Data Structures and Algorithms
(ECEG 4171)

Chapter Three
Fundamental Data Structures

Ephrem A. (M.Sc.)
School of Electrical and Computer Engineeing
Chair of Computer Engineering
28th October, 2019

Abstract Data Types and Data Structures

- Oftentimes abstract data types (ADT) and data structures are used as synonymous. However, it's important to view them as follows:
  - An Abstract Data Type (ADT) is a data type whose representation is hidden from the client. It is a definition of new type, describes its properties and operations.
    - You can formally define (i.e., using mathematical logic) what an ADT is/does. *e.g.*, a Stack is a list implements a LIFO policy on additions/deletions.
  - A data structure is more concrete. Typically, it is a technique or strategy for implementing an ADT. *e.g.* Use a linked list or an array to implement a stack class.
- Some common ADTs that all trained programmers know about:
  - stack, queue, priority queue, dictionary, sequence, set
- Some common data structures used to implement those ADTS:
  - array, linked list, hash table (open, closed, circular hashing)
  - trees (binary search trees, heaps, AVL trees, 2-3 trees, tries, red/black trees, B-trees)

Data Structures Classification

- Data structures are classified into two types:
  1. Linear data structures: Elements are accessed in a sequential order but it is not compulsory to store all elements sequentially (say, Linked Lists). Examples: Linked Lists, Stacks and Queues.
  2. Non-linear data structures: Elements stored/accessed in a nonlinear order. Examples: Trees and graphs.

Abstract Data Types (ADTs)

- To specify the behavior of an ADT, we use an application programming interface (API), which is a list of constructors and instance methods (operations), with an informal description of the effect of each.
- **Example:**

public class	Counter	
	Counter(String id)	*Create a counter named id*
void	increment()	*increment the counter by one*
int	tally()	*number of increments since creation*
String	toString()	*string representation*

```
public class Flips {
  public static void main(String[] args){
    int T = Integer.parseInt(args[0]);
    Counter heads = new Counter("heads");
    Counter tails = new Counter("tails");
    for (int t = 0; t < T; t++)
      if (Math.random() > 0.5)
        heads.increment();
      else tails.increment();
    System.out.println(heads);
    System.out.println(tails);
  }
}
```

Autoboxing and Auto-unboxing

- Sometimes you want to treat a primitive like an object. This was prohibited prior Java 5.0.
- For example, you cannot put a primitive directly into a `Collection` like `ArrayList` in prior Java 5.0.

```
int x = 32;
ArrayList list = new ArrayList();
list.add(x);
```

this won't work before Java 5.0

- Autoboxing is the process by which a primitive type is automatically encapsulated (boxed) into its equivalent type wrapper whenever an object of that type is needed.
  - There is no need to explicitly construct an object. E.g.

```
Integer iOb = 100; // autobox an int. No object is explicitly created through the use of new
```

- Auto-unboxing is the process by which the value of a boxed object is automatically extracted (unboxed) from a type wrapper when its value is needed. E.g.

```
int i = iOb; // auto-unbox
```

Autoboxing and Auto-unboxing

- There's a wrapper class for every primitive type defined in java.lang package, no need to import them.

Primitive	Wrapper class
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

- Hence, after Java 5.0, this is possible

```
ArrayList<Integer> listOfnumbers = new ArrayList<Integer>();
listOfnumbers.add(3); // autoboxing
int x = listOfnumbers.get(0); // auto-unboxing
```

Introduction to Java Generics

- At its core the term generics means parameterized types.
- Parameterized types enable you to create classes, interfaces, and methods in which the type of data upon which they operate is specified as a parameter.
- Using generics, it is possible to create a single class that automatically works with different types of data.
- A class, interface, or method that operates on a parameterized type is called generic, as in *generic class* or *generic method*.
- Generics work only with objects and differ based on their type arguments.

A Simple Generics Example

```
class Gen<T> {
    T ob; // declare an object of type T
    Gen(T o) {
        ob = o;
    }
    T getob() {
        return ob;
    }
    void showType() {
        System.out.println("Type of T is " +
                ob.getClass().getName());
    }
}
```

```
class GenDemo {
    public static void main(String args[]) {
        Gen<Integer> iOb = new Gen<Integer>(88); //autoboxing
        iOb.showType();
        int v = iOb.getob(); //auto-unboxing
        System.out.println("value: " + v);
        System.out.println();
        Gen<String> strOb = new Gen<String>("Generics Test");
        strOb.showType();
        String str = strOb.getob();
        System.out.println("value: " + str);
    }
}
```

Output

```
Type of T is java.lang.Integer
value: 88

Type of T is java.lang.String
value: Generics Test
```

Here, **T** is the name of a type parameter.
Because Gen uses a type parameter, Gen is a generic class, which is also called a *parameterized type*.

How Generics Improve Type Safety

Prior to Java SE 5, generic programming was implemented by relying heavily on Java's **Object class**, which is the universal supertype of all objects.

Example

```
public class ObjectPair {
   Object first ;
   Object second;
   public ObjectPair(Object a, Object b) {
       first = a;
      second = b;
   }
   public Object getFirst () {return first ; }
   public Object getSecond() {return second;}
}
```

Declaration and instantiation:

```
ObjectPair bid = new ObjectPair("ORCL", 32.07);
```

However, the following is wrong:

```
String stock = bid. getFirst (); // compile error
String stock = (String) bid . getFirst (); // Correct
```

How Generics Improve Type Safety

Prior to Java SE 5, generic programming was implemented by relying heavily on Java's **Object class**, which is the universal supertype of all objects.

Example

```
public class ObjectPair {
   Object first ;
   Object second;
   public ObjectPair(Object a, Object b) {
       first = a;
      second = b;
   }
   public Object getFirst () {return first ; }
   public Object getSecond() {return second;}
}
```

Declaration and instantiation:

```
ObjectPair bid = new ObjectPair("ORCL", 32.07);
```

However, the following is wrong:

```
String stock = bid.getFirst (); // compile error
String stock = (String) bid.getFirst (); // Correct
```

Using Generics Framework

```
public class Pair<A,B> {
   A first ;
   B second;
   public Pair(A a, B b) {
       first = a;
      second = b;
   }
   public A getFirst () {return first ;}
   public B getSecond() {return second;}
}
```

Declaration and instantiation:

```
Pair<String,Double> bid = new Pair<>("ORCL", 32.07);
      //Typer inference. OR
Pair<String, Double> bid = new Pair<String,
      Double>("ORCL", 32.07);//Expilicit type
```

Generics and Arrays

1. Code outside a generic class may wish to declare an array storing instances of the generic class with actual type parameters.

```
Pair<String,Double>[ ] holdings;
holdings = new Pair<String, Double>[25]; // illegal ; compile error
holdings = new Pair[25]; // correct, but warning about unchecked cast
holdings[0] = new Pair<>("ORCL", 32.07); // valid element assignment
```

2. A generic class may wish to declare an array storing objects that belong to one of the formal parameter types.

```
public class Portfolio <T> {
   T[] data;
   public Portfolio (int capacity) {
      data = new T[capacity]; // illegal ; compiler error
      data = (T[]) new Object[capacity]; // legal, but compiler warning
   }
   public T get(int index) { return data[index]; }
   public void set(int index, T element) { data[index] = element; }
}
```

Generic Methods

To define generic methods, we include a generic formal type declaration among the method modifiers.

Example: a parameterized static method that can reverse an array containing elements of any object type.

```
public class GenericDemo {
    public static <T> void reverse(T[] data) {
        int low = 0, high = data.length - 1;
        while (low < high) { // swap data[low] and data[high]
            T temp = data[low];
            data[low++] = data[high]; // post-increment of low
            data[high--] = temp; // post-decrement of high
        }
    }
}
```

Bounded Generic Types

A formal parameter type can be restricted by using the extends keyword followed by a class or interface. In that case, only a type that satisfies the stated condition is allowed to substitute for the parameter.

```
public class MyAnimalList<T extends Animal> {
```

Introduction to Nested Classes

- In Java, it is possible to define a class within another class, such classes are known as *nested classes*.
- Nested classes enable you to logically group classes that are only used in one place.
- This increases the use of encapsulation and reduce name conflicts, and create more readable and maintainable code.
- **Syntax:**

```
class OuterClass{
    ....
    class NestedClass{
        ....
    }
}
```

Nested Class

- The scope of a nested class is bounded by the scope of its enclosing class. Thus, class `NestedClass` does not exist independently of class `OuterClass`.
- A nested class has access to the members, including private members, of the class in which it is nested. However, the reverse is not true i.e. the enclosing class does not have access to the members of the nested class.
- A nested class is also a member of its enclosing class.
- As a member of its enclosing class, a nested class can be declared private, public, protected, default.
- Nested classes are divided into two categories:
  - ▷ **static nested class:** Nested classes that are declared static
  - ▷ **inner class:** A non-static nested class.

Static Nested Classes

- As with class methods and variables, a static nested class is associated with its outer class.
- And like static class methods, a static nested class cannot refer directly to instance variables or methods defined in its enclosing class.
- It can use them only through an object reference.
- They are accessed using the enclosing class name.

```
OuterClass.StaticNestedClass
```

- For example, to create an object for the static nested class, use this syntax:

```
OuterClass.StaticNestedClass nestedObject = new OuterClass.StaticNestedClass();
```

Static Nested Class Example

```
class OuterClass{
    static int outer_x = 10;
    int outer_y = 20;
    private static int outer_private = 30;

    static class StaticNestedClass{
        void display(){
            //can access static member of outer class
            System.out.println("outer_x = " + outer_x);

            //can access private static member of outer class
            System.out.println("outer_private = " +
                    outer_private);

            //The following statement will give compilation error
            //as static nested class cannot directly access
                    non-static
            //System.out.println("outer_y = " + outer_y);
        }
    }
}
```

```
public class StaticNestedClassDemo{
    public static void main(String[] args){
        //accessing a static nested class
        OuterClass.StaticNestedClass
                nestedObject = new
                OuterClass.StaticNestedClass();
        nestedObject.display();
    }
}
```

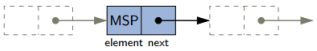
Output

outer_x = 10

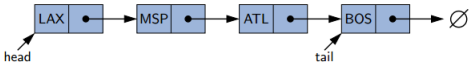
outer_private = 30

Linked List Data Structures

- A linked list is a collection of nodes that collectively form a linear sequence.
- In a singly linked list (SLL), each node stores a reference to an object that is an element of the sequence, as well as a reference to the next node of the list.



- The linked list instance must keep a reference to the first node of the list, known as the head. The last node of the list is known as the tail.
- We can identify the tail as the node having null as its next reference by *traversing* through the nodes but storing explicit reference to the tail node is more efficient.

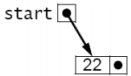


Consider the following node class

```
class Node {
    int data;
    Node next;
    Node(int data) {
        this.data = data;
    }
}
```

```
//Constructing a linked list.
Node start = new Node(22);
```

Notice that the Node class is now *self-referential*. Its *next* field is declared to have type Node.

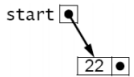


Consider the following node class

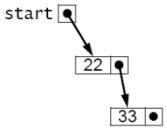
```
class Node {
    int data;
    Node next;
    Node(int data) {
        this.data = data;
    }
}
```

Notice that the Node class is now *self-referential*. Its *next* field is declared to have type Node.

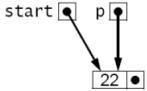
```
//Constructing a linked list.
Node start = new Node(22);
```



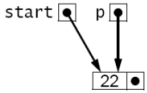

```
start.next = new Node(33);
```



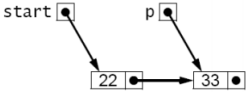
p = start = new Node(22);



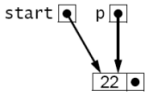
p = start = new Node(22);



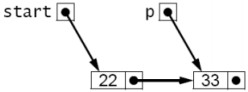
p = p.next = new Node(33);



p = start = new Node(22);

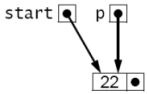


p = p.next = new Node(33);

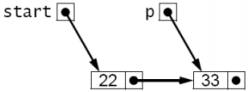


p = p.next = new Node(44);

p = start = new Node(22);



p = p.next = new Node(33);



p = p.next = new Node(44);

Using for loop to print a Linked List

```
for (Node p = start; p != null; p = p.next) {
    System.out.println (p.data);
}
```

The output is:

22
33
44

Implementing a Singly Linked List Class

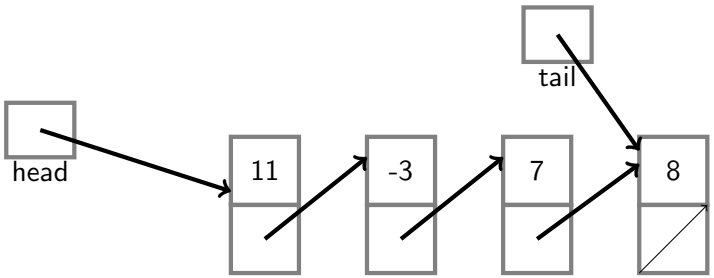
- A complete implementation of a Singly Linked List class, supports the following methods:

size():	Returns the number of elements in the list.
isEmpty():	Returns true if the list is empty, and false otherwise.
first():	Returns (but does not remove) the first element in the list.
last():	Returns (but does not remove) the last element in the list.
addFirst(e):	Adds a new element to the front of the list.
addLast(e):	Adds a new element to the end of the list.
removeFirst():	Removes and returns the first element of the list.

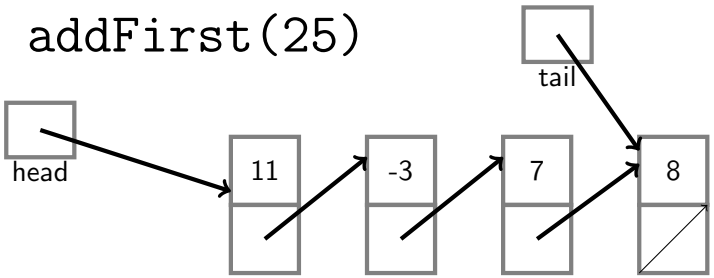
Implementing a SLL Class

```
public class SinglyLinkedList <E> {
   private static class Node<E> {
      private E element;
      private Node<E> next;
      public Node(E e, Node<E> n) {
         element = e;
         next = n;
      }
      public E getElement(){return element;}
      public Node<E> getNext(){return next;}
      public E getElement(E e)){ element = e;}
      public void setNext(Node<E> n){next = n;}
   } //-------- end of nested Node class
        --------
   private Node<E> head = null;
   private Node<E> tail = null;
   private int size = 0;
   public SinglyLinkedList () { }
   // access methods
   public int size () { return size ; }
   public boolean isEmpty() { return size == 0; }
   public E first () {
      if (isEmpty()) return null ;
      return head.getElement();
   }
   public E last () {
      if (isEmpty()) return null ;
      return tail .getElement();
   }
}
```

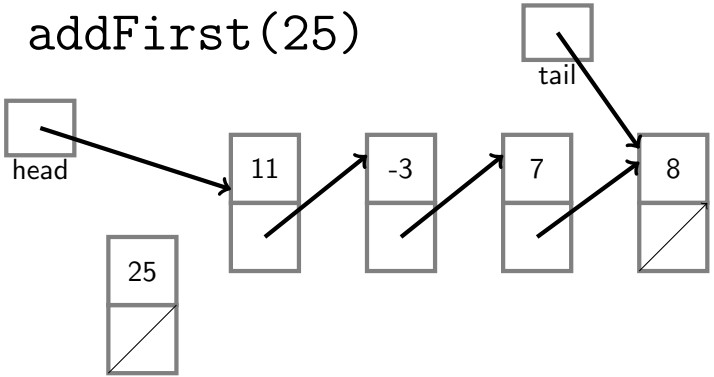
Inserting an Element at the Head of SLL



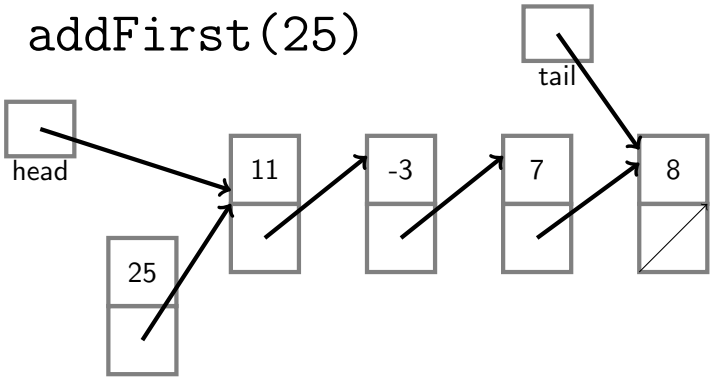
Inserting an Element at the Head of SLL



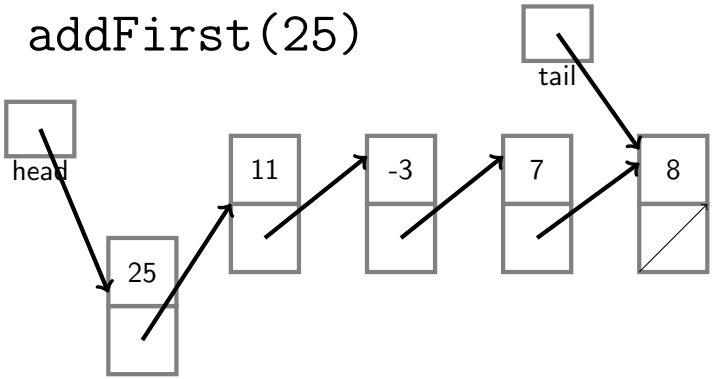
Inserting an Element at the Head of SLL



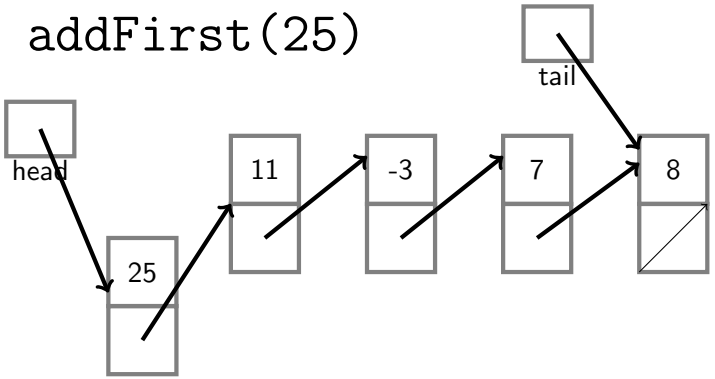
Inserting an Element at the Head of SLL



Inserting an Element at the Head of SLL



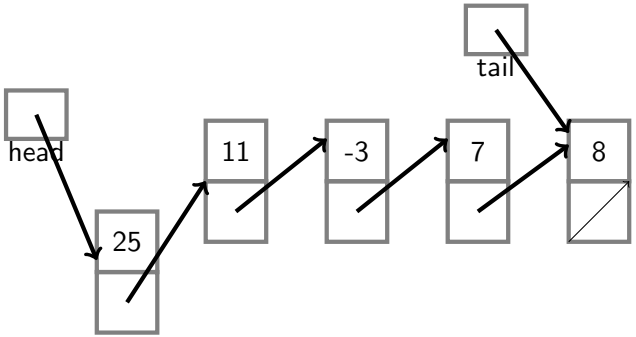
Inserting an Element at the Head of SLL



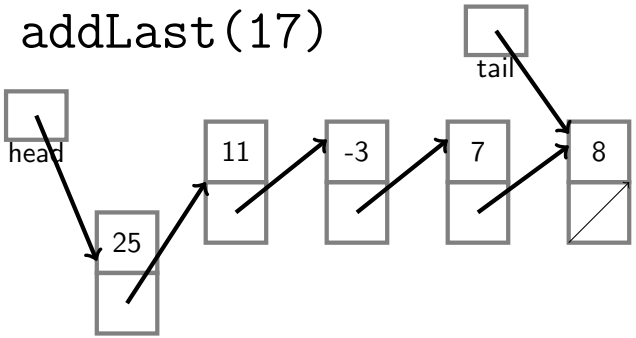

```
public void addFirst(E e) {
    head = new Node<>(e , head );
    if (size == 0)
        tail = head ;
    size ++;
}
```

Running time: $O(1)$

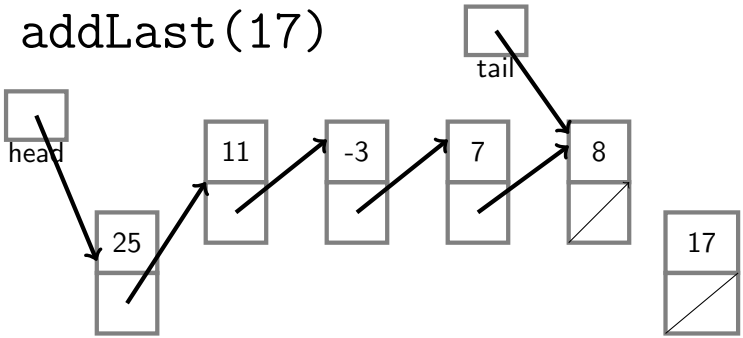
Inserting an Element at the Tail of a SLL



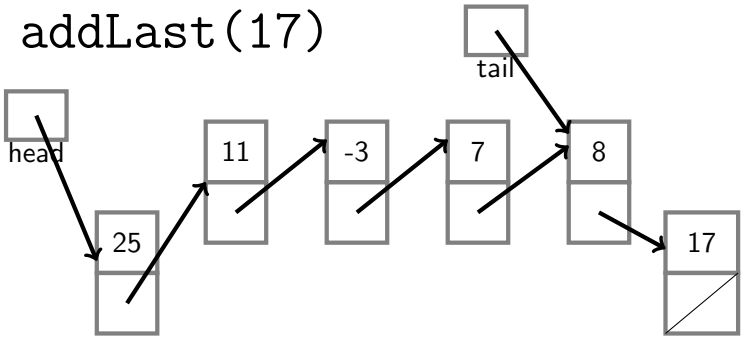
Inserting an Element at the Tail of a SLL



Inserting an Element at the Tail of a SLL

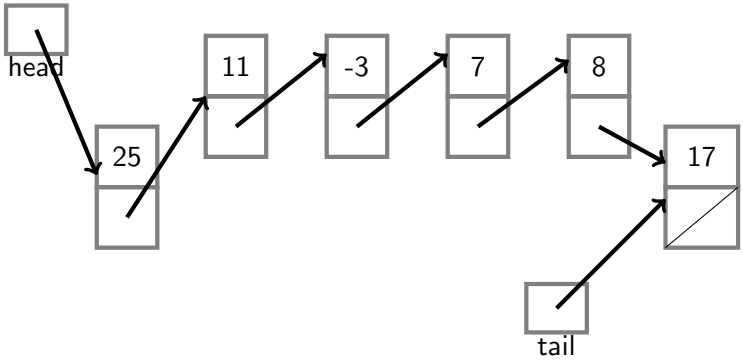


Inserting an Element at the Tail of a SLL



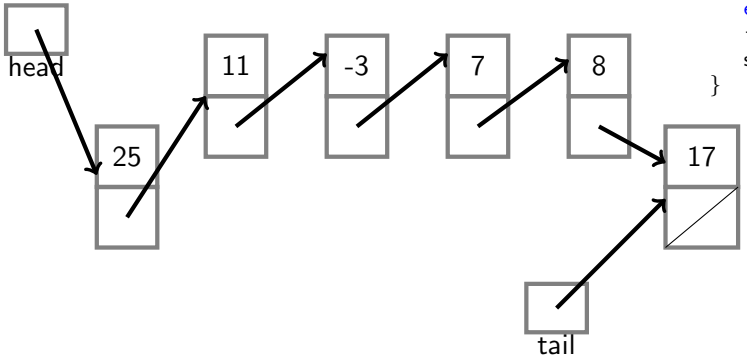
Inserting an Element at the Tail of a SLL

`addLast(17)`



Inserting an Element at the Tail of a SLL

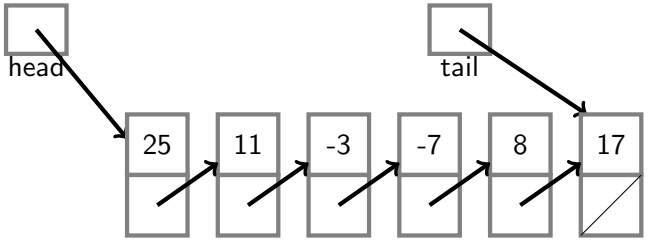
addLast(17)




```
public void addLast(E e) {
    Node<E> newest = new Node<>(e, null);
    if (isEmpty()) head = newest;
    else tail .setNext(newest);
    tail = newest;
    size++;
}
```

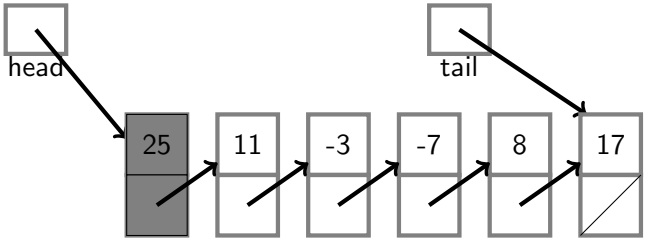
Running Time: O(1)

Removing an Element from a SLL



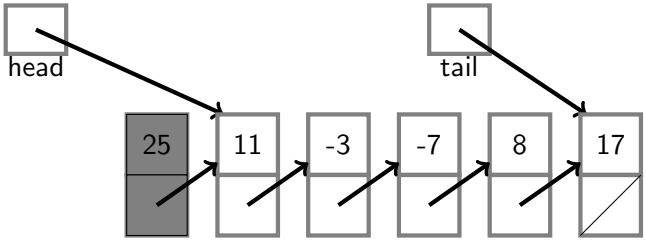
Removing an Element from a SLL

`removeFirst()`



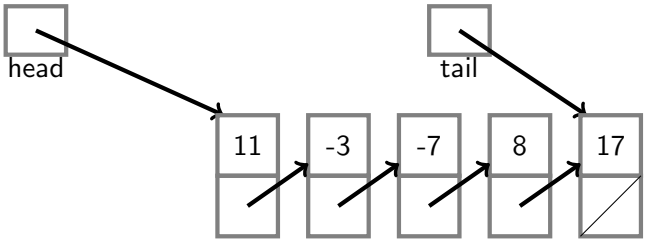
Removing an Element from a SLL

`removeFirst()`



Removing an Element from a SLL

removeFirst()

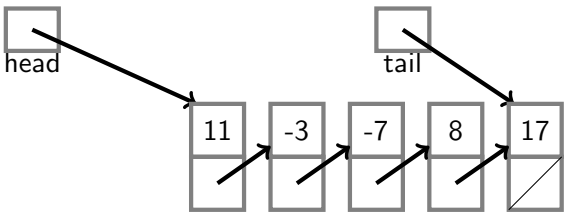



```
public E removeFirst() {
    if (isEmpty()) return null;
    E answer = head.getElement();
    head = head.getNext();
    size--;
    if (size == 0)
        tail = null;
    return answer;
}
```

Running Time: O(1)

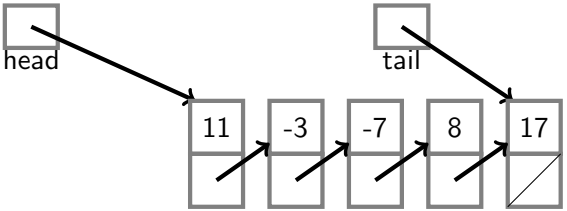
Removing at the Tail

- Removing at the tail of a singly linked list is not efficient ($O(n)$).
- There is no constant-time way to update the tail to point to the previous node.



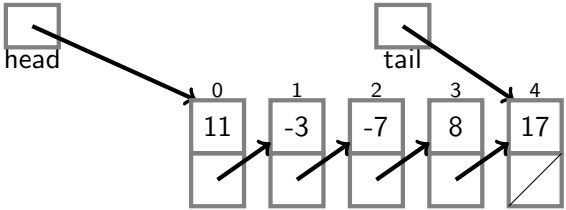
Removing at the Tail

- Removing at the tail of a singly linked list is not efficient ($O(n)$).
- There is no constant-time way to update the tail to point to the previous node.



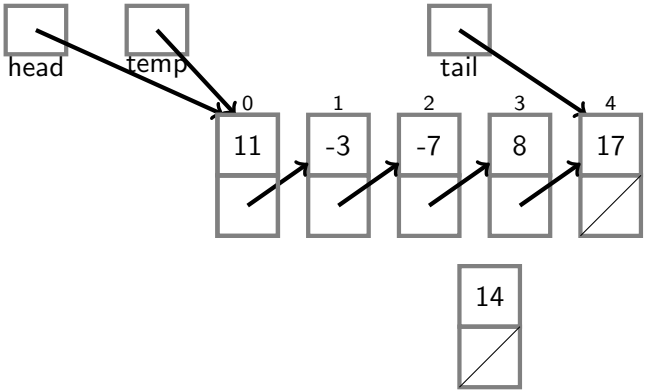

```
public E removeLast() {
    if (isEmpty()) return null;
    if (size == 1){
        E answer = head.getElement();
        head = tail = null;
        size--;
        return answer;
    }
    Node<E> temp = head;
    while(temp.getNext() != tail)
      temp = temp.getNext();
    temp.setNext(null);
    tail = temp;
    E answer = temp.getElement();
    size--;
    return answer;
}
```

Inserting an Element at Position p



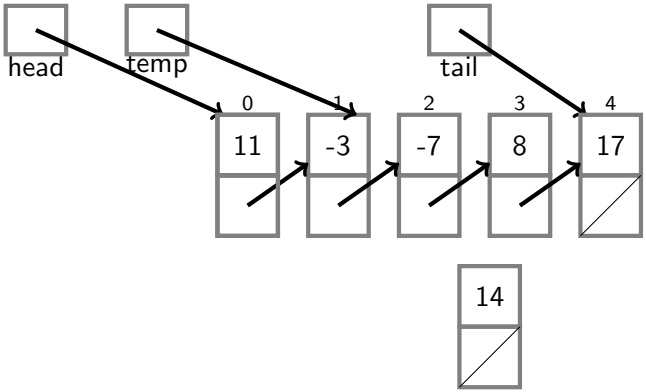
Inserting an Element at Position p

```
insert(14, 3)
```



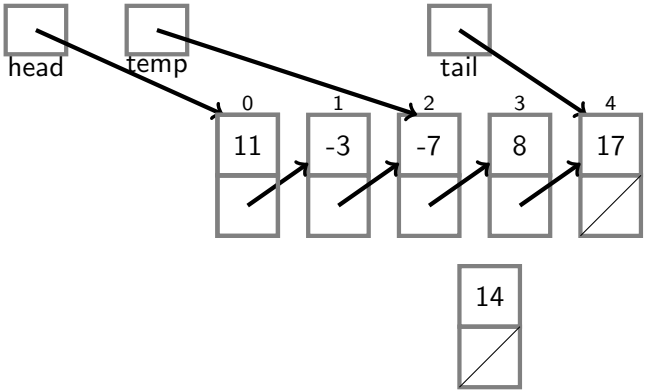
Inserting an Element at Position p

```
insert(14, 3)
```



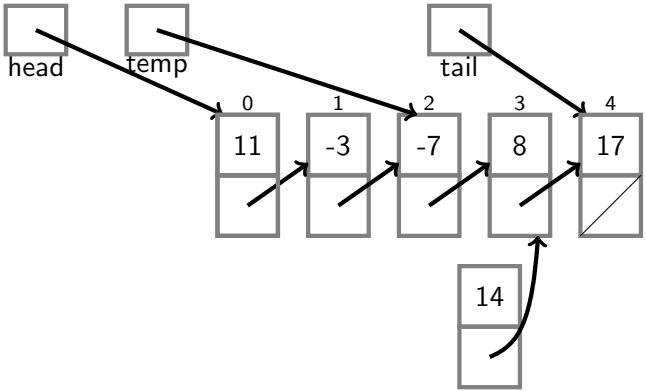
Inserting an Element at Position p

```
insert(14, 3)
```



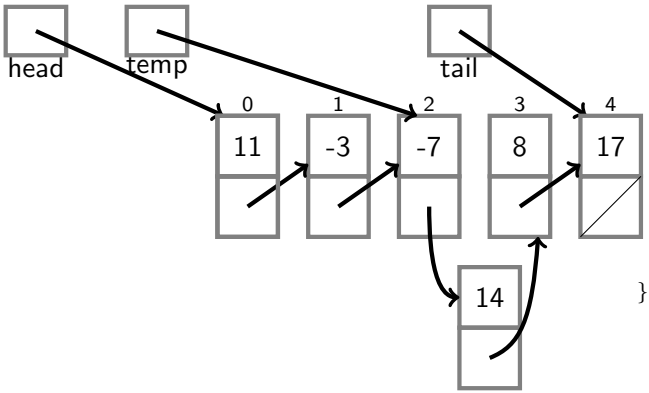
Inserting an Element at Position p

```
insert(14, 3)
```



Inserting an Element at Position p

insert(14, 3)




```
public void insert(E e, int p)
  throws IllegalArgumentException{
    if(p < 0 || p > size()-1)
      throw new IllegalArgumentException("Error msg");
    if(p==0) addFirst(e);
    else{
      Node<E> n = new Node<>(e, null);
      Node<E> temp = head;
      for(int i = 0; i < p-1; i++)
        temp = temp.next;
      n.next = temp.next;
      temp.next = n;
      size++;
    }
}
```

Removing an Element at Position p

Removing an Element at Position p

```
public E remove(int p) throws IllegalArgumentException{
    if (p < 0 || p > size()-1)
        throw new IllegalArgumentException(" Illegal  position ");
    if (p == 0) return removeFirst();
    else {
        Node<E> temp = head;
        for (int i = 0; i < p-1; i++)
            temp = temp.next;
            E ans = temp.next.element;
            //Node<E> n = temp.next;
            temp.next = temp.next.next;
            size--;
            return ans;
    }
}
```

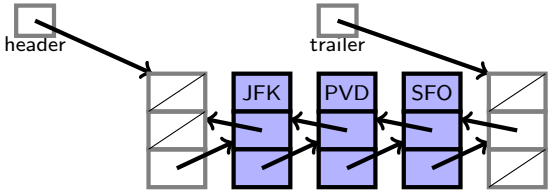
SLL Performance Summary

Operation	Running Time
`addFirst()`	$O(1)$
`first()`	$O(1)$
`removeFirst()`	$O(1)$
`addLast()`	$O(1)$
`last()`	$O(1)$
`removeLast()`	$O(n)$
`insert(e, p)`	$O(n)$
`remove(p)`	$O(n)$

Doubly Linked Lists

- In SLL, we cannot efficiently delete an arbitrary node from an interior position of the list if only given a reference to that node
  - We cannot determine the node that immediately precedes the node to be deleted.
- In Doubly Linked Lists (DLLs), each node keeps an explicit reference to the node before it and a reference to the node after it.
- Insertions and deletions at arbitrary positions within the list are done in $O(1)$-time.
- Special "dummy" nodes:
  - A *header* node at the beginning of the node.
  - A *trailer* node at the end of the node
- These special nodes (**sentinels**) do not store elements of the primary sequence.

OR



Implementing a DLL Class

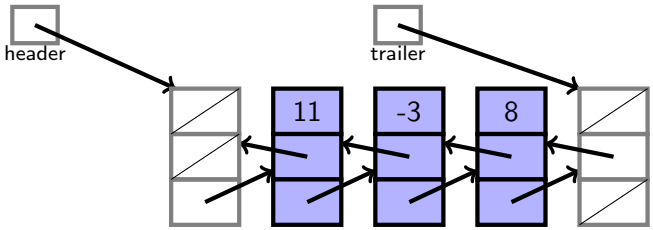
A complete implementation of a Doubly Linked List class, supports the following methods:

size():	Returns the number of elements in the list.
isEmpty():	Returns true if the list is empty, and false otherwise.
first():	Returns (but does not remove) the first element in the list.
last():	Returns (but does not remove) the last element in the list.
addFirst(e):	Adds a new element to the front of the list.
addLast(e):	Adds a new element to the end of the list.
removeFirst():	Removes and returns the first element of the list.
removeLast():	Removes and returns the first element of the list.

Implementing a DLL Class

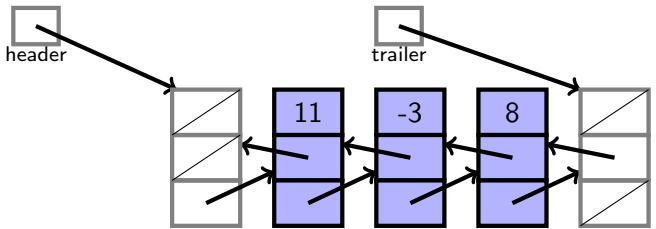
```
public class DoublyLinkedList<E>{
   private static class Node<E>{
      private E element;
      private Node<E> prev;
      private Node<E> next;
      public Node(E e, Node<E> p, Node<E> n){
         element = e;
         prev = p;
         next = n;
      }
      public E getElement() { return element; }
      public Node<E> getPrev() { return prev; }
      public Node<E> getNext() { return next; }
      public void setPrev(Node<E> p) { prev = p;
           }
      public void setNext(Node<E> n) { next = n;
           }
   } /*---End of Nested Class----*/
   private Node<E> header; // header sentinel
   private Node<E> trailer; // trailer sentinel
   private int size = 0;
   public DoublyLinkedList() {
      header = new Node<>(null, null, null);
       trailer = new Node<>(null, header, null);
      header.setNext( trailer ); // header is
            followed by trailer
   }
   public int size() { return size; }
   public boolean isEmpty() { return size == 0; }

   public E first () {
      if (isEmpty()) return null;
      return header.getNext().getElement();
   }
   public E last () {
      if (isEmpty()) return null;
      return trailer .getPrev().getElement();
   }
}
```

Inserting with a DLL



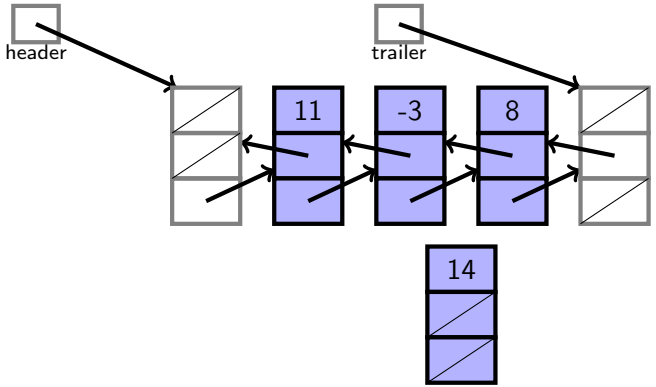
Inserting with a DLL

Insert 14 b/n -3 and 8



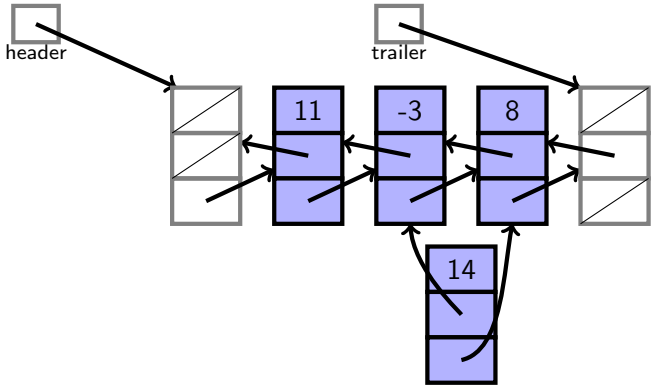
Inserting with a DLL

Insert 14 b/n -3 and 8



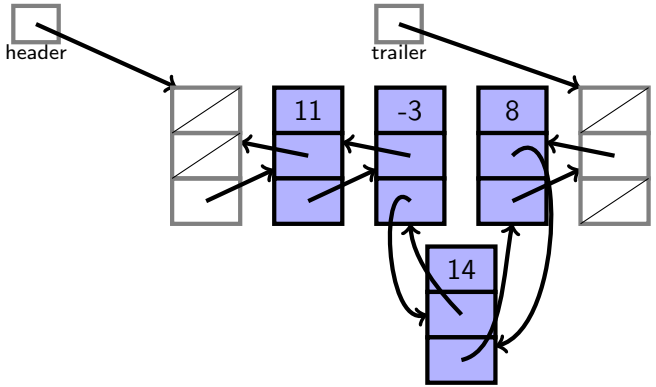
Inserting with a DLL

Insert 14 b/n -3 and 8



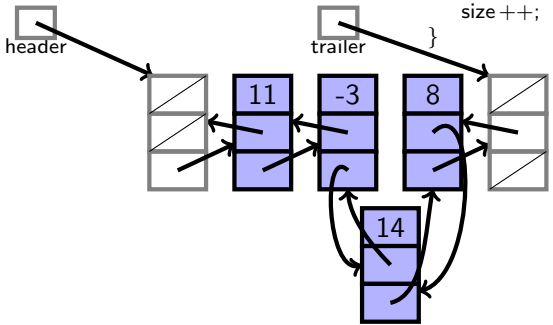
Inserting with a DLL

Insert 14 b/n -3 and 8



Inserting with a DLL

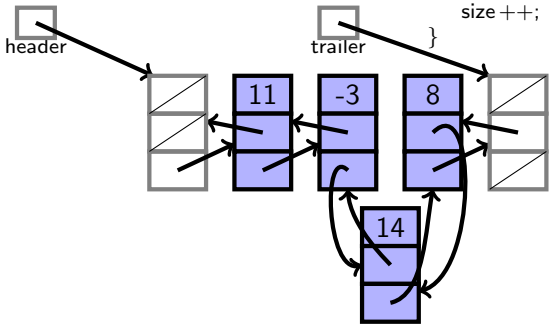
Insert 14 b/n -3 and 8




```
private void addBetween(E e, Node<E> predecessor,
                        Node<E> successor) {
    Node<E> newest = new Node<>(e, predecessor, successor);
    predecessor.setNext(newest);
    successor.setPrev(newest);
    size++;
}
```

Inserting with a DLL

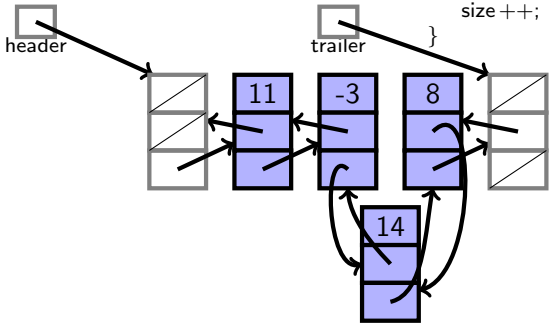
Insert 14 b/n -3 and 8




```
private void addBetween(E e, Node<E> predecessor,
                        Node<E> successor) {
    Node<E> newest = new Node<>(e, predecessor, successor);
    predecessor.setNext(newest);
    successor.setPrev(newest);
    size++;
}

// To add an element to the front of the sequence
public void addFirst(E e) {
    addBetween(e, header, header.getNext());
}
```

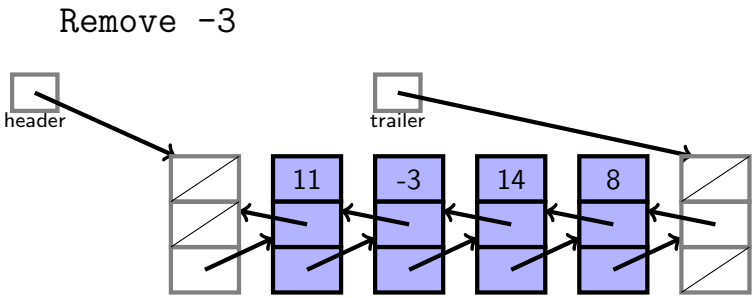
Inserting with a DLL

Insert 14 b/n -3 and 8



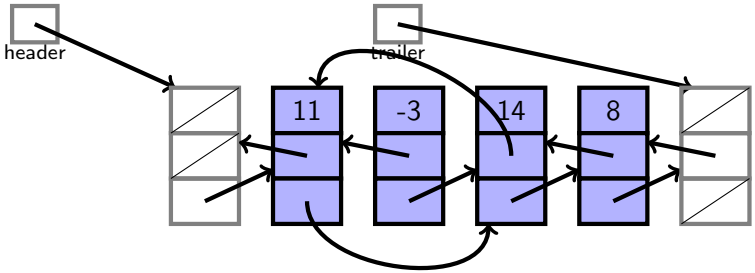

```
private void addBetween(E e, Node<E> predecessor,
                        Node<E> successor) {
    Node<E> newest = new Node<>(e, predecessor, successor);
    predecessor.setNext(newest);
    successor.setPrev(newest);
    size++;
}

// To add an element to the front of the sequence
public void addFirst(E e) {
    addBetween(e, header, header.getNext());
}

// To add an element to the end of the sequence
public void addLast(E e) {
    addBetween(e, trailer.getPrev(), trailer);
}
```

Deleting with a DLL

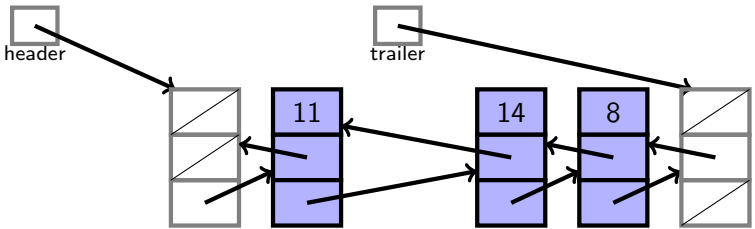


Deleting with a DLL

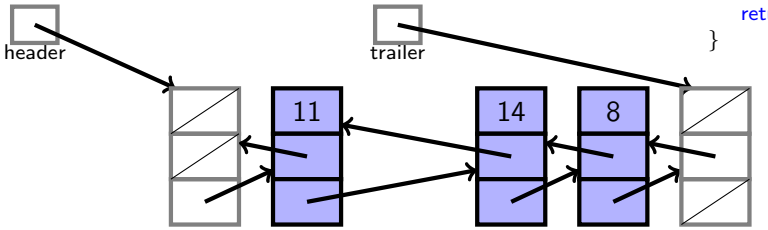
Remove -3



Deleting with a DLL



Deleting with a DLL




```
private E remove(Node<E> node) {
    Node<E> predecessor = node.getPrev();
    Node<E> successor = node.getNext();
    predecessor.setNext(successor);
    successor.setPrev(predecessor);
    size--;
    return node.getElement();
}
```

```
// To remove front element in list
public E removeFirst() {
    if (isEmpty()) return null;
    return remove(header.getNext());
}
```

```
// To remove last element in list
public E removeLast() {
    if (isEmpty()) return null;
    return remove(trailer.getPrev());
}
```

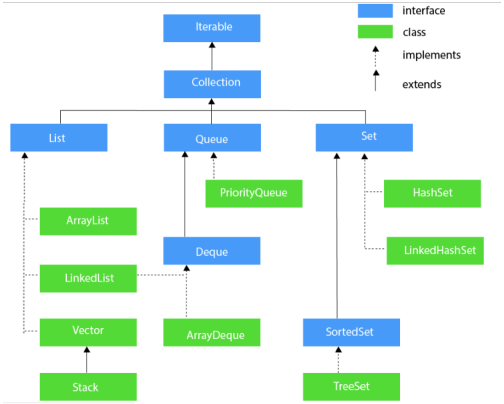
DLL Performance Summary

Operation	Running Time
`addFirst()`	$O(1)$
`first()`	$O(1)$
`removeFirst()`	$O(1)$
`addLast()`	$O(1)$
`last()`	$O(1)$
`removeLast()`	~~$O(n)$~~ $O(1)$
`addBetween(e, pred, succ)`	~~$O(n)$~~ $O(1)$
`remove(node)`	~~$O(n)$~~ $O(1)$

- Disadvantages of DLL:
  1. Each node requires an extra pointer, requiring more space ($O(n)$ memory waste).
  2. Insertion or deletion of a node takes a bit longer (more pointer operations).

The Java API for Linked List

- The **Java Collections Framework (JCF)** is a group of classes and interfaces in the `java.util` package.



The Java API for Linked List

- In Java, `LinkedList` class a doubly linked list implementation of the `List` interface.
- Constructors for `LinkedList`
  1. `LinkedList()`: to create an empty linked list
  2. `LinkedList(Collection C`: to create ordered list which contains all the elements of a collection.
- Example

```
import java.util.LinkedList;
public class Test{
    public static void main(String[] args){
        LinkedList<String> linked = new LinkedList<>();
        linked.add("A");
        linked.addLast("B");
        linked.addFirst("D");
        linked.add(2, "E");
        linked.addFirst("F");
        linked.remove("B");
        linked.remove(3);
        linked.removeLast();
        linked.removeFirst();
        System.out.println(linked); // Output: [D]
    }
}
```

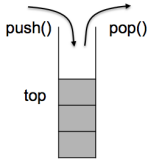
Example: Describe an algorithm for concatenating two singly linked lists L and M, into a single list L' that contains all the nodes of L followed by all the nodes of M.

Example: Describe an algorithm for concatenating two singly linked lists L and M, into a single list L' that contains all the nodes of L followed by all the nodes of M.

```
public SinglyLinkedList merge(SinglyLinkedList s){
    if(s == null) return this;
    SinglyLinkedList merged = new SinglyLinkedList();
    merged.head = head;
    tail.setNext(s.head);
    merged.tail = s.tail;
    merged.size = size + s.size;
   return merged;
}
```

Stack

- A **stack** is a collection of objects that are inserted and removed according to the last-in, first-out (LIFO) principle.
- Anything added to the stack goes on the top of the stack
- Anything removed from the stack is taken from the top of the stack



- Stacks are used in many applications:
  - Internet Web browsers store the addresses of recently visited sites on a stack.
  - Text editors usually provide an ündo̎mechanism that cancels recent editing operations and reverts to former states of a document.
  - In compilers.

Stacks

Formally, a stack is an abstract data type (ADT) that supports the following two methods:

push(e): Insert element e, to be the top of the stack.

pop(): Remove from the stack and return the top element on the stack; returns **null** if the stack is empty.

Additionally, let us also define the following methods:

size(): Return the number of elements in the stack.

isEmpty(): Return a Boolean indicating if the stack is empty.

top(): Return the top element in the stack, without removing it; returns **null** if the stack is empty.

Stacks

Start with an empty stack S.

Method	Return Value	Stack Contents
push(5)	–	(5)
push(3)	–	(5, 3)
size()	2	(5, 3)
pop()	3	(5)
isEmpty()	false	(5)
pop()	5	()
isEmpty()	true	()
pop()	null	()
push(7)	–	(7)
push(9)	–	(7, 9)
top()	9	(7, 9)
push(4)	–	(7, 9, 4)
size()	3	(7, 9, 4)
pop()	4	(7, 9)
push(6)	–	(7, 9, 6)
push(8)	–	(7, 9, 6, 8)
pop()	8	(7, 9, 6)

A Stack Interface in Java

- Our Implementation vs Java.util.Stack Class

Our Stack ADT	Class Java.util.Stack
`size()`	`size()`
`isEmpty()`	`empty()`
`push(e)`	`push(e)`
`pop()`	`pop()`
`top()`	`peek()`

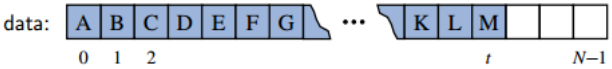
- Methods pop and peek of the java.util.Stack class throw a custom EmptyStackException if called when the stack is empty whereas **null** is returned in our abstraction.

A Stack Interface in Java

```
public interface Stack<E> {
  //Returns the number of elements in the stack.
  int size();
  //Tests whether the stack is empty.
  boolean isEmpty();
  // Inserts an element at the top of the stack.
  void push(E e);
  //Returns, but does not remove, the element at the top of the stack.
  E top();
  //Removes and returns the top element from the stack.
  E pop();
}
```

A Simple Array-Based Stack Implementation

- We store elements in an array, named data, with capacity N for some fixed N.
- The bottom element of the stack is always stored in cell data[0], and the top element of the stack in cell data[t] for index t that is equal to one less than the current size of the stack.

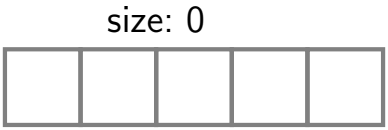


- When the stack holds elements from data[0] to data[t] inclusive, it has size t + 1.
- When the stack is empty it will have t equal to -1 (and thus has size t + 1, which is 0).

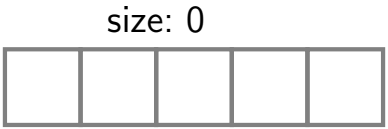
A Simple Array-Based Stack Implementation

```
public class ArrayStack<E> implements Stack<E>
    {
  private static final int CAPACITY=1000;
  private E[] data;
  private int t = -1;
  public ArrayStack() { this(CAPACITY); }
  public ArrayStack(int capacity) {
    data = (E[]) new Object[capacity];
  }
  public int size() { return (t + 1); }
  public boolean isEmpty() { return (t == -1); }
  public void push(E e) throws
      IllegalStateException {
    if (size() == data.length) throw new
        IllegalStateException("Stack is full");
    data[++t] = e;
  }
  public E top() {
    if (isEmpty()) return null;
    return data[t];
  }
  public E pop() {
    if (isEmpty()) return null;
    E answer = data[t];
    data[t] = null; // dereference to help
        garbage collection
    t--;
    return answer;
  }
}
```

Array Based Stack Implementation: Example

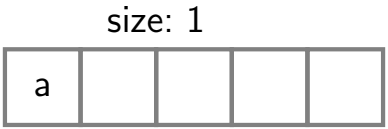


Array Based Stack Implementation: Example



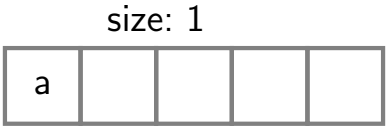
`push(a)`

Array Based Stack Implementation: Example

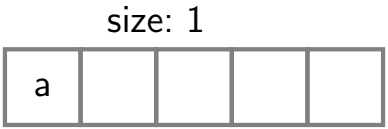


`push(a)`

Array Based Stack Implementation: Example

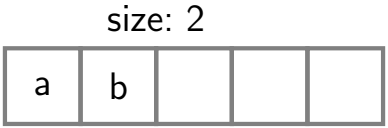


Array Based Stack Implementation: Example



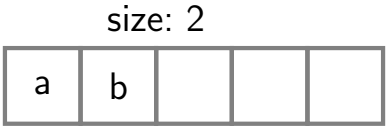
`push(b)`

Array Based Stack Implementation: Example

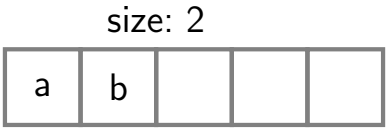


`push(b)`

Array Based Stack Implementation: Example

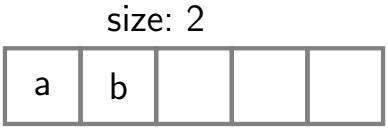


Array Based Stack Implementation: Example



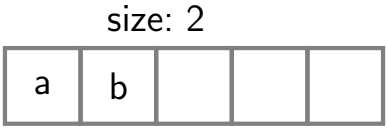
`top()`

Array Based Stack Implementation: Example

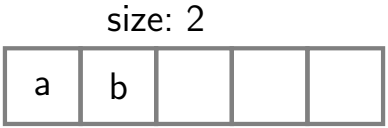


`top()` $\rightarrow$ `b`

Array Based Stack Implementation: Example

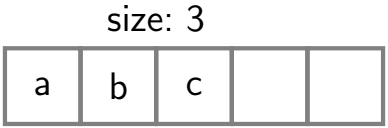


Array Based Stack Implementation: Example



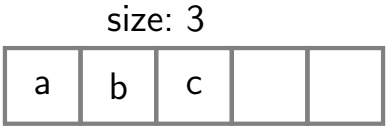
`push(c)`

Array Based Stack Implementation: Example

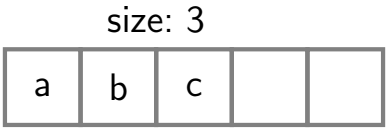


`push(c)`

Array Based Stack Implementation: Example

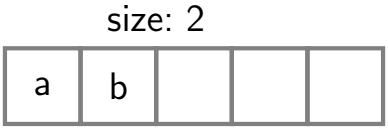


Array Based Stack Implementation: Example



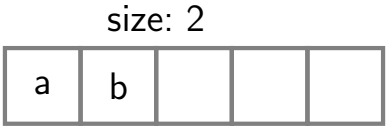
pop()

Array Based Stack Implementation: Example

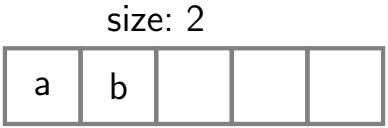


$\texttt{pop()} \rightarrow \texttt{c}$

Array Based Stack Implementation: Example

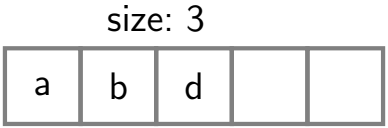


Array Based Stack Implementation: Example



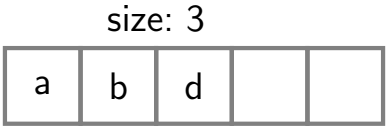
`push(d)`

Array Based Stack Implementation: Example

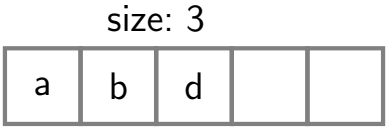


`push(d)`

Array Based Stack Implementation: Example

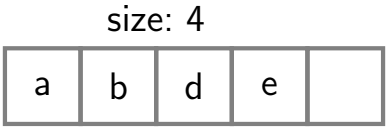


Array Based Stack Implementation: Example



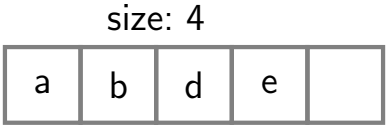
`push(e)`

Array Based Stack Implementation: Example

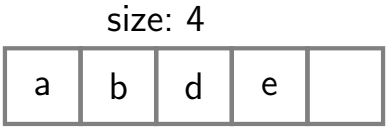


`push(e)`

Array Based Stack Implementation: Example

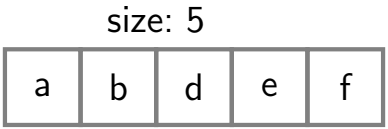


Array Based Stack Implementation: Example



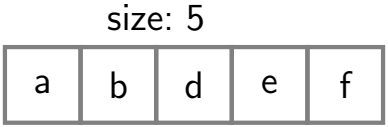

```
push(f)
```

Array Based Stack Implementation: Example




```
push(f)
```

Array Based Stack Implementation: Example

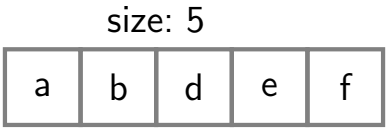


Array Based Stack Implementation: Example

size: 5

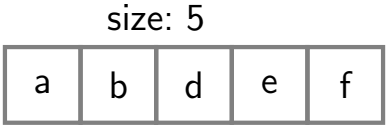
`push(g)`

Array Based Stack Implementation: Example

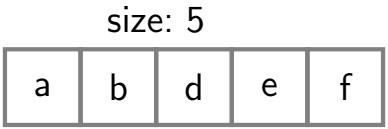


$$\texttt{push(g)} \rightarrow \texttt{ERROR}$$

Array Based Stack Implementation: Example

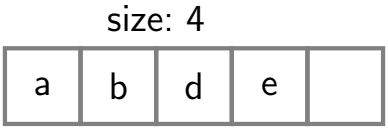


Array Based Stack Implementation: Example



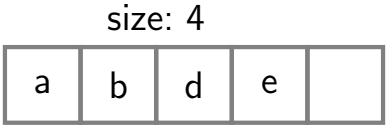
pop()

Array Based Stack Implementation: Example

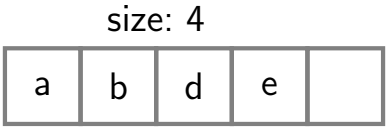


$$\texttt{pop()} \rightarrow \texttt{f}$$

Array Based Stack Implementation: Example

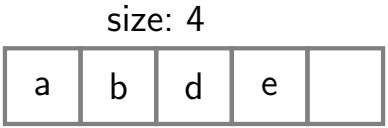


Array Based Stack Implementation: Example



`isEmpty()`

Array Based Stack Implementation: Example



`isEmpty()` $\rightarrow$ `false`

Example

```
Stack<Integer> S = new ArrayStack<>();
S.push(5);
S.push(3);
System.out. println (S. size ());
System.out. println (S.pop());
System.out. println (S.isEmpty());
System.out. println (S.pop());
System.out. println (S.isEmpty());
System.out. println (S.pop());
S.push(7);
S.push(9);
System.out. println (S.top());
S.push(4);
System.out. println (S. size ());
System.out. println (S.pop());
S.push(6);
S.push(8);
System.out. println (S.pop());
```

Example

```
Stack<Integer> S = new ArrayStack<>(); // contents: ()
S.push(5); // contents: (5)
S.push(3); // contents: (5, 3)
System.out.println(S.size()); //contents:(5, 3) outputs 2
System.out.println(S.pop()); // contents:(5) outputs 3
System.out.println(S.isEmpty()); //contents:(5) outputs false
System.out.println(S.pop());//contents:() outputs 5
System.out.println(S.isEmpty());//contents:() outputs true
System.out.println(S.pop()); //contents:() outputs null
S.push(7); // contents: (7)
S.push(9); // contents: (7, 9)
System.out.println(S.top()); //contents:(7, 9) outputs 9
S.push(4); // contents: (7, 9, 4)
System.out.println(S.size()); //contents:(7, 9, 4) outputs 3
System.out.println(S.pop()); //contents:(7, 9) outputs 4
S.push(6); //contents:(7, 9, 6)
S.push(8); //contents:(7, 9, 6, 8)
System.out.println(S.pop()); //contents:(7, 9, 6) outputs 8
```

Implementing a Stack with a Singly Linked List

- One limitation of the array-based stack implementation is it relies on a fixed-capacity array, which limits the ultimate size of the stack.
- The linked-list approach has memory usage that is always proportional to the number of actual elements currently in the stack, and without an arbitrary capacity limit.
- Since we can insert and delete elements in constant time only at the front, the top of the stack should be at the front.

Stack Method	Singly Linked List Method
`size()`	`list.size()`
`isEmpty()`	`list.isEmpty()`
`push(e)`	`list.addFirst(e)`
`pop()`	`list.removeFirst()`
`top()`	`list.first()`

Implementing a Stack with a SLL

```
public class LinkedStack<E> implements Stack<E> {

    //an empty list
    private SinglyLinkedList <E> list = new SinglyLinkedList<>();

    //new stack relies on the initially empty list
    public LinkedStack() { }
    public int size() { return list.size(); }
    public boolean isEmpty() { return list.isEmpty(); }
    public void push(E element) { list.addFirst(element); }
    public E top() { return list.first(); }
    public E pop() { return list.removeFirst(); }
}
```

Example: Reversing an Array Using a Stack

Example: Reversing an Array Using a Stack

```
/** A generic method for reversing an array. */
public static <E> void reverse(E[] a) {
   Stack<E> buffer = new ArrayStack<>(a.length);
   for (int i=0; i < a.length; i++)
      buffer.push(a[i]);
   for (int i=0; i < a.length; i++)
      a[i] = buffer.pop();
}
```

Example: Reversing an Array Using a Stack

```
/** A generic method for reversing an array. */
public static <E> void reverse(E[] a) {
    Stack<E> buffer = new ArrayStack<>(a.length);
    for (int i=0; i < a.length; i++)
        buffer.push(a[i]);
    for (int i=0; i < a.length; i++)
        a[i] = buffer.pop();
}

/** Tester routine for reversing arrays */
public static void main(String args[]) {
    Integer[] a = {4, 8, 15, 16, 23, 42}; // autoboxing allows this
    String[] s = {"Jack", "Kate", "Hurley", "Jin", "Michael"};
    System.out.println("a = " + Arrays.toString(a));
    System.out.println("s = " + Arrays.toString(s));
    System.out.println("Reversing ...");
    reverse(a);
    reverse(s);
    System.out.println("a = " + Arrays.toString(a));
    System.out.println("s = " + Arrays.toString(s));
}
```

Output:

```
a = [4, 8, 15, 16, 23, 42]
s = [Jack, Kate, Hurley, Jin, Michael]
Reversing...
a = [42, 23, 16, 15, 8, 4]
s = [Michael, Jin, Hurley, Kate, Jack]
```

Example: Matching Parentheses

- We consider arithmetic expressions that may contain various pairs of grouping symbols, such as:
  - Parentheses: "(" and ")"
  - Braces: "{" and "}"
  - Brackets: "[" and "]"
- Each opening symbol must match its corresponding closing symbol. For example, $[(5 + x) - (y + z)]$.

Example: Matching Parentheses

- We consider arithmetic expressions that may contain various pairs of grouping symbols, such as:
  - Parentheses: "(" and ")"
  - Braces: "{" and "}"
  - Brackets: "[" and "]"
- Each opening symbol must match its corresponding closing symbol. For example, $[(5 + x) - (y + z)]$.

```
public static boolean isMatched(String expression) {
    final String opening = "({[";  // opening delimiters
    final String closing = ")}]";  // respective closing delimiters
    Stack<Character> buffer = new LinkedStack<>();
    for (char c : expression.toCharArray()) {
        if (opening.indexOf(c) != -1)  // this is a left delimiter
            buffer.push(c);
        else if (closing.indexOf(c) != -1) { // this is a right delimiter
            if (buffer.isEmpty())  // nothing to match with
                return false;
            if (closing.indexOf(c) != opening.indexOf(buffer.pop()))
                return false;  // mismatched delimiter
        }
    }
    return buffer.isEmpty();  // were all opening delimiters matched?
}
```

The JCF Stack Class

- Prior to the development of the JCF, Java extended the `Vector` class to a `Stack` class.
- But that class is now considered obsolete because it isn't consistent with the JCF.
- Instead, the Java API recommends using the `ArrayDeque` class for stacks, like this:

```
Deque<String> stack = new ArrayDeque<String>();
```

- For example

```
import java.util.Queue;
import java.ArrayDequeue;
public class TestStringStack {
   public static void main(String[] args) {
      Dequeue<String> st = new ArrayDeque<String>();
      queue.push("GB");
      queue.push("DE");
      System.out.println(st);
      System.out.println("st.element(): " + st.element());
      System.out.println("st.pop(): " + st.pop());
      System.out.println(st);
      System.out.println("st.pop(): " + st.pop());
      queue.add("IE");
      System.out.println(st);
   }
}
```

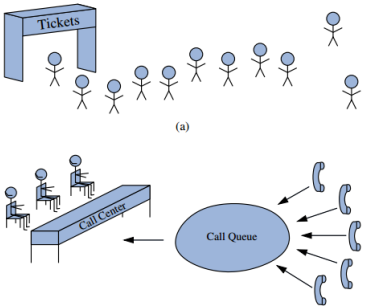
Example: Give a recursive method of removing all elements in a Stack

Example: Give a recursive method of removing all elements in a Stack

```
public void remove(Stack S){
    if ( S.isEmpty())
    return ;
    else {
        S.pop();
        remove(S);
    }
}
```

Queues

- A Queue is a data structure similar to stack.
- Objects are inserted and removed according to the first-in-first-out (FIFO) principle.
- Elements enter the queue at the **rear** and leave at the **front**



The Queue ADT

- A queue ADT supports the following two fundamental methods:

  `enqueue(e):` Insert element e at the rear of the queue.

  `dequeue():` Removes and returns the first element from the queue (or null if the queue is empty).

- It also includes the following supporting methods:

  `first():` Returns the first element of the queue, without removing it (or null if the queue is empty).

  `size():` Return the number of objects in the queue.

  `isEmpty():` Return a Boolean value that indicates whether the queue is empty.

Queue Interface in Java

```
public interface Queue<E> {
    int size(); // Returns number of elements in queue
    boolean isEmpty(); // Returns true if the queue is empty, false
        otherwise
    void enqueue(E e); // Inserts element e into the end of queue
    E first(); // Returns but doesn't remove the element at front of the
        queue
    E dequeue(); // Removes and returns the element at the end of the
        queue.
}
```

Queue operations and their effects

Method	Return Value	first $\leftarrow Q \leftarrow$ last
enqueue(5)	–	(5)
enqueue(3)	–	(5, 3)
size()	2	(5, 3)
dequeue()	5	(3)
isEmpty()	false	(3)
dequeue()	3	()
isEmpty()	true	()
dequeue()	null	()
enqueue(7)	–	(7)
enqueue(9)	–	(7, 9)
first()	7	(7, 9)
enqueue(4)	–	(7, 9, 4)

Queue operations and their effects

Method	Return Value	first ← Q ← last
enqueue(5)	–	(5)
enqueue(3)	–	(5, 3)
size()	2	(5, 3)
dequeue()	5	(3)
isEmpty()	false	(3)
dequeue()	3	()
isEmpty()	true	()
dequeue()	null	()
enqueue(7)	–	(7)
enqueue(9)	–	(7, 9)
first()	7	(7, 9)
enqueue(4)	–	(7, 9, 4)

The java.util.Queue Interface in Java

Our Queue ADT	Interface java.util.Queue	
	throws exceptions	returns special value
enqueue(e)	add(e)	offer(e)
dequeue()	remove()	poll()
first()	element()	peek()
size()	size()	
isEmpty()	isEmpty()	

Array-Based Queue Implementation

- We store them in an array such that the first element is at index 0, the second element at index 1, and so on.

- To implement the dequeue operation:
  - replace a dequeued element in the array with a null reference,
  - maintain an explicit variable f to represent the index of the element that is currently at the front of the queue.

- **Challenge!** How do you store upto N elements, when the back of the queue reaches the end of the array?

Using an Array Circularly

- Assuming the array has fixed length N, new elements are enqueued toward the ëndöf the current queue, progressing from the front to index N-1 and continuing at index 0, 1, ...

- Then, when we dequeue an element and want to advance the front index (f), we use

$$f = (f + 1)\%N$$

- And we enequeue an element at position $(f + sz)\%N$, where sz is the current number of elements in the queue.

A Java Queue Implementation

- The queue class maintains the following three instance variables:

  data: a reference to the underlying array.

  f: an integer that represents the index, within array data, of the first element of the queue (assuming the queue is not empty).

  sz: an integer representing the current number of elements stored in the queue (not to be confused with the length of the array).

Array-Based Queue Implementation

```
public class ArrayQueue<E> implements
    Queue<E> {
  private static final int CAPACITY=1000;
  private E[ ] data; // generic array used for
      storage
  private int f = 0; // index of the front
      element
  private int sz = 0; // current number of
      elements
  public ArrayQueue() {this(CAPACITY);}
  public ArrayQueue(int capacity) {
    data = (E[ ]) new Object[capacity];
  }
  public int size() { return sz; }
  public boolean isEmpty() { return (sz == 0); }
```

```
public void enqueue(E e) throws
    IllegalStateException {
  if (sz == data.length) throw new
      IllegalStateException ("Queue is full");
  int avail = (f + sz) % data.length; // use
      modular arithmetic
  data[avail] = e;
  sz++;
}
public E first () {
  if (isEmpty()) return null;
  return data[f];
}
public E dequeue() {
  if (isEmpty()) return null;
  E answer = data[f];
  data[f] = null; // dereference to help
      garbage collection
  f = (f + 1) % data.length;
  sz--;
  return answer;
}
```

Implementing a Queue with a Singly Linked List

- Without limit any artificial limit on the capacity, we implement the queue ADT using SLL while supporting worst-case $O(1)$-time for all operations.
- We align the front of the queue with the front of the list, and the back of the queue with the tail of the list.
  - Insert in the tail and delete from the front with $O(1)$-time.

```
public class LinkedQueue<E> implements Queue<E> {
    private SinglyLinkedList <E> list = new SinglyLinkedList<>();
    public LinkedQueue() { }
    public int size() { return list.size(); }
    public boolean isEmpty() { return list.isEmpty(); }
    public void enqueue(E element) { list.addLast(element); }
    public E first() { return list.first(); }
    public E dequeue() { return list.removeFirst(); }
}
```

The JCF Queue Class

- The Java Collections Framework includes a `Queue` interface, which is implemented by four classes: the `LinkedList` class, the AbstractQueue class, the `PriorityQueue` class, and the `ArrayDeque` class.
- For simple FIFO queues, the `ArrayDeque` class is the best choice:

```
Queue<String> queue = new ArrayDequeue<String>();
```

OR

```
Queue<String> queue = new LinkedList<String>();
```

- For example

```
import java.util.Queue;
import java.ArrayDequeue;
public class TestStringStack {
    public static void main(String[] args) {
        Queue<String> queue = new ArrayDeque<String>();
        queue.add("GB");
        queue.add("DE");
        System.out.println(queue);
        System.out.println("queue.element(): " + queue.element());
        System.out.println("queue.remove(): " + queue.remove());
        System.out.println(queue);
    }
}
```

Example: Write a method that reverses the elements of a queue using a stack

Example: Write a method that reverses the elements of a queue using a stack

```
public void reverse() {
    Stack<E> S = new ArrayStack<E>(size());
    while (!isEmpty())
      S.push(dequeue());
   while (!S.isEmpty())
      enqueue(S.pop());
}
```

Exercises

1. Suppose that a client performs an intermixed sequence of stack push and pop operations. The push operations push the integers 0 through 9 in order on to the stack; the pop operations print out the return value. Which of the following sequences could not occur?
   a) 4 6 8 7 5 3 2 9 0 1
   b) 2 5 6 7 4 8 9 3 1 0
2. Suppose that a client performs an intermixed sequence of (queue) enqueue and dequeue operations. The enqueue operations put the integers 0 through 9 in order onto the queue; the dequeue operations print out the return value. Which of the following sequence(s) could not occur?
   a) 0 1 2 3 4 5 6 7 8 9
   b) 4 6 8 7 5 3 2 9 0 1
3. Write a method `public static <E> E secondElement(Stack<E> S)` which returns the second element in the specified stack, leaving the stack in its original state.
4. Write a method `public static <E> E secondElement(Queue<E> Q)` which returns the second element in the specified queue, leaving the queue in its original state.