ITI 1121. Introduction to Computing II *

Marcel Turcotte
School of Electrical Engineering and Computer Science

Version of March 24, 2013

Abstract

- Recursive list processing (part I)

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

“To iterate is human, to recurse divine.”
(L. Peter Deutsch)

Recursion: reminders

1. The solution to a given problem can be obtained by combining the solution of sub-problem(s);

Recursion: reminders

1. The solution to a given problem can be obtained by combining the solution of sub-problem(s);

2. The sub-problems are of the same type (i.e. the same strategy applies);

Recursion: reminders

1. The solution to a given problem can be obtained by combining the solution of sub-problem(s);

2. The sub-problems are of the same type (i.e. the same strategy applies);

3. The sub-problems are always smaller (converges);

Recursion: reminders

1. The solution to a given problem can be obtained by combining the solution of sub-problem(s);

2. The sub-problems are of the same type (i.e. the same strategy applies);

3. The sub-problems are always smaller (converges);

4. There is a size of problem that can be solved trivially (no recursive calls).

Recursion: reminders

1. The solution to a given problem can be obtained by combining the solution of sub-problem(s);

2. The sub-problems are of the same type (i.e. the same strategy applies);

3. The sub-problems are always smaller (converges);

4. There is a size of problem that can be solved trivially (no recursive calls).

What problems have you solved using recursion?

Recursion: reminders

1. The solution to a given problem can be obtained by combining the solution of sub-problem(s);

2. The sub-problems are of the same type (i.e. the same strategy applies);

3. The sub-problems are always smaller (converges);

4. There is a size of problem that can be solved trivially (no recursive calls).

What problems have you solved using recursion? Calculating the factorial?

Recursion: reminders

1. The solution to a given problem can be obtained by combining the solution of sub-problem(s);

2. The sub-problems are of the same type (i.e. the same strategy applies);

3. The sub-problems are always smaller (converges);

4. There is a size of problem that can be solved trivially (no recursive calls).

What problems have you solved using recursion? Calculating the factorial? Locating a value in array?

Recursion: reminders

1. The solution to a given problem can be obtained by combining the solution of sub-problem(s);

2. The sub-problems are of the same type (i.e. the same strategy applies);

3. The sub-problems are always smaller (converges);

4. There is a size of problem that can be solved trivially (no recursive calls).

What problems have you solved using recursion? Calculating the factorial? Locating a value in array? Here recursion is used to traverse (singly) linked lists.

Problem (intuition)

Consider calculating the sum of the values of an array t.

Problem (intuition)

Consider calculating the sum of the values of an array t.

The whole problem consists of calculating the sum of the values, $t[k]$, for the interval $k = 0$ to $length - 1$.

Problem (intuition)

Consider calculating the sum of the values of an array t.

The whole problem consists of calculating the sum of the values, $t[k]$, for the interval $k = 0$ to $length - 1$.

Let's call s the sum of all the values for the interval $k = 1$ to $length - 1$.

Problem (intuition)

Consider calculating the sum of the values of an array t.

The whole problem consists of calculating the sum of the values, $t[k]$, for the interval $k = 0$ to $length - 1$.

Let's call s the sum of all the values for the interval $k = 1$ to $length - 1$.

Furthermore, let's assume that s has been pre-calculated.

Problem (intuition)

Consider calculating the sum of the values of an array t.

The whole problem consists of calculating the sum of the values, $t[k]$, for the interval $k = 0$ to $length - 1$.

Let's call s the sum of all the values for the interval $k = 1$ to $length - 1$.

Furthermore, let's assume that s has been pre-calculated. What is the solution to our problem?

Problem (intuition)

Consider calculating the sum of the values of an array t.

The whole problem consists of calculating the sum of the values, $t[k]$, for the interval $k = 0$ to $length - 1$.

Let's call s the sum of all the values for the interval $k = 1$ to $length - 1$.

Furthermore, let's assume that s has been pre-calculated. What is the solution to our problem? $t[0] + s$

Problem (intuition)

Consider calculating the sum of the values of an array t.

The whole problem consists of calculating the sum of the values, $t[k]$, for the interval $k = 0$ to $length - 1$.

Let's call s the sum of all the values for the interval $k = 1$ to $length - 1$.

Furthermore, let's assume that s has been pre-calculated. What is the solution to our problem? $t[0] + s$

How to calculate s, the sum of all the values for the sub-interval $k = 1 \ldots length - 1$.

Problem (intuition)

Consider calculating the sum of the values of an array t.

The whole problem consists of calculating the sum of the values, $t[k]$, for the interval $k = 0$ to $length - 1$.

Let's call s the sum of all the values for the interval $k = 1$ to $length - 1$.

Furthermore, let's assume that s has been pre-calculated. What is the solution to our problem? $t[0] + s$

How to calculate s, the sum of all the values for the sub-interval $k = 1 \ldots length - 1$.

Similarly, except that the interval is smaller by one position.

Problem (intuition)

Consider calculating the sum of the values of an array t.

The whole problem consists of calculating the sum of the values, $t[k]$, for the interval $k = 0$ to $length - 1$.

Let's call s the sum of all the values for the interval $k = 1$ to $length - 1$.

Furthermore, let's assume that s has been pre-calculated. What is the solution to our problem? $t[0] + s$

How to calculate s, the sum of all the values for the sub-interval $k = 1 \ldots length - 1$.

Similarly, except that the interval is smaller by one position.

What size of problem can be solved trivially?

Problem (intuition)

Consider calculating the sum of the values of an array t.

The whole problem consists of calculating the sum of the values, $t[k]$, for the interval $k = 0$ to $length - 1$.

Let's call s the sum of all the values for the interval $k = 1$ to $length - 1$.

Furthermore, let's assume that s has been pre-calculated. What is the solution to our problem? $t[0] + s$

How to calculate s, the sum of all the values for the sub-interval $k = 1 \ldots length - 1$.

Similarly, except that the interval is smaller by one position.

What size of problem can be solved trivially? An interval of size 1.

Problem (continued)

```
private static int sum( int[] t, int k ) {

    int s, result, length = t.length - k;

    if ( length == 1 ) { // Base case

        result = t[ k ];

    } else { // General case

       int k1 = k+1;
       s = sum( t, k1 );
       result = t[ k ] + s;

    }
    return result;
}
```

What would be the initial call?

What would be the initial call?

```
public static int sum( int[] t ) {
    return sum( t, 0 );
}
```

```
public class Sum {
    private static int sum( int[] t, int k ) {
        if ( k == ( t.length - 1 ) ) {
            return t[ k ];
        }
        return t[ k ] + sum( t, k+1 );
    }
    public static int sum( int[] t ) {
        if ( t.length == 0 ) {
            throw new IllegalArgumentException();
        }
        return sum( t, 0 );
    }
    public static void main( String[] args ) {
        int[] t = { 1, 2, 3, 4, 5 };
        System.out.println( sum( t ) );
    }
}
```

Remarks

It's important that the size of the problems gets smaller with each successive call, otherwise this would create an infinite recursion (similar to infinite loops).

Remarks

It's important that the size of the problems gets smaller with each successive call, otherwise this would create an infinite recursion (similar to infinite loops).

To stop the recursion, there must be a size of problem such that the result can be computed directly (no recursive call).

Remarks

It's important that the size of the problems gets smaller with each successive call, otherwise this would create an infinite recursion (similar to infinite loops).

To stop the recursion, there must be a size of problem such that the result can be computed directly (no recursive call).

Let's call the special cases where the result can be obtained directly the **base cases**, there is at least one but there can be more than one base case.

Remarks

It's important that the size of the problems gets smaller with each successive call, otherwise this would create an infinite recursion (similar to infinite loops).

To stop the recursion, there must be a size of problem such that the result can be computed directly (no recursive call).

Let's call the special cases where the result can be obtained directly the **base cases**, there is at least one but there can be more than one base case.

Base cases should be checked first so as to stop the recursion.

Remarks

It's important that the size of the problems gets smaller with each successive call, otherwise this would create an infinite recursion (similar to infinite loops).

To stop the recursion, there must be a size of problem such that the result can be computed directly (no recursive call).

Let's call the special cases where the result can be obtained directly the **base cases**, there is at least one but there can be more than one base case.

Base cases should be checked first so as to stop the recursion.

Programming languages such as **Lisp**, **Prolog** and **Haskell** have no loop control-structures and therefore recursion is the only way to create iterations!

Pattern (pseudo-code)

```
type method( parameters ) {

    type result;

    if ( test parameters for base case ) { // base case

        // calculate the result without recursion, recursion stop here

    } else { // general case

        // pre-processing: partitioning the data for example

        result = method( sub-set of the data ); // recursive call

        // pos-processing: combining the results for example

    }
    return result;
}
```

Factorial

```
public static int factorial( int n ) {
    int s, result;
    if ( n<=1 ) { // base case
        result = 1;
    } else { // general case
        int n1 = n-1;
        s = factorial( n1 );
        result = n * s;
    }
    return result;
}
```

The factorial method corresponds to the general pattern, the base case is tested first, and the result is computed directly without the need for a recursive call (recursion stops here!), the general case is always making the value of the parameter smaller so that eventually the base case will be applied.

Factorial

```
public static int factorial( int n ) {

    if ( n<=1 ) {
        return 1;
    }

    return n * factorial( n-1 );
}
```

A **return** statement returns the control to the caller, it stops the execution of the method, no other statements of the call will be executed.

Binary Search

```
public static int binarySearch( int value, int[] array ) {
    return binSearch( value, array, 0, array.length-1 );
}
```

A **binarySearch** can be applied to search for a target value in a sorted array.

Binary Search

```
public static int binarySearch( int value, int[] array ) {
    return binSearch( value, array, 0, array.length-1 );
}
```

A **binarySearch** can be applied to search for a target value in a sorted array.

This method is efficient because the size of the intervals of positions to search decreases by a factor of two with every successive call.

Binary Search

```
public static int binarySearch( int value, int[] array ) {
    return binSearch( value, array, 0, array.length-1 );
}
```

A **binarySearch** can be applied to search for a target value in a sorted array.

This method is efficient because the size of the intervals of positions to search decreases by a factor of two with every successive call.

Base case: an interval of size zero, no recursive call or the value was found at the position **middle**.

Binary Search

```
public static int binarySearch( int value, int[] array ) {
    return binSearch( value, array, 0, array.length-1 );
}
```

A **binarySearch** can be applied to search for a target value in a sorted array.

This method is efficient because the size of the intervals of positions to search decreases by a factor of two with every successive call.

Base case: an interval of size zero, no recursive call or the value was found at the position **middle**.

General case: creates smaller and smaller intervals of values for the search.

```java
private static int binSearch( int value, int[] array, int lo, int hi ) {

    // Base case: value not found
    if ( lo > hi ) {
        return -1;
    }
    // Base case: value was found
    int middle = ( lo + hi ) / 2;
    if ( value == array[ middle ] ) {
        return middle;
    }
    // General case:
    int newLo, newHi;
    if ( value < array[ middle ] ) {
        newLo = lo;
        newHi = middle - 1;
    } else {
        newLo = middle + 1;
        newHi = hi;
    }
    return binarySearch( value, array, newLo, newHi );
}
```

Recursive list processing — “head + tail” strategy

Let’s develop a general strategy to solve list processing problems recursively.

Recursive list processing — "head + tail" strategy

Let's develop a general strategy to solve list processing problems recursively.

To develop this strategy, let's consider calculating the **size** (length) of a list.

Recursive list processing — “head + tail” strategy

Let’s develop a general strategy to solve list processing problems recursively.

To develop this strategy, let’s consider calculating the **size** (length) of a list.

Let’s divide the initial list in two parts, its **head** (the first element) and its **tail** (the rest of the list).

Recursive list processing — “head + tail” strategy

Let’s develop a general strategy to solve list processing problems recursively.

To develop this strategy, let’s consider calculating the **size** (length) of a list.

Let’s divide the initial list in two parts, its **head** (the first element) and its **tail** (the rest of the list).

By analogy with the method **sum**, let’s say that we can obtain the size of the **tail** by some mean, let’s call this value s.

Recursive list processing —
“head + tail” strategy

Let’s develop a general strategy to solve list processing problems recursively.

To develop this strategy, let’s consider calculating the **size** (length) of a list.

Let’s divide the initial list in two parts, its **head** (the first element) and its **tail** (the rest of the list).

By analogy with the method **sum**, let’s say that we can obtain the size of the **tail** by some mean, let’s call this value s.

Knowing the value of s, what is the size of the entire list?

Recursive list processing — “head + tail” strategy

Let’s develop a general strategy to solve list processing problems recursively.

To develop this strategy, let’s consider calculating the **size** (length) of a list.

Let’s divide the initial list in two parts, its **head** (the first element) and its **tail** (the rest of the list).

By analogy with the method **sum**, let’s say that we can obtain the size of the **tail** by some mean, let’s call this value s.

Knowing the value of s, what is the size of the entire list?

Answer: $s + 1$.

Recursive list processing — “head + tail” strategy

Let’s develop a general strategy to solve list processing problems recursively.

To develop this strategy, let’s consider calculating the **size** (length) of a list.

Let’s divide the initial list in two parts, its **head** (the first element) and its **tail** (the rest of the list).

By analogy with the method **sum**, let’s say that we can obtain the size of the **tail** by some mean, let’s call this value s.

Knowing the value of s, what is the size of the entire list?

Answer: $s + 1$.

How to obtain s?

Recursive list processing — “head + tail” strategy

Let’s develop a general strategy to solve list processing problems recursively.

To develop this strategy, let’s consider calculating the **size** (length) of a list.

Let’s divide the initial list in two parts, its **head** (the first element) and its **tail** (the rest of the list).

By analogy with the method **sum**, let’s say that we can obtain the size of the **tail** by some mean, let’s call this value s.

Knowing the value of s, what is the size of the entire list?

Answer: $s + 1$.

How to obtain s? Well, s is the size of a list,

Recursive list processing —
"head + tail" strategy

Let's develop a general strategy to solve list processing problems recursively.

To develop this strategy, let's consider calculating the **size** (length) of a list.

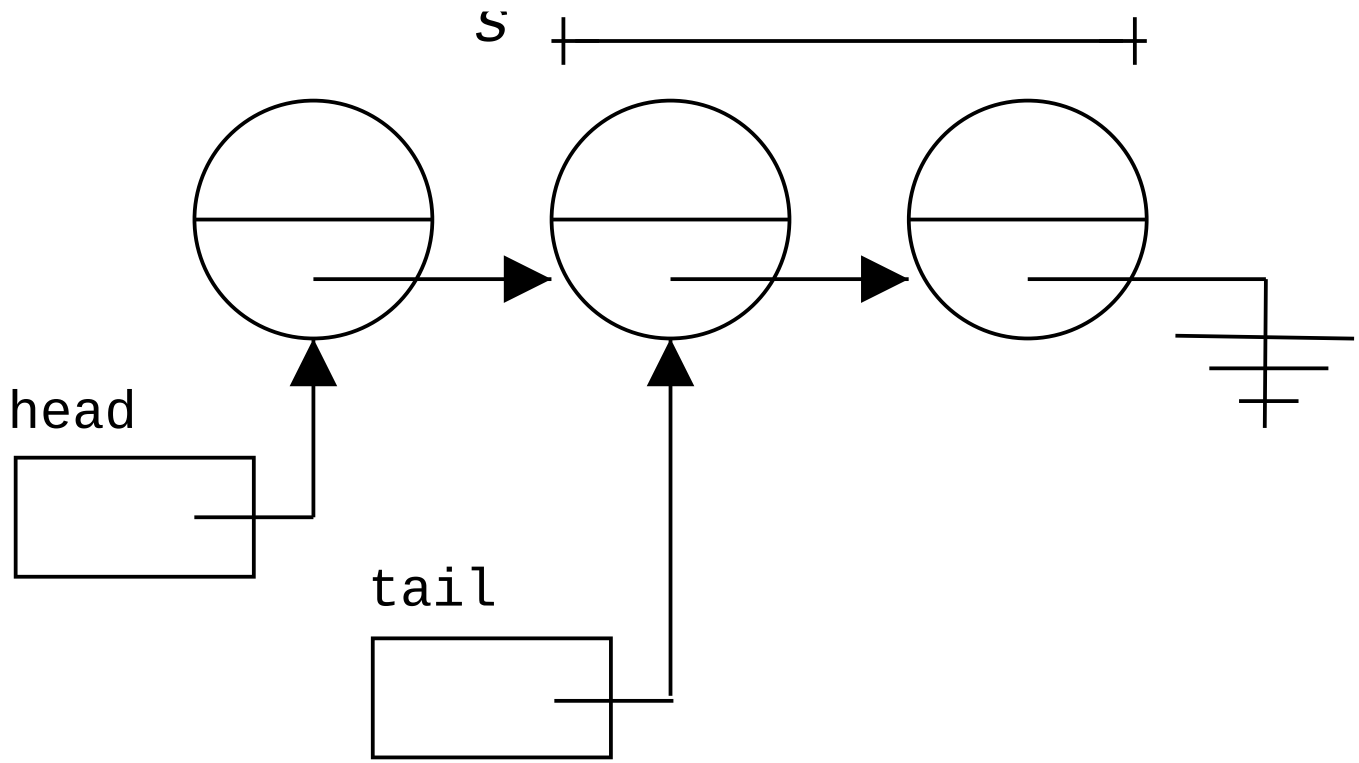
Let's divide the initial list in two parts, its **head** (the first element) and its **tail** (the rest of the list).

By analogy with the method **sum**, let's say that we can obtain the size of the **tail** by some mean, let's call this value s.

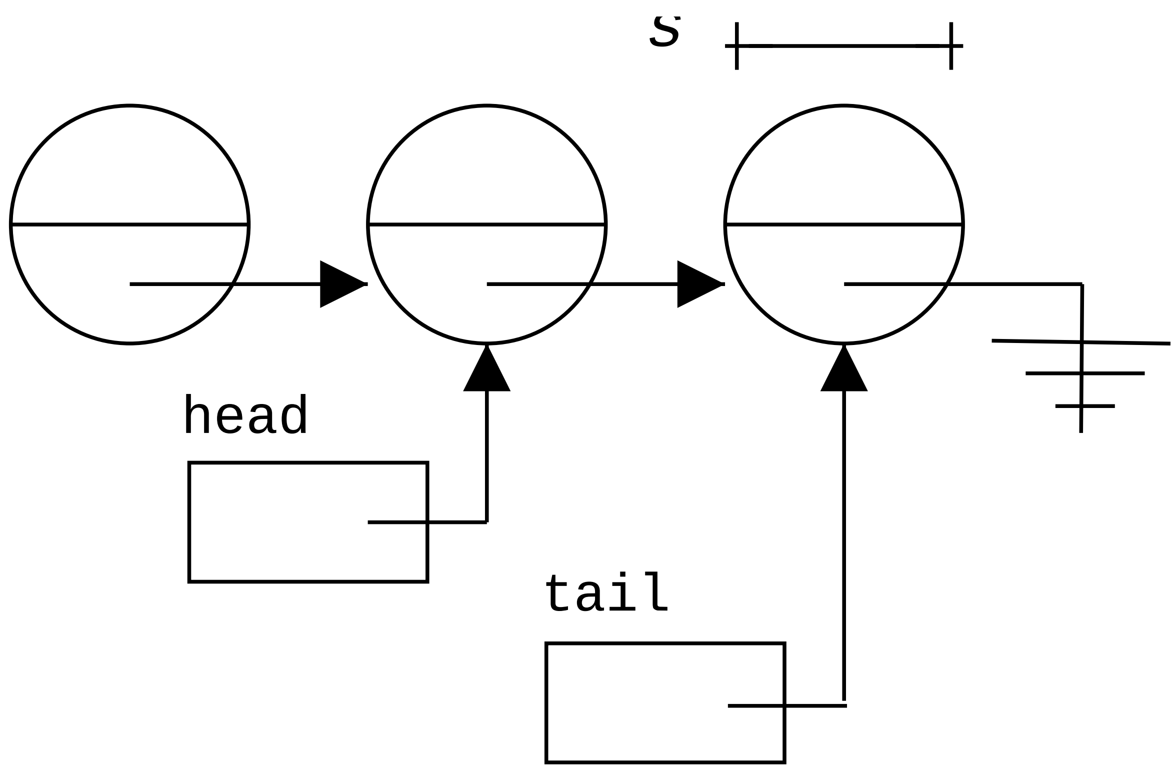
Knowing the value of s, what is the size of the entire list?

Answer: $s + 1$.

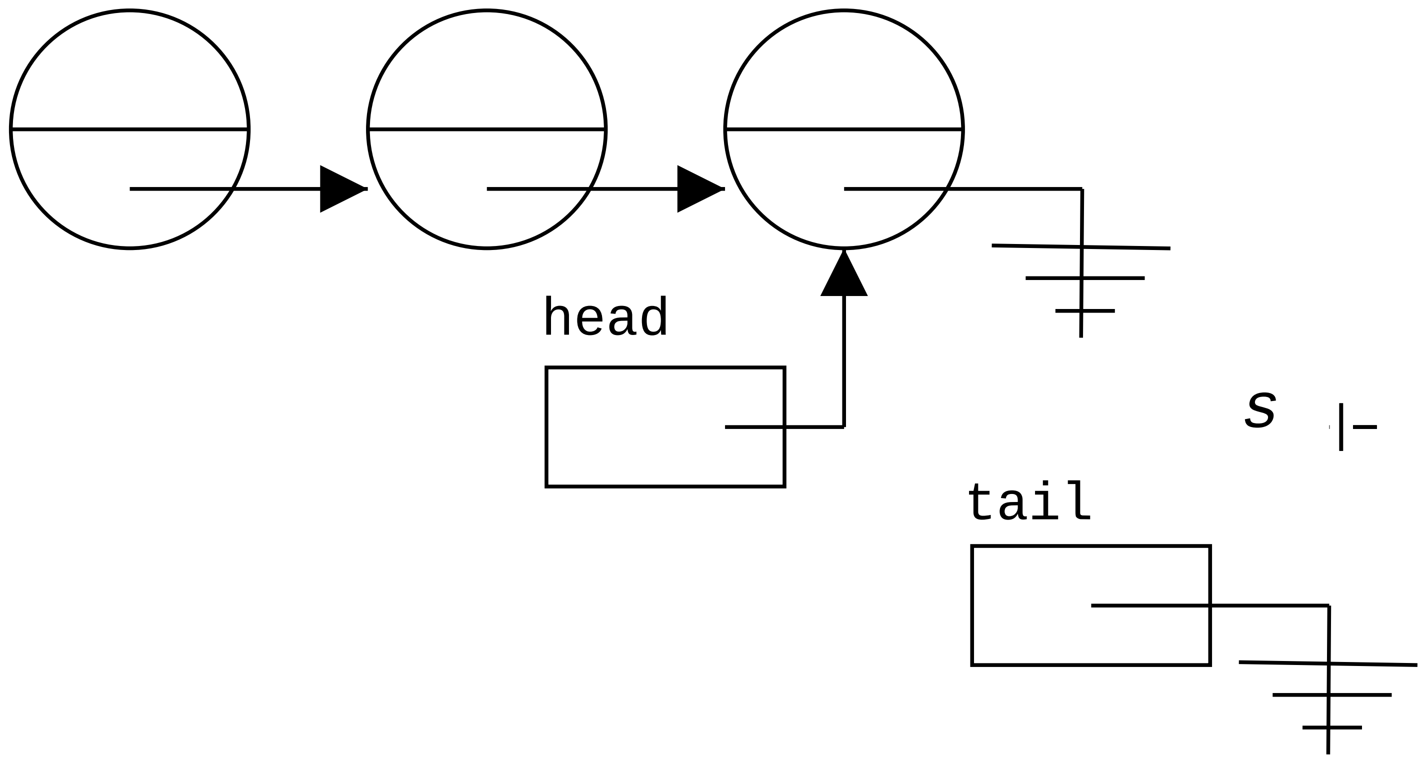
How to obtain s? Well, s is the size of a list, and, furthermore, **it's a shorter one**, we can apply (recursively) the method that we are developing.



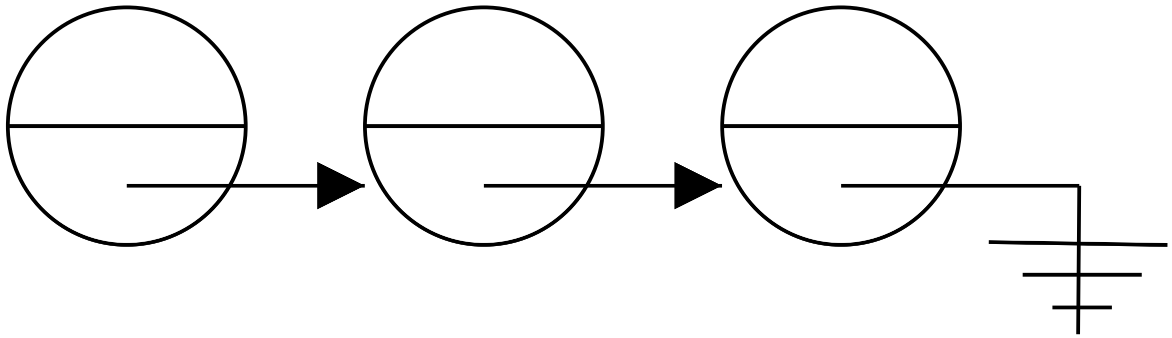
$\Rightarrow$ The total length of the list will $s + 1$. **Note: here, head and tail are referring to the current node and remaining nodes**.

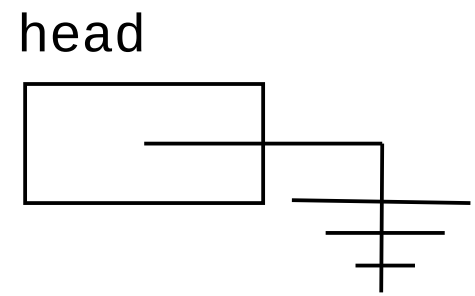


$\Rightarrow$ The length of the list designated by **head** will be length of $s + 1$.



$\Rightarrow$ The length of the list designated by **head** will be length of $s + 1$.





$\Rightarrow$ The length of the list designated by **head** is 0.

Implementation

The methods presented in this lecture are instance methods of a class defined as follows.

```
public class OrderedList< E extends Comparable<E> > {
    private static class Node<E> {
        private E value;
        private Node<E> next;
        Node( E value, Node next ) {
            this.value = value;
            this.next = next;
        }
    }
    private Node<E> first; // <---
    public OrderedList () {
        first = null;
    }
    // other methods ...
}
```

“head + tail” strategy

Given a list **l**, apply the method recursively to the rest of the list (**tail**).

“head + tail” strategy

Given a list **l**, apply the method recursively to the rest of the list (**tail**).

In general, the result obtained by the recursive call is used in combination with the head to calculate a final result.

“head + tail” strategy

Given a list **l**, apply the method recursively to the rest of the list (**tail**).

In general, the result obtained by the recursive call is used in combination with the head to calculate a final result.

This strategy does not solve all recursive list processing problems but it’s the first strategy you should try.

“head + tail” strategy

Given a list **l**, apply the method recursively to the rest of the list (**tail**).

In general, the result obtained by the recursive call is used in combination with the head to calculate a final result.

This strategy does not solve all recursive list processing problems but it’s the first strategy you should try.

For the method **size**, what would be the base case?

“head + tail” strategy

Given a list **l**, apply the method recursively to the rest of the list (**tail**).

In general, the result obtained by the recursive call is used in combination with the head to calculate a final result.

This strategy does not solve all recursive list processing problems but it’s the first strategy you should try.

For the method **size**, what would be the base case?

For each recursive call, the list gets smaller and smaller, what is then the smallest list.

“head + tail” strategy

Given a list **l**, apply the method recursively to the rest of the list (**tail**).

In general, the result obtained by the recursive call is used in combination with the head to calculate a final result.

This strategy does not solve all recursive list processing problems but it’s the first strategy you should try.

For the method **size**, what would be the base case?

For each recursive call, the list gets smaller and smaller, what is then the smallest list.

The empty list!

“head + tail” strategy

Given a list **l**, apply the method recursively to the rest of the list (**tail**).

In general, the result obtained by the recursive call is used in combination with the head to calculate a final result.

This strategy does not solve all recursive list processing problems but it’s the first strategy you should try.

For the method **size**, what would be the base case?

For each recursive call, the list gets smaller and smaller, what is then the smallest list.

The empty list!

The empty list is a valid case, its length is zero, however, it has to be a base case because it has no tail!

“head + tail” strategy

Given a list **l**, apply the method recursively to the rest of the list (**tail**).

In general, the result obtained by the recursive call is used in combination with the head to calculate a final result.

This strategy does not solve all recursive list processing problems but it’s the first strategy you should try.

For the method **size**, what would be the base case?

For each recursive call, the list gets smaller and smaller, what is then the smallest list.

The empty list!

The empty list is a valid case, its length is zero, however, it has to be a base case because it has no tail!

Because of that, for all methods that apply the “head + tail” strategy the empty list cannot be part of the general case.

```
int size( Node<E> p ) {
    int s, length;

    if ( p == null ) { // Base case

        length = 0;

    } else { // General case

        s = size( p.next );
        length = 1 + s;

    }
    return length;
}
```

Or simply,

```
int size( Node<E> p ) {
    if ( p == null ) {
        return 0;
    }
    return 1 + size( p.next );
}
```

Or simply,

```
int size( Node<E> p ) {
    if ( p == null ) {
        return 0;
    }
    return 1 + size( p.next );
}
```

⇒ Can this method be called from outside of the class?

Or simply,

```
int size( Node<E> p ) {
    if ( p == null ) {
        return 0;
    }
    return 1 + size( p.next );
}
```

⇒ Can this method be called from outside of the class? What is the first (initial) call?

No, it cannot be called from outside of the class because it needs to be given a reference to a **Node** (which is an implementation detail and should be **private**).

No, it cannot be called from outside of the class because it needs to be given a reference to a **Node** (which is an implementation detail and should be **private**).

We need an auxiliary method that initiates the process starting with the first element of the list:

```
public int size() {
    return size( first );
}
```

Because, the recursive method cannot and should not be used from outside of the class, it should be declared private:

```
public int size() {
    return size( first );
}

private int size( Node<E> p ) {
    if ( p == null ) {
        return 0;
    }
    return 1 + size( p.next );
}
```

Because, the recursive method cannot and should not be used from outside of the class, it should be declared private:

```
public int size() {
    return size( first );
}

private int size( Node<E> p ) {
    if ( p == null ) {
        return 0;
    }
    return 1 + size( p.next );
}
```

All our recursive methods will obey this pattern. They will all have a **public** part that calls the **recursive** methods, **which is private**. The public method initiates the first call to the recursive method with a reference to the first node.

The "head $+$ tail" strategy is not the only way to solve this problem, we could have chosen to divide the list into two sub-lists of approximately the same size and sum their lengths.

The “head + tail” strategy is not the only way to solve this problem, we could have chosen to divide the list into two sub-lists of approximately the same size and sum their lengths.

In fact, for a list of length n, there are $n - 1$ ways to separate this list into two sub-lists.

The “head + tail” strategy is not the only way to solve this problem, we could have chosen to divide the list into two sub-lists of approximately the same size and sum their lengths.

In fact, for a list of length n, there are $n - 1$ ways to separate this list into two sub-lists.

The “head + tail” strategy is just a special case; however, no recursive call is made for the head.

The “head + tail” strategy is not the only way to solve this problem, we could have chosen to divide the list into two sub-lists of approximately the same size and sum their lengths.

In fact, for a list of length n, there are $n - 1$ ways to separate this list into two sub-lists.

The “head + tail” strategy is just a special case; however, no recursive call is made for the head.

Given our list implementation, the “head + strategy” strategy is simpler to apply.

The "head + tail" strategy is not the only way to solve this problem, we could have chosen to divide the list into two sub-lists of approximately the same size and sum their lengths.

In fact, for a list of length n, there are $n - 1$ ways to separate this list into two sub-lists.

The "head + tail" strategy is just a special case; however, no recursive call is made for the head.

Given our list implementation, the "head + strategy" strategy is simpler to apply.

The strategy that consists in dividing the list into two sub-lists of approximately the same size can lead to more efficient algorithms, but that's a subject left for CSI 2114 (data-structures), here, we'll use the "head + tail" strategy.

“Head + tail”

```
if ( ... ) { // base case
   calculate results
} else { // general case
  // pre-processing
  s = method( p.next ); // recursion
  // post-processing
}
```

“Head + tail”

Steps toward building the general case.

“Head + tail”

Steps toward building the general case.

What is **method(p.next)**?

"Head + tail"

Steps toward building the general case.

What is **method(p.next)**?

The solution to a sub-problem, which is one element smaller.

“Head + tail”

Steps toward building the general case.

What is **method(p.next)**?

The solution to a sub-problem, which is one element smaller.

How can you use it to build the solution for the list starting at **p**?

“Head + tail”

Steps toward building the general case.

What is **method(p.next)**?

The solution to a sub-problem, which is one element smaller.

How can you use it to build the solution for the list starting at **p**?

Building the base case(s).

“Head + tail”

Steps toward building the general case.

What is **method(p.next)**?

The solution to a sub-problem, which is one element smaller.

How can you use it to build the solution for the list starting at **p**?

Building the base case(s).

What is the smallest valid list?

“Head + tail”

Steps toward building the general case.

What is **method(p.next)**?

The solution to a sub-problem, which is one element smaller.

How can you use it to build the solution for the list starting at **p**?

Building the base case(s).

What is the smallest valid list? What is the result?

E findMax()

Write a recursive method that finds the maximum value for a list of $<$ **E extends Comparable**$<$**E**$>$ $>$ objects.

E findMax()

Write a recursive method that finds the maximum value for a list of < **E extends Comparable**<**E**> > objects.

Let's take care of the **public non-recursive** part first.

E findMax()

Write a recursive method that finds the maximum value for a list of $<$ **E extends Comparable**$<$**E**$>$ $>$ objects.

Let's take care of the **public non-recursive** part first.

What is the smallest valid list?

E findMax()

Write a recursive method that finds the maximum value for a list of $<$ **E extends Comparable**$<$**E**$>$ $>$ objects.

Let's take care of the **public non-recursive** part first.

What is the smallest valid list? The smallest valid list contains one element.

E findMax()

Write a recursive method that finds the maximum value for a list of $<$ **E extends Comparable**$<$**E**$>$ $>$ objects.

Let's take care of the **public non-recursive** part first.

What is the smallest valid list? The smallest valid list contains one element.

What are we going to do if the list is empty?

E findMax()

Write a recursive method that finds the maximum value for a list of $<$ **E extends Comparable**$<$**E**$>$ $>$ objects.

Let's take care of the **public non-recursive** part first.

What is the smallest valid list? The smallest valid list contains one element.

What are we going to do if the list is empty? Throwing an exception.

Where should the recursion start?

E findMax()

Write a recursive method that finds the maximum value for a list of $<$ **E extends Comparable**$<$**E**$>$ $>$ objects.

Let's take care of the **public non-recursive** part first.

What is the smallest valid list? The smallest valid list contains one element.

What are we going to do if the list is empty? Throwing an exception.

Where should the recursion start? It'll start with the **first** element of the list.

E findMax()

```
public E findMax() {
    if ( first == null ) {
        throw new NoSuchElementException();
    }
    return findMax( first );
}
```

Remark: the recursive method will always have at least one more parameter than the non-recursive method.

E findMax()

```
public E findMax() {
    if ( first == null ) {
        throw new NoSuchElementException();
    }
    return findMax( first );
}
```

Remark: the recursive method will always have at least one more parameter than the non-recursive method. Why?

E findMax()

```
public E findMax() {
    if ( first == null ) {
        throw new NoSuchElementException();
    }
    return findMax( first );
}
```

Remark: the recursive method will always have at least one more parameter than the non-recursive method. Why? The parameter designates the current element for this step.

E findMax(Node<E> p)

Let's apply the "head+tail" strategy and consider the general case first,

```
E result = findMax( p.next );
```

What does **result** represents?

E findMax(Node<E> p)

Let's apply the "head+tail" strategy and consider the general case first,

```
E result = findMax( p.next );
```

What does **result** represents? Hum, it's the maximum value for the rest of the list.

E findMax(Node<E> p)

Let's apply the "head+tail" strategy and consider the general case first,

```
E result = findMax( p.next );
```

What does **result** represents? Hum, it's the maximum value for the rest of the list.

What should be done next?

E findMax(Node<E> p)

Let's apply the "head+tail" strategy and consider the general case first,

```
E result = findMax( p.next );
```

What does **result** represents? Hum, it's the maximum value for the rest of the list.

What should be done next?

Compare **result** to the value of the current **Node**.

```
if ( result.compareTo( p.value ) > 0 ) {
    return result;
} else {
    return p.value;
}
```

E findMax(Node<E> p)

What should be the base case?

E findMax(Node<E> p)

What should be the base case?

This process makes the list smaller and smaller, what is the smallest list to handle?

E findMax(Node<E> p)

What should be the base case?

This process makes the list smaller and smaller, what is the smallest list to handle?

No,

E findMax(Node<E> p)

What should be the base case?

This process makes the list smaller and smaller, what is the smallest list to handle?

No, not the empty list, we said the smallest valid list contains one element.

E findMax(Node<E> p)

What should be the base case?

This process makes the list smaller and smaller, what is the smallest list to handle?

No, not the empty list, we said the smallest valid list contains one element.

What value should be returned in that case?

E findMax(Node<E> p)

What should be the base case?

This process makes the list smaller and smaller, what is the smallest list to handle?

No, not the empty list, we said the smallest valid list contains one element.

What value should be returned in that case?

The value found at that node.

E findMax(Node<E> p)

What should be the base case?

This process makes the list smaller and smaller, what is the smallest list to handle?

No, not the empty list, we said the smallest valid list contains one element.

What value should be returned in that case?

The value found at that node.

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

E findMax(Node<E> p)

```
public E findMax() {
    if ( first == null ) {
        throw new NoSuchElementException();
    }
    return findMax( first );
}
private E findMax( Node<E> p ) {
    if ( p.next == null ) {
        return p.value;
    }
    E result = findMax( p.next );
    if ( result.compareTo( p.value ) > 0 ) {
        return result;
    } else {
        return p.value;
    }
}
```