



Assignment 5

Felix Friedrich, Lars Widmer, Fabian Stutz

TA lecture, Informatics II D-BAUG

March 18, 2014

“Präsenzstunden” Today

In the same room as in the **first** week

- **HIL E 15.2**
- **15:00 - 18:00**
- Timon Gehr (*arriving 15:45*)

Outline

- 1 Know How
 - Fibonacci Numbers
 - Linked List
 - Bubble Sort
- 2 Prediscussion Assignment 5
 - Fibonacci Implementation
 - Linked List Implementation
- 3 Postdiscussion Assignment 4
 - GrowingArray
 - Test & Comparison
 - Stack

Fibonacci Numbers

The **Fibonacci numbers** are defined as follows.

$$f_1 = f_2 = 1 \quad (1)$$

$$f_n = f_{n-1} + f_{n-2} \quad (2)$$

Calculate Numbers

$$f_1 = f_2 = 1 \quad (3)$$

$$f_n = f_{n-1} + f_{n-2} \quad (4)$$

Calculate the following Fibonacci numbers:

1 2

Calculate Numbers

$$f_1 = f_2 = 1 \quad (3)$$

$$f_n = f_{n-1} + f_{n-2} \quad (4)$$

Calculate the following Fibonacci numbers:

1 2

2 3

Calculate Numbers

$$f_1 = f_2 = 1 \quad (3)$$

$$f_n = f_{n-1} + f_{n-2} \quad (4)$$

Calculate the following Fibonacci numbers:

- 1 2
- 2 3
- 3 5

Calculate Numbers

$$f_1 = f_2 = 1 \quad (3)$$

$$f_n = f_{n-1} + f_{n-2} \quad (4)$$

Calculate the following Fibonacci numbers:

1 2

2 3

3 5

4 8

Calculate Numbers

$$f_1 = f_2 = 1 \quad (3)$$

$$f_n = f_{n-1} + f_{n-2} \quad (4)$$

Calculate the following Fibonacci numbers:

- 1 2
- 2 3
- 3 5
- 4 8
- 5 13

Calculate Numbers

$$f_1 = f_2 = 1 \quad (3)$$

$$f_n = f_{n-1} + f_{n-2} \quad (4)$$

Calculate the following Fibonacci numbers:

- 1 2
- 2 3
- 3 5
- 4 8
- 5 13
- 6 21

Calculate Numbers

$$f_1 = f_2 = 1 \quad (3)$$

$$f_n = f_{n-1} + f_{n-2} \quad (4)$$

Calculate the following Fibonacci numbers:

- 1 2
- 2 3
- 3 5
- 4 8
- 5 13
- 6 21
- 7 34

Calculation in Java

$$f_1 = f_2 = 1 \quad (5)$$

$$f_n = f_{n-1} + f_{n-2} \quad (6)$$

The numbers can be easily calculated and printed with a small Java program.

However, there are two different ways to deal with the calculated numbers:

- **Store all** the numbers in an array.
- Only store the most recent **two** numbers.

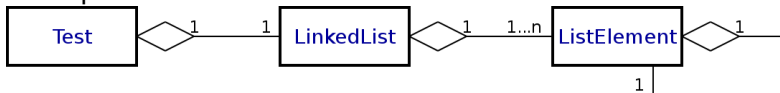
Outline

- 1 Know How
 - Fibonacci Numbers
 - **Linked List**
 - Bubble Sort
- 2 Prediscussion Assignment 5
 - Fibonacci Implementation
 - Linked List Implementation
- 3 Postdiscussion Assignment 4
 - GrowingArray
 - Test & Comparison
 - Stack

Class Diagram (UML)

Aggregation means a **has-a-relation** (“uses”, “aggregates”, “contains”).

Example:



Aggregation is printed as a line with a diamond shaped ending. The **diamond** symbolizes the **object** of the class on the other end of the line.

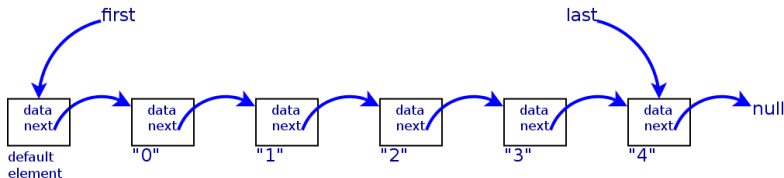
Test has a LinkedList.

LinkedList has multiple ListElements.

Each ListElement has a ListElement.

Chaining Objects

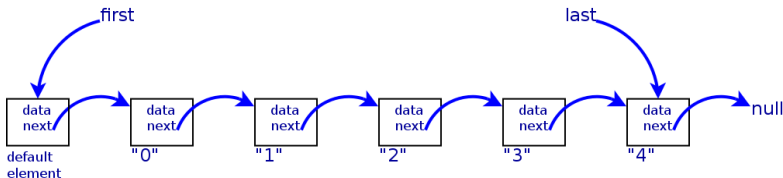
Since an object is stored on the heap using a **reference**, the “*self-aggregation*” of `ListElement` allows to **chain** objects of this class. The **linking** of the chain is therefore achieved by the references.



Such a chain is what we call a **linked list**.

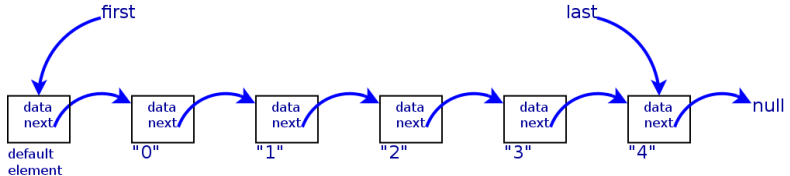
Chaining Objects

Since the links can be rewired while the objects stay at the same spot, a linked list is very flexible. It allows seamless **insertion** and **removal** of random objects.



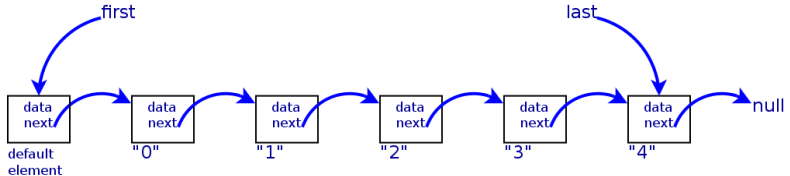
The “default element” is created by the constructor of the class `LinkedList`. It simplifies the implementation, but stays hidden in the class.

Access along the Chain



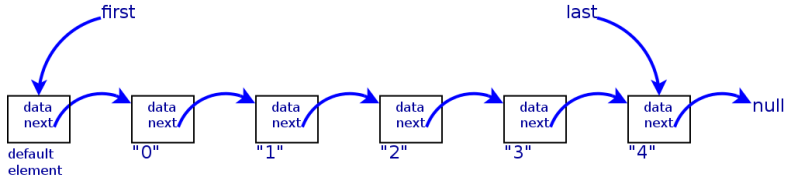
- `first.next` →

Access along the Chain



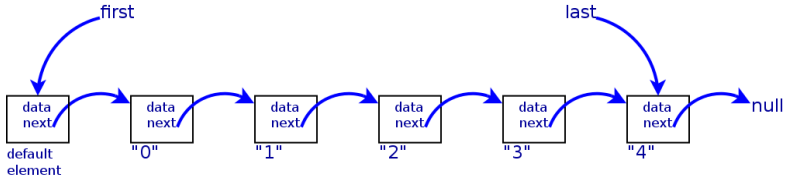
- `first.next` → element "0"
- `first.next.next` →

Access along the Chain



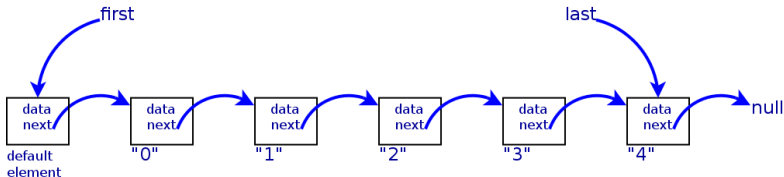
- `first.next` → element "0"
- `first.next.next` → element "1"
- `first.next.next.next` →

Access along the Chain



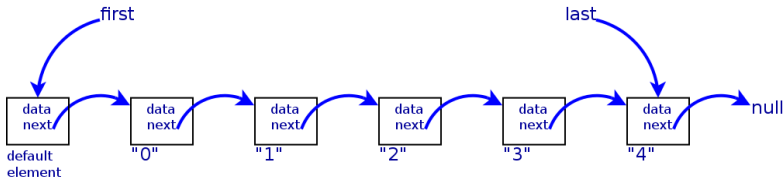
- `first.next` → element "0"
- `first.next.next` → element "1"
- `first.next.next.next` → element "2"
- `first.next.next.next.next` →

Access along the Chain



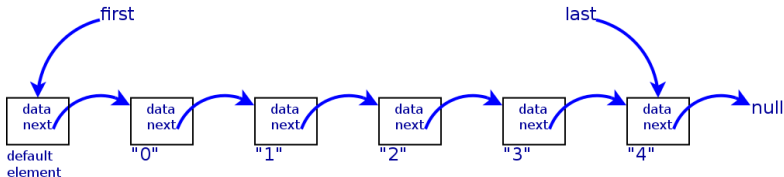
- `first.next` → element "0"
- `first.next.next` → element "1"
- `first.next.next.next` → element "2"
- `first.next.next.next.next` → element "3"
- `first.next.next.next.next.next` →

Access along the Chain



- `first.next` → element "0"
- `first.next.next` → element "1"
- `first.next.next.next` → element "2"
- `first.next.next.next.next` → element "3"
- `first.next.next.next.next.next` → element "4"
- `first.next.next.next.next.next.next` →

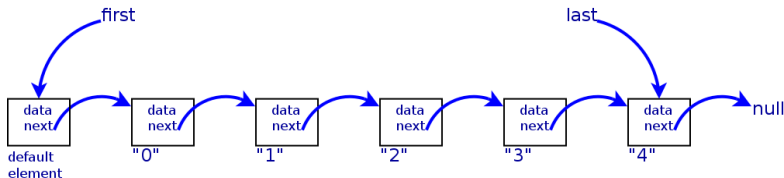
Access along the Chain



- `first.next` → element "0"
- `first.next.next` → element "1"
- `first.next.next.next` → element "2"
- `first.next.next.next.next` → element "3"
- `first.next.next.next.next.next` → element "4"
- `first.next.next.next.next.next.next` →

null pointer exception

Linked List Enumeration

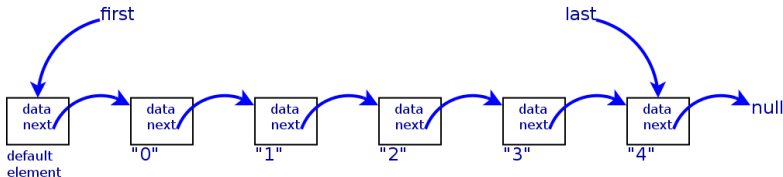


```
1 ListElement le = first; // starting point
2 while (le.next != null)
3 {   le = le.next; // next step
4 }
```

Due to the while condition, no null pointer exception can occur. But the default element (first) must always exist.

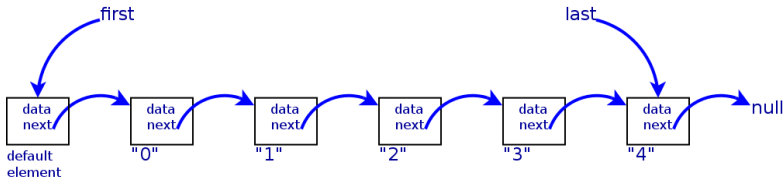
Adding and Insertion

- If we **add** another element to the list in the image, the reference “next” of element 4 would point to this new element (index 5). The reference next of element 5 is then null.



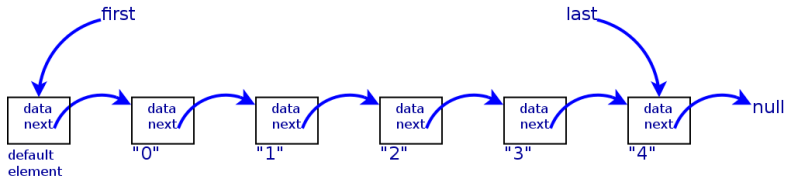
Adding and Insertion

- If we **add** another element to the list in the image, the reference “next” of element 4 would point to this new element (index 5). The reference next of element 5 is then null.
- If an element is **inserted** between elements 1 and 2, the new element is getting linked (*referenced*) by next of 1. Itself it links to the old element 2, which now is number 3.



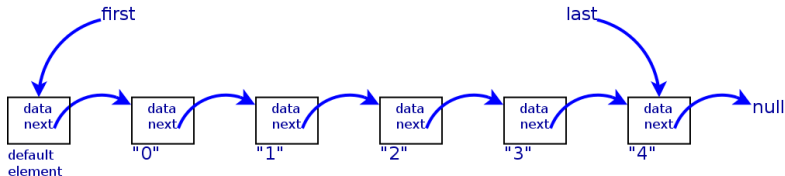
Removal

- Removing element 0 makes the default element from now on link to the element which was 2 before. Having a default element simplifies the implementation. Without it, removal and insertion must treat the case different at the beginning of the list.



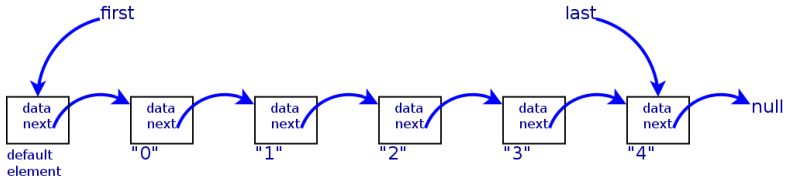
Removal

- Removing element 0 makes the default element from now on link to the element which was 2 before. Having a default element simplifies the implementation. Without it, removal and insertion must treat the case different at the beginning of the list.
- Once an object isn't referenced anymore, it will be deleted automatically. *This is done by a background task of the VM called Garbage Collector.*



Index Numbers

- The index numbers aren't assigned to the elements. *To determine the index, we actually have to count the elements.*

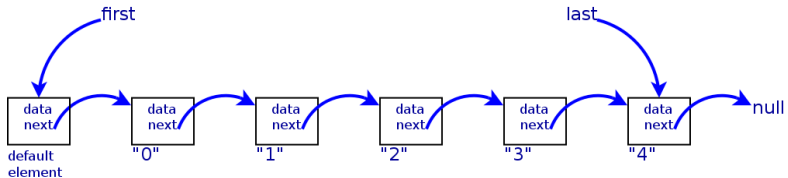


Random Access

- The index numbers aren't assigned to the elements. *To determine the index, we actually have to **count** the elements.*

Question

What do you expect, if you compare the random access speed between an array and a linked list?

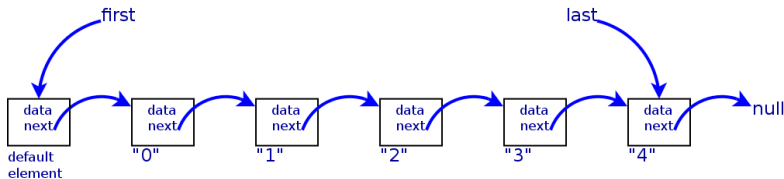


Random Access

Answer

Access by index is **very expensive** in linked lists. Reading and writing an array at random indices is much faster than in a linked list.

There we are: No perfect solution at hand. What structure we choose, heavily depends on the **expected usage**.



Outline

1 Know How

- Fibonacci Numbers
- Linked List
- **Bubble Sort**

2 Prediscussion Assignment 5

- Fibonacci Implementation
- Linked List Implementation

3 Postdiscussion Assignment 4

- GrowingArray
- Test & Comparison
- Stack

Bubble Sort, General Steps

- 1 We want a **simple** algorithm to **sort** the elements of a linked list.

Bubble Sort, General Steps

- ❶ We want a **simple** algorithm to **sort** the elements of a linked list.
- ❷ We always only look at **two elements** at once.

Bubble Sort, General Steps

- ❶ We want a **simple** algorithm to **sort** the elements of a linked list.
- ❷ We always only look at **two elements** at once.
- ❸ **Step by step** we go through the list.

Bubble Sort, General Steps

- ❶ We want a **simple** algorithm to **sort** the elements of a linked list.
- ❷ We always only look at **two elements** at once.
- ❸ **Step by step** we go through the list.
- ❹ If two elements are in the **wrong order**, we **swap** them.

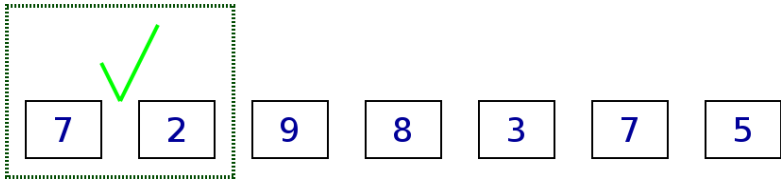
Bubble Sort, General Steps

- 1 We want a **simple** algorithm to **sort** the elements of a linked list.
- 2 We always only look at **two elements** at once.
- 3 **Step by step** we go through the list.
- 4 If two elements are in the **wrong order**, we **swap** them.
- 5 We go through the list **multiple** times.

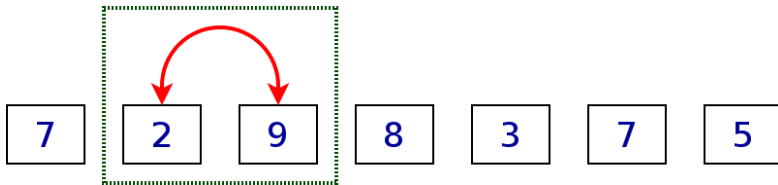
Bubble Sort, General Steps

- 1 We want a **simple** algorithm to **sort** the elements of a linked list.
- 2 We always only look at **two elements** at once.
- 3 **Step by step** we go through the list.
- 4 If two elements are in the **wrong order**, we **swap** them.
- 5 We go through the list **multiple** times.
- 6 We're only **done**, when a full go through the list results in **no** swaps!

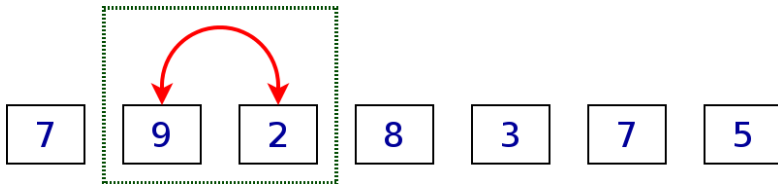
Bubble Sort, Example



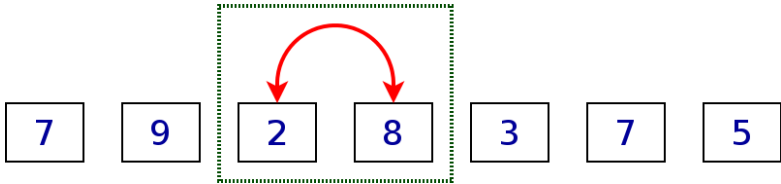
Bubble Sort, Example



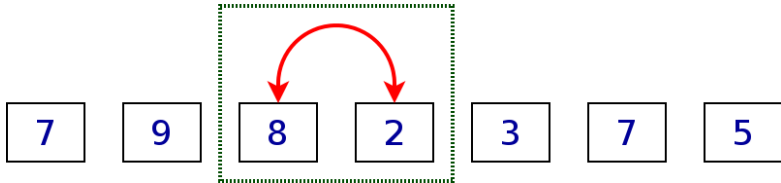
Bubble Sort, Example



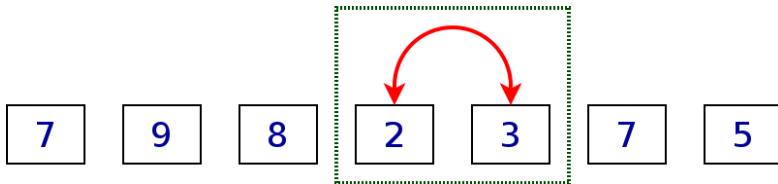
Bubble Sort, Example



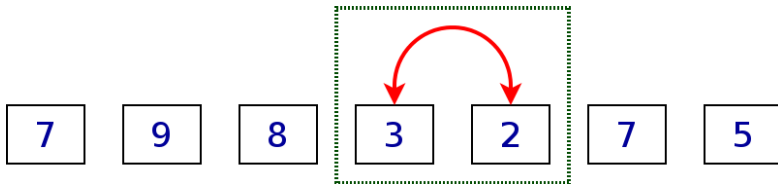
Bubble Sort, Example



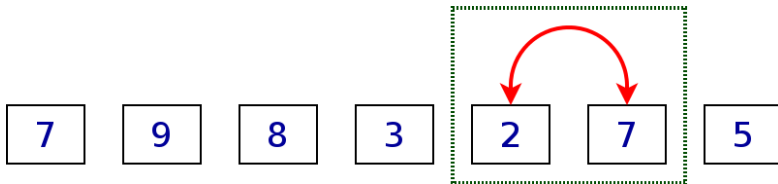
Bubble Sort, Example



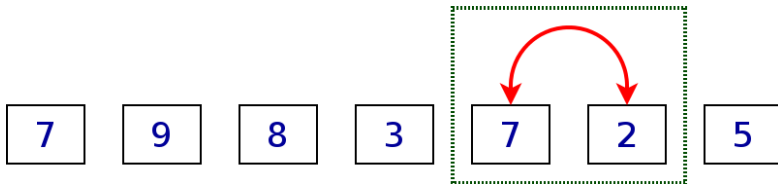
Bubble Sort, Example



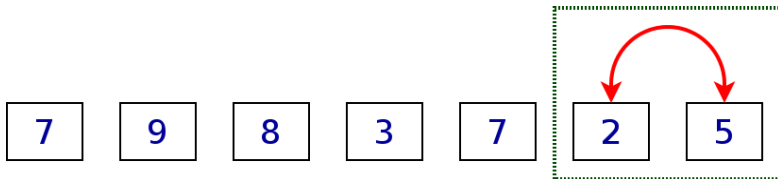
Bubble Sort, Example



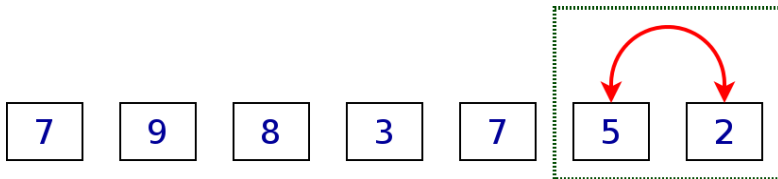
Bubble Sort, Example



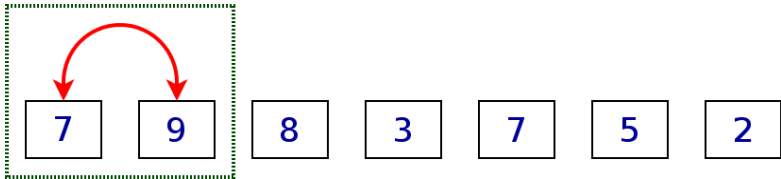
Bubble Sort, Example



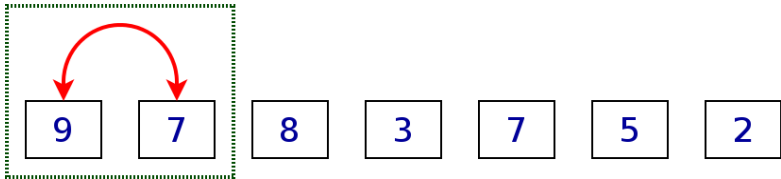
Bubble Sort, Example



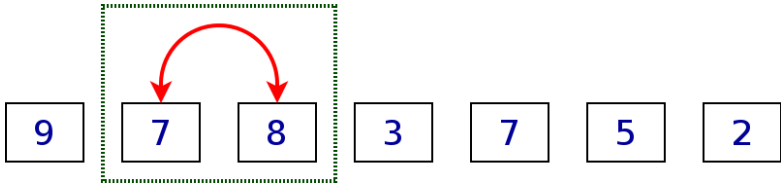
Bubble Sort, Example



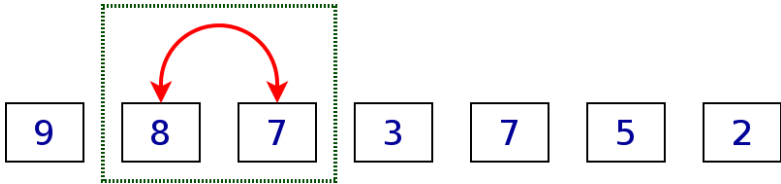
Bubble Sort, Example



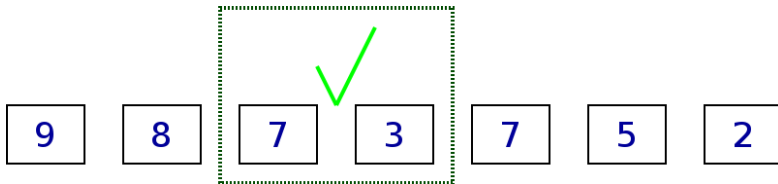
Bubble Sort, Example



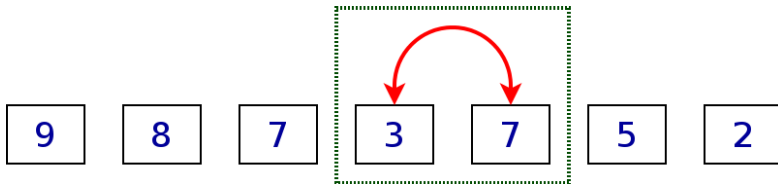
Bubble Sort, Example



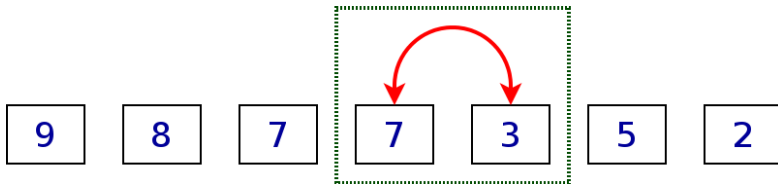
Bubble Sort, Example



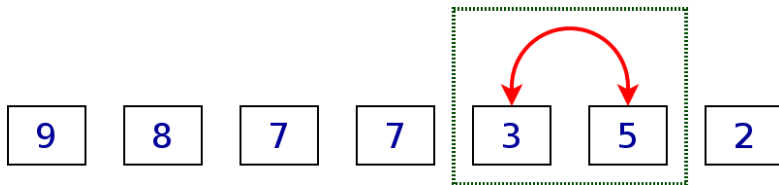
Bubble Sort, Example



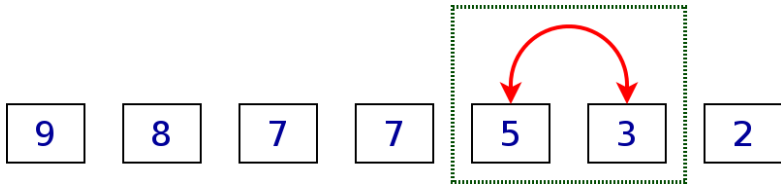
Bubble Sort, Example



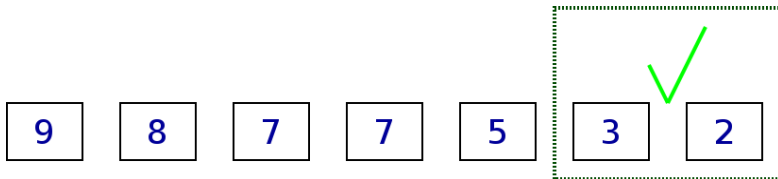
Bubble Sort, Example



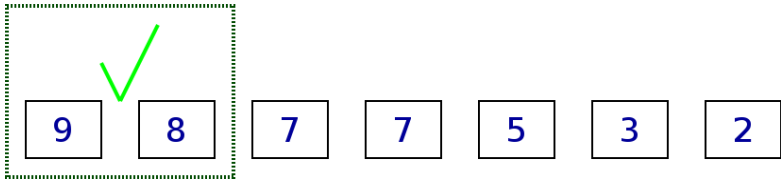
Bubble Sort, Example



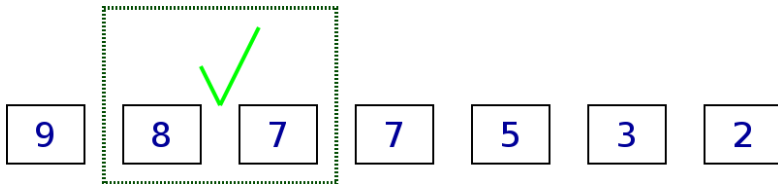
Bubble Sort, Example



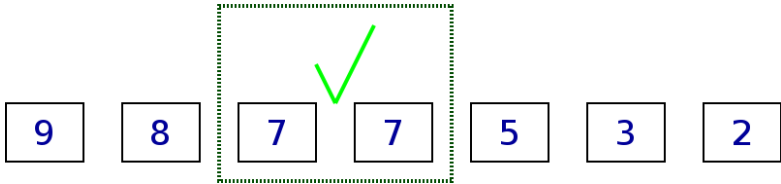
Bubble Sort, Example



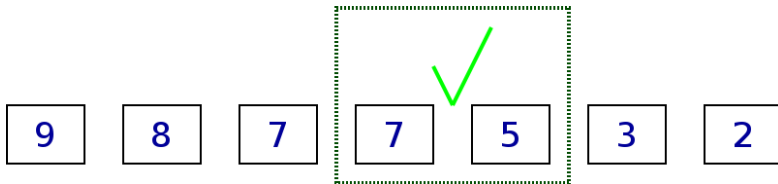
Bubble Sort, Example



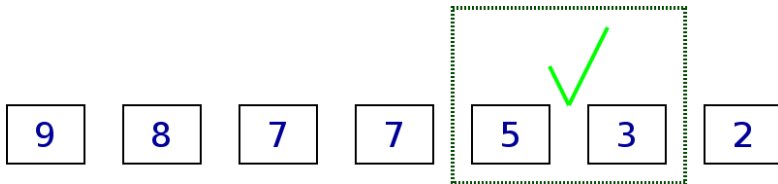
Bubble Sort, Example



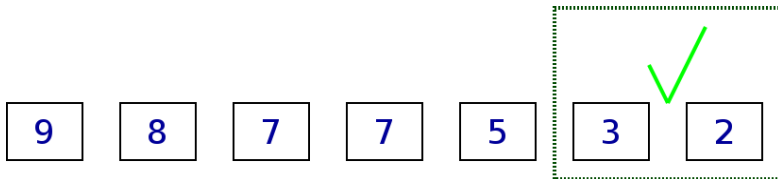
Bubble Sort, Example



Bubble Sort, Example



Bubble Sort, Example



Bubble Sort, Example



Outline

- 1 Know How
 - Fibonacci Numbers
 - Linked List
 - Bubble Sort
- 2 Prediscussion Assignment 5
 - **Fibonacci Implementation**
 - Linked List Implementation
- 3 Postdiscussion Assignment 4
 - GrowingArray
 - Test & Comparison
 - Stack

Fibonacci Code Base

You may use the following code base:

```
1 public class PrintFibonacci
2 {   public static void main(String[] args)
3     {   // TODO: Calculate and print
4         // the 25 first Fibonacci-numbers
5         int x = 1; // example code
6         int y = 1; // example code
7         for (int i=0; i<25; ++i)
8         {   int z = // ...
9             // ...
10        }
11    }
12 }
```

Outline

- 1 Know How
 - Fibonacci Numbers
 - Linked List
 - Bubble Sort
- 2 Prediscussion Assignment 5
 - Fibonacci Implementation
 - **Linked List Implementation**
- 3 Postdiscussion Assignment 4
 - GrowingArray
 - Test & Comparison
 - Stack

ListElement, Class Structure

```
1 public class ListElement
2 {   public int data; // the actual data
3     public ListElement next; // next reference
4
5     // ...
6 }
```

ListElement, Constructor & toString

```
1 public ListElement(int dat) // Constructor
2 {   data = dat; // given parameter value
3     next = null; // default
4 }
5
6 public ListElement(int dat, ListElement after)
7 {   data = dat; // given parameter value
8     next = after; // given parameter value
9 }
10
11 public String toString() // for easy printing
12 {   return "_" + data + "_"; // underscore delimiters
13 } // java translates the int into a String
```

LinkedList, Class Structure & Constructor

```
1 public class LinkedList
2 {   private ListElement first; // default element
3     private ListElement last; // last element
4
5     public LinkedList() // Constructor
6     {   first = new ListElement(0, null); // default
7         last = first; // indicates an empty list
8     }
9
10    // ...
11 }
```

LinkedList, Overwriting toString

```
1 public String toString() // readable representation
2 {   ListElement le = first; // starting point
3     String str = "";
4     // TODO: Add code here
5     // Go through list + sum the Strings
6     return str;
7 }
```

The return value of `toString()` is what is printed if we do `System.out.println(leXYZ)` when `leXYZ` is an object of class `ListElement`.

LinkedList, Methods isEmpty & purge

```
1 public boolean isEmpty() // true if list is empty
2 {   return (first == last); // only default element
3 }
4
5 public void purge() // erases the whole content
6 {   last = first; // reset last
7     first.next = null; // remove reference
8 }
```

Purge just unlinks the whole list except the default element.

LinkedList, Method length

```
1 public int length() // number of elements
2 {   ListElement le = first; // starting point
3     int size = 0;
4     while (le.next != null)
5     {   le = le.next; // go through list
6         ++size; // count elements
7     }
8     return size; // return result
9 }
```

LinkedList, Method add

```
1 public void add(int n) // add at end of the list
2 {   ListElement le = new ListElement(n, null);
3     last.next = le; // add it to the list
4     last = le; // fix last reference
5 }
```

Add is simple, since we always keep a pointer to the last element. If add would be a seldom operation we could omit it.

LinkedList, Helper Method locate

```
1 private ListElement locate(int pos) // helper
2 {   ListElement le = first; // starting point
3     int i = 0;
4     while (i <= pos && le.next != null) // search
5     {   le = le.next; // go through the list
6         ++i; // count elements to determine index
7     }
8     return le;
9 }
10
11 public int get(int pos) // using helper locate
12 {   return locate(pos).data; // return data
13 }
```

LinkedList, Method insert

```
1 public void insert(int pos, int dat) // given index
2 {   ListElement le = new ListElement(dat); // new
3     // TODO: Add code here ...
4     // locate element before the new element
5     // get the element after the new element
6     // wire to the new element
7     // wire from the new element
8     // if at the end, update last
9 }
```

LinkedList, Method delete

```
1 public void delete(int pos) // delete element
2 {   if (!isEmpty()) // don't delete in empty list
3     {   // TODO: Add code here
4         // find element before the removed one
5         // get element after the removed one
6         // link before and after
7     }
8 }
```

Make sure to reconnect the list properly after removing the element.

LinkedList, Creation and Adding

```
1 public class Test
2 {   public static void main(String[] args) {
3     LinkedList sll = new LinkedList();
4     for (int i=1; i<8; ++i) // 1 to 7
5     {   sll.add(i); // add value of i
6     }
7     // ...
8 }
9 }
```

At first we create an object of `LinkedList` and add 7 numbers to it.

LinkedList, Further Tests

```
1 System.out.println(sll); // print the list
2 sll.insert(3,99); // insert 99 at index 3
3 System.out.println(sll); // uses toString()
4 System.out.println(sll.get(3)); // print "3"
5 sll.delete(3); // delete the element "3"
6 System.out.println(sll);
7 System.out.println(sll.length()); // print length
8 sll.purge(); // erase the list
```

Outline

- 1 Know How
 - Fibonacci Numbers
 - Linked List
 - Bubble Sort
- 2 Prediscussion Assignment 5
 - Fibonacci Implementation
 - Linked List Implementation
- 3 Postdiscussion Assignment 4
 - **GrowingArray**
 - Test & Comparison
 - Stack

GrowingArray, Class Structure

```
1 public class GrowingArray
2 {   private int[] data;
3
4     public GrowingArray() // Constructor
5     {   data = new int[10]; // default size
6     }
7
8     // ...
9 }
```

The **default size** is an important design choice. If we make it too big it's a waste of storage. If we make it too small too many resizes will result.

GrowingArray, Helper Method `resize`

```
1  private void resize(int minLength) // helper
2  {    int[] tmp = data; // store old reference
3      data = new int[minLength*2]; // new object
4      for (int i=0; i<tmp.length; ++i)
5          {    data[i] = tmp[i]; // copy elements
6              }
7  }
```

Since I decided to `resize` in `get` and in `set`, I've put the code for it in a helper method. This way the code doesn't get duplicated. It's always good to avoid code duplication.

GrowingArray, Method get

```
1 public int get(int ind)
2 {     if (ind >= data.length)
3         {     resize(ind); // resize if too small
4         }
5     return data[ind];
6 }
```

Simple, isn't it?

GrowingArray, Method set

```
1 public void set(int ind, int val)
2 {   if (ind >= data.length)
3     {   resize(ind); // resize if too small
4     }
5     data[ind] = val;
6 }
```

Abstraction

The methods `get` & `set` implement an **abstraction layer**. This layer hides the data structure below. That's what enabled us to implement a growing array.

```
1 public void set(int ind, int val) { ... }  
2 public int get(int ind) { ... }
```

But below the layer can be a normal array, a growing array, a linked list, whatever you want.

Abstraction Robot

A rather crazy example for **abstraction** is a **robot** to deal with dangerous objects. The **goggles** and the **joystick** are somehow similar to our get- & set-methods, they give access to something which is separated by some **layer of abstraction**.



Abstraction Word Processor

An example more closely related to the assignment is a **word processor** like MicrosoftTM Word. We may use it to create, load, modify and store documents. But most of us don't have a clue, how Word is saving our data on the harddisk.

We even don't have to know about it in order to use it. That's the **beauty** of **abstraction**.

Outline

- 1 Know How
 - Fibonacci Numbers
 - Linked List
 - Bubble Sort
- 2 Prediscussion Assignment 5
 - Fibonacci Implementation
 - Linked List Implementation
- 3 Postdiscussion Assignment 4
 - GrowingArray
 - Test & Comparison
 - Stack

Comparing Lists & Arrays

```
1 final int MAX = 12345678;
2 long time, duration;
3
4 int[] arr = new int[MAX];
5 GrowingArray ga = new GrowingArray();
6 ArrayList<Integer> al = new ArrayList<Integer>();
7 Vector<Integer> vec = new Vector<Integer>();
8 LinkedList<Integer> ll = new LinkedList<Integer>();
```

Testing Lists e. g. Vector

```
1 time = System.currentTimeMillis();  
2 for (int i=0; i<MAX; ++i)  
3 {   vec.add(i);  
4 }  
5 duration = System.currentTimeMillis() - time;  
6 System.out.println("Vector: " + duration);
```

We measure the number of milliseconds while we add 12345678 numbers to the list. This can be done for every data structure from the previous slide.

Comparison

Question

- What's the slowest Implementation?
- What's the fastest?
- Why?

Comparison Results

Comparison

- LinkedList: 4888.8ms

Comparison Results

Comparison

- LinkedList: 4888.8ms
- Vector: 1361.4ms

Comparison Results

Comparison

- LinkedList: 4888.8ms
- Vector: 1361.4ms
- ArrayList: 915.8ms

Comparison Results

Comparison

- LinkedList: 4888.8ms
- Vector: 1361.4ms
- ArrayList: 915.8ms
- GrowingArray: 377.6ms

Comparison Results

Comparison

- LinkedList: 4888.8ms
- Vector: 1361.4ms
- ArrayList: 915.8ms
- GrowingArray: 377.6ms
- Array: 14.8ms

Outline

- 1 Know How
 - Fibonacci Numbers
 - Linked List
 - Bubble Sort
- 2 Prediscussion Assignment 5
 - Fibonacci Implementation
 - Linked List Implementation
- 3 Postdiscussion Assignment 4
 - GrowingArray
 - Test & Comparison
 - Stack

Stack class Structure

```
1 public class ArrayStack
2 {   private GrowingArray ga;
3     private int top;
4
5     public ArrayStack()
6     {   ga = new GrowingArray();
7         top = 0;
8     }
9
10    // ...
11 }
```

Stack push

```
1 public void push(int v)
2 {   ++top;
3     ga.set(top, v);
4 }
```

It's important to increment `top` before we set the element.

Stack get

```
1 public int pop()
2 {   if (top > 0) // if stack not empty
3     {   int t = ga.get(top);
4         --top;
5         return t;
6     }
7     return -1; // error , pop empty stack
8 }
```

It's important to decrement `top` after we read the element. We therefore use a temporary variable to store the result before decrementing `top`. If we would put `--top;` after the `return` statement, it wouldn't be executed (*dead code*).

Questions from your side?

Please

- Feedback?
- Additions?
- Remarks?
- Wishes?
- ...



All the Best!

