ETH zürich

Assignment 5

Felix Friedrich, Lars Widmer, Fabian Stutz
TA lecture, Informatics II D-BAUG

March 18, 2014

ETH zürich

“Präsenzstunden” Today

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

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

ETH zürich

Outline

ETH zürich

Fibonacci Numbers

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

$$f_1 = f_2 = 1 \quad (1)$$
$$f_n = f_{n-1} + f_{n-2} \quad (2)$$

ETH zürich

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

ETH zürich

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.

ETH zürich

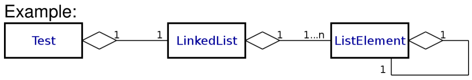
Outline

ETH zürich

Class Diagram (UML)

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

Example:

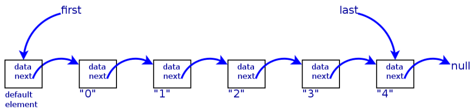


Aggregation is printed as a line with a diamond shaped ending. The **diamond** symbolizes the **object** of the class on the other end of the line.

`Test` has a `LinkedList`.
`LinkedList` has multiple `ListElement`s.
Each `ListElement` has a `ListElement`.

Chaining Objects

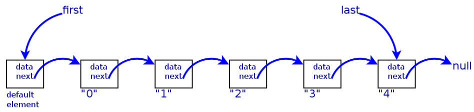
Since an object is stored on the heap using a **reference**, the "*self-aggregation*" of `ListElement` allows to **chain** objects of this class. The **linking** of the chain is therefore achieved by the references.



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

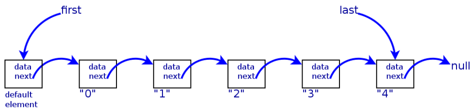
Chaining Objects

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.

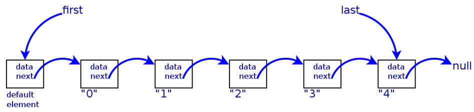
Access along the Chain



- first.next → 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

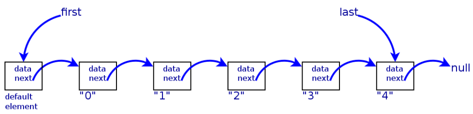



```
ListElement le = first; // starting point
while (le.next != null)
{    le = le.next; // next step
}
```

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

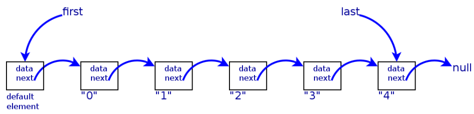
Adding and Insertion

- 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.
- If an element is **inserted** between elements 1 and 2, the new element is getting linked *(referenced)* by `next` of 1. Itself it linkes to the old element 2, which now is number 3.



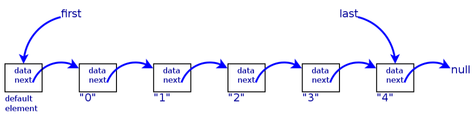
Removal

- 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.
- 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.*

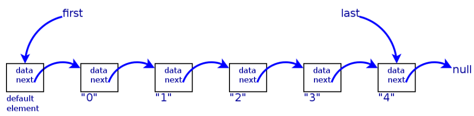


Random Access

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

Question

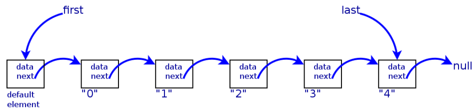
What do you expect, if you compare the random access speed between an array and a linked list?



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**.



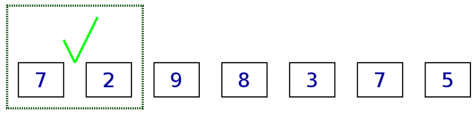
Outline

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

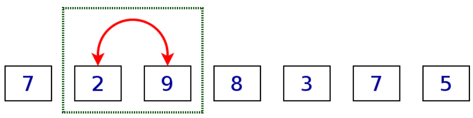
Bubble Sort, General Steps

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

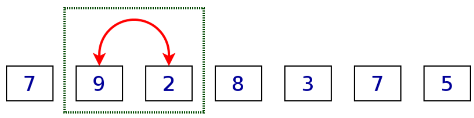
Bubble Sort, Example



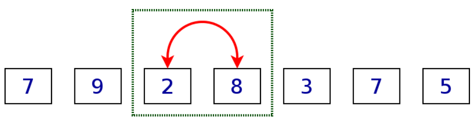
Bubble Sort, Example



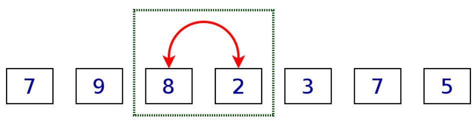
Bubble Sort, Example



Bubble Sort, Example



Bubble Sort, Example



Bubble Sort, Example

7 9 8 2 3 7 5

Bubble Sort, Example

7 9 8 3 2 7 5

Bubble Sort, Example

7 9 8 3 2 7 5

Bubble Sort, Example

7 9 8 3 7 2 5

Bubble Sort, Example

7 9 8 3 7 2 5

Bubble Sort, Example

7 9 8 3 7 5 2

Bubble Sort, Example

7 9 8 3 7 5 2

Bubble Sort, Example

9 7 8 3 7 5 2

Bubble Sort, Example

9	7	8	3	7	5	2

Bubble Sort, Example

9	8	7	3	7	5	2

Bubble Sort, Example

9	8	7	3	7	5	2

Bubble Sort, Example

9	8	7	3	7	5	2

Bubble Sort, Example

9	8	7	7	3	5	2

Bubble Sort, Example

9	8	7	7	3	5	2

Bubble Sort, Example

9	8	7	7	5	3	2

Bubble Sort, Example

9	8	7	7	5	3	2

Bubble Sort, Example

9 8 7 7 5 3 2

Bubble Sort, Example

9 8 7 7 5 3 2

Bubble Sort, Example

9 8 7 7 5 3 2

Bubble Sort, Example

9 8 7 7 5 3 2

Bubble Sort, Example

9 8 7 7 5 3 2

Bubble Sort, Example

9 8 7 7 5 3 2

Bubble Sort, Example

9 8 7 7 5 3 2

Outline

Fibonacci Code Base

You may use the following code base:

```java
public class PrintFibonacci
{   public static void main(String[] args)
    {   // TODO: Calculate and print
        // the 25 first Fibonacci-numbers
        int x = 1; // example code
        int y = 1; // example code
        for (int i=0; i<25; ++i)
        {   int z = // ...
            // ...
        }
    }
}
```


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

```java
public class ListElement
{   public int data; // the actual data
    public ListElement next; // next reference

    // ...
}
```


`ListElement`, Constructor & `toString`

```java
public ListElement(int dat) // Constructor
{   data = dat; // given parameter value
    next = null; // default
}

public ListElement(int dat, ListElement after)
{   data = dat; // given parameter value
    next = after; // given parameter value
}

public String toString() // for easy printing
{   return "_"+data+"_"; // underscore delimiters
}   // java translates the int into a String
```


`LinkedList`, Class Structure & Constructor

```java
public class LinkedList
{   private ListElement first; // default element
    private ListElement last; // last element

    public LinkedList() // Constructor
    {   first = new ListElement(0,null); // default
        last = first; // indicates an empty list
    }

    // ...
}
```


`LinkedList`, Overwriting `toString`

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

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`

```java
public boolean isEmpty() // true if list is empty
{   return (first == last); // only default element
}

public void purge() // erases the whole content
{   last = first; // reset last
    first.next = null; // remove reference
}
```

Purge just unlinkes the whole list except the default element.

`LinkedList`, Method `length`

```java
public int length() // number of elements
{   ListElement le = first; // starting point
    int size = 0;
    while (le.next != null)
    {   le = le.next; // go through list
        ++size; // count elements
    }
    return size; // return result
}
```

`LinkedList`, Method `add`

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

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

`LinkedList`, Helper Method `locate`

```java
private ListElement locate(int pos) // helper
{   ListElement le = first; // starting point
    int i = 0;
    while (i <= pos && le.next != null) // search
    {   le = le.next; // go through the list
        ++i; // count elements to determine index
    }
    return le;
}

public int get(int pos) // using helper locate
{   return locate(pos).data; // return data
}
```


`LinkedList`, Method `insert`

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


`LinkedList`, Method `delete`

```java
public void delete(int pos) // delete element
{   if (!isEmpty()) // don't delete in empty list
    {   // TODO: Add code here
        // find element before the removed one
        // get element after the removed one
        // link before and after
    }
}
```

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

`LinkedList`, Creation and Adding

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

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

`LinkedList`, Further Tests

```java
System.out.println(sll); // print the list
sll.insert(3,99); // insert 99 at index 3
System.out.println(sll); // uses toString()
System.out.println(sll.get(3)); // print "3"
sll.delete(3); // delete the element "3"
System.out.println(sll);
System.out.println(sll.length()); // print length
sll.purge(); // erase the list
```


Outline

`GrowingArray`, Class Structure

```java
public class GrowingArray
{   private int[] data;

    public GrowingArray() // Constructor
    {   data = new int[10]; // default size
    }

    // ...
}
```

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`

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

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`

```java
public int get(int ind)
{   if (ind >= data.length)
    {   resize(ind); // resize if too small
    }
    return data[ind];
}
```

Simple, isn't it?

`GrowingArray`, Method `set`

```java
public void set(int ind, int val)
{   if (ind >= data.length)
    {   resize(ind); // resize if too small
    }
    data[ind] = val;
}
```


Abstraction

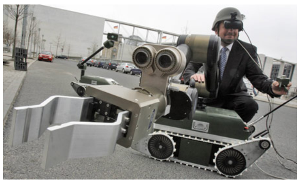
The methods `get` & `set` implement an **abstraction layer**. This layer hides the data structure below. That's what enabled us to implement a growing array.

```java
public void set(int ind, int val) { ... }
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 seperated by some **layer of abstraction**.



Abstraction Word Processor

An example more closely related to the assignment is a **word processor** like Microsoft™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

Comparing Lists & Arrays

```java
final int MAX = 12345678;
long time, duration;

int[] arr = new int[MAX];
GrowingArray ga = new GrowingArray();
ArrayList<Integer> al = new ArrayList<Integer>();
Vector<Integer> vec = new Vector<Integer>();
LinkedList<Integer> ll = new LinkedList<Integer>();
```

Testing Lists e. g. `Vector`

```java
time = System.currentTimeMillis();
for (int i=0; i<MAX; ++i)
{   vec.add(i);
}
duration = System.currentTimeMillis() - time;
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
- Vector: 1361.4ms
- ArrayList: 915.8ms
- GrowingArray: 377.6ms
- Array: $14.8ms$

Outline

Stack `class` Structure

```java
public class ArrayStack
{   private GrowingArray ga;
    private int top;

    public ArrayStack()
    {   ga = new GrowingArray();
        top = 0;
    }

    // ...
}
```


Stack `push`

```java
public void push(int v)
{   ++top;
    ga.set(top,v);
}
```

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

Stack `get`

```java
public int pop()
{   if (top > 0) // if stack not empty
    {   int t = ga.get(top);
        --top;
        return t;
    }
    return -1; // error, pop empty stack
}
```

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!