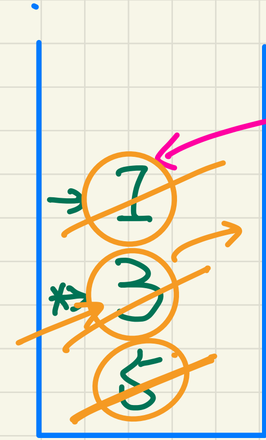
Lecture 3

Part B

Stack ADT -
Last In First Out (LIFO)
Implementations in Java

Stack ADT: Illustration

	isEmpty	size	top
new stack	T	0	n.a.
push(5)	F	1	5
push(3)	F	2	3
push(1)	F	3	1
pop rtn. 1	F	2	3
pop rtn. 3	F	1	5
pop rtn. 5	T	0	n.a.



last pushed element

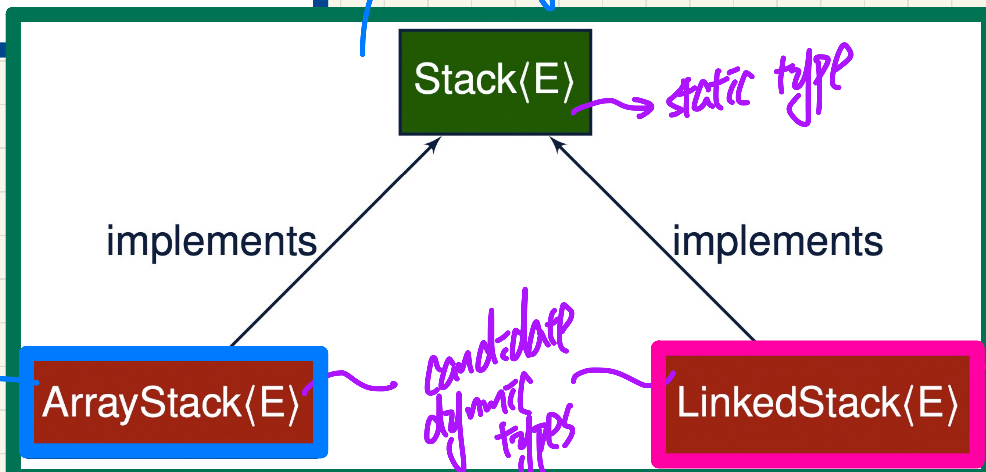
2nd last pushed element

(LIFO)

The order in which items are popped off the stack is the reverse of how these items were pushed.

Implementing the Stack ADT in Java: Architecture

```java
public interface Stack< E > {
  public int size();
  public boolean isEmpty();
  public E top();
  public void push( E e);
  public E pop();
}
```



1. polymorphism
2. dynamic binding

static type

candidate dynamic types

① all operations O(1)
② inflexible by a pre-set size

S2: pop: O(n) unless DLL

S1: all ops. O(1)

Implementing the Stack ADT using an Array

```java
public class ArrayStack<E> implements Stack<E> {
  private final int MAX_CAPACITY = 1000;
  private E[] data;
  private int t; /* index of top */
  public ArrayStack() {
    data = (E[]) new Object[MAX_CAPACITY];
    t = -1;
  }

  public int size() { return (t + 1); }
  public boolean isEmpty() { return (t == -1); }

  public E top() {
    if (isEmpty()) { /* Precondition Violated */ }
    else { return data[t]; }
  }
  public void push(E e) {
    if (size() == MAX_CAPACITY) { /* Precondition Violated */ }
    else { t ++; data[t] = e; }
  }
  public E pop() {
    E result;
    if (isEmpty()) { /* Precondition Violated */ }
    else { result = data[t]; data[t] = null; t --; }
    return result;
  }
}
```

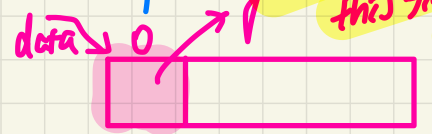
O(1)
O(1)
O(1)
O(1)
O(1)

limitation: fixed size

ArrayStack<String>
↳ instantiates E for Stack: Stack<String>

(E[]) Object[___]
↳ what you have to write in Java.

efficient impl. of stack



goal: treat this first item as the top.

Implementing the Stack ADT using a SLL

Improved to O(1) if a DLL is used.

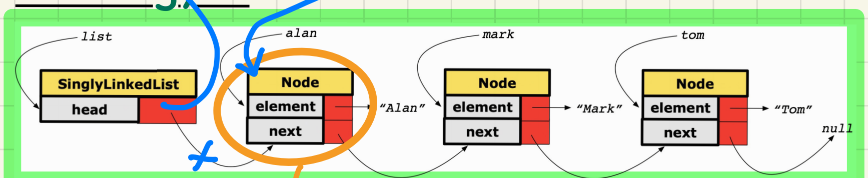
```
public class LinkedStack<E> implements Stack<E> {
  private SinglyLinkedList<E> list;
  ...
}
```

Stack Method	Singly-Linked List Method	
	Strategy 1	Strategy 2
size	list.size	list.size
isEmpty	list.isEmpty	list.isEmpty
top	list.first	list.last
push	list.addFirst	list.addLast
pop	list.removeFirst	list.removeLast

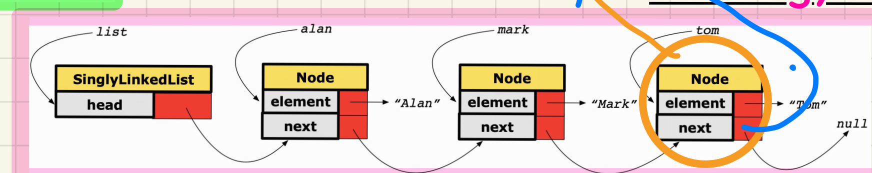
Strategy 1: list.removeFirst is O(1)

Strategy 2: list.removeLast is O(n)

Strategy 1

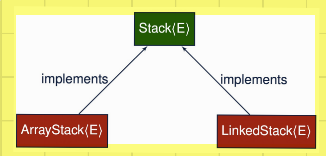


Strategy 2



Stack ADT: Testing Alternative Implementations

Stack<S.> s = new Stack<>(); ✗ → interface can't be a DT.




```java
@Test
public void testPolymorphicStacks() {
  Stack<String> s = new ArrayStack<>();
  s.push("Alan"); /* dynamic binding */
  s.push("Mark"); /* dynamic binding */
  s.push("Tom"); /* dynamic binding */
  assertTrue(s.size() == 3 && !s.isEmpty());
  assertEquals("Tom", s.top());
  s = new LinkedStack<>();
  s.push("Alan"); /* dynamic binding */
  s.push("Mark"); /* dynamic binding */
  s.push("Tom"); /* dynamic binding */
  assertTrue(s.size() == 3 && !s.isEmpty());
  assertEquals("Tom", s.top());
}
```

```java
public class ArrayStack<E> implements Stack<E> {
  private final int MAX_CAPACITY = 1000;
  private E[] data;
  private t; /* index of top */
  public ArrayStack() {
    data = (E[]) new Object[MAX_CAPACITY];
    t = -1;
  }

  public int size() { return (t + 1); }
  public boolean isEmpty() { return (t == -1); }

  public E top() {
    if (isEmpty()) { /* Precondition Violated */ }
    else { return data[t]; }
  }
  public void push(E e) {
    if (size() == MAX_CAPACITY) { /* Precondition Violated */ }
    else { t ++; data[t] = e; }
  }
  public E pop() {
    E result;
    if (isEmpty()) { /* Precondition Violated */ }
    else { result = data[t]; data[t] = null; t --; }
    return result;
  }
}
```

static type; dynamic type; DT: AS; DT: LS; version in AS.; version in LS.; DT changes; class of Stack?

Is the DT of s a descendant

	*	**
s instanceof Stack	T	T
s instanceof ArrayStack	T	F
s instanceof LinkedStack	F	T

Lecture 3

Part C

Stack ADT -
Algorithms using the Stack ADT

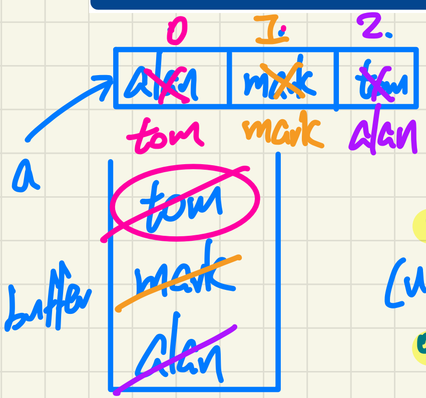
Algorithm using Stack: Reversing an Array

```java
public static <E> void reverse(E[] a) {
  Stack<E> buffer = new ArrayStack<E>();
  for (int i = 0; i < a.length; i ++) {
    buffer.push(a[i]);
  }
  for (int i = 0; i < a.length; i ++) {
    a[i] = buffer.pop();
  }
}
```

generic parameter at the method level

reverse({"alan", "mark"}) — String[]

reverse({23, 46})



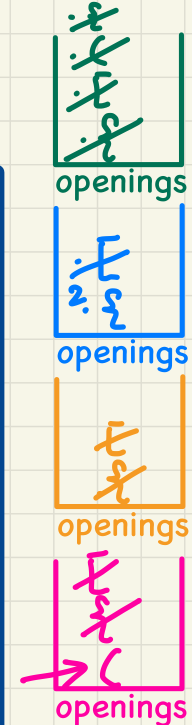

```java
@Test
public void testReverseViaStack() {
  String[] names = {"Alan", "Mark", "Tom"};
  String[] expectedReverseOfNames = {"Tom", "Mark", "Alan"};
  StackUtilities.reverse(names);
  assertArrayEquals(expectedReverseOfNames, names);

  Integer[] numbers = {46, 23, 68};
  Integer[] expectedReverseOfNumbers= {68, 23, 46};
  StackUtilities.reverse(numbers);
  assertArrayEquals(expectedReverseOfNumbers, numbers);
}
```

handling an array of strings (using a stack of strings)

handling an array of ints (using a stack of ints)

Algorithm using Stack: Matching Delimiters

```
public static boolean isMatched(String expression) {
  final String opening = "([{";
  final String closing = ")]}";
  Stack<Character> openings = new LinkedStack<Character>();
  int i = 0;
  boolean foundError = false;
  while (!foundError && i < expression.length()) {
    char c = expression.charAt(i);
    if(opening.indexOf(c) != -1) { openings.push(c); }
    else if (closing.indexOf(c) != -1) {
      if(openings.isEmpty()) { foundError = true; }
      else {
        if (opening.indexOf(openings.top()) == closing.indexOf(c)) {
          openings.pop();
        }
        else { foundError = true; }
      }
    }
    i ++;
  }
  return !foundError && openings.isEmpty();
}
```




```
@Test
public void testMatchingDelimiters() {
  assertTrue(StackUtilities.isMatched(""));
  assertTrue(StackUtilities.isMatched("{[]}({})"));
  assertFalse(StackUtilities.isMatched("{[])"));
  assertFalse(StackUtilities.isMatched("{[]})"));
  assertFalse(StackUtilities.isMatched("({[]}"));
}
```

Algorithm using Stack: Calculating Postfix Expressions

Sketch of Algorithm

- When input is an ***operand*** (i.e., a number), ***push*** it to the stack.
- When input is an ***operator***, obtain its two ***operands*** by ***popping*** off the stack **twice**, evaluate, then ***push*** the result back to stack.
- When finishing reading the input, there should be **only one** number left in the stack.

Input 1: 3 4 5 * - ≡ 3 - (4 * 5)

Input 2: 3 4 - 5 * ≡ (3 - 4) * 5

Input 3: 5 2 3 + * + ≡ + 5 * (2 + 3)

Input 4: 5 4 + 6 ≡ 5 + 4 6

5 + 4 = 20

(insufficient operator)

2 + 3 = 5

5 * 5 = 25

? + 25

(insufficient operands)

4 * 5 = 20

3 - 20 = -17

3 - 4 = -1

-1 * 5 = -5

Lecture 3

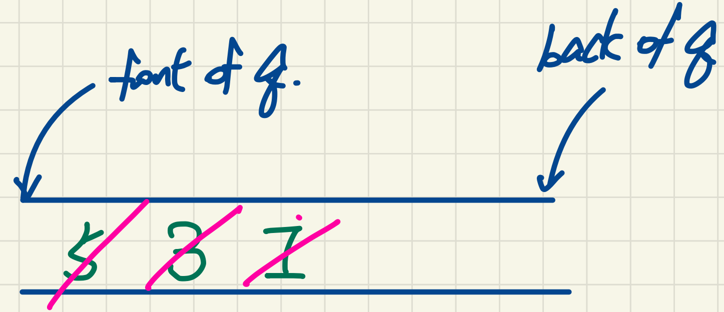
Part D

Queue ADT -
First In First Out (FIFO)
Implementations in Java

Queue ADT: Illustration

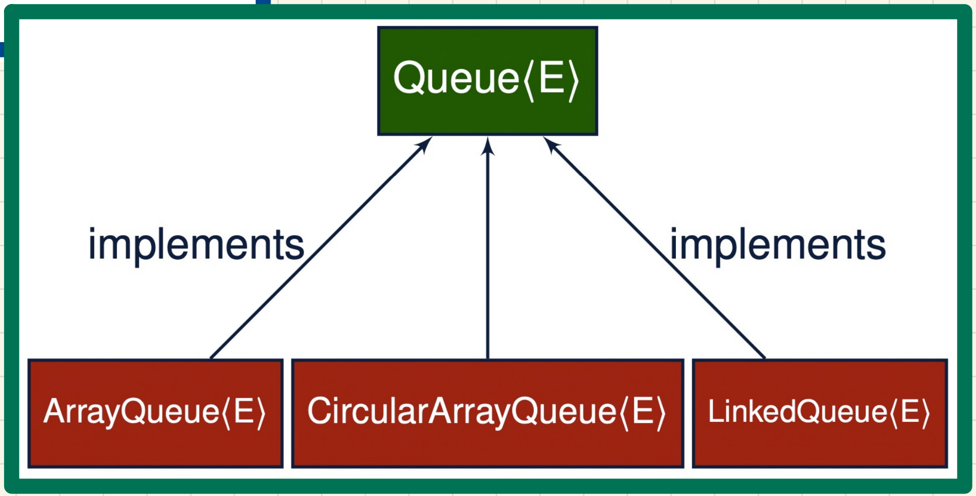
	isEmpty	size	first
new queue	T	0	n.a.
enqueue(5)	F	1	5
enqueue(3)	F	2	5
enqueue(1)	F	3	5
dequeue (rtn. 5)	F	2	3
dequeue (rtn. 3)	F	1	1
dequeue (rtn. 1)	T	0	n.a.

First-In First-Out (FIFO)



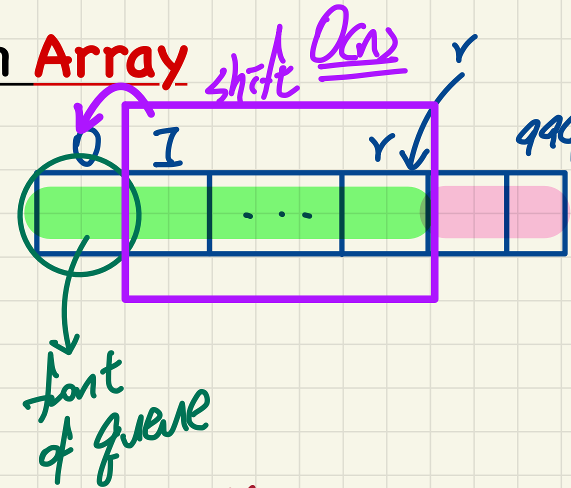
Implementing the Queue ADT in Java: Architecture

```java
public interface Queue< E > {
  public int size();
  public boolean isEmpty();
  public E first();
  public void enqueue( E e);
  public E dequeue();
}
```



Implementing the Queue ADT using an Array

```
public class ArrayQueue<E> implements Queue<E> {
  private final int MAX_CAPACITY = 1000;
  private E[] data;
  private int r; /* rear index */
  public ArrayQueue() {
    data = (E[]) new Object[MAX_CAPACITY];
    r = -1;
  }
  public int size() { return (r + 1); }
  public boolean isEmpty() { return (r == -1); }
  public E first() {
    if (isEmpty()) { /* Precondition Violated */ }
    else { return data[0]; }
  }
  public void enqueue(E e) {
    if (size() == MAX_CAPACITY) { /* Precondition Violated */ }
    else { r ++; data[r] = e; }
  }
  public E dequeue() {
    if (isEmpty()) { /* Precondition Violated */ }
    else {
      E result = data[0];
      for (int i = 0; i < r; i ++) { data[i] = data[i + 1]; }
      data[r] = null; r --;
      return result;
    }
  }
}
```



O(1) O(1) O(1) O(1) O(n)

front index

limitation: no resizing.

shifting "2nd item" and onwards to the left by one position

to improve this, we need to be flexible about where the front index is ⇒ circular array.

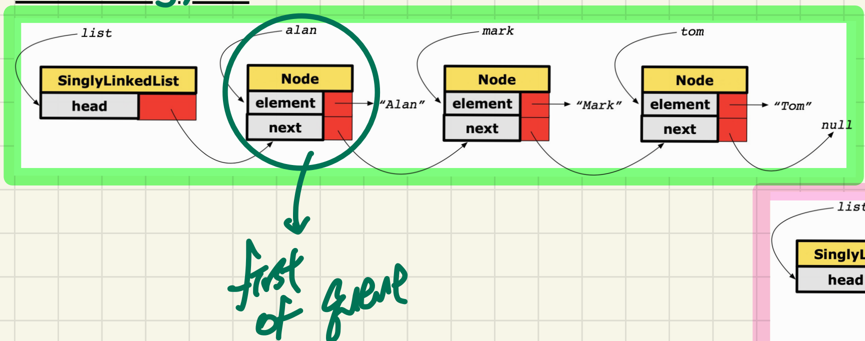
Implementing the Queue ADT using a SLL

```
public class LinkedQueue<E> implements Queue<E> {
  private SinglyLinkedList<E> list;
  ...
}
```

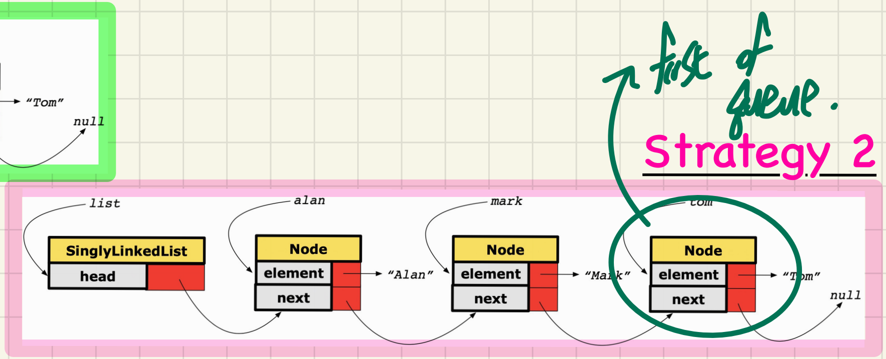
O(n)
① use SL instead
② use DLL instead

Queue Method	Singly-Linked List Method	
	Strategy 1	Strategy 2
size	list.size	
isEmpty	list.isEmpty	
first	list.first	list.last
enqueue	list.addLast	list.addFirst
dequeue	list.removeFirst	list.removeLast

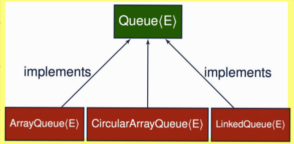
Strategy 1



Strategy 2



Stack ADT: Testing Alternative Implementations




```java
public class ArrayQueue<E> implements Queue<E> {
  private final int MAX_CAPACITY = 1000;
  private E[] data;
  private int r = -1; /* rear index */
  public ArrayQueue() {
    data = (E[]) new Object[MAX_CAPACITY];
    r = -1;
  }
  public int size() { return (r + 1); }
  public boolean isEmpty() { return (r == -1); }
  public E first() {
    if (isEmpty()) { /* Precondition Violated */ }
    else { return data[0]; }
  }
  public void enqueue(E e) {
    if (size() == MAX_CAPACITY) { /* Precondition Violated */ }
    else { r ++; data[r] = e; }
  }
  public E dequeue() {
    if (isEmpty()) { /* Precondition Violated */ }
    else {
      E result = data[0];
      for (int i = 0; i < r; i ++) { data[i] = data[i + 1]; }
      data[r] = null; r --;
      return result;
    }
  }
}
```

dynamic binding

```java
@Test
public void testPolymorphicQueues() {
  Queue<String> q = new ArrayQueue<>();
  q.enqueue("Alan"); /* dynamic binding */
  q.enqueue("Mark"); /* dynamic binding */
  q.enqueue("Tom"); /* dynamic binding */
  assertTrue(q.size() == 3 && !q.isEmpty());
  assertEquals("Alan", q.first());

  q = new LinkedQueue<>();
  q.enqueue("Alan"); /* dynamic binding */
  q.enqueue("Mark"); /* dynamic binding */
  q.enqueue("Tom"); /* dynamic binding */
  assertTrue(q.size() == 3 && !q.isEmpty());
  assertEquals("Alan", q.first());
}
```

Polymorphism