Algorithms and Data Structures for Object-Oriented Design

Linear Abstract Data Types

Instructor: Pirathayini Srikantha

Western University

January 21, 2019

Readings/References

- Goodrich (3.2, 6.1,6.2)
- Standish (2)

Table of Contents

1. Introduction to Data Structures
2. Linked Lists
3. Queues
4. Stacks
5. List
6. Iterators
7. Java Collections Framework

Table of Contents

1. Introduction to Data Structures
2. Linked Lists
3. Queues
4. Stacks
5. List
6. Iterators
7. Java Collections Framework

Abstract Data Types

- General purpose intermediate level data structures serve as **building blocks** for applications
- **Abstract Data Types (ADT)** define methods to package intermediate level data structures and operations on these
- These are **popular** techniques that have been carefully studied by many
  - Have **precisely defined behaviour** that can be used for clear reasoning of properties of programs that use these ADTs
- ADTs have **simple** and **clean** interfaces
- Low-level implementations are typically **hidden**

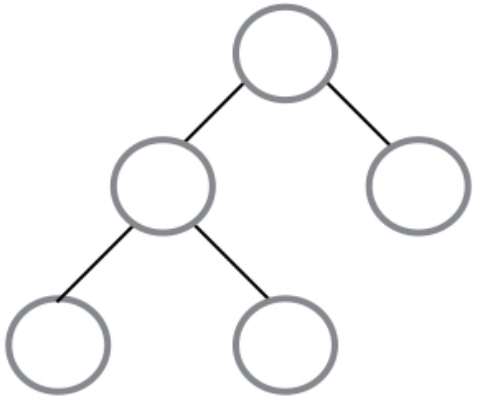
Abstract Data Types

- **Examples**:
  - Arrays
  - Linked Lists
  - Queues
  - Stacks
  - Trees
  - Graphs
  - Hash Tables
  - etc.
- **Properties studied**:
  - Efficiency
  - Tradeoffs
    - Space
    - Time

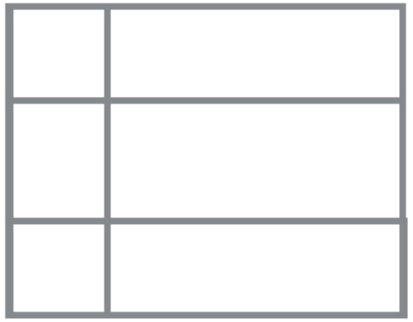
Abstract Data Types

Linear ADTs

Tree ADTs



Hash Tables



Graph ADTs

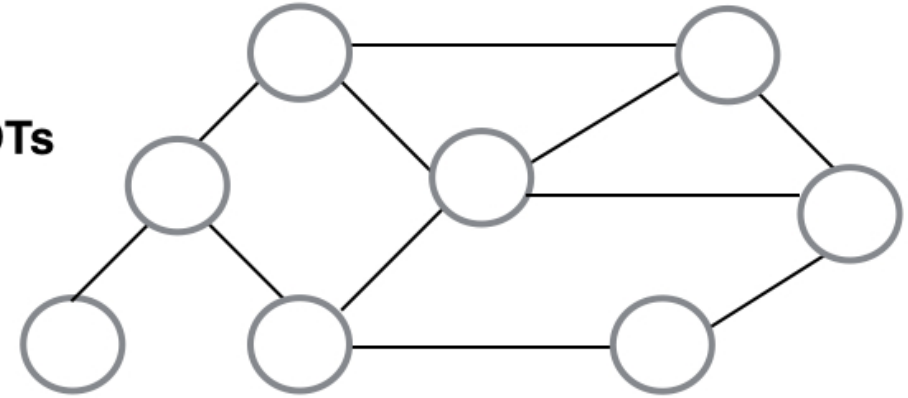
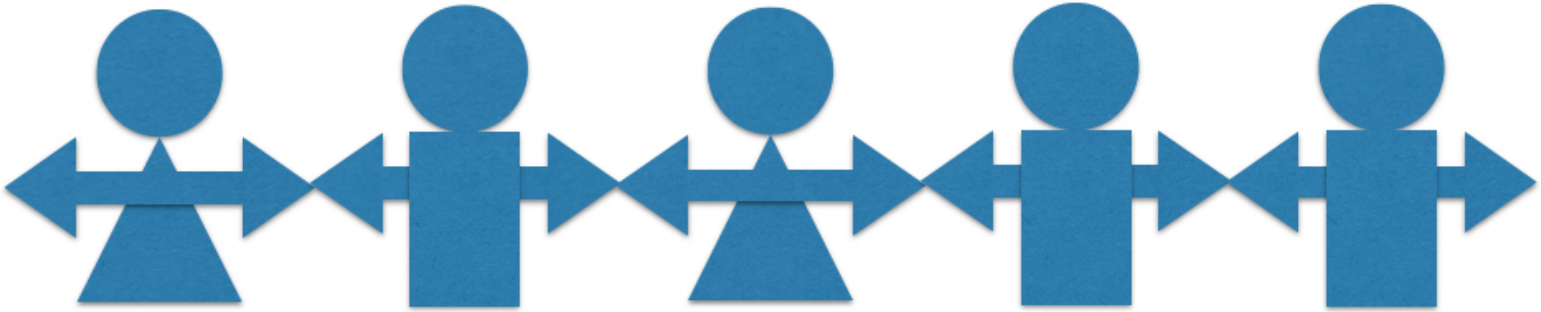


Table of Contents

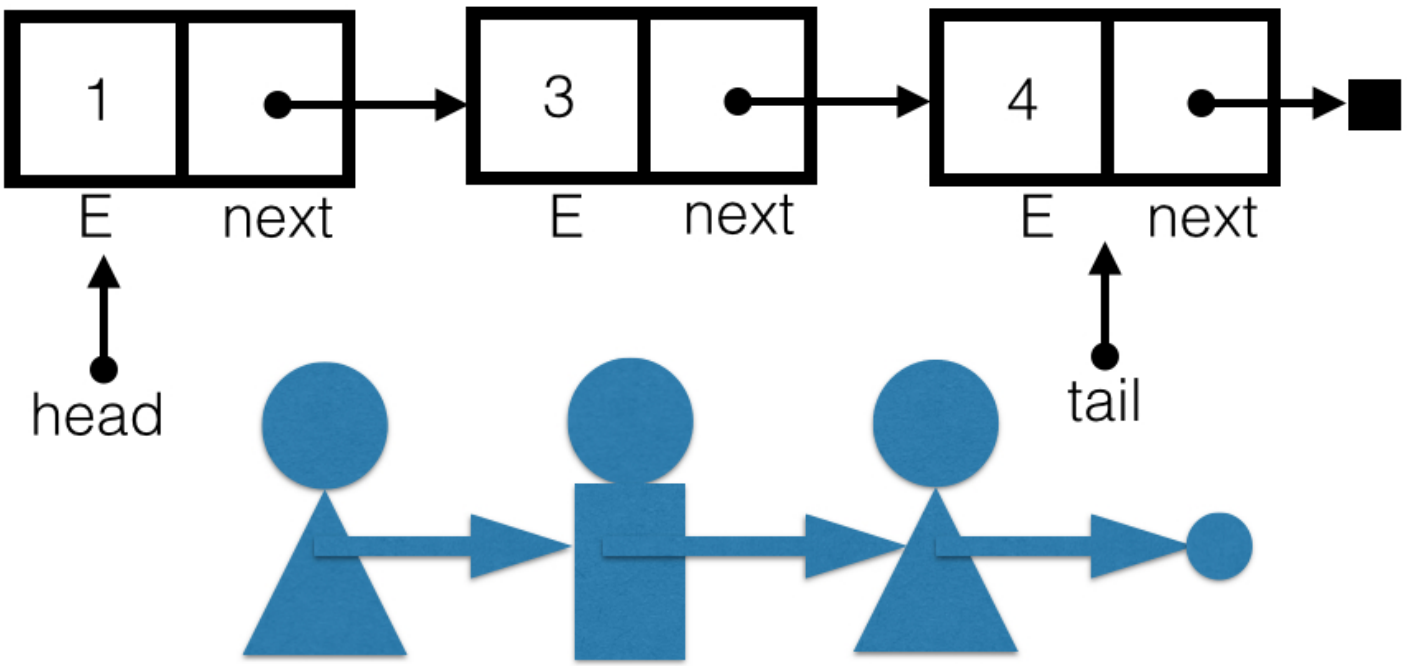
1. Introduction to Data Structures
2. Linked Lists
3. Queues
4. Stacks
5. List
6. Iterators
7. Java Collections Framework

What are Linked Lists?



- In linked data representations, a *reference* is used to connect separate blocks of objects/nodes together
- **Reminder**: Reference contains the **memory address** of an object that is dynamically created
- If a particular node has a reference to a second node, we can **follow** this reference to access the second node
- If the second node has a reference to another node, then the above can be repeated until no nodes are referred to

Singly Linked List



Singly Linked List

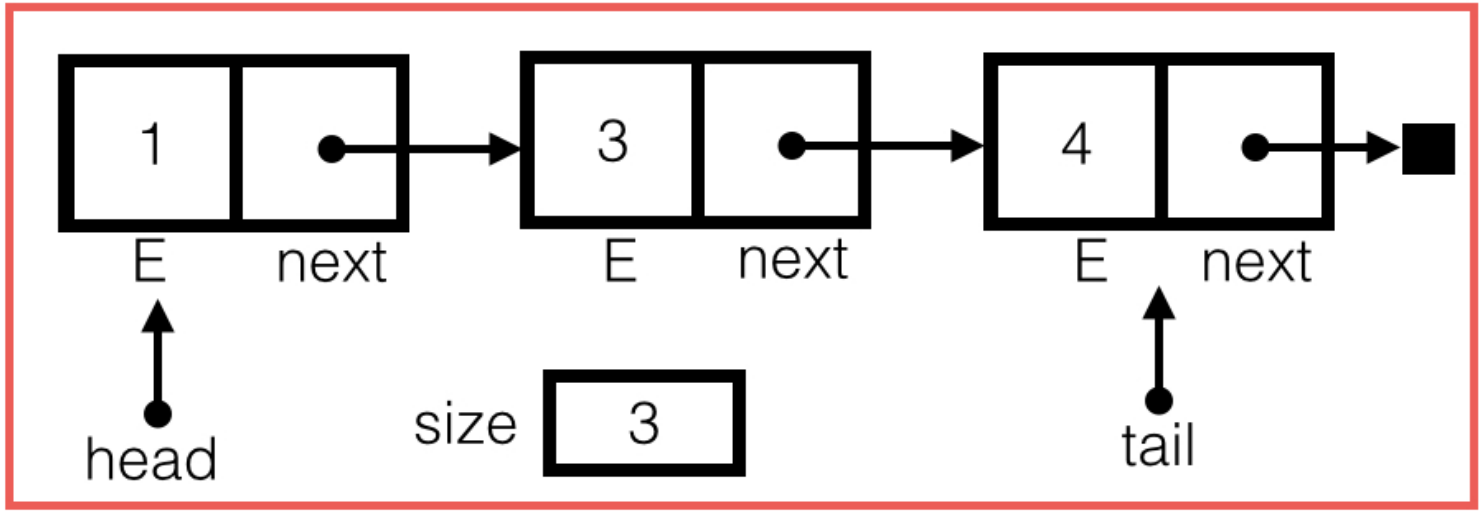
Supported operations:

- **Inserting** to either the front and back (why not the middle?)
- **Reading** content of the first or the last node (why not the middle?)
- **Deleting** from the front (why not the back?) and returning the data content of the deleted element

Why can Linked Lists be a Better Choice?

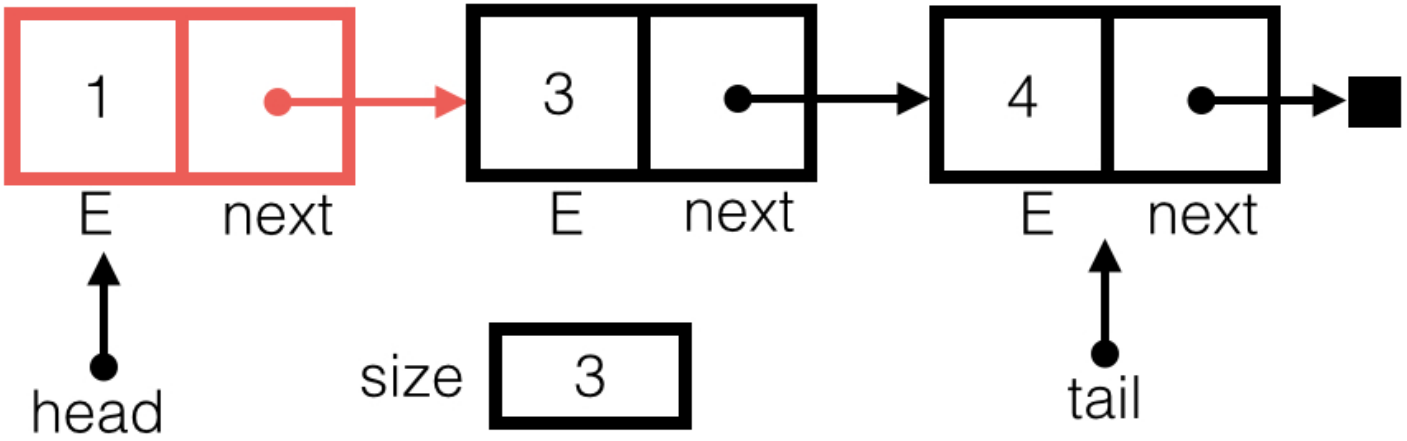
- In linked lists, we can keep on **adding** nodes until the computer runs out of memory
- Arrays can get filled up quickly and it is necessary to **resize** the array when this happens
- Resizing is not efficient as the array will occupy **unnecessary space** (especially when the array is sparse)

Structure of the Class and Components



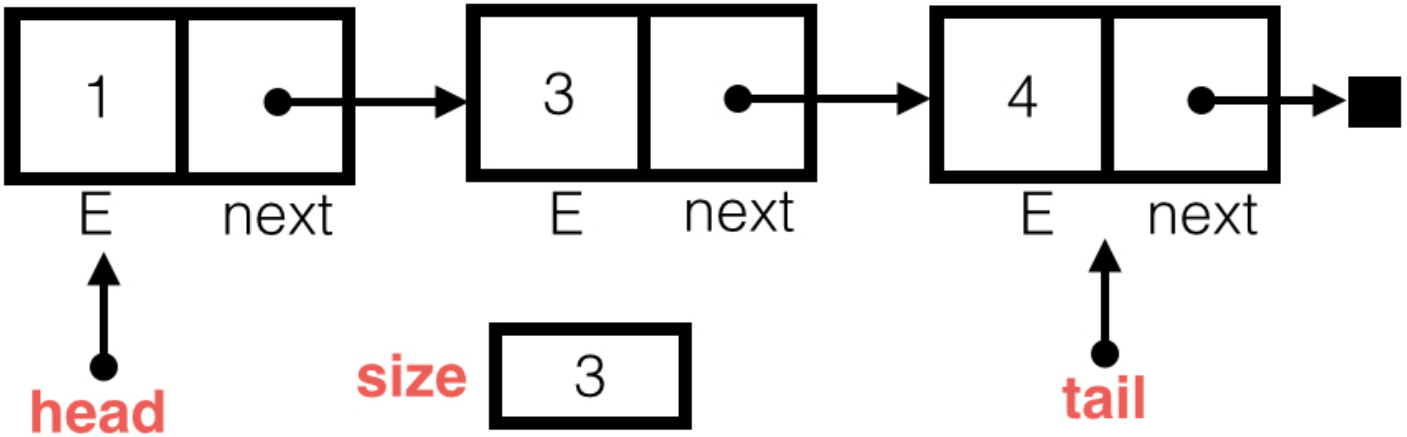

```
public class SinglyLinkedList<E>{
    private static class Node<E>{ ... }
    private Node<E> head;
    private Node<E> tail;
    private int size;
    public SinglyLinkedList(){ ... }
    public int size(){ ... }
    public boolean isEmpty(){ ... }
    public E first(){ ... }
    public E last(){ ... }
    public void addFirst(E element){ ... }
    public void addLast(E element){ ... }
    public E removeFirst(){ ... }
}
```

Structure of the Class and Components



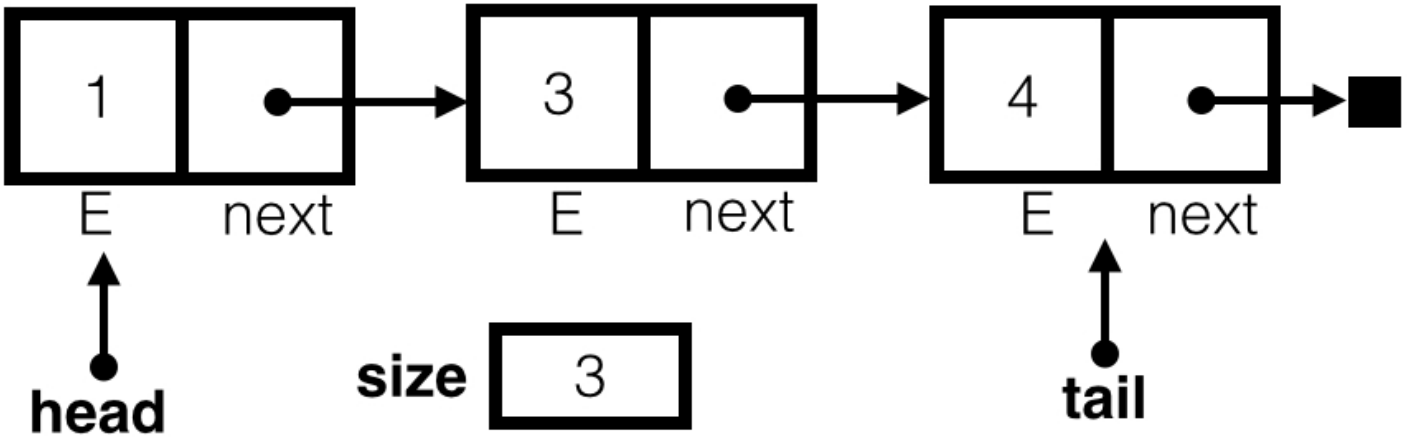

```
public class SinglyLinkedList<E>{
  private static class Node<E>{ ... }
  private Node<E> head;
  private Node<E> tail;
  private int size;
  public SinglyLinkedList(){ ... }
  public int size(){ ... }
  public boolean isEmpty(){ ... }
  public E first(){ ... }
  public E last(){ ... }
  public void addFirst(E element){ ... }
  public void addLast(E element){ ... }
  public E removeFirst(){ ... }
}
```

Structure of the Class and Components




```
public class SinglyLinkedList<E>{
  private static class Node<E>{ ... }
  private Node<E> head;
  private Node<E> tail;
  private int size;
  public SinglyLinkedList(){ ... }
  public int size(){ ... }
  public boolean isEmpty(){ ... }
  public E first(){ ... }
  public E last(){ ... }
  public void addFirst(E element){ ... }
  public void addLast(E element){ ... }
  public E removeFirst(){ ... }
}
```

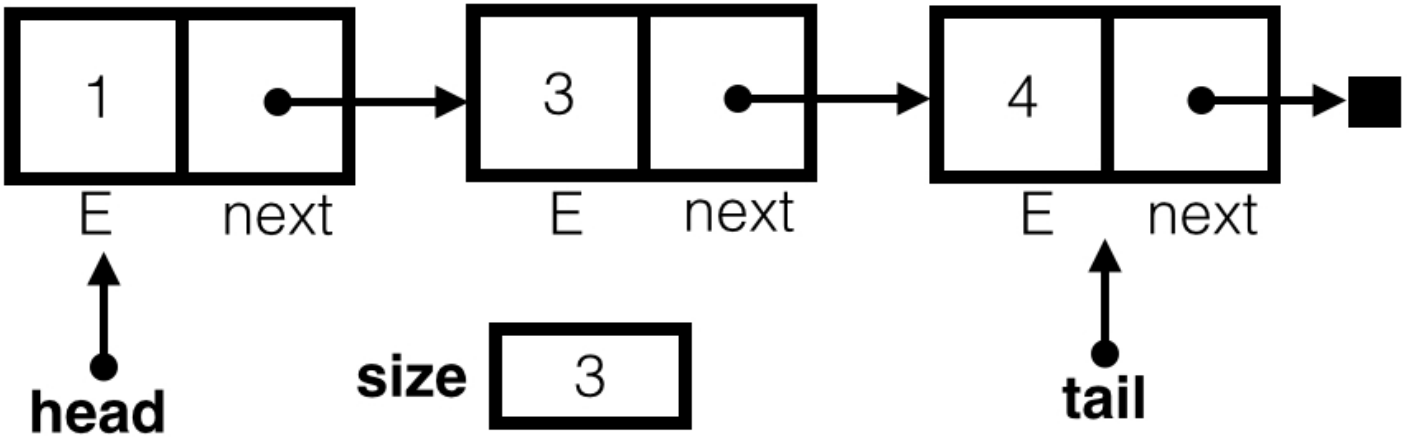
Structure of the Class and Components




```
public class SinglyLinkedList<E>{
  private static class Node<E>{ ... }
  private Node<E> head;
  private Node<E> tail;
  private int size;
  public SinglyLinkedList(){ ... }
  public int size(){ ... }
  public boolean isEmpty(){ ... }
  public E first(){ ... }
  public E last(){ ... }
  public void addFirst(E element){ ... }
  public void addLast(E element){ ... }
  public E removeFirst(){ ... }
}
```

Auxiliary helper functions

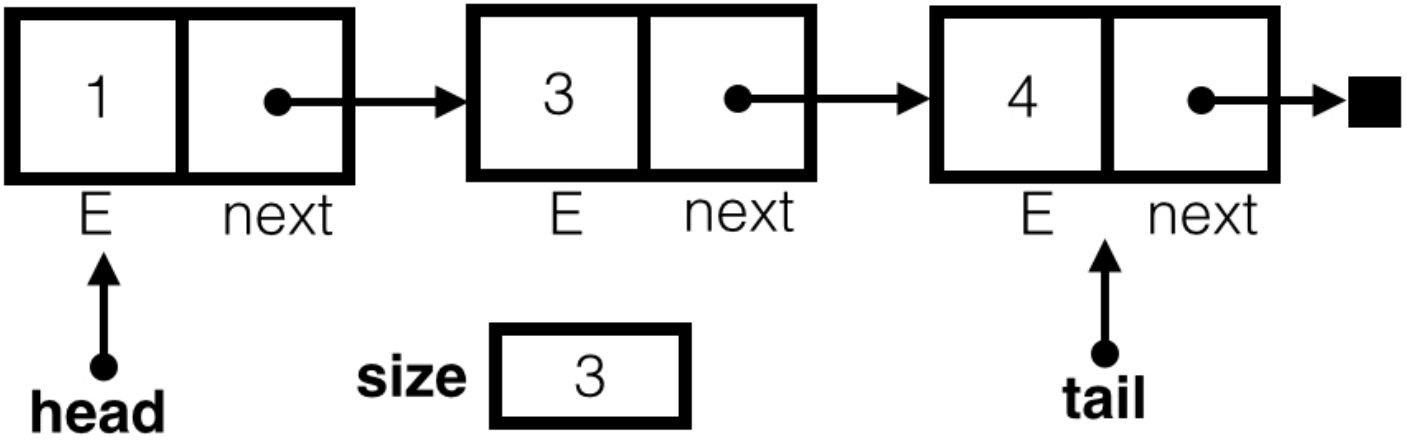
Structure of the Class and Components




```
public class SinglyLinkedList<E>{
  private static class Node<E>{ ... }
  private Node<E> head;
  private Node<E> tail;
  private int size;
  public SinglyLinkedList(){ ... }
  public int size(){ ... }
  public boolean isEmpty(){ ... }
  public E first(){ ... }
  public E last(){ ... }
  public void addFirst(E element){ ... }
  public void addLast(E element){ ... }
  public E removeFirst(){ ... }
}
```

Read content

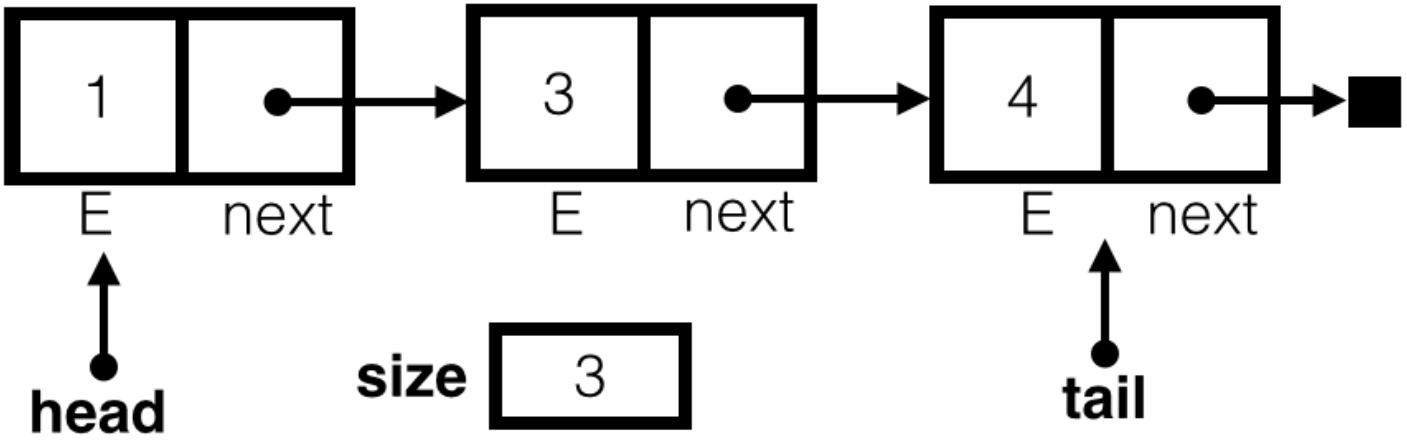
Structure of the Class and Components




```
public class SinglyLinkedList<E>{
  private static class Node<E>{ ... }
  private Node<E> head;
  private Node<E> tail;
  private int size;
  public SinglyLinkedList(){ ... }
  public int size(){ ... }
  public boolean isEmpty(){ ... }
  public E first(){ ... }
  public E last(){ ... }
  public void addFirst(E element){ ... }
  public void addLast(E element){ ... }
  public E removeFirst(){ ... }
}
```

Add content

Structure of the Class and Components

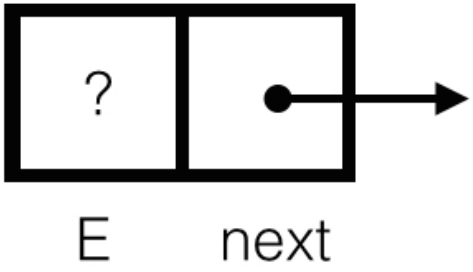



```
public class SinglyLinkedList<E>{
  private static class Node<E>{ ... }
  private Node<E> head;
  private Node<E> tail;
  private int size;
  public SinglyLinkedList(){ ... }
  public int size(){ ... }
  public boolean isEmpty(){ ... }
  public E first(){ ... }
  public E last(){ ... }
  public void addFirst(E element){ ... }
  public void addLast(E element){ ... }
  public E removeFirst(){ ... }
}
```

Read and remove content

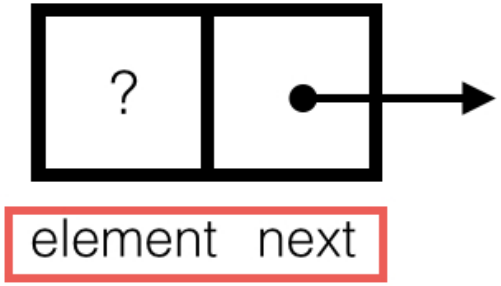
Defining the Node Class

- A node is composed of **two fields**:
  - 1) Element field and 2) Reference to the next node
- Need to be **generic** as not aware of what element will be stored apriori
- Read 1) Element stored in a node and 2) Reference to the next node
- Reset the reference to the next node (in case of a deletion or an insertion)
- **Methods**: constructor, `getElement`, `getNext`, `setNext`
- **Instance variables**: `element`, `next`



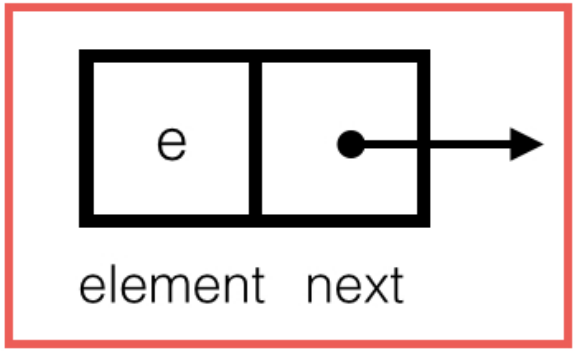
Defining the Node Class: Fields

```
private static class Node<E>{
  E element;
  private Node<E> next=null;
  public Node(E e, Node<E> n){
    element=e;
    next=n;
  }
  public E getElement(){
    return element;
  }
  public Node<E> getNext(){
    return next;
  }
  public void setNext(Node<E> n){
    next=n;
  }
}
```



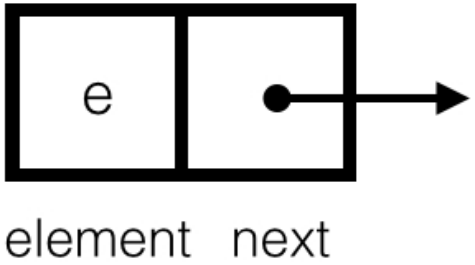
Defining the Node Class: Constructor

```
private static class Node<E>{
  E element;
  private Node<E> next=null;
  public Node(E e, Node<E> n){
    element=e;
    next=n;
  }
  public E getElement(){
    return element;
  }
  public Node<E> getNext(){
    return next;
  }
  public void setNext(Node<E> n){
    next=n;
  }
}
```

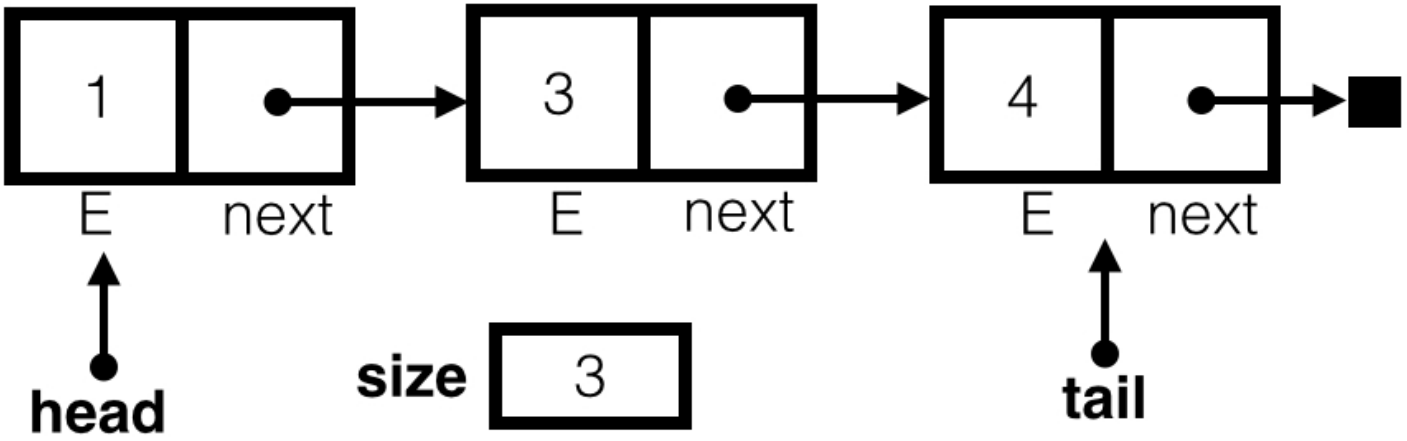


Defining the Node Class: Accessing and Modifying a Node

```
private static class Node<E>{
  E element;
  private Node<E> next=null;
  public Node(E e, Node<E> n){
    element=e;
    next=n;
  }
  public E getElement(){
    return element;
  }
  public Node<E> getNext(){
    return next;
  }
  public void setNext(Node<E> n){
    next=n;
  }
}
```



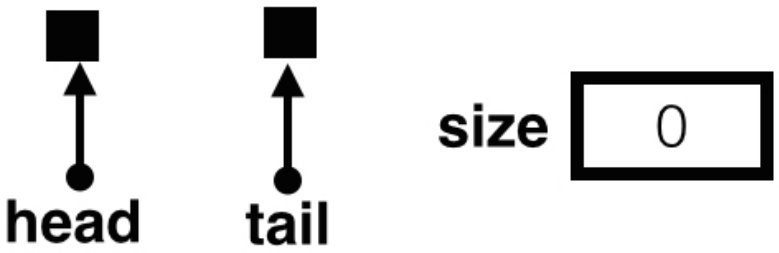
Constructor for Singly Linked List




```
public class SinglyLinkedList<E>{
  private static class Node<E>{ ... }
  private Node<E> head;
  private Node<E> tail;
  private int size;
  public SinglyLinkedList(){ ... }
  public int size(){ ... }
  public boolean isEmpty(){ ... }
  public E first(){ ... }
  public E last(){ ... }
  public void addFirst(E element){ ... }
  public void addLast(E element){ ... }
  public E removeFirst(){ ... }
}
```

Constructor for Singly Linked List

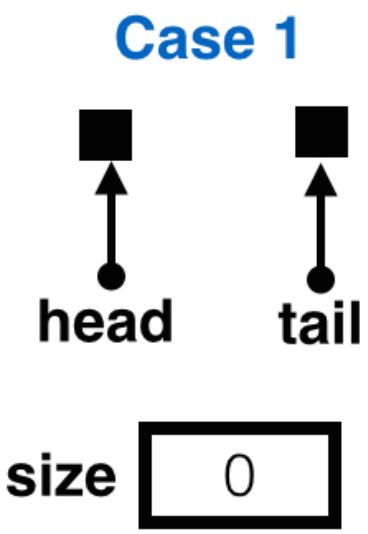
```
public SinglyLinkedList(){
  head=null;
  tail=null;
  size=0;
}
```

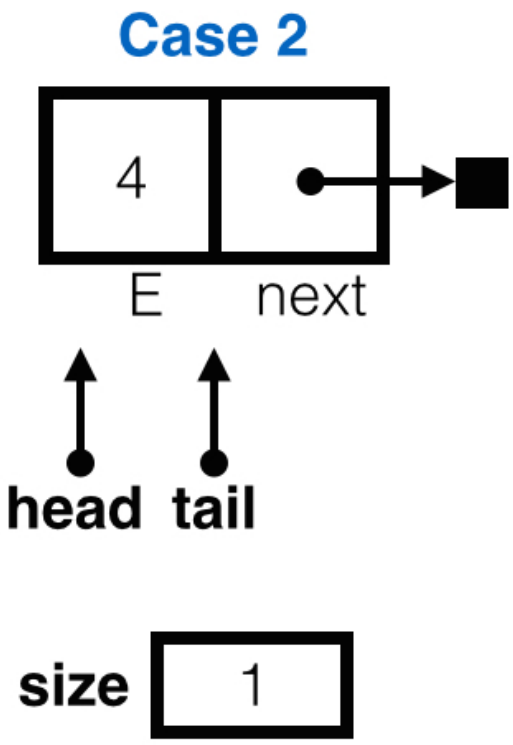


Inserting A Node to the Front of a Linked List

What are the various cases?

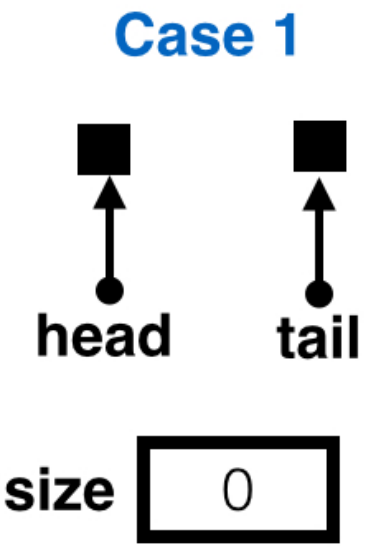
1. The list is currently empty (i.e. no nodes are in the list)
2. The list is not empty (i.e. at least one node is in the list)





Inserting A Node to the Front of a Linked List: Case 1

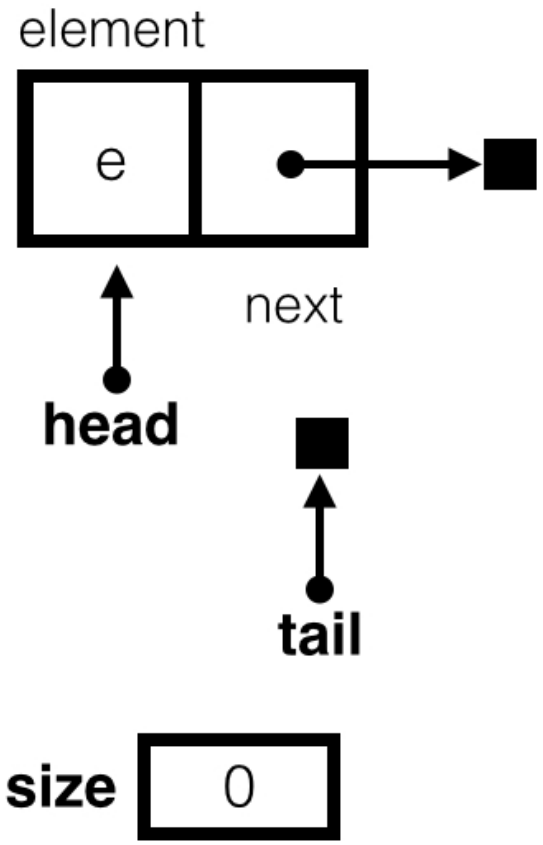
```
public void addFirst(E element){
  head = new Node<E>(element, head);
  if(isEmpty())
    tail=head;
  size++;
}
```

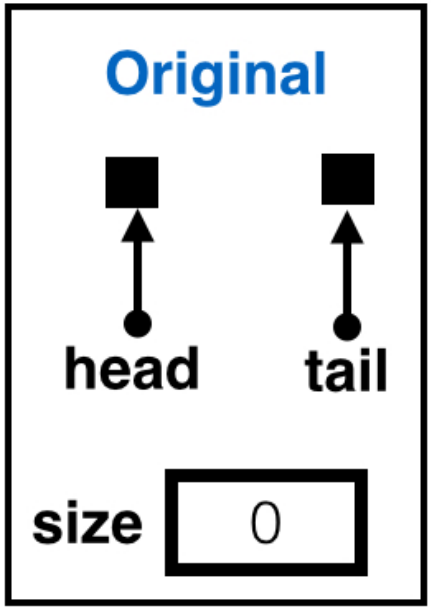


Inserting A Node to the Front of a Linked List: Case 1

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

Case 1

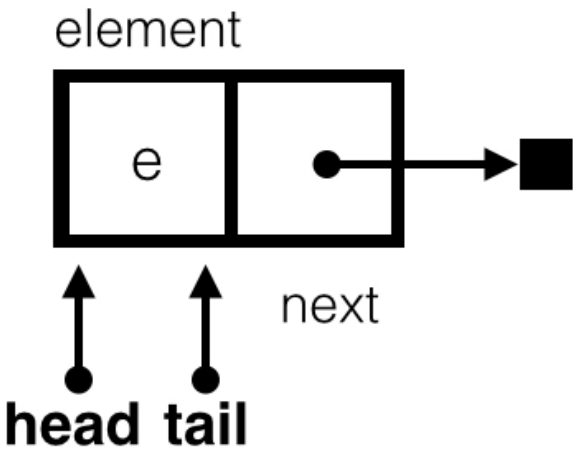


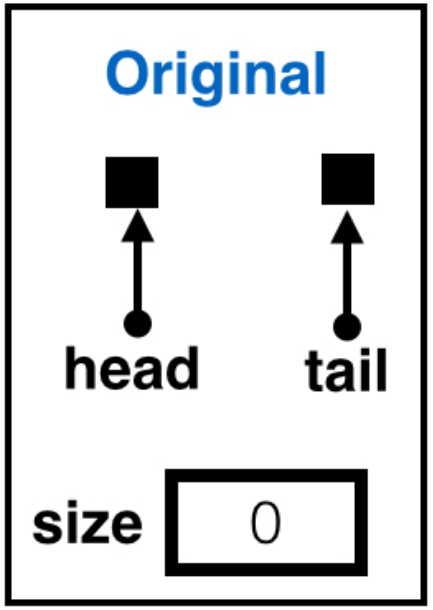


Inserting A Node to the Front of a Linked List: Case 1

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

Case 1

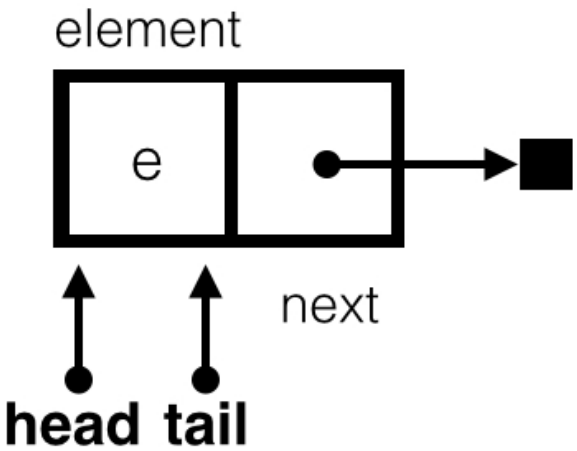


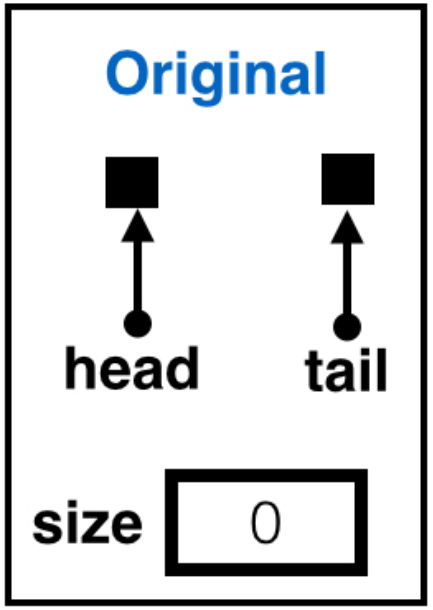


Inserting A Node to the Front of a Linked List: Case 1

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

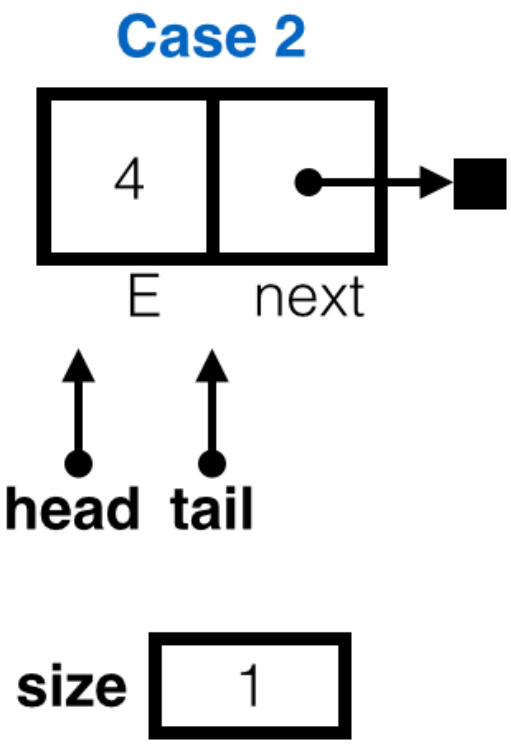
Case 1





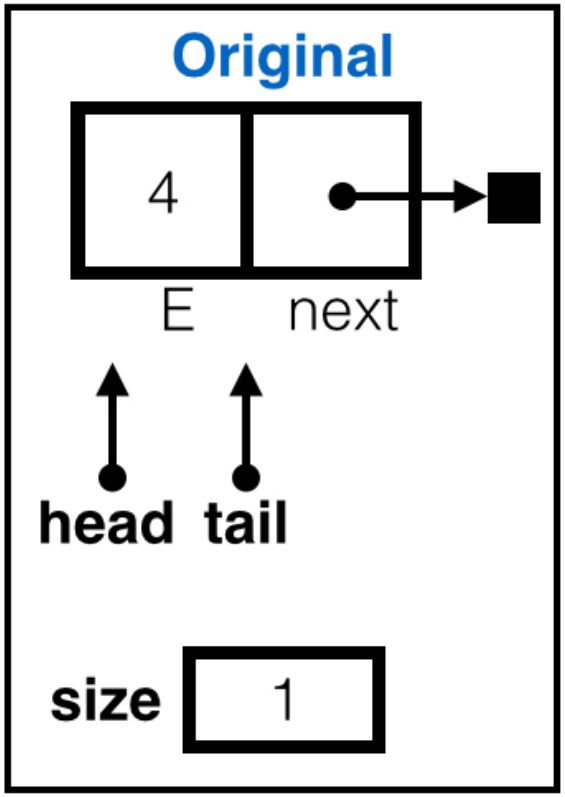
Inserting A Node to the Front of a Linked List: Case 2

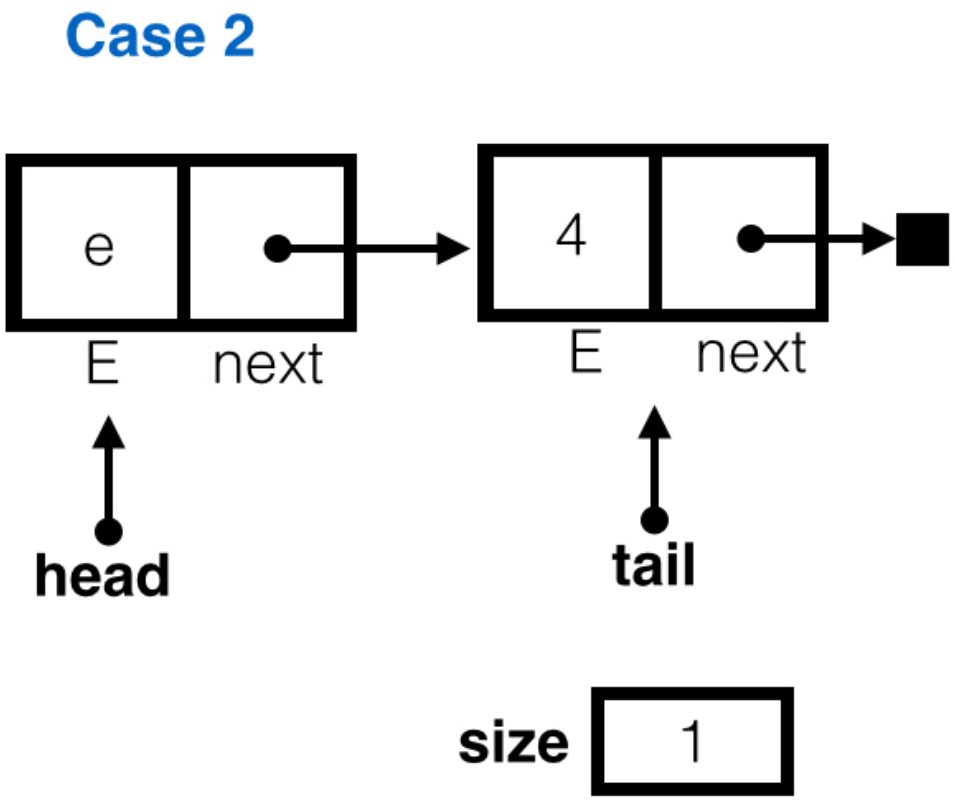
```
public void addFirst(E element){
  head = new Node<E>(element, head);
  if(isEmpty())
    tail=head;
  size++;
}
```



Inserting A Node to the Front of a Linked List: Case 2

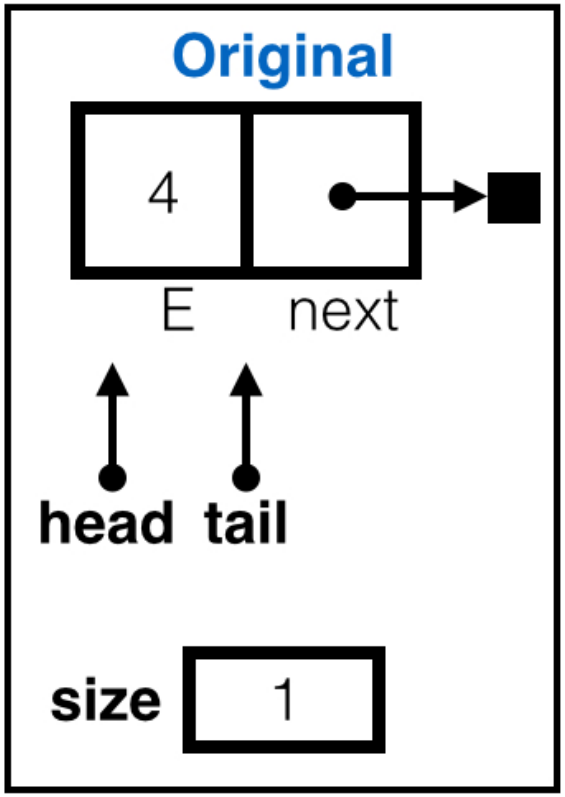
```
public void addFirst(E element){
  head = new Node<E>(element, head);
  if(isEmpty())
    tail=head;
  size++;
}
```

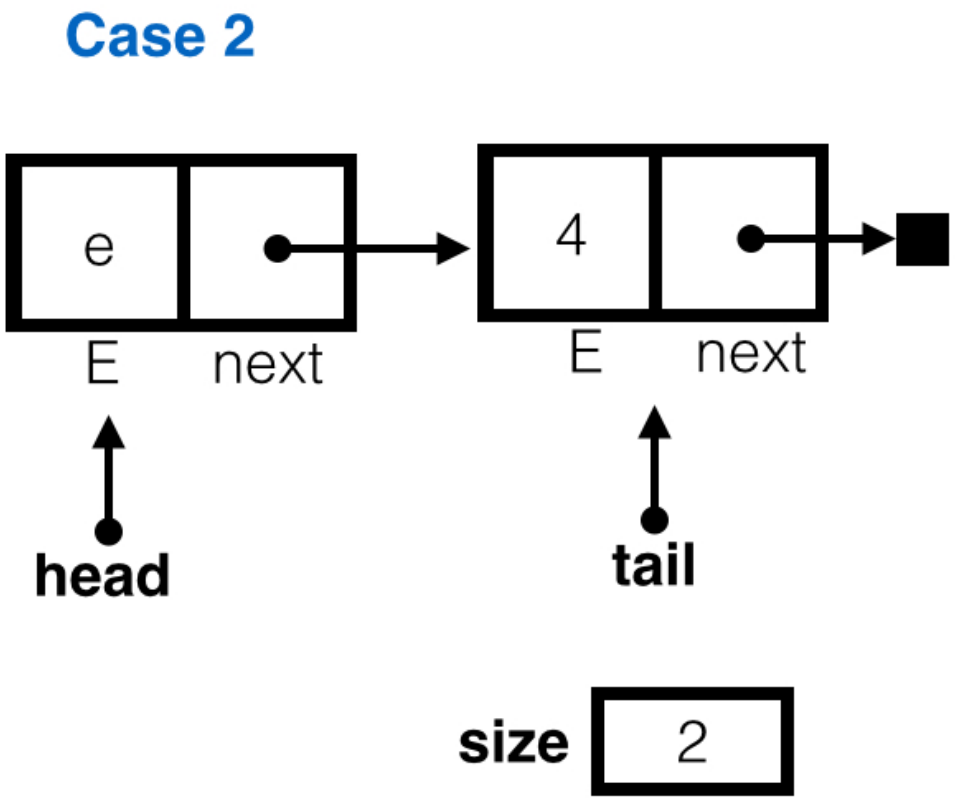




Inserting A Node to the Front of a Linked List: Case 2

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

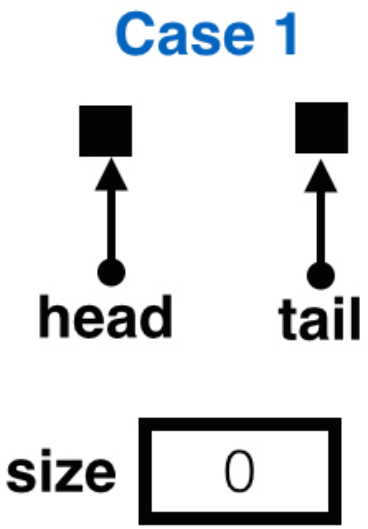


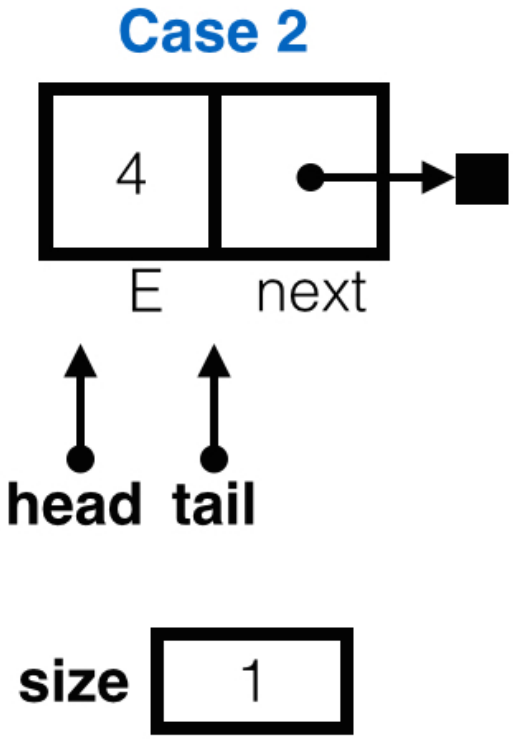


Inserting A Node to the End of a Linked List

- What are the various cases?
  1. The list is currently empty (i.e. no nodes are in the list)
  2. The list is not empty (i.e. at least one node is in the list)

Brainstorm the steps needed to implement both of these cases





Complexity of Inserting Nodes into a Linked List

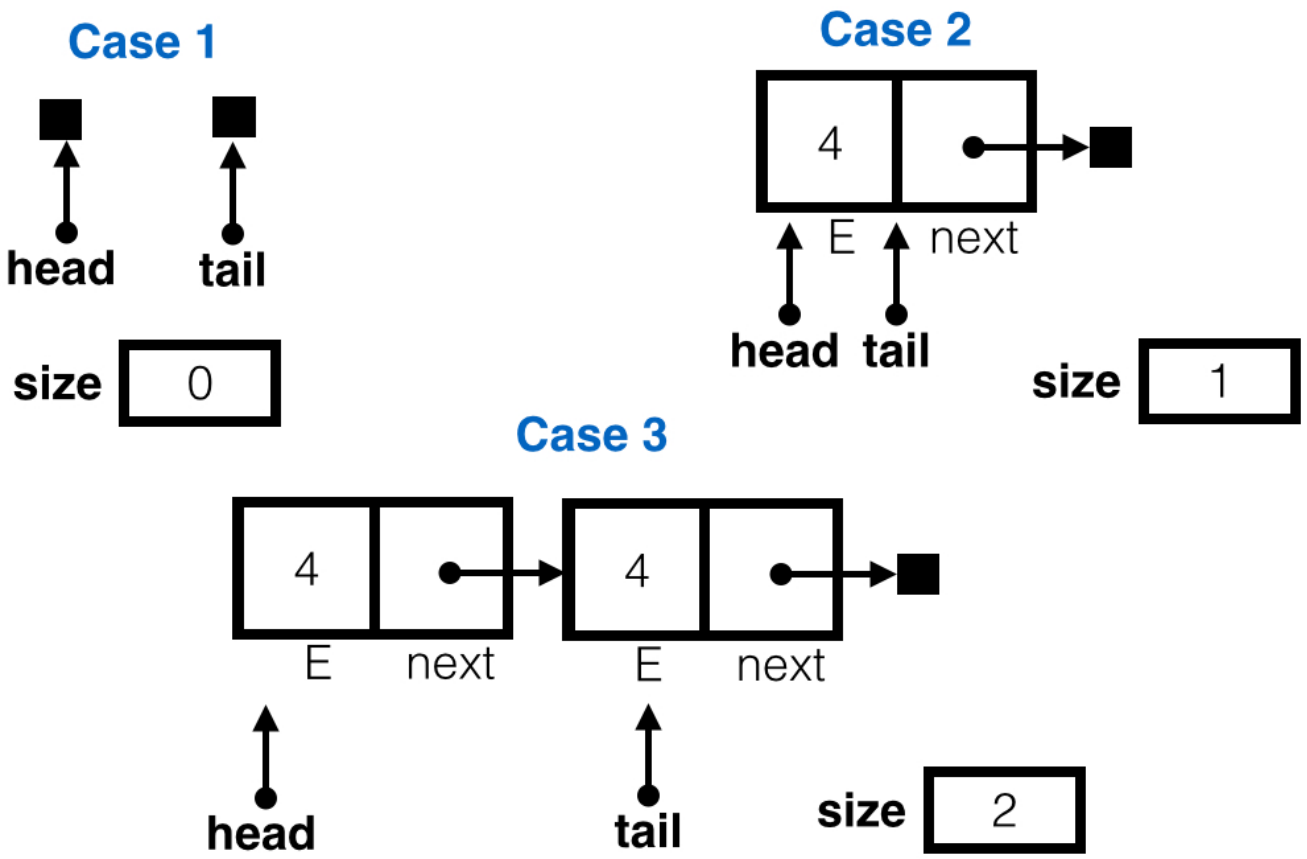
Assuming that there are n nodes, what is the complexity of inserting a node:

- At the front of an array
- At the front of a linked list
- At the end of an array
- At the end of a linked list
- In between nodes in an array at position p
- In between nodes of a linked list at position p

Deleting Nodes from the Front of a Linked List

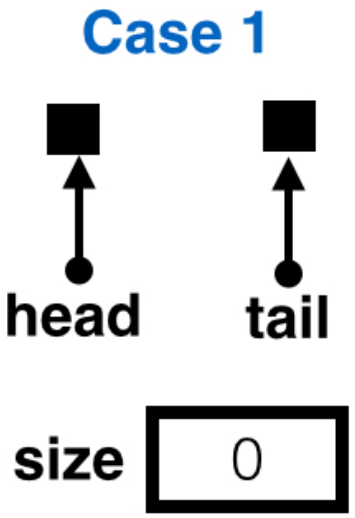
What are the possible cases?

- List is empty (i.e. no nodes)
- List has only one node
- List has more than one node



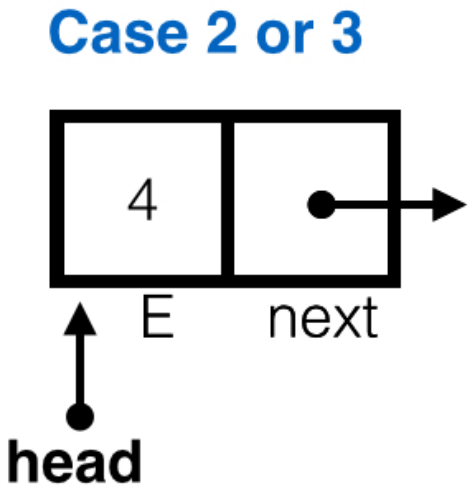
Deleting Nodes from the Front of a Linked List

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



Deleting Nodes from the Front of a Linked List

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

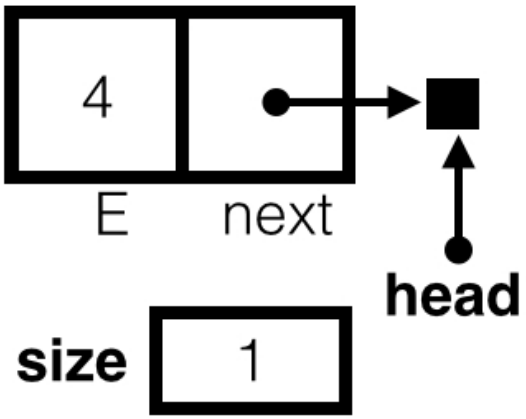


element: 4

Deleting Nodes from the Front of a Linked List

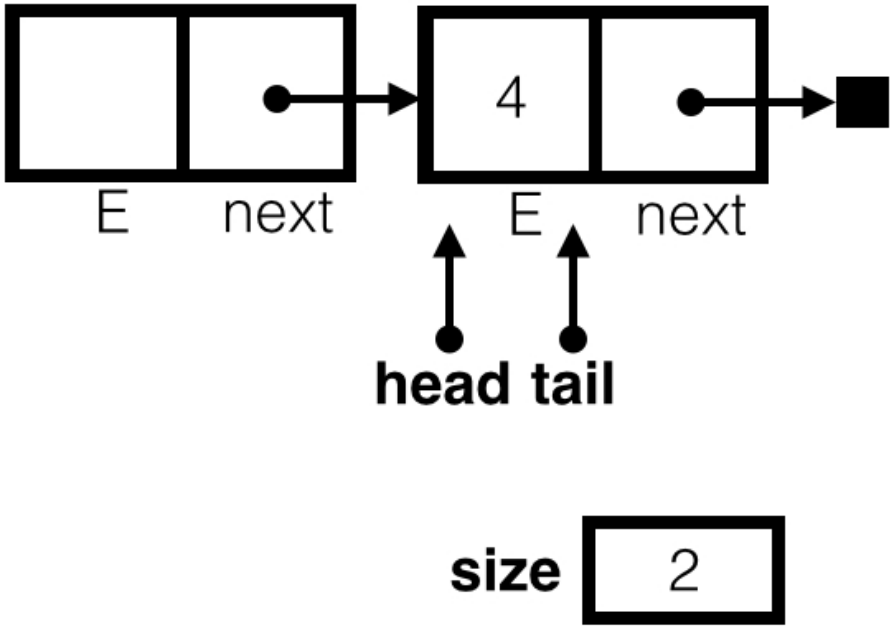
```
public E removeFirst(){
  if(isEmpty())
    return null;
  E element=head.getElement();
  head=head.getNext();
  size--;
  if(size==0)
    tail=null;
  return element;
}
```

Case 2



element: 4

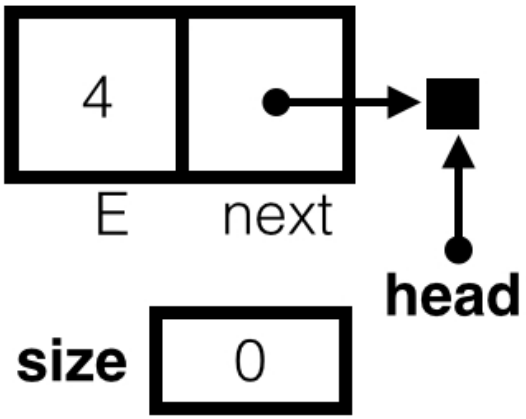
Case 3



Deleting Nodes from the Front of a Linked List

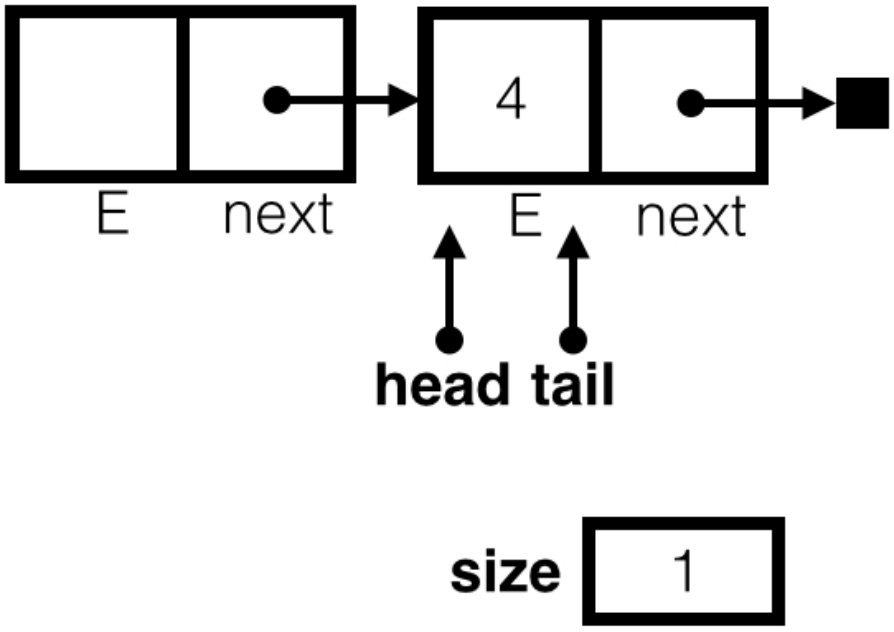
```
public E removeFirst(){
  if(isEmpty())
    return null;
  E element=head.getElement();
  head=head.getNext();
  size--;
  if(size==0)
    tail=null;
  return element;
}
```

Case 2



element: 4

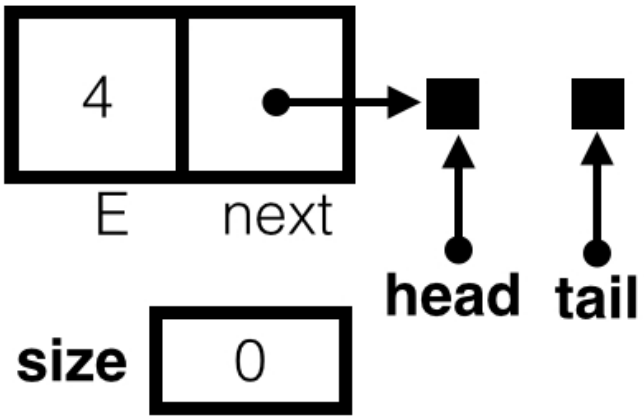
Case 3



Deleting Nodes from the Front of a Linked List

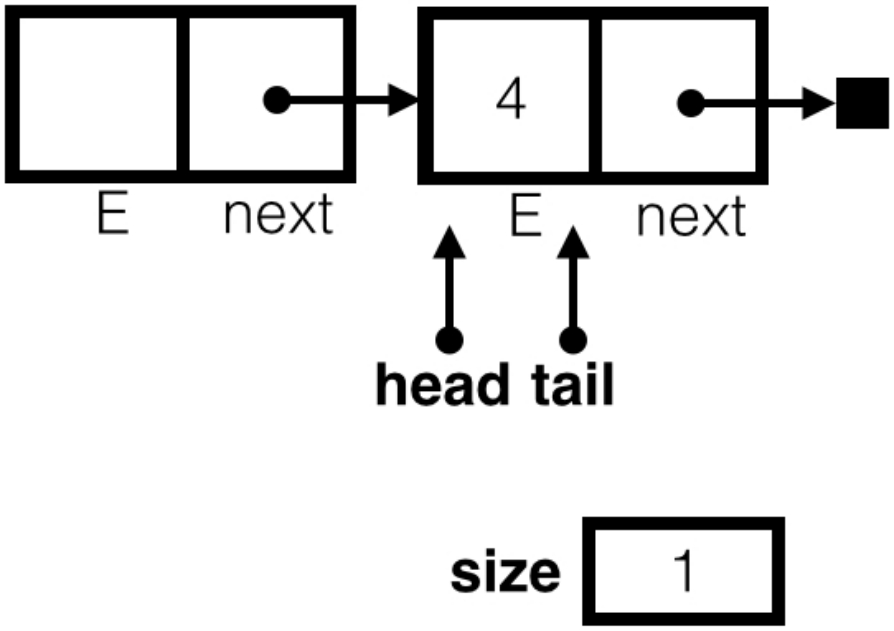
```
public E removeFirst(){
  if(isEmpty())
    return null;
  E element=head.getElement();
  head=head.getNext();
  size--;
  if(size==0)
    tail=null;
  return element;
}
```

Case 2



element: 4

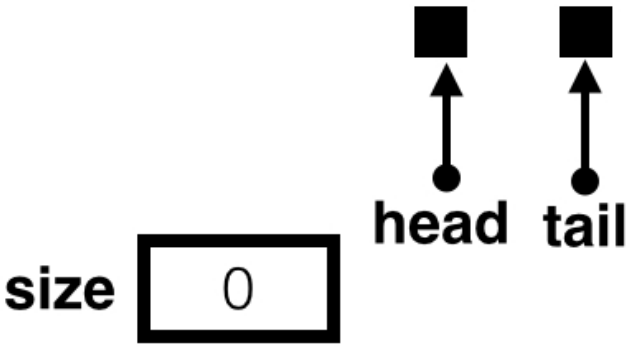
Case 3



Deleting Nodes from the Front of a Linked List

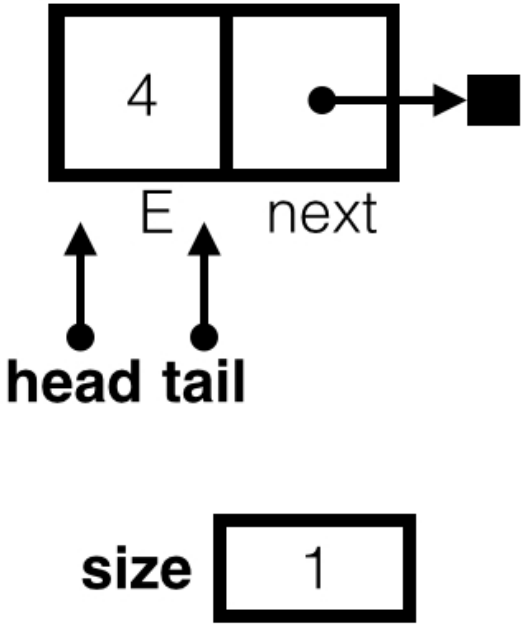
```
public E removeFirst(){
  if(isEmpty())
    return null;
  E element=head.getElement();
  head=head.getNext();
  size--;
  if(size==0)
    tail=null;
  return element;
}
```

Case 2



element: 4

Case 3

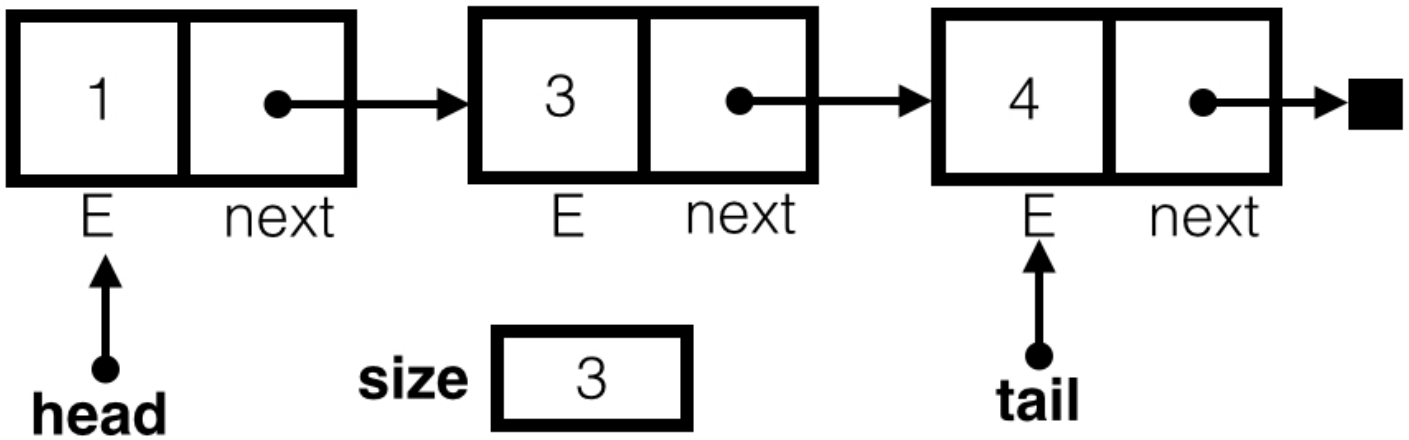


Reading from First/Last Nodes of a Linked List

- Need to check if the linked list is empty
- If reading from the front, then read the element stored in the head
- If reading from the end, then read the element stored in the tail

```
public E first(){
  if(head!=null)
    return head.getElement();
  else
    return null;
}
```

```
public E last(){
  if(tail!=null)
    return tail.getElement();
  else
    return null;
}
```



Helper Functions

- Useful for simplifying the implementation of the main methods (adding/deleting/reading)

```
public int size(){
  return size;
}

public boolean isEmpty(){
  if(size==0)
    return true;
  else
    return false;
}
```

Linked Lists vs Arrays

- In linked lists, we can keep on **adding** nodes until the computer runs out of memory
- Arrays can get filled up quickly and it is necessary to **resize** the array when this happens
- Resizing is not efficient as the array will occupy **unnecessary space** (especially when the array is sparse)
- One problem with linked list is that it is not possible to directly access an element even if the index is known
- Need to traverse through a set of nodes O(n)
- With an array, accessing elements in known indices is O(1)

Doubly Linked List

- Each node has references to both previous and next nodes
- What are the various ways of implementing a doubly linked list?
- What are the challenges with inserting/deleting nodes from a doubly linked list?
- Goodrich introduces the notion of 'sentinels'
  - Dummy nodes are inserted at the beginning and end of the doubly linked list
- **Challenge: Implement without using sentinels**

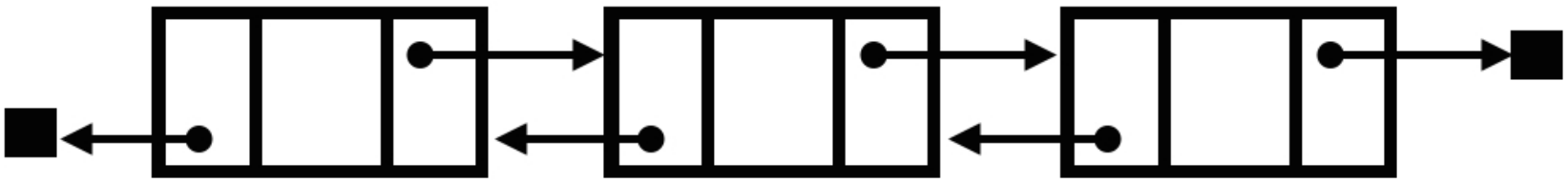


Table of Contents

1. Introduction to Data Structures
2. Linked Lists
3. Queues
4. Stacks
5. List
6. Iterators
7. Java Collections Framework

Introduction to Queues

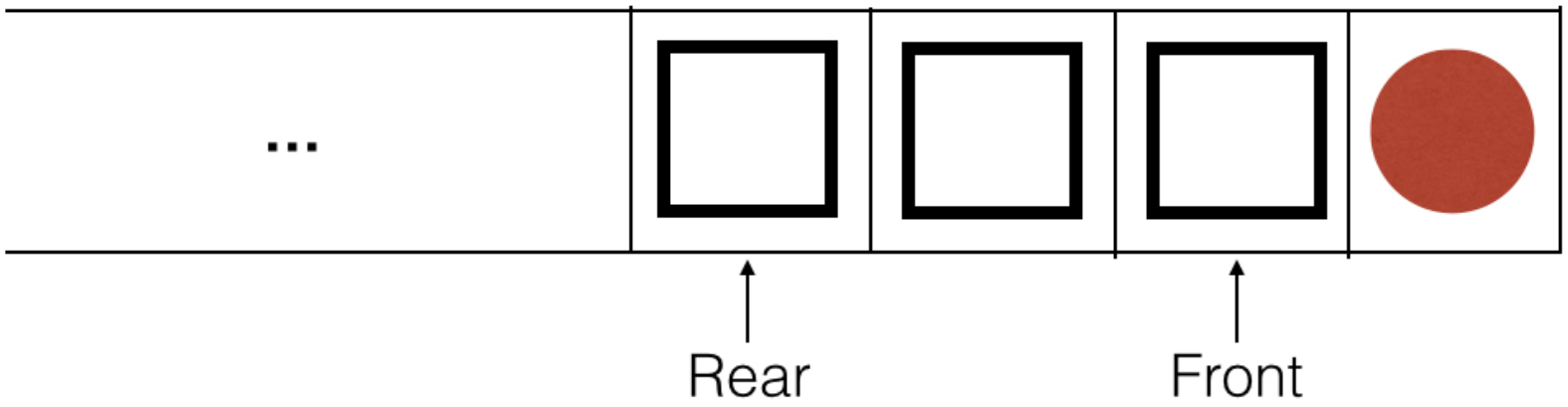
- **Queue** is an Abstract Data Type (ADT)
- Queue is a **linear data structure**
  - Collection of nodes in a queue are arranged in a **sequence** or a straight line
- Components are added and removed from a queue in a **particular** manner
  - These operations distinguish a queue from other ADTs
- There may or may not be a restriction on the number of nodes in a queue
- Queues can be implemented via **linked** (linked lists) or **sequential** (arrays) representation

Introduction to Queues

- Queues are analogous to systems in which it is necessary to **wait in lines**
- Those who arrive at the queue first are served first
- This queuing policy is known **first in first out** (FIFO)
- This is a **"fair"** policy
- Examples include:
  - Client-server (need to wait in a line to obtain service)
  - Operating systems (task records, printing documents, CPU services, etc.)
  - Simulation and modelling (flight routes, congestion in networks, etc.)
- Queues may be infinite or finite in length

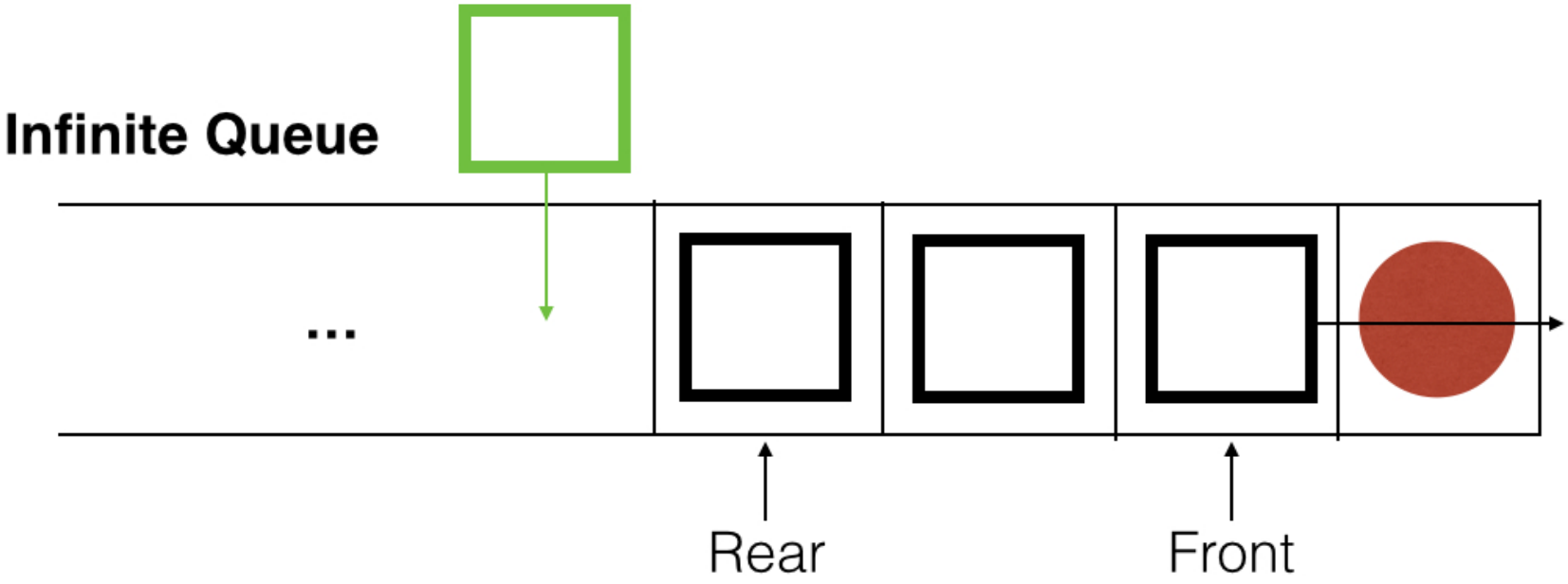
Queues: Analogous to Waiting in a Line

Infinite Queue



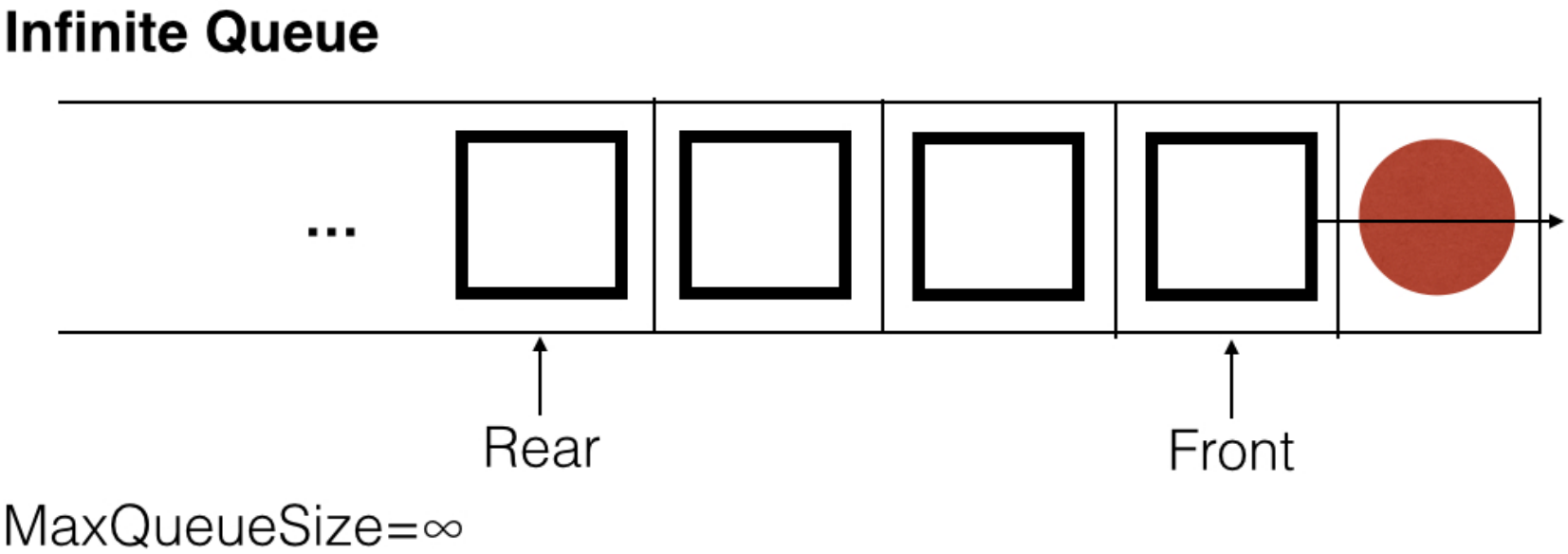
MaxQueueSize=∞

Queues: Analogous to Waiting in a Line



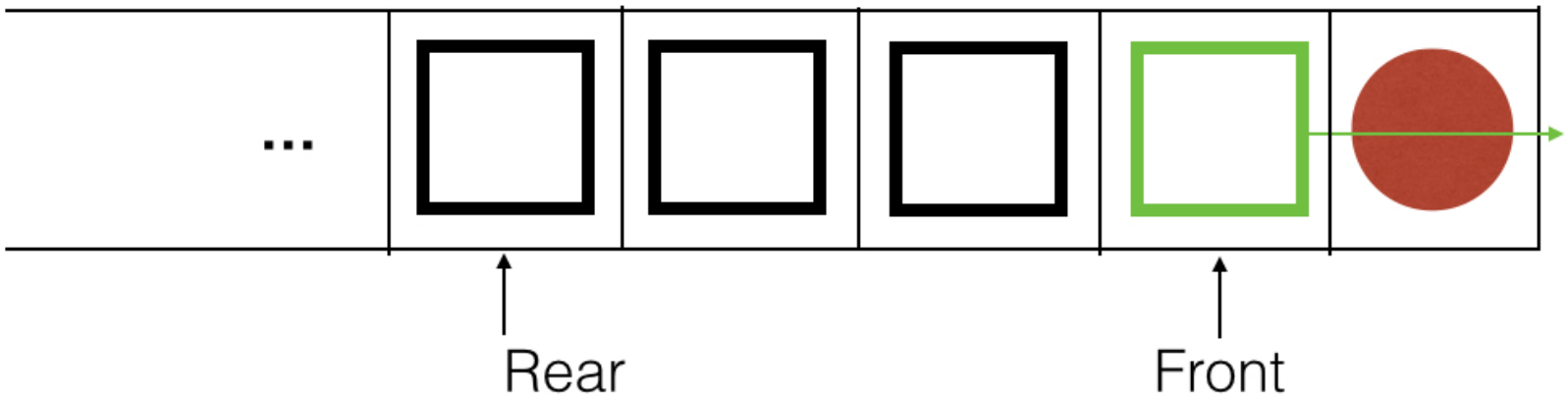
MaxQueueSize=∞

Queues: Analogous to Waiting in a Line



Queues: Analogous to Waiting in a Line

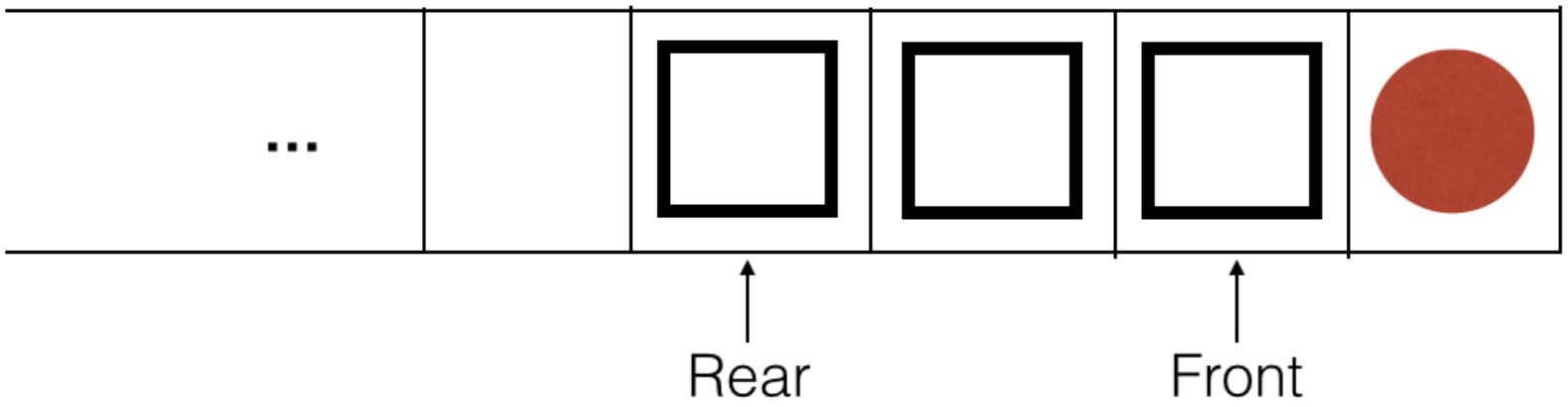
Infinite Queue



MaxQueueSize=∞

Queues: Analogous to Waiting in a Line

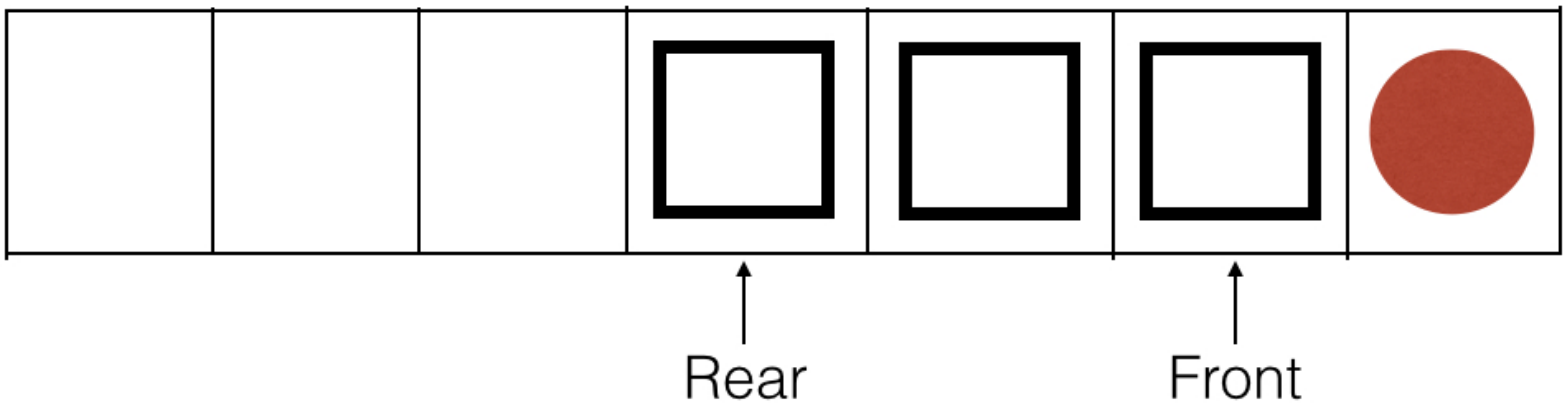
Infinite Queue



MaxQueueSize=∞

Queues: Analogous to Waiting in a Line

Finite Queue



MaxQueueSize<∞

Queues Interfaces

- Interfaces to a queue: `enqueue` and `dequeue` operations
- `enqueue` will insert into the **rear** of the queue a new node
- `dequeue` will remove the node at the **front** and indirectly/directly returns the data stored in the removed node
- `first` will read the element at the front of the queue
- Logistics about the queue: `size` and `isEmpty`
- All these operations can be implemented either via **arrays** or as **linked lists**
- These implementations are **hidden** to the user as he or she will only be interacting with the ADT via the **well-defined interfaces**
- Hence, how everything operates "under the hood" will not be an issue

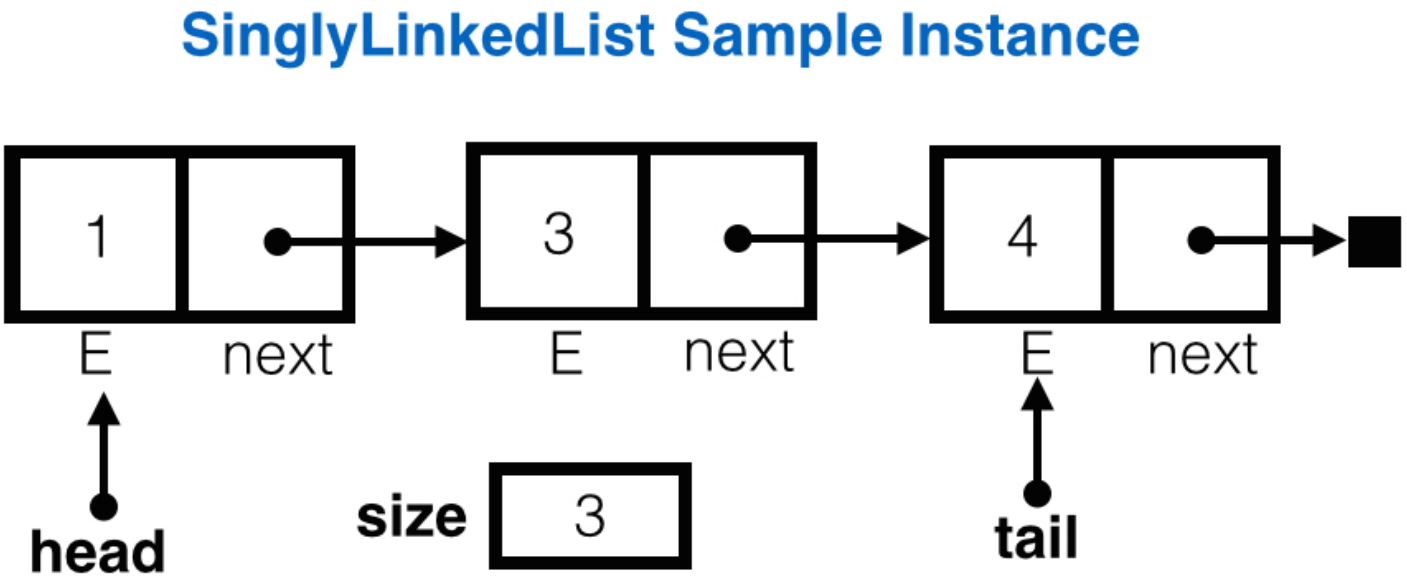
Impact of Underlying Implementation

- Regardless of what the underlying implementation is, a queue is accessed and operated upon via the **same interfaces or functions**
- Hence, there is no need to **distinguish** implementations
- For an array implementation, the only concern or limitation is that the array is **fixed in length** and can support only a **finite length queue**

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

Linked List Implementation of Queue

- Recall the `SinglyLinkedList` ADT?
- Can use this ADT to implement the linked list version of the `Queue`

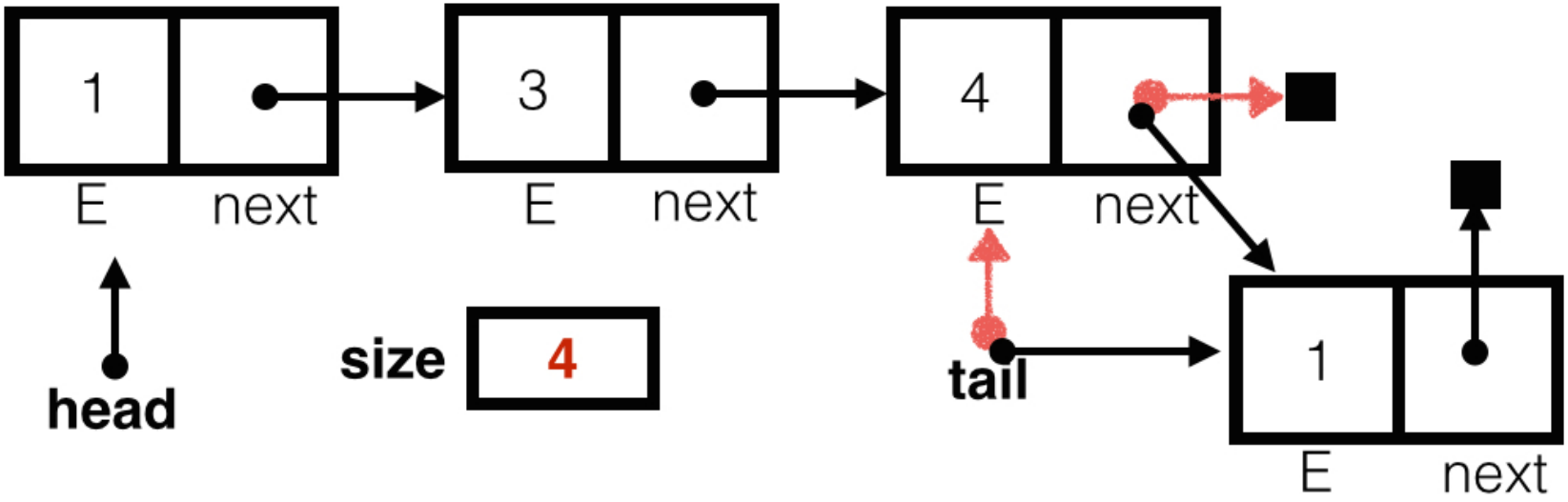



```
public class LinkedListQueue<E> implements Queue<E>{
  private SinglyLinkedList<E> list = new SinglyLinkedList<E>();
  public LinkedListQueue(){}
  public int size(){...}
  public boolean isEmpty(){...}
  public E first(){...}
  public void enqueue(E node){...}
  public E dequeue(){...}
}
```

Linked List Implementation of Queue: Enqueue Operation

- Enqueue entails the insertion of new node to the back of the list (FIFO)
- Can call the addLast function defined in SinglyLinkedList class

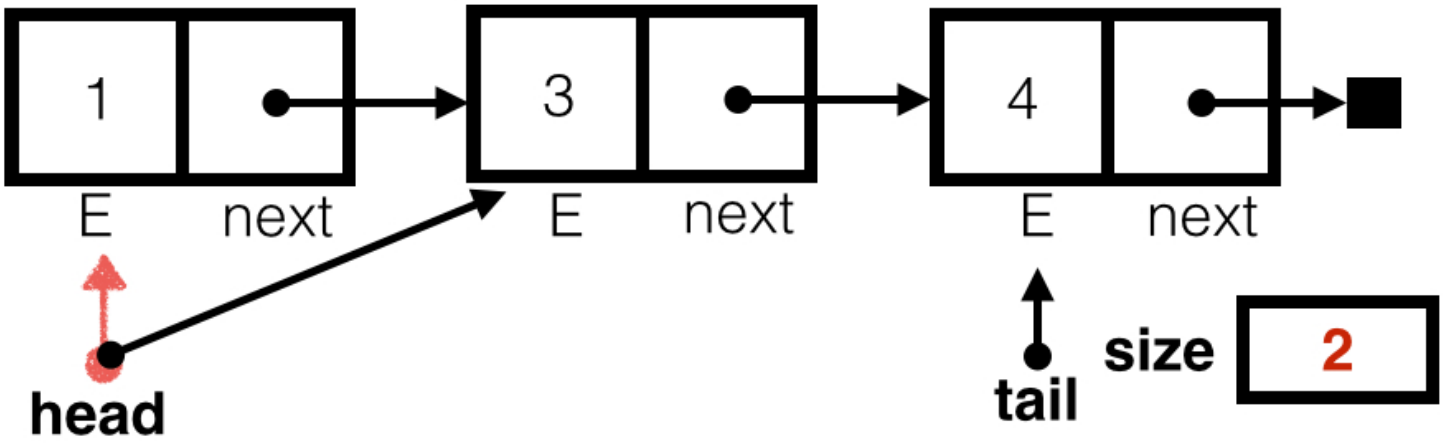
```
public class LinkedListQueue<E> implements Queue<E>{
  private SinglyLinkedList<E> list = new SinglyLinkedList<E>();
  public LinkedListQueue(){}
  public int size(){...}
  public boolean isEmpty(){...}
  public E first(){...}
  public void enqueue(E node){ list.addLast(node);}
  public E dequeue(){...}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



Linked List Implementation of Queue: Dequeue Operation

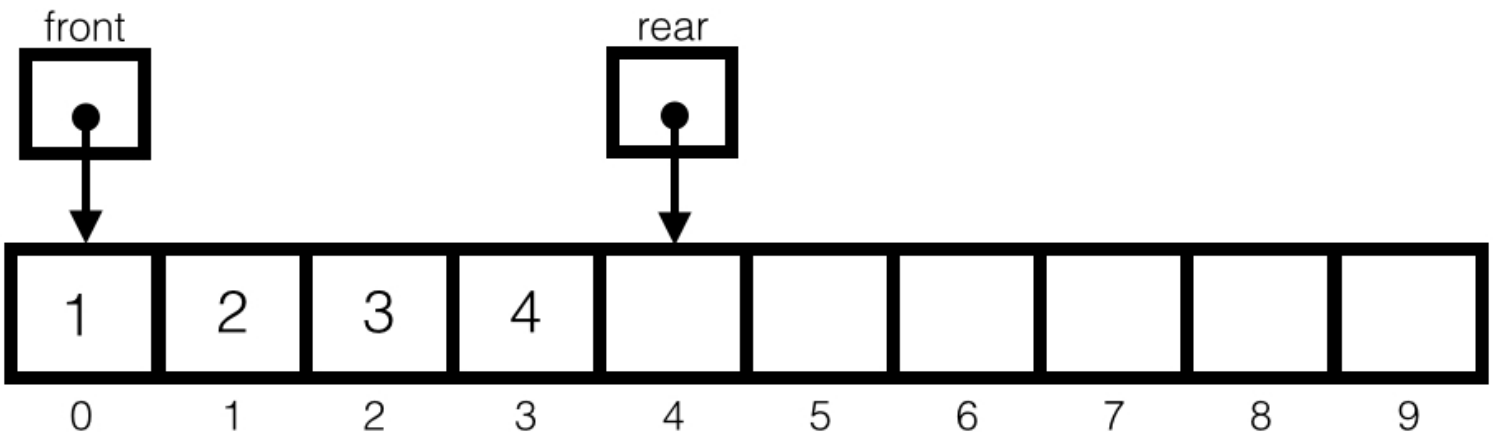
- Dequeue entails the removal of node at the front of the list
- Can call the `removeFirst` function defined in `SinglyLinkedList` class

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



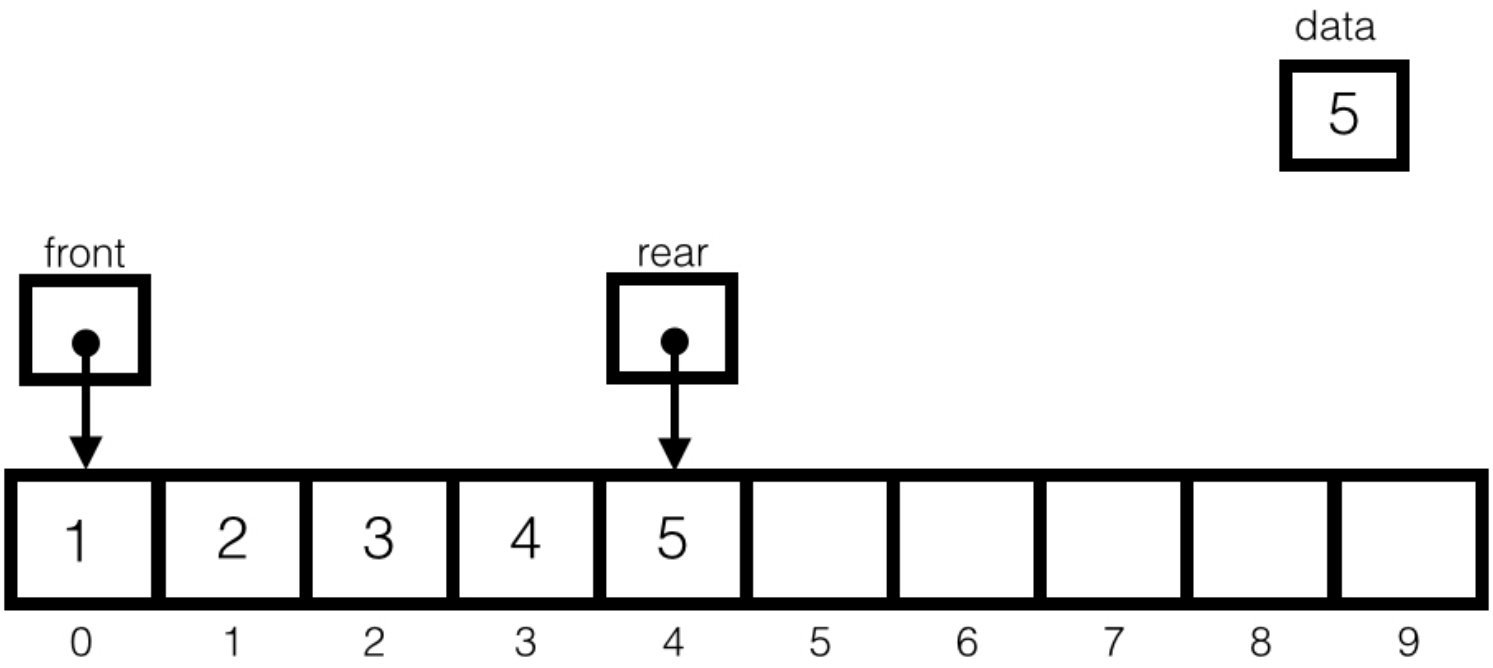
Array Implementation of Queue

Enqueuing Nodes into a Queue



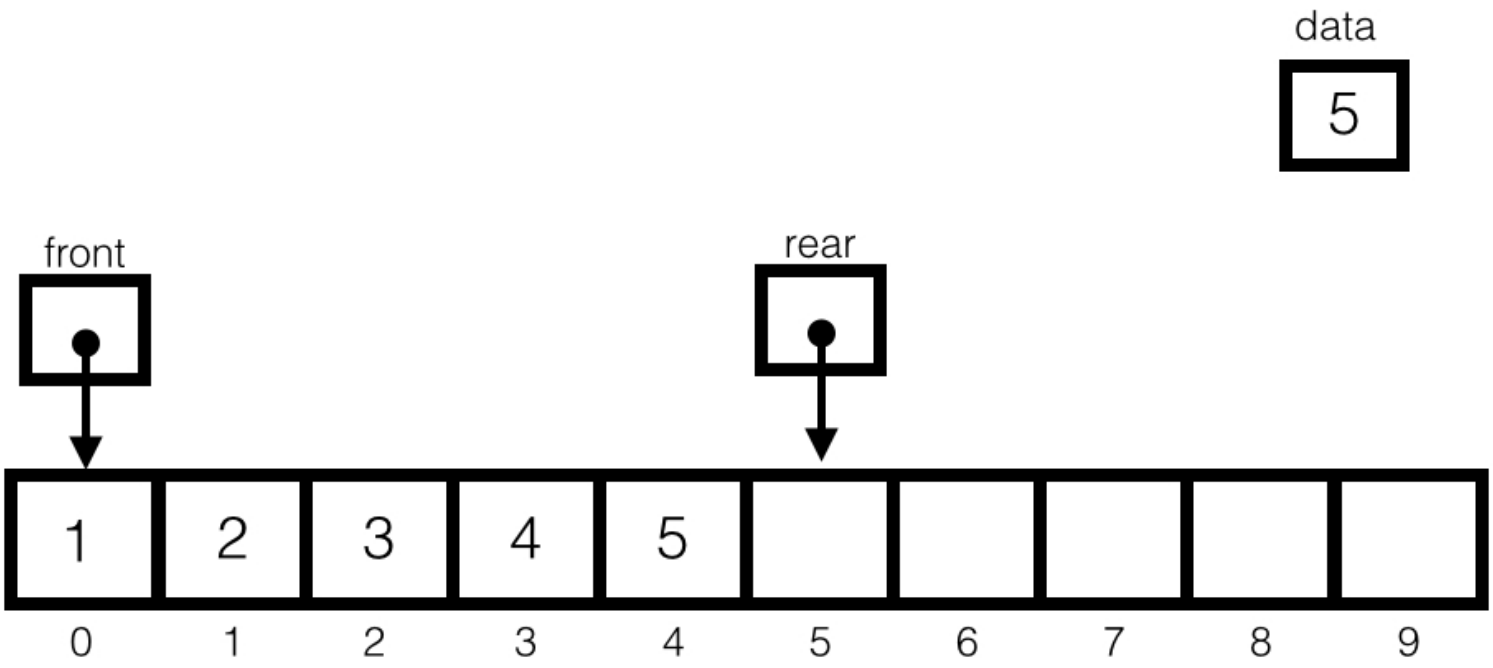
Array Implementation of Queue

Enqueuing Nodes into a Queue



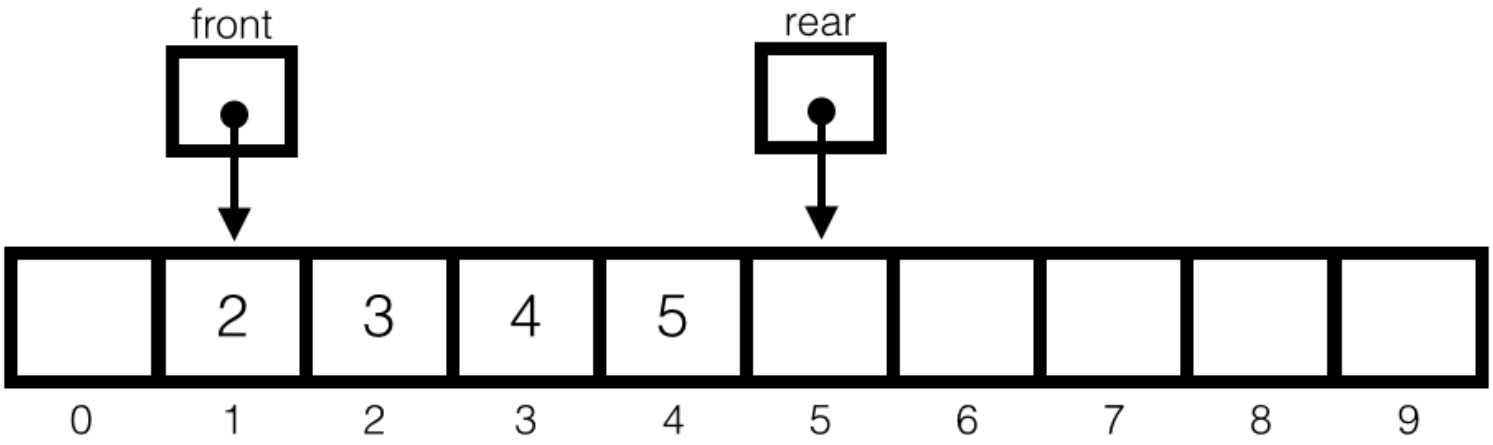
Array Implementation of Queue

Enqueuing Nodes into a Queue



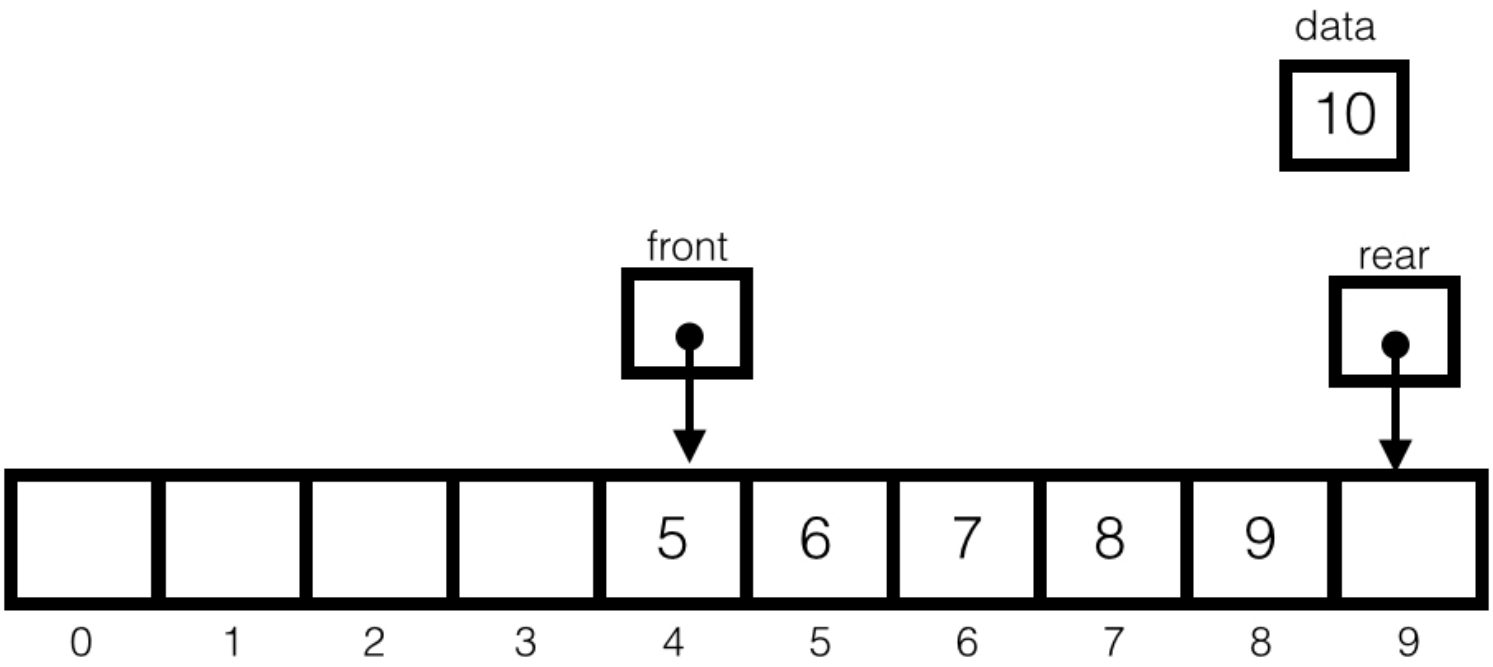
Array Implementation of Queue

Dequeuing Nodes from a Queue



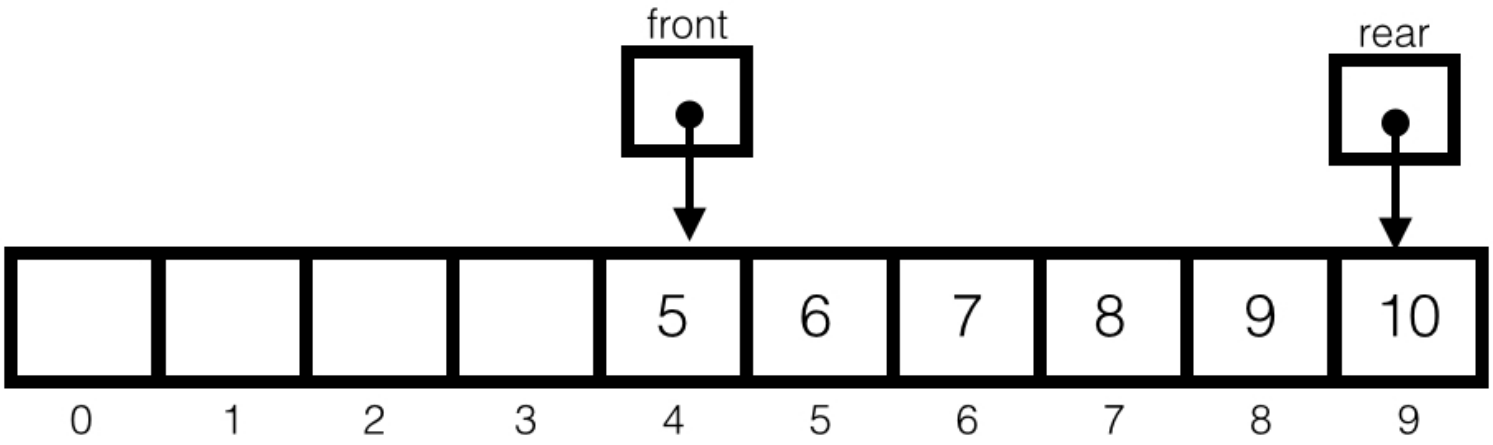
Array Implementation of Queue

What happens when `rear` reaches the end of the array?



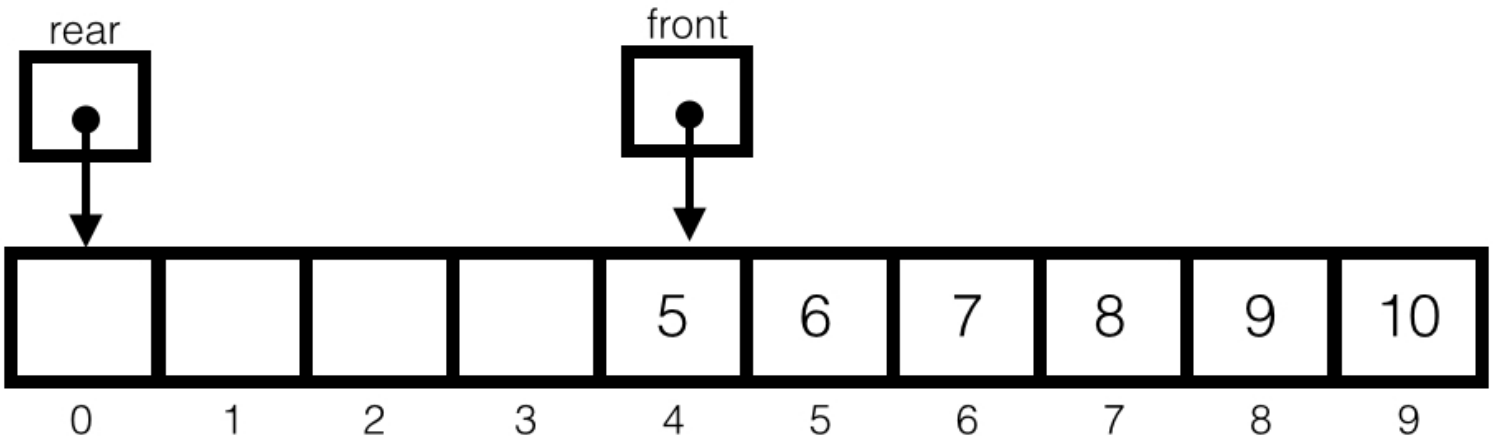
Array Implementation of Queue

What happens when `rear` reaches the end of the array?



Array Implementation of Queue

What happens when `rear` reaches the end of the array?



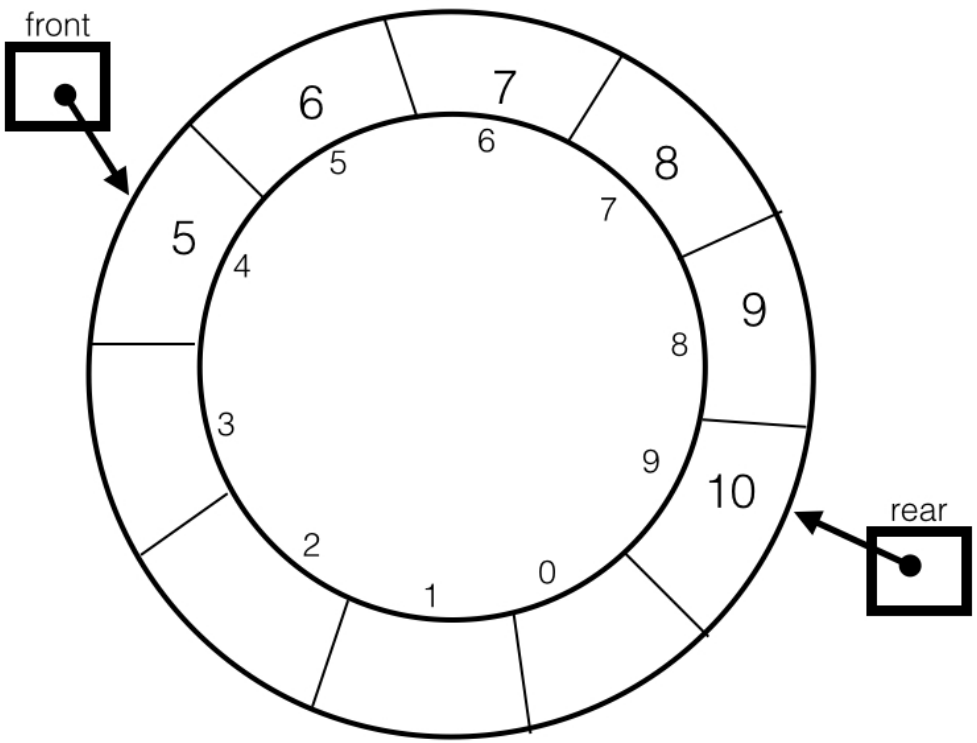
Array Implementation of Queue

What happens when `rear` reaches the end of the array?

- As elements may **wrap around** the queue array, assignment of values to `rear` and `front` should take this into account
- When new nodes are `enqueued` into the queue, `rear=(rear+1)%size`
- When nodes are `dequeued` from the queue, `front=(front+1)%size`
- This implementation is called the **circular array**

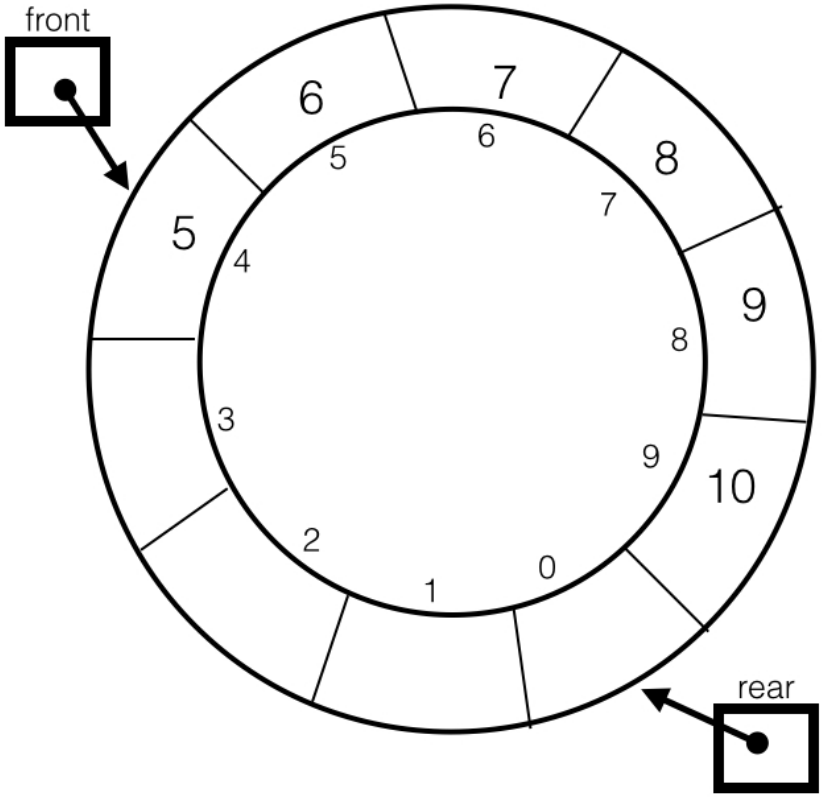
Array Implementation of Queue

Circular Array



Array Implementation of Queue

Circular Array



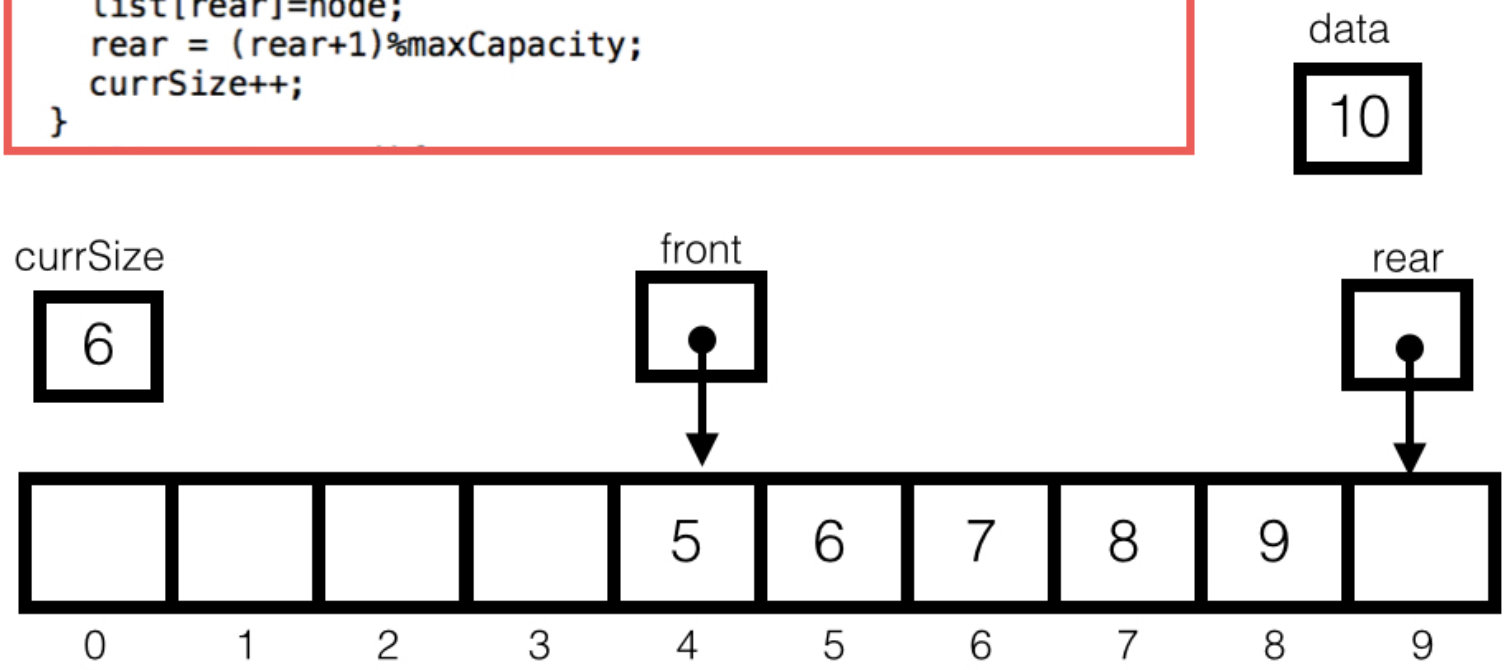
Array Implementation of Queue: Components

```java
public class ArrayListQueue<E> implements Queue<E>{
  private E[] list;
  private int maxCapacity=2, front=0, rear=0, currSize=0;
  public ArrayListQueue(){
    list = (E[])new Object[maxCapacity];
  }
  public int size(){return list.length;}
  public boolean isEmpty(){
    if(currSize==0)
      return true;
    else
      return false;
  }
  public E first(){
    return list[front];
  }
  public void enqueue(E node)throws IllegalStateException{
    if(currSize==maxCapacity)
       throw new IllegalStateException("Queue is full");
    list[rear]=node;
    rear = (rear+1)%maxCapacity;
    currSize++;
  }
  public E dequeue(){
    if(currSize==0)
      return null;
    E e = list[front];
    front=(front+1)%maxCapacity;
    currSize--;
    return e;
  }
}
```

Initialization

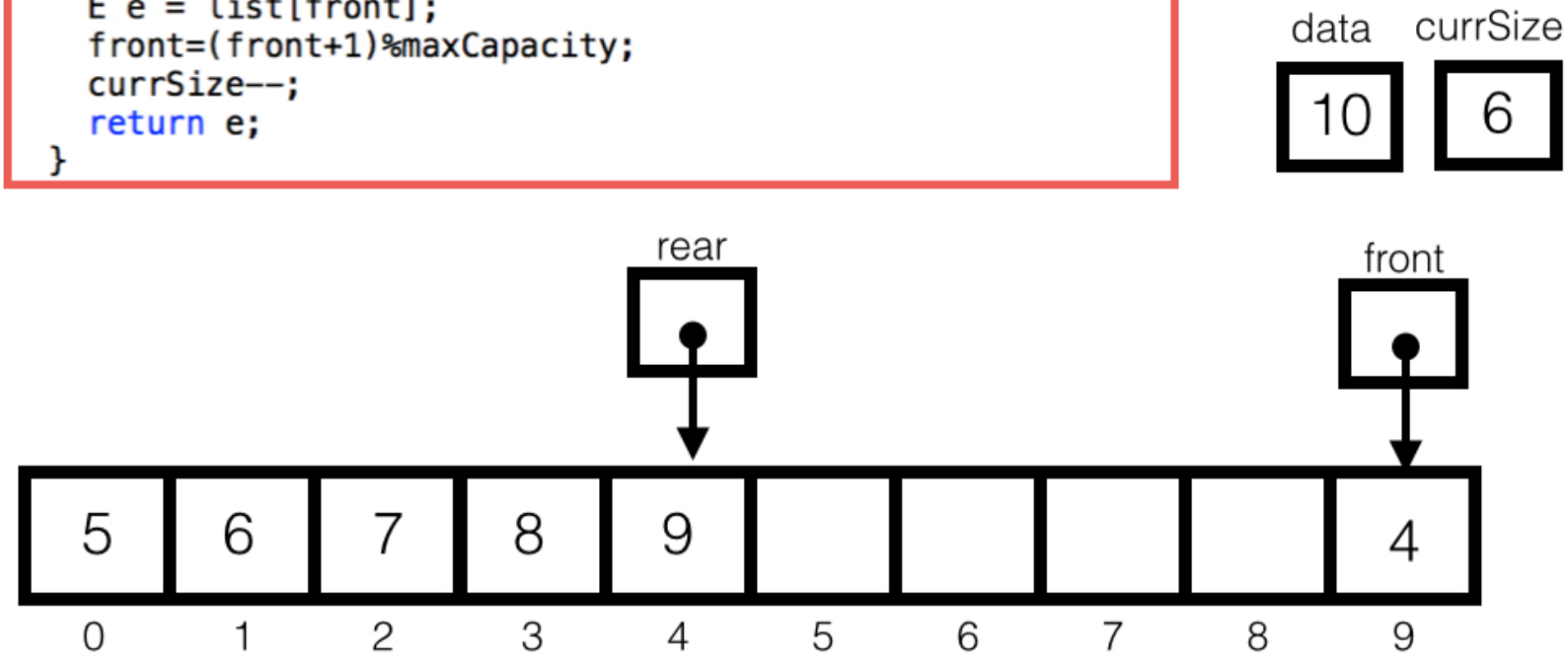
Array Implementation of Queue: Enqueue

```
public void enqueue(E node)throws IllegalStateException{
  if(currSize==maxCapacity)
     throw new IllegalStateException("Queue is full");
  list[rear]=node;
  rear = (rear+1)%maxCapacity;
  currSize++;
}
```



Array Implementation of Queue: Dequeue

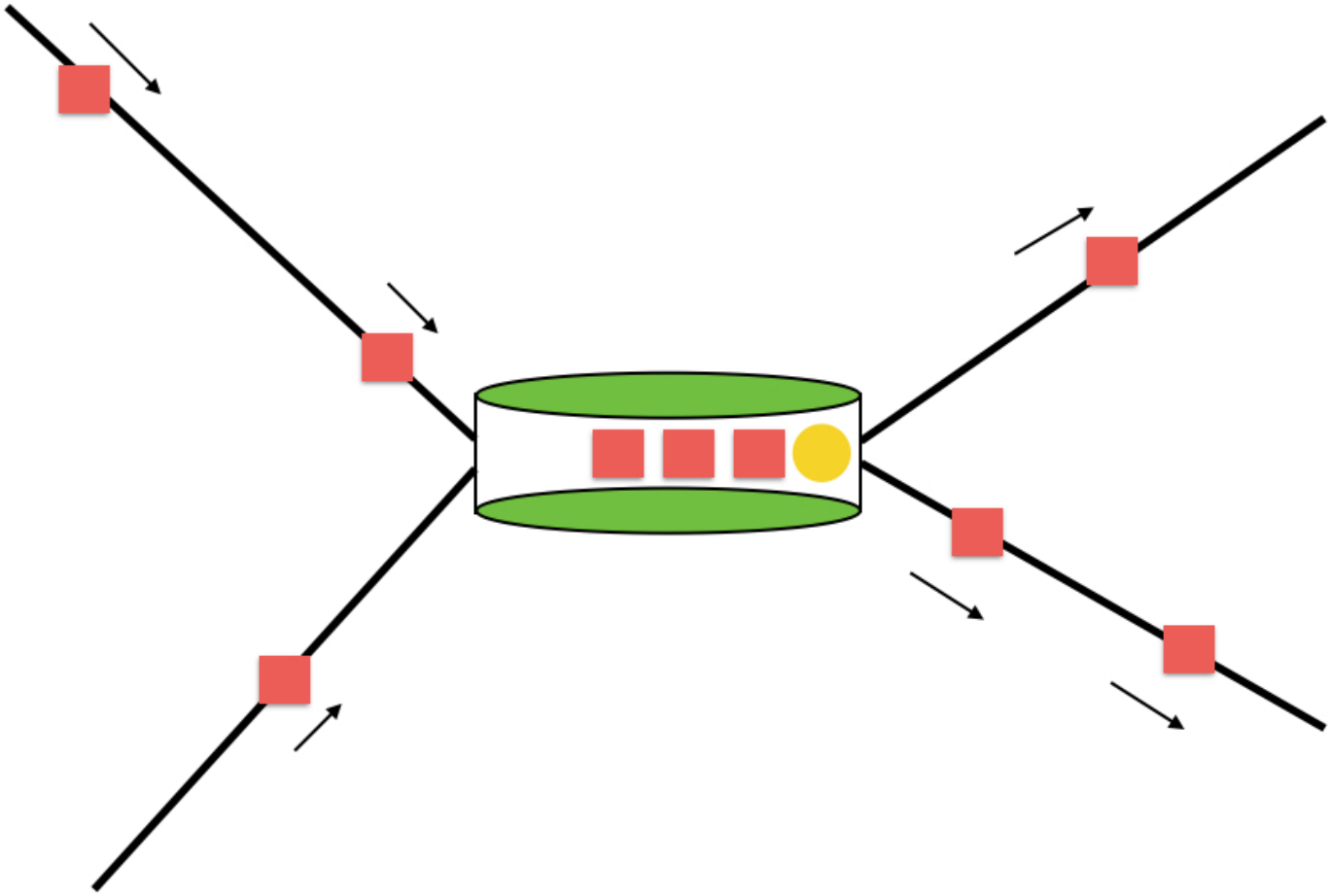
```
public E dequeue(){
  if(currSize==0)
    return null;
  E e = list[front];
  front=(front+1)%maxCapacity;
  currSize--;
  return e;
}
```



Comparison of Array and Linked Implementations

- **Linked implementation**:
  - Not limited to finite queues
  - Enqueuing and dequeuing are both O(1) operations
  - Space consumption is dynamic (i.e. grows as nodes are inserted)
- **Array implementation**:
  - Limited to finite queues
  - Enqueuing and dequeuing are both O(1) operations
  - Space consumption is static (i.e. fixed amount of memory is consumed)

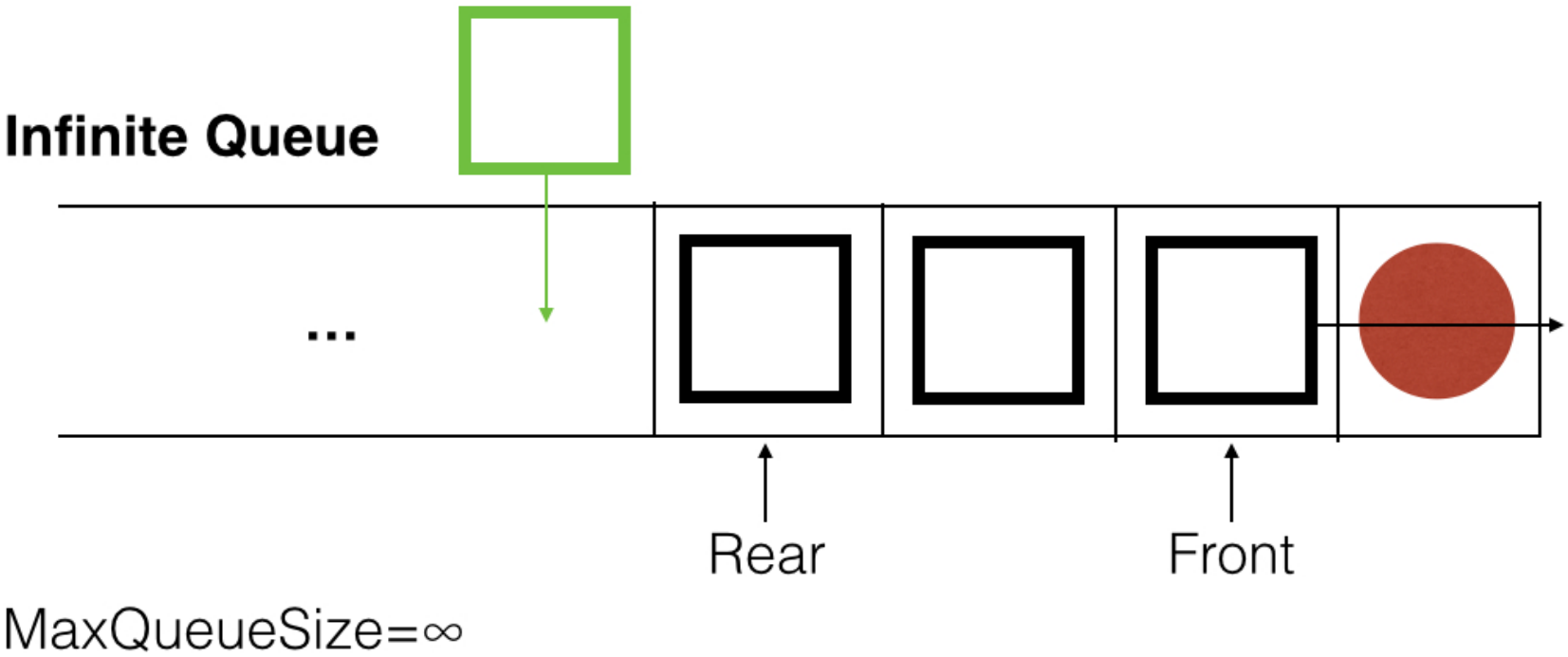
Simulating Congestion in a Data Network



Simulating Congestion in a Data Network

- Packets of data arrive at a router from various sources
- Packets of data arrive at the router and wait in a buffer (the buffer is like a queue)
- These packets are forwarded by that router to various links based on the final destination of these packets
- Assume that:
  - Packets arrive at a fixed rate (arrival rate λ)
  - Routers service each packet for a fixed period of time time
  - There is unlimited space in the buffer/queue

Simulating Congestion in a Data Network



Some Interesting Questions!

- How does the arrival rate of the packets affect how full the queue gets (i.e. congestion)?
- When does the average waiting time transition from reasonable to unreasonable?

Table of Contents

1. Introduction to Data Structures
2. Linked Lists
3. Queues
4. Stacks
5. List
6. Iterators
7. Java Collections Framework

Introduction to Stacks

- **Stack** is an Abstract Data Type (ADT)
- Stack is a **linear data structure**
  - Collection of nodes in a queue are arranged in a **sequence** or a straight line
- Components are added and removed from a stack in a **particular** manner
  - These operations distinguish a stack from other ADTs such as queues and linked lists
- There may or may not be a restriction on the number of nodes in a stack
- Stacks can be implemented via **linked** (linked lists) or **sequential** (arrays) representation

Introduction to Stacks

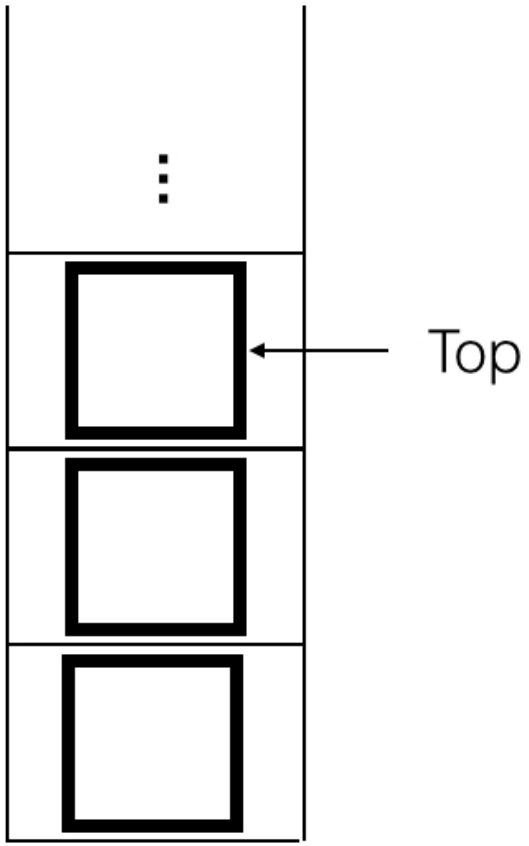
- Stacks are analogous to piling of items one on top of the other (e.g. stacking of books)
- Those who arrive at the stack first are served last (bottom of the book pile)
- This stacking policy is known **last in first out** (LIFO)
- Stacks are useful for processing nested structures, subprocesses or postponed obligations
- Examples include:
  - Storing program information about **function calls** (variables, parameters, etc.) in memory
  - Used to **evaluate** recursive functions, algebraic expressions
  - **Backtracking** in an algorithm
  - Aids with evaluation of expressions or parsing through syntax

Interfaces to Stacks

- Only two main operations are permitted on stacks: `push` and `pop`
- Nodes are **inserted** into the top of the stack via `push`
- Nodes are **removed** from the top of the stack via `pop`
- Logistical operations are: `isEmpty` and `isFull`

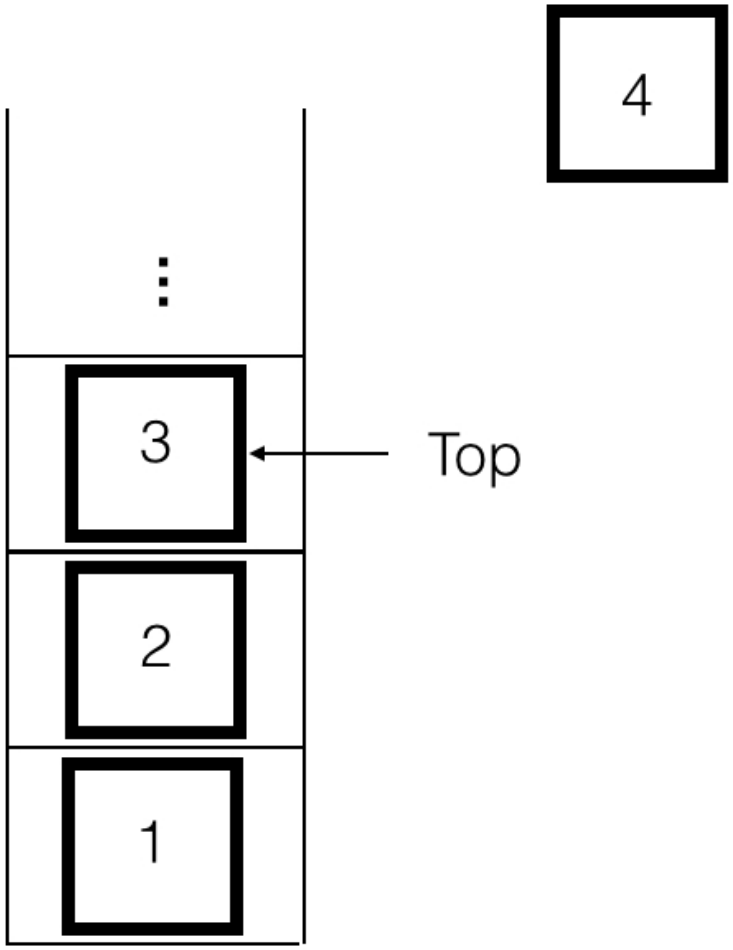
Push into a Stack: A Visualization

Stacks



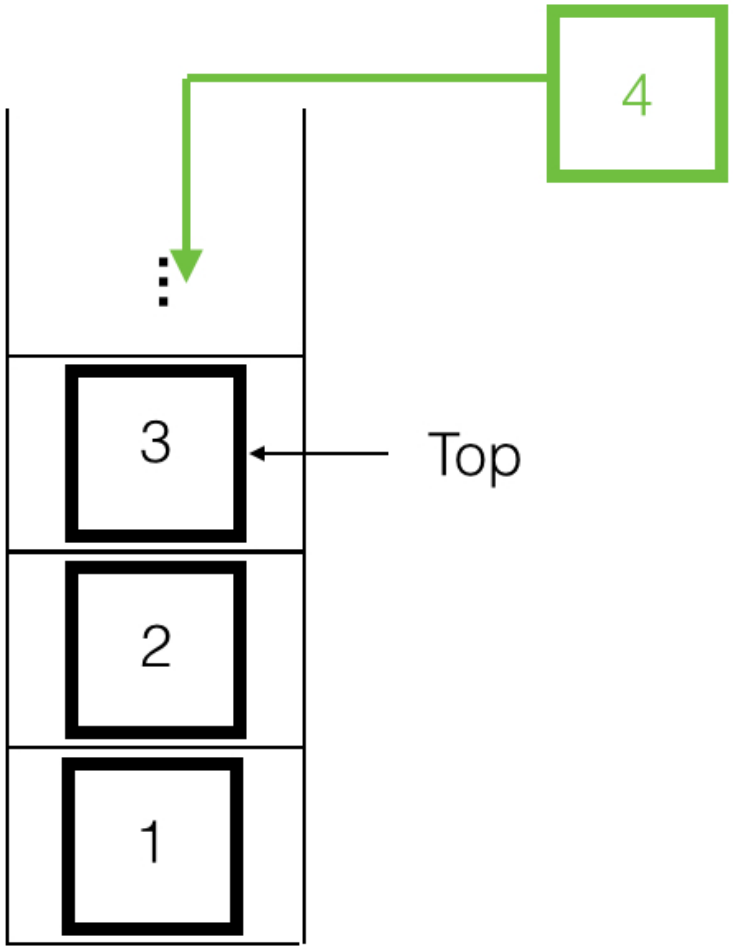
Push into a Stack: A Visualization

Stacks



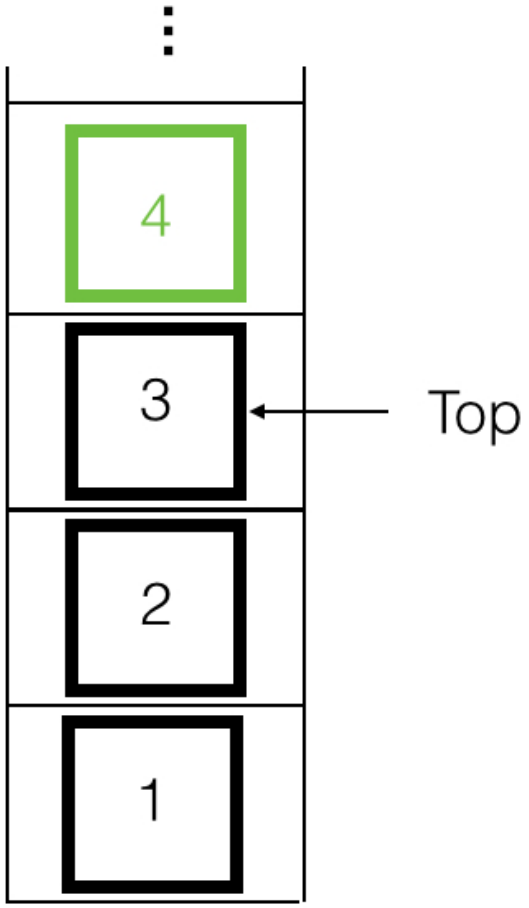
Push into a Stack: A Visualization

Stacks



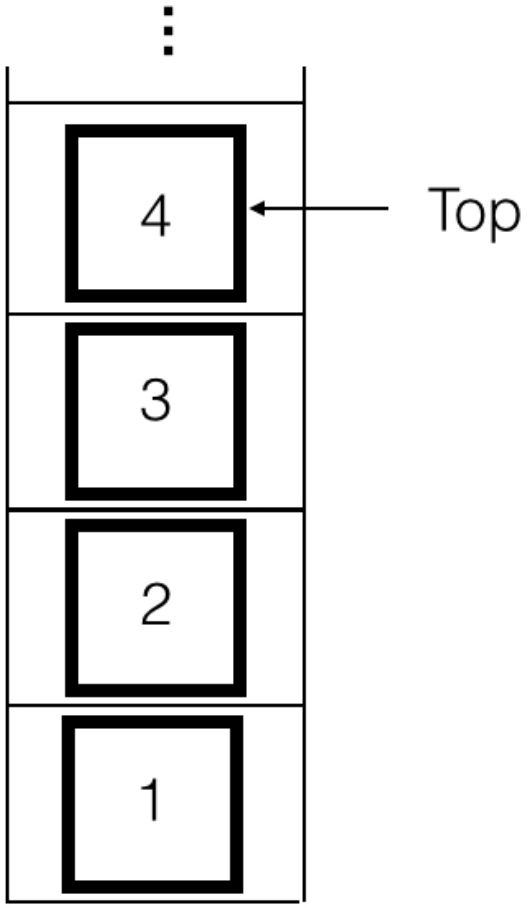
Push into a Stack: A Visualization

Stacks

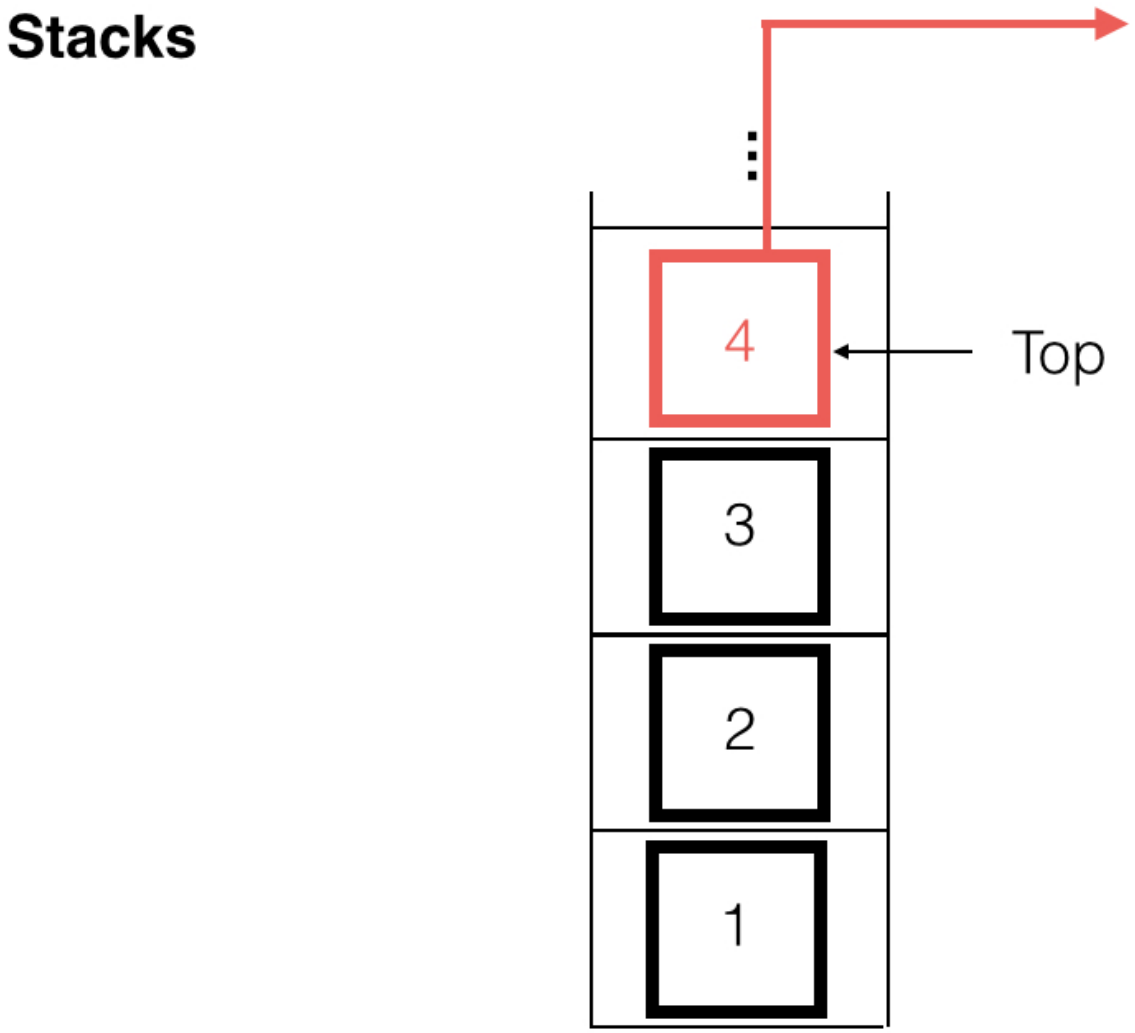


Pop from a Stack: A Visualization

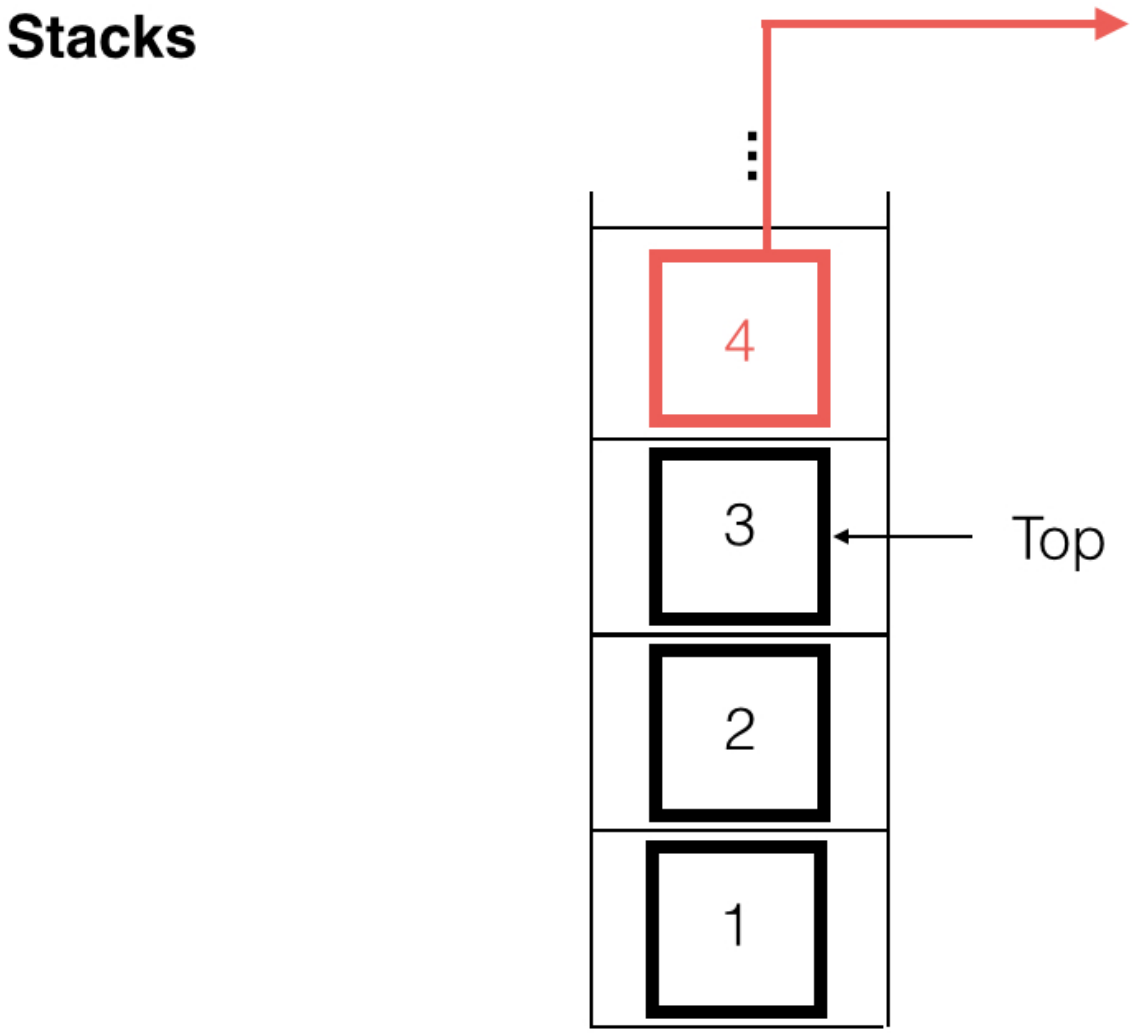
Stacks



Pop from a Stack: A Visualization

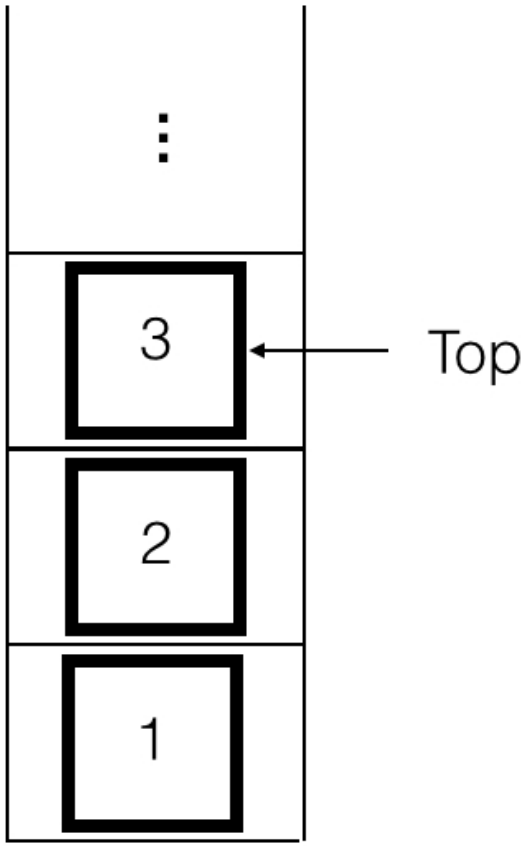


Pop from a Stack: A Visualization



Pop from a Stack: A Visualization

Stacks



Implementation of a Stack: Interface

- Composed of `pop` and `push` methods
- Can be implemented via **singly linked list** or **array**
- Array has a limited capacity while singly linked list can be extended as long as memory exists
- All operations are executed at the **end** of the ADT

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

Linked List Implementation of a Stack

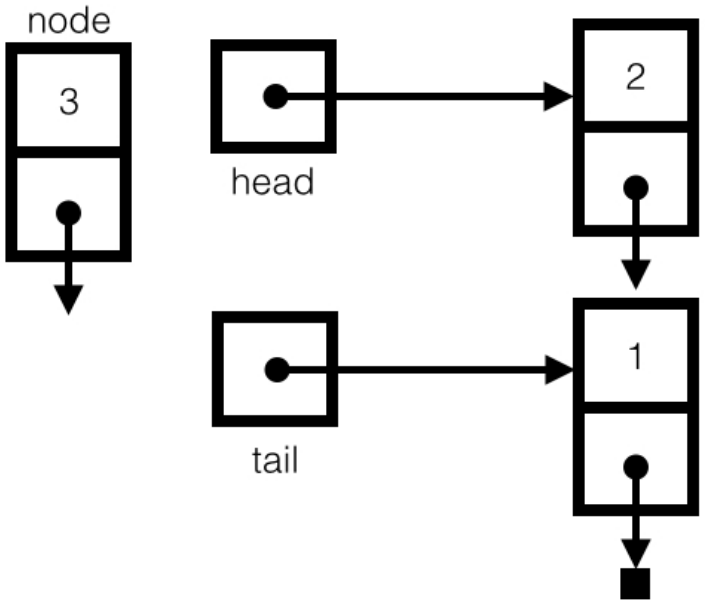
- Use `addFirst` and `removeFirst` methods in the `SinglyLinkedList` class

```
public class LinkedListStack<E> implements Stack<E>{
  private SinglyLinkedList<E> list = new SinglyLinkedList<E>();
  public LinkedListStack(){}
  public int size(){
    return list.size();
  }
  public boolean isEmpty(){
    return list.isEmpty();
  }
  public E top(){
    return list.first();
  }
  public void push(E node){
    list.addFirst(node);
  }
  public E pop(){
    return list.removeFirst();
  }
}
```

Initialization

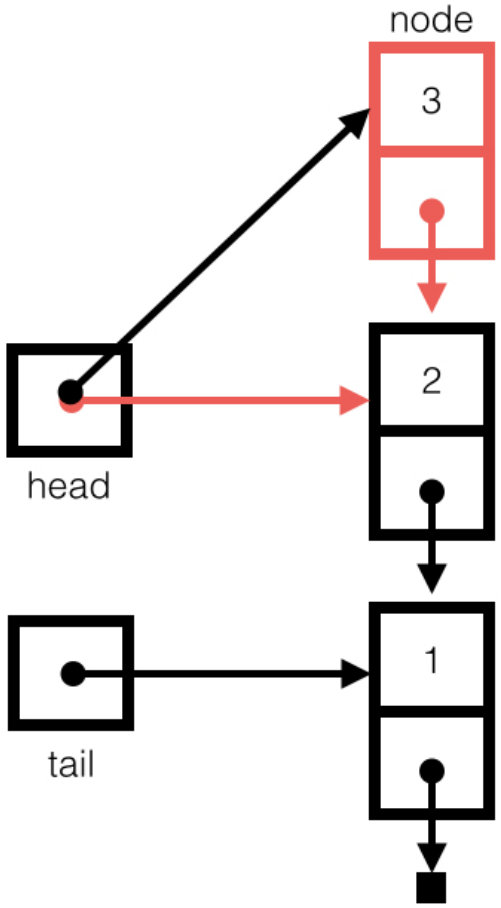
Linked List Implementation of a Stack: Push

```
public class LinkedListStack<E> implements Stack<E>{
  private SinglyLinkedList<E> list = new SinglyLinkedList<E>();
  public LinkedListStack(){}
  public int size(){
    return list.size();
  }
  public boolean isEmpty(){
    return list.isEmpty();
  }
  public E top(){
    return list.first();
  }
  public void push(E node){
    list.addFirst(node);
  }
  public E pop(){
    return list.removeFirst();
  }
}
```



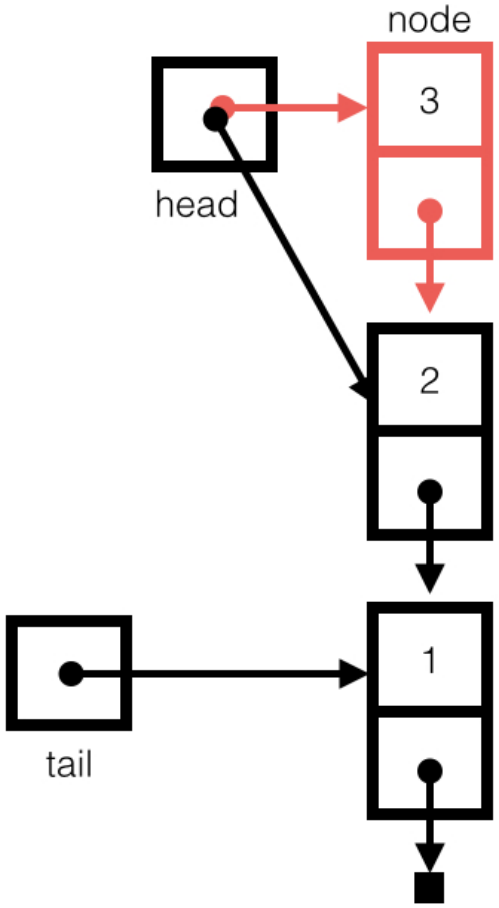
Linked List Implementation of a Stack: Push

```
public class LinkedListStack<E> implements Stack<E>{
  private SinglyLinkedList<E> list = new SinglyLinkedList<E>();
  public LinkedListStack(){}
  public int size(){
    return list.size();
  }
  public boolean isEmpty(){
    return list.isEmpty();
  }
  public E top(){
    return list.first();
  }
  public void push(E node){
    list.addFirst(node);
  }
  public E pop(){
    return list.removeFirst();
  }
}
```



Linked List Implementation of a Stack: Pop

```
public class LinkedListStack<E> implements Stack<E>{
  private SinglyLinkedList<E> list = new SinglyLinkedList<E>();
  public LinkedListStack(){}
  public int size(){
    return list.size();
  }
  public boolean isEmpty(){
    return list.isEmpty();
  }
  public E top(){
    return list.first();
  }
  public void push(E node){
    list.addFirst(node);
  }
  public E pop(){
    return list.removeFirst();
  }
}
```



Array Implementation of a Stack

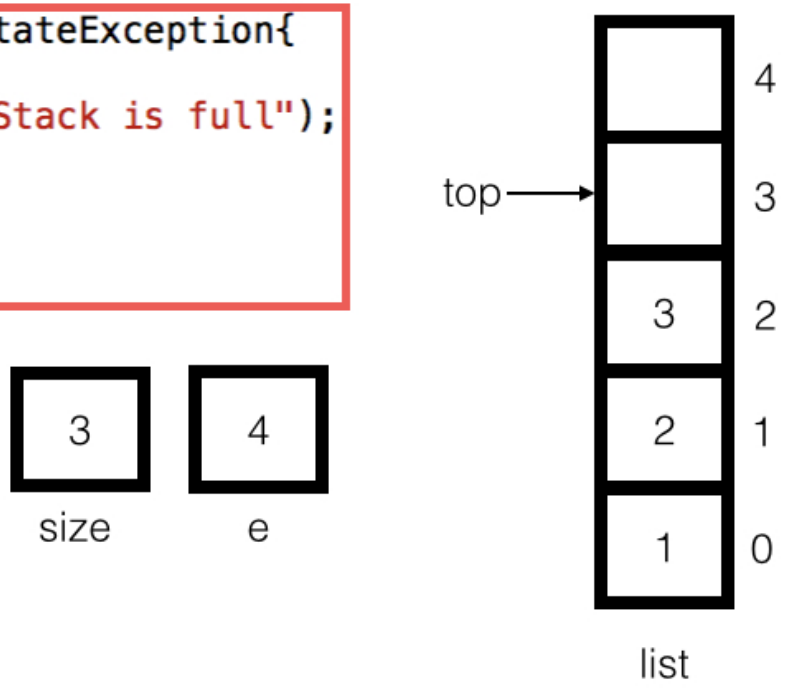
- Keep track of only the last index where a new element can be inserted

```
public class ArrayStack<E> implements Stack<E>{
  private E[] list;
  public static final int maxCapacity=10;
  private int top=0, size=0;
  public ArrayStack(){list = (E[])new Object[maxCapacity];}
  public int size(){return size;}
  public boolean isEmpty(){
    if(size==0)
      return true;
    else
      return false;
  }
  public void push(E e) throws IllegalStateException{
    if(size==maxCapacity)
      throw new IllegalStateException("Stack is full");
    list[top]=e;
    top++;
    size++;
  }
  public E pop(){
    if(size==0)
      return null;
    top--;
    E e = list[top];
    size--;
    return e;
  }
  public E top(){return list[top-1];}
}
```

Initialization

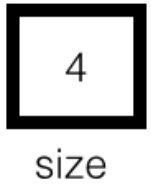
Array Implementation of a Stack: Push

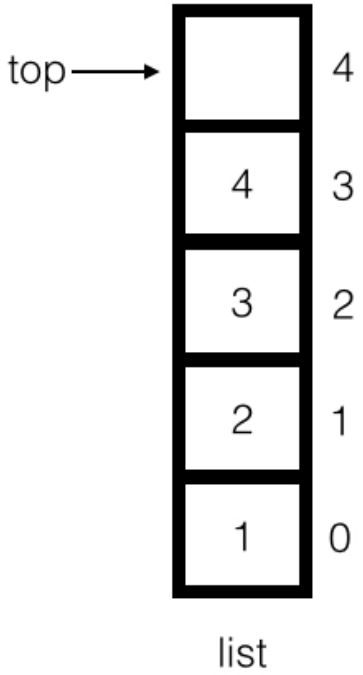
```
public class ArrayStack<E> implements Stack<E>{
  private E[] list;
  public static final int maxCapacity=10;
  private int top=0, size=0;
  public ArrayStack(){list = (E[])new Object[maxCapacity];}
  public int size(){return size;}
  public boolean isEmpty(){
    if(size==0)
      return true;
    else
      return false;
  }
  public void push(E e) throws IllegalStateException{
    if(size==maxCapacity)
      throw new IllegalStateException("Stack is full");
    list[top]=e;
    top++;
    size++;
  }
  public E pop(){
    if(size==0)
      return null;
    top--;
    E e = list[top];
    size--;
    return e;
  }
  public E top(){return list[top-1];}
}
```



Array Implementation of a Stack: Push

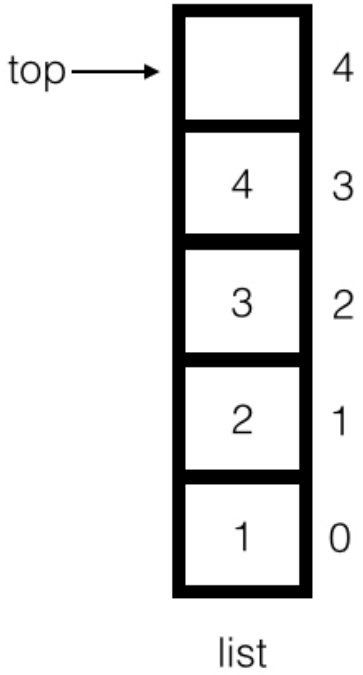
```
public class ArrayStack<E> implements Stack<E>{
  private E[] list;
  public static final int maxCapacity=10;
  private int top=0, size=0;
  public ArrayStack(){list = (E[])new Object[maxCapacity];}
  public int size(){return size;}
  public boolean isEmpty(){
    if(size==0)
      return true;
    else
      return false;
  }
  public void push(E e) throws IllegalStateException{
    if(size==maxCapacity)
      throw new IllegalStateException("Stack is full");
    list[top]=e;
    top++;
    size++;
  }
  public E pop(){
    if(size==0)
      return null;
    top--;
    E e = list[top];
    size--;
    return e;
  }
  public E top(){return list[top-1];}
}
```





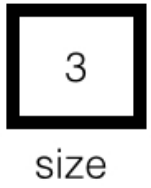
Array Implementation of a Stack: Pop

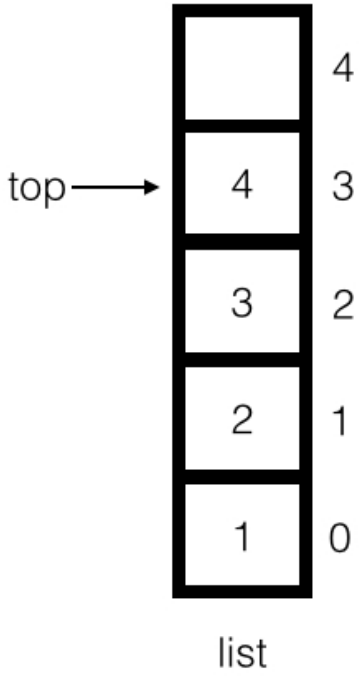
```
public class ArrayStack<E> implements Stack<E>{
  private E[] list;
  public static final int maxCapacity=10;
  private int top=0, size=0;
  public ArrayStack(){list = (E[])new Object[maxCapacity];}
  public int size(){return size;}
  public boolean isEmpty(){
    if(size==0)
      return true;
    else
      return false;
  }
  public void push(E e) throws IllegalStateException{
    if(size==maxCapacity)
      throw new IllegalStateException("Stack is full");
    list[top]=e;
    top++;
    size++;
  }
  public E pop(){
    if(size==0)
      return null;
    top--;
    E e = list[top];
    size--;
    return e;
  }
  public E top(){return list[top-1];}
}
```



Array Implementation of a Stack: Pop

```java
public class ArrayStack<E> implements Stack<E>{
  private E[] list;
  public static final int maxCapacity=10;
  private int top=0, size=0;
  public ArrayStack(){list = (E[])new Object[maxCapacity];}
  public int size(){return size;}
  public boolean isEmpty(){
    if(size==0)
      return true;
    else
      return false;
  }
  public void push(E e) throws IllegalStateException{
    if(size==maxCapacity)
      throw new IllegalStateException("Stack is full");
    list[top]=e;
    top++;
    size++;
  }
  public E pop(){
    if(size==0)
      return null;
    top--;
    E e = list[top];
    size--;
    return e;
  }
  public E top(){return list[top-1];}
}
```





Stack Applications: Popping from a Sequential Stack

- In the next few examples, you will see how a stack can be useful ADT for applications
- These applications use stacks as **building blocks** and will not be affected by the underlying implementation

Stack Applications: Balancing Parenthesis

Balancing Parenthesis

Does this expression have a balanced parenthesis?

{a+b*[c+d]}

How about this expression?

{a+b*[c+d}

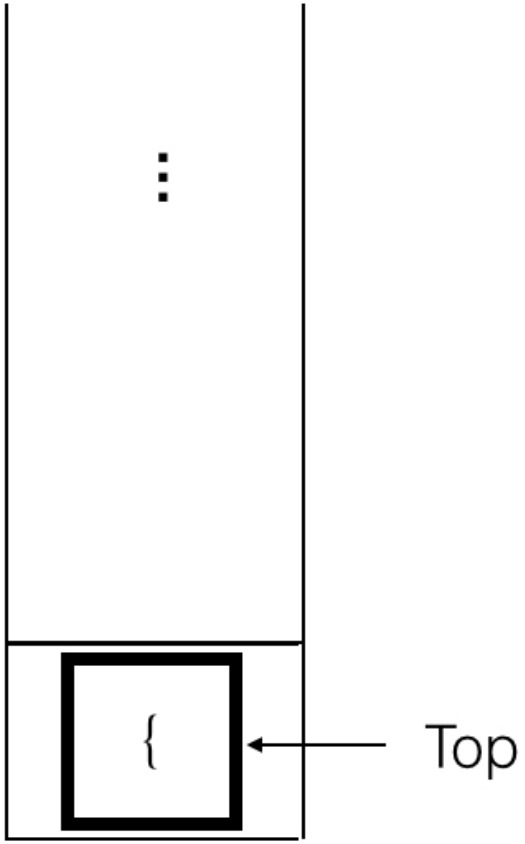
Stack Applications: Balancing Parenthesis

- Can use a stack to check whether parentheses are balanced
- Scan through the input string holding the expression
- When left parentheses are encountered during the scanning, push these into the stack
- As soon as a right parenthesis is encountered, pop the stack and see if the left parenthesis matches with the right
- Continue until the stack is empty

Stack Applications: Balancing Parenthesis

Balancing Parenthesis: {a+b*[c+d]}

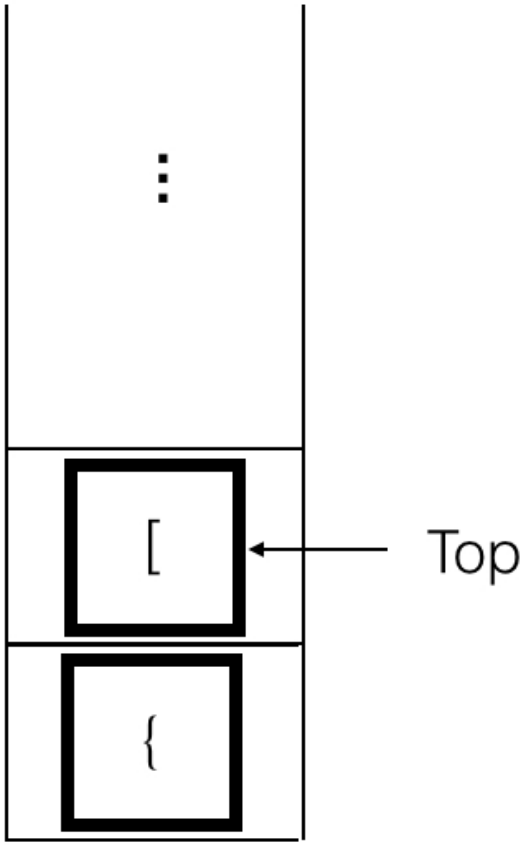
{a+b*[c+d]}



Stack Applications: Balancing Parenthesis

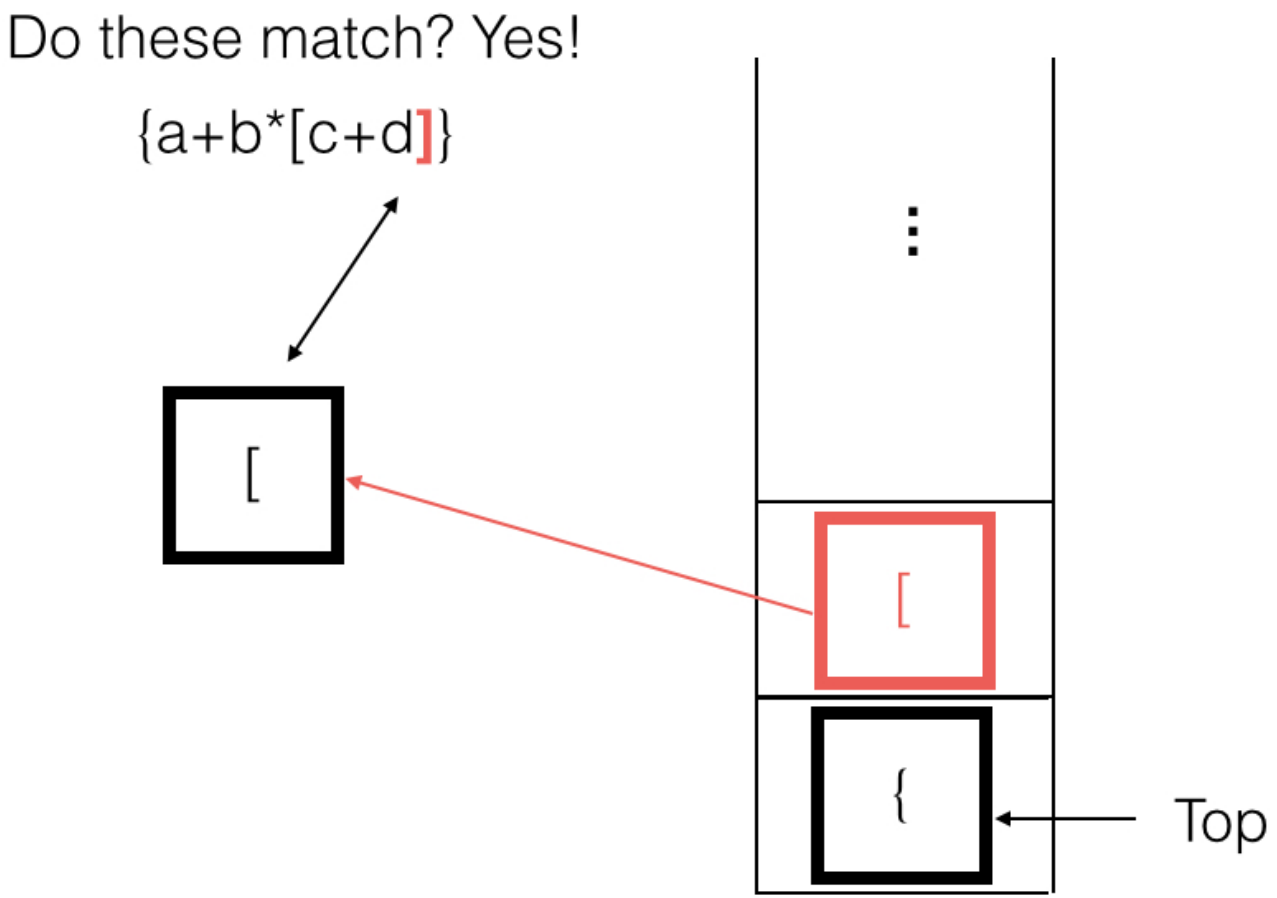
Balancing Parenthesis: {a+b*[c+d]}

{a+b*[c+d]}



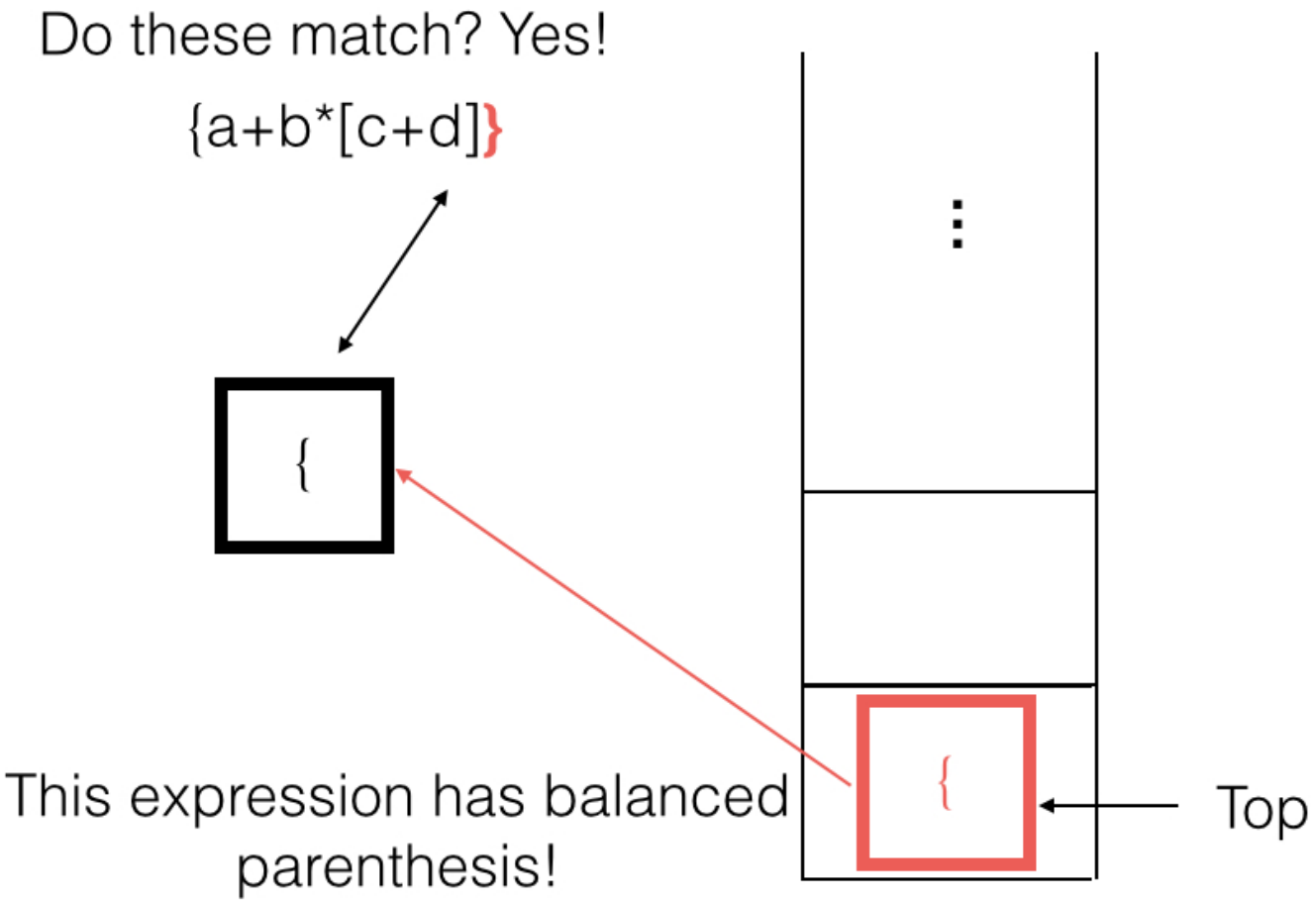
Stack Applications: Balancing Parenthesis

Balancing Parenthesis: {a+b*[c+d]}



Stack Applications: Balancing Parenthesis

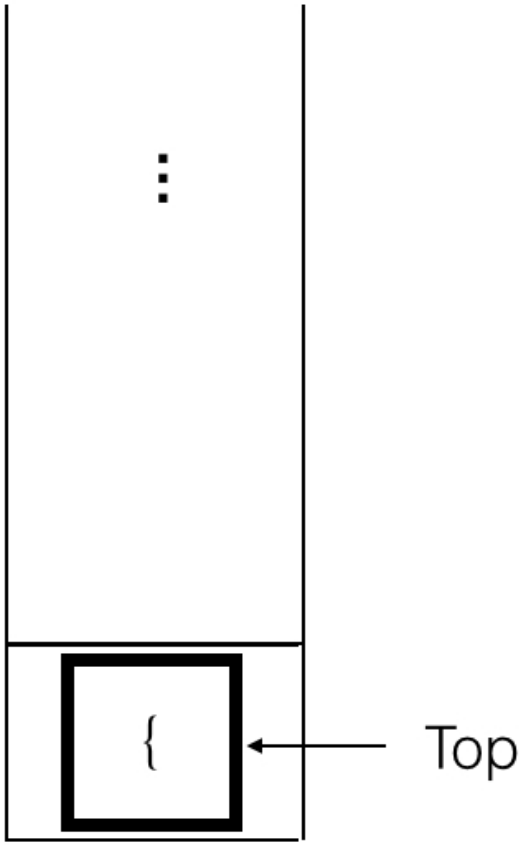
Balancing Parenthesis: {a+b*[c+d]}



Stack Applications: Balancing Parenthesis

Balancing Parenthesis: {a+b*[c+d}

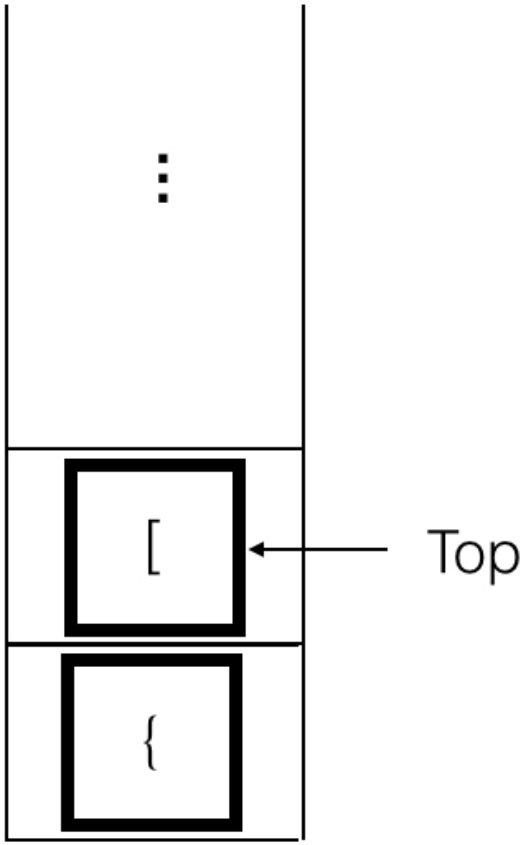
{a+b*[c+d}



Stack Applications: Balancing Parenthesis

Balancing Parenthesis: {a+b*[c+d}

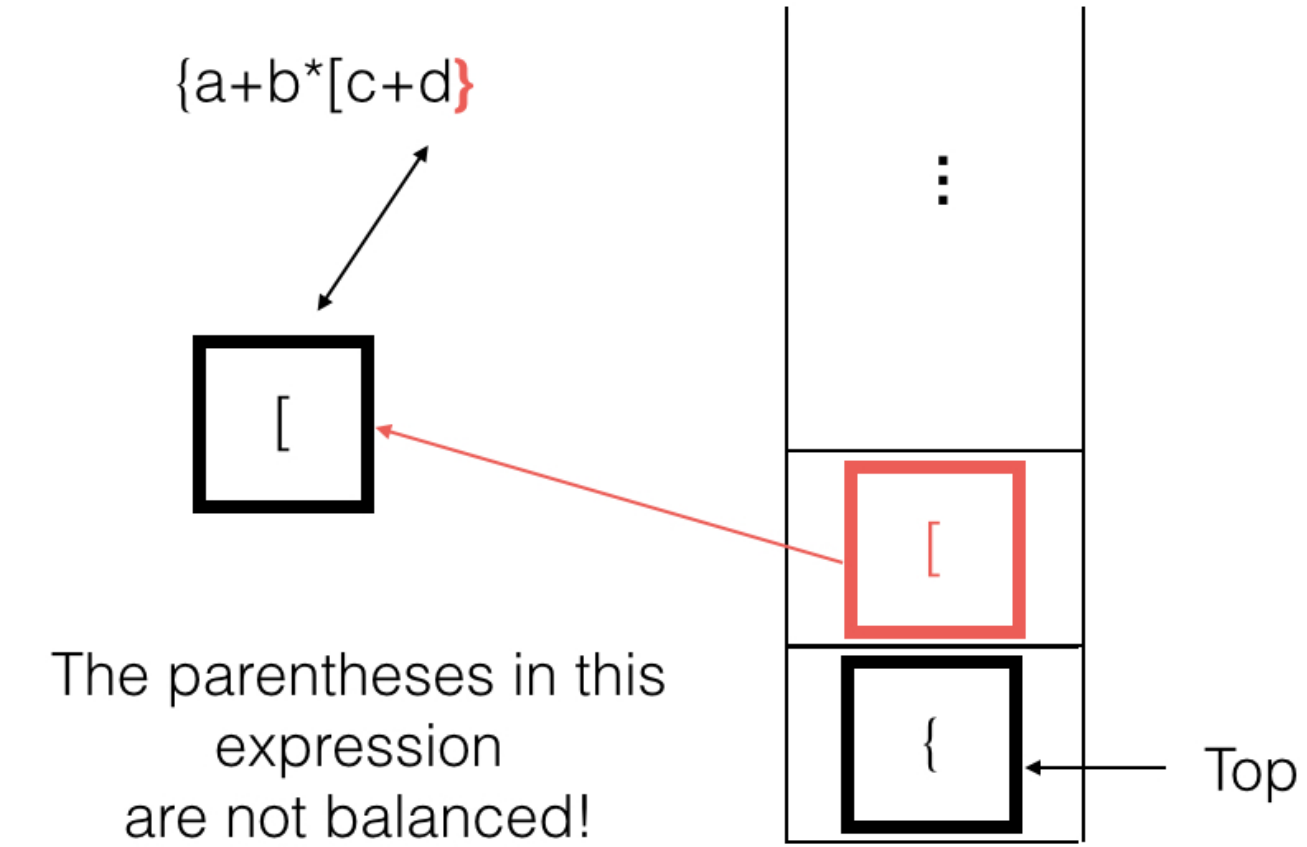
{a+b*[c+d}



Stack Applications: Balancing Parenthesis

Balancing Parenthesis: {a+b*[c+d}

Do these match? No!



The parentheses in this expression are not balanced!

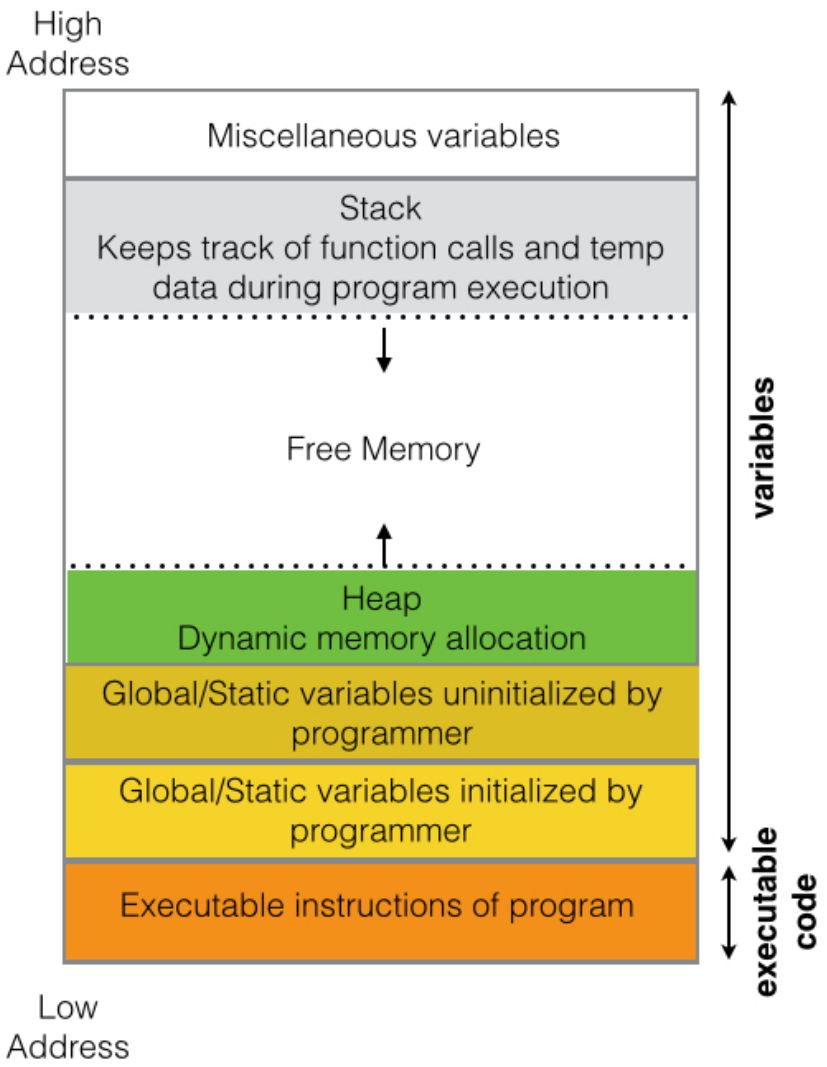
Stack Applications: Balancing Parenthesis Implementation

- Check if the inner-most brackets are balanced and then move out
- As inner-most bracket must be processed first, LIFO policy fits well
- Scan the string from left to right
- If there is a parenthesis (e.g. "(", ")", "[", "]", "{", "}",), need to identify whether this is a left or right parenthesis
- If it is a left parenthesis, need to insert into the stack (as you will be waiting for the balancing right parenthesis)
- If it is a right parenthesis, need to remove the first element in the stack and see if there is a match between the parenthesis
- When there is no match, the parentheses are not balanced!

Implement `MatchParenthesis` class

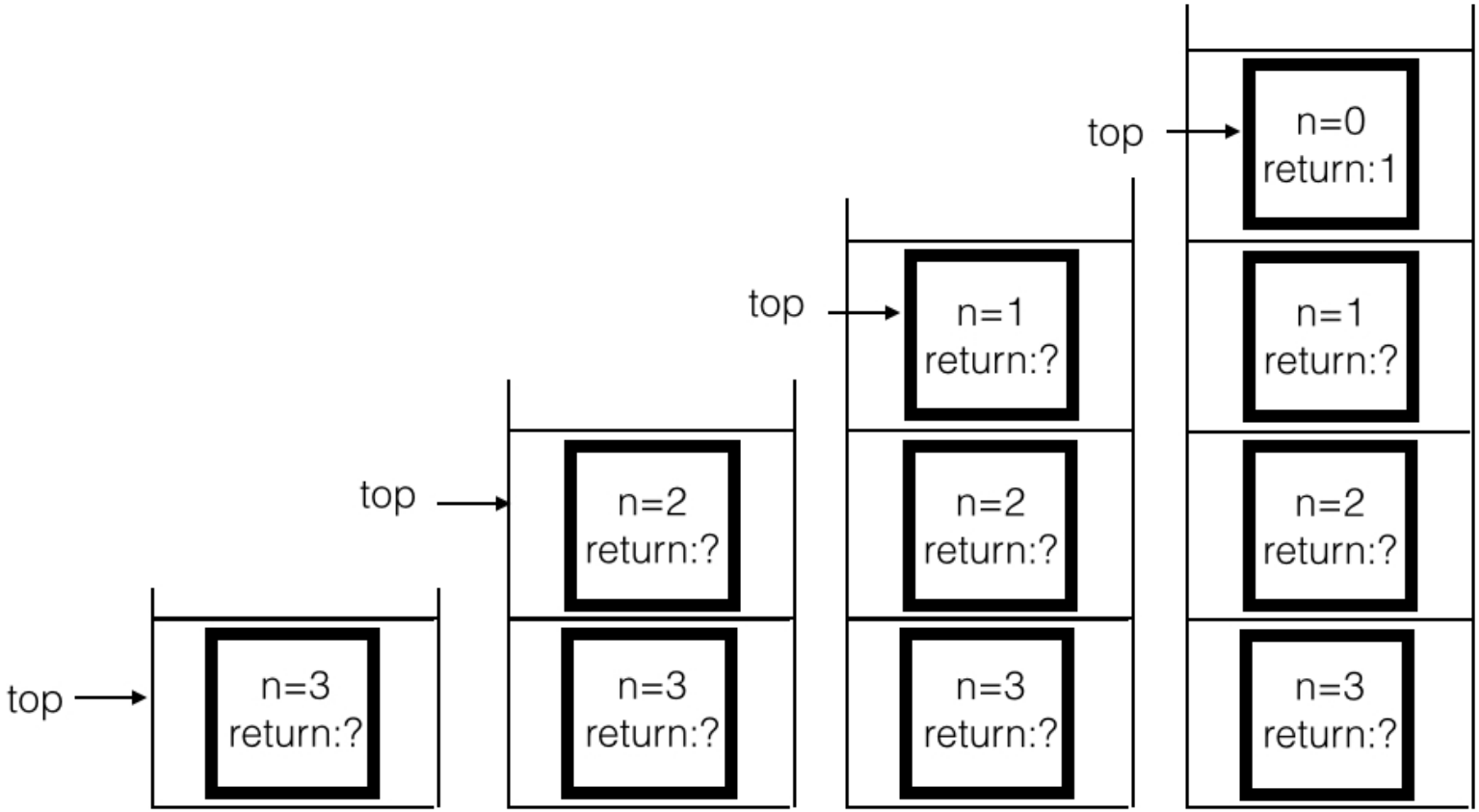
Stack Applications: Recursive Function Calls

Revisiting the Memory Diagram



Stack Applications: Recursive Function Calls

```
public int powerOf2(int n){
  if(n==0)
    return 1;
  else
    return 2*powerOf2(n-1);
}
```



Stack Applications: Recursive Function Calls

```
public int powerOf2(int n){
  if(n==0)
    return 1;
  else
    return 2*powerOf2(n-1);
}
```

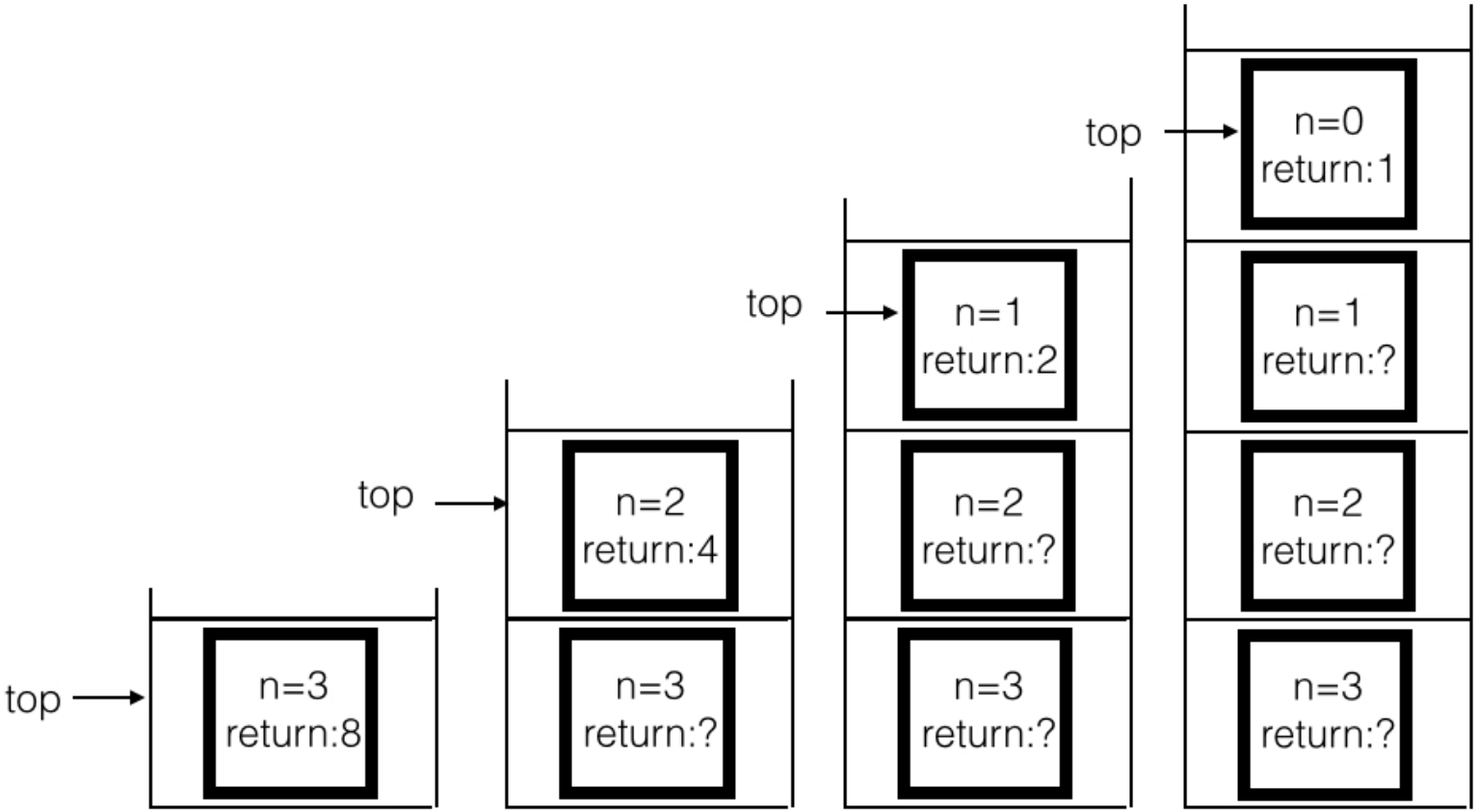


Table of Contents

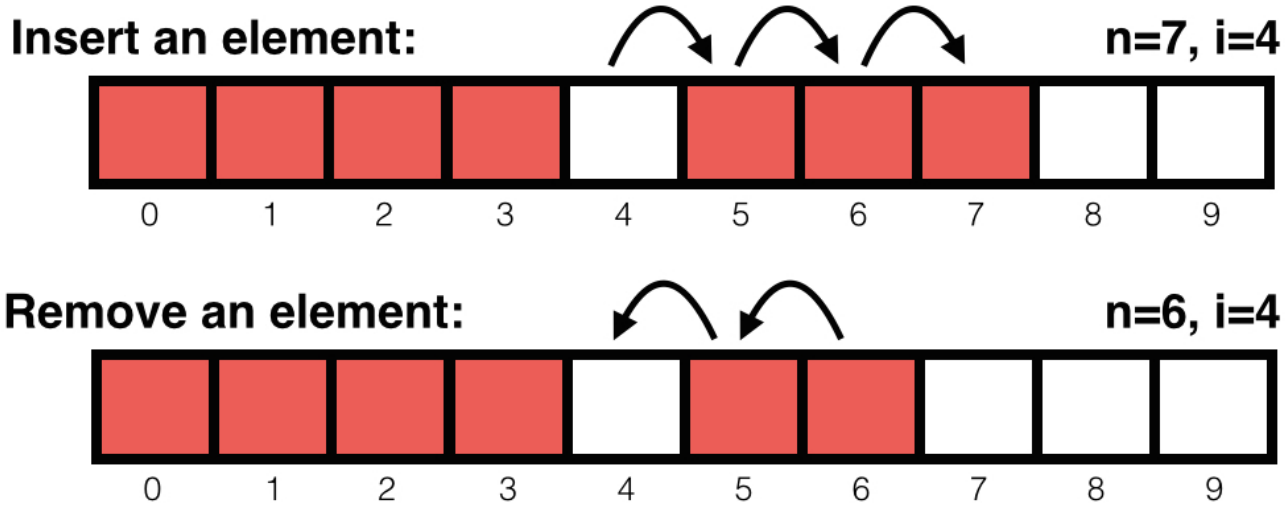
1. Introduction to Data Structures
2. Linked Lists
3. Queues
4. Stacks
5. List
6. Iterators
7. Java Collections Framework

List ADT

- What if you would like to insert at a **particular** position?
- Valid range of positions is 0 to $n-1$ where n is the current number of nodes in the ADT
- This is called the **List** ADT
- Supports methods that focus on operations exacted on a **specific position of the data structure**:
  - `get(i)`: Return the element stored at location i
  - `set(i,e)`: **Replaces** an element in position i with e
  - `add(i,e)`: **Inserts** the element e into position i without replacing the original element in that spot
  - `remove(i)`: **Removes** element at location i and return the element stored at the index
- Indices can change as elements are added and removed

List ADT: Implementation Challenges

- For an **array implementation**, the number of elements that can be added is **limited** (bound to the size of the array)
  - **Dynamic array**: resize the array, everytime it fills up (i.e. create a new array that is double the size and copy all the old elements)
- Replacing an element into an array at position i is $O(1)$ as index can be **directly accessed**
- Inserting an element into an array at position i is $O(n)$ as $n - i$ elements must be **shifted to the right** by one to free up a spot for inserting a new element
- Removing an element at position i is also $O(n)$ as $n - i$ elements must be **shifted to the left**

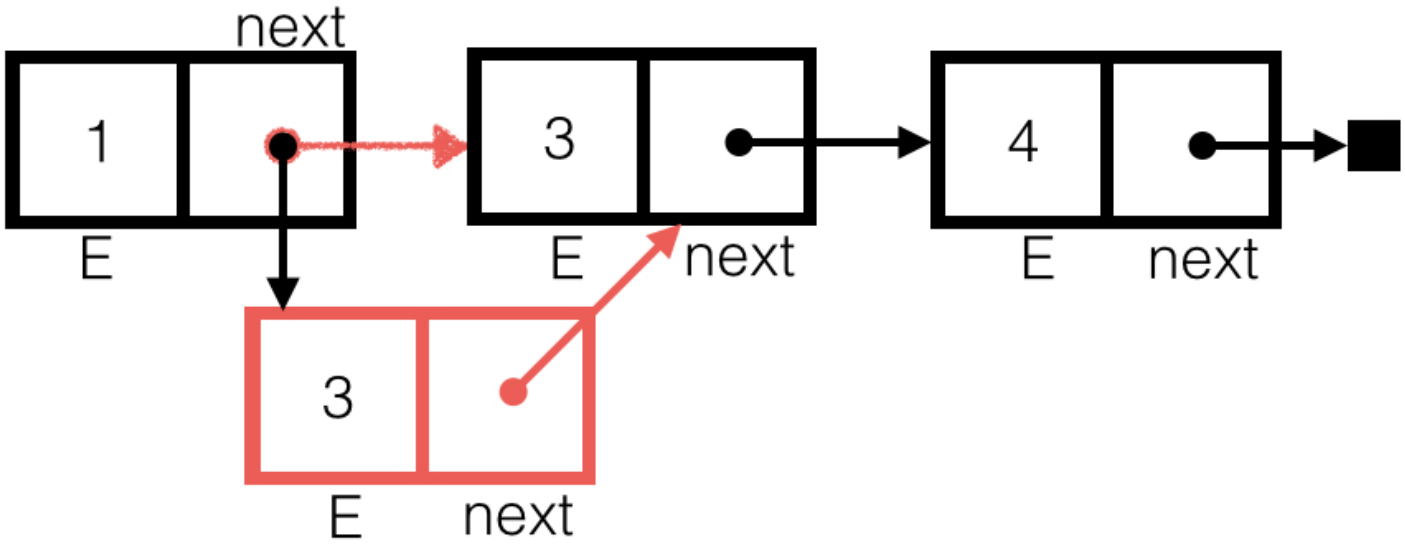


List ADT: Implementation Challenges

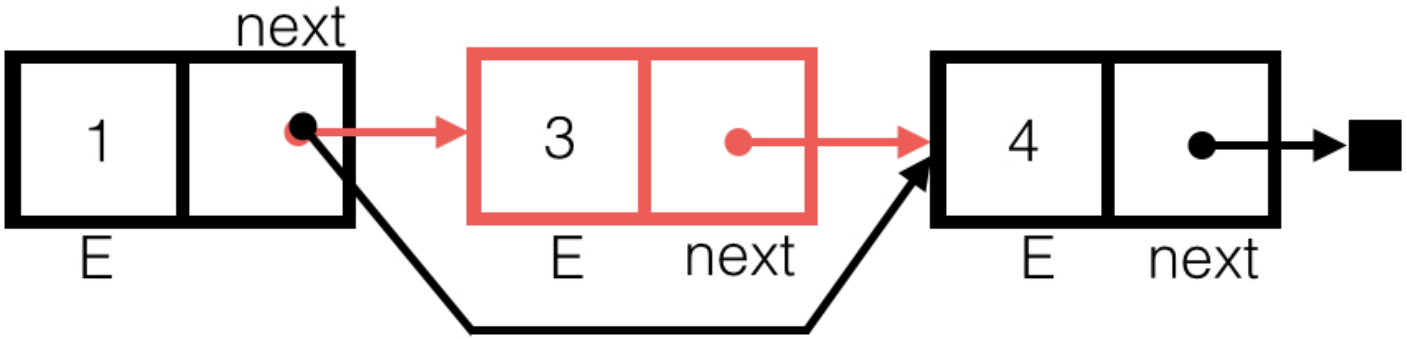
- For a **linked list implementation**, any number of elements can be added as long as there is space in the memory
- Replacing an element into an array at position i is $O(n)$ as index **cannot** be directly accessed (need to traverse through i nodes)
- Inserting an element into an array at position i is $O(n)$ as i nodes are traversed and then a re-arrangement of references occur
- Removing an element at position i is also $O(n)$ as i elements must be traversed

List ADT: Implementation Challenges

Insert an element: i=1



Remove an element: i=1

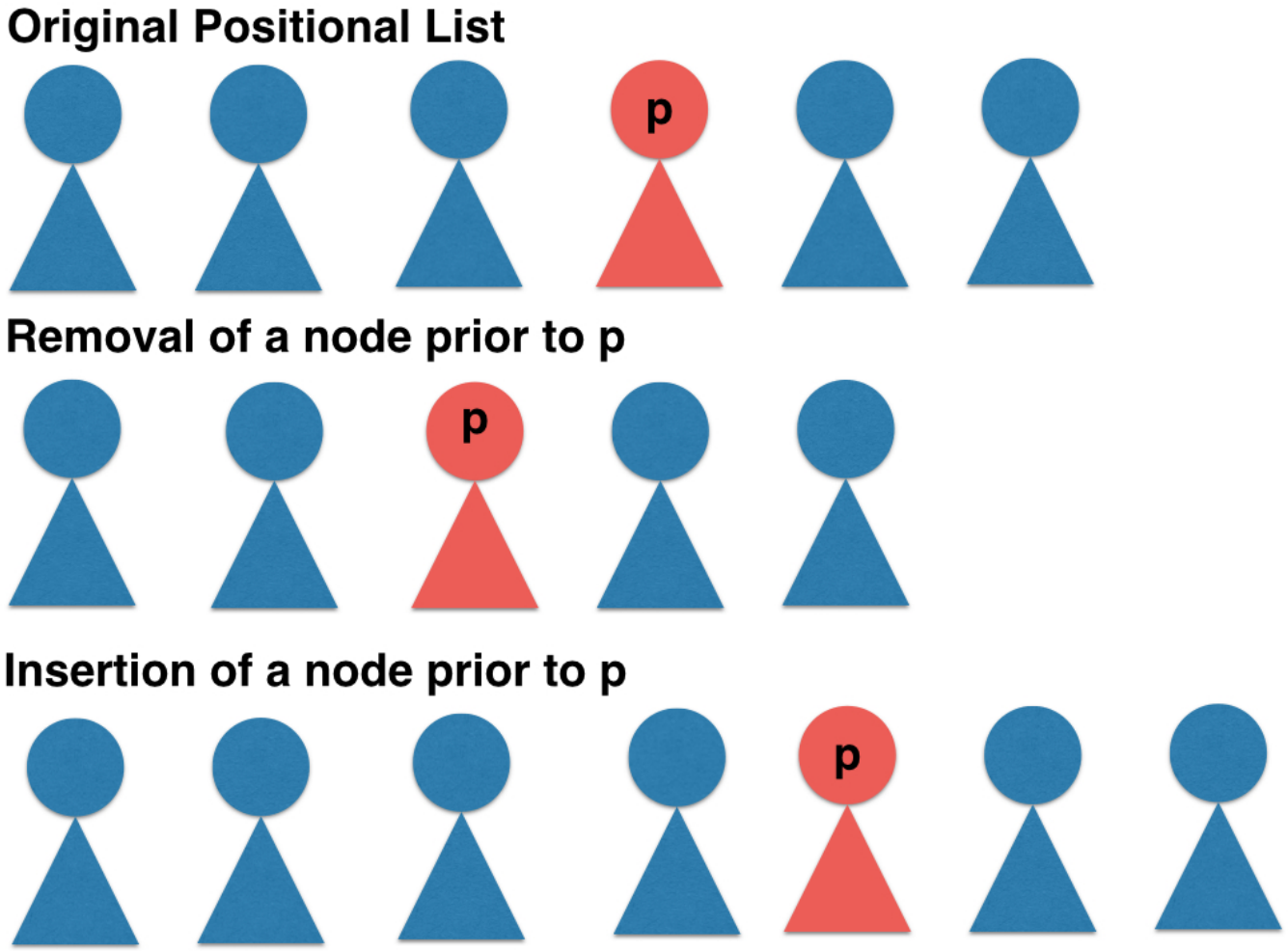


Implement the `List` ADT using the array and linked structures

List ADT: Implementation Challenges

- Traversing i nodes to get/insert/remove a node at position i is **cumbersome**
- Moreover, index i representing the location of an element e **can change** if there are insertions/deletions prior to that position
- Can use a **positional list** which keeps track of the reference to a particular node p (like a cursor)
- Adding or removing elements around position p will be $O(1)$ in a linked list implementation (not so in an array!)
- Addition or removal of elements prior to position p will not affect the position reference

List ADT: Implementation Challenges



Implement the `Positional List` ADT using the array and linked structures

Table of Contents

1. Introduction to Data Structures
2. Linked Lists
3. Queues
4. Stacks
5. List
6. Iterators
7. Java Collections Framework

Iterator

- **Iterator** is a software design pattern that enables the scanning of a sequence of elements one by one
- Supported by `java.util.Iterator`
- Two methods for **scanning** through the sequence: `hasNext()`, `next()`
- Every time `next` is called, the iterator progresses to the next element
- A new iterator can be created using `new iterator()` to create **multiple instances** of the iterations
- `for`-loop and `while`-loop can iterate through collections:

while-loop

```
ArrayList<Double> data;
Iterator<Double> iter = data.iterator();
while(iter.hasNext()){
  Double temp = data.next();
  System.out.println(temp);
}
```

for-loop

```
ArrayList<Double> data;
for(Double temp : data){
  System.out.println(temp);
}
```

Iterator: Implementation

- **Snapshot** iterators: Maintain a **duplicate** copy of the collection and iterate
- Will not be affected by changes to the original data structure
- Requires $O(n)$ storage space and copying complexity
- **Lazy** iterators: Use the **existing** collection to iterate
- Although additional space is not required, will be affected by changes made to the original data structure
- Implementation will depend on the underlying data structure used to represent the collection (i.e. array or linked list)

Table of Contents

1. Introduction to Data Structures
2. Linked Lists
3. Queues
4. Stacks
5. List
6. Iterators
7. Java Collections Framework

Java Collections Framework

- Java supports **built-in** interfaces, concrete classes and methods that **implement** data structures and iterators
- These are part of the `java.util` package:

	Interfaces			Properties			Storage	
Class	Queue	Deque	List	Capacity Limit	Thread-Safe	Blocking	Array	Linked List
ArrayBlockingQueue	✓			✓	✓	✓	✓	
LinkedBlockingQueue	✓			✓	✓	✓		✓
ConcurrentLinkedQueue	✓				✓		✓	
ArrayDeque	✓	✓					✓	
LinkedBlockingDeque	✓	✓		✓	✓	✓		✓
ConcurrentLinkedDeque	✓	✓			✓			✓
ArrayList			✓				✓	
LinkedList	✓	✓	✓					✓

Goodrich et al, 2014