Data Structures

Queues

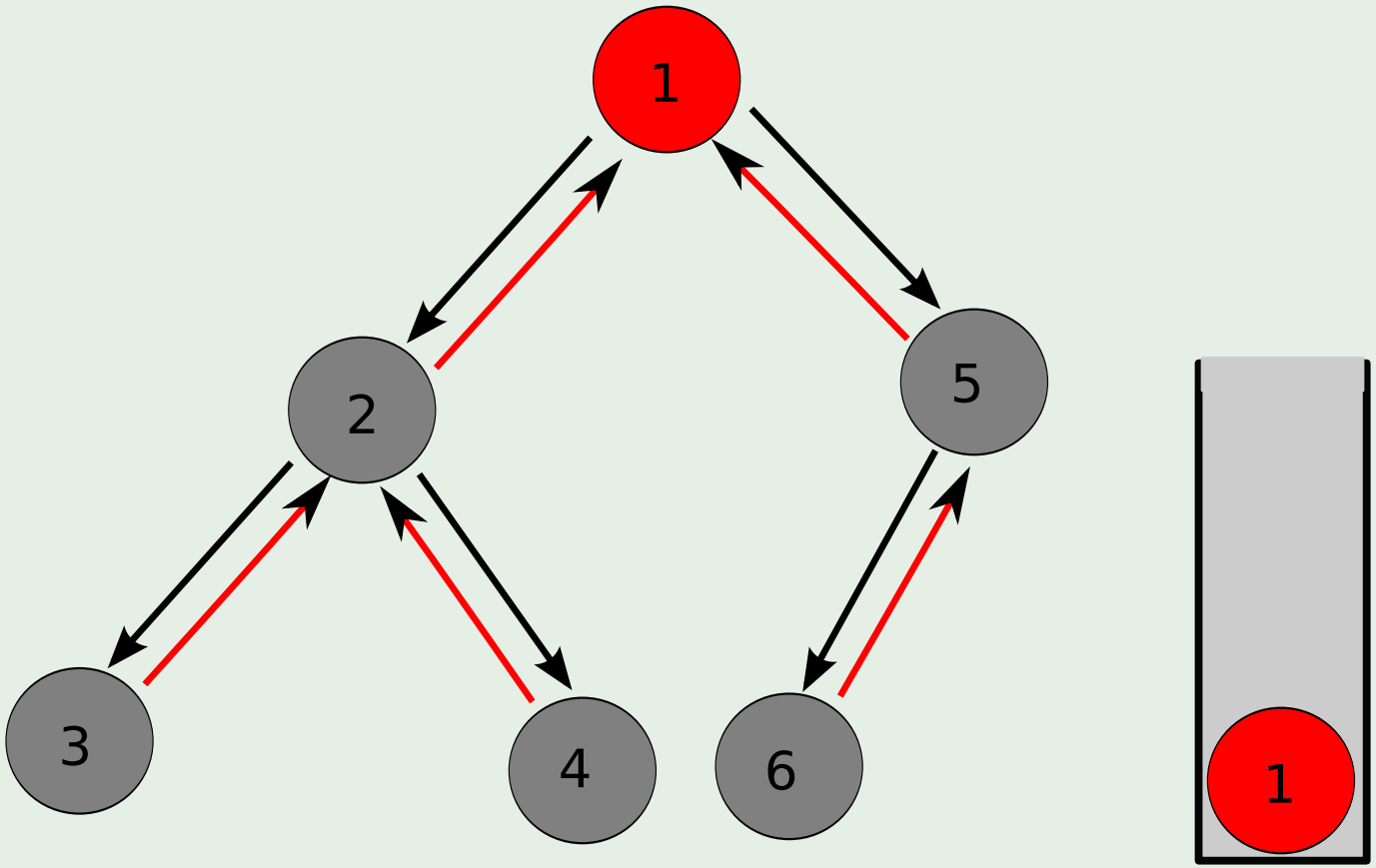
Andres Mendez-Vazquez

August 24, 2016

Outline

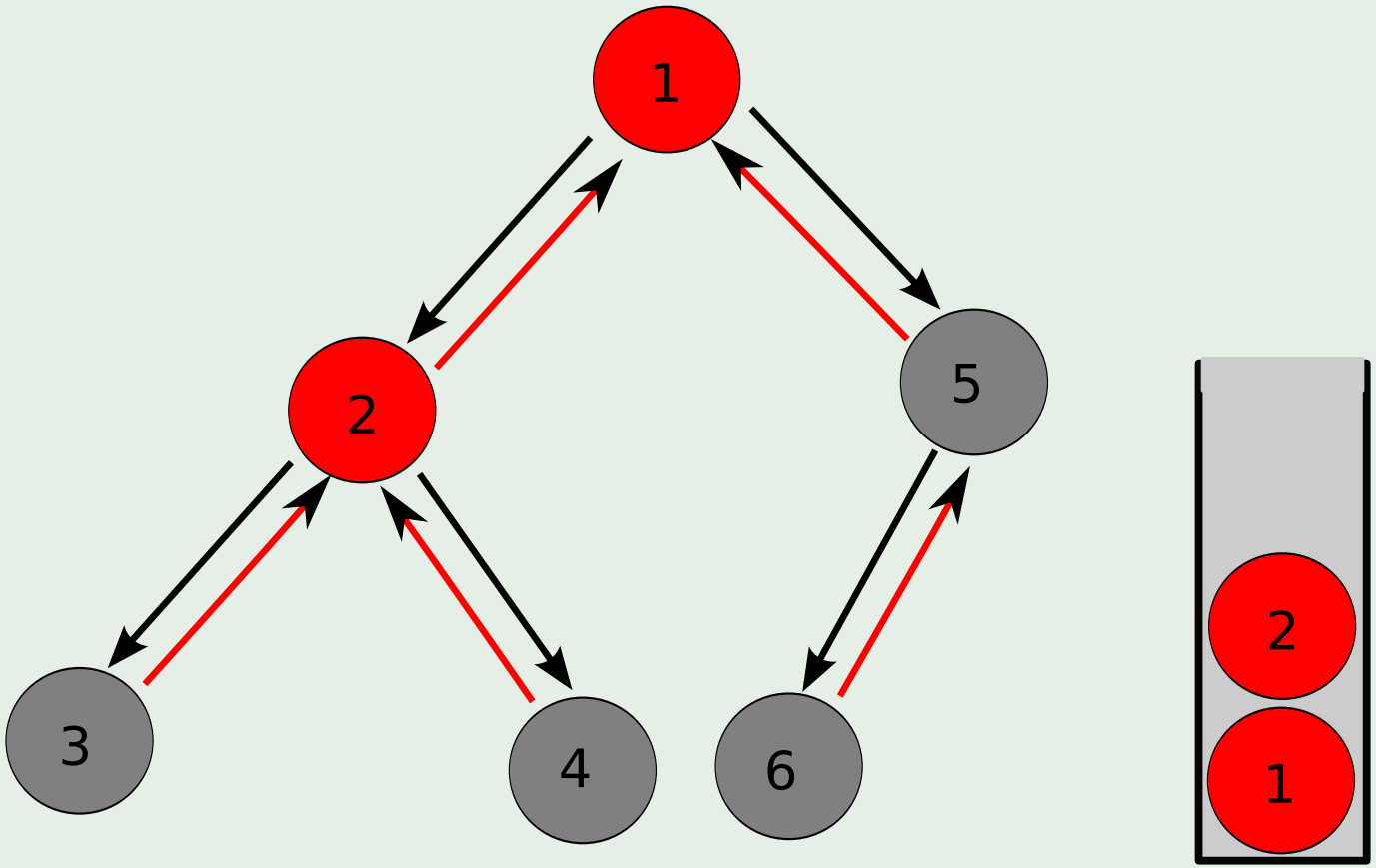
Imagine the following

Stack Order



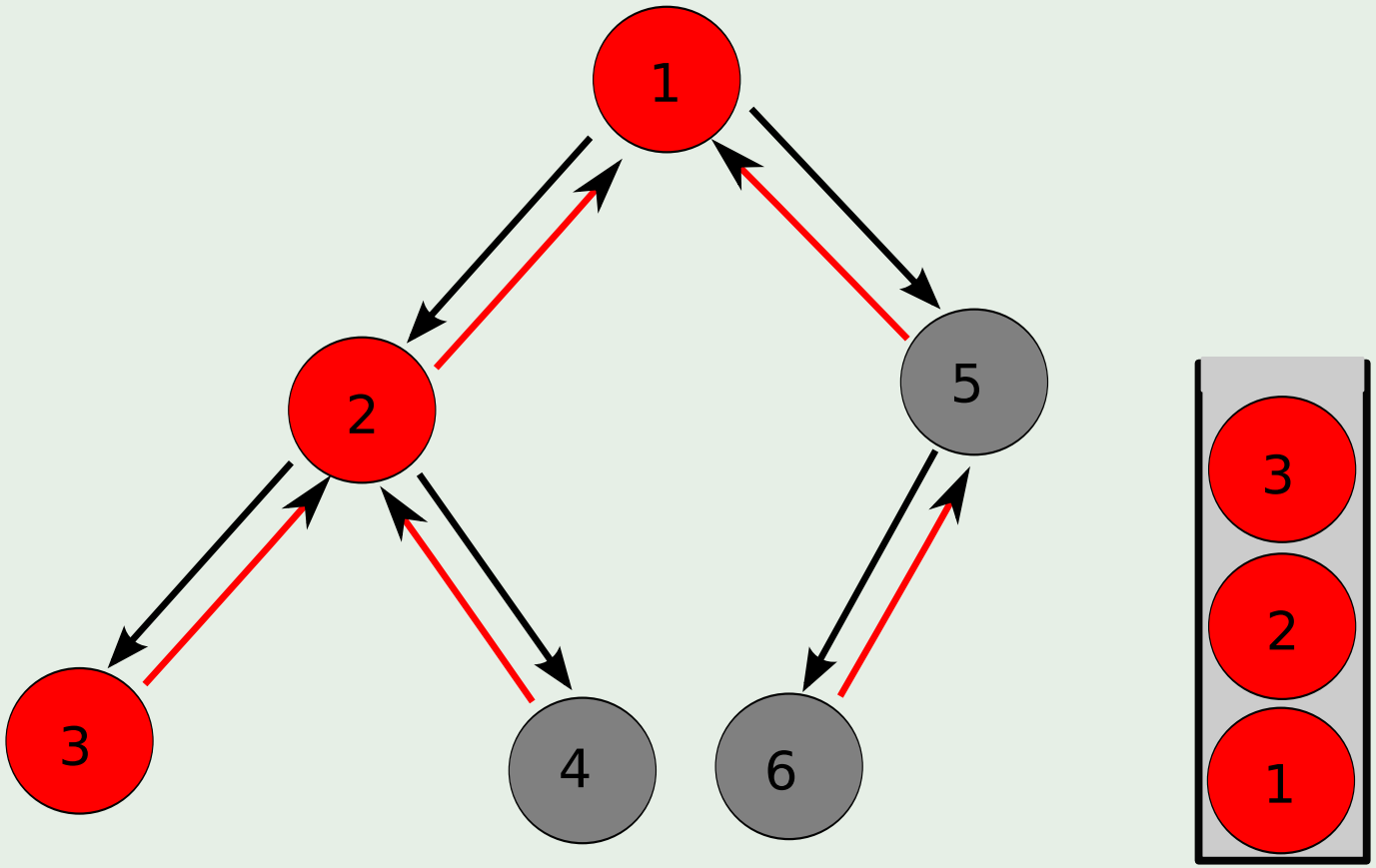
Imagine the following

Stack Order



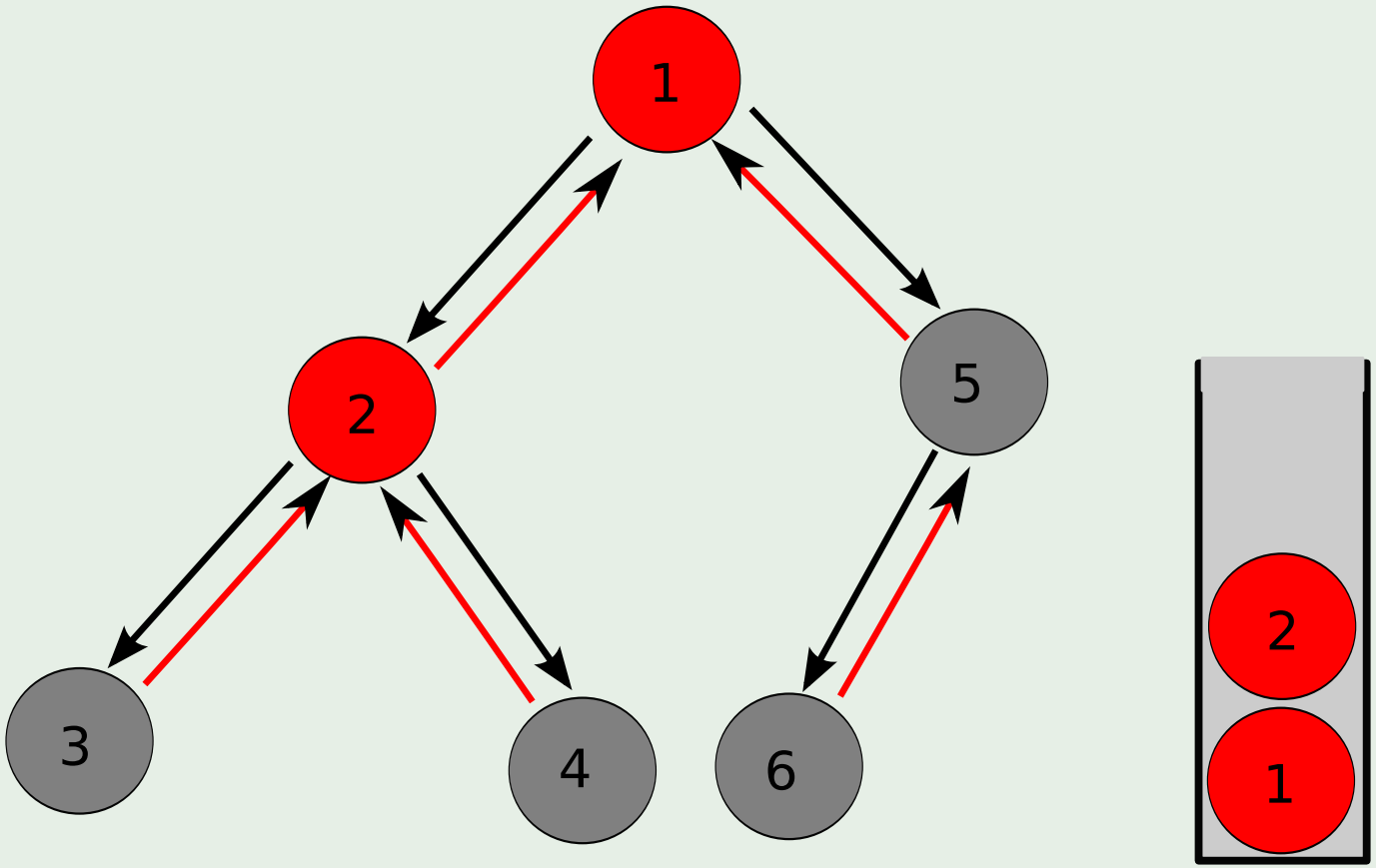
Imagine the following

Stack Order



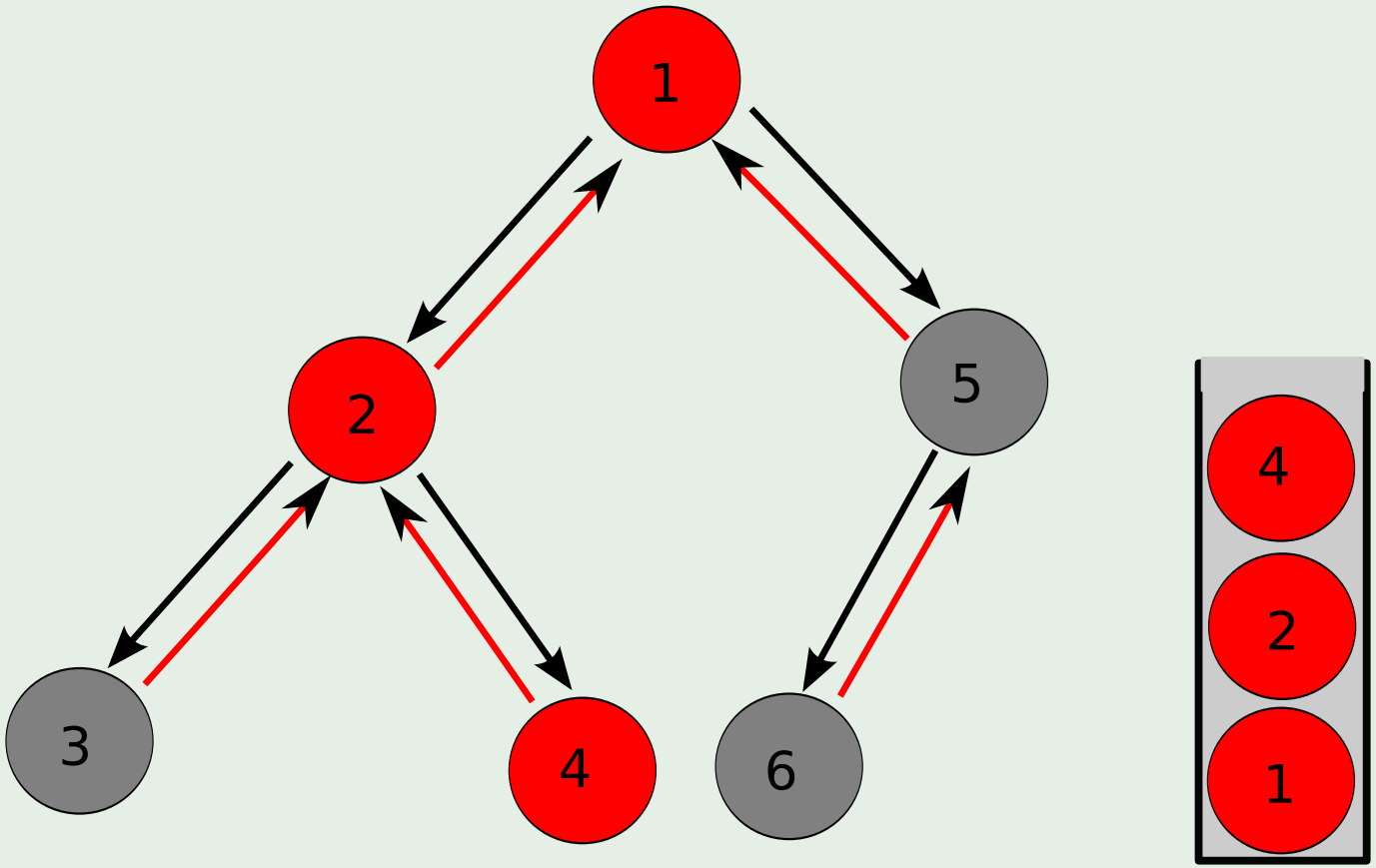
Imagine the following

Stack Order



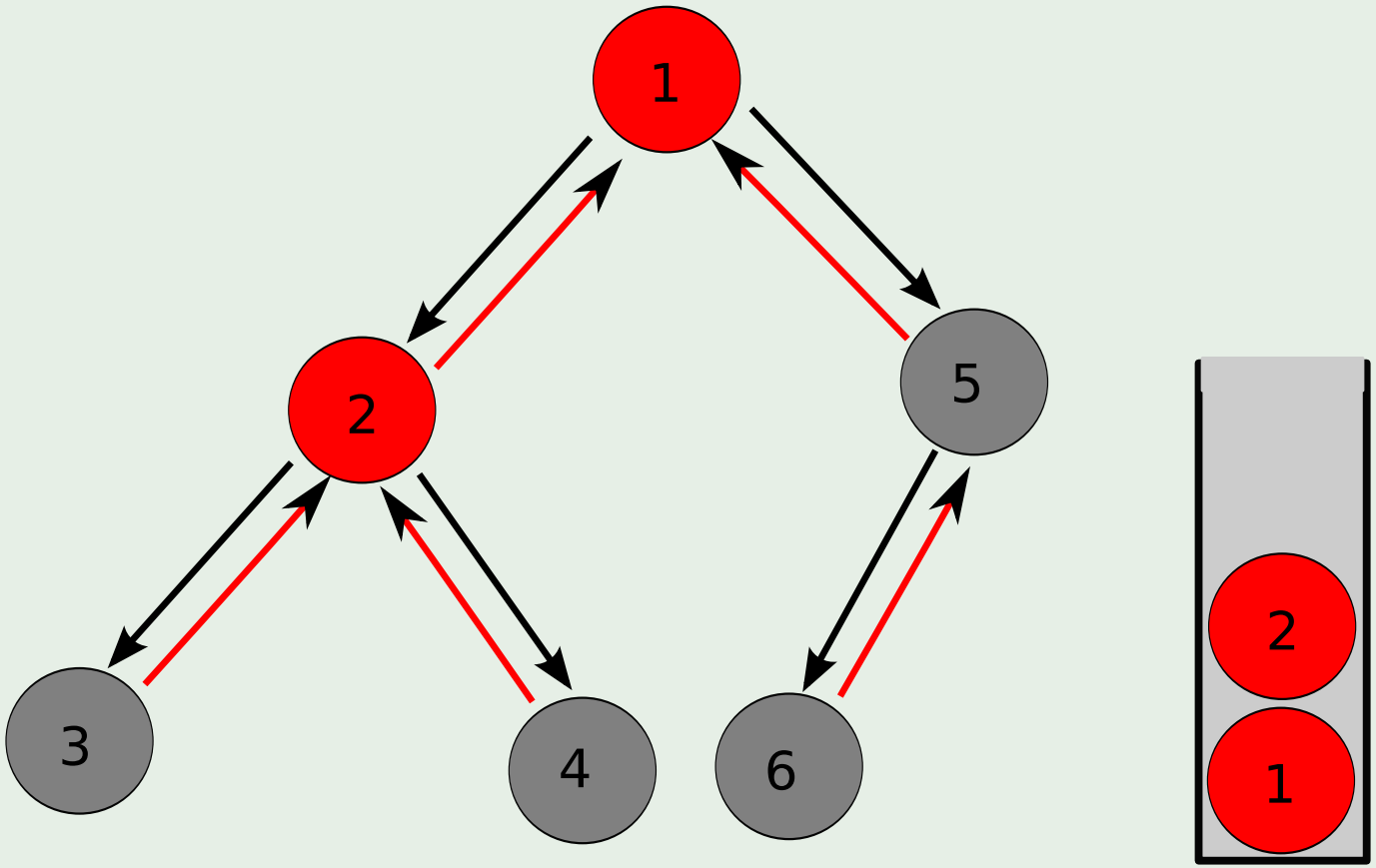
Imagine the following

Stack Order



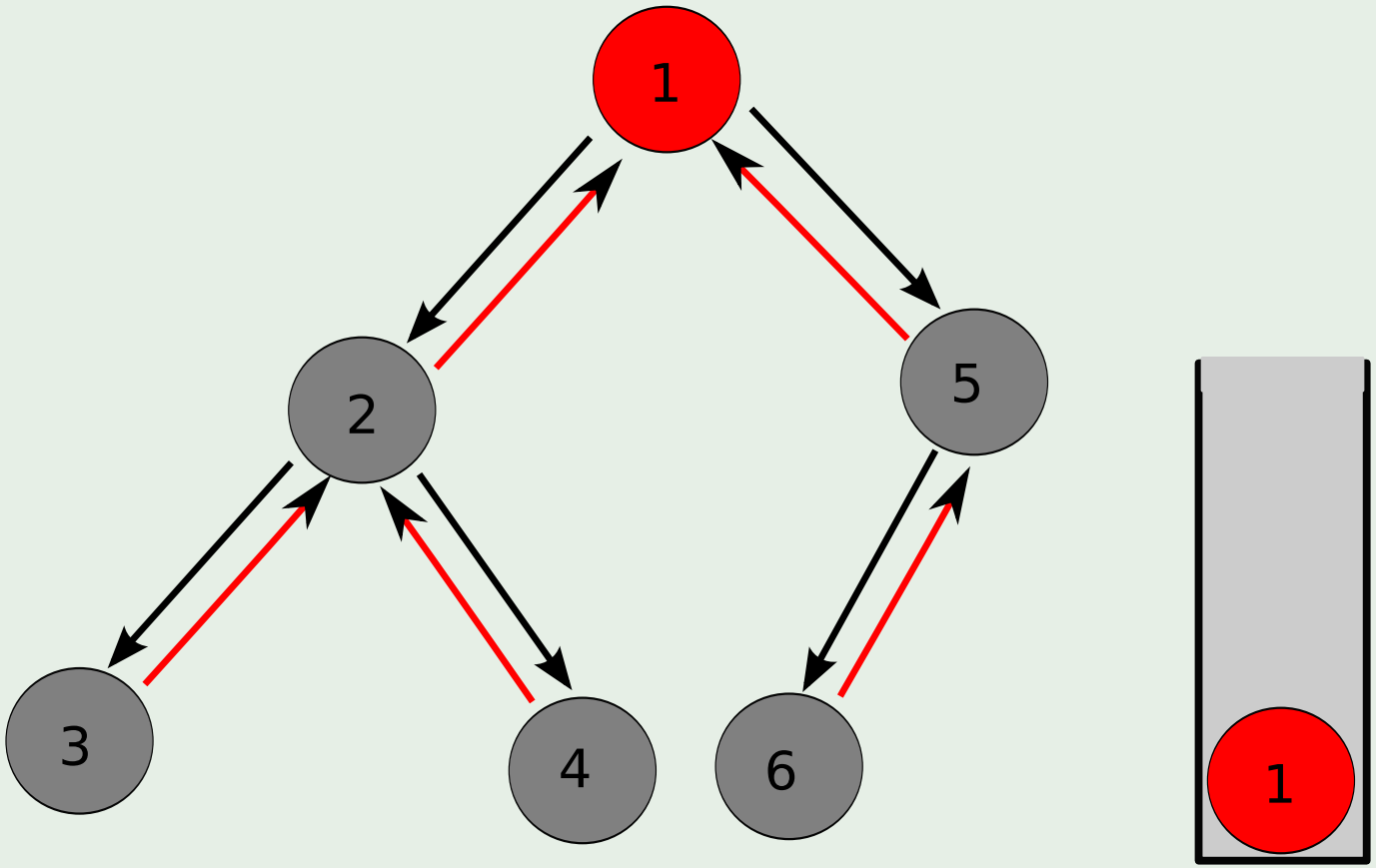
Imagine the following

Stack Order



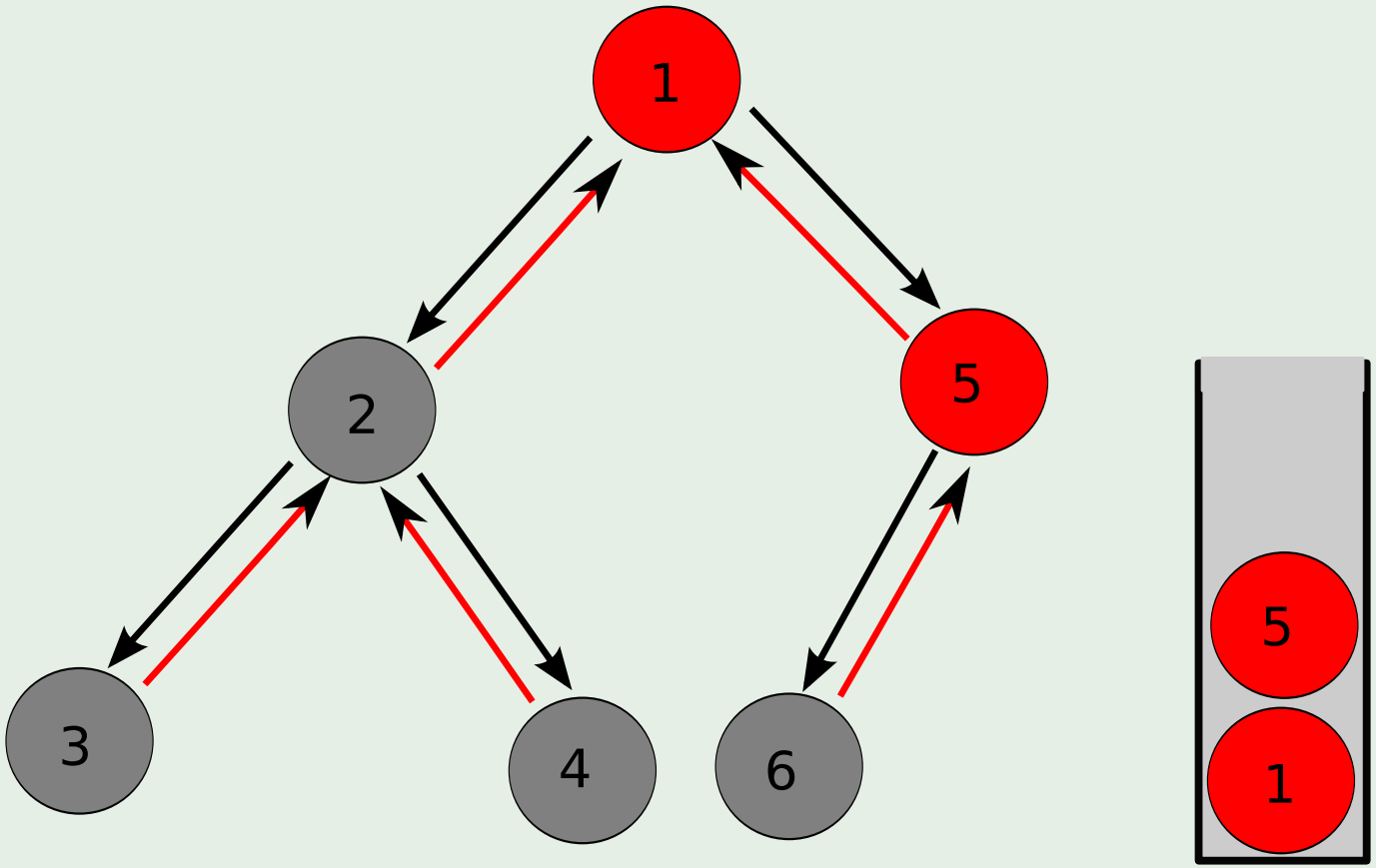
Imagine the following

Stack Order



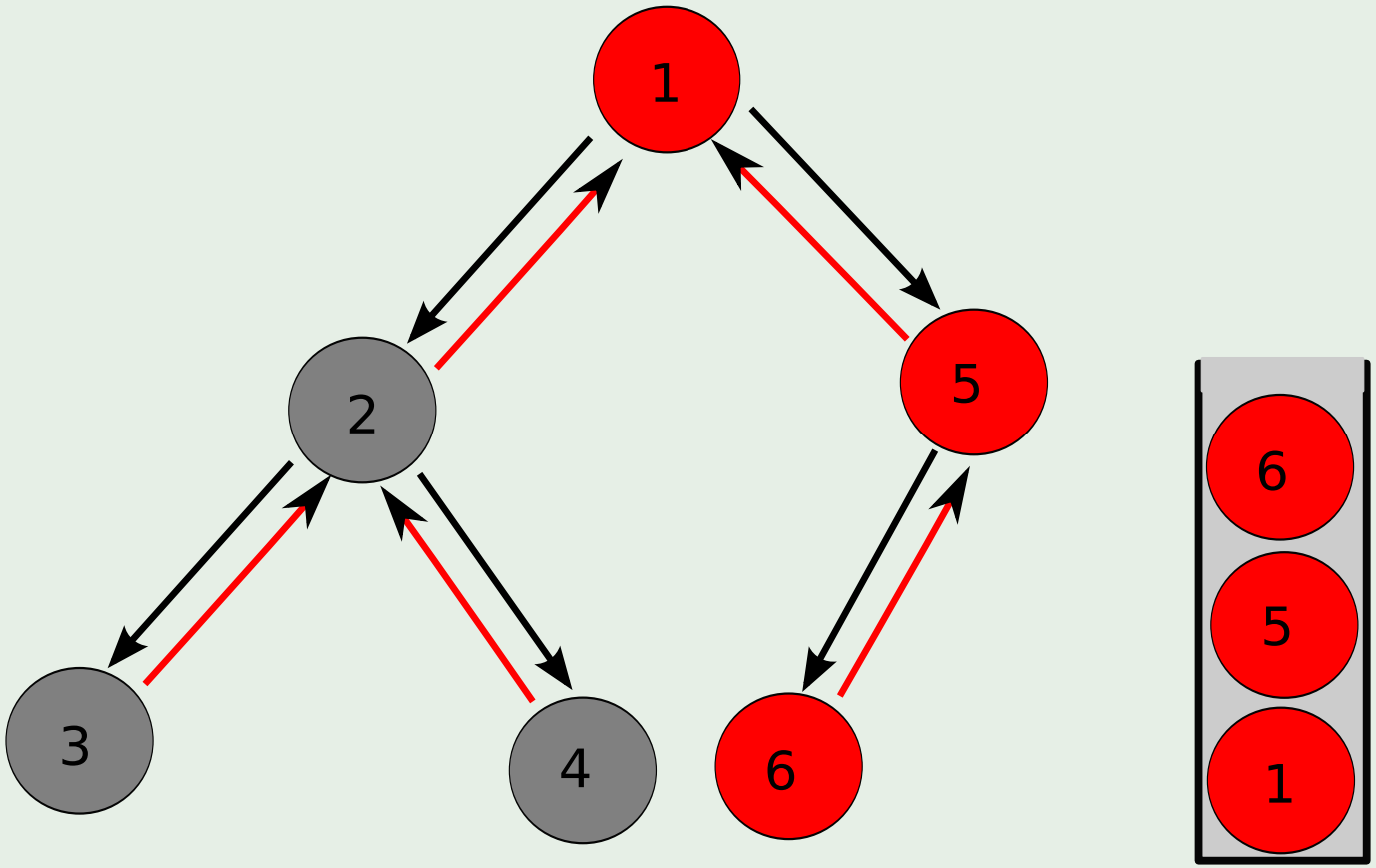
Imagine the following

Stack Order



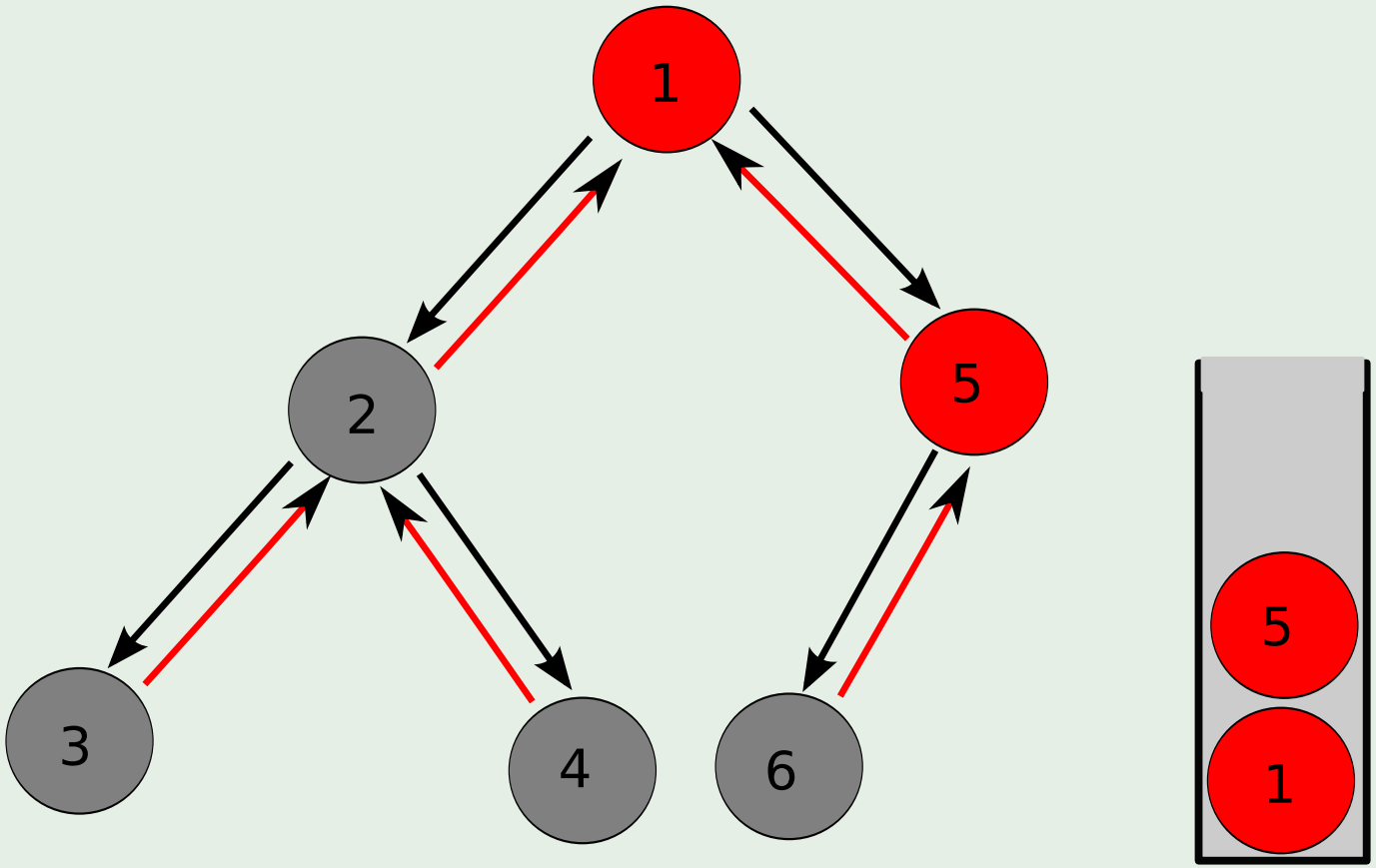
Imagine the following

Stack Order



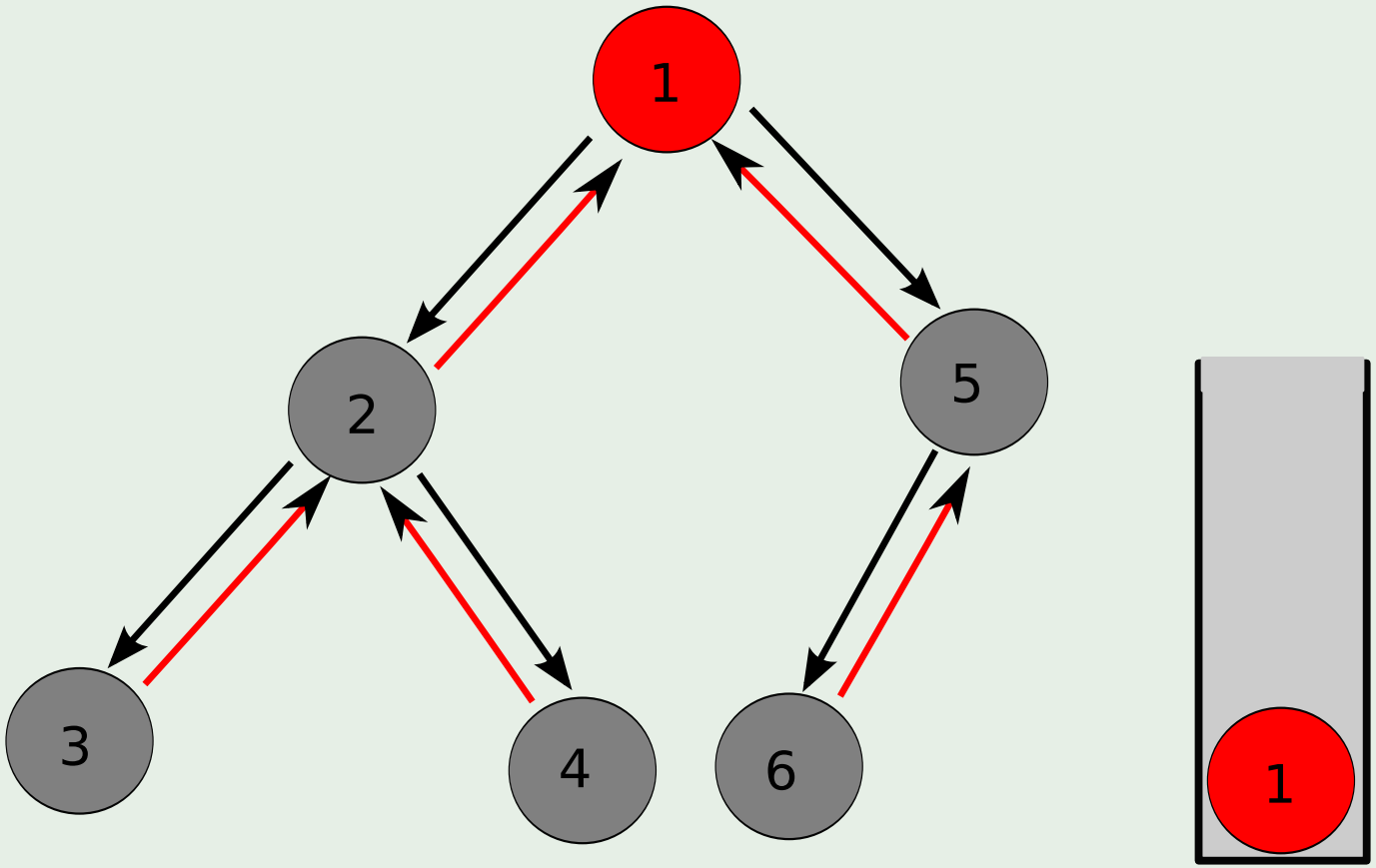
Imagine the following

Stack Order



Imagine the following

Stack Order



Recursion ≈ Depth First Search

Actually

This is the classic order when recursion is done!!!

What if...

We need a different order?

Recursion ≈ Depth First Search

Actually

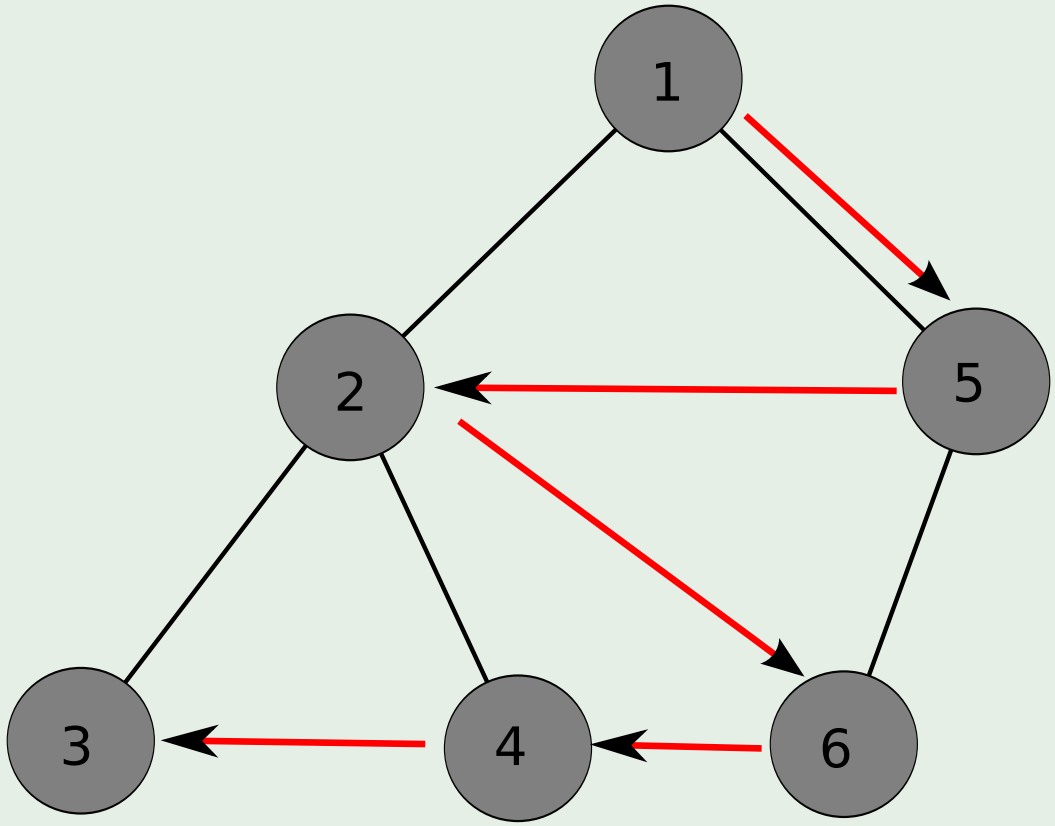
This is the classic order when recursion is done!!!

What if...

We need a different order?

Level Order

How?



Queues

Definition of Queues

- A queue is a abstract data structure that models/enforces the **first-come first-serve order**, or equivalently the First-In First-Out (FIFO) order.

Thus

Using ADT Which is the first thing that comes to your mind to implement a queue?

IMPORTANT

- IN A QUEUE, THE FIRST ITEM INSERTED WILL BE THE FIRST ITEM DELETED: FIFO (FIRST-IN, FIRST-OUT)

Queues

Definition of Queues

- A queue is a abstract data structure that models/enforces the **first-come first-serve order**, or equivalently the First-In First-Out (FIFO) order.

Thus

Using ADT Which is the first thing that comes to your mind to implement a queue?

IMPORTANT

- IN A QUEUE, THE FIRST ITEM INSERTED WILL BE THE FIRST ITEM DELETED: FIFO (FIRST-IN, FIRST-OUT)

Queues

Definition of Queues

- A queue is a abstract data structure that models/enforces the **first-come first-serve order**, or equivalently the First-In First-Out (FIFO) order.

Thus

Using ADT Which is the first thing that comes to your mind to implement a queue?

IMPORTANT

- IN A QUEUE, THE FIRST ITEM INSERTED WILL BE THE FIRST ITEM DELETED: FIFO (FIRST-IN, FIRST-OUT)

Definition

We have then

- A linear list.

Entry Points

- One end is called front.
- Other end is called rear.

Insertion and Deletions

- Additions are done at the rear only.
- Removals are made from the front only.

Definition

We have then

- A linear list.

Entry Points

- One end is called front.
- Other end is called rear.

Insertion and Deletions

- Additions are done at the rear only.
- Removals are made from the front only.

Definition

We have then

- A linear list.

Entry Points

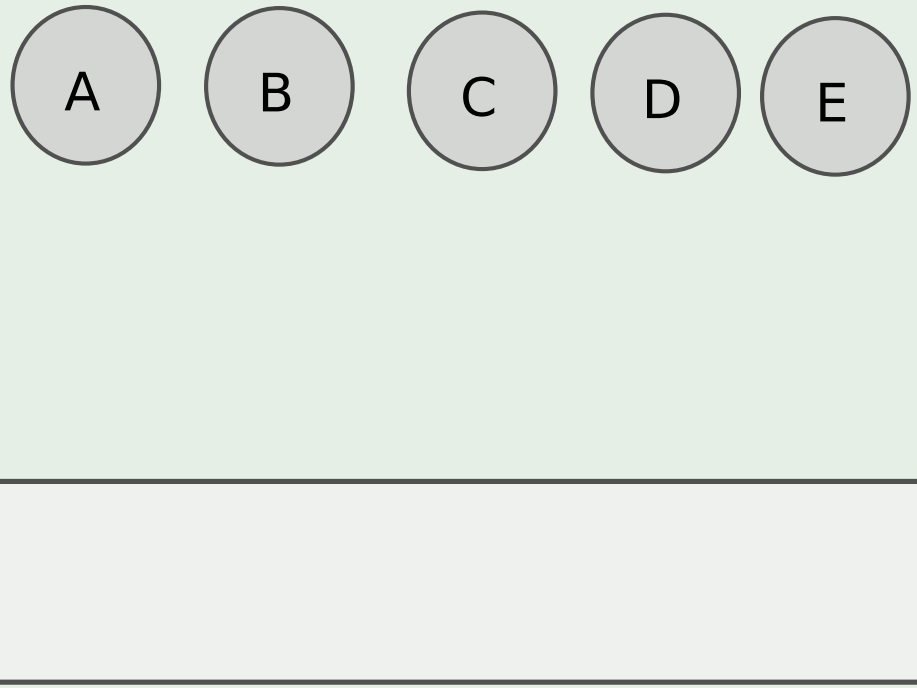
- One end is called front.
- Other end is called rear.

Insertion and Deletions

- Additions are done at the rear only.
- Removals are made from the front only.

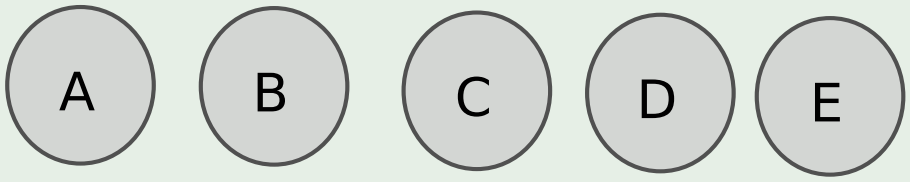
Example

Enqueue - Put



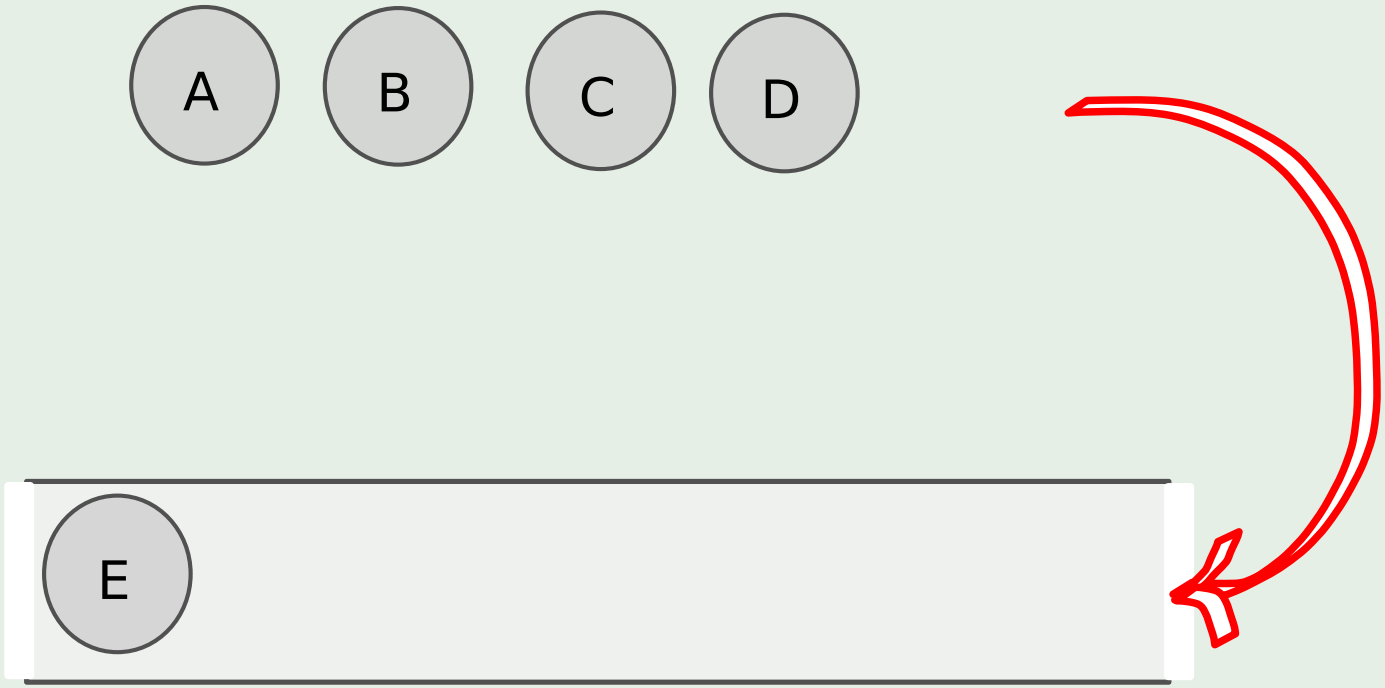
Example

Enqueue - Put



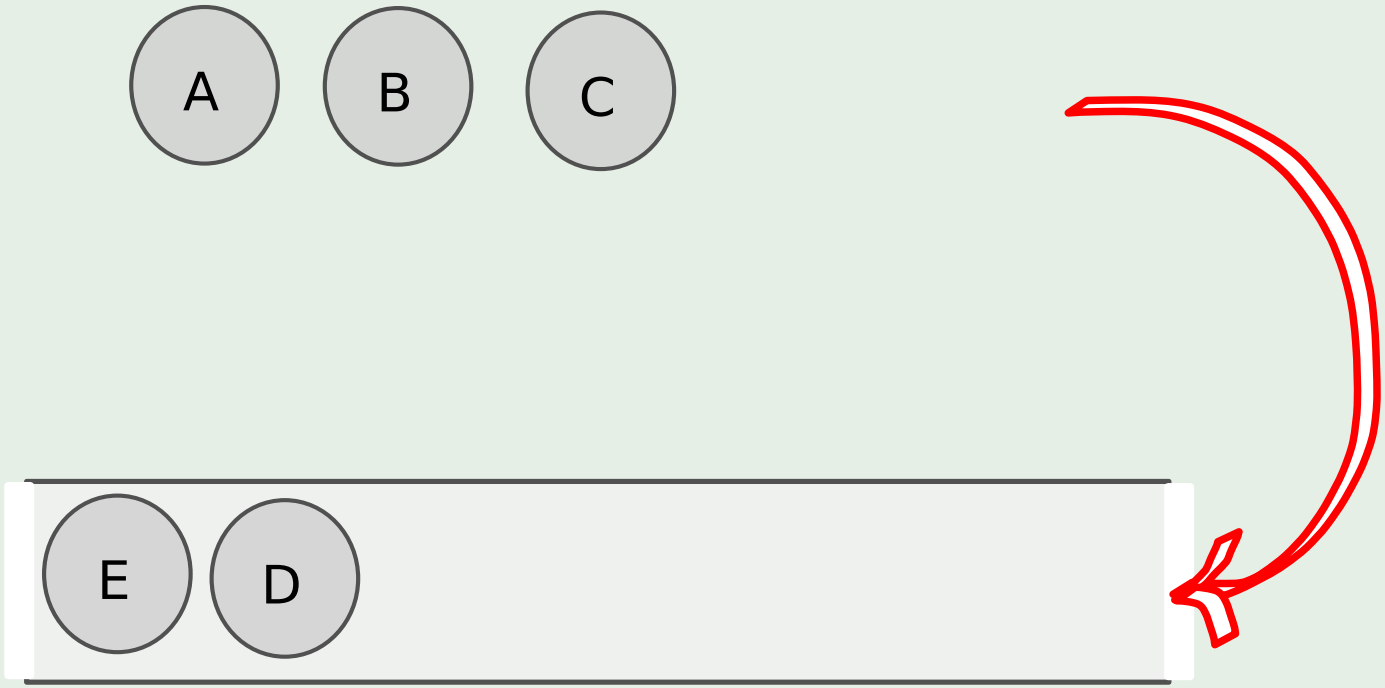
Example

Enqueue - Put



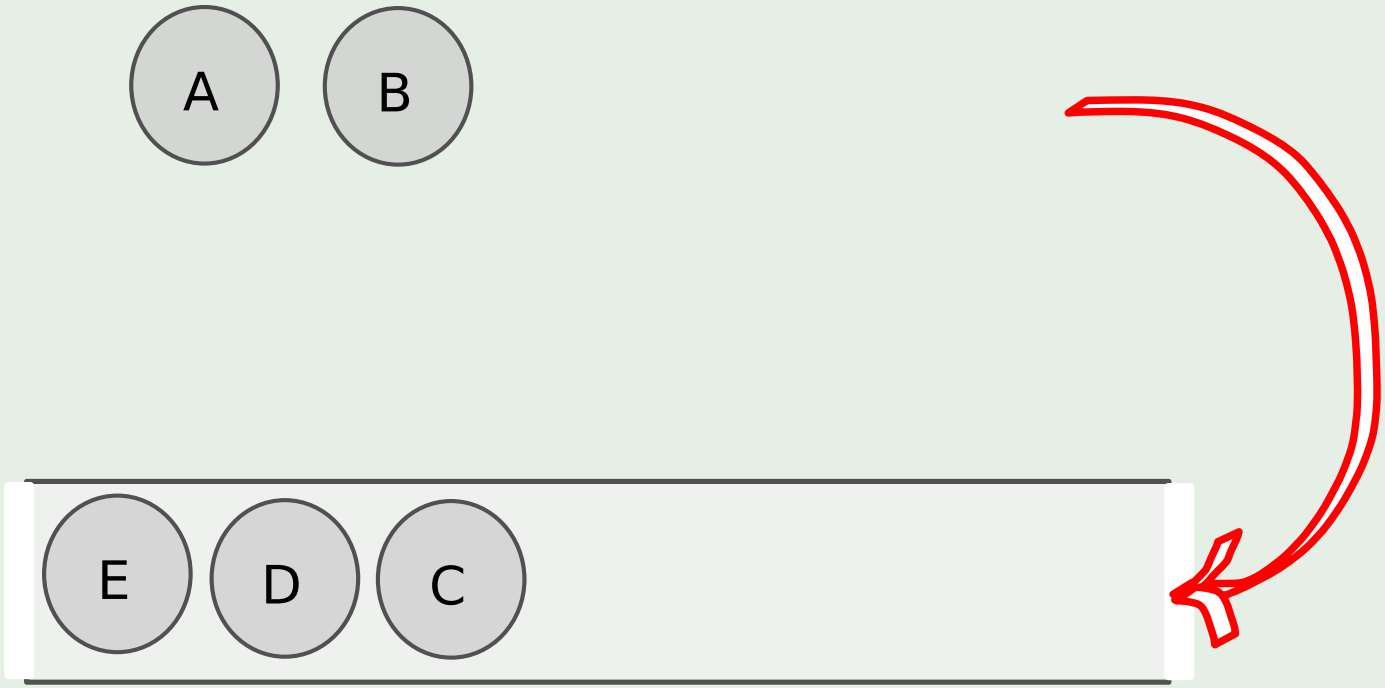
Example

Enqueue - Put



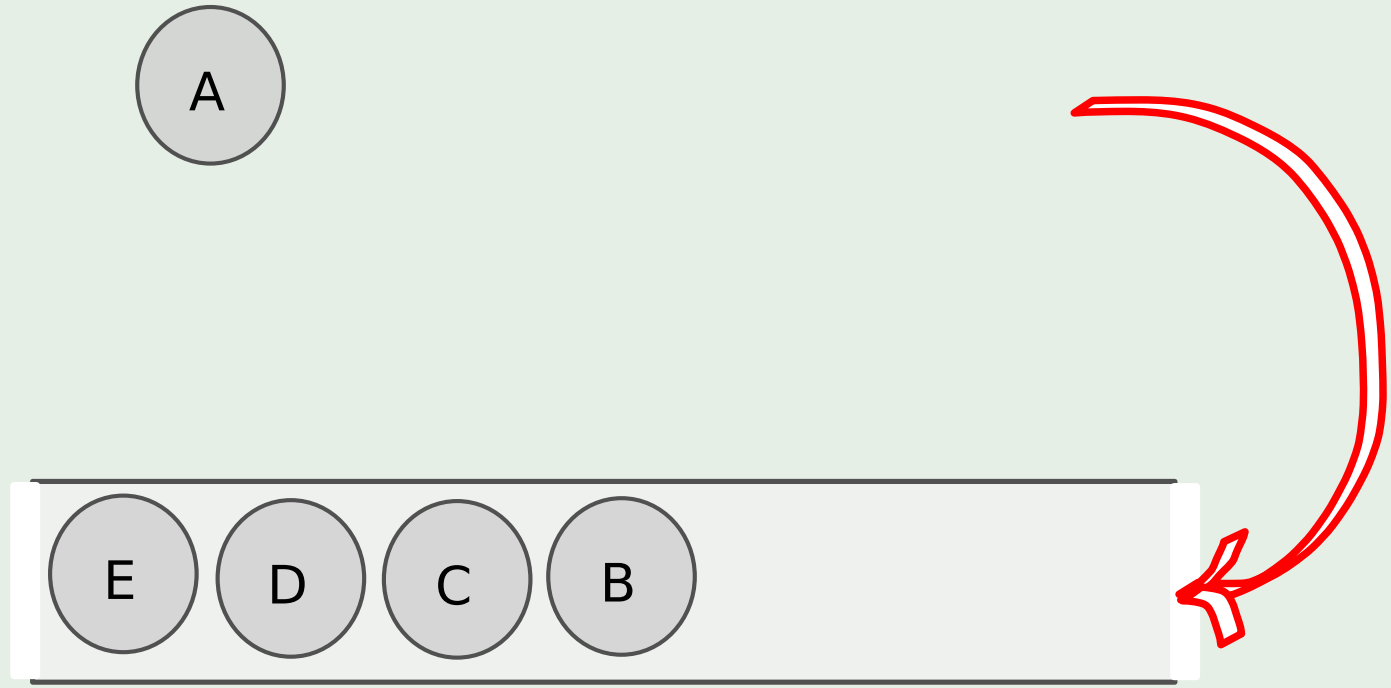
Example

Enqueue - Put



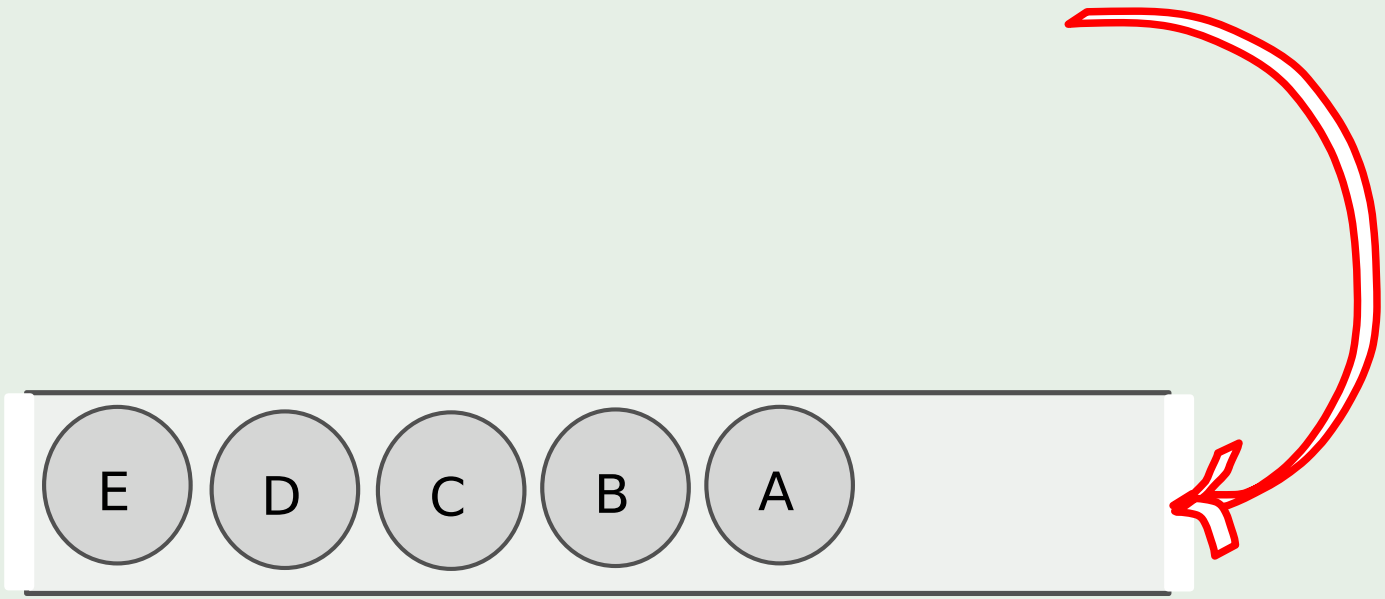
Example

Enqueue - Put



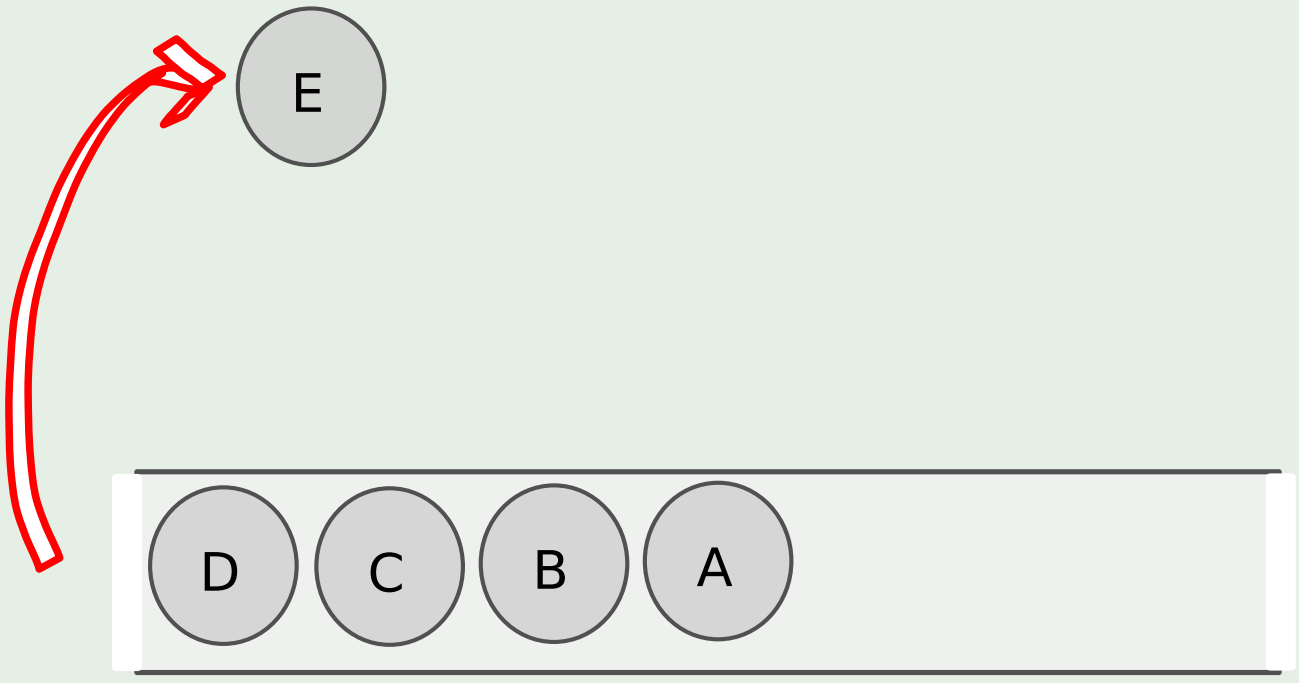
Example

Enqueue - Put



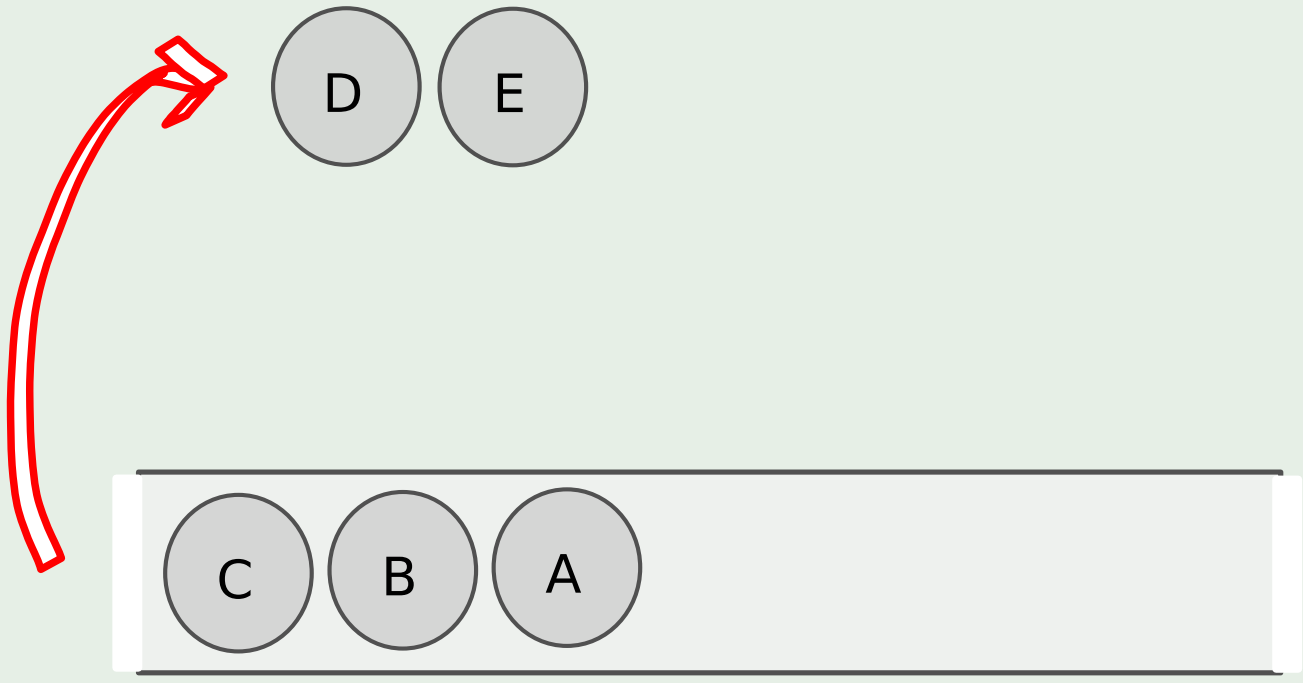
Example

Dequeue - Remove



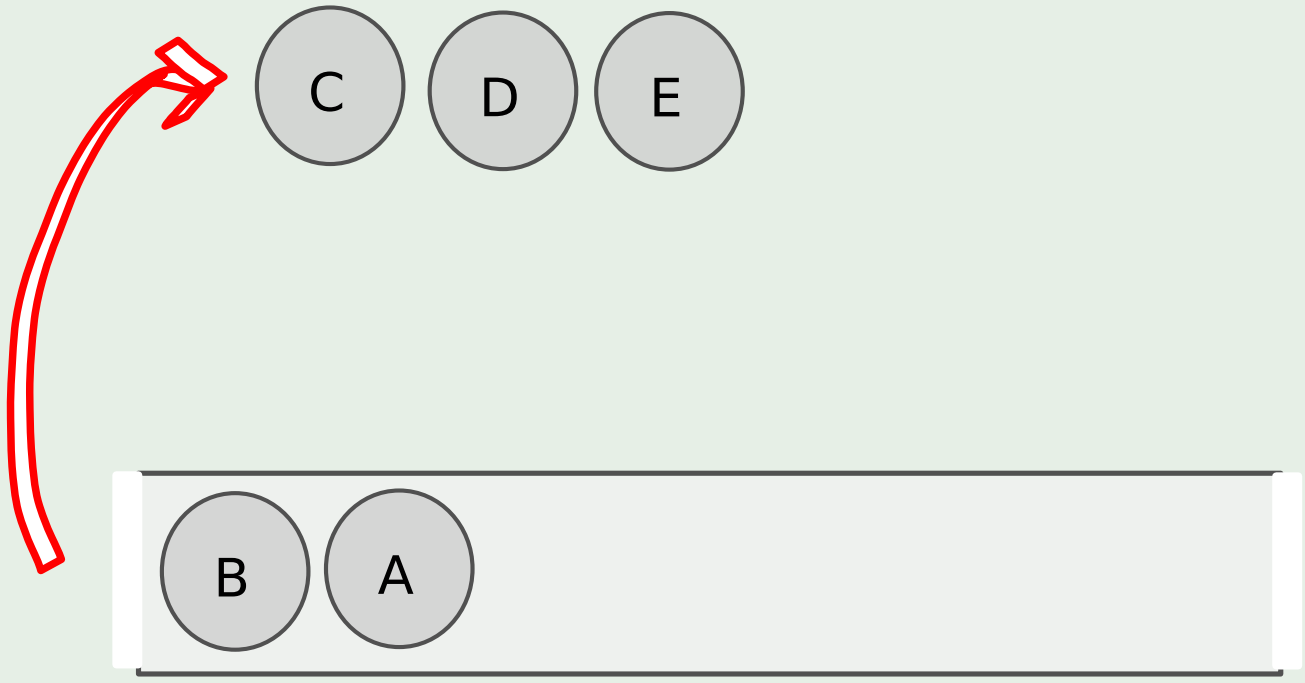
Example

Dequeue - Remove



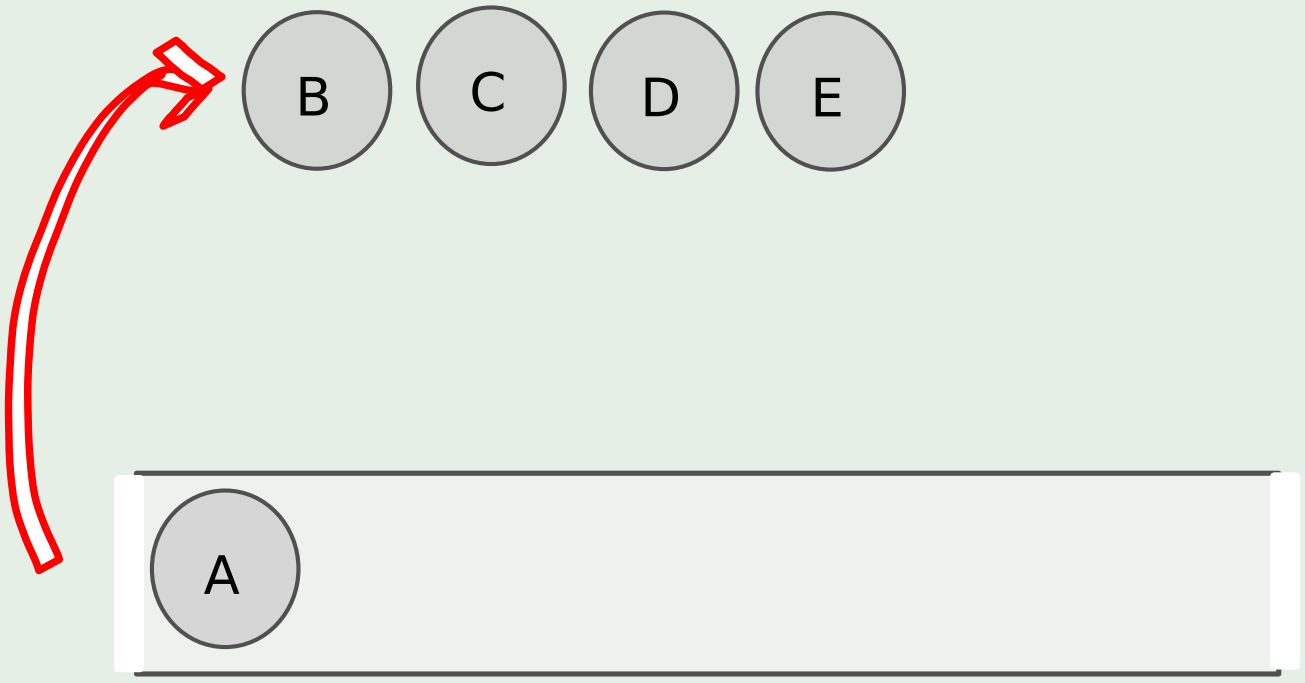
Example

Dequeue - Remove



Example

Dequeue - Remove



Outline

Queue Interface

Code

```
public interface Queue<Item>{
      public boolean empty();
      public Item front();
      public Item rear();
      public Item Dequeue();
      public void Enqueue(Item theObject);
      public int size();
}
```

Explanation

public boolean empty()

- Check whether the queue is empty.
- Return TRUE if it is empty and FALSE otherwise.

Example

public Item front()

- Return the value of the item at the font of the queue without removing it.

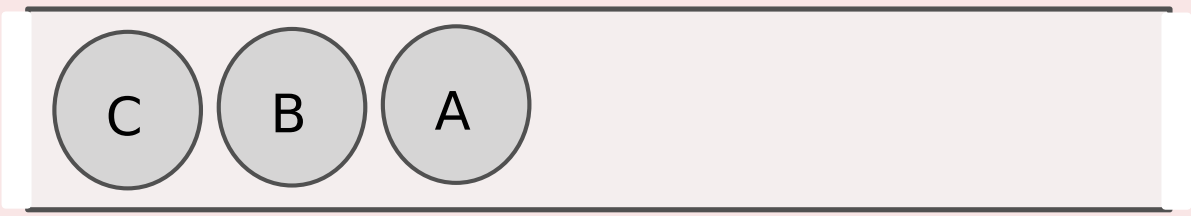
Precondition: The queue is not empty.

Explanation

public boolean empty()

- Check whether the queue is empty.
- Return TRUE if it is empty and FALSE otherwise.

Example



public Item front()

- Return the value of the item at the font of the queue without removing it.

Precondition: The queue is not empty.

Explanation

public boolean empty()

- Check whether the queue is empty.
- Return TRUE if it is empty and FALSE otherwise.

Example

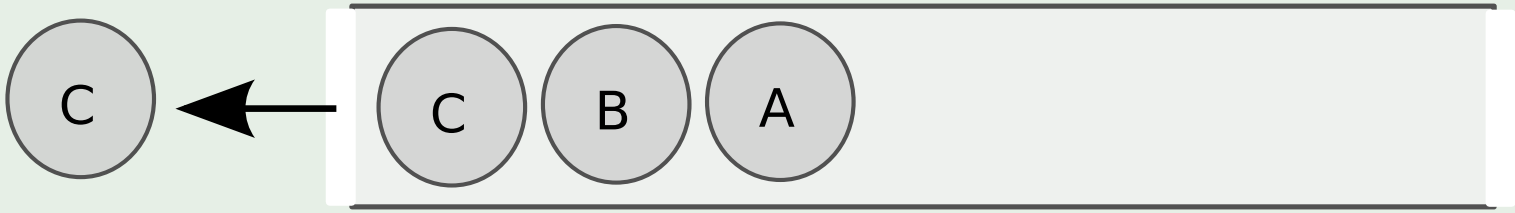
public Item front()

- Return the value of the item at the font of the queue without removing it.

Precondition: The queue is not empty.

Explanation

Example



public Item rear()

- Return the value of the item at the rear of the queue without removing it.

Precondition: The queue is not empty.

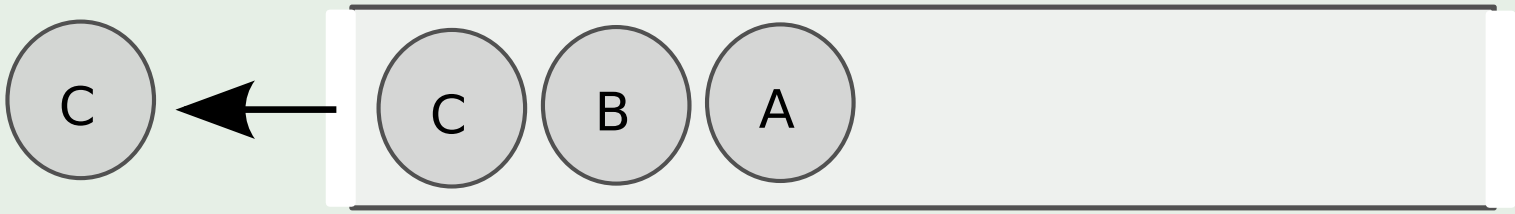
public void Enqueue(Item theObject)

- Insert the argument item at the back of the queue.

Postcondition: The queue has a new item at the back

Explanation

Example



public Item rear()

- Return the value of the item at the rear of the queue without removing it.

Precondition: The queue is not empty.

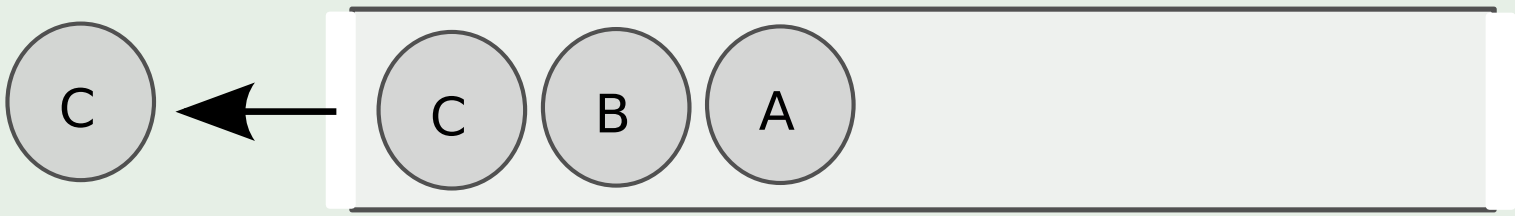
public void Enqueue(Item theObject)

- Insert the argument item at the back of the queue.

Postcondition: The queue has a new item at the back

Explanation

Example



public Item rear()

- Return the value of the item at the rear of the queue without removing it.

Precondition: The queue is not empty.

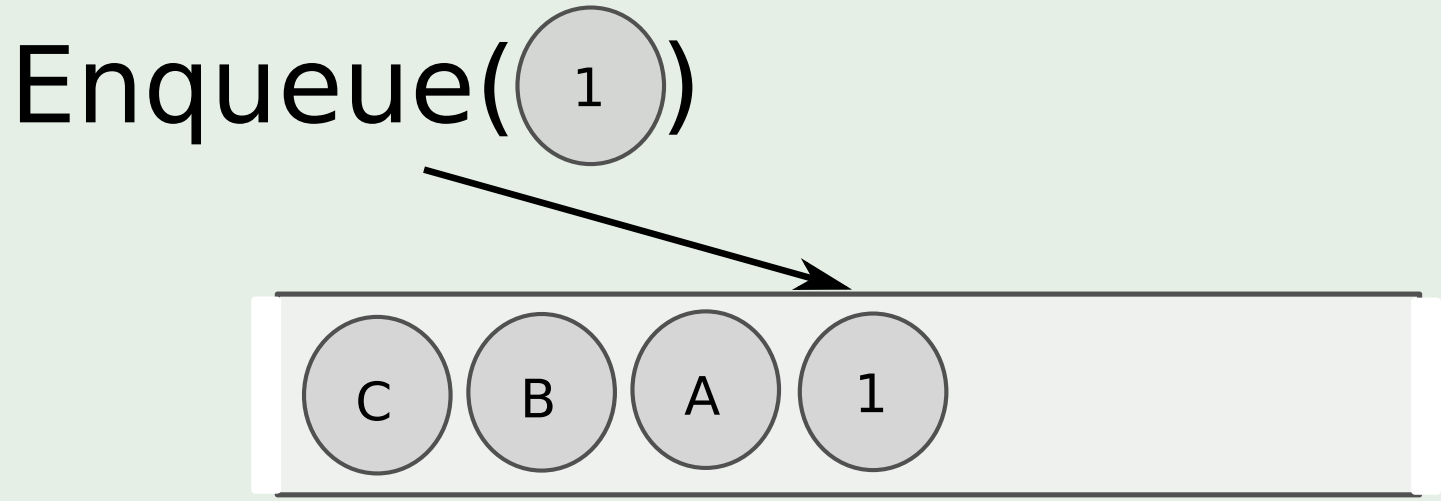
public void Enqueue(Item theObject)

- Insert the argument item at the back of the queue.

Postcondition: The queue has a new item at the back

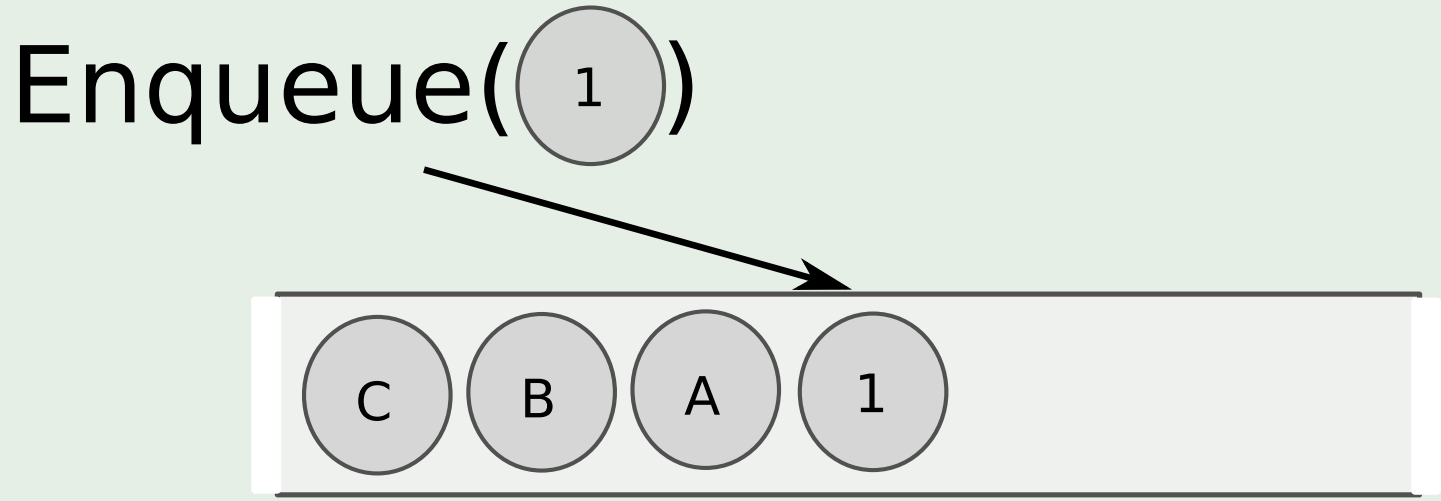
Explanation

Example



Explanation

Example



public Item Dequeue()

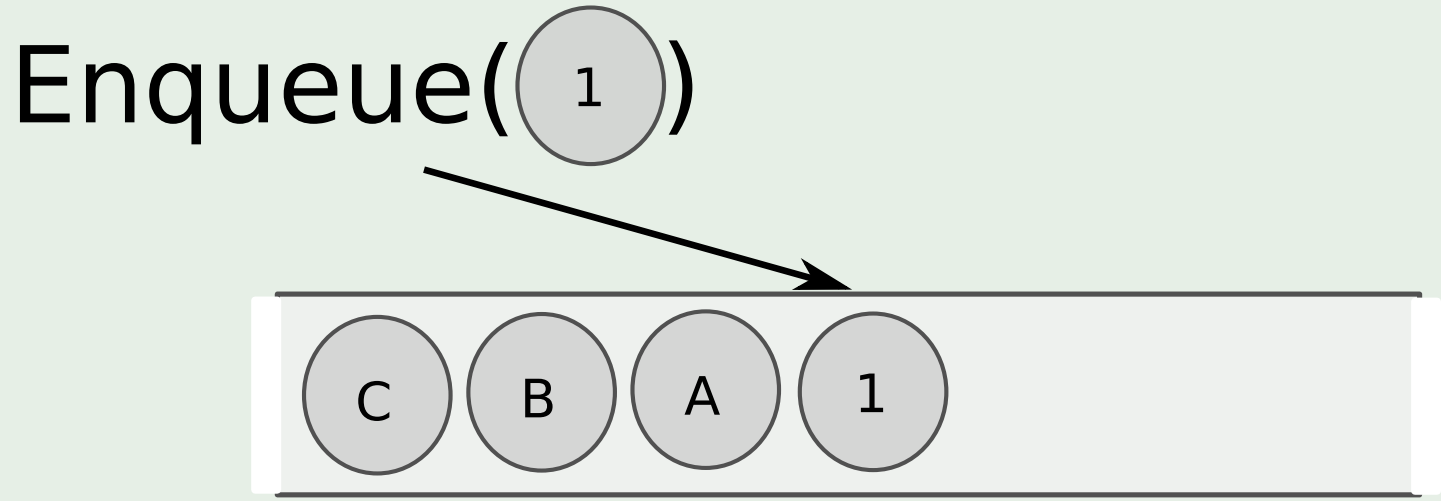
- Remove the item from the front of the queue.

Precondition: The queue is not empty.

Postcondition: The element at the front of the queue is the element that was added immediately after the element just popped or the queue is empty.

Explanation

Example



public Item Dequeue()

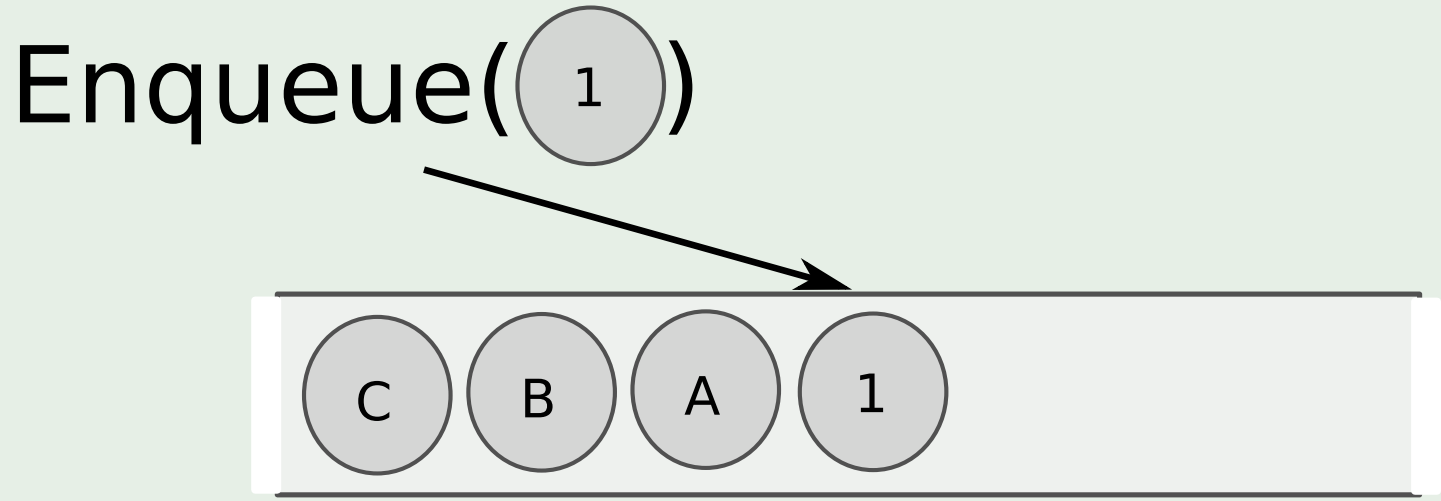
- Remove the item from the front of the queue.

Precondition: The queue is not empty.

Postcondition: The element at the front of the queue is the element that was added immediately after the element just popped or the queue is empty.

Explanation

Example



public Item Dequeue()

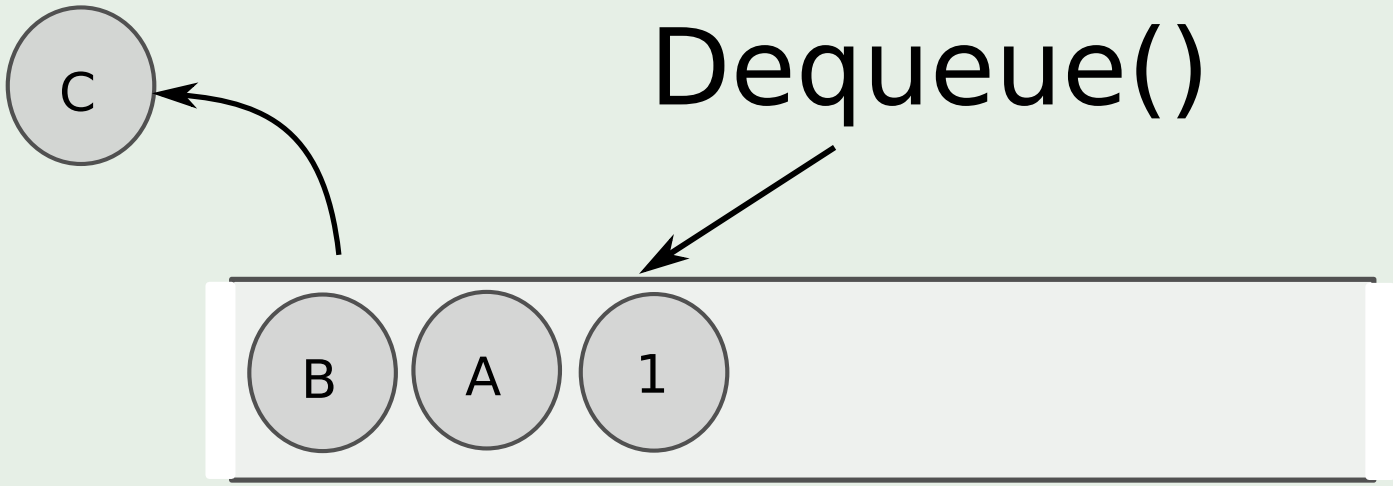
- Remove the item from the front of the queue.

Precondition: The queue is not empty.

Postcondition: The element at the front of the queue is the element that was added immediately after the element just popped or the queue is empty.

Explanation

Example

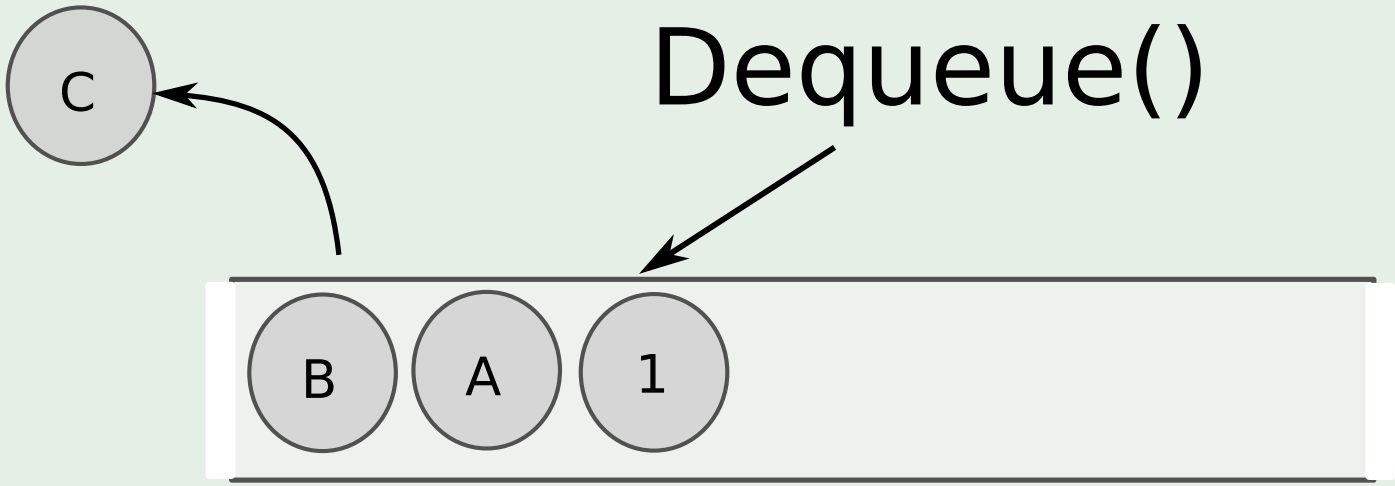


public int size()

It returns the size.

Explanation

Example



public int size()

It returns the size.

Applications of Queues

Direct applications

- Waiting lists
  - Queue Theory for Networking
- Bureaucracy Access to shared resources (e.g., printer)
- Multiprogramming
  - Schedulers

Applications of Queues

Direct applications

- Waiting lists
  - Queue Theory for Networking
- Bureaucracy Access to shared resources (e.g., printer)
- Multiprogramming
  - Schedulers

Indirect applications

- Auxiliary data structure for algorithms
- Component of other data structures

Applications of Queues

Direct applications

- Waiting lists
  - Queue Theory for Networking
- Bureaucracy Access to shared resources (e.g., printer)
- Multiprogramming
  - Schedulers

Indirect applications

- Auxiliary data structure for algorithms
- Component of other data structures

Applications of Queues

Direct applications

- Waiting lists
  - Queue Theory for Networking
- Bureaucracy Access to shared resources (e.g., printer)
- Multiprogramming
  - Schedulers

Indirect applications

- Auxiliary data structure for algorithms
- Component of other data structures

Applications of Queues

Direct applications

- Waiting lists
  - Queue Theory for Networking
- Bureaucracy Access to shared resources (e.g., printer)
- Multiprogramming
  - Schedulers

Indirect applications

- Auxiliary data structure for algorithms
- Component of other data structures

Applications of Queues

Direct applications

- Waiting lists
  - Queue Theory for Networking
- Bureaucracy Access to shared resources (e.g., printer)
- Multiprogramming
  - Schedulers

Indirect applications

- Auxiliary data structure for algorithms
- Component of other data structures

Applications of Queues

Direct applications

- Waiting lists
  - Queue Theory for Networking
- Bureaucracy Access to shared resources (e.g., printer)
- Multiprogramming
  - Schedulers

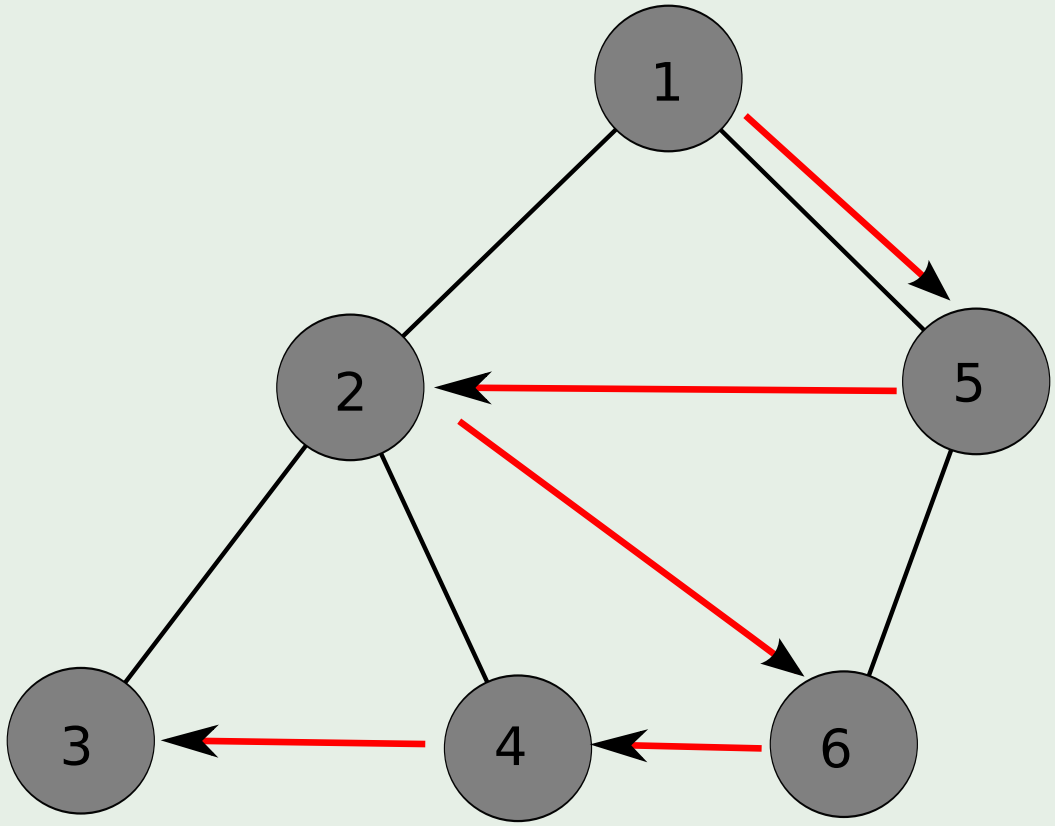
Indirect applications

- Auxiliary data structure for algorithms
- Component of other data structures

Outline

Change the Order of Recursion

Remember



Thus, Using a Trick

and Queue

- We can change the direction of the recursion!!!
- Look at the board

Outline

Radix Sort Using Bins

Example

Order ten 2 digit numbers in 10 bins (0-9) from least significative number to most significative number.

Digits

91,06,85,15,92,35,30,22,39

Let us do it

In the board!!!

Radix Sort Using Bins

Example

Order ten 2 digit numbers in 10 bins (0-9) from least significative number to most significative number.

Digits

91,06,85,15,92,35,30,22,39

Let us do it

In the board!!!

Radix Sort Using Bins

Example

Order ten 2 digit numbers in 10 bins (0-9) from least significative number to most significative number.

Digits

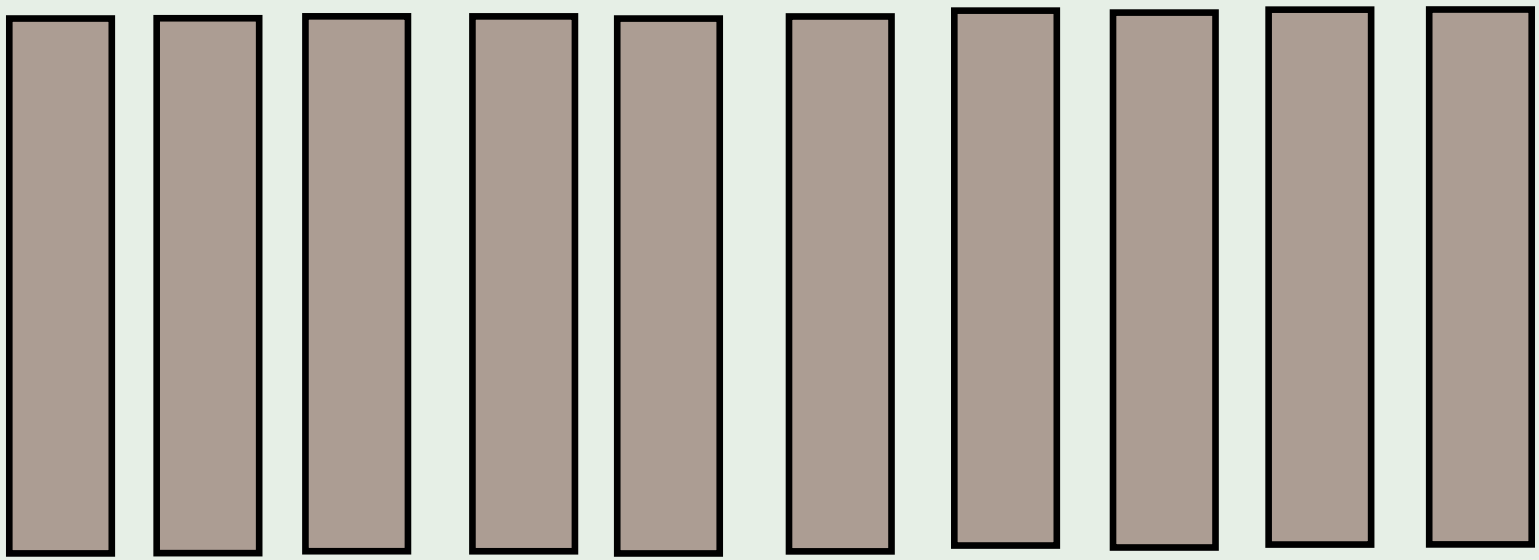
91,06,85,15,92,35,30,22,39

Let us do it

In the board!!!

Example

Pass 0: Distribute the digits into bins according to the 1's digit (10^0).



Finally

Next

- Dequeue the values from the queue 0 to queue 9.
- Put values in a list in that order.

Finally

Next

- Dequeue the values from the queue 0 to queue 9.
- Put values in a list in that order.

Next

Pass 1: Take the new sequence and distribute the cards into bins determined by the 10's digit (10^1)

Next

- Put values in a list in that order.

Finally

Next

- Dequeue the values from the queue 0 to queue 9.
- Put values in a list in that order.

Next

Pass 1: Take the new sequence and distribute the cards into bins determined by the 10's digit (10^1)

Finally

- Dequeue the values from the queue 0 to queue 9.
- Put values in a list in that order.

Finally

Next

- Dequeue the values from the queue 0 to queue 9.
- Put values in a list in that order.

Next

Pass 1: Take the new sequence and distribute the cards into bins determined by the 10's digit (10^1)

Finally

- Dequeue the values from the queue 0 to queue 9.
- Put values in a list in that order.

Finally

Next

- Dequeue the values from the queue 0 to queue 9.
- Put values in a list in that order.

Next

Pass 1: Take the new sequence and distribute the cards into bins determined by the 10's digit (10^1)

Finally

- Dequeue the values from the queue 0 to queue 9.
- Put values in a list in that order.

Code

We need something else

```
public static RadixSort(LinearList Element, int k){
    int Radix = 10
    int power = 1
    int digit;
    Queue<Integer>[] digitQueue = (Queue<Integer>[])
                                    new Queue[10];
    for (int i=0; i<k ; i++)
                {
            for (int j=0; j< Element.size(); j++){
                digit = Element.remove(0);
                digitQueue[(digit/power)%10].
                enqueue(digit);
            }
            for (int j=0;j< Radix; j++){
                    // WHAT?
                                    }
        power *= 10;
                }
}
```

Radix Sort: Complexity

Lemma 1

Given n d-digit numbers in which each digit can take on up to k possible values, RADIX-SORT correctly sorts these numbers in $O(d(n+k))$ time.

Outline

Simulating Waiting Lines

Simulation is used to study the performance

- Of a physical ("real") system.
- By using a physical, mathematical, or computer model of the system.

Simulation allows designers to estimate performance

- Before building a system

Simulation can lead to design improvements

- Giving better expected performance of the system

Simulating Waiting Lines

Simulation is used to study the performance

- Of a physical ("real") system.
- By using a physical, mathematical, or computer model of the system.

Simulation allows designers to estimate performance

- Before building a system

Simulation can lead to design improvements

- Giving better expected performance of the system

Simulating Waiting Lines

Simulation is used to study the performance

- Of a physical ("real") system.
- By using a physical, mathematical, or computer model of the system.

Simulation allows designers to estimate performance

- Before building a system

Simulation can lead to design improvements

- Giving better expected performance of the system

Queuing Theory

Constraints

We will maintain a simulated clock Counts in integer “ticks”, from 0

Queuing Theory

Constraints

We will maintain a simulated clock Counts in integer “ticks”, from 0

At each tick

- Frequent flyer (FF) passenger arrives in line
- Regular (R) passenger arrives in line
- One agent with priorities
  1. Agent finishes, then serves next FF passenger
  2. Agent finishes, then serves next R passenger
- Agent is idle (both lines empty)

Queuing Theory

Constraints

We will maintain a simulated clock Counts in integer “ticks”, from 0

At each tick

- Frequent flyer (FF) passenger arrives in line
- Regular (R) passenger arrives in line
- One agent with priorities
  1. Agent finishes, then serves next FF passenger
  2. Agent finishes, then serves next R passenger
- Agent is idle (both lines empty)

Using some other constraints

- Max # FF served between regular passengers
- Arrival rate of FF passengers
- Arrival rate of R passengers
- Service time

Queuing Theory

Constraints

We will maintain a simulated clock Counts in integer "ticks", from 0

At each tick

- Frequent flyer (FF) passenger arrives in line
- Regular (R) passenger arrives in line
- One agent with priorities
  1. Agent finishes, then serves next FF passenger
  2. Agent finishes, then serves next R passenger
- Agent is idle (both lines empty)

Using some other constraints

- Max # FF served between regular passengers
- Arrival rate of FF passengers
- Arrival rate of R passengers
- Service time

Queuing Theory

Constraints

We will maintain a simulated clock Counts in integer "ticks", from 0

At each tick

- Frequent flyer (FF) passenger arrives in line
- Regular (R) passenger arrives in line
- One agent with priorities
  1. Agent finishes, then serves next FF passenger
  2. Agent finishes, then serves next R passenger
- Agent is idle (both lines empty)

Using some other constraints

- Max # FF served between regular passengers
- Arrival rate of FF passengers
- Arrival rate of R passengers
- Service time

Queuing Theory

Constraints

We will maintain a simulated clock Counts in integer “ticks”, from 0

At each tick

- Frequent flyer (FF) passenger arrives in line
- Regular (R) passenger arrives in line
- One agent with priorities
  1. Agent finishes, then serves next FF passenger
  2. Agent finishes, then serves next R passenger
- Agent is idle (both lines empty)

Using some other constraints

- Max # FF served between regular passengers
- Arrival rate of FF passengers
- Arrival rate of R passengers
- Service time

Queuing Theory

Constraints

We will maintain a simulated clock Counts in integer “ticks”, from 0

At each tick

- Frequent flyer (FF) passenger arrives in line
- Regular (R) passenger arrives in line
- One agent with priorities
  1. Agent finishes, then serves next FF passenger
  2. Agent finishes, then serves next R passenger
- Agent is idle (both lines empty)

Using some other constraints

- Max # FF served between regular passengers
- Arrival rate of FF passengers
- Arrival rate of R passengers
- Service time

Queuing Theory

Constraints

We will maintain a simulated clock Counts in integer “ticks”, from 0

At each tick

- Frequent flyer (FF) passenger arrives in line
- Regular (R) passenger arrives in line
- One agent with priorities
  1. Agent finishes, then serves next FF passenger
  2. Agent finishes, then serves next R passenger
- Agent is idle (both lines empty)

Using some other constraints

- Max # FF served between regular passengers
- Arrival rate of FF passengers
- Arrival rate of R passengers
- Service time

Queuing Theory

Constraints

We will maintain a simulated clock Counts in integer “ticks”, from 0

At each tick

- Frequent flyer (FF) passenger arrives in line
- Regular (R) passenger arrives in line
- One agent with priorities
  1. Agent finishes, then serves next FF passenger
  2. Agent finishes, then serves next R passenger
- Agent is idle (both lines empty)

Using some other constraints

- Max # FF served between regular passengers
- Arrival rate of FF passengers
- Arrival rate of R passengers
- Service time

Queuing Theory

Constraints

We will maintain a simulated clock Counts in integer "ticks", from 0

At each tick

- Frequent flyer (FF) passenger arrives in line
- Regular (R) passenger arrives in line
- One agent with priorities
  1. Agent finishes, then serves next FF passenger
  2. Agent finishes, then serves next R passenger
- Agent is idle (both lines empty)

Using some other constraints

- Max # FF served between regular passengers
- Arrival rate of FF passengers
- Arrival rate of R passengers
- Service time

Queuing Theory

Constraints

We will maintain a simulated clock Counts in integer "ticks", from 0

At each tick

- Frequent flyer (FF) passenger arrives in line
- Regular (R) passenger arrives in line
- One agent with priorities
  1. Agent finishes, then serves next FF passenger
  2. Agent finishes, then serves next R passenger
- Agent is idle (both lines empty)

Using some other constraints

- Max # FF served between regular passengers
- Arrival rate of FF passengers
- Arrival rate of R passengers
- Service time

Thus

Desired Output

- Statistics on waiting times, agent idle time, etc.
- Optionally, a detailed trace

Thus

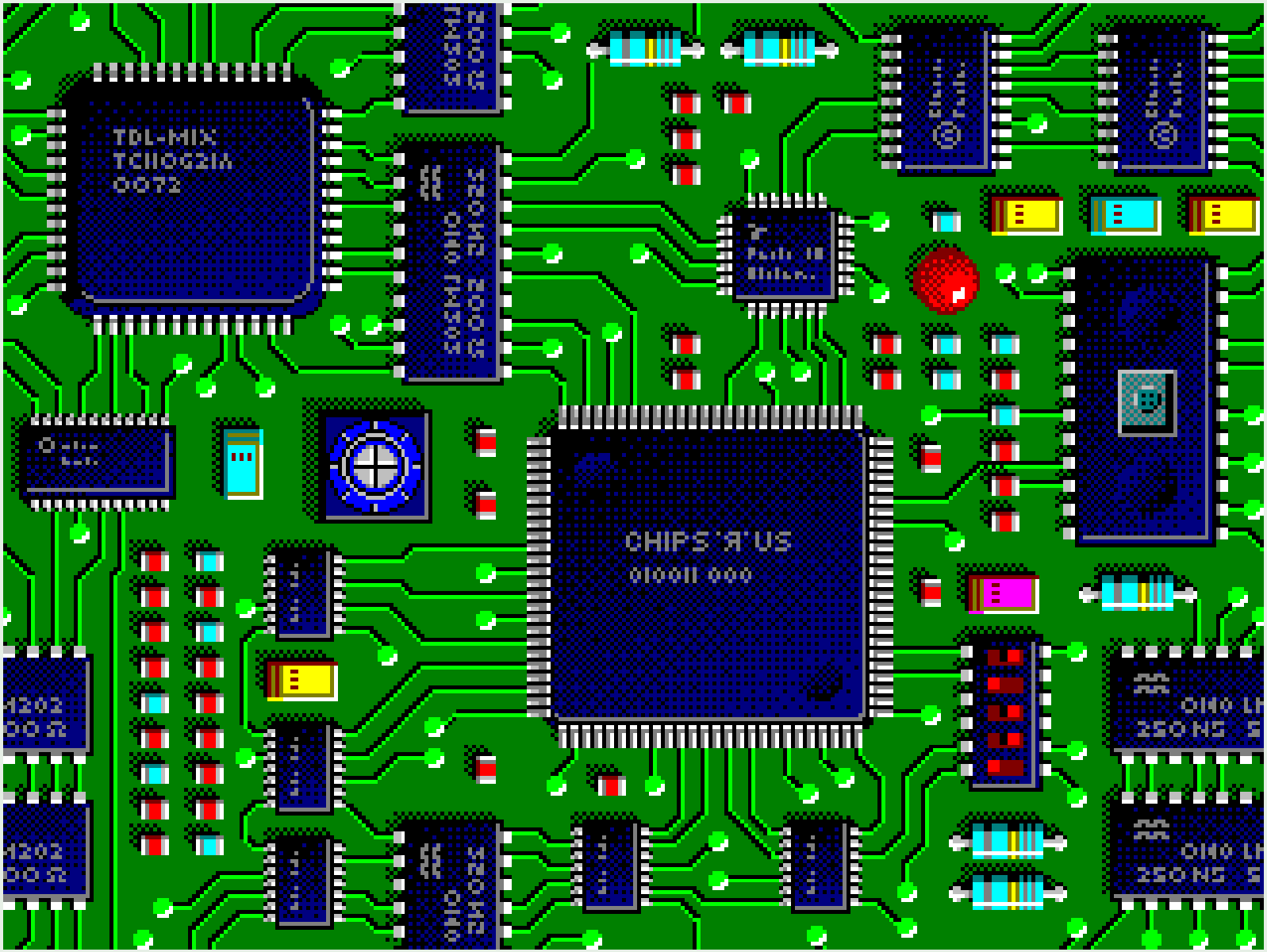
Desired Output

- Statistics on waiting times, agent idle time, etc.
- Optionally, a detailed trace

Outline

Example

Circuit Board

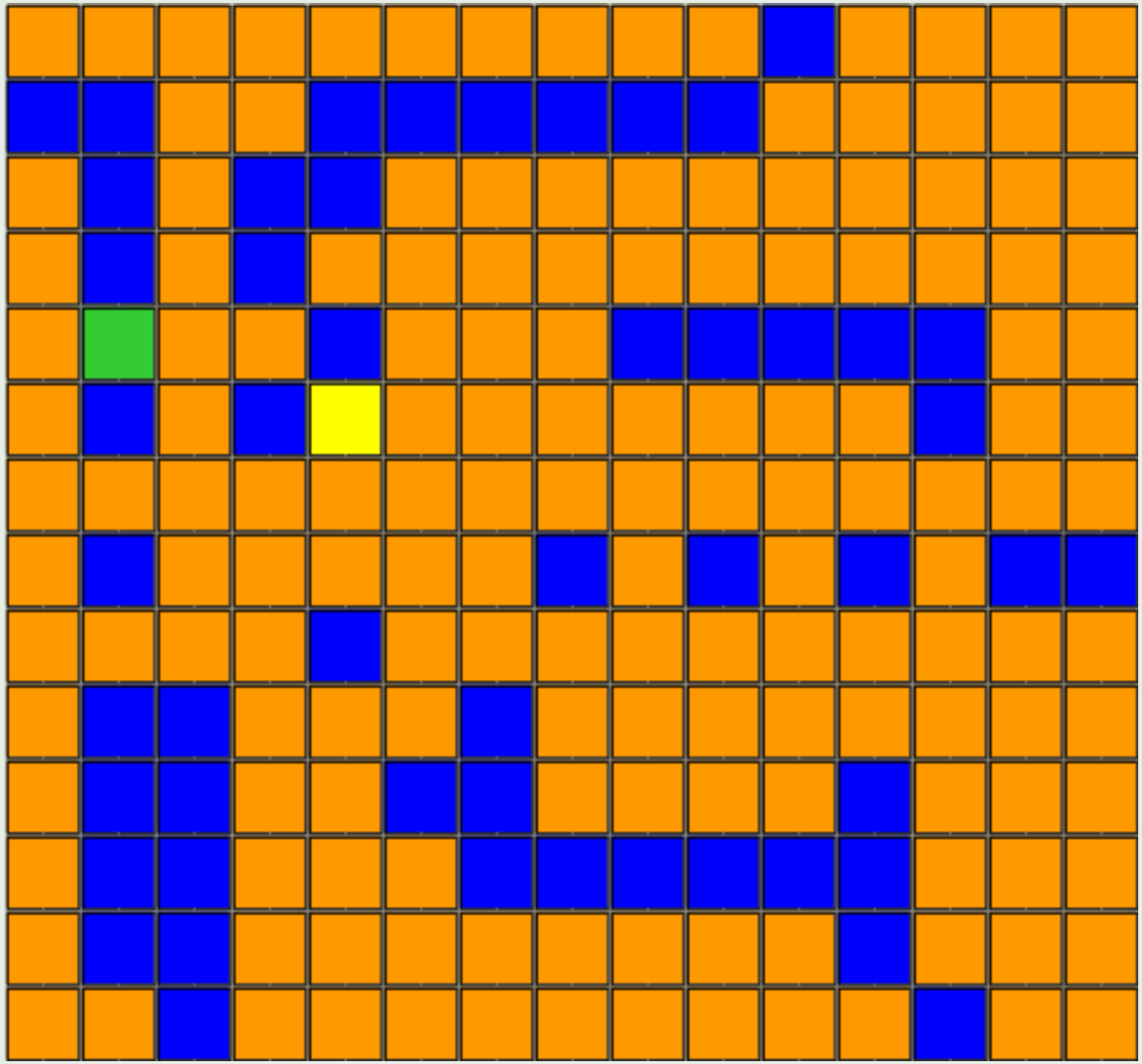


Example

Circuit Board - Label all reachable squares 1 unit from start

start pin

end pin

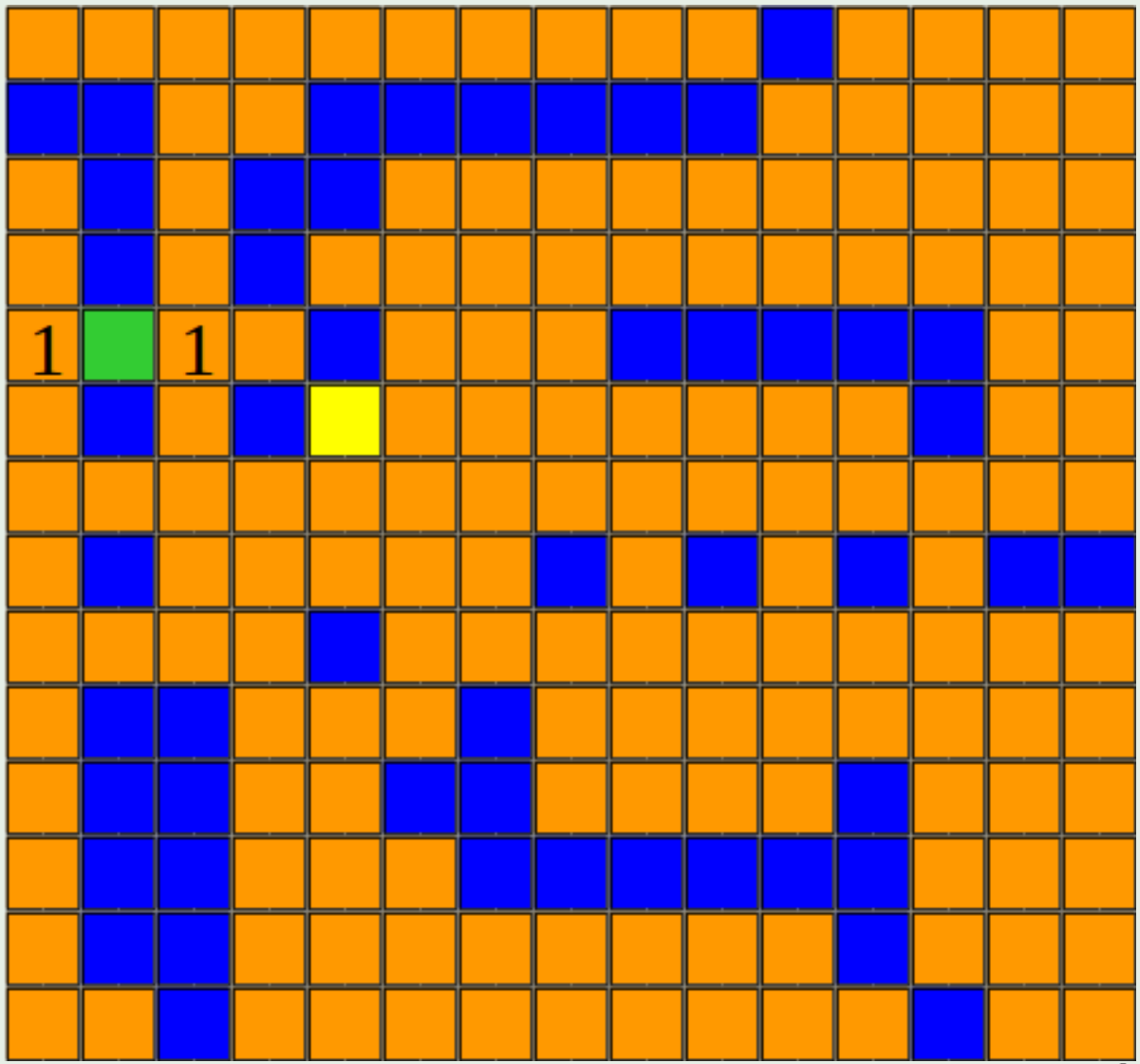


Example

Circuit Board - Label all reachable unlabeled squares 2 units from start.

start pin

end pin

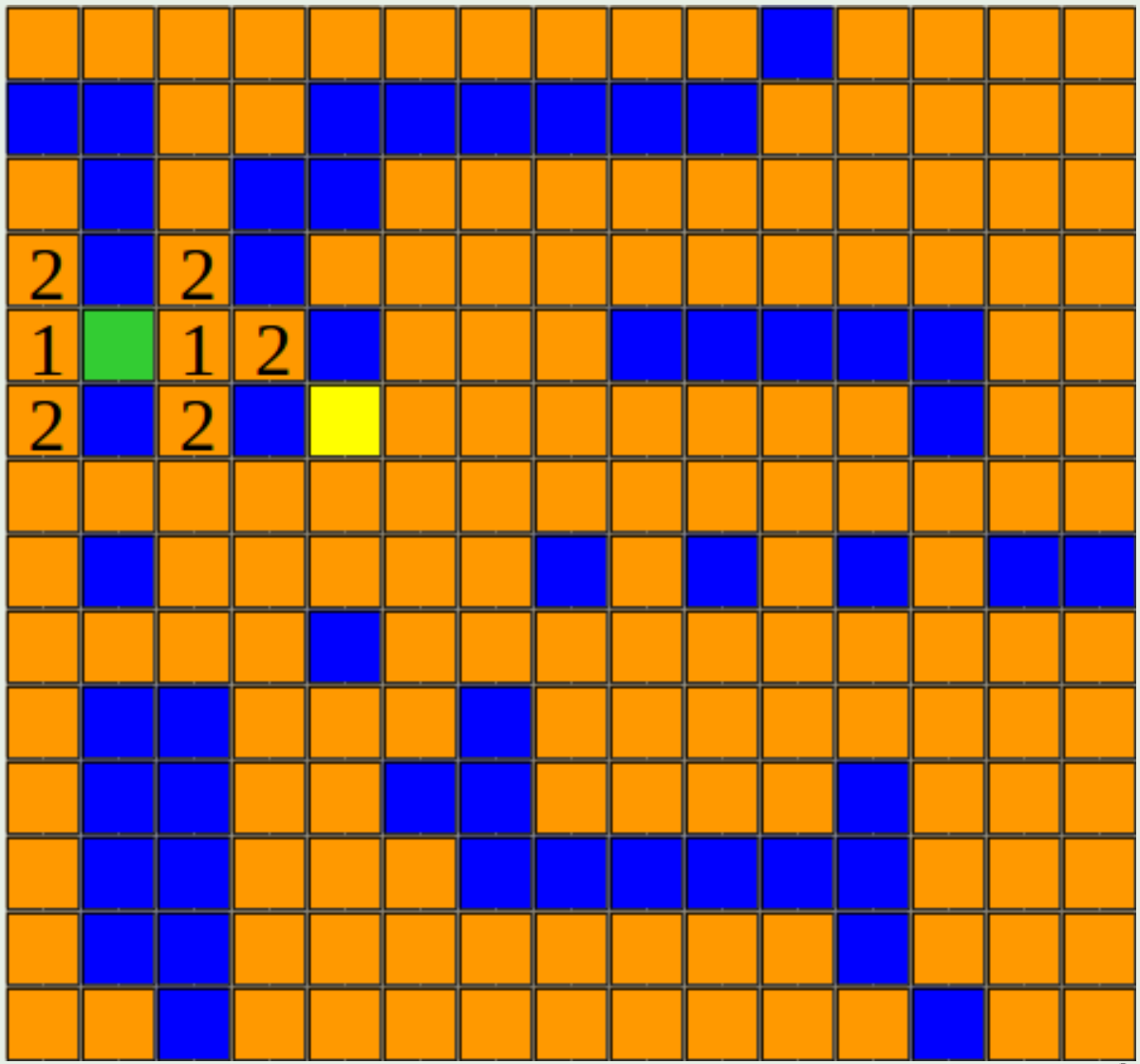


Example

Circuit Board - Label all reachable unlabeled squares 3 units from start.

start pin

end pin

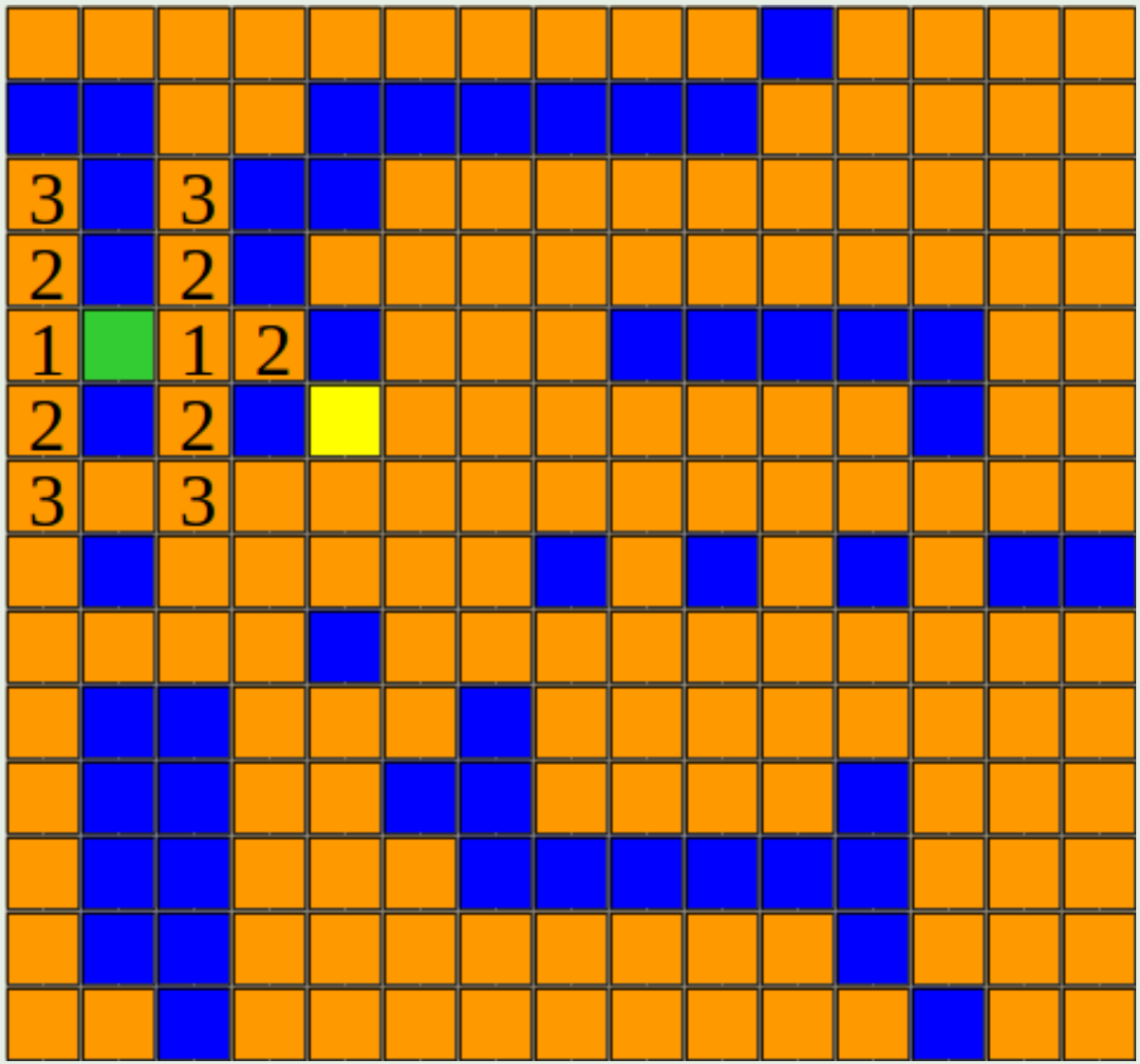


Example

Circuit Board - Label all reachable unlabeled squares 4 units from start.

start pin

end pin

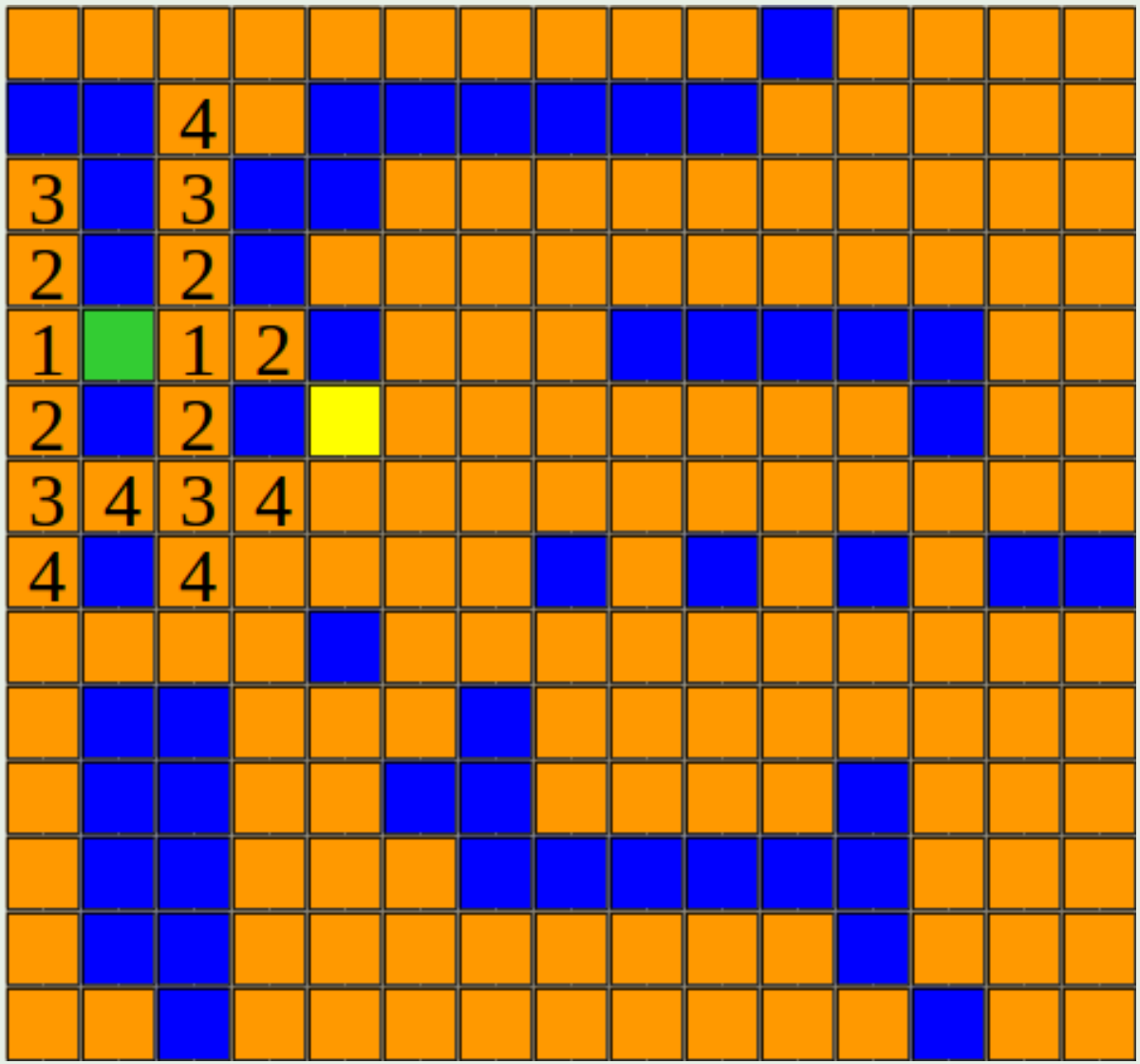


Example

Circuit Board - Label all reachable unlabeled squares 5 units from start.

start pin

end pin

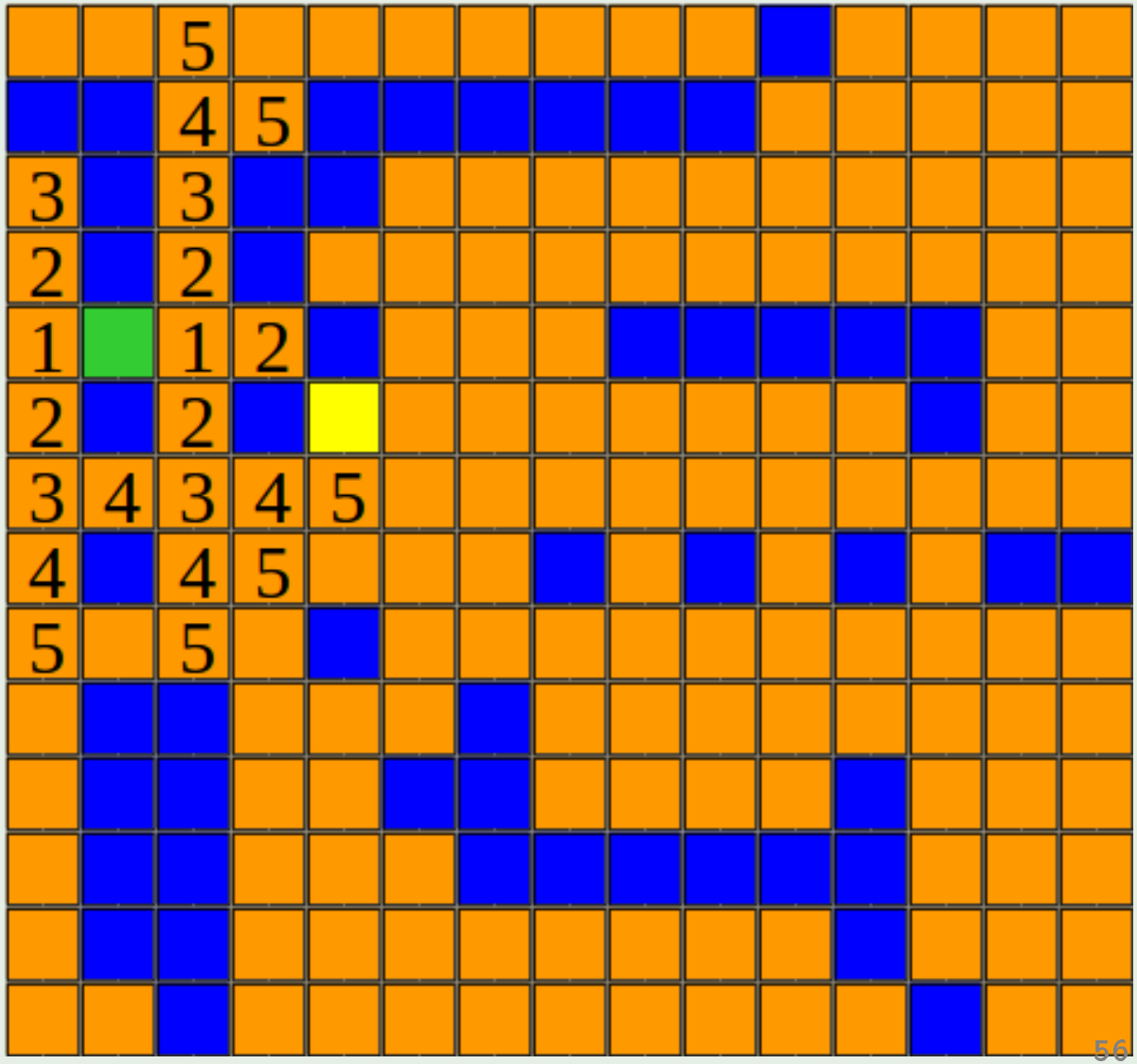


Example

Circuit Board - Label all reachable unlabeled squares 6 units from start.

start pin

end pin

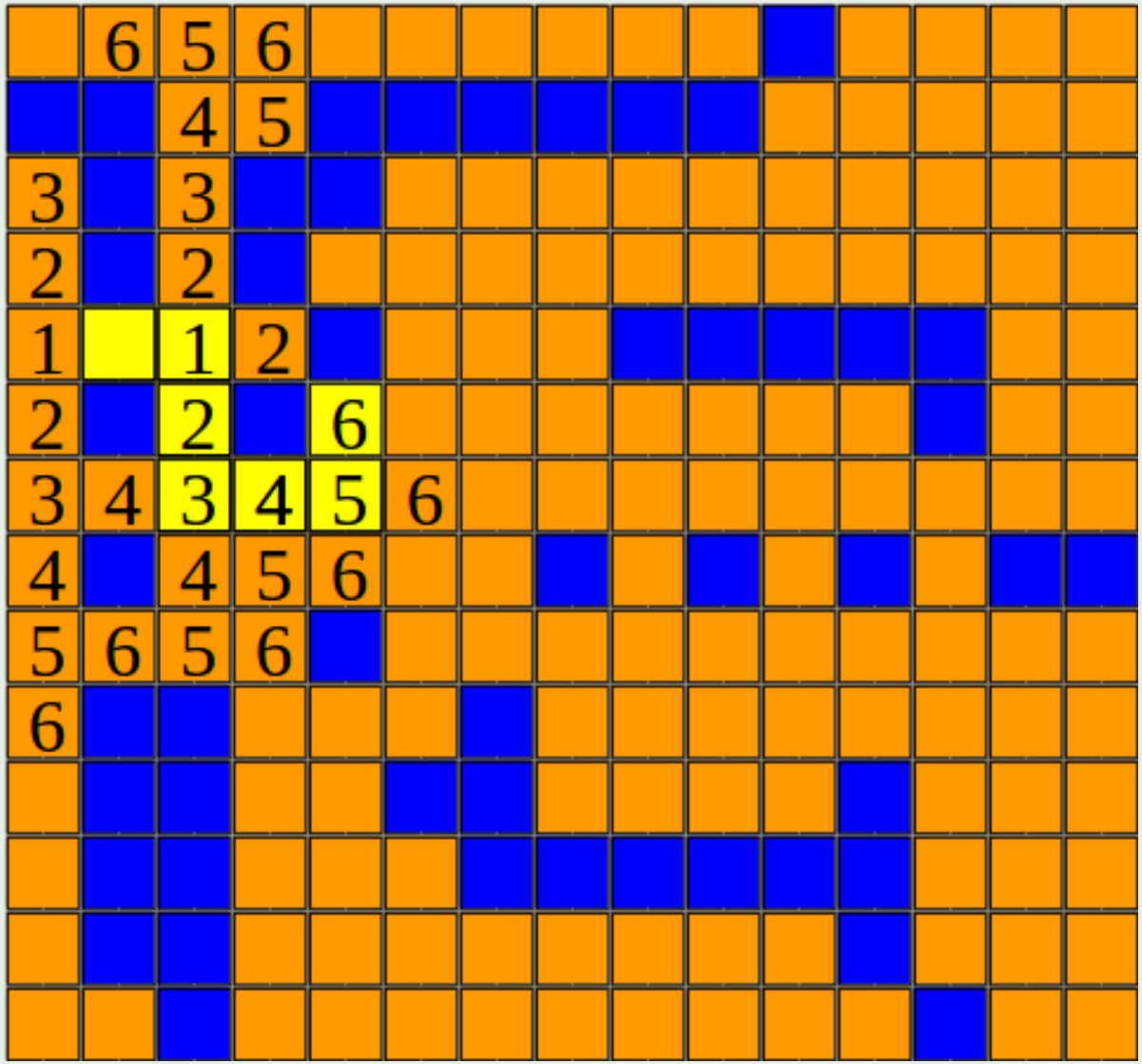


Example

Circuit Board - Traceback.

start pin

end pin



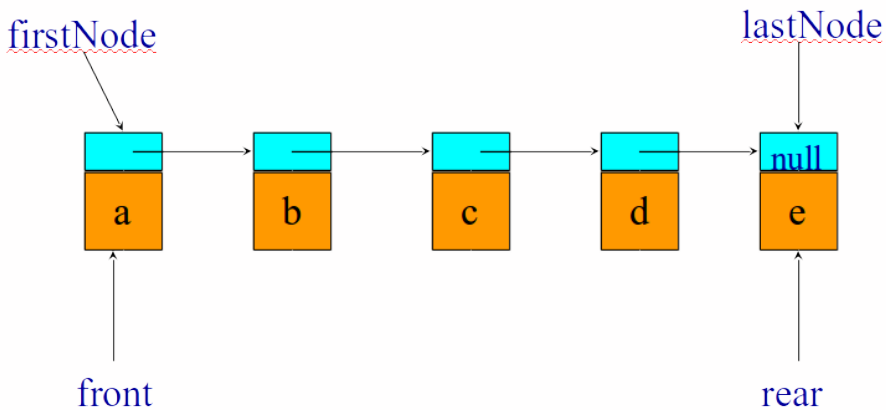
What Implementations

Directly Extending Classes

- From ArrayLinearList

  Note: Not a so good idea!!!

Extending from class but adding some pointers



From Scratch

Circular Array

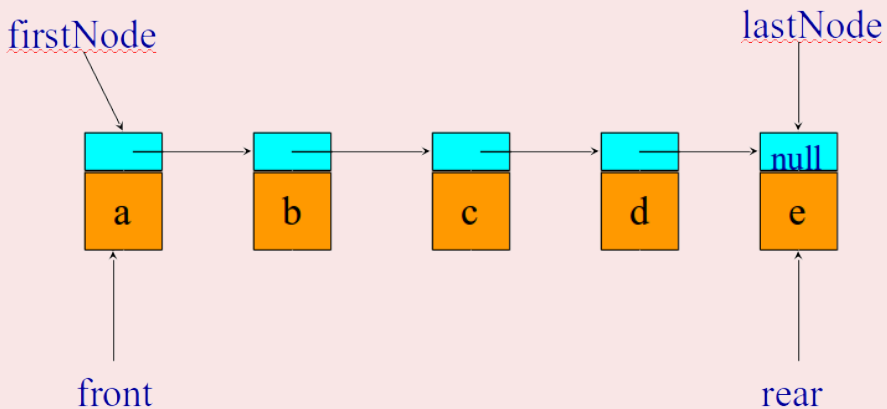
What Implementations

Directly Extending Classes

- From ArrayLinearList

  Note: Not a so good idea!!!

Extending from class but adding some pointers



From Scratch

Circular Array

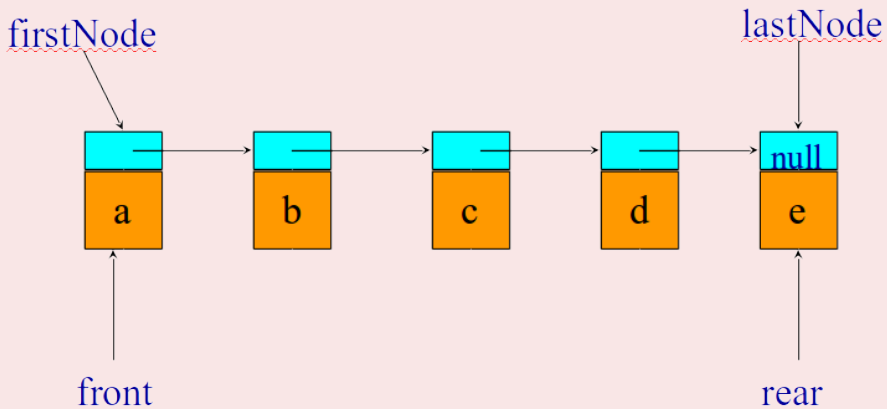
What Implementations

Directly Extending Classes

- From ArrayLinearList

  Note: Not a so good idea!!!

Extending from class but adding some pointers



From Scratch

Circular Array

Outline

Derive from ArrayLinearList

Here

We do not extend our data structure.

Simply use

Whatever is available in the base class.

Derive from ArrayLinearList

Here

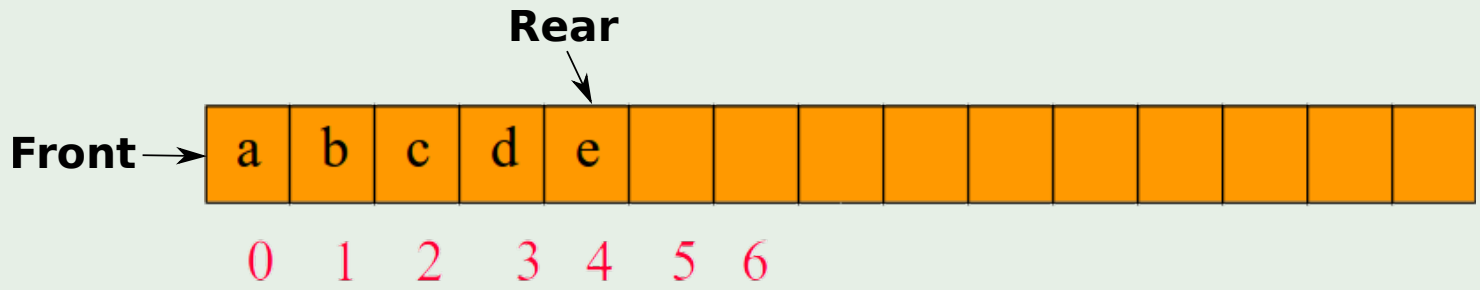
We do not extend our data structure.

Simply use

Whatever is available in the base class.

Derive from ArrayLinearList

We have then

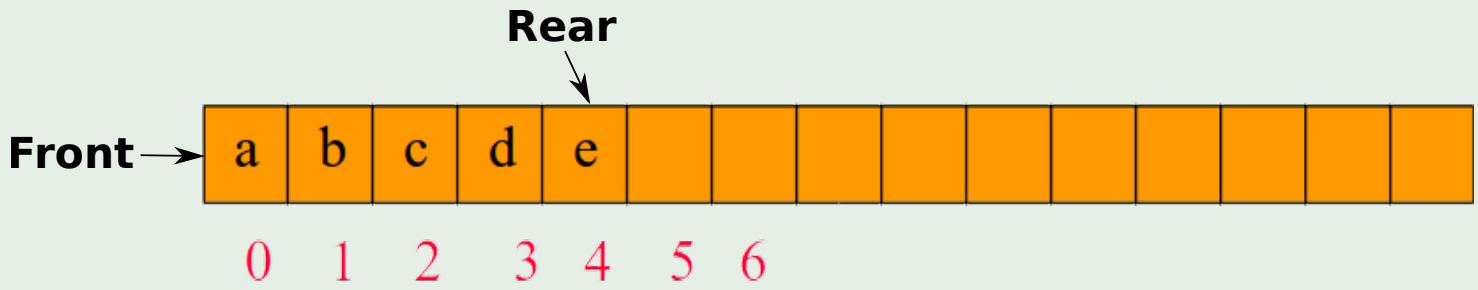


- When the **front** is the left end of list and the **rear** is the right end

Operation in Queue	Supporting method from parent class	Complexity
empty()	super.isEmpty()	O(1)
front()	get(0)	O(1)
rear()	get(size()-1)	O(1)
Enqueue(TheObject)	add(size(),theObject)	O(1)
Dequeue()	remove(0)	O(size)

Derive from ArrayLinearList

We have then

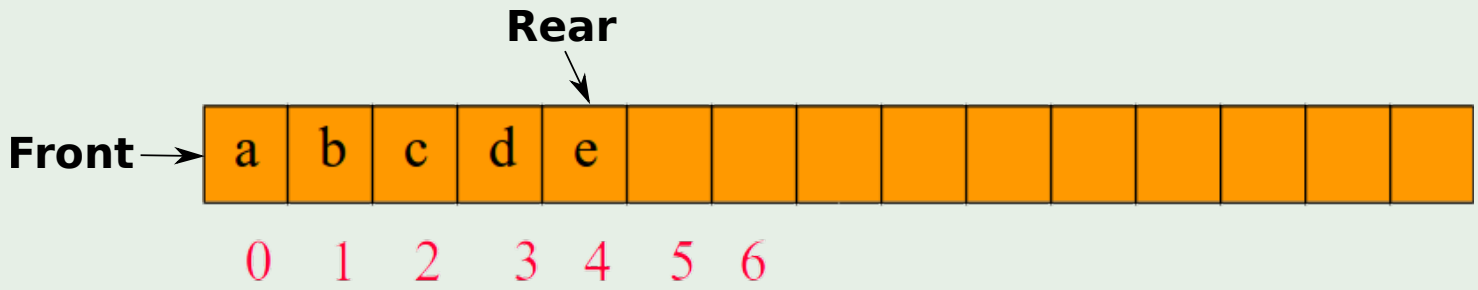


- When the **front is the left** end of list and the **rear is the right** end

Operation in Queue	Supporting method from parent class	Complexity
empty()	super.isEmpty()	O(1)
front()	get(0)	O(1)
rear()	get(size()-1)	O(1)
Enqueue(TheObject)	add(size(),theObject)	O(1)
Dequeue()	remove(0)	O(size)

Derive from ArrayLinearList

We have then

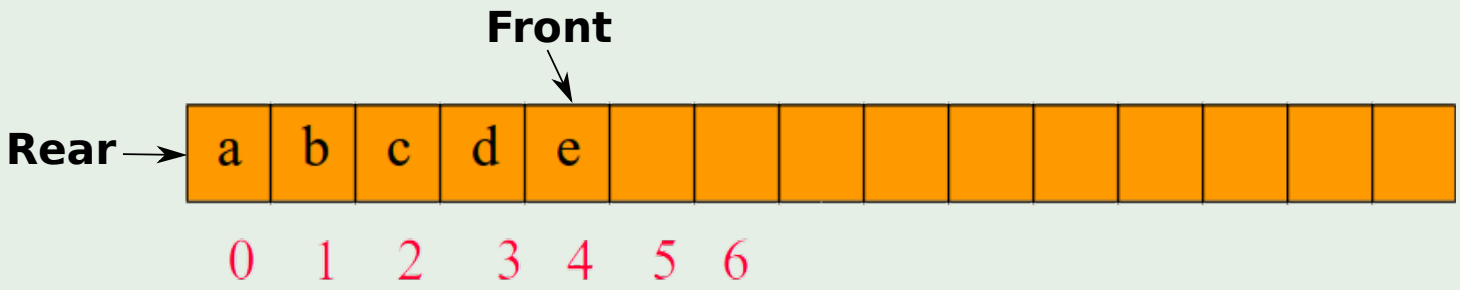


- When the **front is the left** end of list and the **rear is the right** end

Operation in Queue	Supporting method from parent class	Complexity
empty()	super.isEmpty()	O(1)
front()	get(0)	O(1)
rear()	get(size()-1)	O(1)
Enqueue(TheObject)	add(size(),theObject)	O(1)
Dequeue()	remove(0)	**O(size)**

Shift the front and rear pointers!!!

We have

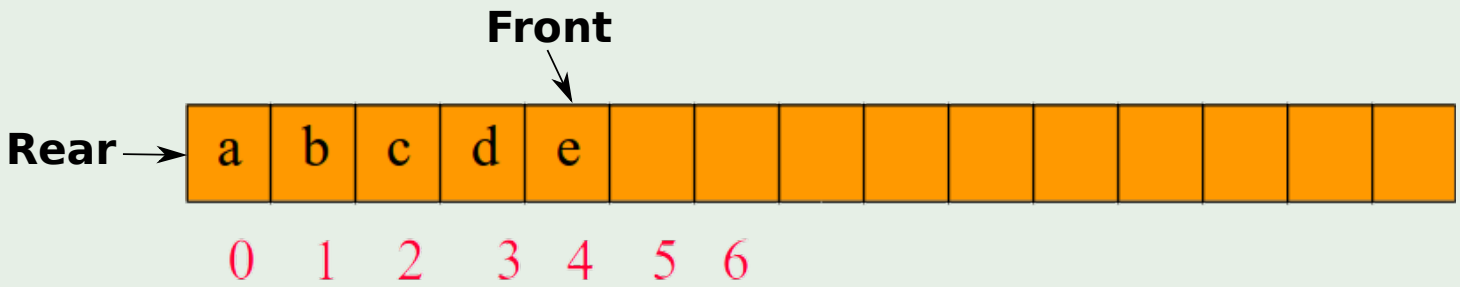


- When the rear is the left end of list and the front is the right end

Operation in Queue	Supporting method from parent class	Complexity
empty()	super.isEmpty()	O(1)
front()	get(size()-1)	O(1)
rear()	get(0)	O(1)
Enqueue(TheObject)	add(0,theObject)	O(size)
Dequeue()	remove(size()-1)	O(1)

Shift the front and rear pointers!!!

We have

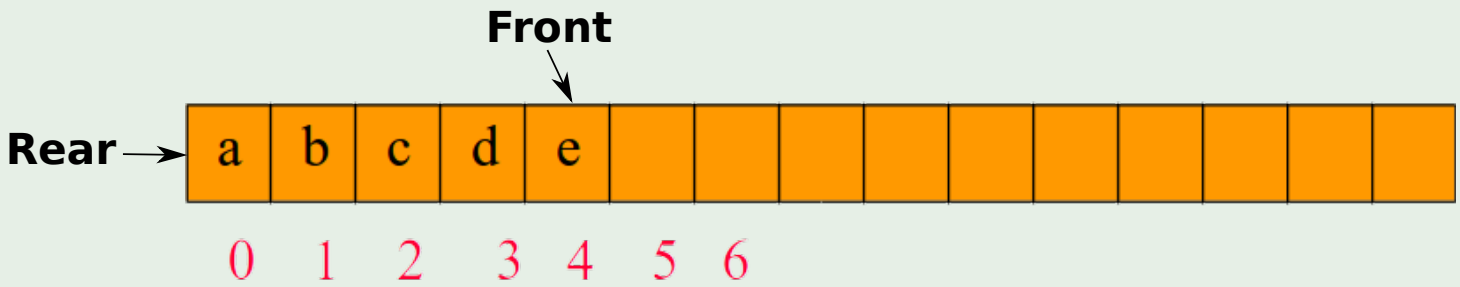


- When the **rear is the left end** of list and the **front is the right end**

Operation in Queue	Supporting method from parent class	Complexity
empty()	super.isEmpty()	O(1)
front()	get(size()-1)	O(1)
rear()	get(0)	O(1)
Enqueue(TheObject)	add(0,theObject)	O(size)
Dequeue()	remove(size()-1)	O(1)

Shift the front and rear pointers!!!

We have



- When the **rear is the left end** of list and the **front is the right end**

Operation in Queue	Supporting method from parent class	Complexity
empty()	super.isEmpty()	O(1)
front()	get(size()-1)	O(1)
rear()	get(0)	O(1)
Enqueue(TheObject)	add(0,theObject)	**O(size)**
Dequeue()	remove(size()-1)	O(1)

Moral of the Story

We have

to perform each operation in O(1) time (excluding array doubling), we need a customized array representation.

We need to extend the data structure

We can do that using the circular idea!!!

Moral of the Story

We have

to perform each operation in O(1) time (excluding array doubling), we need a customized array representation.

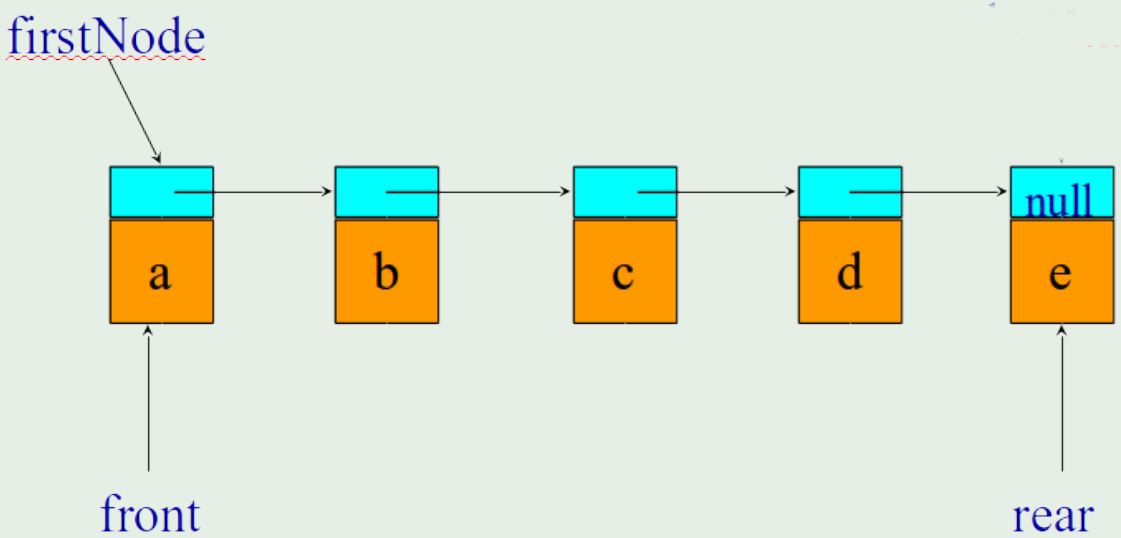
We need to extend the data structure

We can do that using the circular idea!!!

Outline

What if we derive directly from the Chain List

We have

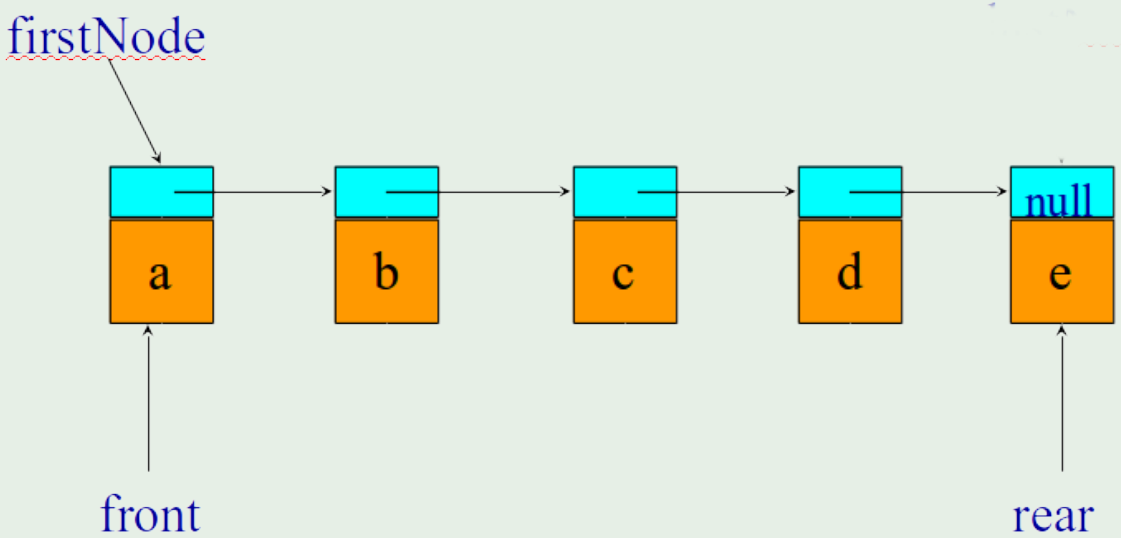


What about the operations?

We might decide that the first node is the front and the last node is the rear!!!

What if we derive directly from the Chain List

We have



What about the operations?

We might decide that the **first node is the front** and **the last node is the rear**!!!

We have the following

Operations

- Queue.empty() => super.isEmpty()
  - O(1) time
- front() => get(0) No problem here
  - O(1) time

We have the following

Operations

- Queue.empty() => super.isEmpty()
  - O(1) time
- front() => get(0) No problem here
  - O(1) time

What about these ones

- rear() => get(size()-1)
  - O(size) time
- Enqueue(theObject) => add(0,TheObject)
  - O(1) time
- Dequeue() => remove(size()-1)
  - O(size) time

We have the following

Operations

- Queue.empty() => super.isEmpty()
  - O(1) time
- front() => get(0) No problem here
  - O(1) time

What about these ones

- rear() => get(size()-1)
  - O(size) time
- Enqueue(theObject) => add(0,TheObject)
  - O(1) time
- Dequeue() => remove(size()-1)
  - O(size) time

We have the following

Operations

- Queue.empty() => super.isEmpty()
  - O(1) time
- front() => get(0) No problem here
  - O(1) time

What about these ones

- rear() => get(size()-1)
  - O(size) time
- Enqueue(theObject) => add(0,TheObject)
  - O(1) time
- Dequeue() => remove(size()-1)
  - O(size) time

We have the following

Operations

- Queue.empty() => super.isEmpty()
  - O(1) time
- front() => get(0) No problem here
  - O(1) time

What about these ones

- rear() => get(size()-1)
  - O(size) time
- Enqueue(theObject) => add(0,TheObject)
  - O(1) time
- Dequeue() => remove(size()-1)
  - O(size) time

We have the following

Operations

- Queue.empty() => super.isEmpty()
  - O(1) time
- front() => get(0) No problem here
  - O(1) time

What about these ones

- rear() => get(size()-1)
  - O(size) time
- Enqueue(theObject) => add(0,TheObject)
  - O(1) time
- Dequeue() => remove(size()-1)
  - O(size) time

We have the following

Operations

- Queue.empty() => super.isEmpty()
  - O(1) time
- front() => get(0) No problem here
  - O(1) time

What about these ones

- rear() => get(size()-1)
  - O(size) time
- Enqueue(theObject) => add(0,TheObject)
  - O(1) time
- Dequeue() => remove(size()-1)
  - O(size) time

We have the following

Operations

- Queue.empty() => super.isEmpty()
  - O(1) time
- front() => get(0) No problem here
  - O(1) time

What about these ones

- rear() => get(size()-1)
  - O(size) time
- Enqueue(theObject) => add(0,TheObject)
  - O(1) time
- Dequeue() => remove(size()-1)
  - O(size) time

We have the following

Operations

- Queue.empty() => super.isEmpty()
  - O(1) time
- front() => get(0) No problem here
  - O(1) time

What about these ones

- rear() => get(size()-1)
  - O(size) time
- Enqueue(theObject) => add(0,TheObject)
  - O(1) time
- Dequeue() => remove(size()-1)
  - O(size) time

We have the following

Operations

- Queue.empty() => super.isEmpty()
  - O(1) time
- front() => get(0) No problem here
  - O(1) time

What about these ones

- rear() => get(size()-1)
  - O(size) time
- Enqueue(theObject) => add(0,TheObject)
  - O(1) time
- Dequeue() => remove(size()-1)
  - O(size) time

Not Good At ALL

Even

If we change the front and the rear we do not get the performance we want!!!

Better Derive From Chain

We need to extend the data structure

To have another pointer to the last node!!!

Better Derive From Chain

We need to extend the data structure

To have another pointer to the last node!!!

We need to add other methods to the extended class

- getLast()
  - This returns the element pointed by the lastNode
- append(TheObject)
  - append elements at the end of the list

Better Derive From Chain

We need to extend the data structure

To have another pointer to the last node!!!

We need to add other methods to the extended class

- getLast()
  - This returns the element pointed by the lastNode
- append(TheObject)
  - append elements at the end of the list

Better Derive From Chain

We need to extend the data structure

To have another pointer to the last node!!!

We need to add other methods to the extended class

- getLast()
  - This returns the element pointed by the lastNode
- append(TheObject)
  - append elements at the end of the list

Better Derive From Chain

We need to extend the data structure

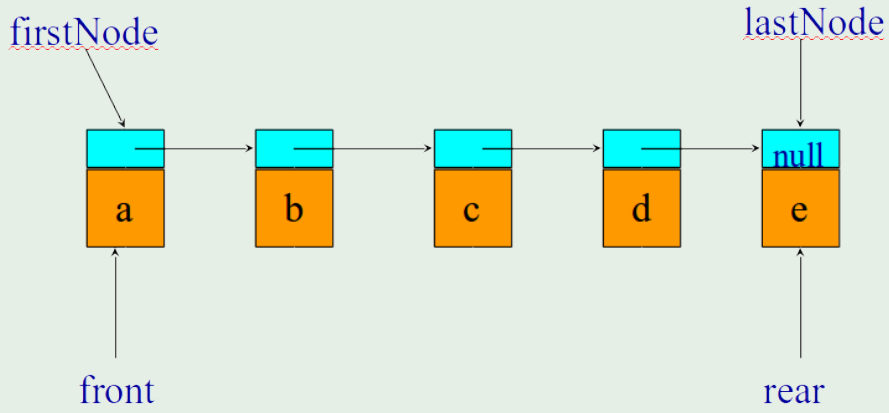
To have another pointer to the last node!!!

We need to add other methods to the extended class

- getLast()
  - This returns the element pointed by the lastNode
- append(TheObject)
  - append elements at the end of the list

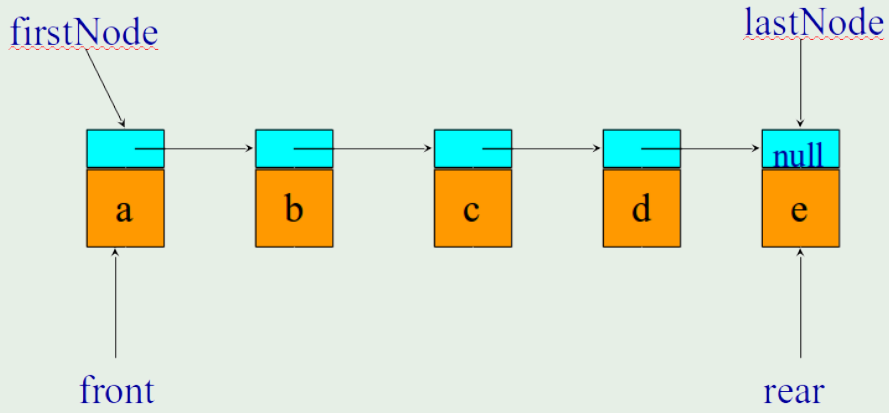
Derive From Chain

We have



Derive From Chain

We have

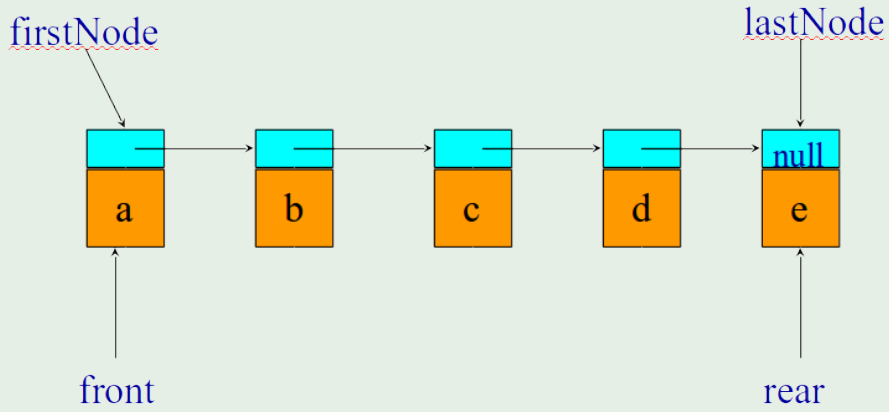


When the **front is the left end** of list and the **rear is the right end**

- Queue.empty() => super.isEmpty()
  - O(1) time
- front() => get(0)
  - O(1) time

Derive From Chain

We have

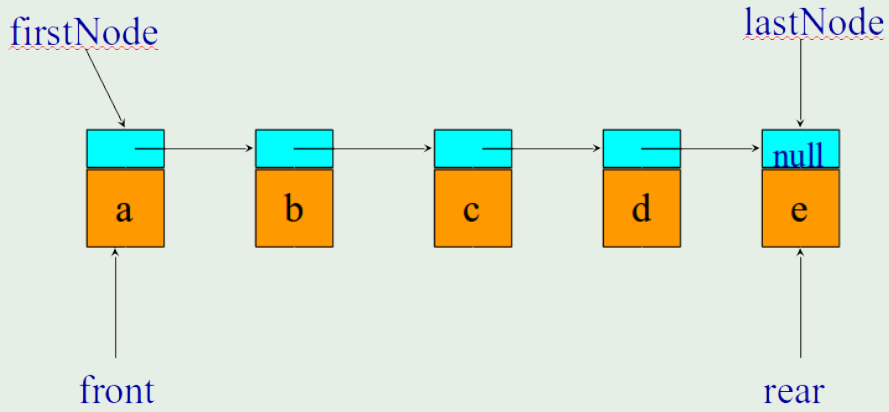


When the **front is the left end** of list and the **rear is the right end**

- Queue.empty() => super.isEmpty()
  - O(1) time
- front() => get(0)
  - O(1) time

Derive From Chain

We have

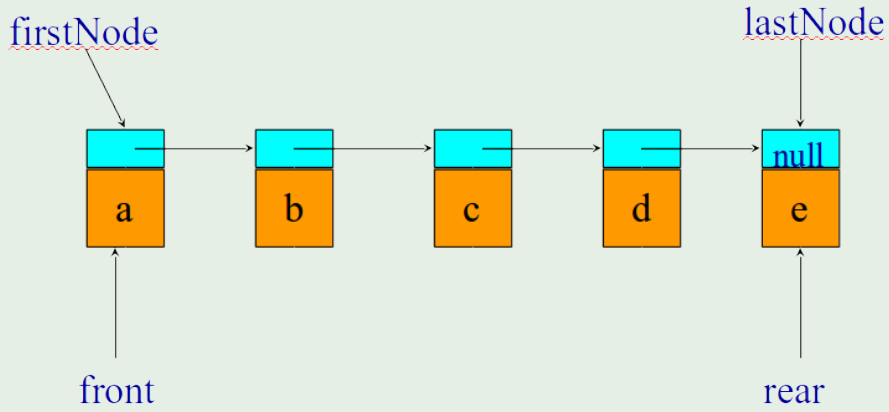


When the **front is the left end** of list and the **rear is the right end**

- Queue.empty() => super.isEmpty()
  - O(1) time
- front() => get(0)
  - O(1) time

Derive From Chain

We have

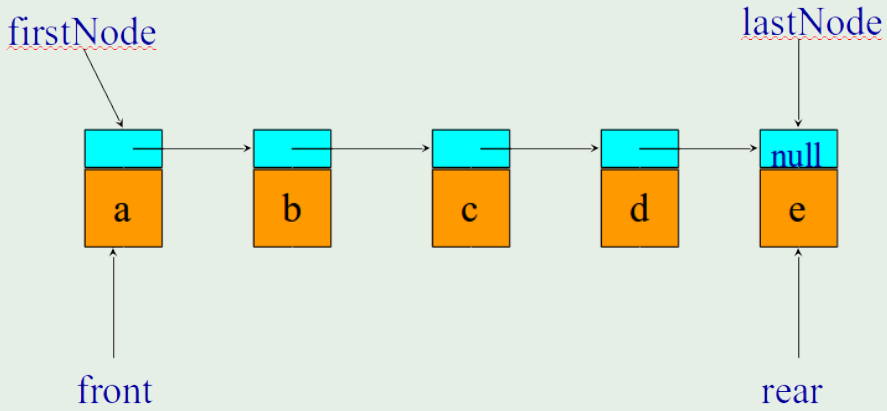


When the **front is the left end** of list and the **rear is the right end**

- Queue.empty() => super.isEmpty()
  - O(1) time
- front() => get(0)
  - O(1) time

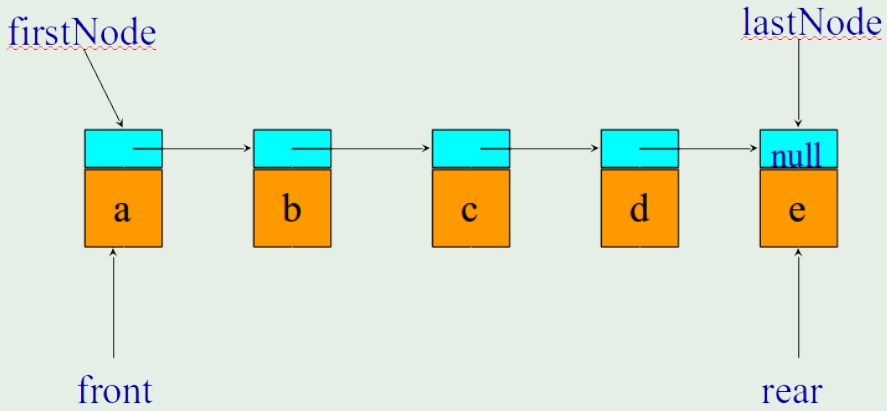
Derive From Chain

We have



Derive From Chain

We have

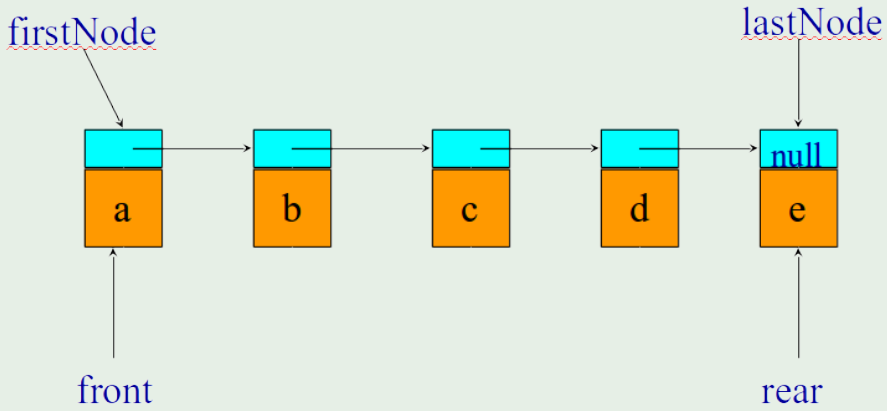


When the **front is the left end** of list and the **rear is the right end**

- rear() => getLast() . . . new method
  - O(1) time
- Enqueue(theObject) => append(theObject) . . . new method
  - O(1) time
- Dequeue() => remove(0)
  - O(1) time

Derive From Chain

We have

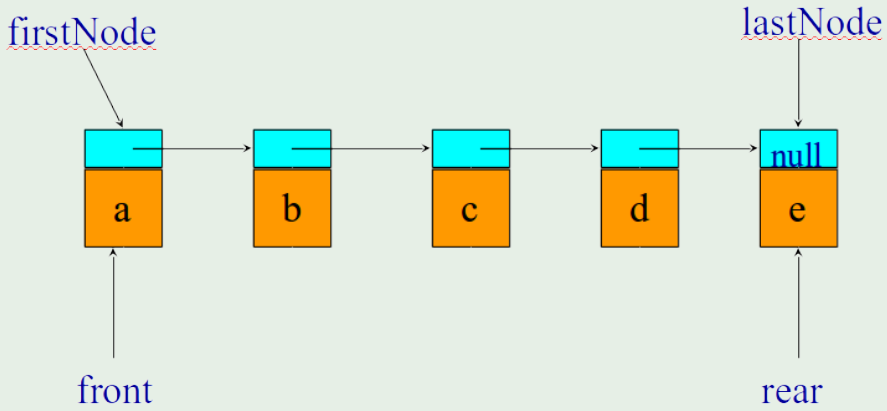


When the **front is the left end** of list and the **rear is the right end**

- rear() => getLast() . . . new method
  - O(1) time
- Enqueue(theObject) => append(theObject) . . . new method
  - O(1) time
- Dequeue() => remove(0)
  - O(1) time

Derive From Chain

We have

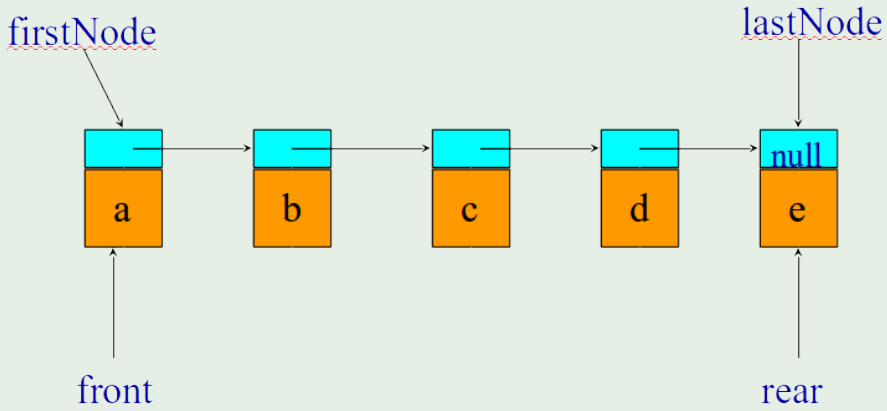


When the **front is the left end** of list and the **rear is the right end**

- rear() => getLast() . . . new method
  - O(1) time
- Enqueue(theObject) => append(theObject) . . . new method
  - O(1) time
- Dequeue() => remove(0)
  - O(1) time

Derive From Chain

We have

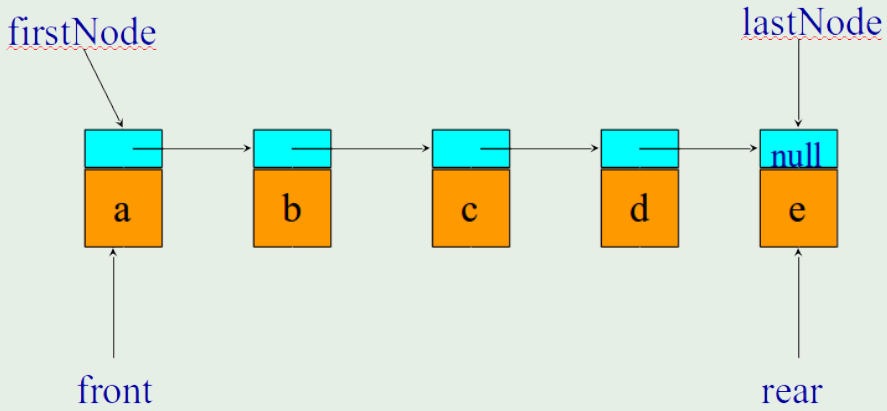


When the **front is the left end** of list and the **rear is the right end**

- rear() => getLast() . . . new method
  - O(1) time
- Enqueue(theObject) => append(theObject) . . . new method
  - O(1) time
- Dequeue() => remove(0)
  - O(1) time

Derive From Chain

We have

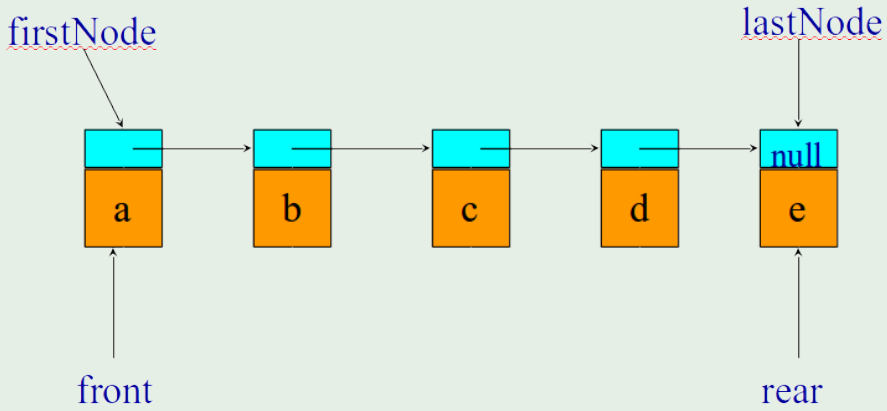


When the **front is the left end** of list and the **rear is the right end**

- rear() => getLast() . . . new method
  - O(1) time
- Enqueue(theObject) => append(theObject) . . . new method
  - O(1) time
- Dequeue() => remove(0)
  - O(1) time

Derive From Chain

We have



When the **front is the left end** of list and the **rear is the right end**

- rear() => getLast() . . . new method
  - O(1) time
- Enqueue(theObject) => append(theObject) . . . new method
  - O(1) time
- Dequeue() => remove(0)
  - O(1) time

But even with this implementation

Problems

We have do still some code that does not belong to the ADT queue!!!

But even with this implementation

Problems

We have do still some code that does not belong to the ADT queue!!!

Example

For example the idea of index checking!!!

But even with this implementation

Problems

We have do still some code that does not belong to the ADT queue!!!

Example

For example the idea of index checking!!!

Moral of the Story

Better to implement from scratch... **when possible**!!!

Outline

A Linked Implementation of a Queue

From our experience extending the Chain Class

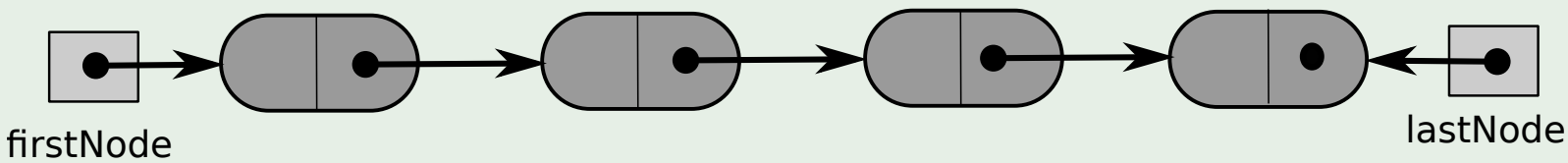
- We use two pointer for the front and the back of the chain:
  - firstNode == at the beginning of the Chain
  - lastNode == the end of the Chain

A Linked Implementation of a Queue

From our experience extending the Chain Class

- We use two pointer for the front and the back of the chain:
  - firstNode == at the beginning of the Chain
  - lastNode == the end of the Chain

Thus...



A Linked Implementation of a Queue

From our experience extending the Chain Class

- We use two pointer for the front and the back of the chain:
  - firstNode == at the beginning of the Chain
  - lastNode == the end of the Chain

Thus...

A Linked Implementation of a Queue

From our experience extending the Chain Class

- We use two pointer for the front and the back of the chain:
  - firstNode == at the beginning of the Chain
  - lastNode == the end of the Chain

Thus...

The Code For this Implementation

Sketch of the Code - Question: What is the problem with private class Node?

```
public class LinkedQueue<Item> implements Queue<Item>
{
   private Node firstNode;
   private Node lastNode;
   private int size;
   public LinkedQueue(){
      firstNode = null;
      lastNode = null;
      size = 0;
   } // end default constructor

   private class Node
   {
      private Item data; // entry in queue
      private Node next; // link to next node
    } // end Node
} // end LinkedQueue
```

Yes!!!

With private fields

It is necessary to have public methods to access them!!!

- setDataNode(Item Object)
- setNextNode(Node newNode);

Yes!!!

With private fields

It is necessary to have public methods to access them!!!

- setDataNode(Item Object)
- setNextNode(Node newNode);

Yes!!!

With private fields

It is necessary to have public methods to access them!!!

- setDataNode(Item Object)
- setNextNode(Node newNode);

Some Operations: Enqueue

We have always two cases

1. Adding to an empty Queue
2. Adding to a non-empty Queue

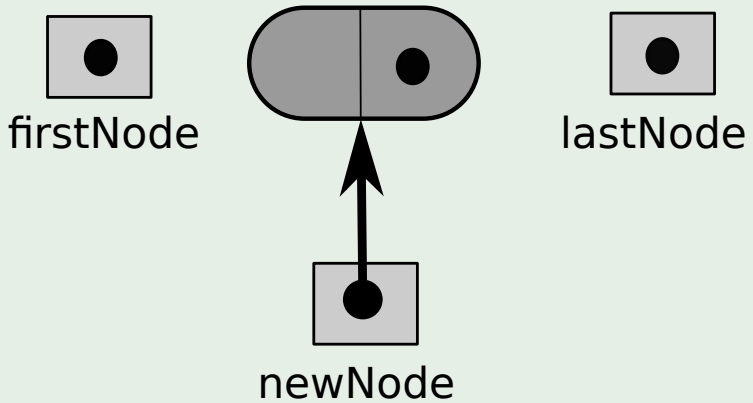
Some Operations: Enqueue

We have always two cases

1. Adding to an empty Queue
2. Adding to a non-empty Queue

Example: Adding to an Empty List

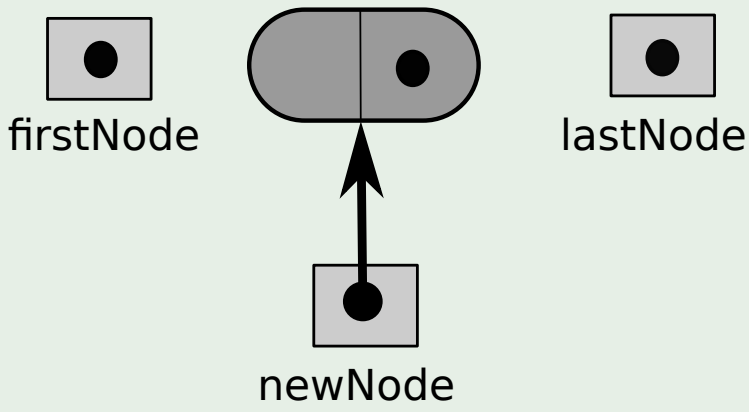
Before Inserting



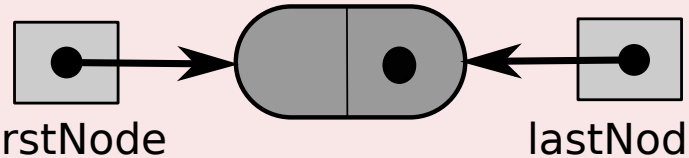
After Inserting

Example: Adding to an Empty List

Before Inserting

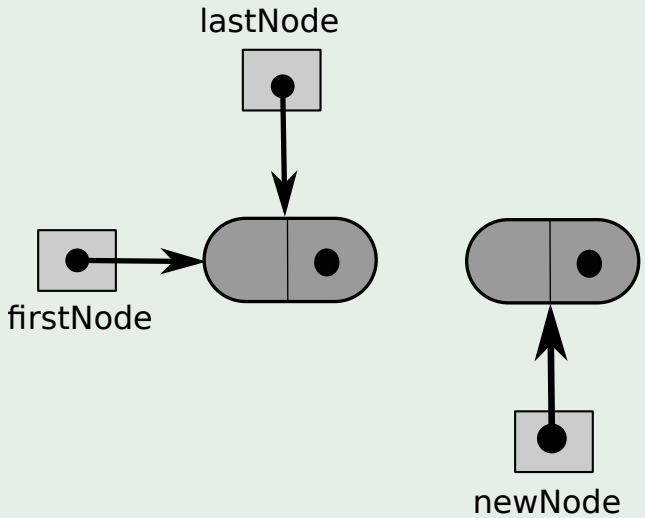


After Inserting



Example: Adding to a Non-Empty List

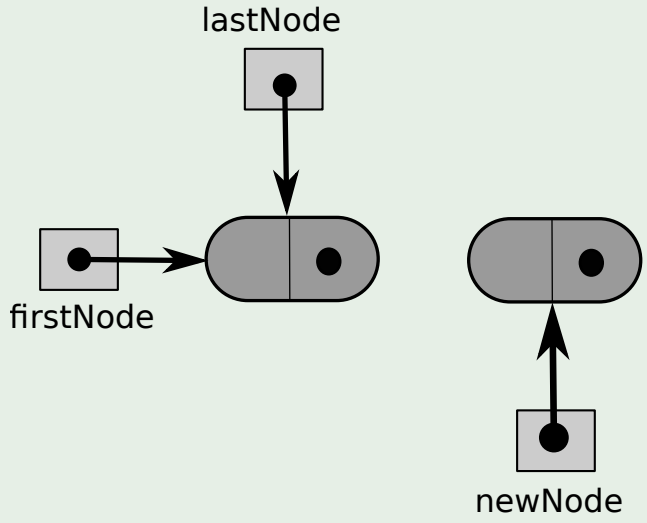
Create New Node



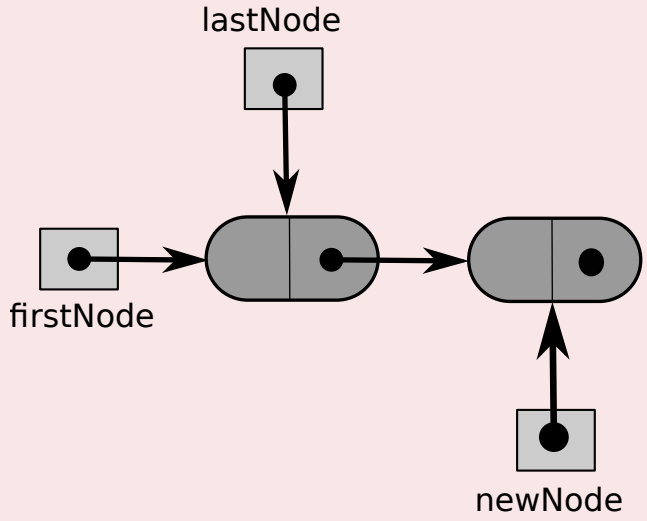
Make next from lastNode...

Example: Adding to a Non-Empty List

Create New Node

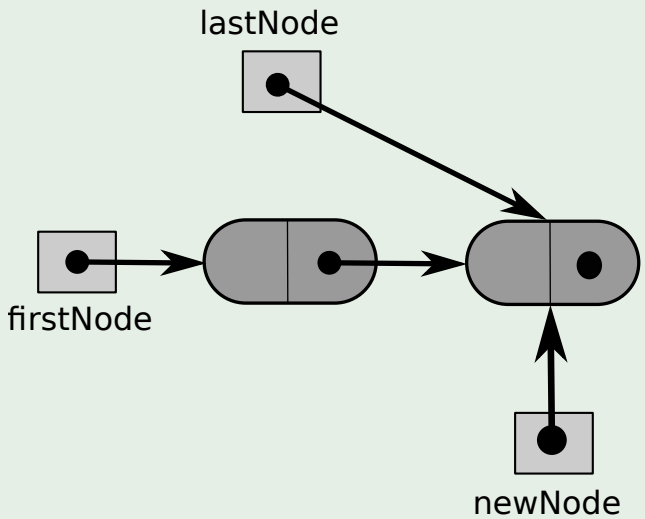


Make next from lastNode...



Example: Adding to a Non-Empty List

Move lastNode



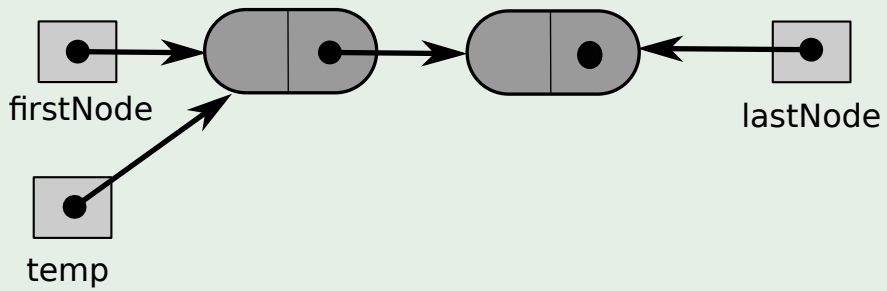
Final Code

Code

```
public void enqueue(Item newEntry)
{
    Node newNode = new Node(newEntry, null);
    if (empty())
            firstNode = newNode;
    else
            lastNode.setNextNode(newNode);
    lastNode = newNode;
} // end enqueue
```

What about Dequeue?

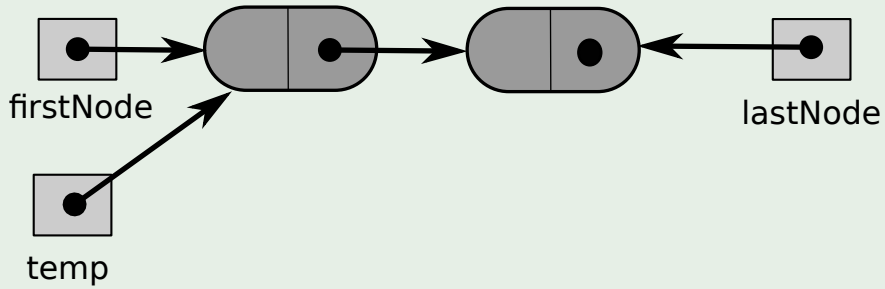
Point a temporary node to the front node



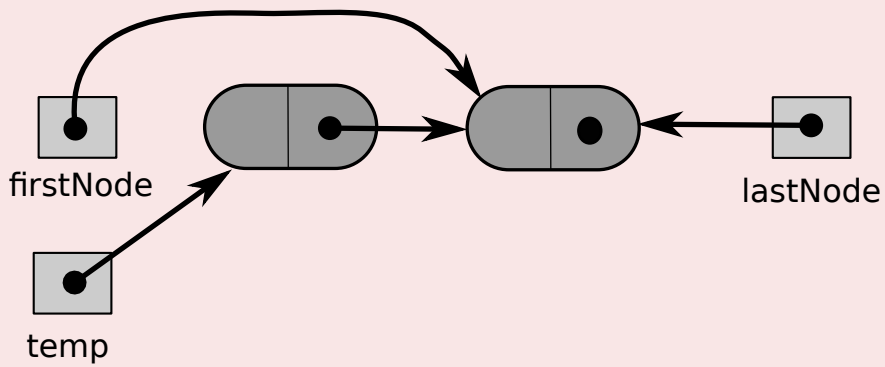
Point firstNode to temp.next

What about Dequeue?

Point a temporary node to the front node

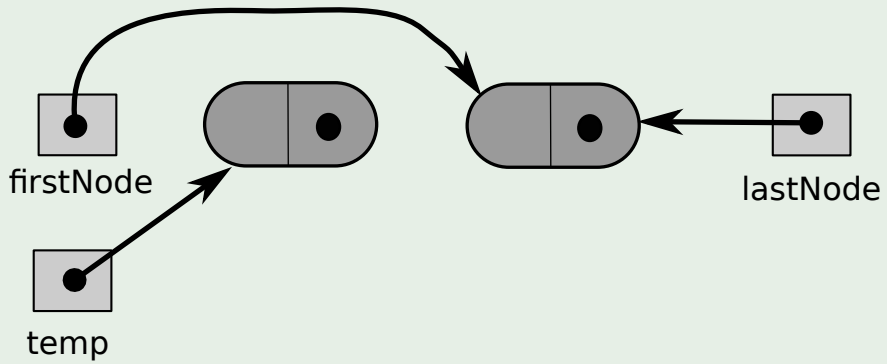


Point firstNode to temp.next



Empty

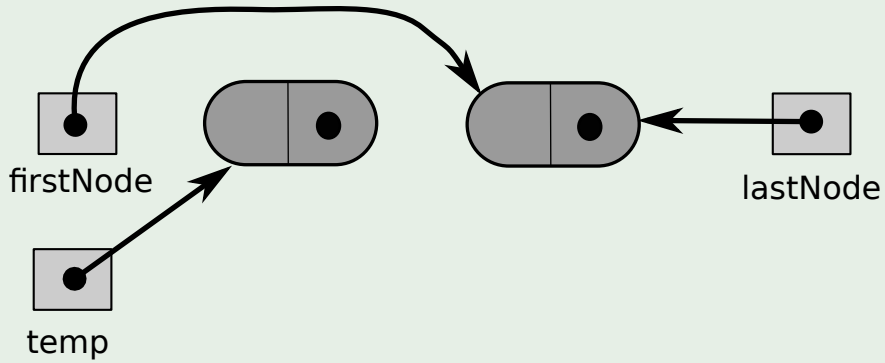
Make temp.next = null



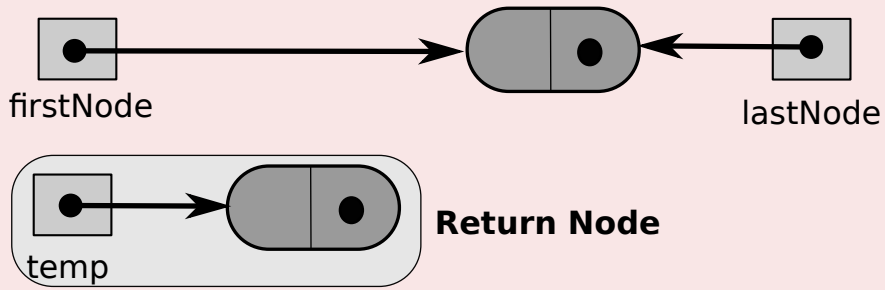
Return Node

Empty

Make temp.next = null



Return Node



Thus...

The Rest of Operations

You can think about them... they are not complex...

Complexities

Operation in Scratch Queue using Chains	Complexity
empty()	O(1)
front()	O(1)
rear()	O(1)
Enqueue(TheObject)	O(1)
Dequeue()	O(1)

Thus...

The Rest of Operations

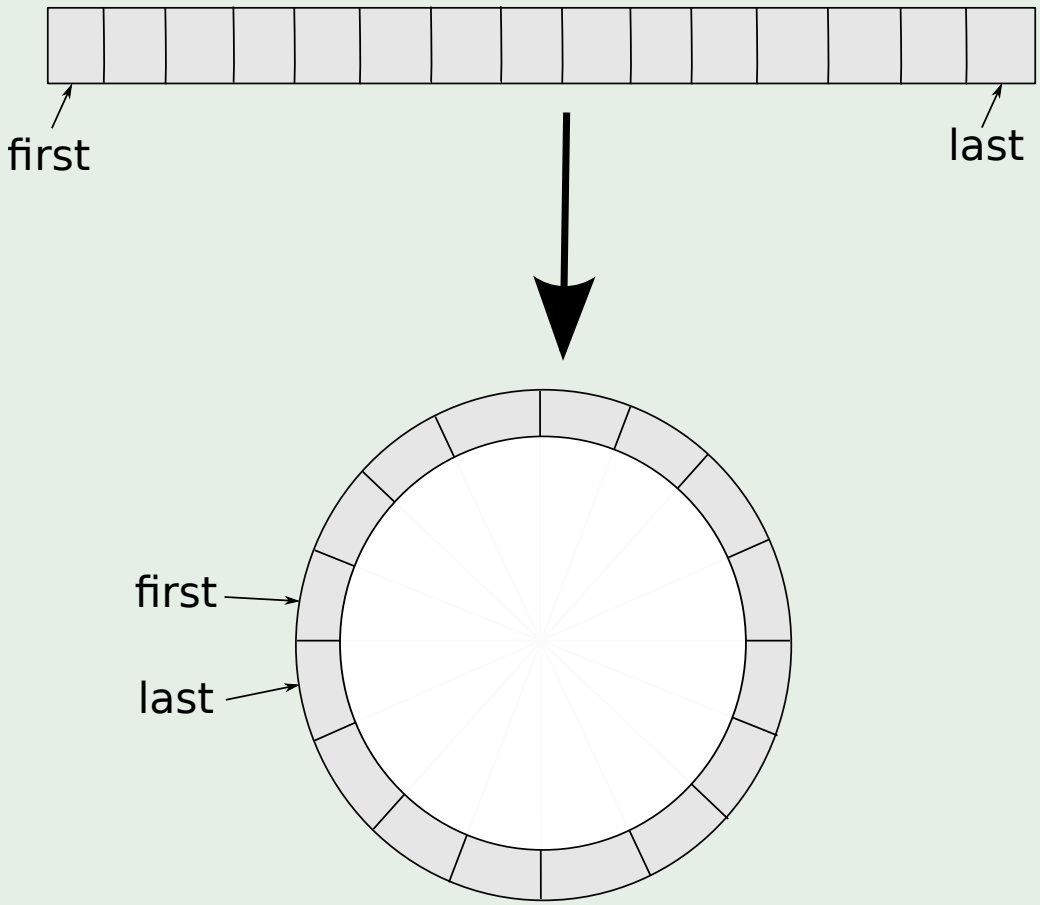
You can think about them... they are not complex...

Complexities

Operation in Scratch Queue using Chains	Complexity
empty()	O(1)
front()	O(1)
rear()	O(1)
Enqueue(TheObject)	O(1)
Dequeue()	O(1)

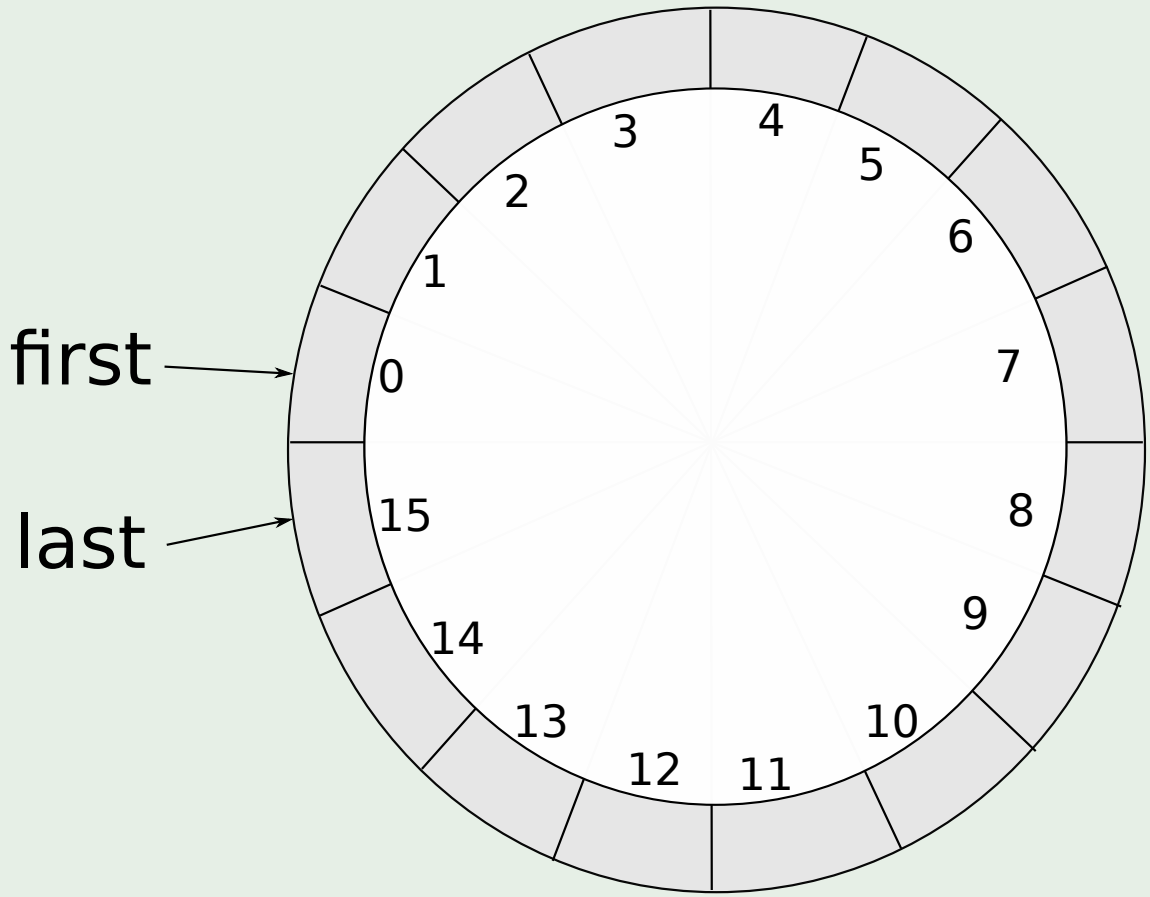
From Scratch Using an Array!!!

If we can do the following...



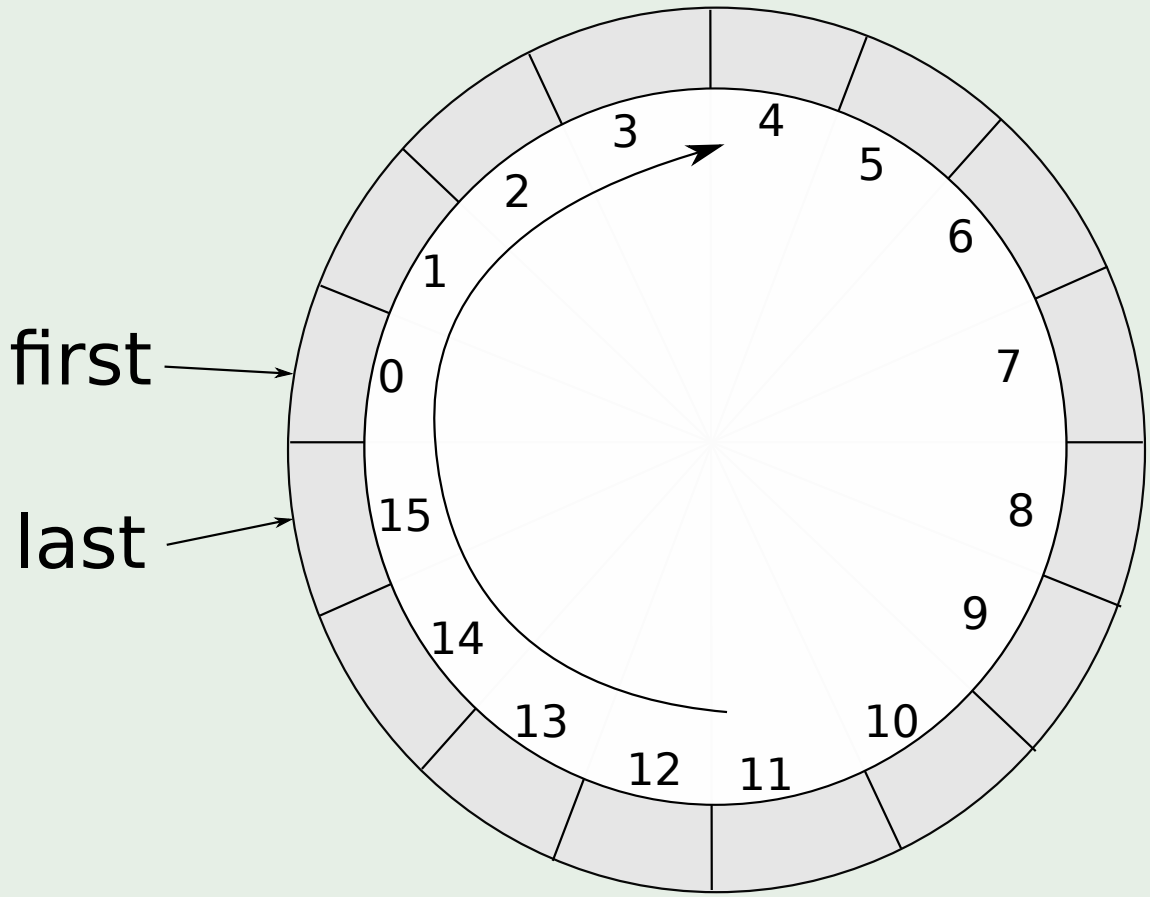
A closer Look...

Somebody can notice something?



Direction of Reading

Repeating numbers in a certain range



How we can simulate this number repetition?

Actually there is function that can help

$$\mod m : \mathbb{N} \to \{0, 1, 2, ..., m-1\} \tag{1}$$

Example for $m = 5$

n	$n \mod m$
0	0
1	1
2	2
3	3
4	4
5	0
6	1
7	2
8	3
etc...	...

How we can simulate this number repetition?

Actually there is function that can help

$$\mod m : \mathbb{N} \to \{0, 1, 2, ..., m-1\} \tag{1}$$

Example for $m = 5$

n	$n \mod m$
0	0
1	1
2	2
3	3
4	4
5	0
6	1
7	2
8	3
etc...	...

Thus, we still we have two indexes

frontIndex

We need to know where to remove!!!

backIndex

We need to know where to add!!

Thus, we still we have two indexes

frontIndex

We need to know where to remove!!!

backIndex

We need to know where to add!!

Thus

If we want to add

backIndex = (backIndex + 1)% queue.length

If we want to remove

frontIndex = (frontIndex + 1)% queue.length

Thus

If we want to add

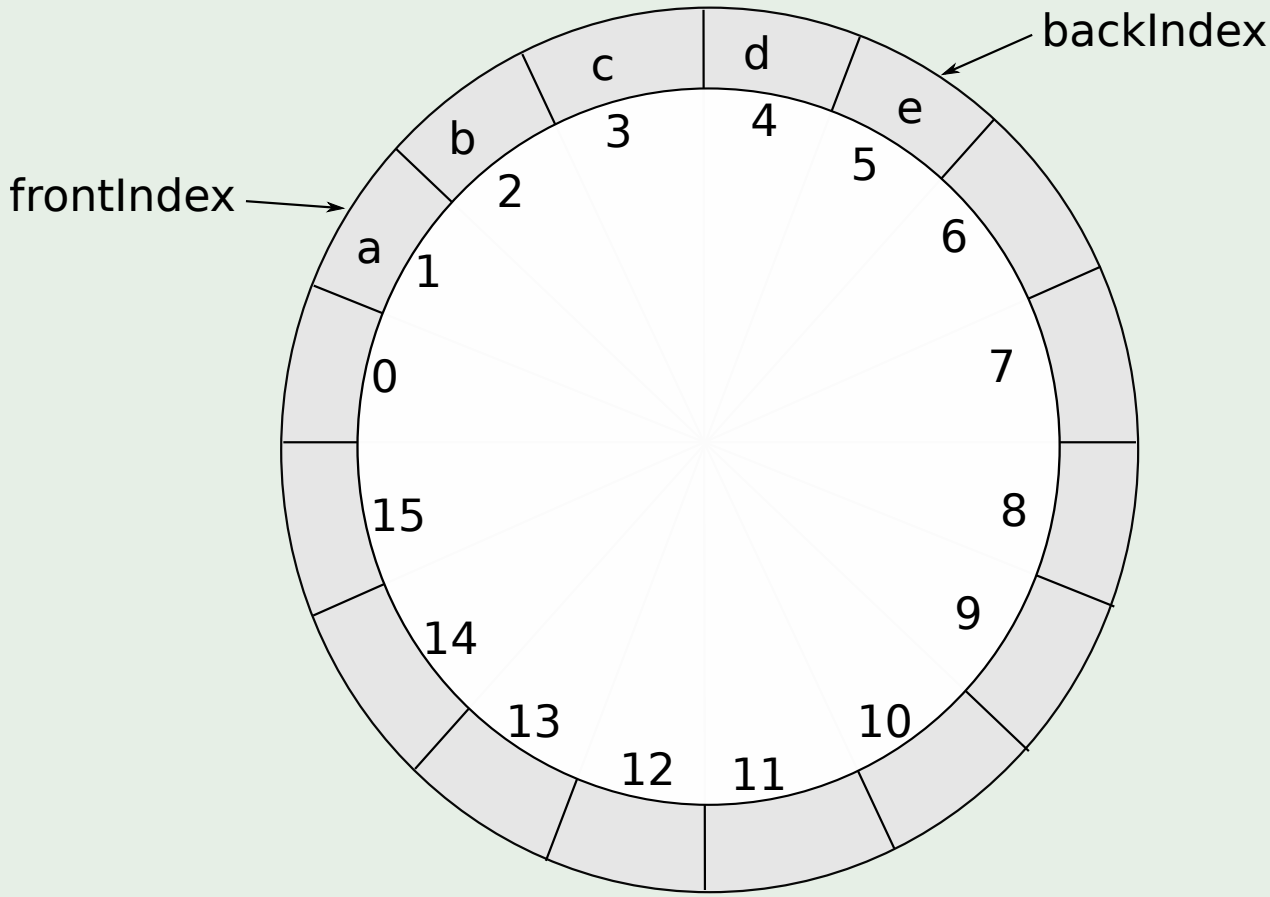
backIndex = (backIndex + 1)% queue.length

If we want to remove

frontIndex = (frontIndex + 1)% queue.length

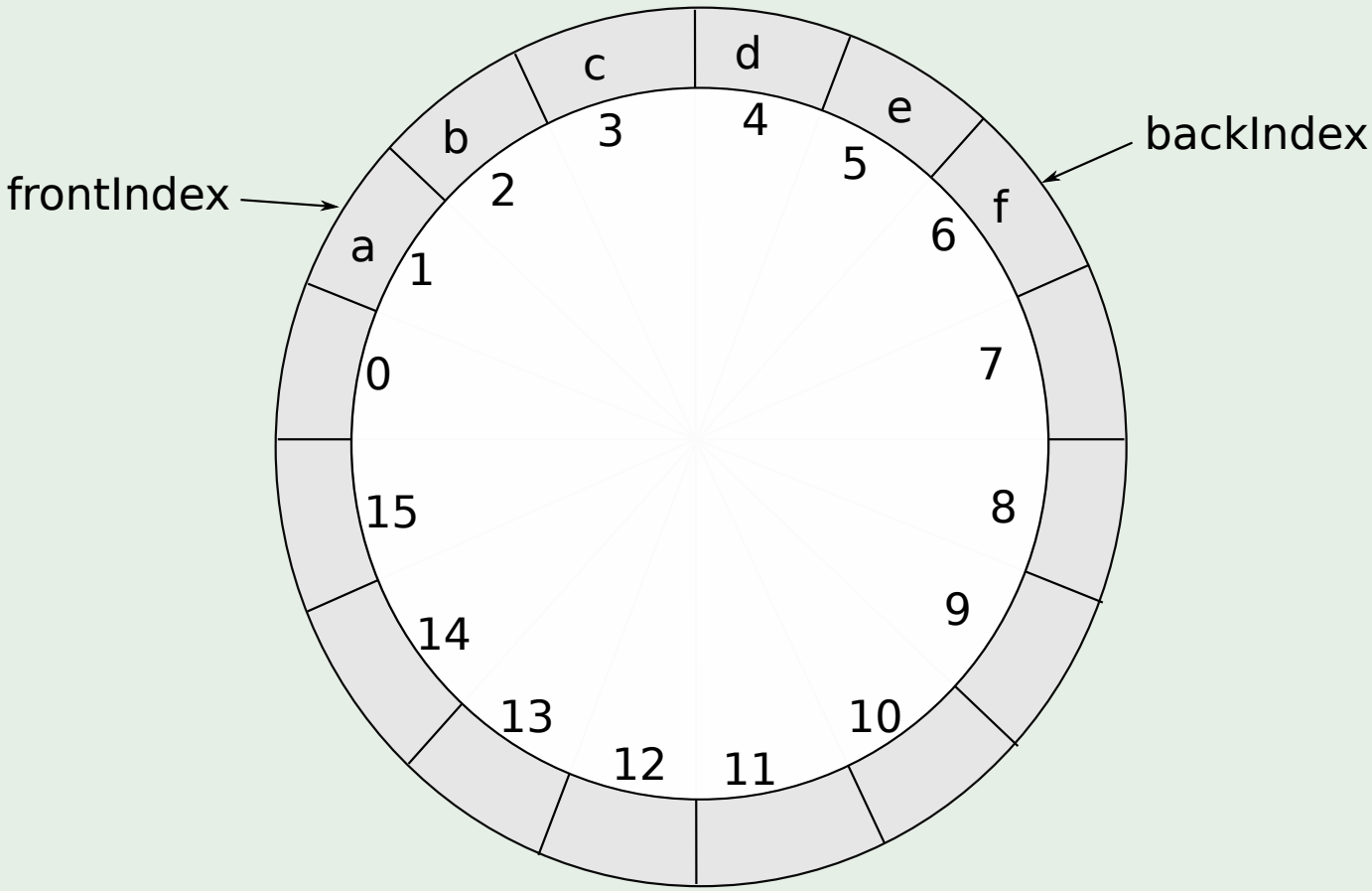
Example

Adding stuff into the back



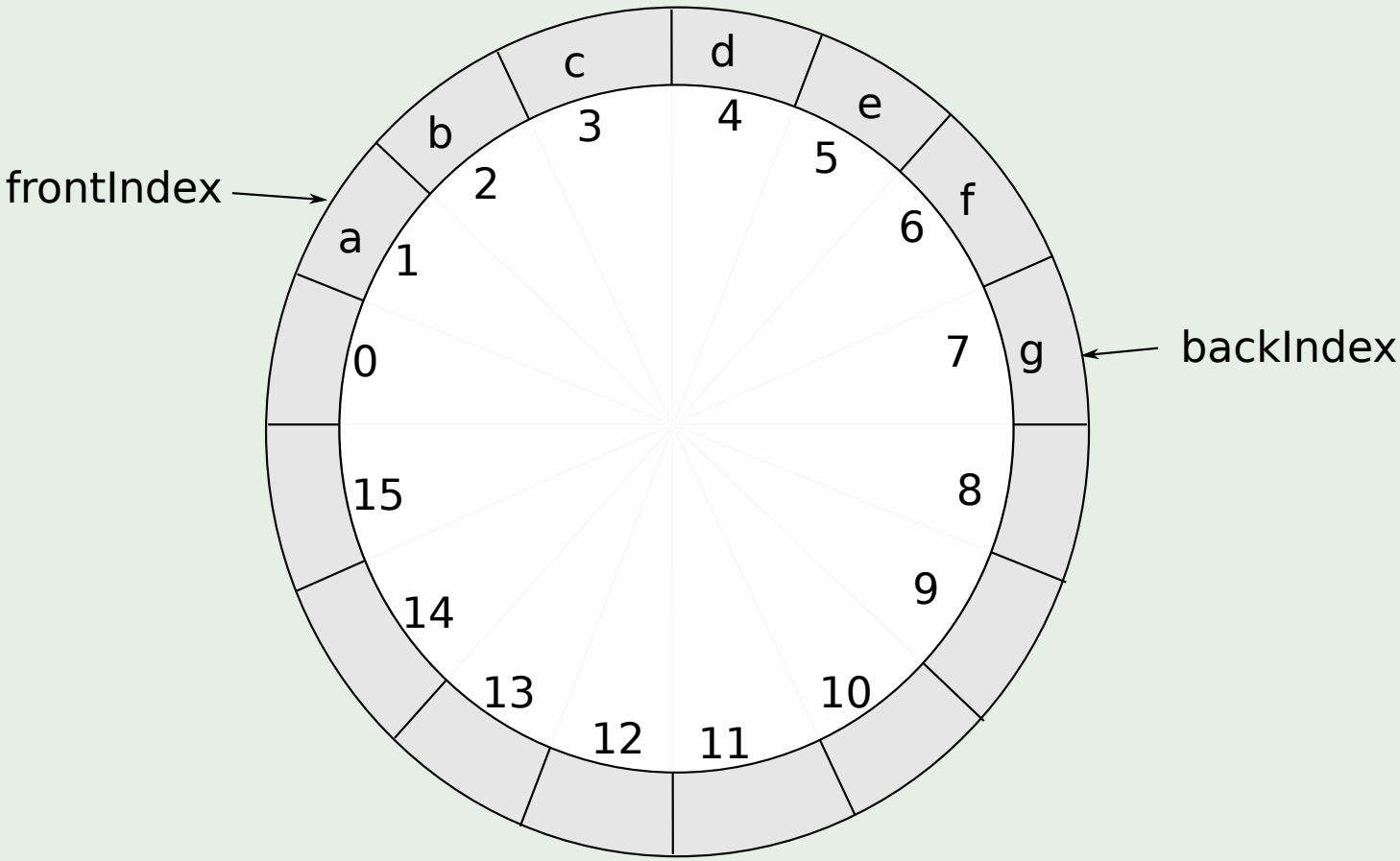
Example

Adding stuff into the back



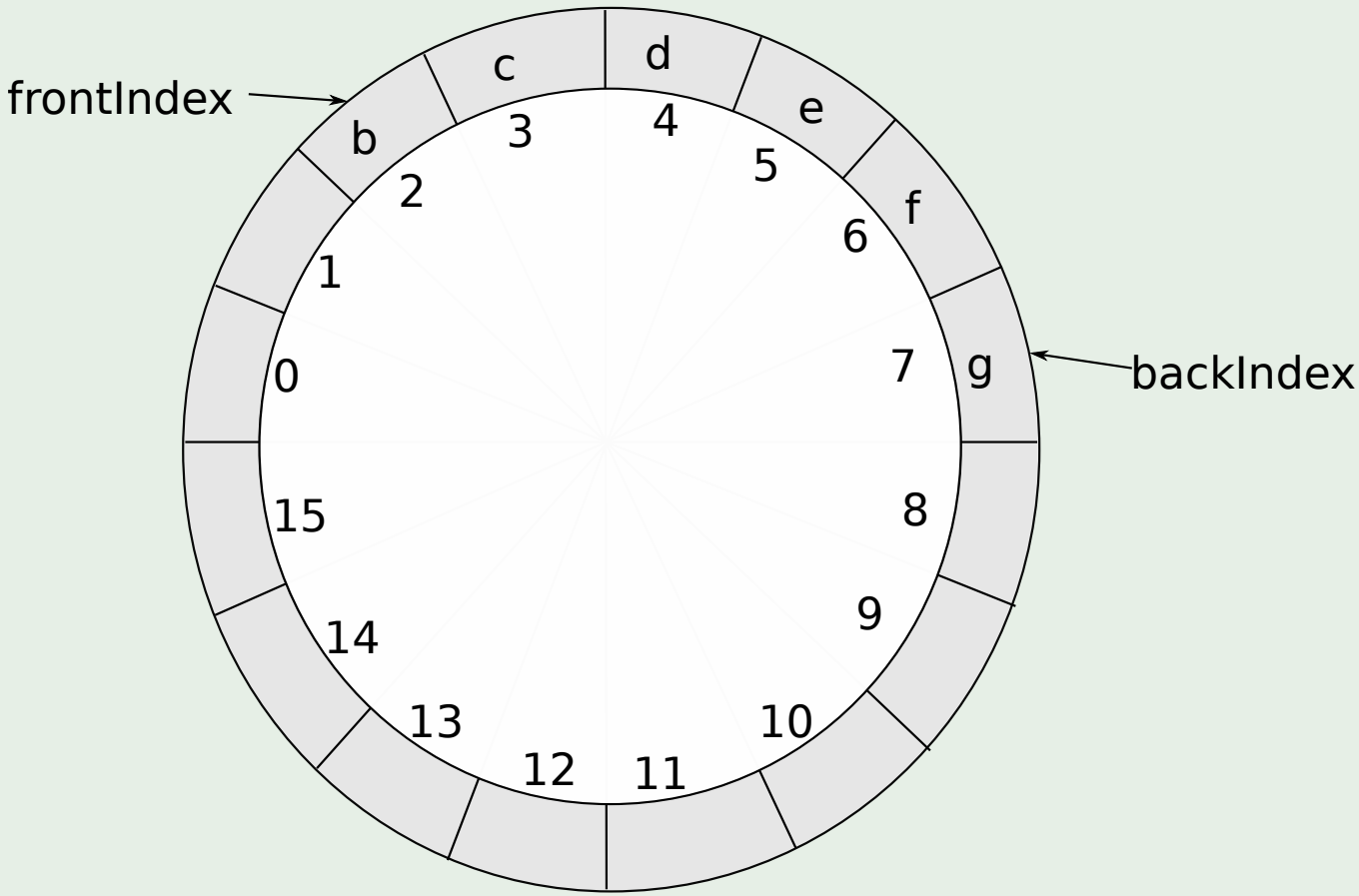
Example

Adding stuff into the back



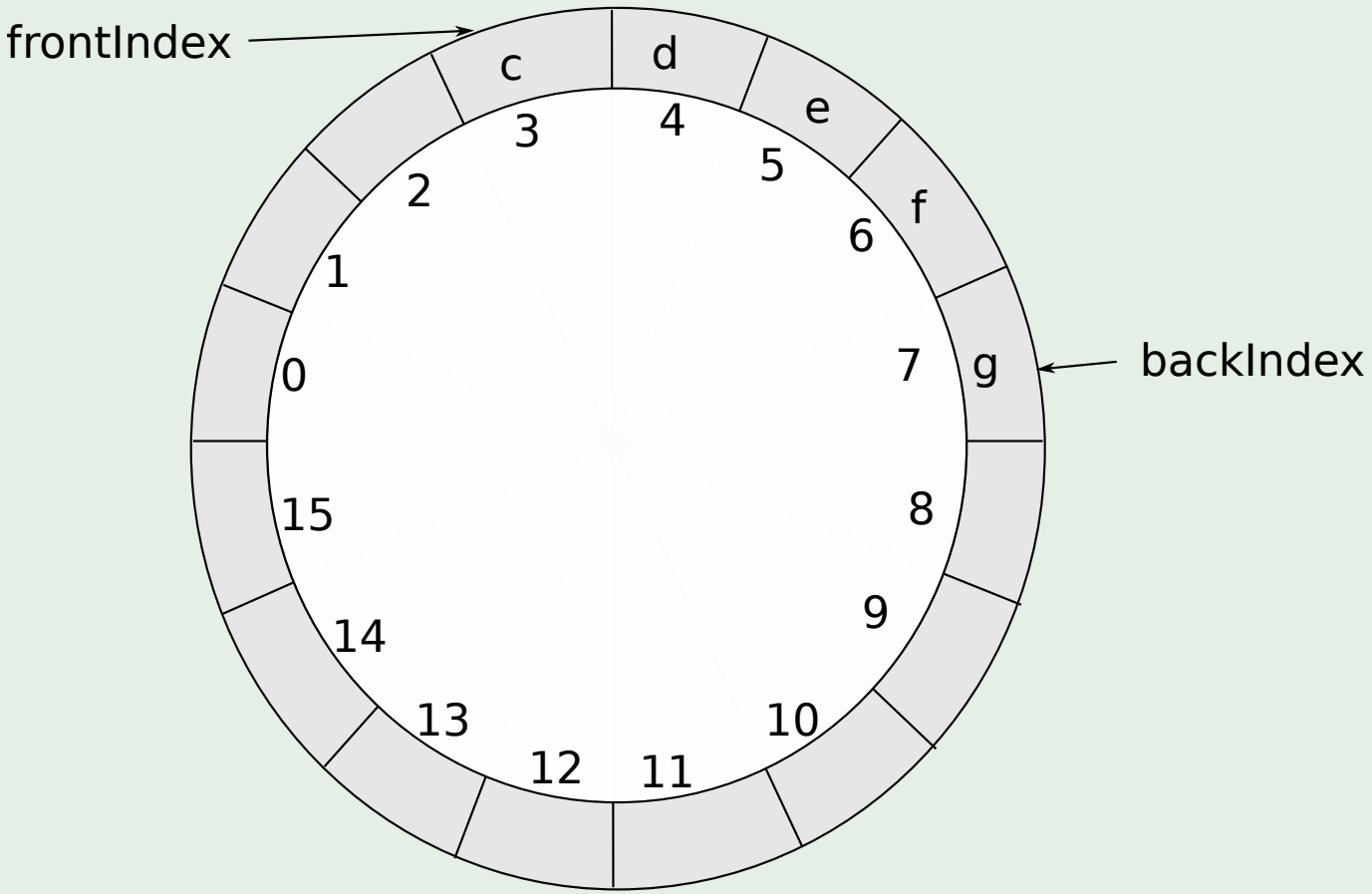
Example

Remove stuff from the front



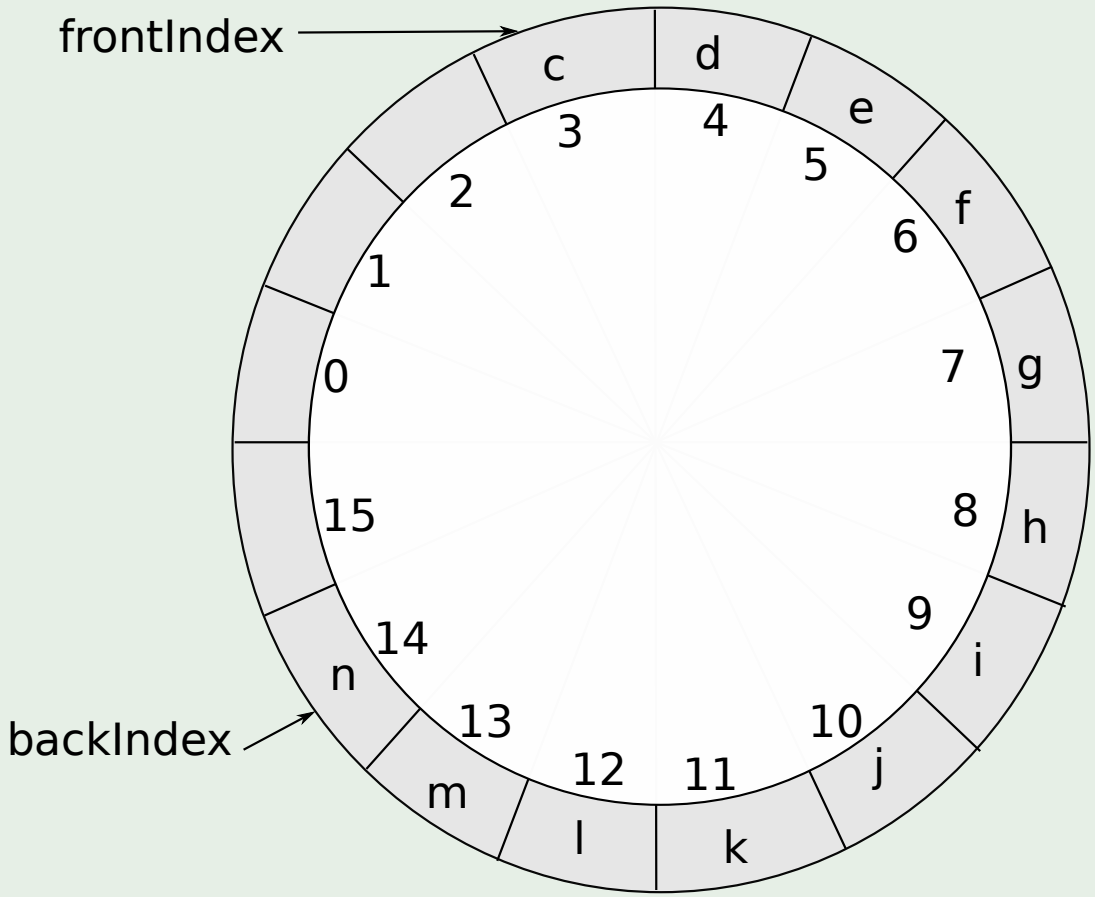
Example

Remove stuff from the front



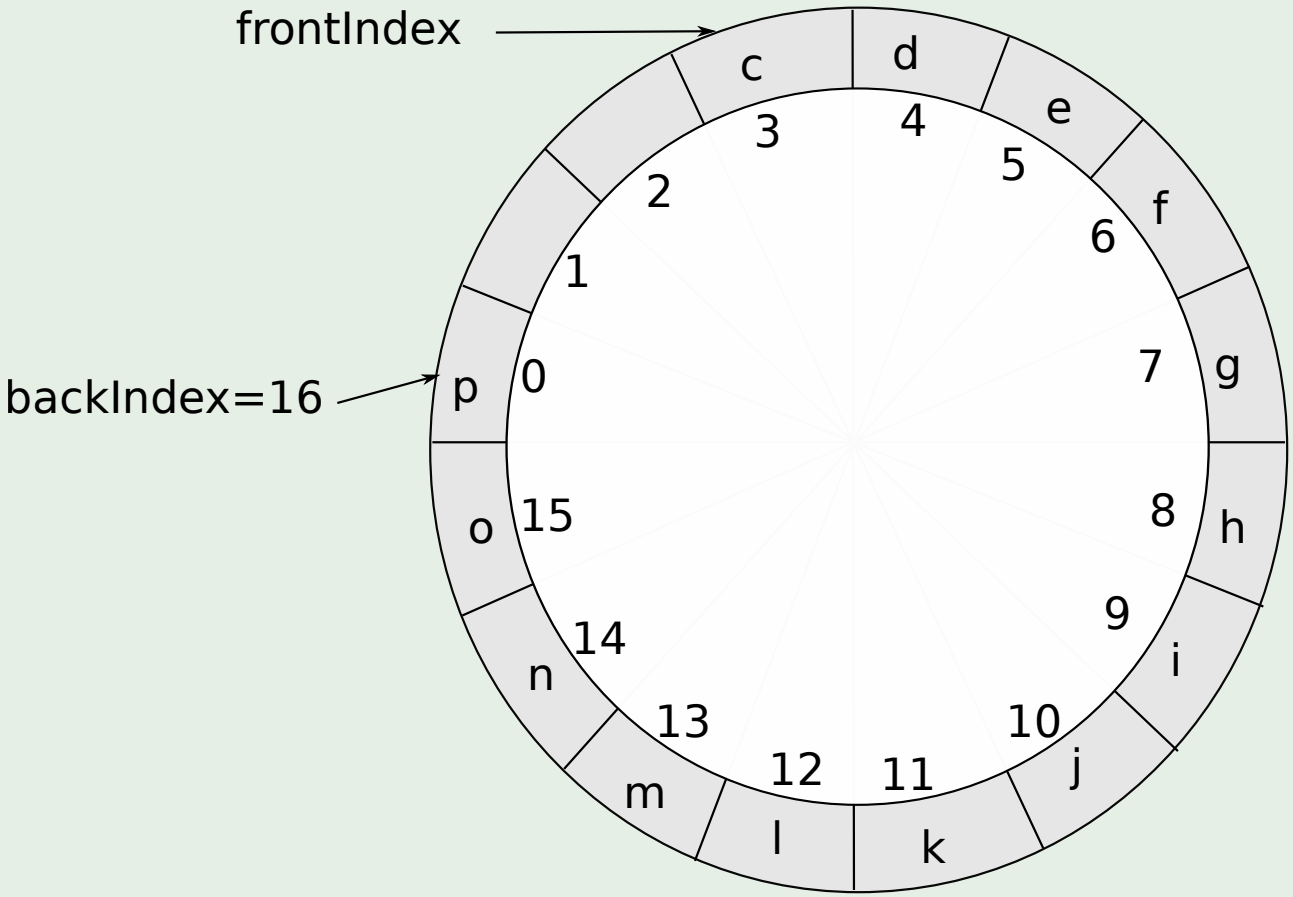
Example

Keep Adding



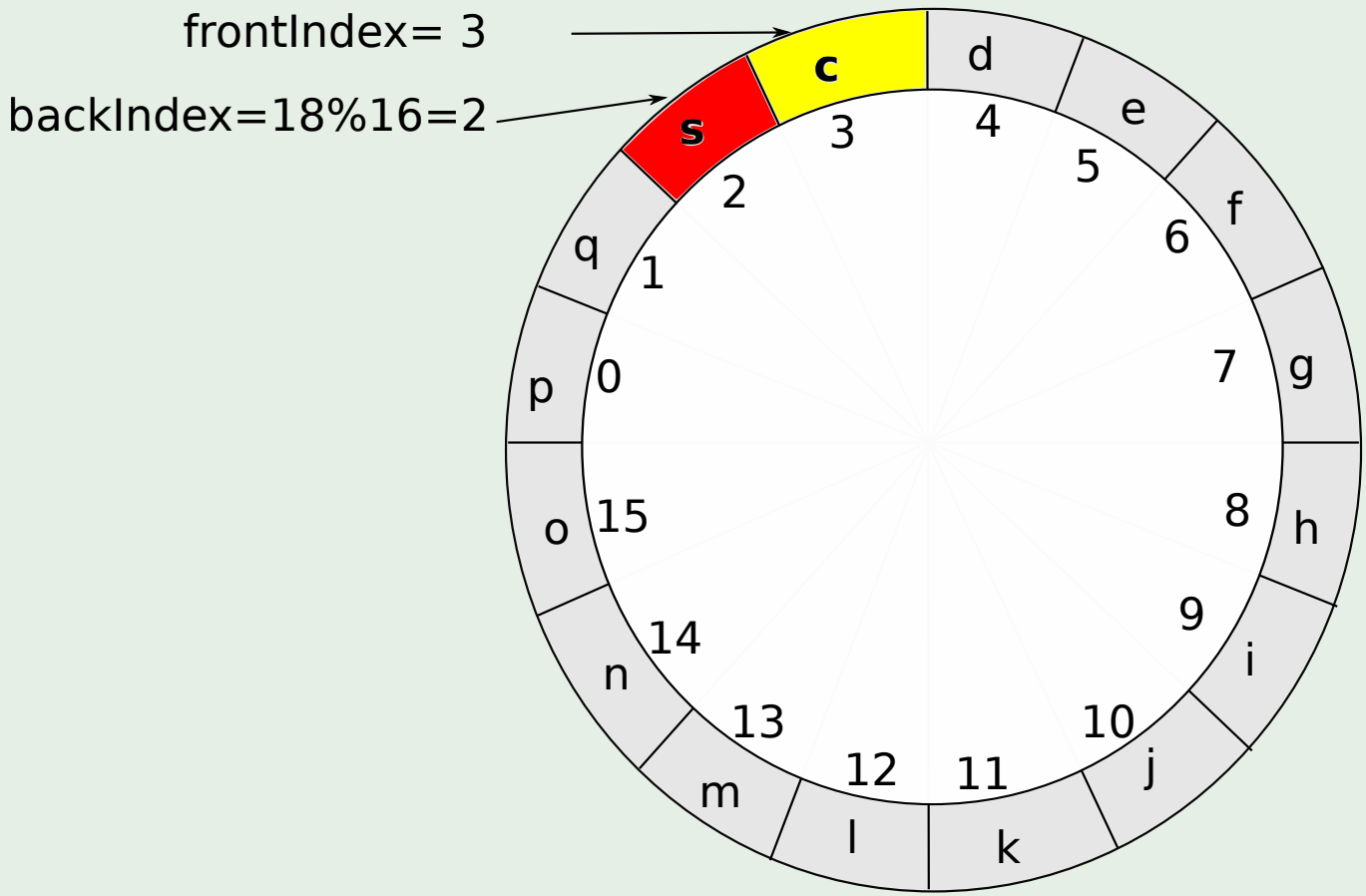
Example

The modulo helps here



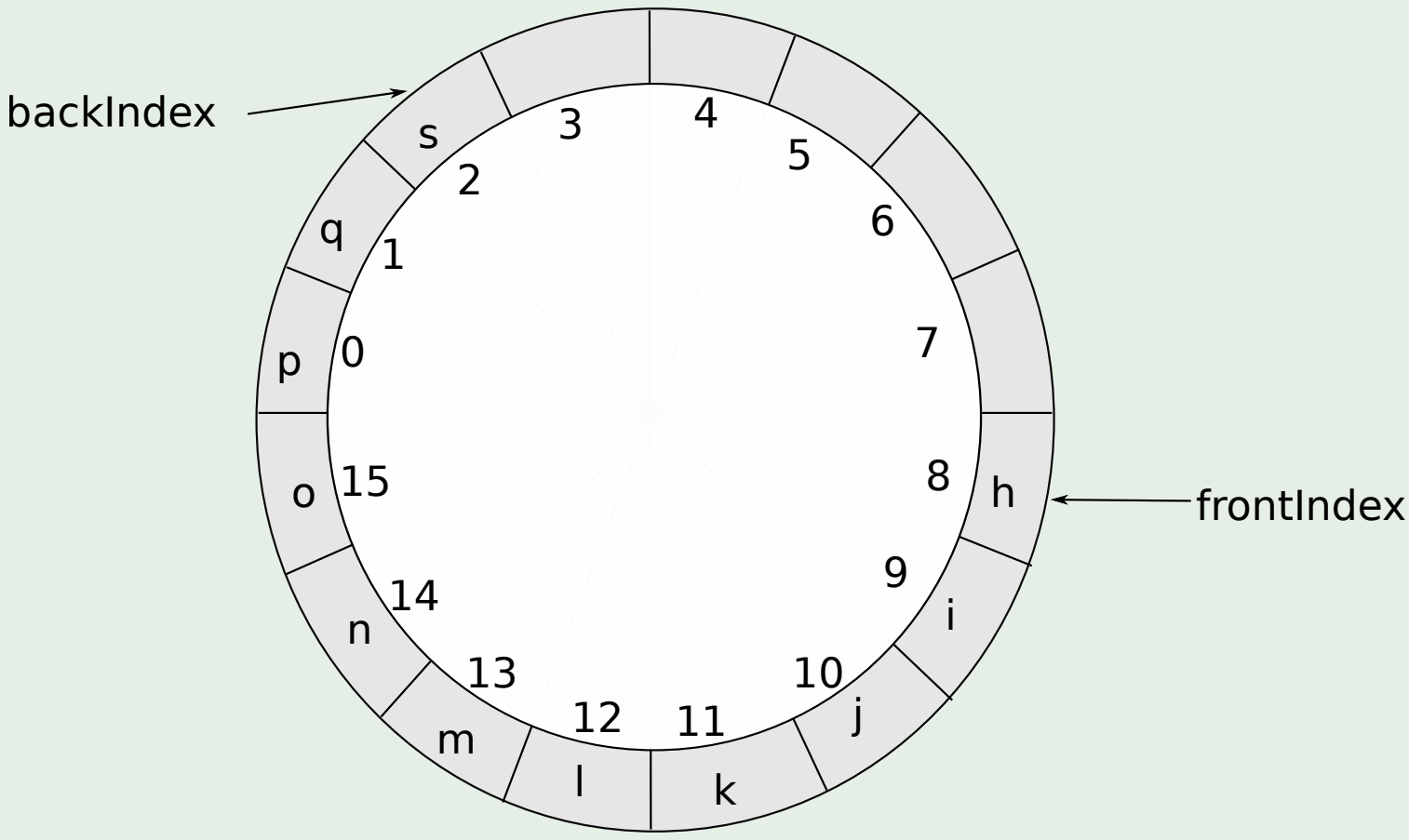
It looks Cool, but...

Problem 1 when full



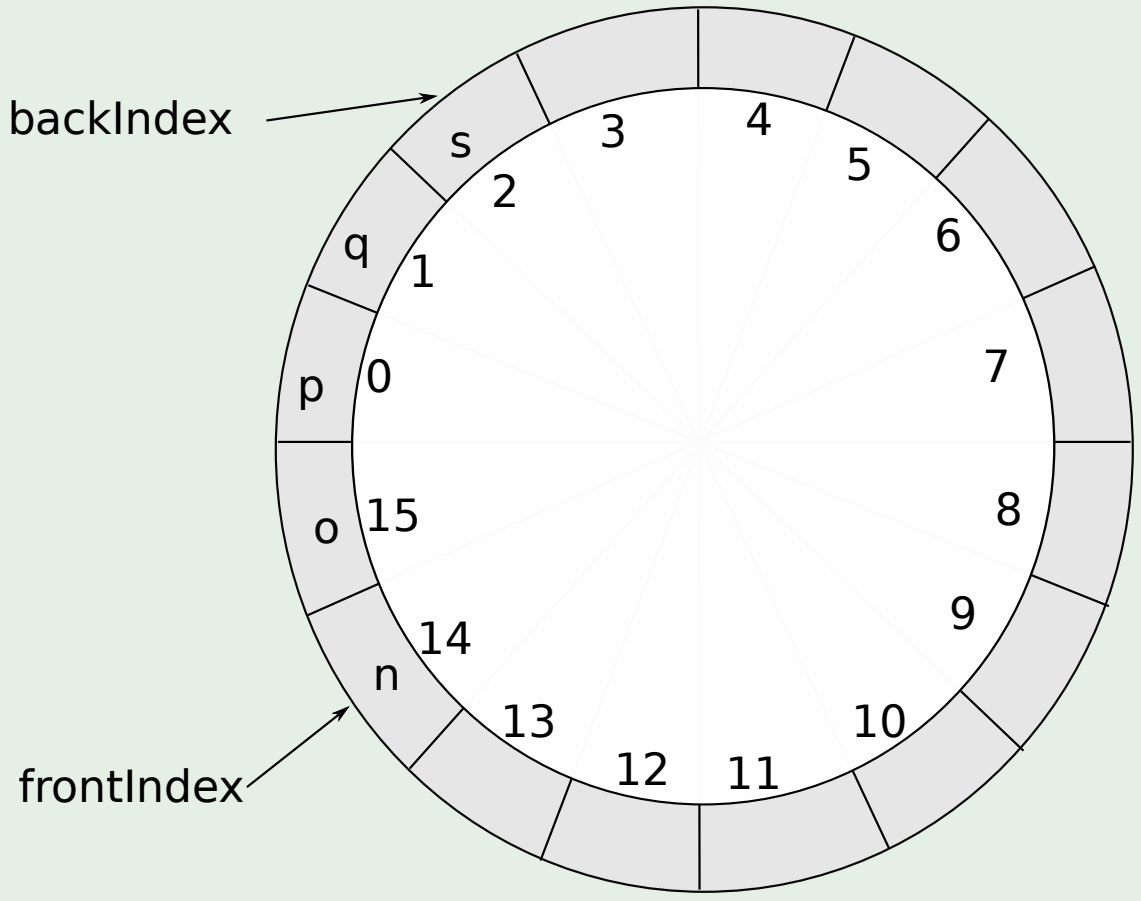
But when we remove all elements

We go...



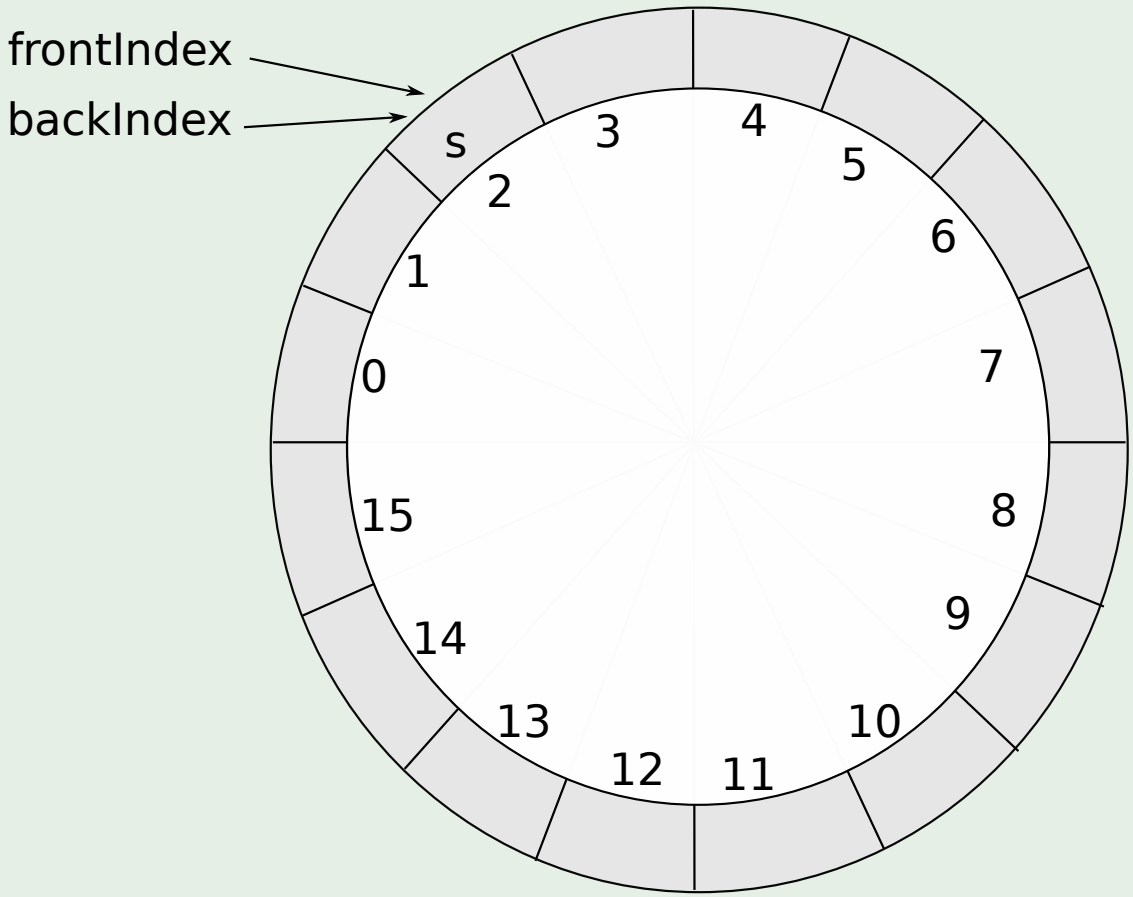
But when we remove all elements

We go...



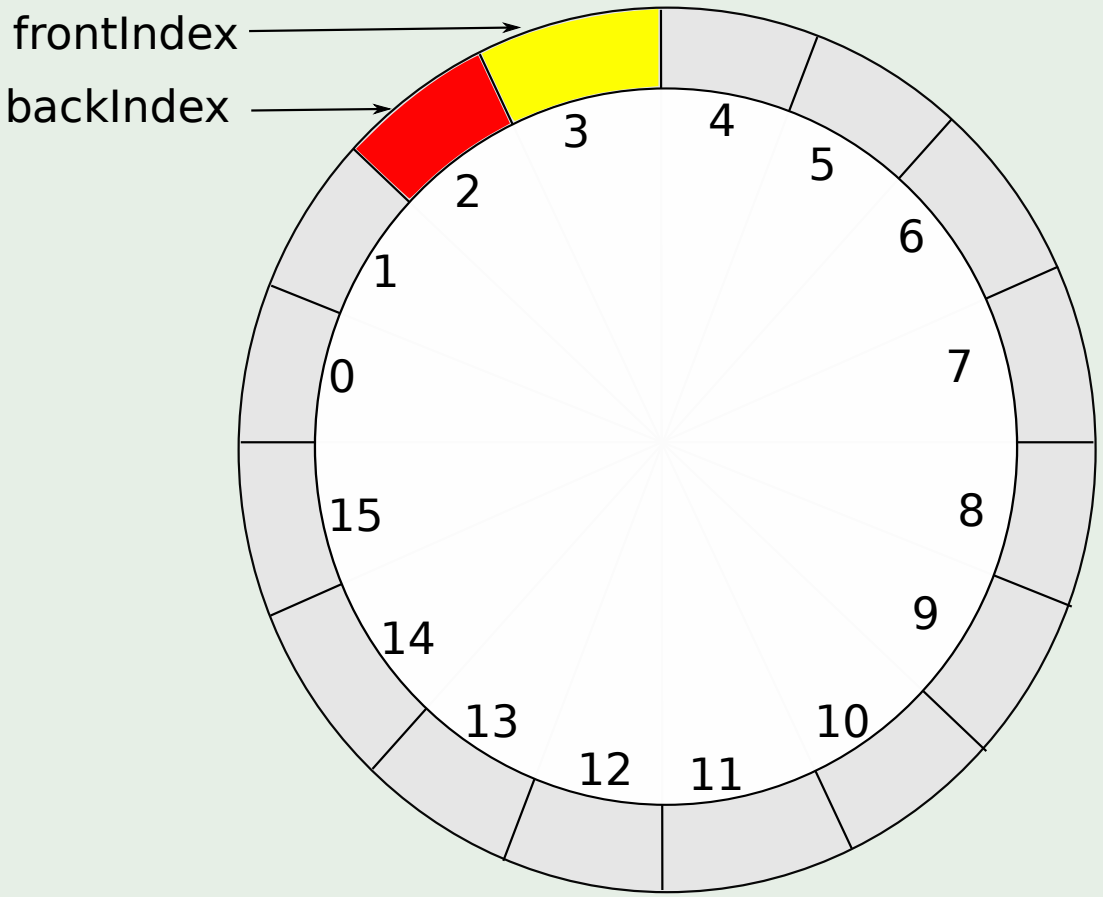
This is the main problem

And here is the problem when removing s



This is the main problem

Then full an empty cases cannot be identified!!!



Thus

The Problem

frontIndex == (backIndex + 1) % queue.length

A possible solution

Somewhat simple

- Use an extra field "**size**"

Then each time

- If frontIndex === (backIndex + 1) % queue.length ⇒ check size
  - Then do something what?

A possible solution

Somewhat simple

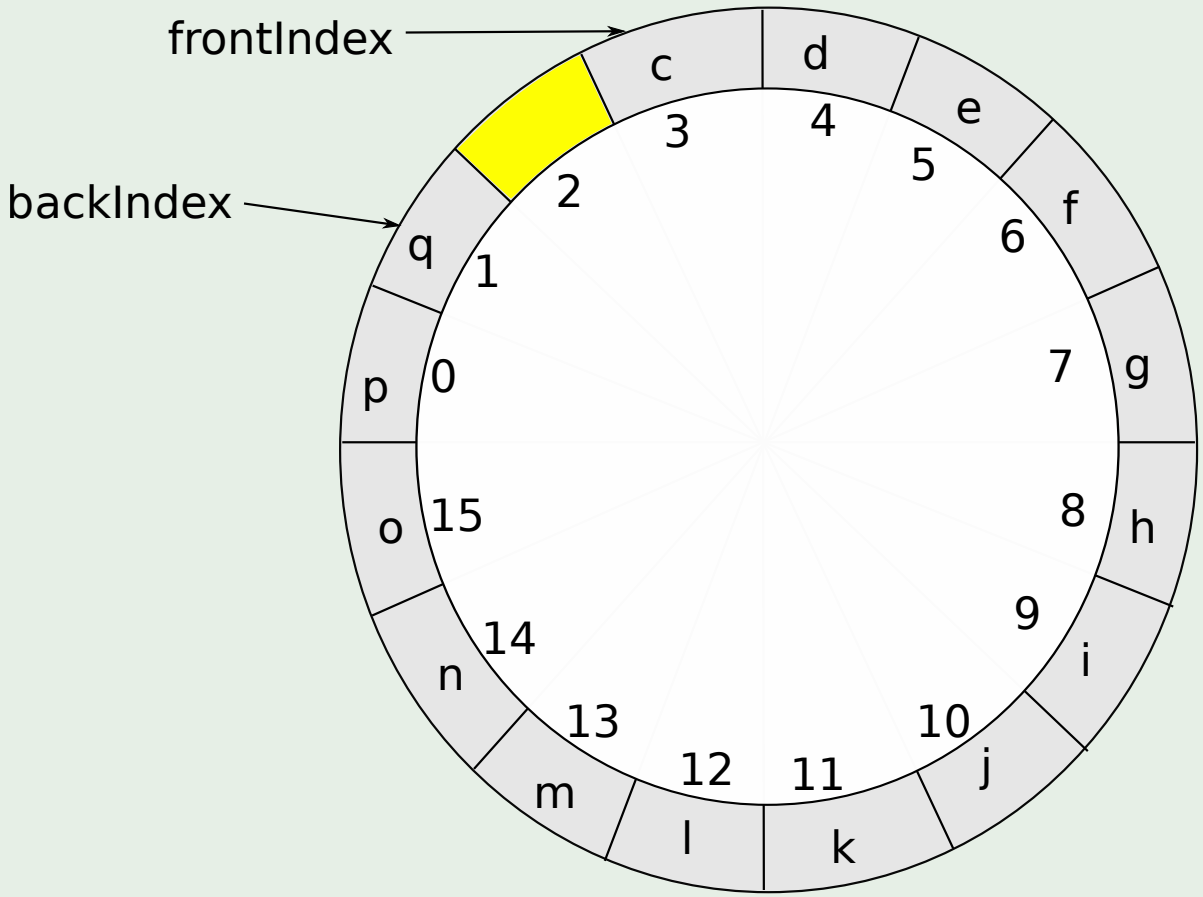
- Use an extra field "**size**"

Then each time

- If frontIndex == (backIndex + 1) % queue.length ⇒ check **size**
  - Then do something what?

Another solution!!!

Assume an space between the frontIndex and backIndex



Thus,

When the queue is full

frontIndex == (backIndex + 2) % queue.length

When the queue is empty

frontIndex === (backIndex + 1) % queue.length

Thus,

When the queue is full

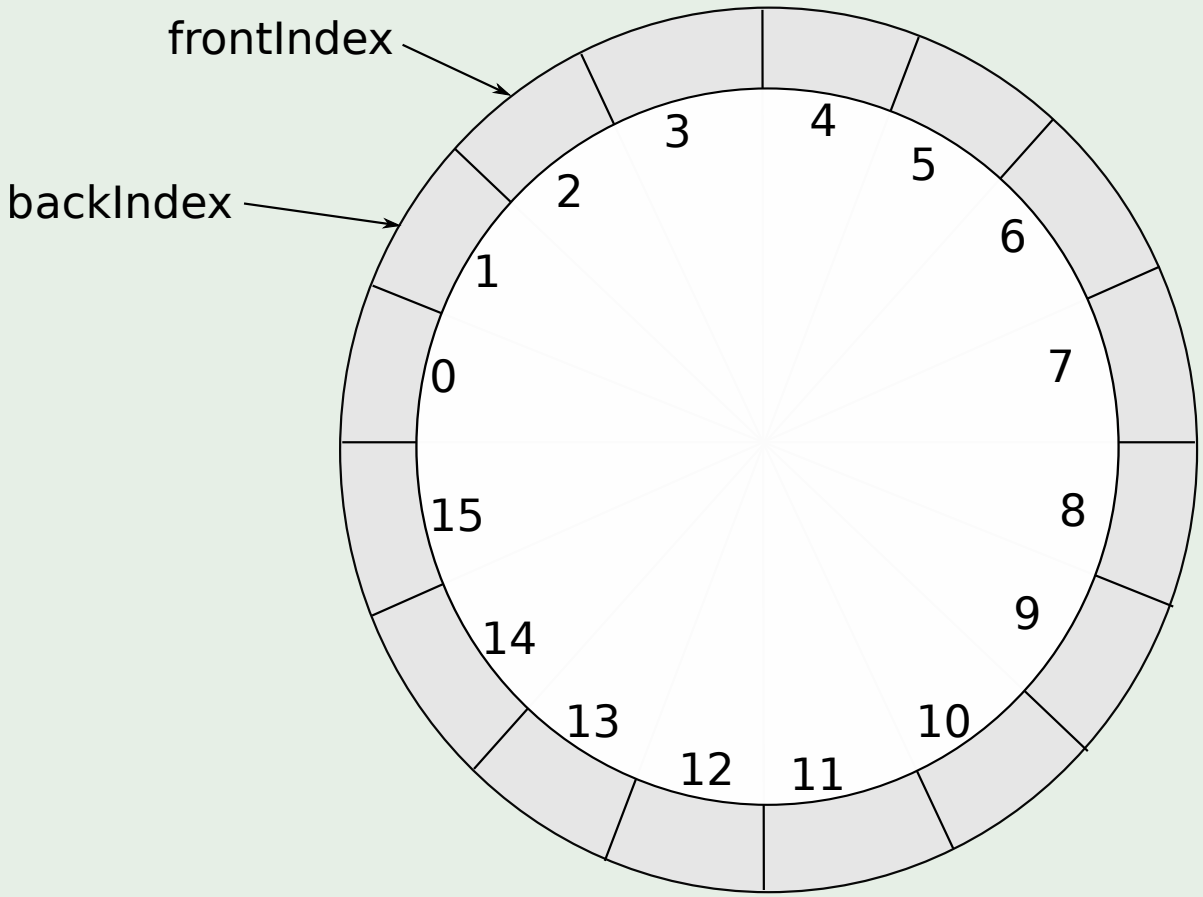
frontIndex == (backIndex + 2) % queue.length

When the queue is empty

frontIndex == (backIndex + 1) % queue.length

A solution!!!

Then for an empty queue



Class Members

Code

```
public class ArrayQueue<Item> implements QueueInterface<Item>
{
    private Item[] queue; // circular array of queue entries
                          // and one unused location
    private int frontIndex;
    private int backIndex;
    private static final int DEFAULT_INITIAL_CAPACITY = 50;
}
```

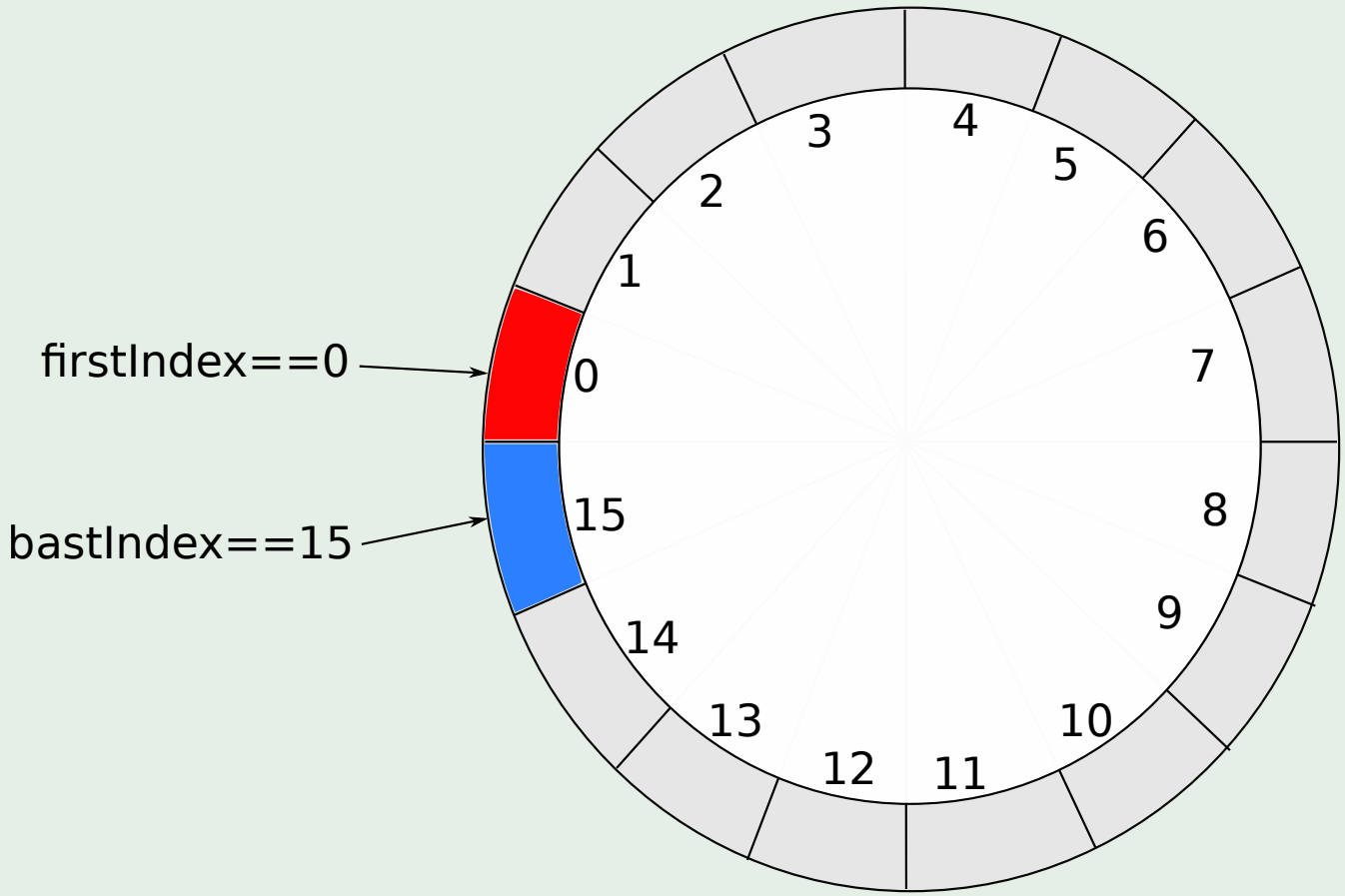
Basic Constructors

Code

```
public ArrayQueue()
{
  this(DEFAULT_INITIAL_CAPACITY);
} // end default constructor

public ArrayQueue(int initialCapacity)
{
// the cast is safe because the new
// array contains null entries
// The 1 is for empty entry
Item[] tempQueue = (Item[]) new Object[initialCapacity + 1];
queue = tempQueue;

frontIndex = 0;
backIndex = initialCapacity;

} // end constructor
```

At the Beginning

At the Instantiation of the Object Queue



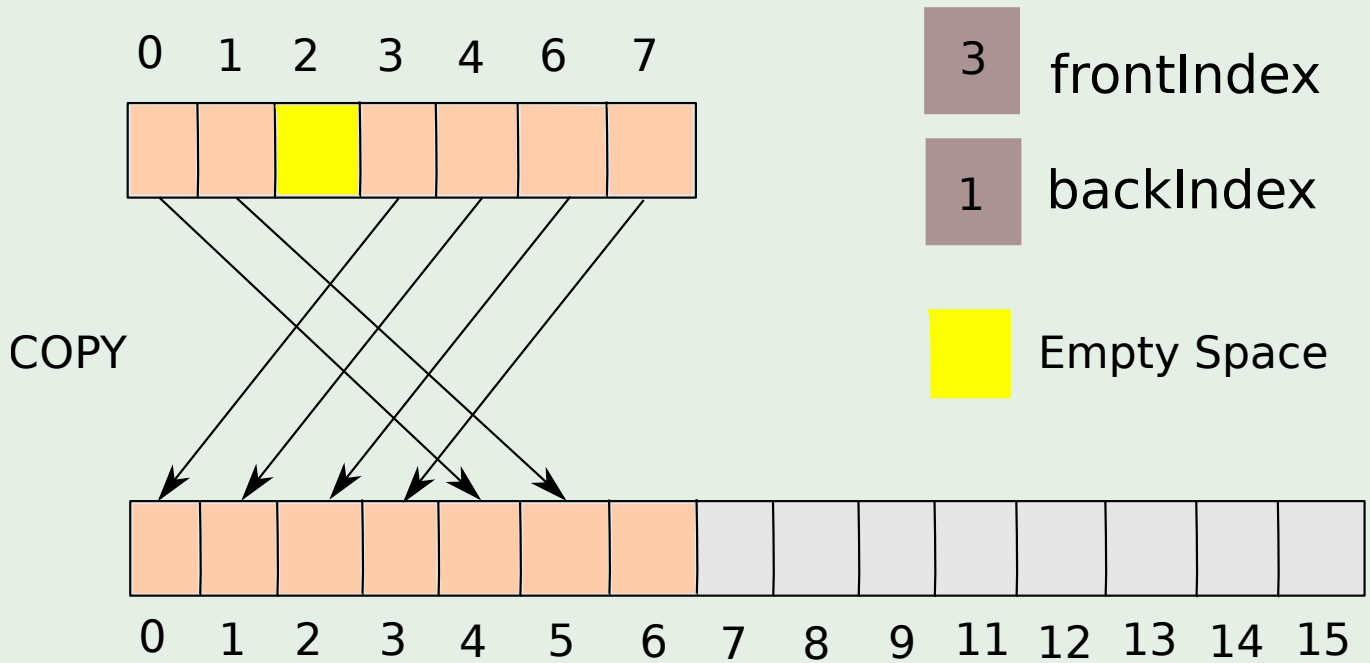
Enqueue in a circular array

Code

```
public void enqueue(Item newEntry)
{
    if (frontIndex == ((backIndex + 2) % queue.length))
    {

     <Something Here....>

    }
    backIndex = (backIndex + 1) % queue.length;
    queue[backIndex] = newEntry;

} // end enqueue
```

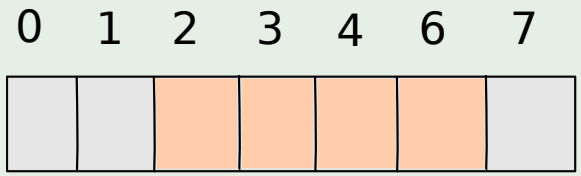
What is this "Something Here"?

Expanding Array and Copying values



Dequeue

At the beginning

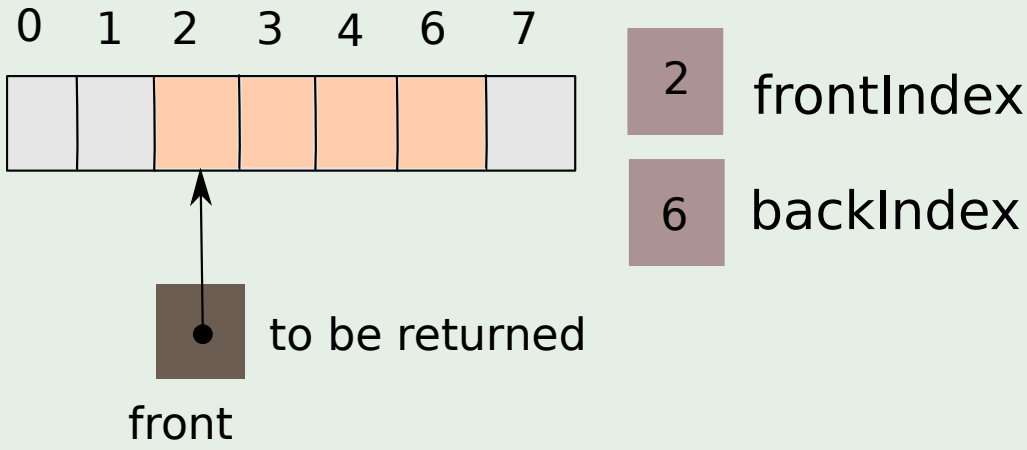


2 frontIndex

6 backIndex

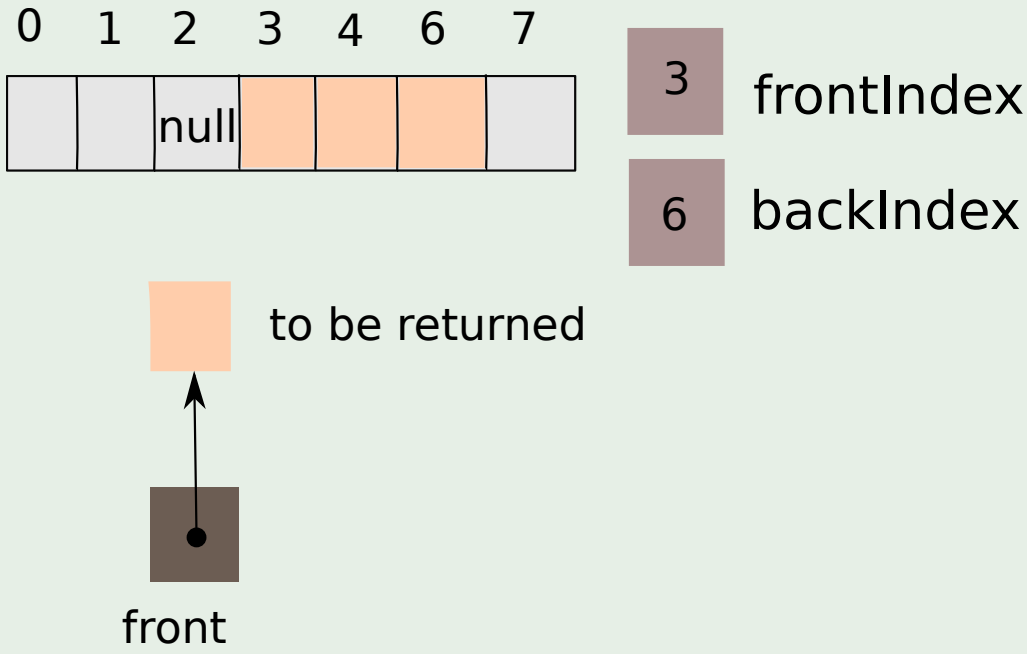
Dequeue

Use a temporary pointer "front"



Dequeue

Remove the element



Enqueue in a circular array

Code

```
public Item dequeue()
{
    Item front = null;
    if (!empty())
    {
        front = queue[frontIndex];
        queue[frontIndex] = null;
        frontIndex = (frontIndex + 1) % queue.length;
    } // End If
    return front;
} // end dequeue
```

Thus...

The Rest of Operations

You can think about them... they are not so complex...

Complexities

Operation in Scratch Queue using Chains	Complexity
empty()	O(1)
front()	O(1)
rear()	O(1)
Enqueue(TheObject)	O(1)
Dequeue()	O(1)

Thus...

The Rest of Operations

You can think about them... they are not so complex...

Complexities

Operation in Scratch Queue using Chains	Complexity
empty()	O(1)
front()	O(1)
rear()	O(1)
Enqueue(TheObject)	O(1)
Dequeue()	O(1)