ITI 1121. Introduction to Computing II

Stack: linked elements

by

Marcel Turcotte

Version February 9, 2020

Preamble

Preamble

Overview

Stack: linked elements

We implement a stack using linked elements.

General objective:

- This week you will be able to implement a stack using linked elements.

Preamble

Learning objectives

Learning objectives

- **Implement** a stack using linked elements.
- **Compare** the implementations using arrays and linked elements of a stack.

Readings:

- Pages 75-83, 157-159 of E. Koffman and P. Wolfgang.

Preamble

Plan

Plan

1. Preamble
2. Implementation using linked elements
3. Prologue

Implementation using linked elements

Implementation of a stack using linked elements

Implementation using linked elements

Reminder

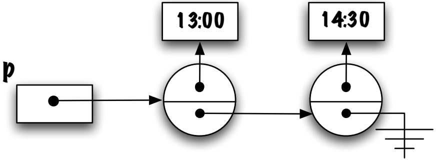
On the implementation of a stack using an array

- **Access to the elements** of an array is **very fast**, it always requires a constant number of operations.
- However, since arrays have a **fixed size**, there are some applications for which they are not appropriate.
- A frequently used technique to get around this limitation is to copy the elements of the array into a new, larger array and replace the old one with the new one (**dynamic arrays**).
- On the other hand, this makes the insertions **more expensive** (compared to the execution time because you have to copy all the elements of the old array to the new one) and memory usage is increased because the **physical** size of the data structure will generally be larger than its **logical** size.

Motivation

Linked structures

Let's now consider an implementation always using **an amount of memory proportional to the number of elements** contained in the structure.



- These structures are efficient, in terms of execution time (for some operations), because **they avoid copying elements**.
- The structures considered here are **linear**, i.e. each element has a predecessor and a successor (except for the first and last element).
- Unlike array-based data structures, the elements in those structures are **not contiguous in memory**.

Experimentation

The class Elem

Consider the following declaration *:

```java
class Elem<E> {
   E value;
   Elem<E> next;
}
```

- What's so special about the definition of **Elem**?
- The instance variable **next** is a reference to an object of the class **Elem**.
- Is it valid?
- Try it for yourself!

  ```
  > javac Elem.java
  ```
- Yes, it's valid, although it does seem circular.

*The issue of the visibility of variables will be addressed shortly..

What's that for?

- Declaring a variable of type **Elem**:

```
Elem<Time> p;
```

- Create an object of the class **Elem**:

```
new Elem<Time>();
```

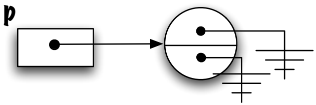
What's that for?

- Save the reference in the variable **p**..

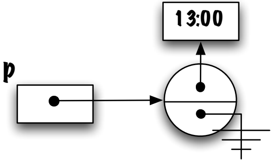
```
Elem<Time> p;
p = new Elem<Time>();
```

Notation: I will always use circles to represent the objects of the class **Elem**. The top part represents the instance variable **value** while the bottom part represents the variable **next**.

What's the point?



- How do we change the content of the instance variable **value** of the newly created object?

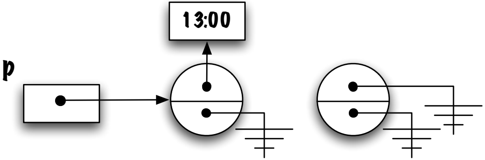


- We use the **dot-notation** in order to access the attributes of the object.

What’s that for?

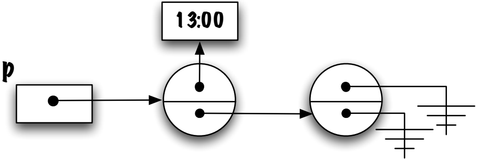
- Create a new object of the class **Elem**.

  ```
  new Elem<Time>();
  ```



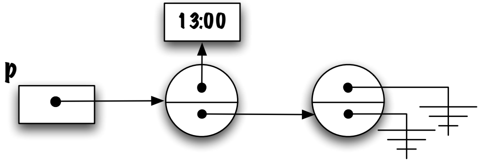
- How do you **link** the elements together?

What’s that for?



- The variable **next** of the object designated by the reference variable **p** receives the reference of the newly created object **Elem**.

What’s that for?

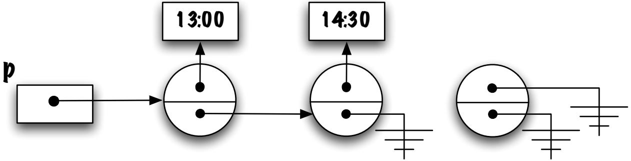


- Change the contents of the variable **value** of the newly created object.

What's that for?

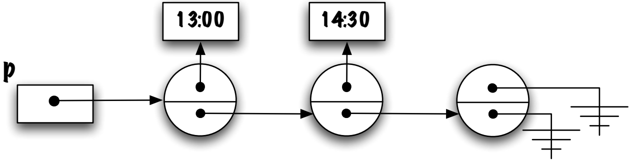
- Create a new object of the class **Elem**.

```
new Elem<Time>();
```



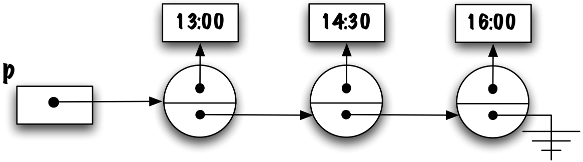
- How do we **link** this element to the others?

What’s that for?



- Change the content of the variable **value** of the newly created object.

What's that for?



```
p.next.next = p;
```

- What does the above statement do?

What’s that for?

```
p.next.next = p;
```

- A **circular** structure has been created!
- The last item is **no longer accessible**;
- It’ll be picked up by the garbage collector (**System.gc()**).

⇒ This is the basis of the linked structures: **informations** (values) are linked to each other by **links** (references).

Linked structures

Linked strutures

```
class Elem<E> {
    E value;
    Elem<E> next;
}
```

Linked data structures, such as this one, allow us:

- to represent **linear** data structures, such as stacks, queues and lists;
- they always use a **quantity of memory proportional to the number of elements**;
- all this is made possible because the class declares an instance variable whose type is a reference to an object of the same class.

⇒ When the structures are linear like these, we talk about **(singly) linked lists**.

Summary

- **Linked structures** are an alternative to **arrays** for saving values.
- They always use a **quantity of memory proportional to the number of elements saved** since each element is saved in its container, an object of the class **Elem**. Each container is linked to the next one by a reference variable.
- For now, we limit ourselves to **linear structures**, but **graphs** or **trees** are also possible.

Constructor

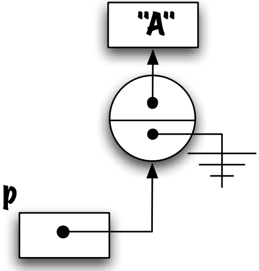
Constructor

This is the usual constructor of the class **Elem**:

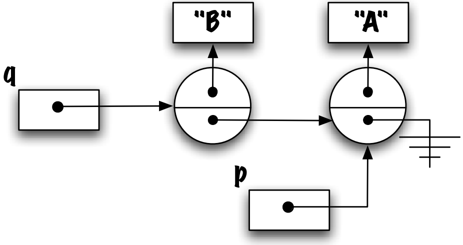
```
public class Elem<E> {

  E value;
  Elem<E> next;

  Elem(E value, Elem<E> next) {
      this.value = value;
      this.next = next;
  }
}
```

and the usual **usage**,

```
p = new Elem<String>("A", null);
```



```
q = new Elem<String>("B", p);
```



Implementing the Stack interface

Implementing a Stack using linked elements

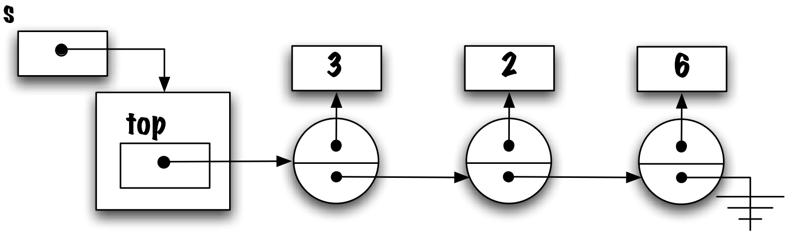
```
public class LinkedStack<E> implements Stack<E> {

    public boolean empty() {

    }
    public void push(E o) {

    }
    public E peek() {

    }
    public E pop() {

    }
}
```

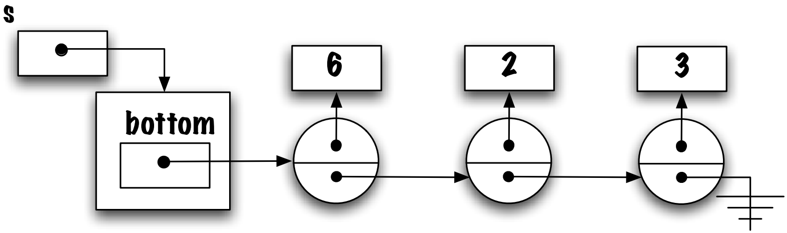
- What are the **instance variables**?

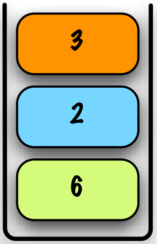
Instance variables

What are the instance variables?

- Which of the following two strategies is **preferable**?







Discussion

Nested class

Class Elem and encapsulation principle

The **visibility of the instance variables** is not acceptable. It is a violation of the principle of encapsulation.

- What options do we have?

```java
public class Elem<E> {

  private E value;
  private Elem<E> next;

  public Elem(E value, Elem<E> next) {
      this.value = value;
      this.next = next;
  }
  public void setValue(E value) {
      this.value = value;
  }
  public void setNext(Elem<E> next) {
      this.next = next;
  }
  public E getValue() {
      return value;
  }
  public Elem<E> getNext() {
      return next;
  }
}
```

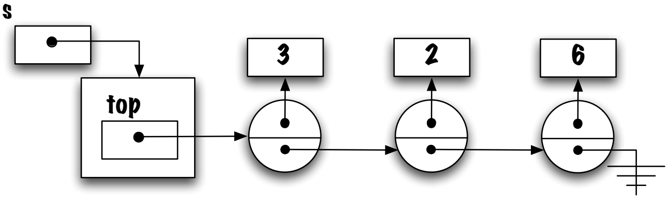
Java: nested class

Java: nested

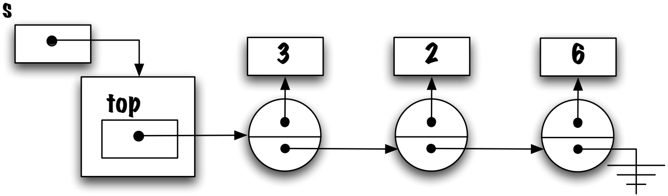
- **Elem** is a **nested** class of the class **LinkedStack**.
- Although the visibility of the class and its variables is **private**, the class **LinkedStack** has access to the instance variables of the class **Elem** because its implementation is nested.
- For now, the nested classes will be “static”. We will use them as if they were top-level classes except that 1) the declaration is nested and 2) the implementation is accessible to the outside class.
- Later we will see that there is a second category of nested classes.

Implementation of the methods

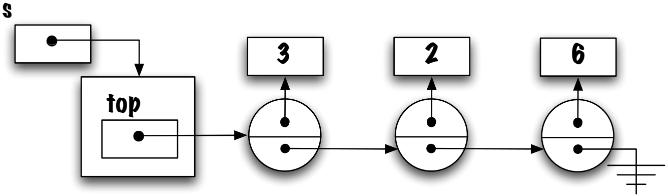
boolean isEmpty()



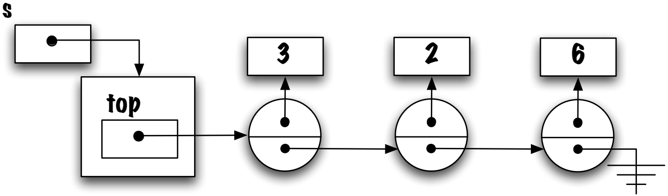
E peek()



void push(E value)



E pop()



Summary

- The concept of **reference variable** is central to linked implementations.
- The class **Elem** has two instance variables, one of them is used to save an element of information, the other one is used as a tether for the next element of the list.

Next module

- Error handling in Java: **Exception**

References I

E. B. Koffman and Wolfgang P. A. T.
***Data Structures: Abstraction and Design Using Java*.**
John Wiley & Sons, 3e edition, 2016.

Marcel **Turcotte**

Marcel.Turcotte@uOttawa.ca

School of Electrical Engineering and **Computer Science** (EECS)

University of Ottawa