COMP20010

Data Structures and Algorithms I
10 - Queues

Dr. Aonghus Lawlor
aonghus.lawlor@ucd.ie

Queues

Queue

A queue is a waiting line:

- grows by adding elements to its end
- shrinks by taking elements from its front.

Unlike a stack, a queue is a structure in which both ends are used:

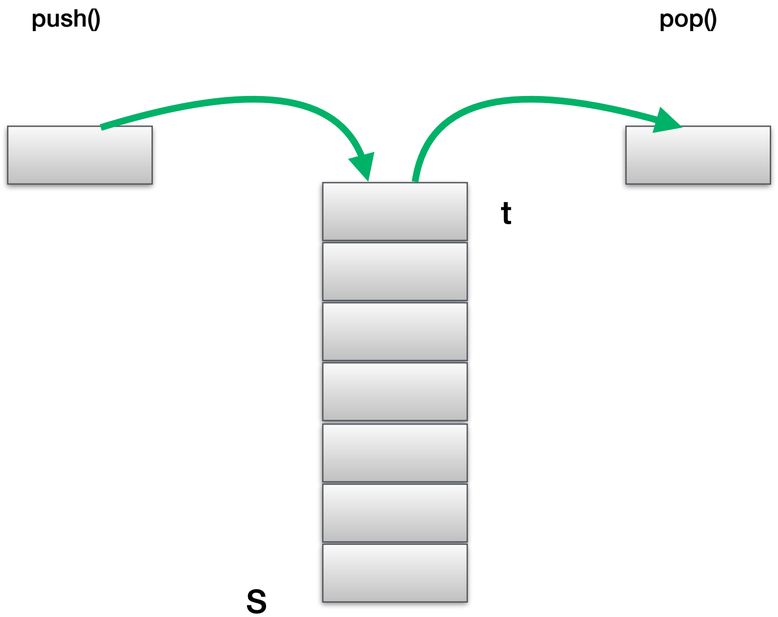
- one for adding new elements
- one for removing them

the last element has to wait until all elements preceding it on the queue are removed.

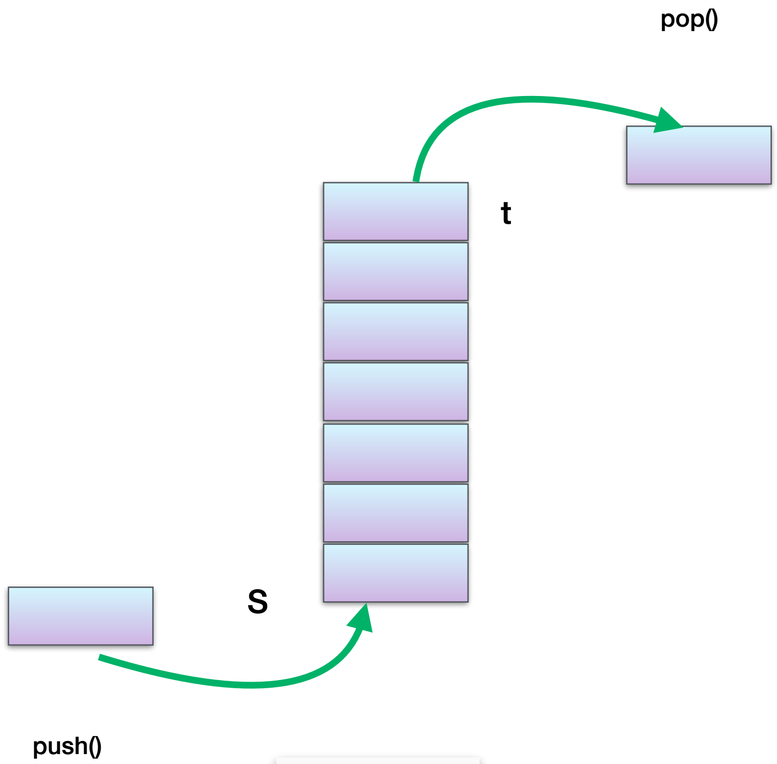
Queues

- Queue is a data structure where we add elements at the back and remove elements from the front.
- “waiting in line”: the first one to be added to the queue will be the first one to be removed
- FIFO (First In First Out) data structure.
- Applications:
  - Round robin
  - scheduling
  - buffers

Queues



Stack
LIFO

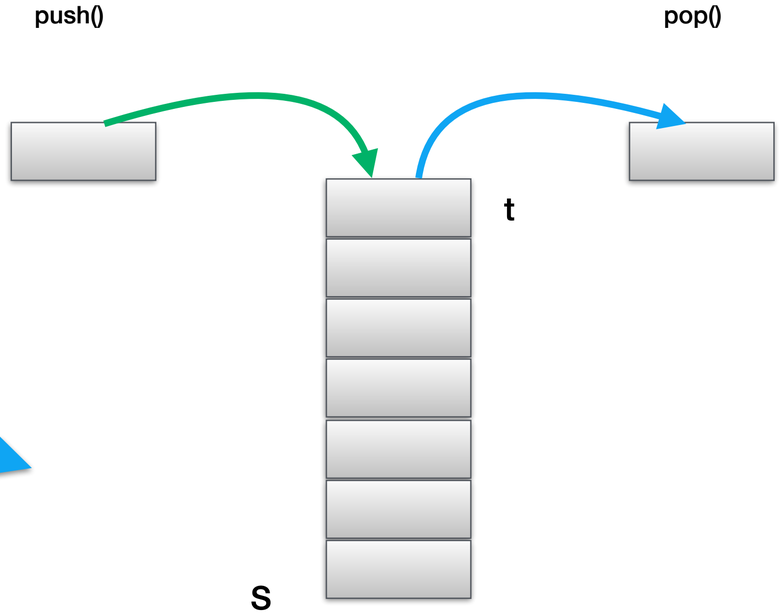


Queue
FIFO

Queues

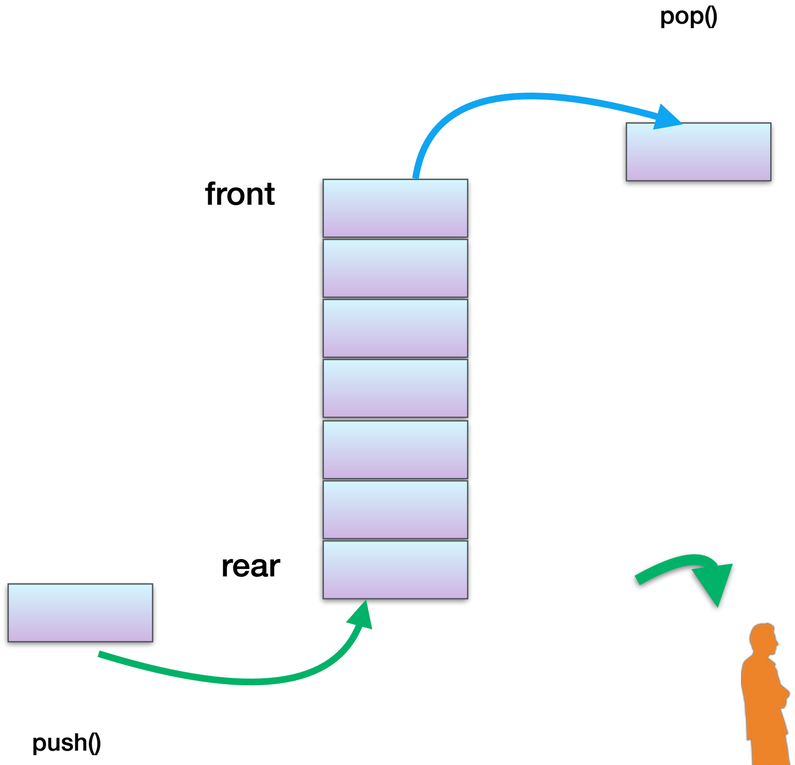
Stack

LIFO



Queue

FIFO



Queues

- ***queue*** is a close relative of the stack.
- a *queue* is a container of elements that are inserted and removed according to the ***first-in first-out*** (***FIFO***) principle.
- Elements can be inserted in a queue at any time, but only the element that has been in the queue the longest can be removed at any time.
- elements enter the queue at the ***rear*** and are removed from the ***front***.

Abstract Data Type

Stack ADT

push(e)	insert element e at top of stack
pop()	remove the top element from the stack
top()	return a reference to the top element of the stack without removing it
size()	number of elements in the stack
empty()	true if the stack is empty, false otherwise

Abstract Data Type

Queue ADT

enqueue(e)	insert element e at rear of the queue
dequeue()	remove the element from the front of the queue
front()	return a reference to the front element of the queue without removing it
size()	number of elements in the stack
empty()	true if the stack is empty, false otherwise

Queues - ADT

- the queue abstract data type defines a container that keeps elements in a sequence
- element access and deletion are restricted to the first element in the sequence, which is called the ***front*** of the queue
- element insertion is restricted to the end of the sequence, which is called the ***rear*** of the queue.

Queue operations

- insert at one end (rear)
- delete from the other (front)

operation	size
push(A)	0
push(B)	1
push(C)	2
pop()	1
push(D)	2
push(E)	3
push(F)	4

0	1	2	3	4	5
A					
A	B				
A	B	C			
B	C				
B	C	D			
B	C	D	E		
B	C	D	E	F	

Queue Applications

Web Crawler URL Queue

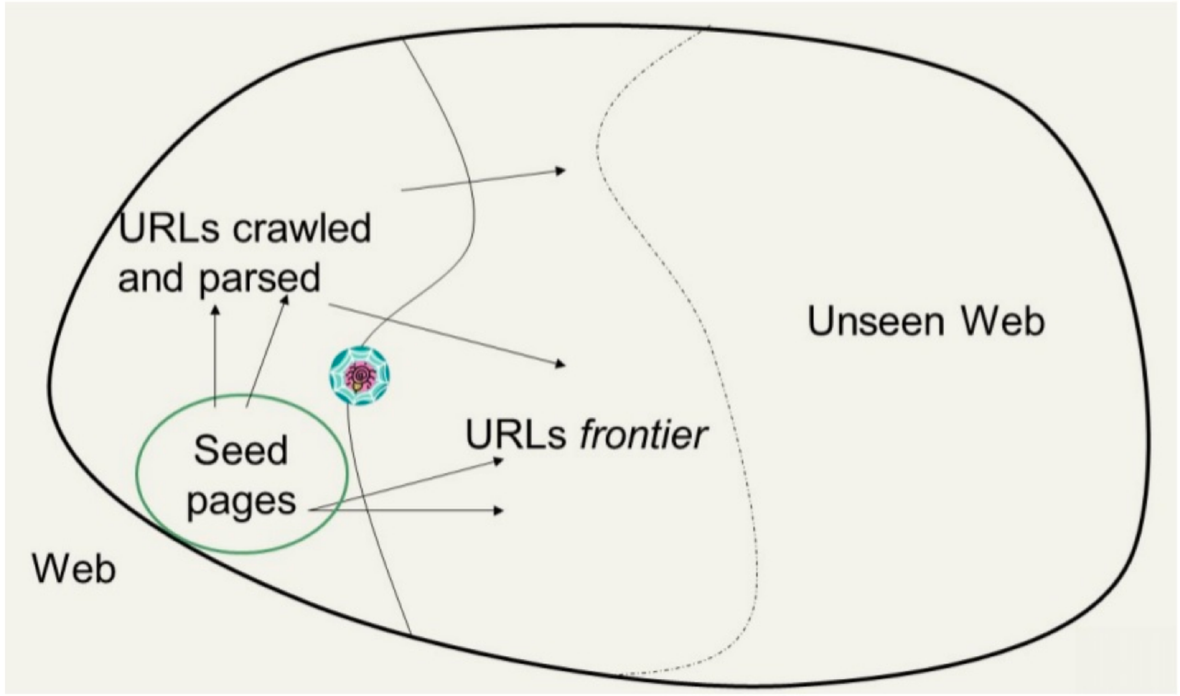
start with a list of seed url's which we push to the URL Queue

crawlers pop from the queue and retrieve the pages

the retrieved pages are extracted, the data is saved to db

new links which are found are added to the URL Queue

the process continues until the URL Queue is empty



Web Crawler URL Queue

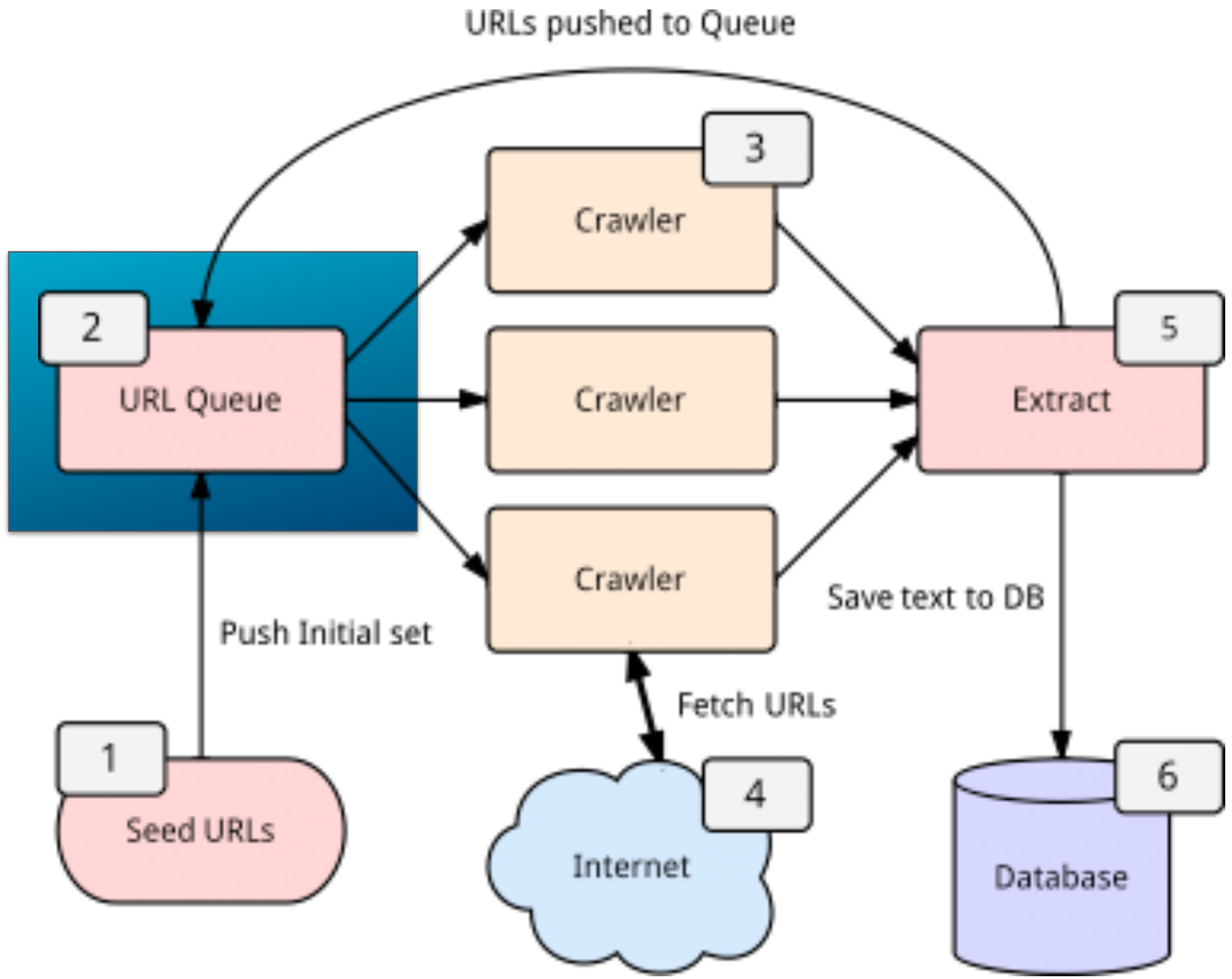
start with a list of seed url's which we push to the URL Queue

crawlers pop from the queue and retrieve the pages

the retrieved pages are extracted, the data is saved to db

new links which are found are added to the URL Queue

the process continues until the URL Queue is empty



Web Crawler URL Queue

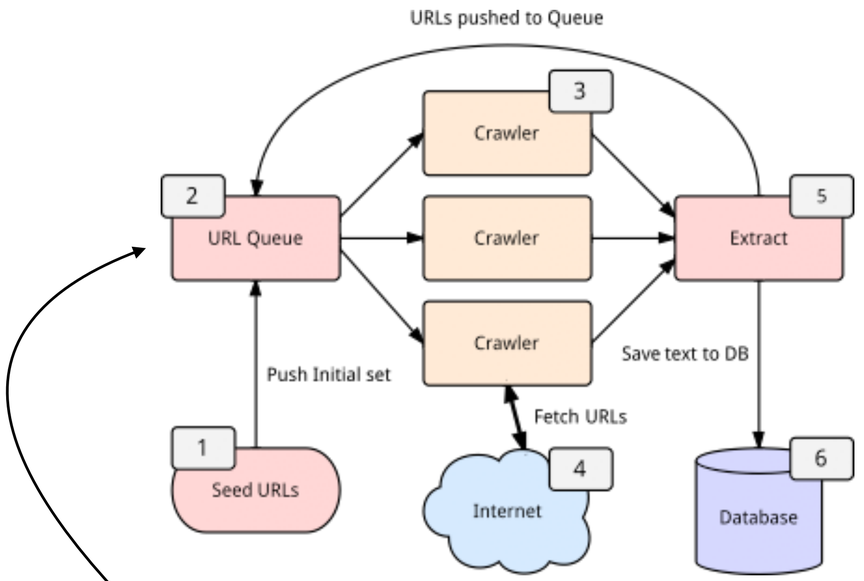
start with a list of seed url's which we push to the URL Queue

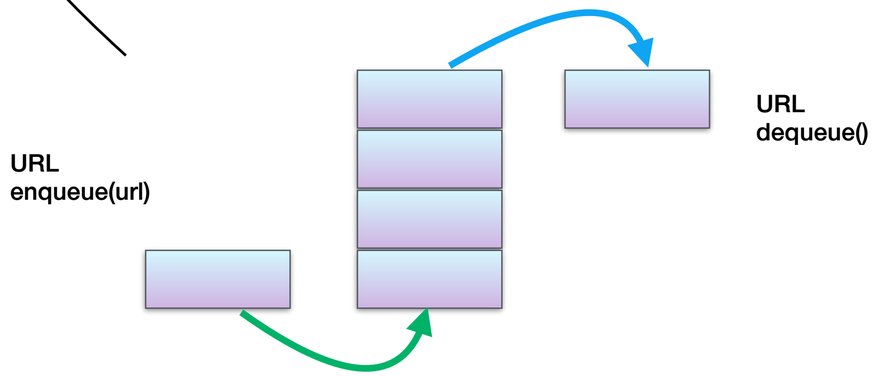
crawlers pop from the queue and retrieve the pages

the retrieved pages are extracted, the data is saved to db

new links which are found are added to the URL Queue

the process continues until the URL Queue is empty





URL Queue

Queue: Applications

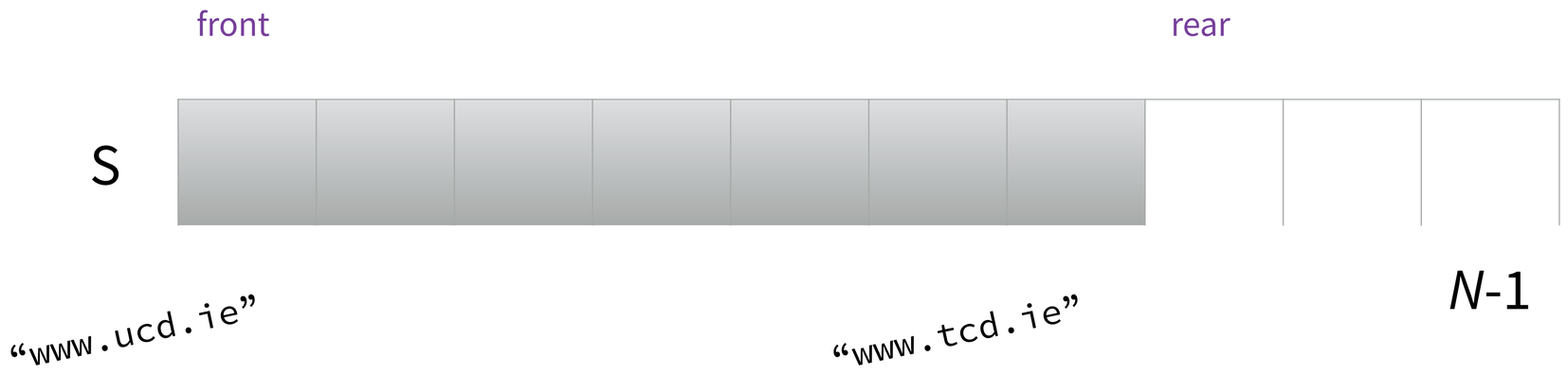
- Direct
  - waiting lists
  - access to shared resources (printer queues, ...)
  - parallel programming

- Indirect
  - auxiliary data structure
  - component of other data structures

Queue (Array Implementation)

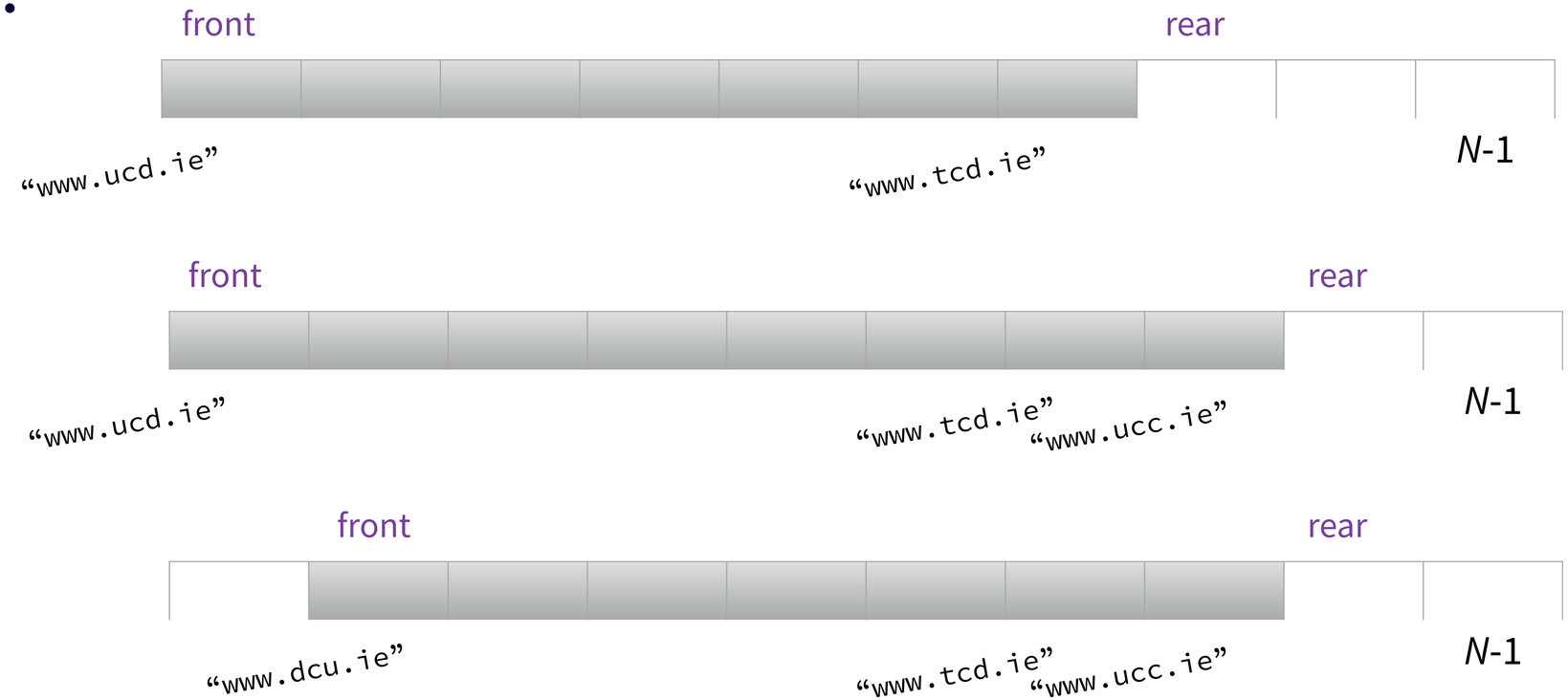
Array based Queue

- use an array of size N to hold the elements
- two variables to keep track of *front* and *rear*
- the *rear* location is always empty

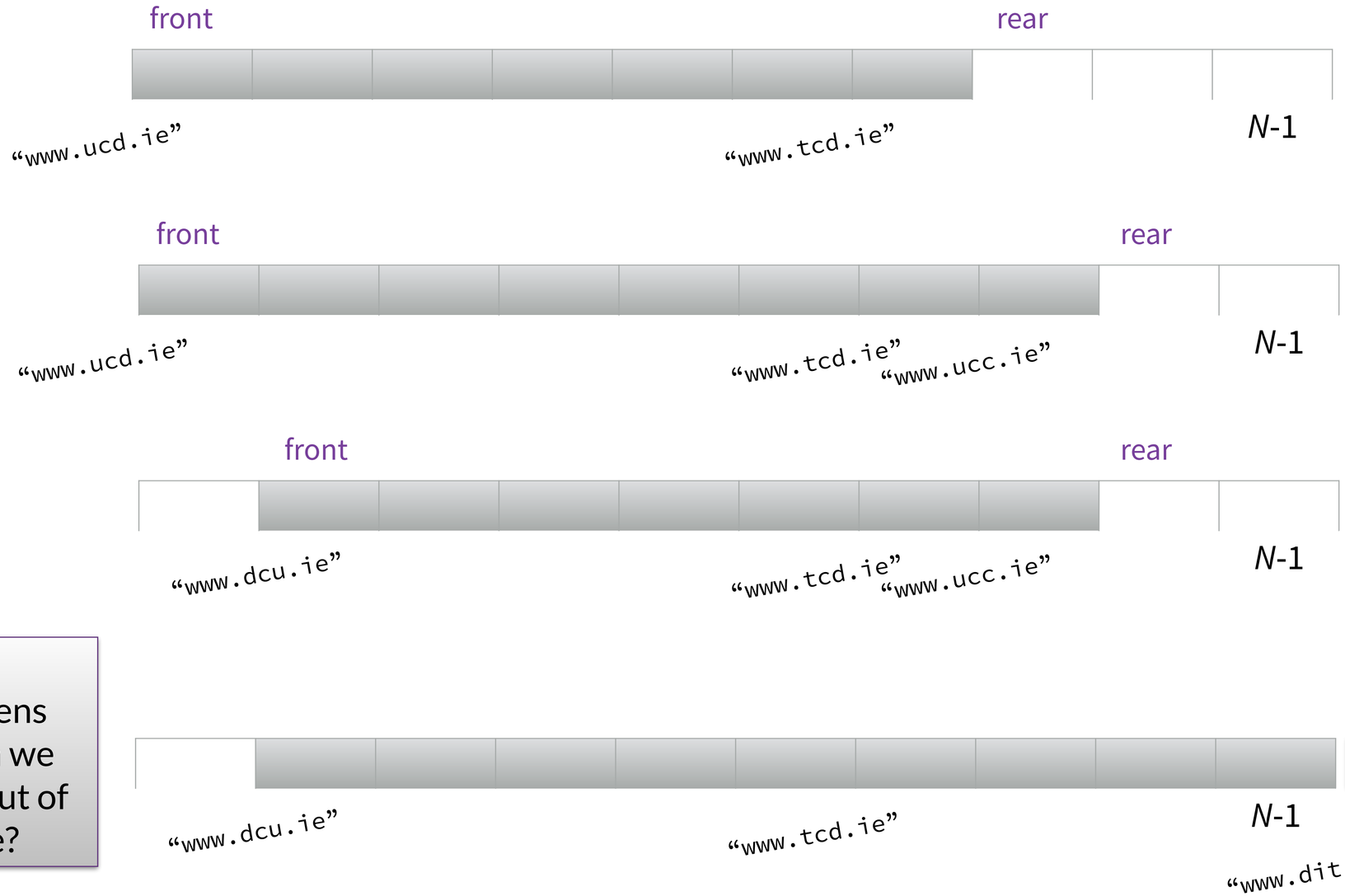


Array based Queue

- enqueue'ing elements grows the *rear* variable
- dequeue'ing elements grows the *front* variable
- the elements 'crawl' to the right as we modify the queue:



Array based Queue



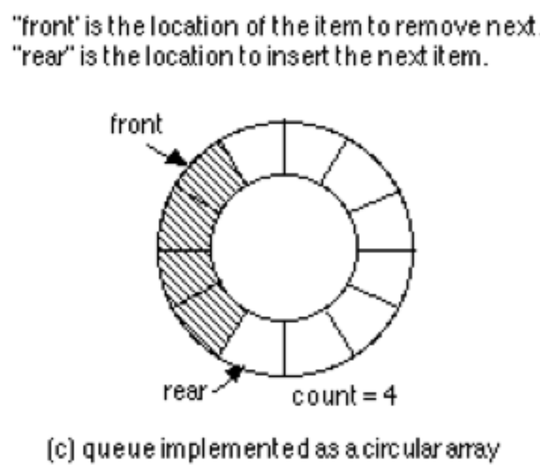
what happens when we run out of space?

Array based Queue

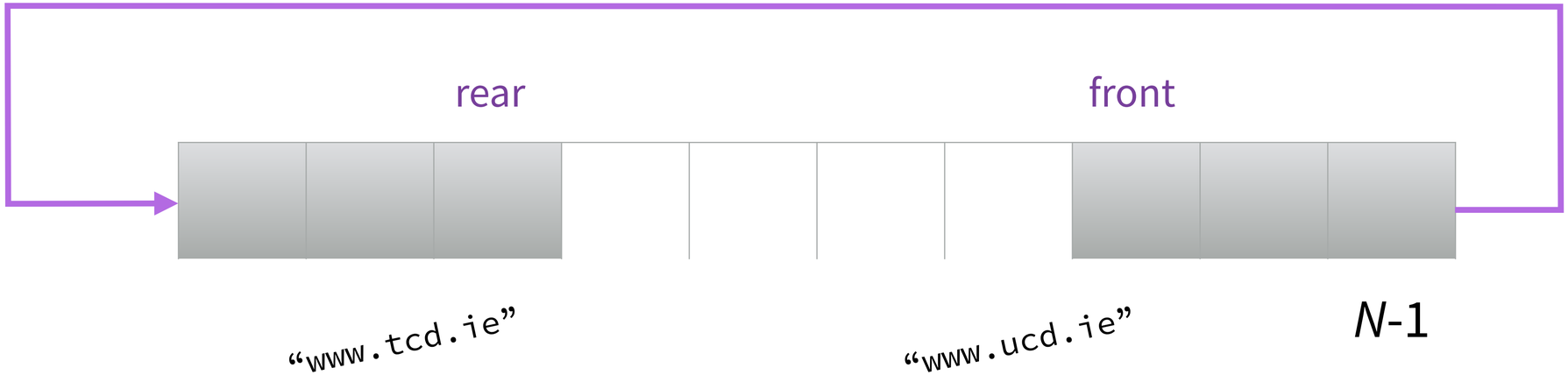
- we could adapt the array-based approach we used for the stack implementation
- the front of the queue would always be the first element of the array: S[0]
- but this is not an efficient solution as it requires that we move all the elements forward one array cell each time we perform a dequeue operation.
- this would therefore require O(n) time to perform the dequeue function
- better solution: Circular arrays

Array based Queue

- treat the array as circular
- join both ends
- no need to move objects



(c) queue implemented as a circular array

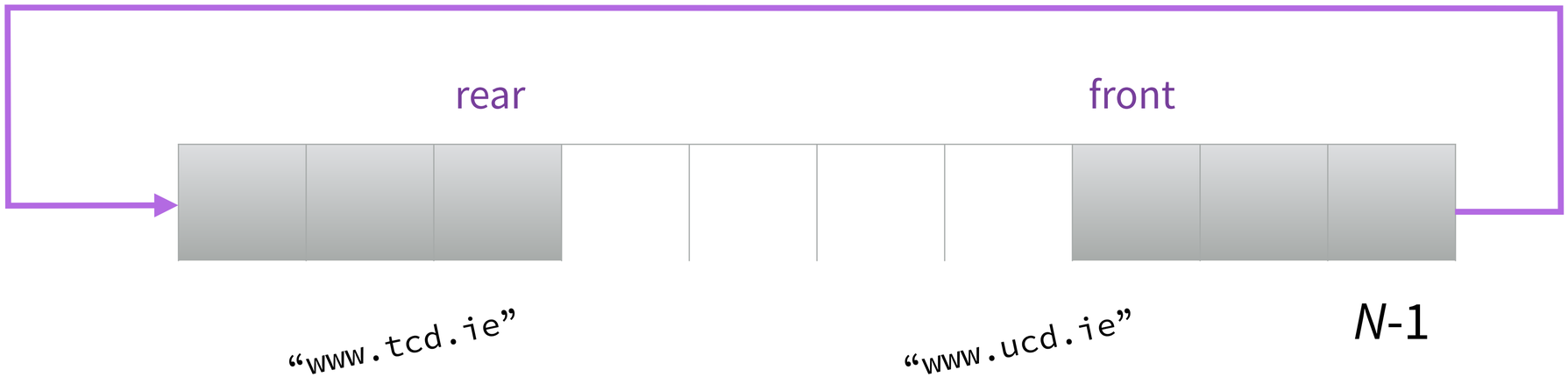


Array based Queue

- treat the array as circular
- join both ends

```
Algorithm enqueue(o):
    Input: object o
    Output: none
    S[rear] ← o
    rear ← (rear + 1) % N
```

rear index wraps around

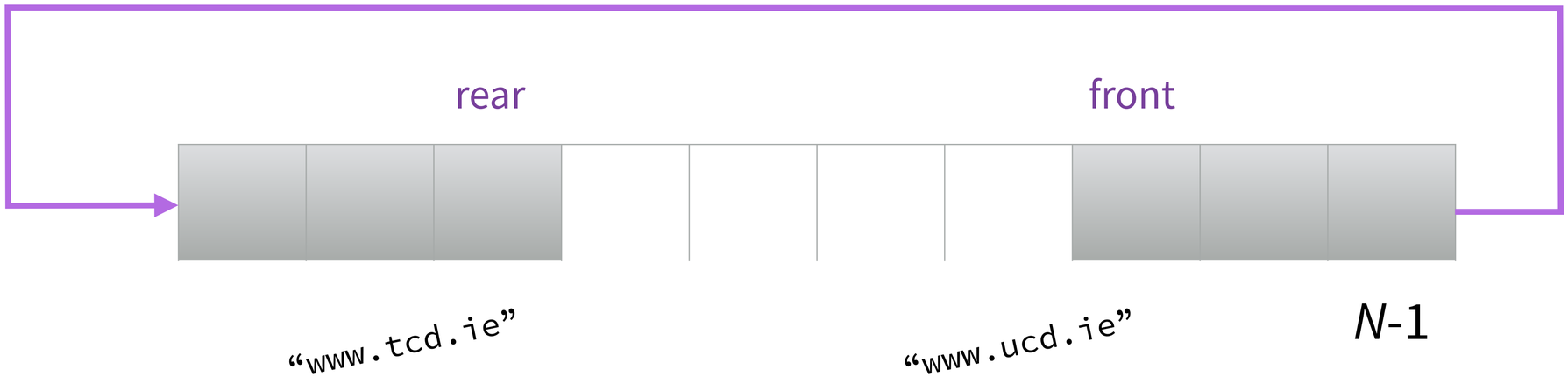


Array based Queue

- treat the array as circular
- join both ends

```
Algorithm dequeue():
    Input: none
    Output: the front object
    o ← S[front]
    front ← (front + 1) % N
    return o
```

front index wraps around

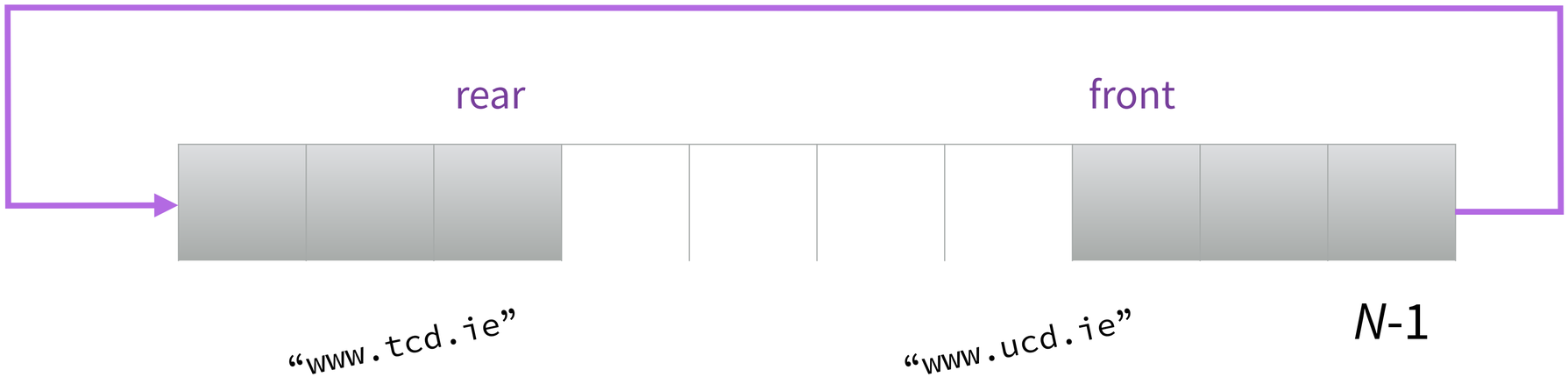


Array based Queue

- treat the array as circular
- join both ends

```
Algorithm size():
    return (N - front + rear) % N

Algorithm empty():
    return front == rear
```



Array based queues

- using a circular array based container, the queue operations are all constant time O(*1*)

Operation	Time
size()	O(1)
empty()	O(1)
front()	O(1)
dequeue()	O(1)
enqueue()	O(1)

Queue (Linked Implementation)

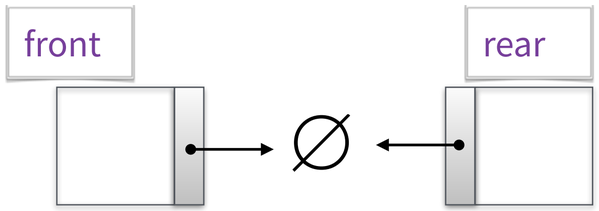
Link-based Queue

Auxiliary data structure: Node

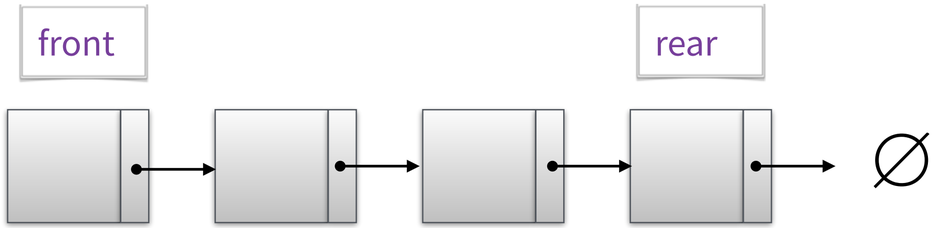
- a reference to the object stored in the queue
- a link to the next node in the queue

Key Nodes:

- the “front” node of the queue
- the “rear” node of the queue
- the size of the queue



empty queue



queue size() == 4

Link-based Queue

Operations at the front of a singly-linked list are O(*1*),

but sometimes at the end they are O(*n*), because you have to find the last element each time

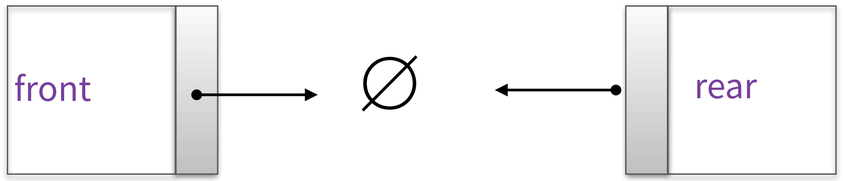
BUT: we can avoid this and have both insertions and deletions run in O(*1*) time:

if we store an additional pointer to the *last* node in the list

Alternatively, we could use a Doubly Linked List

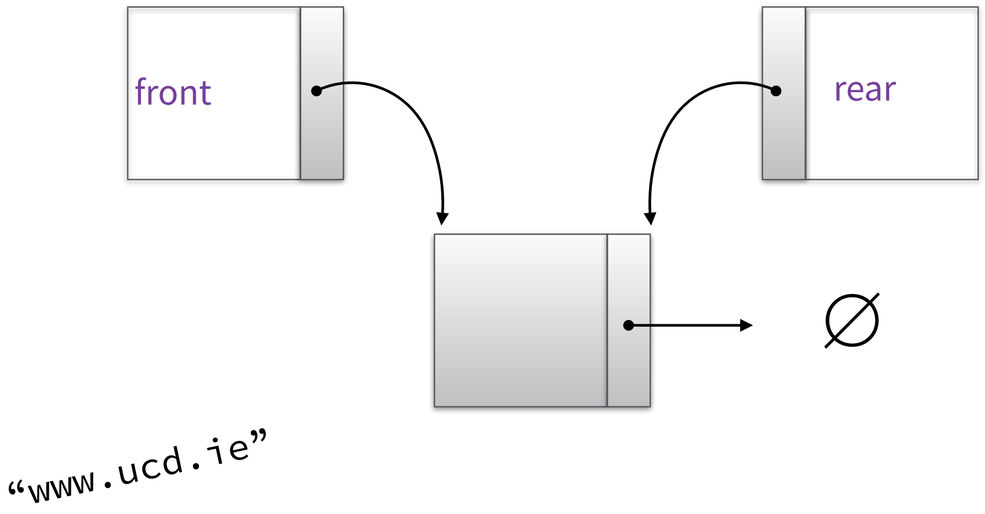
Link-based Queue - enqueue

enqueue("www.ucd.ie")



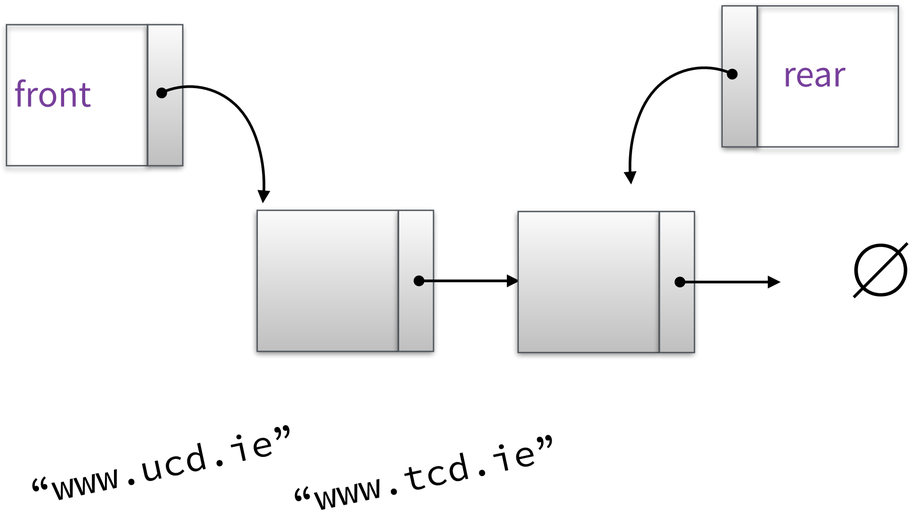
Link-based Queue - enqueue

enqueue(“*www.ucd.ie*”)



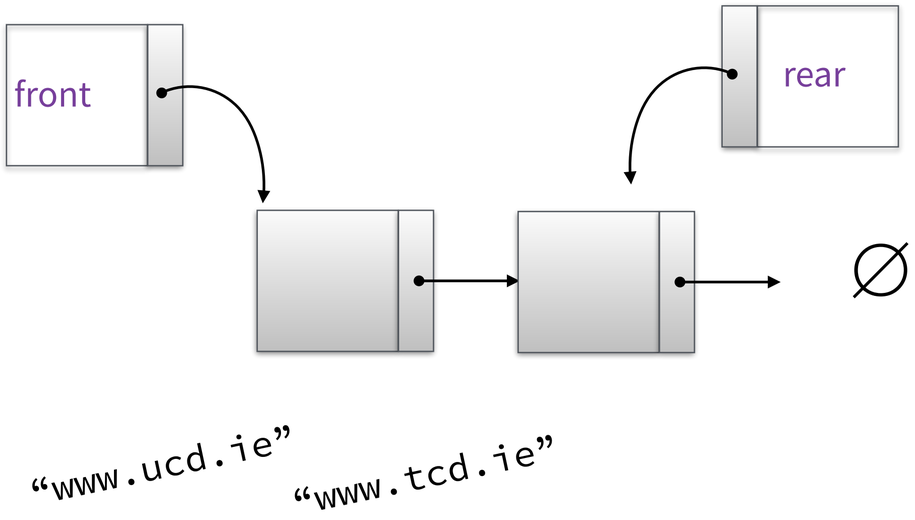
Link-based Queue - enqueue

enqueue(“*www.tcd.ie*”)



Link-based Queue - enqueue

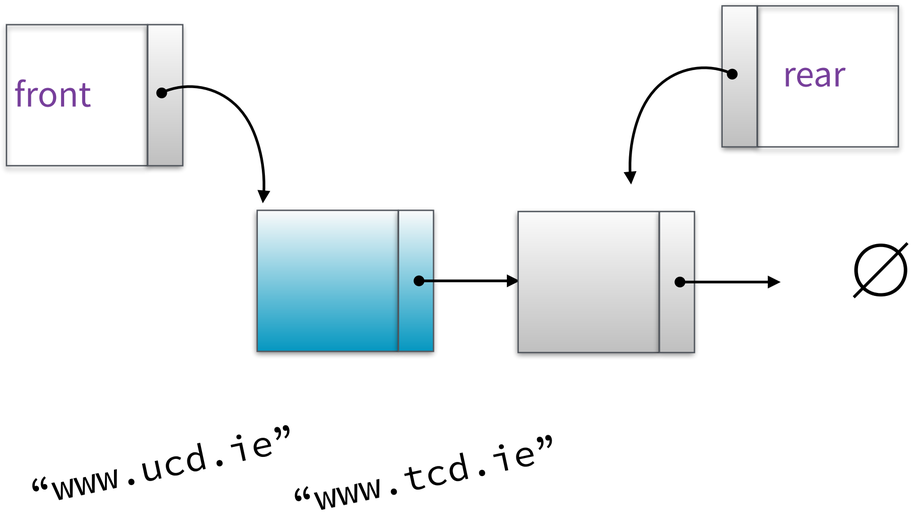
enqueue(“*www.tcd.ie*”)




```
Algorithm enqueue(o):
    Input: an object o
    Output: none
    node ← new Node(o)
    if rear is null then:
        front ← node
    else:
        rear.next ← node
    rear ← node
    size ← size + 1
```

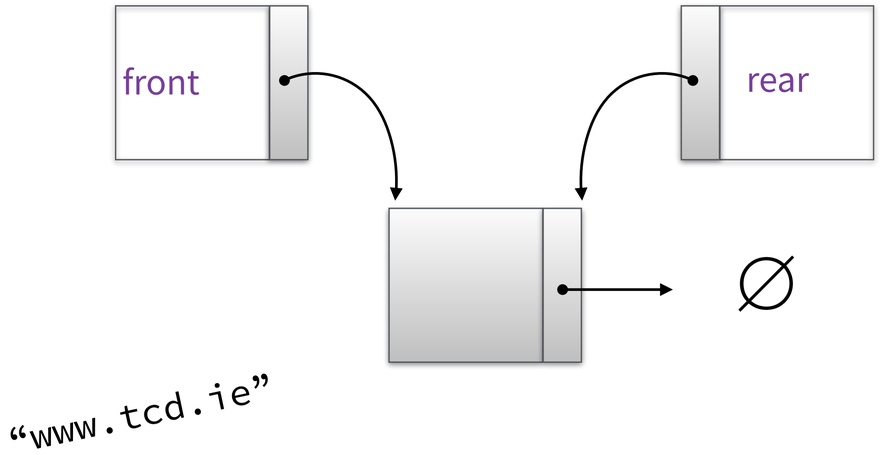
Link-based Queue - dequeue

dequeue()

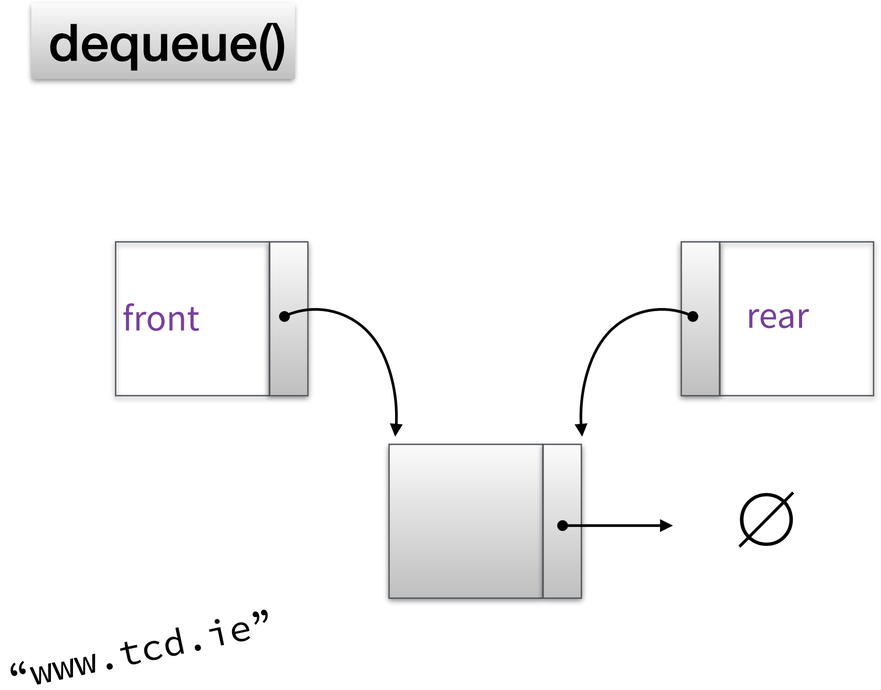


Link-based Queue - dequeue

dequeue()



Link-based Queue - enqueue




```
Algorithm dequeue():
    Input: none
    Output: the front object
    e ← front.element
    front ← front.next
    if front is null then:
        rear ← null
    size ← size - 1
    return e
```

Link-based Queue

```
Algorithm front():
    Input: none
    Output: the top object
    return front.element
```

```
Algorithm size():
    Input: none
    Output: number of objects in the queue
    return size
```

```
Algorithm empty():
    Input: none
    Output: true if queue is empty, false otherwise
    return size == 0
```

Link-based queues - analysis

- using a link based container, the queue operations are all constant time O(*1*)

Operation	Time
size()	O(1)
empty()	O(1)
front()	O(1)
dequeue()	O(1)
enqueue()	O(1)

Queue Implementations

Java Queue (jdk1.8)

java.util
Interface Queue<E>

- Type Parameters:
  E - the type of elements held in this collection

  All Superinterfaces:
  Collection<E>, Iterable<E>

  All Known Subinterfaces:
  BlockingDeque<E>, BlockingQueue<E>, Deque<E>, TransferQueue<E>

  All Known Implementing Classes:
  AbstractQueue, ArrayBlockingQueue, ArrayDeque, ConcurrentLinkedDeque, ConcurrentLinkedQueue, DelayQueue, LinkedBlockingDeque, LinkedBlockingQueue, LinkedList, LinkedTransferQueue, PriorityBlockingQueue, PriorityQueue, SynchronousQueue

Java Queue (jdk1.8)

A collection designed for holding elements prior to processing.
Besides basic Collection operations, queues provide additional insertion, extraction, and inspection operations.
Each of these methods exists in two forms: one throws an exception if the operation fails, the other returns a special value (either null or false, depending on the operation). The latter form of the insert operation is designed specifically for use with capacity-restricted Queue implementations; in most implementations, insert operations cannot fail.

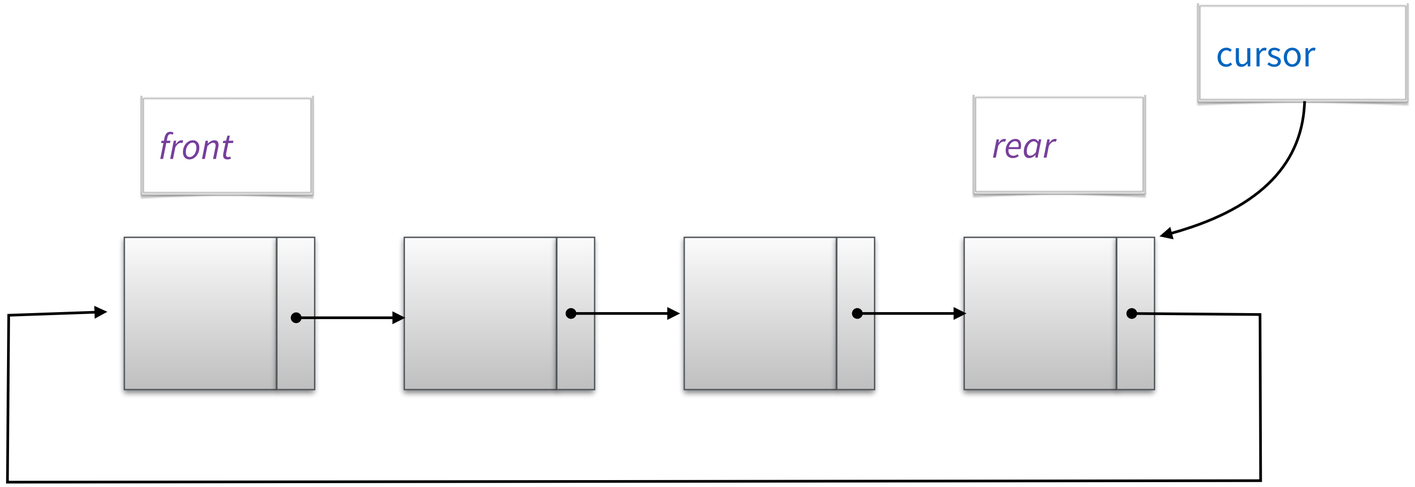
	Throws exception	Returns special value
Insert	add(e)	offer(e)
Remove	remove()	poll()
Examine	element()	peek()

Queue (Circular Linked List Implementation)

Circularly Linked List Queue

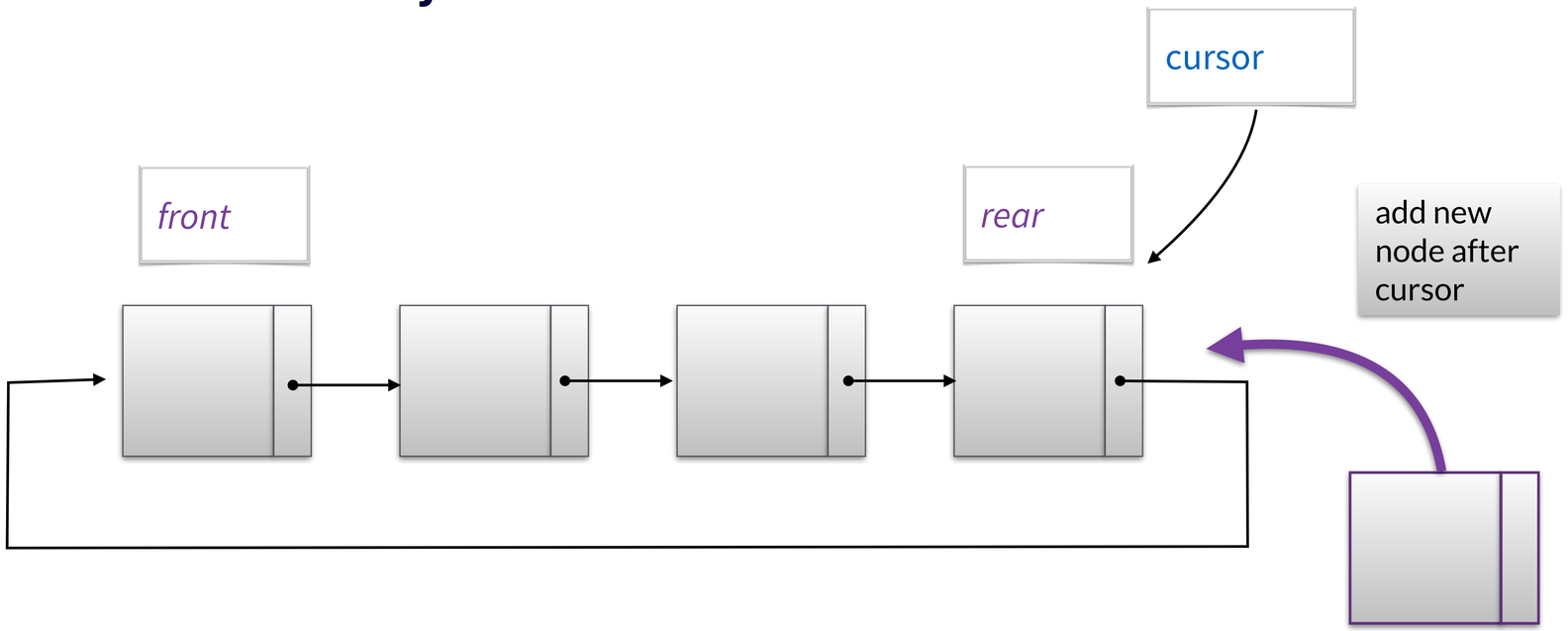
- we delete from the head of the queue and insert at the rear
- our singly linked list is not very efficient, since it provides efficient access only to one side of the list.
- try our circularly linked list
- the ***cursor*** points to one node of the list
- there are two member functions
  - back(): reference to the element to which the cursor points
  - front(): reference to the element that immediately follows it
- Queue:
  - back() -> rear of the queue
  - front() -> front of the queue

Queues - Circularly Linked List - Enqueue()



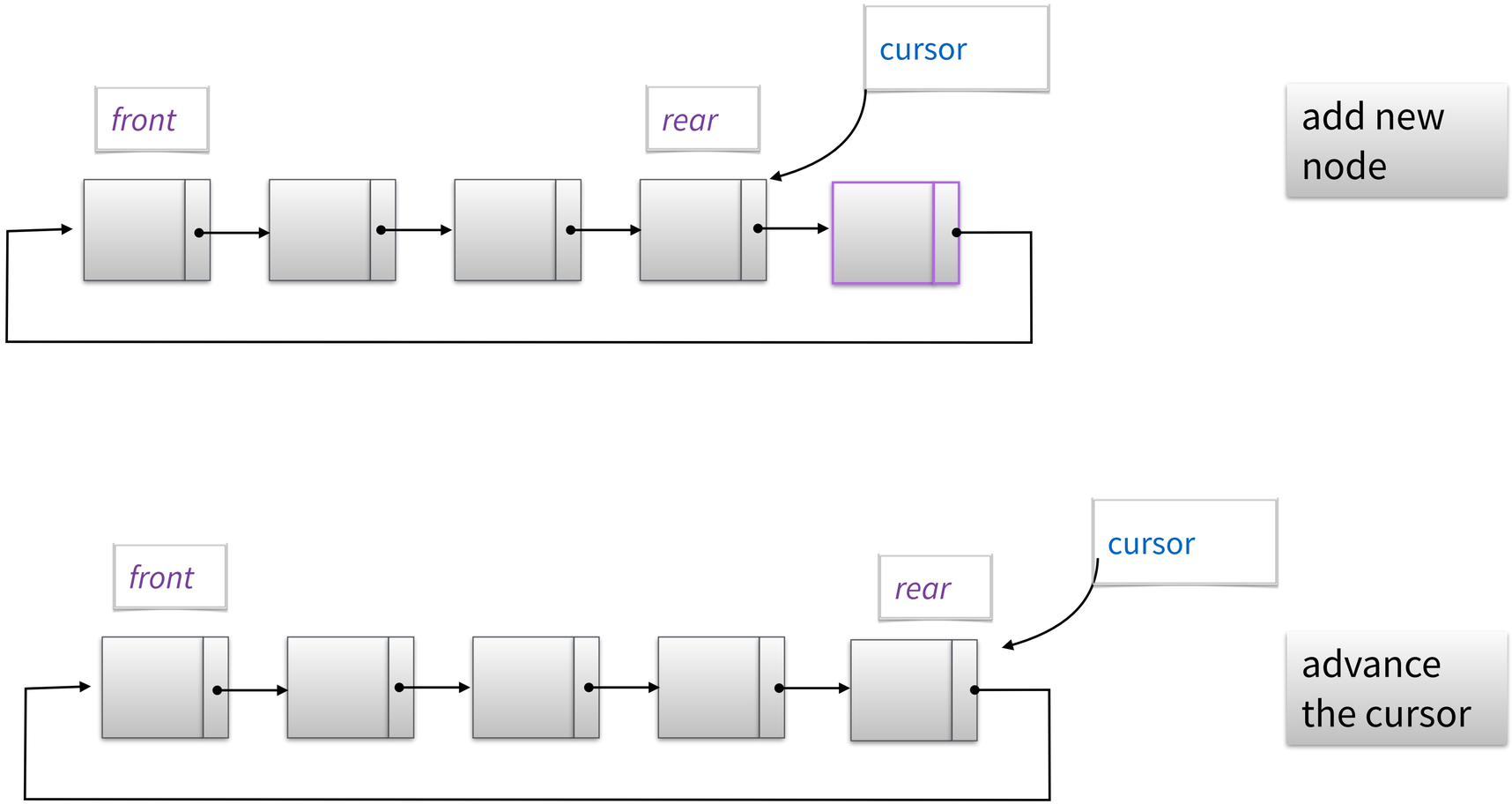
Queues - Circularly Linked List - Enqueue()

- to implement *enqueue()*, we first invoke the function *add()*, which inserts a new element just after the cursor



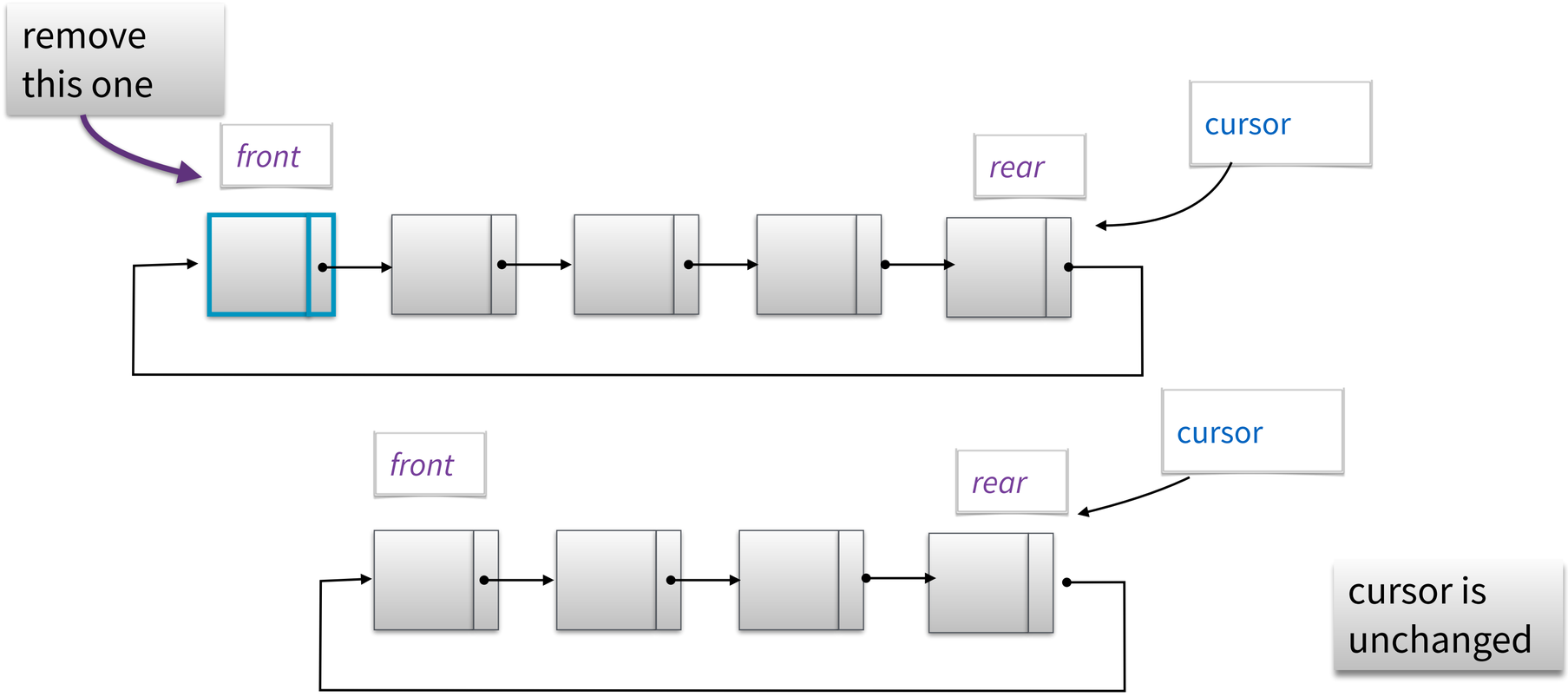
Queues - Circularly Linked List - Enqueue()

- we then invoke *advance()*, which advances the cursor to this new element, and the new node is now the rear of the queue.



Queues - Circularly Linked List - Dequeue()

- to implement *dequeue()*, we remove the node immediately following the cursor



Queue Java Implementation

Queues - Java Implementation

```
public interface Queue<E> {
  /**
   * Returns the number of elements in
the queue.
   */
  int size();

  /**
   * Tests whether the queue is empty.
  */
  boolean isEmpty();

  /**
   * Inserts an element at the rear of
the queue.
  */
  void enqueue(E e);

  /**
   * Returns, but does not remove, the
first element of the queue.
  */
  E first();

  /**
   * Removes and returns the first
element of the queue.
  */
  E dequeue();
}
```

Start with the Queue ADT

Array Queue

```
public class ArrayQueue<E> implements Queue<E>
{
  /** Default array capacity. */
  public static final int CAPACITY = 1000; //
default array capacity

  private E[] data; // generic array used for
storage

  private int f = 0; // index of the front
element

  private int sz = 0; // current number of
elements

  // constructors
  public ArrayQueue() {
    this(CAPACITY);
  } // constructs queue with default capacity


  /**
   * Constructs and empty queue with the given
array capacity.
   */
  public ArrayQueue(int capacity) { //
constructs queue with given capacity
    data = (E[]) new Object[capacity]; // safe
cast; compiler may give warning
  }

  /**
   * Returns the number of elements in the
queue.
   */
  public int size() {
    return sz;
  }

  /** Tests whether the queue is empty. */
  public boolean isEmpty() {
    return (sz == 0);
  }
```

Array Queue

```
/**
   * Inserts an element at the rear of
the queue. This method runs in O(1)
time.
   */
  public void enqueue(E e) throws
IllegalStateException {
    if (sz == data.length)
      throw new
IllegalStateException("Queue is full”);
    // use modular arithmetic
    int avail = (f + sz) % data.length;
    data[avail] = e;
    sz++;
  }
```

```
  /**
   * Removes and returns the first
element of the queue.
   */
  public E dequeue() {
    if (isEmpty())
      return null;
    E answer = data[f];
    data[f] = null; // dereference to
help garbage collection
    f = (f + 1) % data.length;
    sz--;
    return answer;
  }
```

```
/**
 * Returns, but does not remove, the first element of the queue.
 *
 */
public E first() {
    if (isEmpty())
        return null;
    return data[f];
}
```

Linked Queue

```
public class LinkedQueue<E> implements
Queue<E> {

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

  public LinkedQueue() {
  } // new queue relies on the initially
empty list

  /**
   * Returns the number of elements in the
queue.
   */
  public int size() {
    return list.size();
  }
```

```
  /**
   * Tests whether the queue is empty.
   */
  public boolean isEmpty() {
    return list.isEmpty();
  }

  /**
   * Returns, but does not remove, the
first element of the queue.
   */
  public E first() {
    return list.first();
  }
```

Linked Queue

```
/**
   * Inserts an element at the rear of the queue.
   */
  public void enqueue(E element) {
    list.addLast(element);
  }


  /**
   * Removes and returns the first element of the queue.
   */
  public E dequeue() {
    return list.removeFirst();
  }
```

Double Ended Queue

Double Ended Queues

- we might like to have a queue-like data structure that supports insertion and deletion at both the front and the rear of the queue
- this extension of a queue is called a ***double-ended queue***, or ***deque***
- usually pronounced "deck" to avoid confusion with the dequeue function of the regular queue ADT, which is pronounced like the abbreviation "D.Q."

Abstract Data Type

Deque ADT

insertFront(e)	insert element e at front of the queue
insertBack(e)	insert element e at rear of the queue
eraseFront()	remove element at front of the queue
eraseBack()	remove element at back of the queue
front()	return a reference to the front element of the queue without removing it
back()	return a reference to the back element of the queue without removing it
size()	number of elements in the stack
empty()	true if the stack is empty, false otherwise

deque Java

Interface Deque<E>

- **Type Parameters:**
  E - the type of elements held in this collection

  All Superinterfaces:
  Collection<E>, Iterable<E>, Queue<E>

  All Known Subinterfaces:
  BlockingDeque<E>

  All Known Implementing Classes:
  ArrayDeque, ConcurrentLinkedDeque, LinkedBlockingDeque, LinkedList

deque Java

A linear collection that supports element insertion and removal at both ends. Most Deque implementations place no fixed limits on the number of elements they may contain, but this interface supports capacity-restricted deques as well as those with no fixed size limit.

This interface defines methods to access the elements at both ends of the deque. Methods are provided to insert, remove, and examine the element. Each of these methods exists in two forms: one throws an exception if the operation fails, the other returns a special value (either null or false, depending on the operation).

Deque

	First Element (Head)		Last Element (Tail)	
	Throws exception	Special value	Throws exception	Special value
Insert	addFirst(e)	offerFirst(e)	addLast(e)	offerLast(e)
Remove	removeFirst()	pollFirst()	removeLast()	pollLast()
Examine	getFirst()	peekFirst()	getLast()	peekLast()

Deque

Queues -> Deques

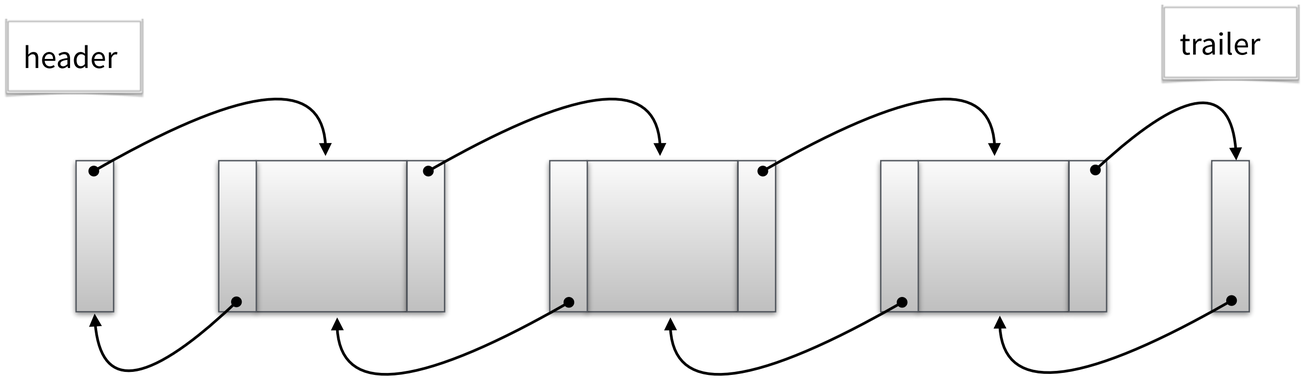
Queue Method	Equivalent Deque Method
add(e)	addLast(e)
offer(e)	offerLast(e)
remove()	removeFirst()
poll()	pollFirst()
element()	getFirst()
peek()	peekFirst()

Stacks -> Deques

Stack Method	Equivalent Deque Method
push(e)	addFirst(e)
pop()	removeFirst()
peek()	peekFirst()

deque

- implement the deque ADT using a linked representation
- a deque supports efficient access at both ends of the list
- a doubly linked list implementation is suitable for this
- we place the front of the deque at the header of the linked list and the rear of the queue at the trailer.



deque - performance

- with the doubly linked list implementation, all operations are constant time

Operation	Time
size()	O(1)
empty()	O(1)
front()/back()	O(1)
insertFront()/insertBack	O(1)
eraseFront()/eraseBack()	O(1)

deque

- Specific libraries may implement *deques* in different ways, often as some form of dynamic array, or linked list
- they provide a functionality similar to Vectors, but with efficient insertion and deletion of elements also at the beginning of the sequence, and not only at its end.
- *deques* are not guaranteed to store all their elements in contiguous storage locations: accessing elements in a deque by offsetting a pointer to another element causes undefined behaviour.
- Unlike the `List` interface, Deque does not provide support for indexed access to elements.

deque

```
public class ArrayDequeCS<Item>
implements Iterable<Item> {
   private static final int
MIN_ARRAY_SIZE = 2;

   private Item[] items;

   private int itemCount, firstPosition,
lastPosition;

   public ArrayDequeCS() {
        itemCount = 0;
        items = (Item[]) new
Object[MIN_ARRAY_SIZE];
        firstPosition = 0;
        lastPosition = 1;
    }
```

```
    /**
     * Checks if {@code Deque} is empty
    */
    public boolean isEmpty() {
        return size() == 0;
    }

    /**
     * Returns the number of items in
{@code Deque}
    */
    public int size() {
        return itemCount;
    }
```

deque

```
/**
 * Adds given item to the front of the {@code Deque} and
 * increases number of items in it by one. Also this method can
 * resize the array if there is no extra space for inserted item
*/
public void addFirst(Item item) {
    if (item == null) {
        throw new NullPointerException("You can not add Null to deque!");
    }
    if (firstPosition < 0) {
        resize(items.length + itemCount, Side.FRONT);
    }
    items[firstPosition--] = item;
    itemCount++;
}
```

deque

```
/**
 * Adds given item to the end of the {@code Deque} and
 * increases number of items in it by one. Also this method can
 * resize the array if there is no extra space for inserted item
*/
public void addLast(Item item) {
    if (item == null) {
        throw new NullPointerException("You can not add Null to deque!");
    }
    if (lastPosition == items.length) {
        resize(items.length + itemCount, Side.END);
    }
    items[lastPosition++] = item;
    itemCount++;
}
```

deque

```
    /**
     * Removes and returns the item form the front of the {@code Deque}. This method
     * decreases the number of items in the {@code Deque} by one and can shrink the array
     * if there is a lot extra space in the front of it.
    */
public Item removeFirst() {
    if (isEmpty()) {
        throw new NoSuchElementException("Deque is already empty!");
    }
    Item item = items[++firstPosition];
    items[firstPosition] = null;
    itemCount--;
    if (itemCount > 0 && firstPosition + items.length - lastPosition >= itemCount << 2) {
        resize(itemCount << 1, Side.BOTH);
    }
    return item;
}
```

deque

```
    /**
     * Removes and returns the item form the end of the {@code Deque}. This method
     * decreases the number of items in the {@code Deque} by one and can shrink the array
     * if there is a lot extra space in the end of it.
    */
public Item removeLast() {
    if (isEmpty()) {
        throw new NoSuchElementException("Deque is already empty!");
    }
    Item item = items[--lastPosition];
    items[lastPosition] = null;
    itemCount--;
    if (itemCount > 0 && firstPosition + items.length - lastPosition >= itemCount << 2) {
        resize(itemCount << 1, Side.BOTH);
    }
    return item;
}
```

Deque

a lot easier if you use a java.util.Vector!

```
    /**
     * Enum representing sides of the {@code Deque}
     */
    private enum Side {
        /**
         * Front side
         */
        FRONT,
        /**
         * End side
         */
        END,
        /**
         * Both sides
         */
        BOTH
    }
    /**
     * Increases or decreases the size of the array with items by
given capacity from given side.
     * This method does nothing if capacity is less than the number
of items in {@code Deque}
     */
   private void resize(int capacity, Side side) {
        if (capacity < itemCount) {
            return;
        }
        Item[] tmpArr = (Item[]) new Object[capacity];
        int destinationPosition = 0;
        int length = lastPosition;
        int start = 0;
        switch (side){
            case BOTH :
                start = firstPosition + 1;
                destinationPosition = (capacity - itemCount) >> 1;
                firstPosition = destinationPosition - 1;
                length = itemCount;
                lastPosition = destinationPosition + length;
                break;
            case FRONT:
                if (capacity < items.length) {
                    start = firstPosition;
                    int difference = items.length - capacity;
                    firstPosition -= difference;
                    lastPosition -= difference;
                    destinationPosition = firstPosition;
                    length = itemCount + 1;
                } else {
                    firstPosition = itemCount - 1;
                    lastPosition += itemCount;
                    destinationPosition = length = itemCount;
                }
                break;
            default:
                // no actions required if side is END
                break;
        }

        System.arraycopy(items, start, tmpArr, destinationPosition,
length);
        items = tmpArr;
    }
```

Priority Queues

Priority Queues

A priority queue stores a collection of entries

Each entry is a pair (key, value)

Main methods of the Priority Queue ADT

Priority Queue ADT

insert(k, v)	insert an entry with key k and value v
removeMin()	removes and returns the entry with the smallest key, or null if the queue is empty
min()	returns but does not remove the entry with the smallest key, or null if the queue is empty
size()	number of elements in the queue
empty()	true if the queue is empty, false otherwise

Priority Queues

A sequence of priority queue methods:

Method	Return Value	Priority Queue Contents
insert(5,A)		{ (5,A) }
insert(9,C)		{ (5,A), (9,C) }
insert(3,B)		{ (3,B), (5,A), (9,C) }
min( )	(3,B)	{ (3,B), (5,A), (9,C) }
removeMin( )	(3,B)	{ (5,A), (9,C) }
insert(7,D)		{ (5,A), (7,D), (9,C) }
removeMin( )	(5,A)	{ (7,D), (9,C) }
removeMin( )	(7,D)	{ (9,C) }
removeMin( )	(9,C)	{ }
removeMin( )	null	{ }
isEmpty( )	true	{ }

Priority Queues

Keys in a priority queue can be arbitrary objects on which an order is defined

Two distinct entries in a priority queue can have the same key

Mathematical concept of total order relation $\leq$

Comparability property:

either $x \leq y$ or $y \leq x$

Antisymmetric property:

$x \leq y$ and $y \leq x$

$\Rightarrow x = y$

Transitive property:

$x \leq y$ and $y \leq z$

$\Rightarrow x \leq z$

Entry ADT

An entry in a priority queue is simply a key-value pair

Priority queues store entries to allow for efficient insertion and removal based on keys

»

Entry ADT

getKey()	return the key for this entry
getValue()	returns the value for this entry

```
/**
 * Interface for a key-value pair entry
 **/
public interface Entry<K, V> {
  K getKey();

  V getValue();
}
```

Comparator ADT

- A comparator encapsulates the action of comparing two objects according to a given total order relation
- A generic priority queue uses an auxiliary comparator
- The comparator is external to the keys being compared
- When the priority queue needs to compare two keys, it uses its comparator

Comparator ADT

compare(x,y)	returns an integer i, such that i < 0 if x < y i = 0 if x = y i > 0 if x > y

Comparator

```
ArrayList<Point> coordinateList = new ArrayList<Point>();
//
Collections.sort(coordinateList, new PointCompare());
```

```
public class PointCompare implements Comparator<Point> {

  public int compare(final Point a, final Point b) {
    if (a.x < b.x) {
      return -1;
    } else if (a.x > b.x) {
      return 1;
    } else {
      return 0;
    }
  }
}
```

Priority Queue - Sequence Based

Implementation with an ***unsorted*** list

Performance:

insert takes O(1) time since we can insert the item at the beginning or end of the sequence

removeMin and min take O(n) time since we have to traverse the entire sequence to find the smallest key

Implementation with a ***sorted*** list

Performance:

insert takes O(n) time since we have to find the place where to insert the item

removeMin and min take O(1) time, since the smallest key is at the beginning

Priority Queue - Sorted Sequence Based

```
/**
 * Inserts a key-value pair and returns the entry created.
 */
public Entry<K, V> insert(K key, V value) throws IllegalArgumentException {
  checkKey(key); // auxiliary key-checking method (could throw exception)
  Entry<K, V> newest = new PQEntry<>(key, value);

  // start at the end of the list
  Node<Entry<K, V>> curr = list.tail();

  // walk backward, looking for smaller key
  while (curr != null && compare(newest, curr.getElement()) < 0) {
    curr = curr.getPrev();
  }
  if(curr == list.getHeader()) {
    list.addFirst(newest); // new key is smallest
  }
  else {
    list.addAfter(curr, newest); // newest goes after walk
  }
  return newest;
}
```

keeps the PQ sorted as we insert entries

Priority Queue - Sequence Based

```java
/**
 * Returns (but does not remove) an entry with minimal key.
 * @return entry having a minimal key (or null if empty)
 */
@Override
public Entry<K,V> min() {
  if (list.isEmpty()) return null;
  return list.first().getElement();
}

/**
 * Removes and returns an entry with minimal key.
 * @return the removed entry (or null if empty)
 */
@Override
public Entry<K,V> removeMin() {
  if (list.isEmpty()) return null;
  return list.remove(list.first());
}
```

now, the min just gets the first node in the sorted list

Priority Queue - Unsorted sequence

- Alternatively, we could just insert the entry at the end of the list
- Do the sorting when we need to remove the minimum
- Which is better?
- Depends on the usage...

Sorting with Priority Queues

We can use a priority queue to sort a list of comparable elements

Insert the elements one by one with a series of insert operations

Remove the elements in sorted order with a series of removeMin operations

The running time of this sorting method depends on the priority queue implementation

```
1:  function PriorityQueueSort(S, C)
2:      Input list S and comparator C for the elements of S
3:      Output list S sorted in increasing order according to C
4:      P ← priority queue with comparator C
5:      while !S.isEmpty() do
6:          e ← S.remove(S.first())
7:          P.insert(e, ∅)
8:      end while
9:      while !P.isEmpty() do
10:         e ← P.removeMin().getKey()
11:         S.addLast(e)
12:     end while
13: end function
```

Sorting with Priority Queues - Selection Sort

Selection-sort is the variation of PQ-sort where the priority queue is implemented with an unsorted sequence

Running time of Selection-sort:

1. Inserting the elements into the priority queue with n insert operations takes O(n) time
2. Removing the elements in sorted order from the priority queue with n removeMin operations takes time proportional to

$$1 + 2 + \ldots + n$$

Selection-sort runs in $O(n^2)$ time

Sorting with Priority Queues - Insertion Sort

Insertion-sort is the variation of PQ-sort where the priority queue is implemented with a sorted sequence

Running time of Insertion-sort:

1. Inserting the elements into the priority queue with n insert operations takes time proportional to

$$1 + 2 + \ldots + n$$

2. Removing the elements in sorted order from the priority queue with a series of n removeMin operations takes O(n) time

Insertion-sort runs in $O(n^2)$ time