[TAP:TQFYC] Vector vs SLL

- What is an advantage of Singly Linked List (SLL) over Vector?
  > A. Faster access to any element~~s~~
  > [✓ B.] Faster add() to the head
  C. Faster add() to the tail
  D. Having the ability to resize
  E. Whatever

Administrative Details

- Lab 1
  - Feedback on GitHub as a “Pull Request”
    - In a separate `TA-feedback` branch
  - `//$` and `/*$    */` comments are from TAs/instructors.
  - Comment on any of the PR lines if you have any questions!
- Lab 4
  - Optional partners again: please fill out form whether working alone or in pairs!

Agenda

- List
  - ⦿ Singly Linked List (SLL)
  - Circularly Linked List (CLL)
  - Doubly Linked List (DLL)

The List Interface

```
interface List {
    size()
    isEmpty()
    contains(e)
    get(i)
    set(i, e)
    add(i, e)
    remove(i)
    addFirst(e)
    getLast()
    .
    .
    .
}
```

Vector implements List

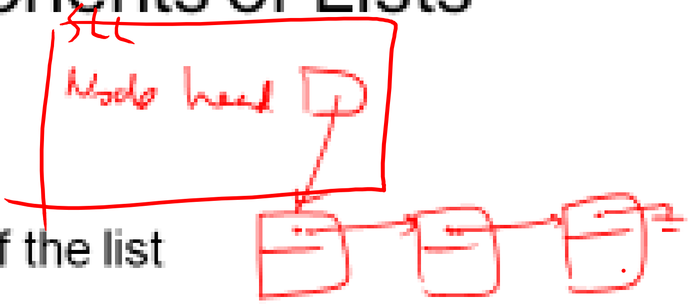
⋮

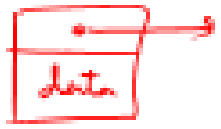
Singly Linked list

Singly Linked List

- There are two key components of Lists
  - The list itself
    - Instance varibles
      - (Pointer to) the `head` node of the list

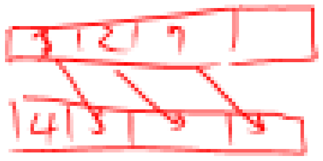
        # of elements
    - Methods
      - Those declared in the List interface
  - Nodes
    - Instance variables
      - data
      - (Pointer to) the "next" element
    - Methods
      - Getters and setters





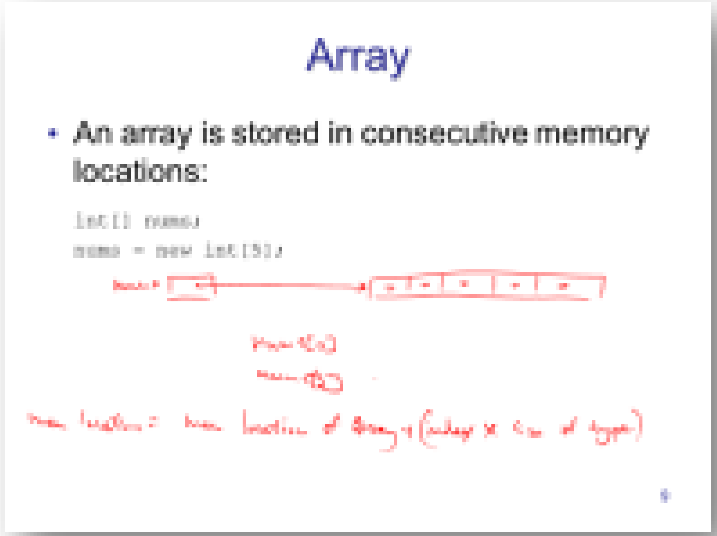
Pros and Cons of Vectors

Singly Linked List



Pros constant time random access

- Fast access to elements



Array

- An array is stored in consecutive memory locations:

```
int[] nums;
nums = new int[5];
```

nums

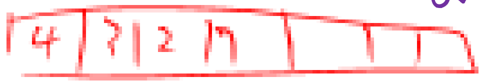
nums[0]

nums[3]

mem location = mem location of array + (index x size of type)

Cons

- ~~Slow~~ fast updates to front of list
- ~~Hard~~ easy to predict time for add (depends on internal array size) but O(n) always
- Potentially wasted space no wasted space, but there is overhead



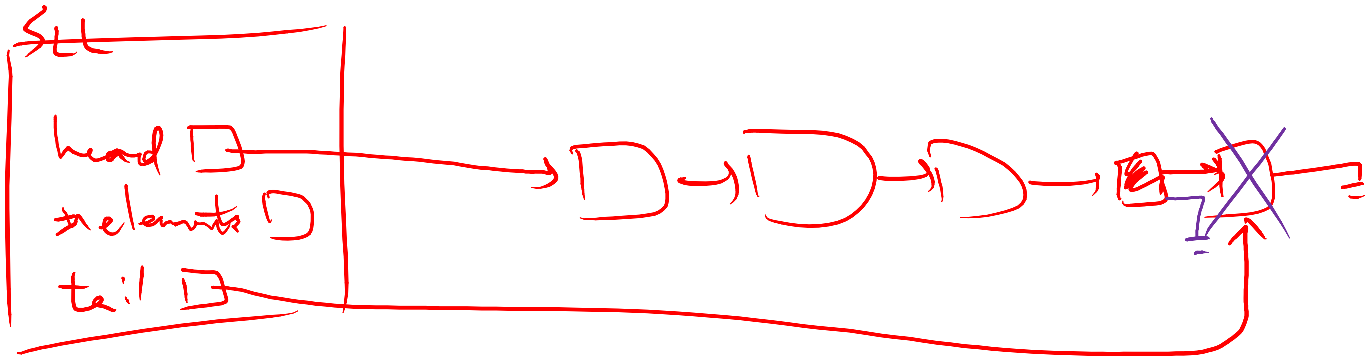
- Dynamically Resizeable? yes, but inefficient

(Worst-case) Time Complexity

Operation	Vector	SLL
size	O(1)	O(1)
addLast	O(1) or O(n) (with resizing)	O(n)
removeLast	O(1)	O(n)
getLast	O(1)	O(n)
addFirst	O(n)	O(1)
removeFirst	O(n)	O(1)
getFirst	O(1)	O(1)
get(i)	O(1)	O(n)
set(i)	O(1)	O(n)
remove(i)	O(n)	O(n)
contains	O(n)	O(n)
remove(o)	O(n)	O(n)

Food for Thought: SLL Improvements to Tail Ops

- In addition to `Node head`, `int elementCount`, add `Node tail` reference to SLL



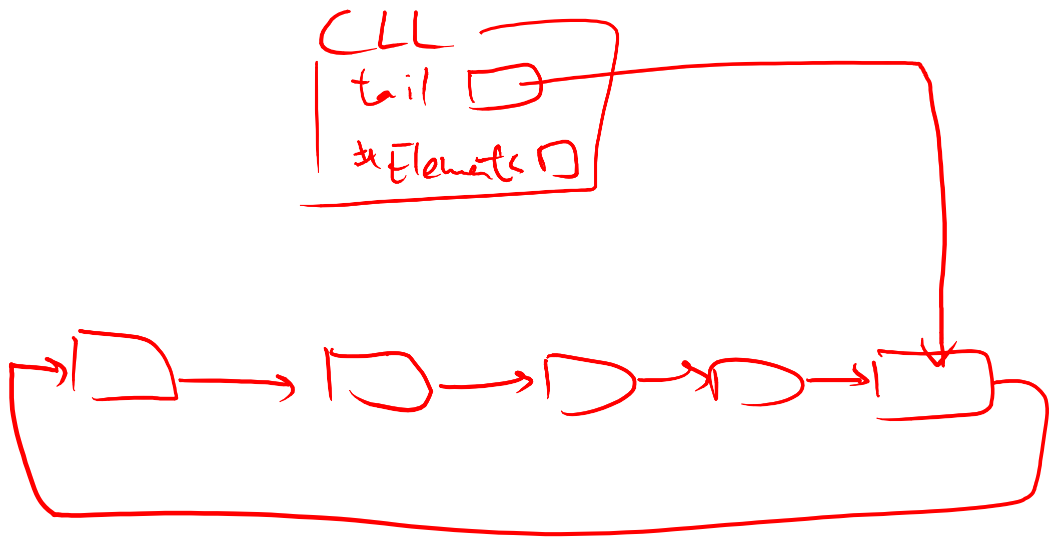
addLast()	~~O(n)~~ O(1)
removeLast()	O(n)
getLast()	~~O(n)~~ O(1)

Agenda

- List
  - Singly Linked List (SLL)
  - ⊙ Circularly Linked List (CLL)
  - Doubly Linked List (DLL)

Circularly Linked Lists

- Use *next* reference of last element to reference head of list
- Replace ***head*** reference with ***tail*** reference

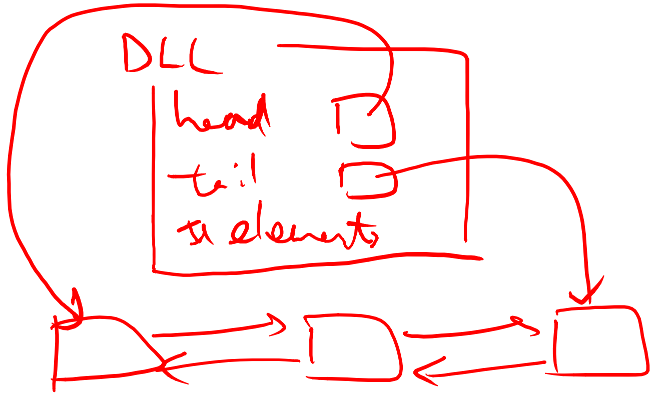


Agenda

- List
  - Singly Linked List (SLL)
  - Circularly Linked List (CLL)
  - ⊙ Doubly Linked List (DLL)

Doubly Linked Lists

- Keep reference/links in **both** directions
  - previous and next



Lab 4: Dummy Nodes

- We will implement a modified version of DLL

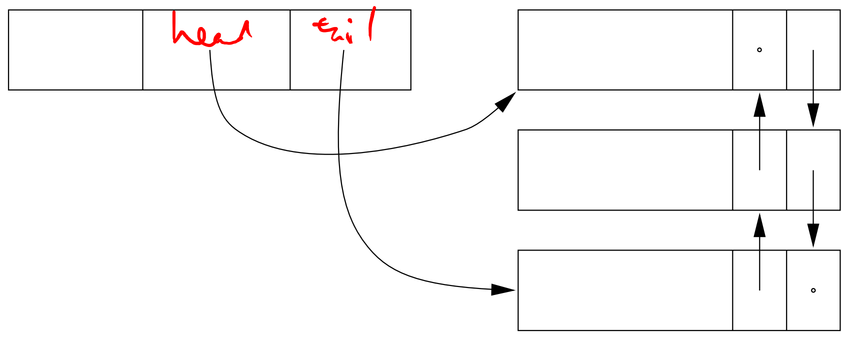
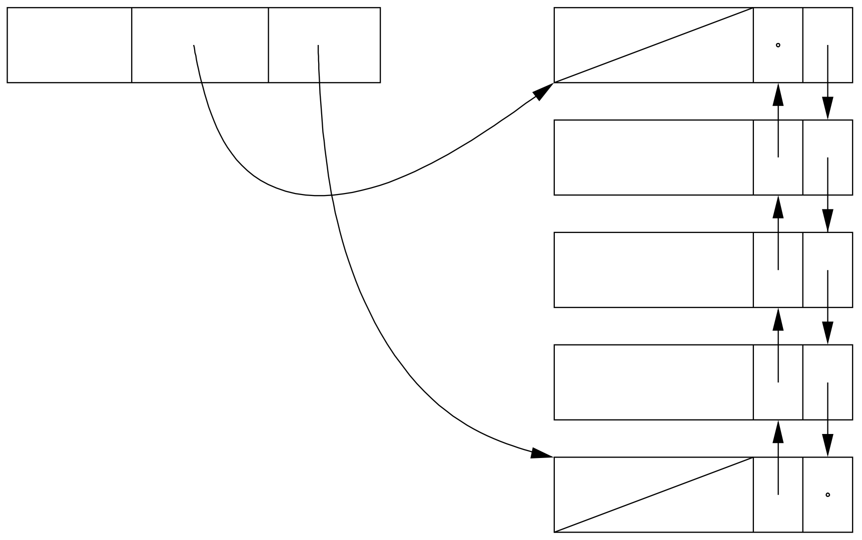


Figure 9.7, *Bailey* pg. 202



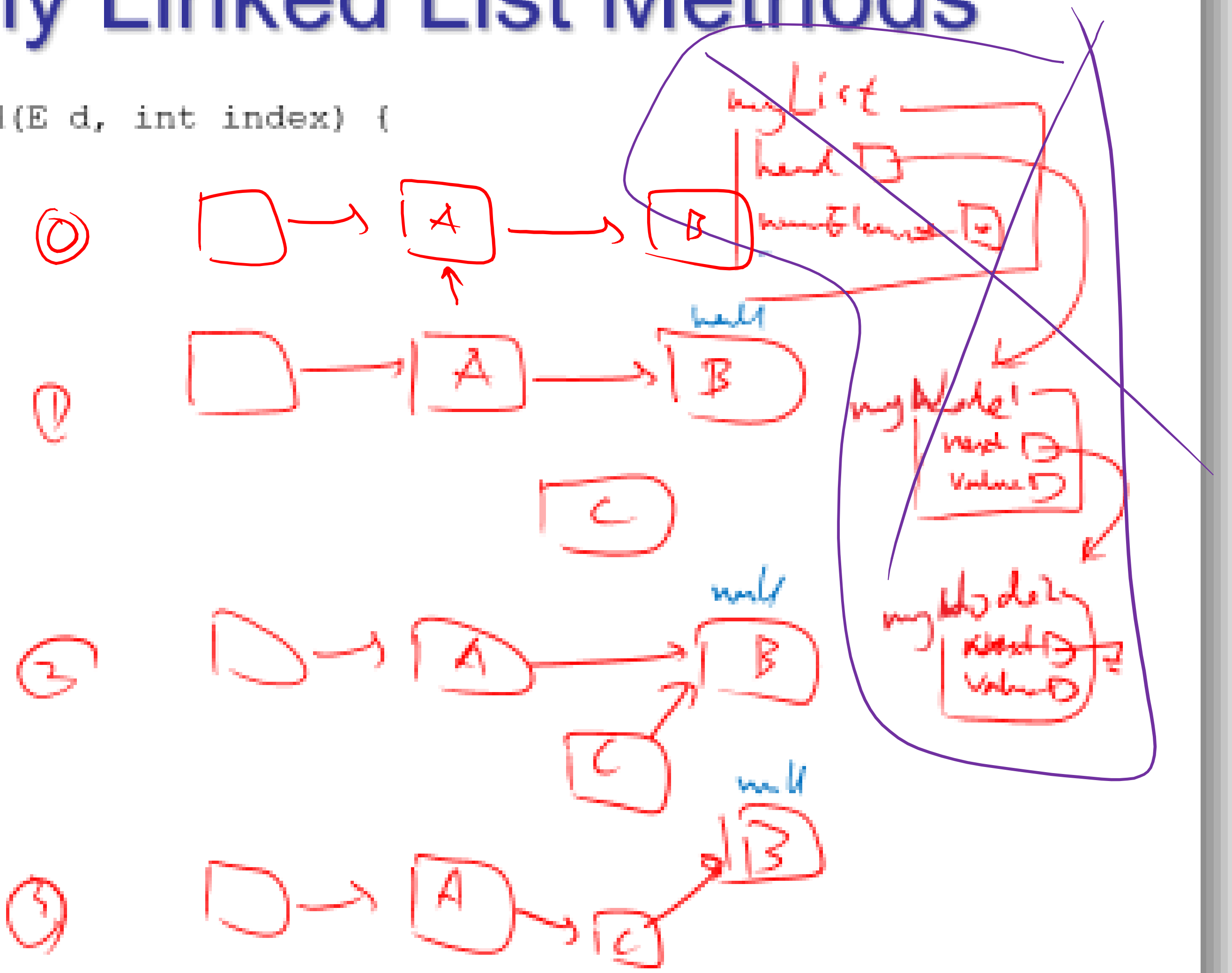
Bailey pg. 215

Agenda

- List
  - Singly Linked List (SLL)
  - Circularly Linked List (CLL)
  - Doubly Linked List (DLL)

Singly Linked List Methods

```
public void add(E d, int index) {
```



Singly Linked List Methods

```
public void add(E d, int index) {

    if (index == 0)
        addFirst(d);
    elseif (index == numElements)
        addLast(d);
    else {
        Node finger = head;

        for (int i=0; i < index-1; i++){
            finger = finger.next();
        }                                   ⓪

        Node el = new Node(d, finger.next()); // ① ②
        finger.setNext(el); // ③
        numElements++;
    }
}
```