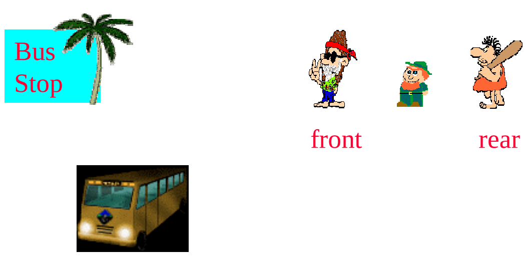
Queues

- Linear list.
- One end is called front.
- Other end is called rear.
- Additions are done at the rear only.
- Removals are made from the front only.

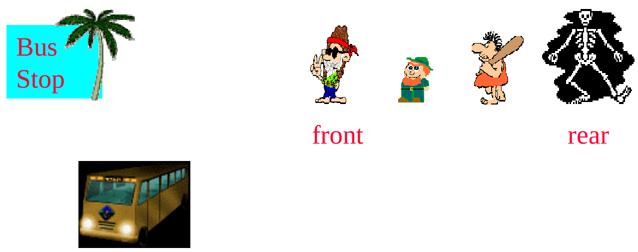
Bus Stop Queue

Bus Stop Queue

Bus Stop Queue



Bus Stop Queue



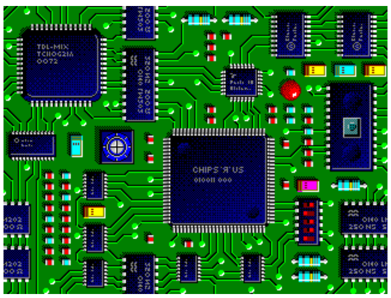
The Interface Queue

```
public interface Queue
{
    public boolean isEmpty();
    public Object getFrontEelement();
    public Object getRearEelement();
    public void put(Object theObject);
    public Object remove();
}
```

Revisit Of Stack Applications

- Applications in which the stack cannot be replaced with a queue.
  - Parentheses matching.
  - Towers of Hanoi.
  - Switchbox routing.
  - Method invocation and return.
  - Try-catch-throw implementation.
- Application in which the stack may be replaced with a queue.
  - Rat in a maze.
    - Results in finding shortest path to exit.

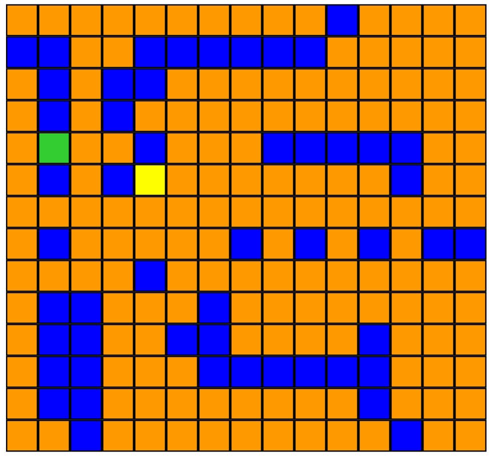
Wire Routing



Lee's Wire Router

start pin

end pin

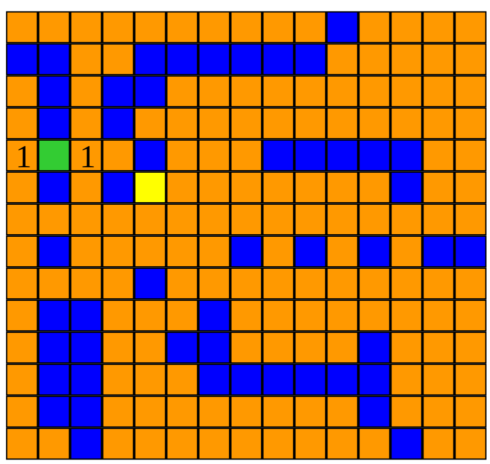


Label all reachable squares 1 unit from start.

Lee's Wire Router

start pin

end pin

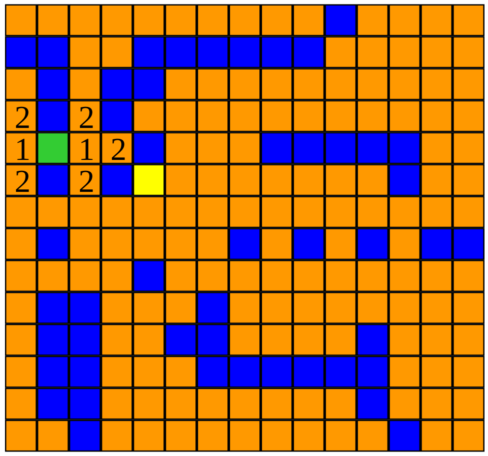


Label all reachable unlabeled squares 2 units from start.

Lee’s Wire Router

start pin

end pin

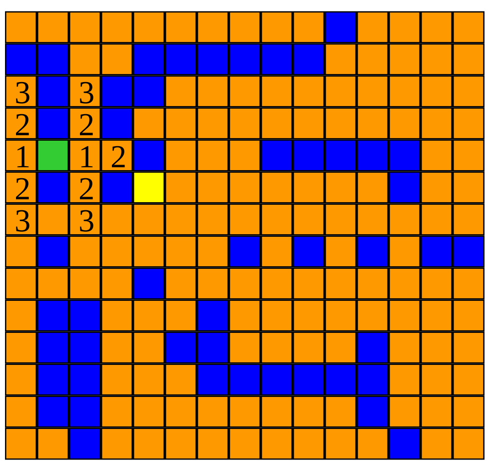


Label all reachable unlabeled squares 3 units from start.

Lee’s Wire Router

start pin

end pin

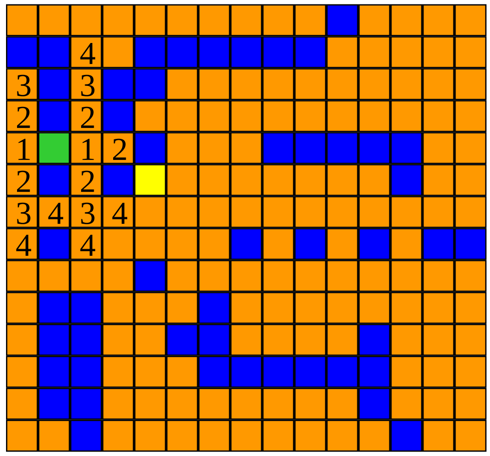


Label all reachable unlabeled squares 4 units from start.

Lee’s Wire Router

start pin

end pin

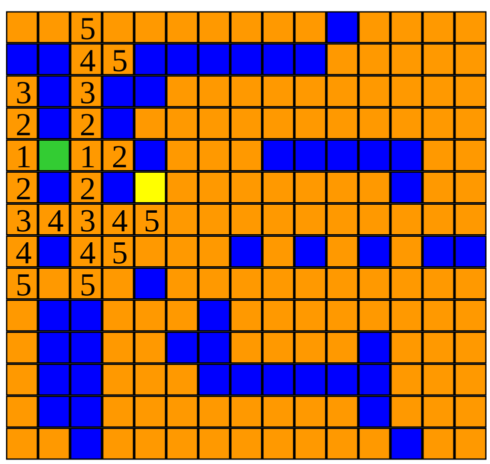


Label all reachable unlabeled squares 5 units from start.

Lee’s Wire Router

start pin

end pin

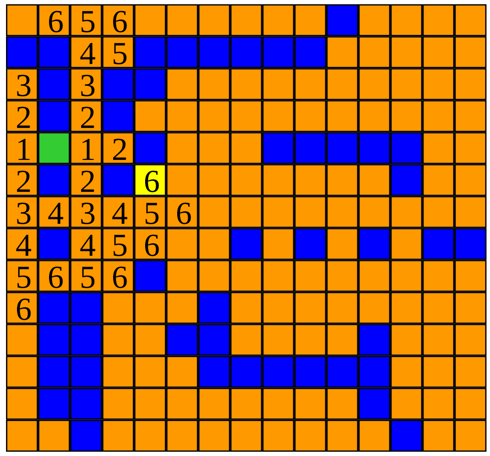


Label all reachable unlabeled squares 6 units from start.

Lee’s Wire Router

start pin

end pin

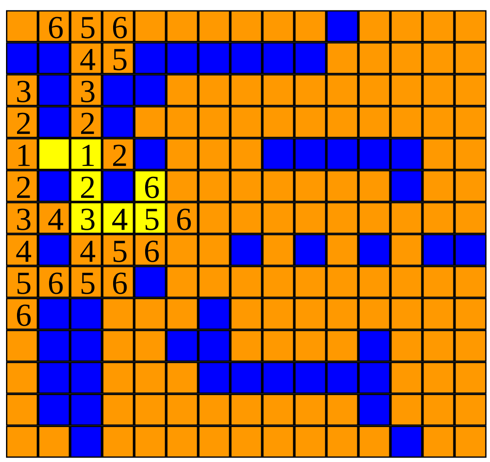


End pin reached. Traceback.

Lee’s Wire Router

start pin

end pin



End pin reached. Traceback.

Derive From ArrayLinearList

a	b	c	d	e										

0 1 2 3 4 5 6

- when front is left end of list and rear is right end
  - Queue.isEmpty() => super.isEmpty()
    - O(1) time
  - getFrontElement() => get(0)
    - O(1) time
  - getRearElement() => get(size() - 1)
    - O(1) time
  - put(theObject) => add(size(), theObject)
    - O(1) time
  - remove() => remove(0)
    - O(size) time

Derive From ArrayLinearList

e	d	c	b	a										

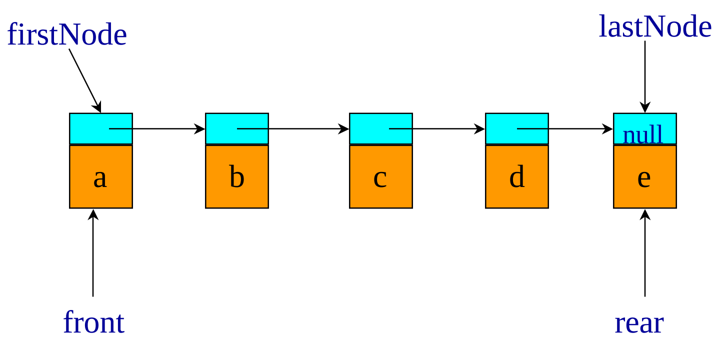
0 1 2 3 4 5 6

- when rear is left end of list and front is right end
  - Queue.isEmpty() => super.isEmpty()
    - O(1) time
  - getFrontElement() => get(size() - 1)
    - O(1) time
  - getRearElement() => get(0)
    - O(1) time
  - put(theObject) => add(0, theObject)
    - O(size) time
  - remove() => remove(size() - 1)
    - O(1) time

Derive From ArrayLinearList

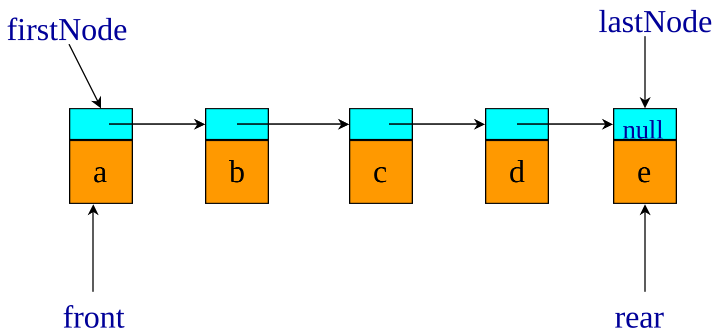
- to perform each opertion in O(1) time (excluding array doubling), we need a customized array representation.

Derive From ExtendedChain



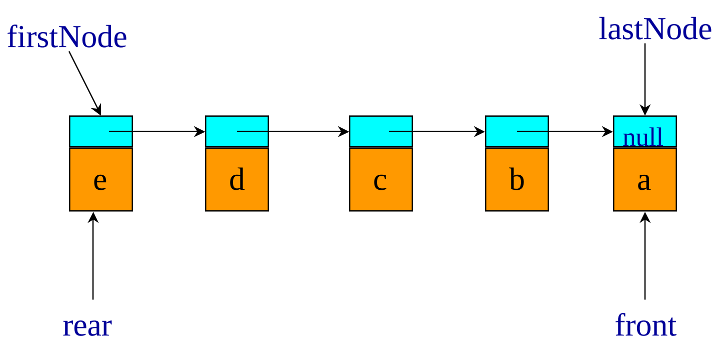
- when front is left end of list and rear is right end
  - Queue.isEmpty() => super.isEmpty()
    - O(1) time
  - getFrontElement() => get(0)
    - O(1) time

Derive From ExtendedChain



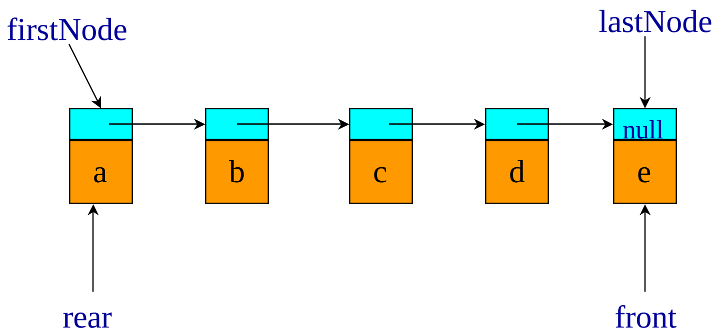
- getRearElement() => getLast() … new method
  - O(1) time
- put(theObject) => append(theObject)
  - O(1) time
- remove() => remove(0)
  - O(1) time

Derive From ExtendedChain



- when front is right end of list and rear is left end
  - Queue.isEmpty() => super.isEmpty()
    - O(1) time
  - getFrontElement() => getLast()
    - O(1) time

Derive From ExtendedChain



- getRearElement() => get(0)
  - O(1) time
- put(theObject) => add(0, theObject)
  - O(1) time
- remove() => remove(size-1)
  - O(size) time

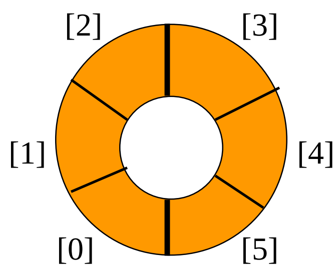
Custom Linked Code

- Develop a linked class for Queue from scratch to get better preformance than obtainable by deriving from ExtendedChain.

Custom Array Queue

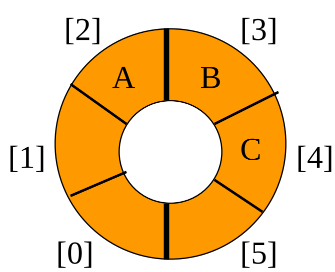
- Use a 1D array queue.

- Circular view of array.



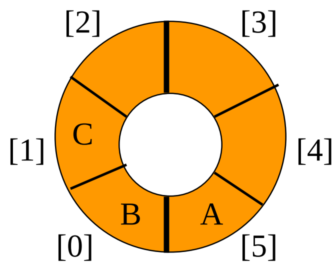
Custom Array Queue

- Possible configuration with 3 elements.



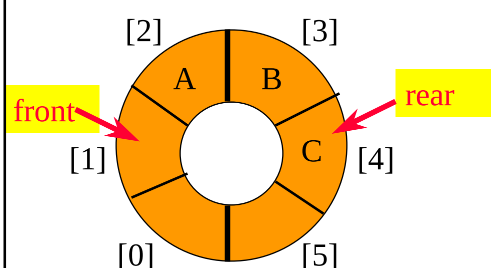
Custom Array Queue

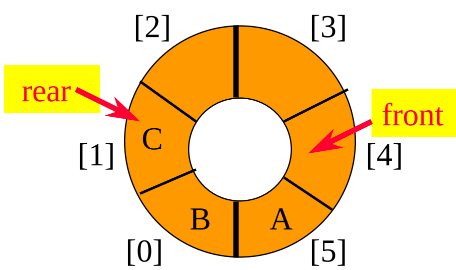
- Another possible configuration with 3 elements.



Custom Array Queue

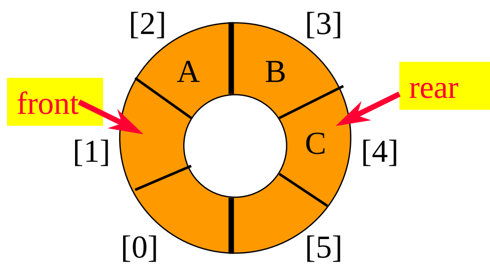
- Use integer variables front and rear.
  - front is one position counterclockwise from first element
  - rear gives position of last element





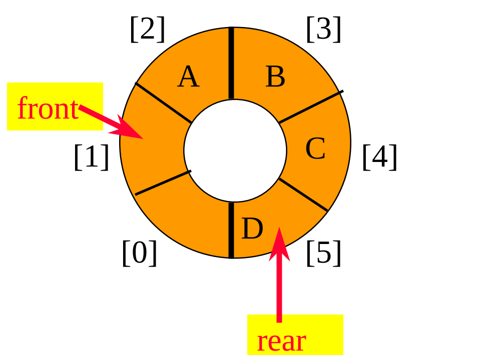
Add An Element

- Move rear one clockwise.



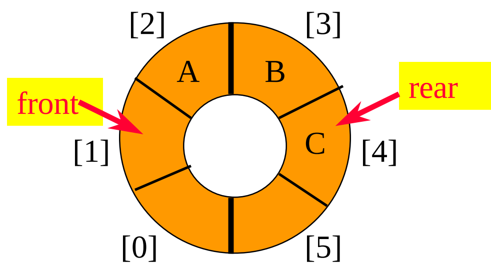
Add An Element

- Move rear one clockwise.
- Then put into queue[rear].



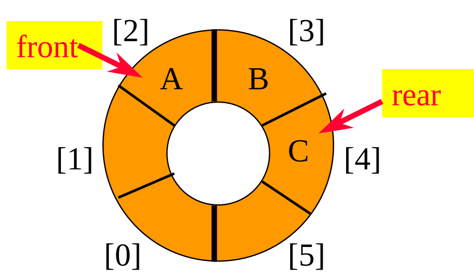
Remove An Element

- Move front one clockwise.



Remove An Element

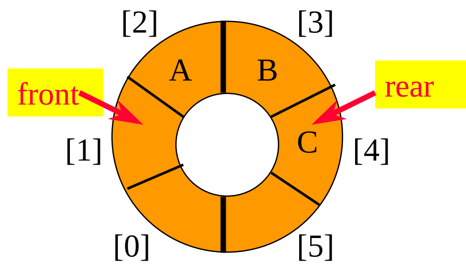
- Move front one clockwise.
- Then extract from queue[front].



Moving rear Clockwise

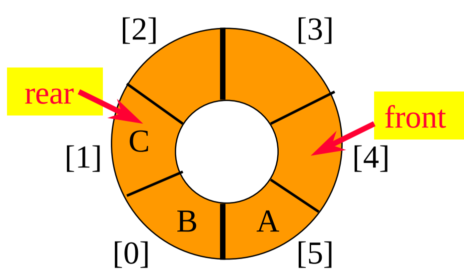
- rear++;

  if (rear = = queue.length) rear = 0;

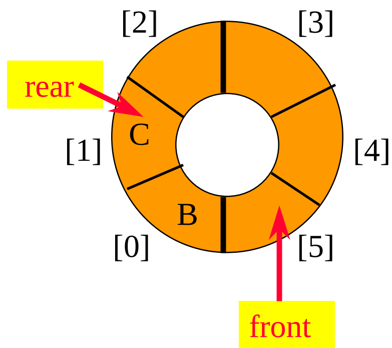


- rear = (rear + 1) % queue.length;

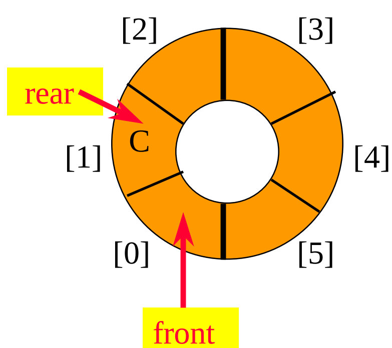
Empty That Queue



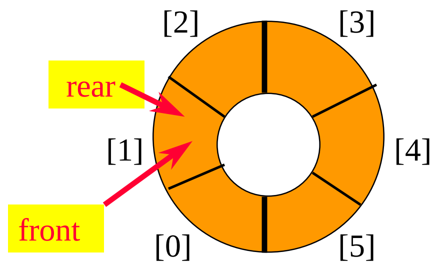
Empty That Queue



Empty That Queue

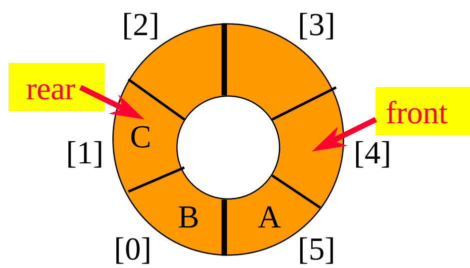


Empty That Queue

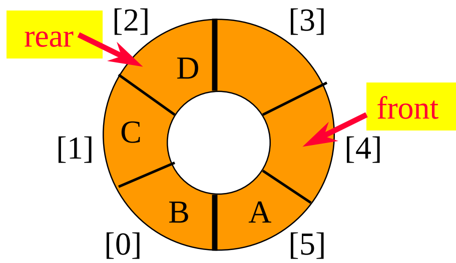


- When a series of removes causes the queue to become empty, front = rear.
- When a queue is constructed, it is empty.
- So initialize front = rear = 0.

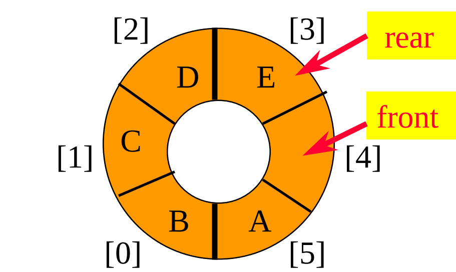
A Full Tank Please



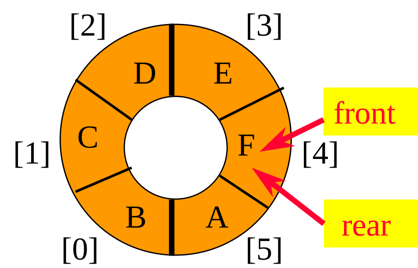
A Full Tank Please



A Full Tank Please



A Full Tank Please



- When a series of adds causes the queue to become full, front = rear.
- So we cannot distinguish between a full queue and an empty queue!

Ouch!!!!!

- Remedies.
  - Don’t let the queue get full.
    - When the addition of an element will cause the queue to be full, increase array size.
    - This is what the text does.
  - Define a boolean variable lastOperationIsPut.
    - Following each put set this variable to true.
    - Following each remove set to false.
    - Queue is empty iff (front == rear) && !lastOperationIsPut
    - Queue is full iff (front == rear) && lastOperationIsPut

Ouch!!!!!

- Remedies (continued).
  - Define an integer variable size.
    - Following each put do size++.
    - Following each remove do size--.
    - Queue is empty iff (size == 0)
    - Queue is full iff (size == queue.length)
  - Performance is slightly better when first strategy is used.