ITI 1121. Introduction to Computing II

List: concept

by

Marcel Turcotte

Version March 12, 2020

Preamble

Preamble

Overview

Overview

List: concept

Following our exploration of stacks and queues, we look at another linear abstract type of data (ADT), the list. We discover the generality of this ADT. We mention the implementation using an array, but we focus our attention on three implementations using linked elements: the singly linked list, the doubly linked list, and the circular doubly linked list, starting with a dummy node.

General objective :

- This week, you will be able to design an industrial-grade implementation of the abstract data type list.

Preamble

Learning objectives

Learning objectives

- **Explain** the role of reference variables in the implementation of a linked list.
- **Design** a method to traverse a singly linked list.

Readings:

- Pages 63-84 of E. Koffman and P. Wolfgang.

Preamble

Plan

Plan

Definitions

Definition

A list (**List**) is an abstract data type (ADT) to store objects, such that each element has a predecessor and a successor (thus linear, ordered), and **having no data access restrictions**; one can inspect, insert or delete anywhere in the list. A.K.A. **Sequence**.

Operations

The basic **operations** are:

int size(): returns the number of saved elements; the empty list has a size of 0;

int get(int index): access to the elements is by position (or by content).

- What will be the index of the first element in the list, 0 or 1?
  - As for the arrays, the first element is at position 0;

void add(int index, E elem): adding an element at any position;

void remove(int index): removal of an element by position (or content).

Remarks

- So **lists** are more general than stacks and queues.
- These can be implemented using a list.

List

```
public interface List<E> {
    void add(int index, E elem);
    boolean add(E elem);
    E remove(int index);
    boolean remove(E o);
    E get(int index);
    E set(int index, E element);
    int indexOf(E o);
    int lastIndexOf(E o);
    boolean contains(E o);
    int size();
    boolean isEmpty();
}
```

- The interface above declares a subset of the methods of the interface **java.util.List**.

Implementations

Implementations

- **ArrayList**
- **LinkedList**
  - **Singly** linked list
  - **Doubly** linked list
  - List with a **dummy node**
  - **Iterative** processing (**Iterator**)
  - **Recursive** processing.

New concepts will be introduced as needed to improve the efficiency of our implementations. **Efficiency** with respect to the **execution time** and/or **memory use**; we will be mainly interested in the execution speed.

Implementations

ArrayList

Discussion: implement List using an array

- **Summarize** the main features of implementing a list using an **array**:
  - An **instance variable** designates an array;
  - An **instance variable** to count the number of elements;
  - We **create the array** in the constructor of the class;
  - We're using the **dynamic array** technique.
- Given a list containing the elements **a**, **b**, **c** and **d**, give the contents of the array after the call **add(2, z)**.
- For the resulting list, give the result of the call **remove(0)**.

Implementations

Singly linked list

Singly linked list

- The simplest implementation is the singly linked list (**SinglyLinkedList**).
- We will use a "static" nested class to represent the nodes in the list. Each node contains a value and is connected to its next one.

```
private static class Node<T> {
    private T value;
    private Node<T> next;
    private Node(T value, Node<T> next) {
        this.value = value;
        this.next = next;
    }
}
```

Singly linked list

- The class **SinglyLinkedList** has an instance variable that designates the first element of the list, which we will call **head**.
- The nested class is sometimes called **Elem** or **Entry**.

Implementations

addFirst(E elem)

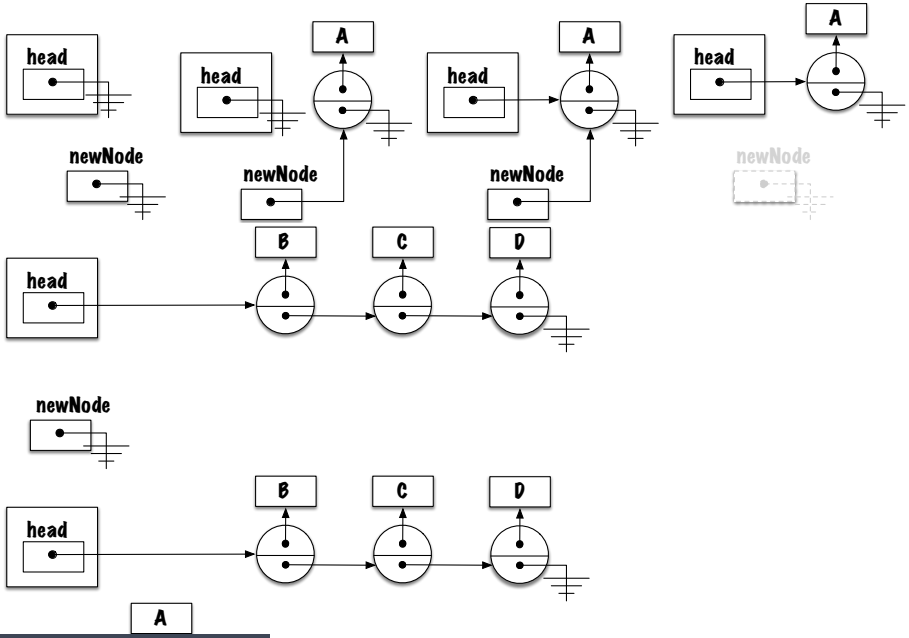
addFirst(E elem)

Inserting an element at the start of the list requires:

1. the creation of a **new node**, as well as
2. **adding** the element to the list.

```
public void addFirst(E elem) {

    Node<E> newNode;
    newNode = new Node<E>(elem, null);

    if (head == null) {
        head = newNode;
    } else {
        newNode.next = head;
        head = newNode;
    }
}
```

addFirst(E elem)



Discussion

- Is this distinction between the case of the empty **list** and the case of the **list having elements** really necessary?
- **What do you think** of this implementation?

```
public void addFirst(E elem) {
    head = new Node<E>(elem, head);
}
```

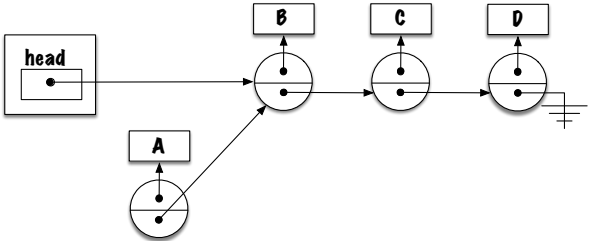
Discussion (continued)

- **Does that work** for the **special case** and the **general case**?
  - **Yes**, it does.
  - **Why?**
    - Because Java first evaluates the right side of the expression.

addFirst(E elem)

Evaluation on the right side.

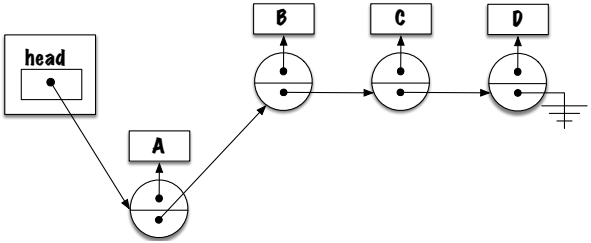
```
head = new Node<E>(elem, head);
```



addFirst(E elem)

The result (a reference to the newly created element) is assigned to the variable **head**.

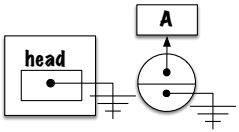
```
head = new Node(elem, head);
```



addFirst(E elem)

Similarly, the result of the evaluation of the right side, here **head** is **null**, is assigned to the instance variable **next** of the node.

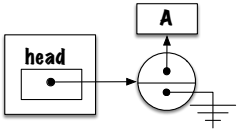
```
head = new Node<E>(elem, head);
```



addFirst(E elem)

The result (a reference to the newly created element) is assigned to the variable **head**.

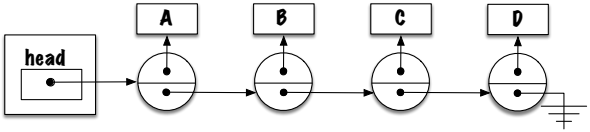
```
head = new Node<E>( elem , head );
```



Implementations

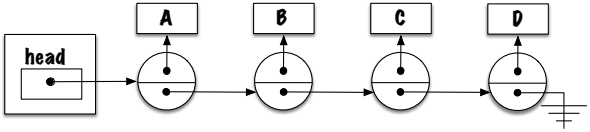
add(E elem)

Discussion add(E elem)



- The method **add(E Elem)** adds the element **at the end of the list**.
  - Without adding a reference to the last node, **how** is the element added to the end of the list?
    - The solution has to be **general**!

add(E elem)



add(E elem)

- What will be the **termination condition** of the loop?

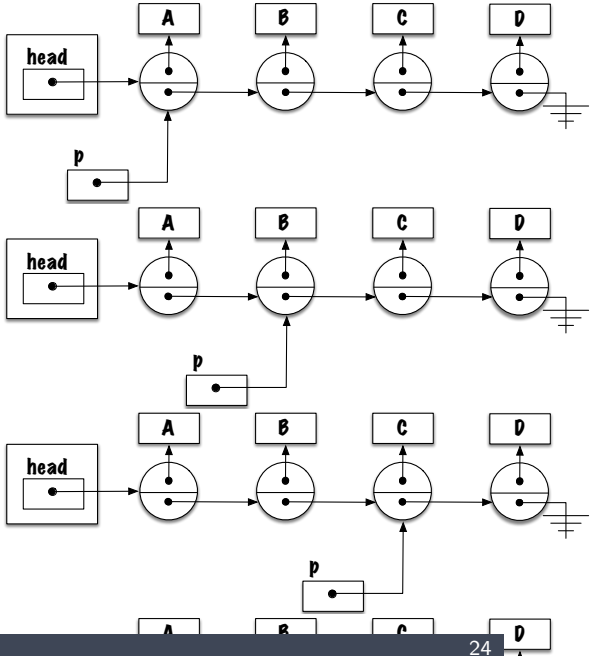
```
public void add(E elem) {

    Node<E> newNode, p;

    newNode = new Node<E>(elem, null);

    p = head;
    while (           ) {
        p = p.next;
    }

    p.next = newNode;
}
```

add(E elem)

- **What do you think?**

```
public void add(E elem) {

    Node<E> newNode, p;

    newNode = new Node<E>(elem, null);

    p = head;
    while (p != null) {
        p = p.next;
    }

    p.next = newNode;
}
```

add(E elem)



add(E elem)

- After executing the loop, the value of **p** is **null**, an exception of type **NullPointerException** will be thrown when executing **p.next = newNode**.

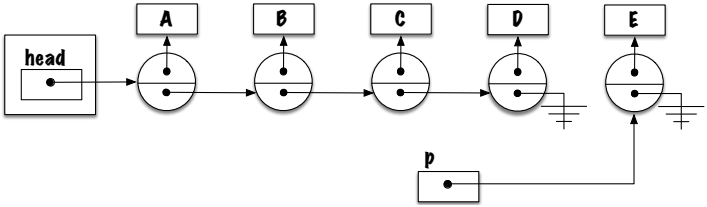
```
public void add(E elem) {

    Node<E> newNode, p;

    newNode = new Node<E>(elem, null);
    p = head;

    while (p.next != null) {
        p = p.next;
    }

    p.next = newNode;
}
```

add(E elem)

- New proposal!
- **What do you think of that?**

```
public void add(E elem) {

    Node<E> newNode, p;

    newNode = new Node<E>(elem, null);
    p = head;

    while (p != null) {
        p = p.next;
    }

    p = newNode;
}
```

add(E elem)



add(E elem)

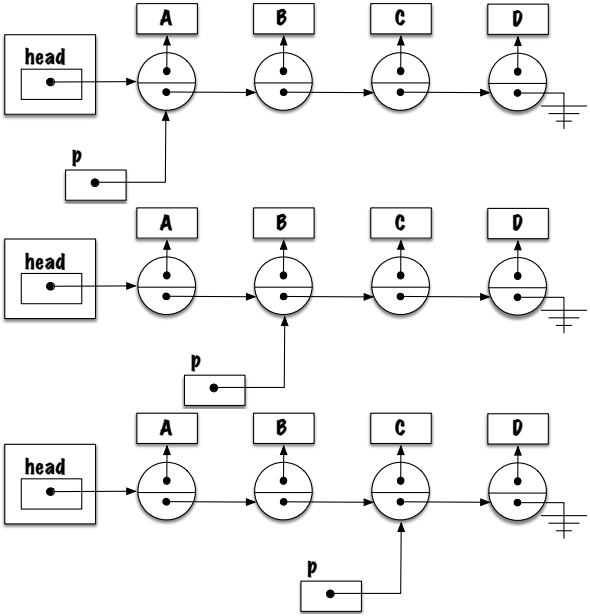
- What will be the **termination condition** of the loop?

```
public void add(E elem) {

    Node<E> newNode, p;

    newNode = new Node<E>(elem, null);

    p = head;
    while (           ) {
        p = p.next;
    }

    p.next = newNode;
}
```

add(E elem)

```
public void add(E elem) {

    Node<E> newNode, p;

    newNode = new Node<E>(elem, null);

    p = head;
    while (p.next != null) {
        p = p.next;
    }

    p.next = newNode;
}
```

add(E elem)



add(E elem)

- **What happens** if the list is empty?
  - **p.next != null** causes a **NullPointerException**!
- **What variable** should be pointing at the new element?

```
public void add(E elem) {
    Node<E> newNode, p;
    newNode = new Node<E>(elem, null);
    p = head;
    while (p.next != null) {
        p = p.next;
    }
    p.next = newNode;
}
```

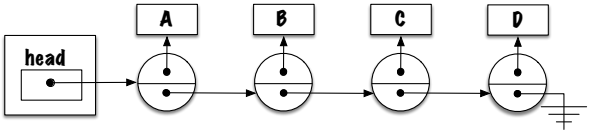
add(E elem)

```
public void add(E elem) {
    Node<E> newNode;
    newNode = new Node<E>(elem, null);

    if (head == null) {
        head = newNode;
    } else {
        Node<E> p;
        p = head;
        while (p.next != null) {
            p = p.next;
        }
        p.next = newNode;
    }
}
```

Pitfall!

```
public void add(E elem) {
    Node<E> newNode;
    newNode = new Node<E>(elem, null);

    if (head == null) {
        head = newNode;
    } else {
        while (head.next != null)
            head = head.next;
        head.next = newNode;
    }
}
```

add(E elem)



Implementations

Exercises

Exercises

Implement these methods:

- **E removeFirst()**
- **E removeLast()**
  - What will be the **stop-criterion** for the loop?

Implementations

remove(E elem)

remove(E elem)

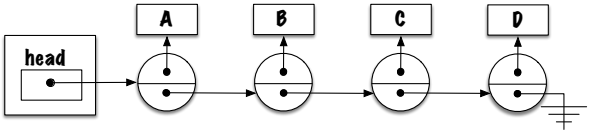
- Accessing elements **through content**!
- Return **true** is **elem** has been removed and **false** otherwise.

remove(E elem)

Formulate your **strategy**!

1. **Traverse** the list
2. **Stop criterium?**
3. **Removal**

remove(E elem)



remove(E elem)

- **What do you think?**

```
public boolean remove(E elem) {
        Node<E> p, r;
        p = head;
        while (p != null && ! p.value.equals(elem)) {
            p = p.next;
        }
        r = p;
        // ...
    return true;
}
```

```
public boolean remove(E elem) {

    Node<E> p, r;
    p = head;

    while (p.next != null && ! p.next.value.equals(elem)) {
      p = p.next;
    }

    r = p.next;
    p.next = r.next;

    return true;
}
```

- What happens if the list is **empty**
- What happens if the element is **missing** from the list?

```
public boolean remove(E elem) {
    boolean result = true;
    if (head == null) {
        result = false;
    } else if (head.value.equals(elem) ) {
        head = head.next;
    } else {
        Node<E> p;
        p = head;
        while (p.next != null && ! p.next.value.equals(elem)) {
            p = p.next;
        }
        if (p.next == null) {
            result = false;
        } else {
            p.next = p.next.next;
        }
    }
    return result;
}
```

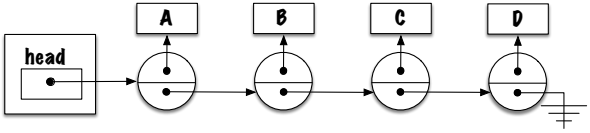
Implementations

E get(int pos)

E get(int pos)

- Returns the value saved at the position **pos** in the list.
  - Accessing **by position**!
- The **first element** in the list is at **position 0**.
- **Elaborate** your strategy.

E get(int pos)



E get(int pos)

```
public E get(int pos) {

    Node<E> p;
    p = head;

    for (int i=0; i<pos; i++) {
        p = p.next;
    }

    return p.value;
}
```

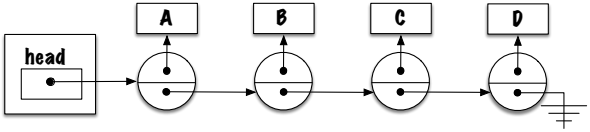
Implementations

E remove(int pos)

Exercise

- Implement the method **E remove(int pos)**.

E remove(int pos)



E remove(int pos)

```
public E remove(int pos) {
    // pre-conditions: ?
    E saved;
    Node<E> r;
    Node<E> p;
    p = head;
    for (int i=0; i<(pos-1); i++) {
        p = p.next;
    }
    r = p.next;
    p.next = r.next;
    saved = r.value;
    return saved;
}
```

- What happens if **pos == 0**?

E remove(int pos)

```
public E remove(int pos) {
    E saved;
    Node<E> r;
    if (pos == 0) {
        r = head;
        head = head.next;
    } else {
        Node<E> p;
        p = head;
        for (int i=0; i<(pos-1); i++) {
            p = p.next;
        }
        r = p.next;
        p.next = r.next;
    }
    saved = r.value;
    return saved;
}
```

Prologue

Summary

- A list (**List**) is an abstract data type (ADT) to store objects, such that each element has a predecessor and a successor (thus linear), and **having no restrictions on data access**; one can inspect, insert or delete anywhere. in the list.
- To **traverse** a list, we use a local variable of type **Node** that we initialize with the value of **head**.

Next module

- **List** : implementation techniques

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