EECS2011 Fundamentals of Data Structures (Winter 2022)

Q&A - Week 3 Lecture

Wednesday, February 2

Announcements

1. addLast
2. removeLast

Node<String>
Node<Integer>

- Lecture W4 released (SLL and generics review)
- Assignment 1 (on SLLs) released on Monday.

grading: starters + additional

suggestions in comments.

1. Recursion problems in Q&A
   ↳ not necessary WTI.
2. Some RT analysis of recursion in Lab-W7

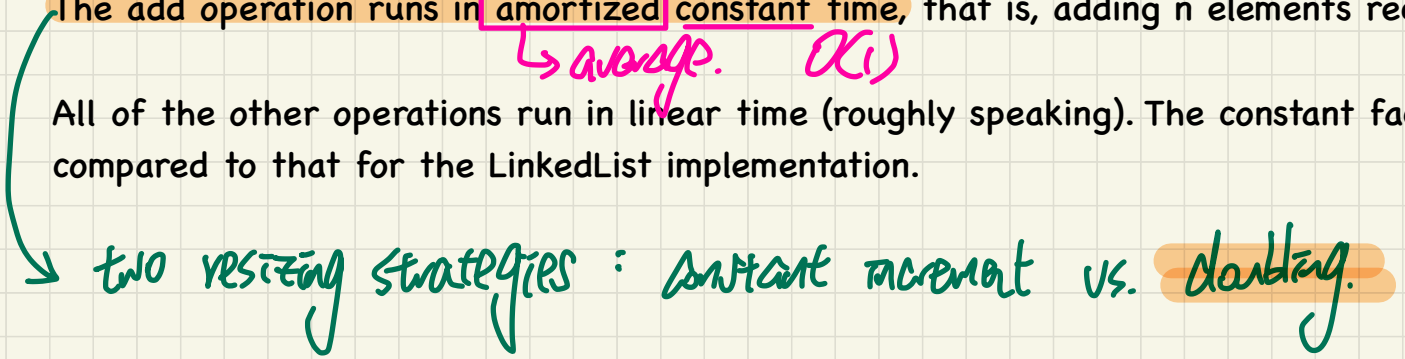
Hello professor.
My question is, is inserting and deleting elements in the arraylist OP and is O(1)?
I don't think it is but I want to make sure. Thank you.

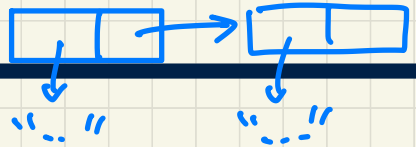
https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

Resizable-array implementation of the List interface. Implements all optional list operations, and permits all elements, including null. ...

The size, isEmpty, get, set, iterator, and listIterator operations run in constant time.
The add operation runs in amortized constant time, that is, adding n elements requires O(n) time.

All of the other operations run in linear time (roughly speaking). The constant factor is low compared to that for the LinkedList implementation.





Hi professor.

I was wondering, using generics, are we allowed to have different types of 'elements' in one chain of linked list?

Because I remember you saying we shouldn't mix the types in eecs2030 but i wanted to make sure it is the same with linked lists.

Thank you.

parameter of a type

```
class Node<E> {
    private E element;
    private Node<E> next;
    ⋮
}
```

```
Node<String> n1;
Node<Integer> n2;
n1.setElement("Tom"); ✓
n1.setElement(34); ✗
```

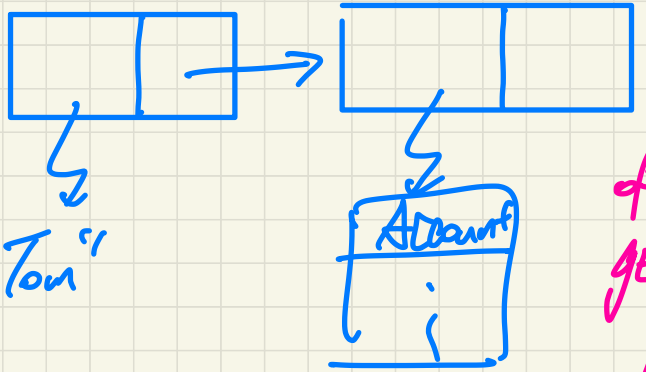
Node<Object> nl = new Node<Object>();

Object[]

- Super class of every class
- every class is a subclass of Object

nl.setElement("Tom");

nl.setElement(new Account(...))

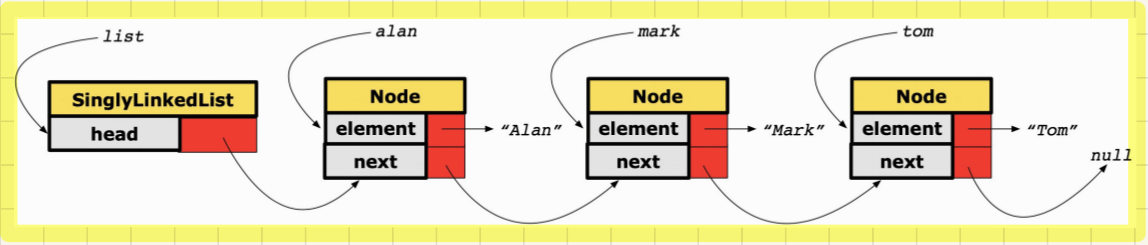


poor choice of instantiating generic type parameter

`Node<String>` → restricts what the next node can store

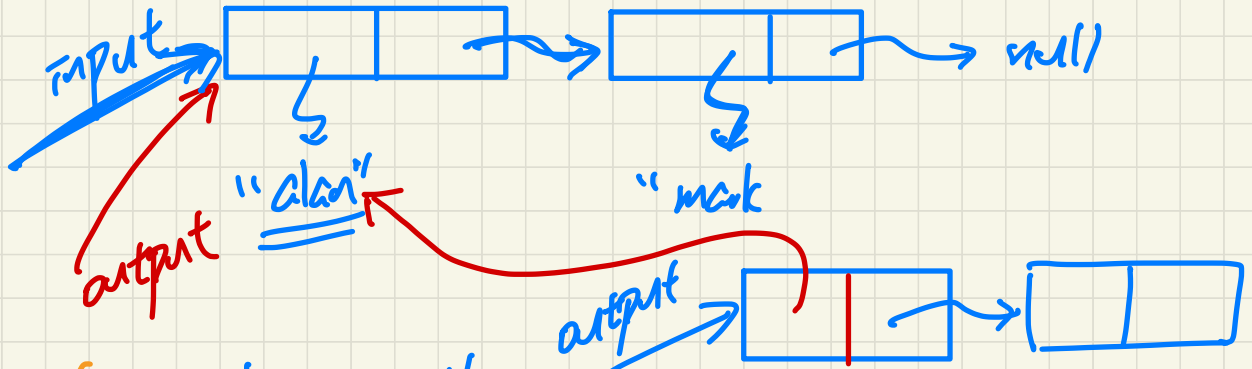
```
Node<String> n1 = new Node<>("a", null);
n1.setNext(new Node<Integer>(3, null));
```

When you asked us to try and find the size of a linked list,
I came up with a solution very different from yours.
My method simply has a "size" function in the Node class
that returns 1 if the node pointer is null,
and otherwise returns 1+nextNode.size().
Would this be correct?
In general, in different situations how can I tell which technique is better?




```
public int getSize() {
    if (head == null) { return 0; }
    else { 1 + next.getSize(); }
}
```

get back here tomorrow!



X output = input;

A1

Node<String> output = new Node<>(input.getElement());

output.setNext(new Node(input.getNext.getElement(), null));

Problem on SLL: Reversing a Chain of Nodes

You are asked to program this method:

public Node<String> reverseOf(**Node<String>** input)

The returned node references the front of a separate chain of nodes representing the reverse of the input.

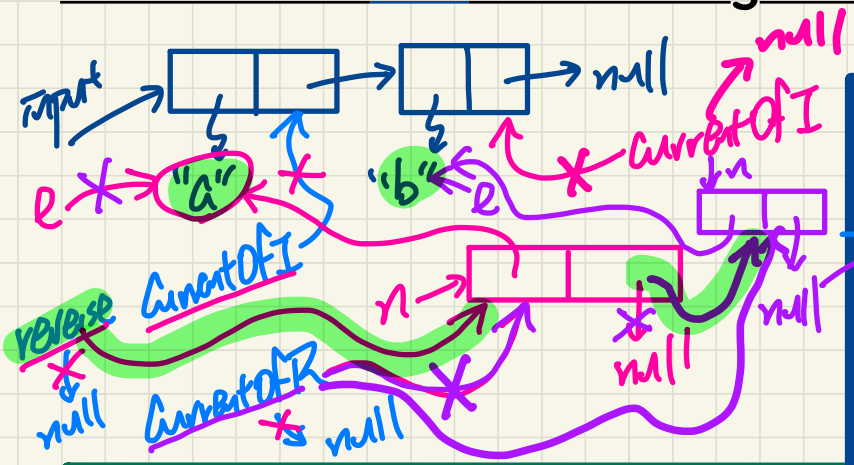
Requirement: The input node may or may not be null.

```
@Test
public void test() {
    ListUtilities util = new ListUtilities();

    Node<String> input = null;
    Node<String> output = util.reverseOf(input);
    assertNull(output);

    input = new Node<>("Alan", new Node<>("Mark", new Node<>("Tom", null)));
    output = util.reverseOf(input);
    assertEquals("Tom", output.getElement());
    assertEquals("Mark", output.getNext().getElement());
    assertEquals("Alan", output.getNext().getNext().getElement());
    assertNull(output.getNext().getNext().getNext());
}
```

Problem on SLL: Reversing a Chain of Nodes?




```java
public Node<String> reverseOf(Node<String> input) {
    Node<String> reverse = null;
    Node<String> currentOfInput = input;
    Node<String> currentOfReverse = null;
    while(currentOfInput != null) {
        String e = currentOfInput.getElement();
        Node<String> n = new Node<>(e, null);
        if(reverse == null) {
            reverse = n;
            currentOfReverse = reverse;
        }
        else {
            currentOfReverse.setNext(n);
            currentOfReverse = currentOfReverse.getNext();
        }
        currentOfInput = currentOfInput.getNext();
    }
    return reverse;
}
```

```java
@Test
public void test() {
    ListUtilities util = new ListUtilities();

    Node<String> input = null;
    Node<String> output = util.reverseOf(input);
    assertNull(output);

    input = new Node<>("Alan", new Node<>("Mark", new Node<>("Tom", null)));
    output = util.reverseOf(input);
    assertEquals("Tom", output.getElement());
    assertEquals("Mark", output.getNext().getElement());
    assertEquals("Alan", output.getNext().getNext().getElement());
    assertNull(output.getNext().getNext().getNext());
}
```

Is this the correct algorithm for reversing a chain of nodes?

No...