Lists

Sridhar Alagar

List ADT

List is a sequence (ordered collection) of elements

- List may contain duplicates, null
- List is an interface in Java

Notation: $\langle A_0, A_1, \ldots, A_{n-1} \rangle$

e.g., <5, 7, 8, 1, 25>

Note: element in a list has a position

List ADT - Operations

Basic operations:

- insert/add(x): add new element x at the end of the list
- find/contains(x): does x appear in the list?
- delete/remove(x): delete the first occurrence of x

Other common operations:

- size(): returns the number of elements in the list
- isEmpty():
- clear():

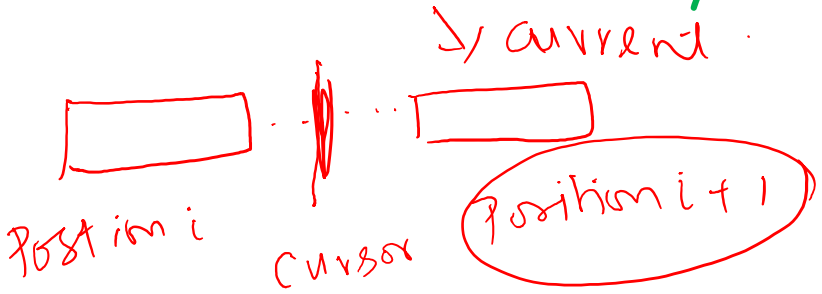
Indexed operations:

- get(i): returns the element at position i in the list
- set(i,x): sets the element at position i to x
- add(i, x): inserts x at position i
- remove(i): delete element at position i

List ADT - Operations

Iterator: goes through the element of the list one at a time

- hasNext(): is there a next item?
- next(): returns the next element in the list
- remove(): deletes the last element returned by next()



"ListIterator"

prev

List ADT - Implementations

Implementations:

- Based on Arrays
- Based on Linked Lists

ArrayList

```
public class ArrayList<E> implements List<E>{
        private E[] array;
        int n; // no. of elements in array

        public ArrayList(){
                n = 0;
                array = new E[4];
        }
}
```

Compile error: cannot create array of parametric type

ArrayList

```
Public class ArrayList<E> implements List<E>{
        private E[] array;
        int n; // no. of elements in array

        public ArrayList(){
                n = 0;
                array = (E) new Object[4];
        }
}
```

ArrayList: get() and set() methods

```
public E get(int i){
    ↪ if (i < 0 && i > n - 1) throw new IndexOutOfBoundException
      return array[i];
}

public void set(int i, E elem){
      if (i < 0 && i > n - 1) throw new IndexOutOfBoundException
      array[i] = elem;

}
```

RT: O(1) for both get() and set()

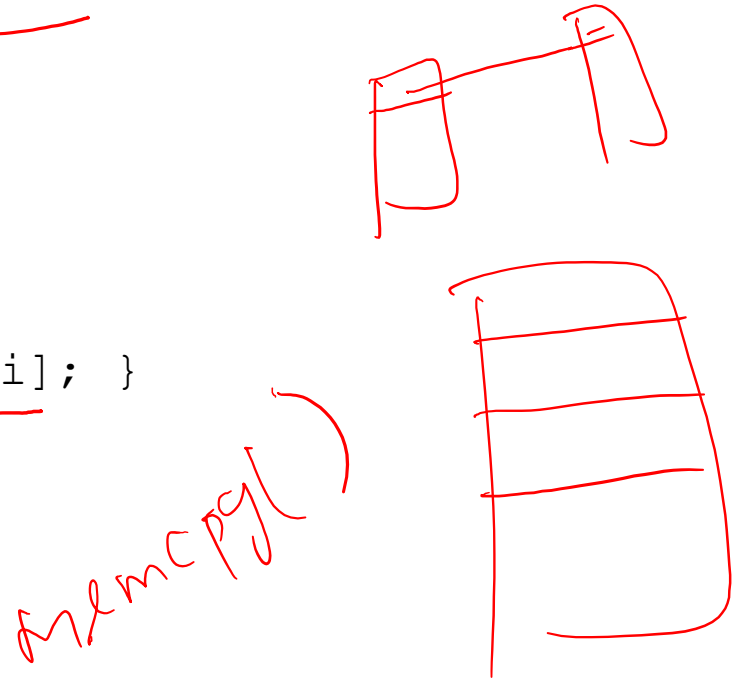
ArrayList: add(i, elem)

```
public boolean add(int i, E elem){
        if (i < 0 && i > n) throw new IndexOutOfBoundException
        if (n+1 > array.length) grow(); //increase the size of array
        for (int j = n; j > i; j--) {array[j] = array[j-1];}
        array[i] = elem;
        n++;
        return true;
}
private void grow(){
        E[] tmp = (E) new Object[2*n];
        for(int i < 0; i < n; i++){ tmp[i] = array[i]; }
        array = tmp;
}
```

RT for grow: O(1) amortized over n add operations
RT for add(i, elem): O(n-i) amortized over n add operations

ArrayList: add(i, elem)

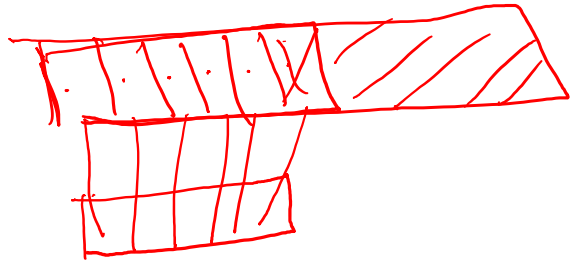
```
public boolean add(int i, E elem){
        if (i < 0 && i > n) throw new IndexOutOfBoundException
        if (n+1 > array.length) grow(); //increase the size of array
        for (int j = n; j > i; j--) {array[j] = array[j-1];}
        array[i] = elem;
        n++;
        return true;
}
private void grow(){
        E[] tmp = (E) new Object[2*n];
        for(int i < 0; i < n; i++){ tmp[i] = array[i]; }
        array = tmp;
}
```



Efficient implementation?

use any bulk copy: System.arraycopy()

ArrayList: remove(i)



```
public boolean remove(int i){
        if (i < 0 && i > n) throw new IndexOutOfBoundException
        if (n < array.length / 2) shrink(); //decrease the size of array
        for (int j = i; j < n-1; j++) {array[j] = array[j+1];}
        n--;
        return true;
}

Private void shrink(){
        E[] tmp = (E) new Object[n];
        for(int i < 0; i < n; i++){ tmp[i] = array[i]; }
        array = tmp;
}
```

RT: O(n-i) amortized over n remove operations

ArrayList: add(element), remove()

```
public boolean add(E elem){ // add at the end
       add(n, element);
       return true;
}

public boolean remove(){// remove the first element
       remove(0);
       return true;
}
```

RT for add(): O(1) amortized over n add operations
RT for remove(): O(n)

ArrayList: contains(element)

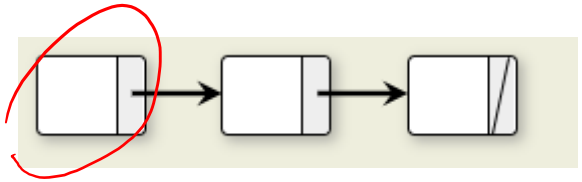
```
public boolean contains(E element){

        for (int j = 1; j < n; j++) {
                if (array[j].equals(element) == 0)
                        return true;
        }
        return false;
}
```

Time complexity: O(n)

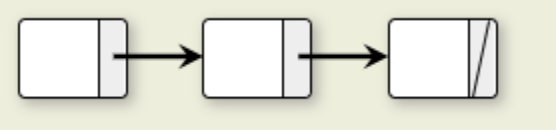
Linked List based implementation

Memory for list elements is dynamically allocated as needed

List of structures (nodes) that contains reference to neighbors (next/prev)

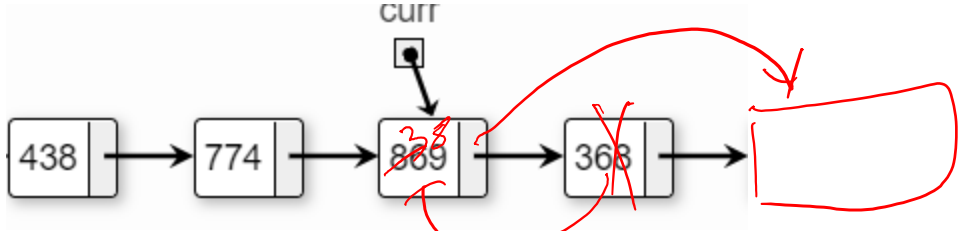


Singly Linked List



```
public class Entry <E> {
        E element;
        Entry<E> next;
        public Entry() {//constructor}
}

public class SinglyLinkedList<T> implements List<T>{
        Entry<T> head, tail;
        int n; // no. of elements in array

        public SinglyLinkedList(){
                head = new Entry();
                head.next = tail;
                tail = null;
                n = 0;
        }
}
```

How to delete the current element in a SLL?



curr.next.next

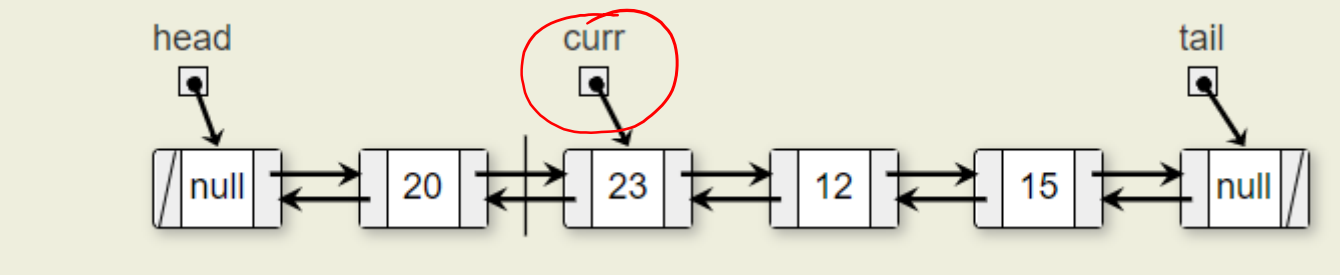
curr.elem = curr.next.elem

curr.

Doubly Linked List

Allows convenient access to both previous and next nodes

Code for DLL implementation is easier than SLL



Doubly Linked List

In java, LinkedList is doubly linked list

```
public class Entry <E> {
        E element;
        Entry<E> next, prev;
        public Entry() {//constructor}
}

Public class LinkedList<E> implements List<E>{
        Entry<E> head, tail;
        int n; // no. of elements in the list

        public LinkedList(){
                head = new Entry();
                tail = new Entry();
                head.next = tail;
                tail.prev = head;
                n = 0;
        }
}
```

LinkedList Operations

Helper method: getEntry (int i)

```
private Entry<E> getEntry(int i){
        if ( i < n/2){// start from the head
                tmp = head.next;
                for (j = 0; j < i; j++)
                        tmp = tmp.next;
        }
        else{// start from the tail
                tmp = tail.prev;
                 for (j = n - 1; j > i; j--)
                        tmp = tmp.prev;
        }
        return tmp;
}
```

LinkedList Operations

methods: get (i), set(i, element)

```
public E get(int i){
        if (i < 0 && i > n) throw new IndexOutOfBoundException;
        return getEntry(i).element;
}

public E set(int i, E element){
        if (i < 0 && i > n) throw new IndexOutOfBoundException;
        tmp = getEntry(i);
        e = tmp.element;
        tmp.element = element;
        return(e);
}
```

RT: O(min(i, n-i)) O(n)

LinkedList Operations

methods: add(i, element)

```
public void add(int i, E element){
        if (i < 0 && i > n) throw new IndexOutOfBoundException;
        tmp = getEntry(i); // get the ith entry
        e = new Entry<>(element);
        e.next = tmp;
        e.prev = tmp.prev;
        e.prev.next = e;
        tmp.prev = e;
}
```

RT: O(min(i, n-i))

LinkedList Operations

methods: remove(i)

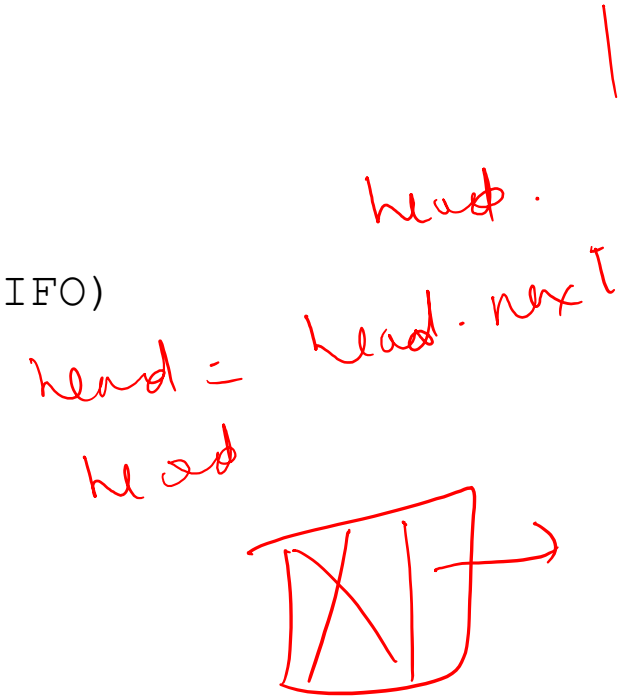
```
public E remove(int i){
        if (i < 0 && i > n) throw new IndexOutOfBoundException;
        tmp = getEntry(i);
        tmp.prev.next = tmp.next;
        tmp.next.prev = tmp.prev;
        return(tmp.element);
}
```

RT: O(min(i, n-i))

LinkedList Operations

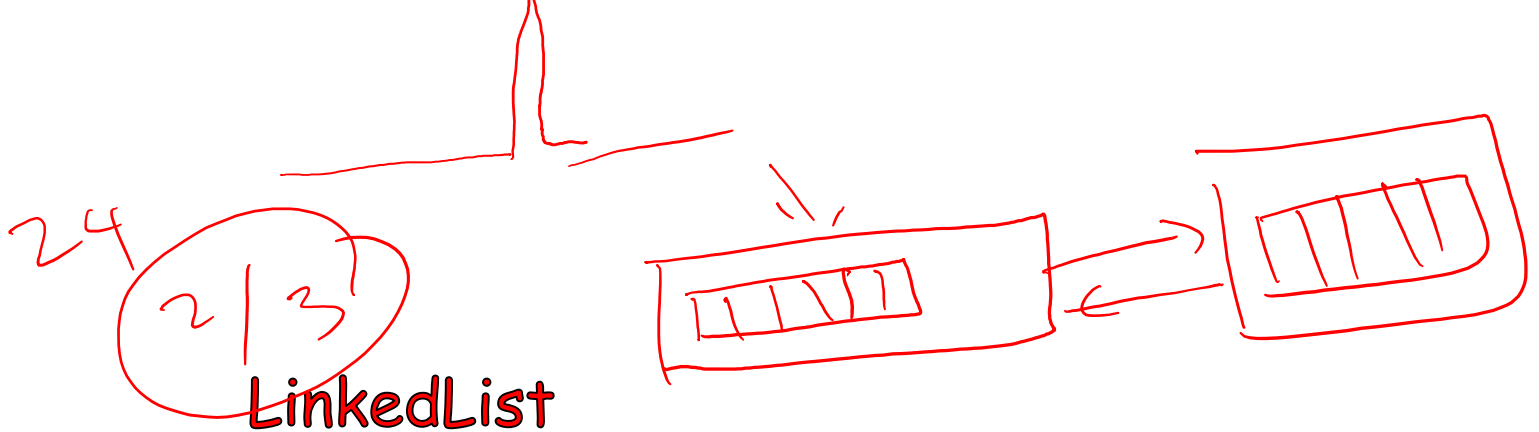
methods: add(), remove()

```
public void add(int i, E element){//add at the tail
        e = new Entry<>(element);
        e.next = tail;
        e.prev = tail.prev;
        e.prev.next = e;
        e.next.prev = e;
}
public E remove(){ // from the head (FIFO)
        tmp = head.next;
        tmp.prev.next = tmp.next;
        tmp.next.prev = tmp.prev;
        return(tmp.element);
}
```



Time complexity: O(1)

RT Comparison



Operations	ArrayList	LinkedList
get(i)	O(1)	O(n)
set(i, e)	O(1)	O(n)
add(i, e)	O(n-i)	O(n)
remove(i)	O(n-i)	O(n)
add()	O(1) amort	O(1)
remove()	O(n)	O(1)
indexOf(e)	O(n)	O(n)

2 x n

Trade-off between AL and LL?

List of bounded arrays

Which uses less memory?

List Extensions

Queue ADT

List in FIFO order

Operations

- enqueue(x)/add(x) – add element to the rear of the queue
- dequeue()/remove() – remove and return element at the head of the queue
- peek() – return element at the head of the queue
- size(), isEmpty()

based on FIFO

Data Structures for queue (FIFO):

- ArrayList, Linked List

Queue implementation - LinkedList

LinkedList list;

add(x) { list.add(x) } RT: O(1)

remove { list.remove() } RT: O(1)

In Java

Queue <E> q = new LinkedList<>()

Queue implementation - ArrayList

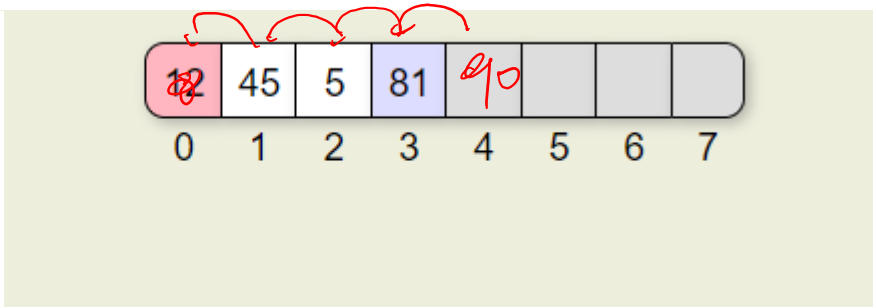
ArrayList list;

add(x) {list.add(x)}

remove {list.remove(0)}

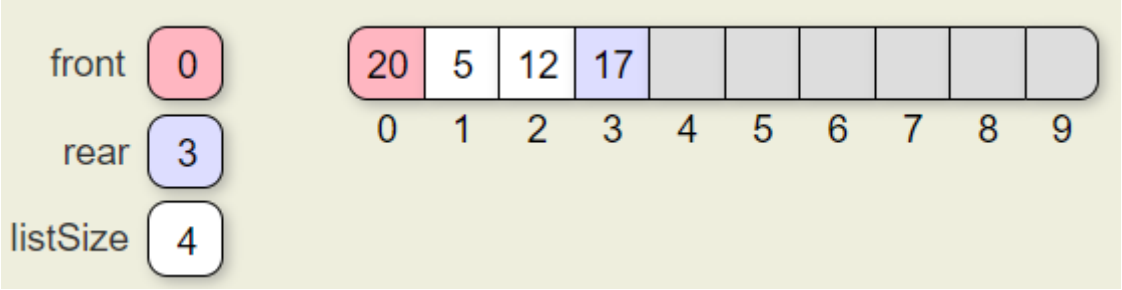
RT: O(1)

RT: O(n)



Queue implementation - ArrayList

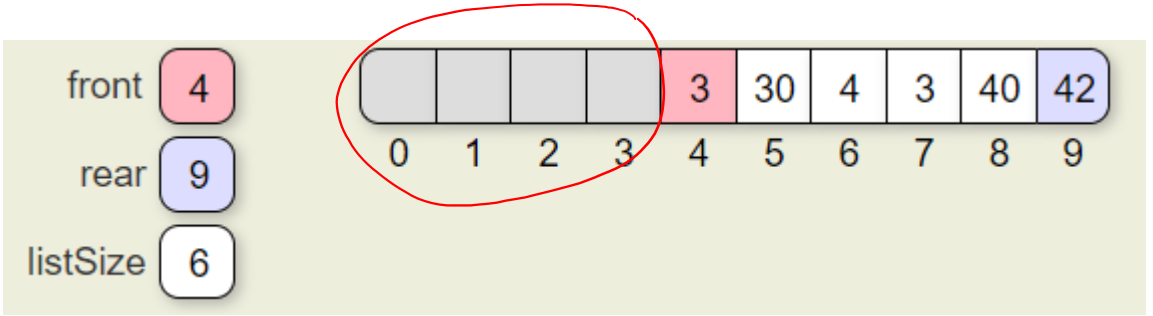
Allow active elements to drift along the array



Queue implementation - Arrays

Allow active elements to drift along the array

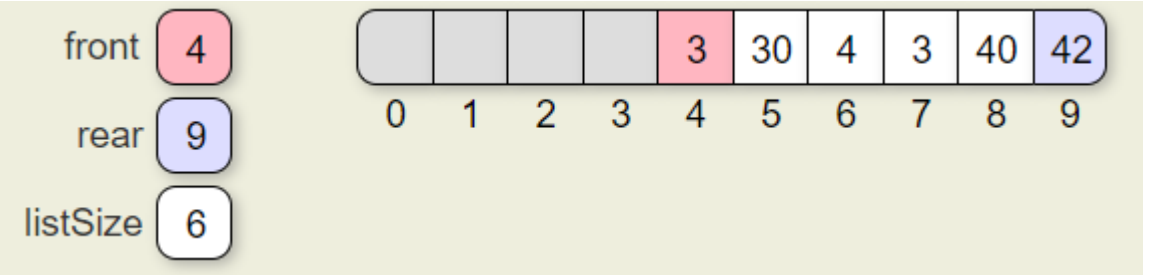
First element of the queue need not be stored at index 0



Queue implementation - Arrays

Allow active elements to drift along the array

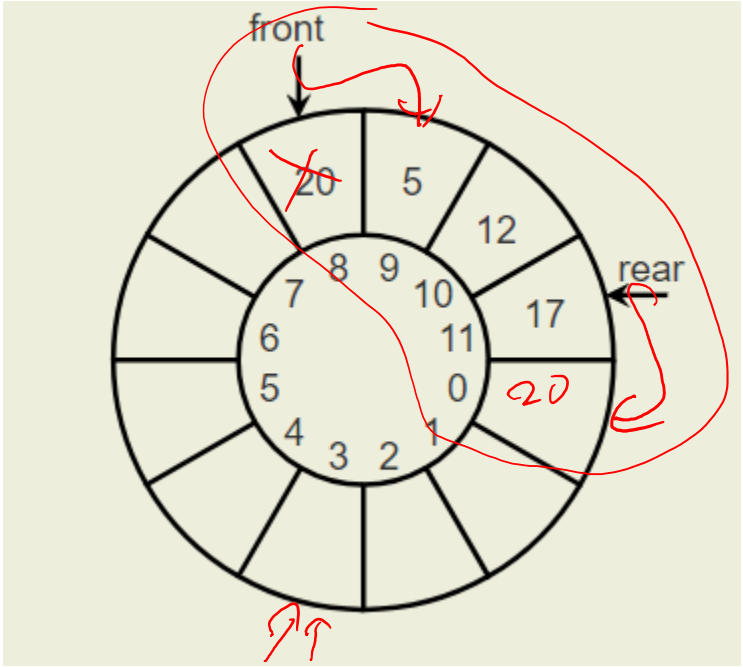
First element of the queue need not be stored at index 0



Remove is efficient

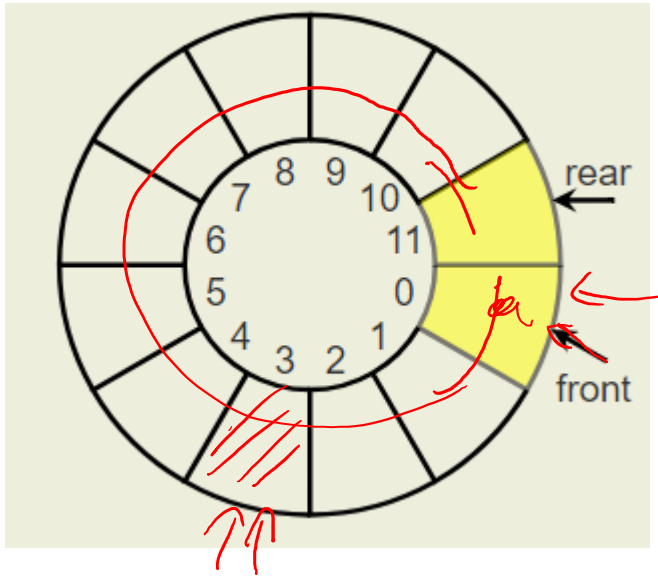
Cannot add more elements despite space availability

Queue implementation - Circular Array



Queue implementation - Circular Array

Is it empty or full?



Queue implementation - ArrayDeque

Array based double ended queue

Active elements do not start from index 0

They occupy some part of the array

In Java

```
Deque <E> q = new ArrayDeque<>()
```

Faster than using LinkList for queue

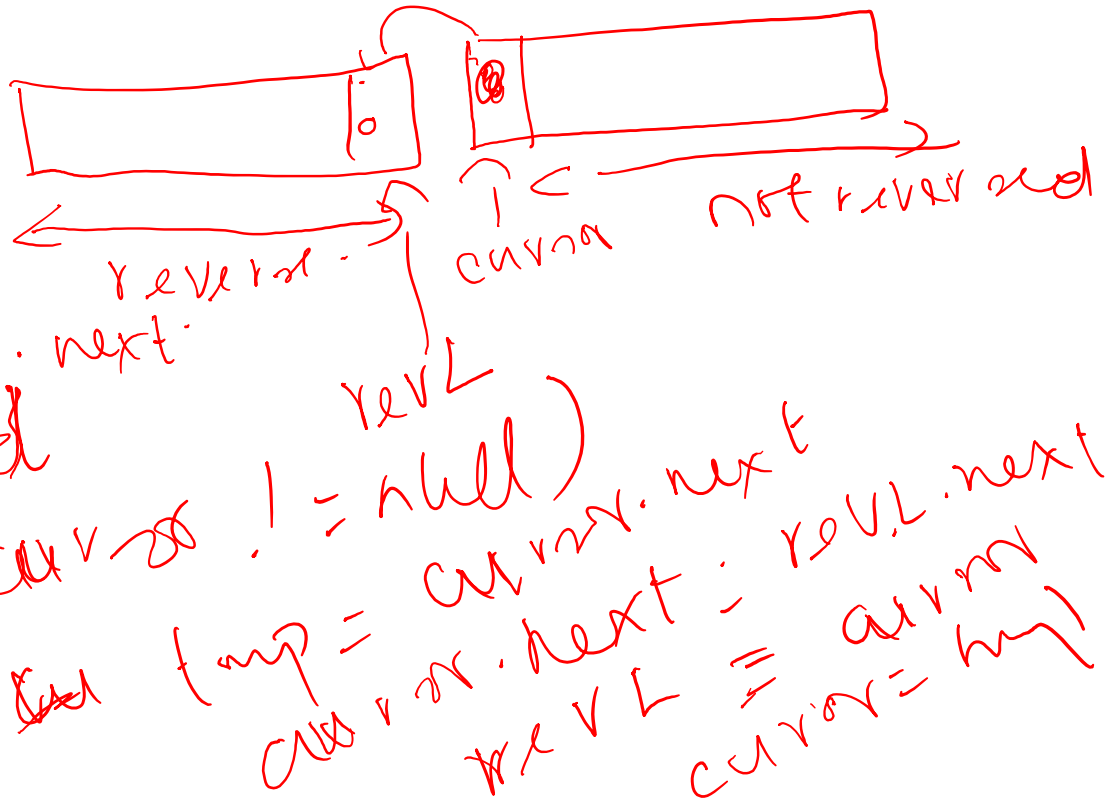
Stack

List in LIFO order – elements are added and removed form only one end (stack top)

- Operations:
  - common: push(), pop()
  - other: peek()
- Data Structures for stack:
  - ArrayList: where is top of the stack?
  - LinkedList: head points to top of the stack
- Implementations in Java:
  - Stack: `Stack<E> s = new Stack<>(); //old way`
  - ArrayDeque: `Deque<E> s = new ArrayDeque<>(); // preferred`
  - LinkedList: slow - memory needs to be allocated for entry during push

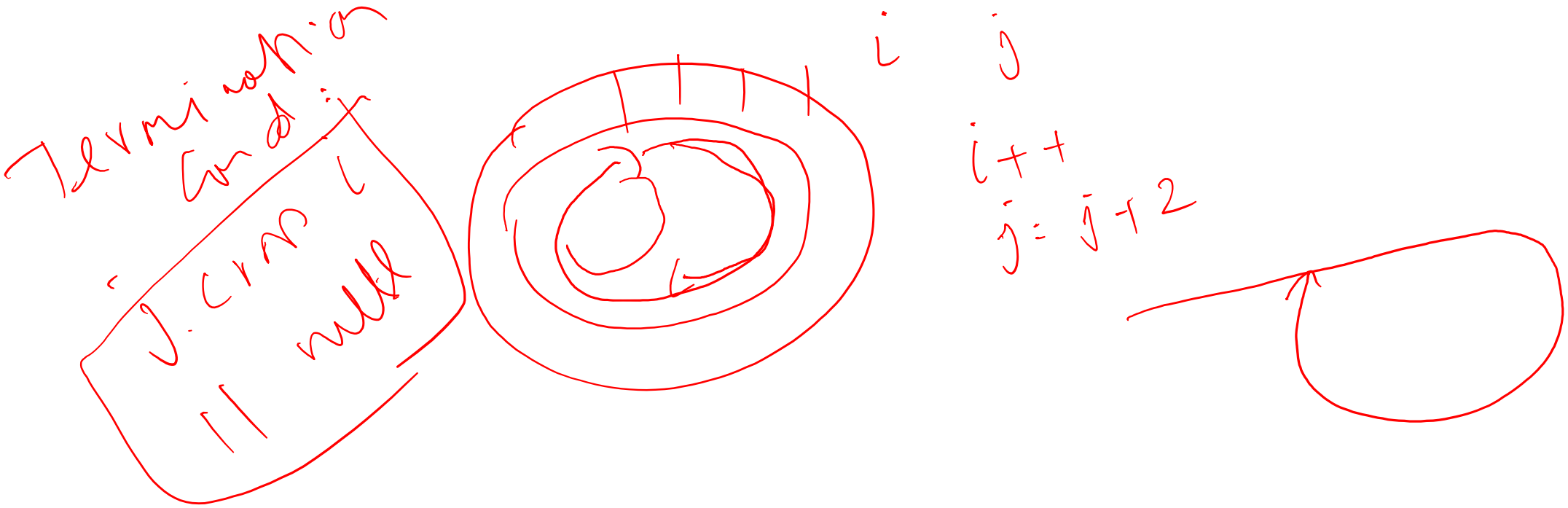
Problem of the Day #1

Write a method reverse() in SinglyLinkedList class. It should be non-recursive, cannot use any other data structure, and cannot create a node/entry.



Problem of the Day #2

Write a method to detect a cycle in a singly linked list.



Problem of the Day #3

Write a method to check whether a given singly linked list is a palindrome. Element type is character.