg ) {
        return args.intValue() > 0;
    }
});
```

# “Head + tail”

```
if ( ... ) { // base case

   calculate results

} else { // general case

  // pre-processing

  s = method( p.next ); // recursion

  // post-processing

}
```