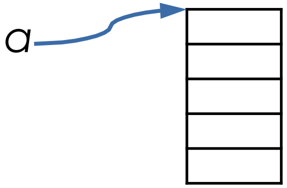
Linked List Motivation

Arrays

- Arrays Store data contiguously in memory

```
int[] a = new int[6];
```

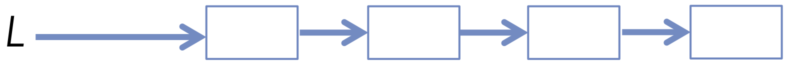


a has the address of the first entry in the array

- Once space is allocated it cannot be grown or shrunk
  - Often it is not possible to envision how many entries are necessary;
  - Space is wasted if too many entries are allocated, or;
  - If too little entries are allocated a new array must be created and all entries copied over.

A Linked List is flexible and overcomes this problem by allocating units of space on demand. Whenever a unit is needed it is created and added to the LL.

- Each unit of the linked list is hooked to the next one forming a list of units

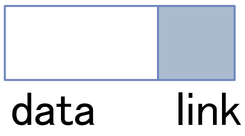


Linked List Unit: Node

Each unit of space in a linked list is called a **node** and it has two parts:

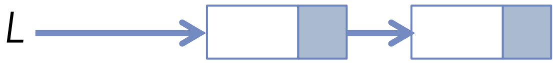
- a data part that holds information;
- a link part that points to the next node on the linked list.

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

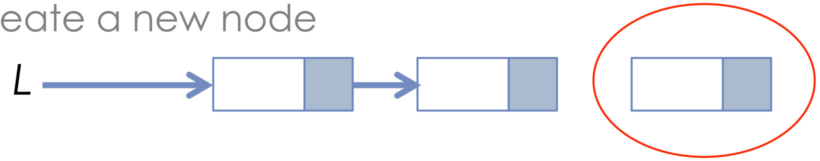


- When a node is allocated it comes from anywhere in memory, unrelated to any previous allocated node. Once it's been created it is then added to the LL by connecting it to a node already in the LL

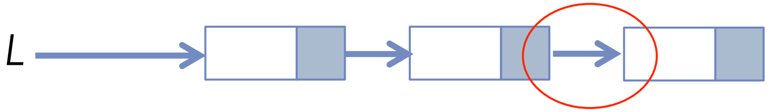
1 Linked list with two nodes



2 Create a new node



3 Connect them



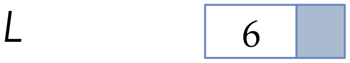
Creating an Integer Linked List

1. Start by creating an access reference pointer to the beginning of the LL

L

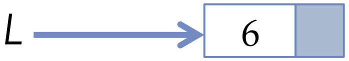
```
Node L = null;
```

2. Create the first node



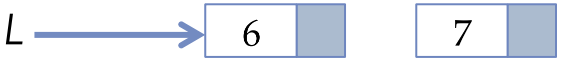
```
Node f = new Node(6, null);
```

3. Make the beginning of the LL point to the first node



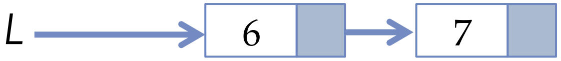
```
L = f;
```

4. Create the second node



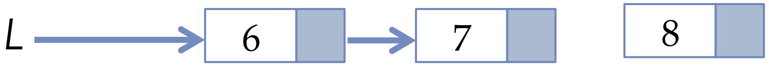
```
Node s = new Node(7, null);
```

5. Make the first point to the second node



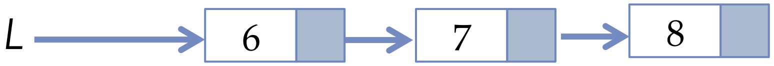
```
f.next = s;
```

6. Create a third node



```
Node t = new Node(8, null);
```

7. Make the second point to the third node

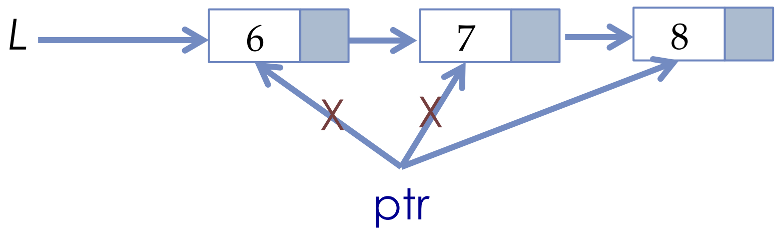


```
s.next = t;
```

Traversing a Linked List

Starting from L, use a sequence of .next incantations

- All entries of a LL from the beginning to end follow a chain of references.
- To traverse, start at the beginning of the list and follow the links



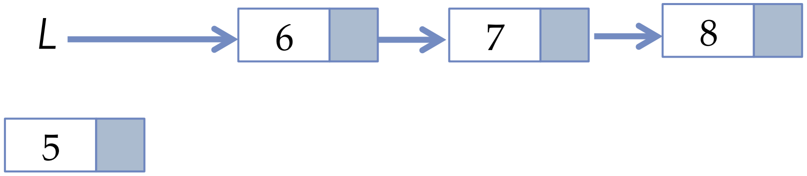
```
Node ptr = L;
while (ptr != null) {
    ptr = ptr.next;
}
```

- What is the running time to traverse the linked list?
  - ptr = ptr.next takes constant time
  - how many times the assignment is done?
    - the size of the linked list
  - O(n)

LL: Insert front

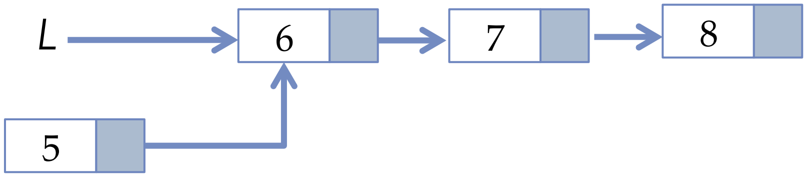
Insert to the front of the list

1. Create a node new node



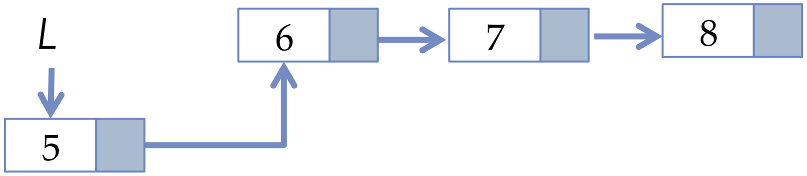
```
Node n = new Node(5, null);
```

2. Make the new node's next point to the first node



```
n.next = L;
```

3. Make the reference to the beginning of the list point to new node



```
L = n;
```

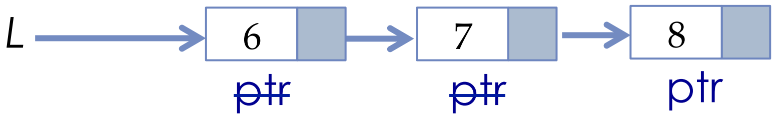
What is the running?

- create a new node and assignments take constant time: O(1)

LL: Insert After Target

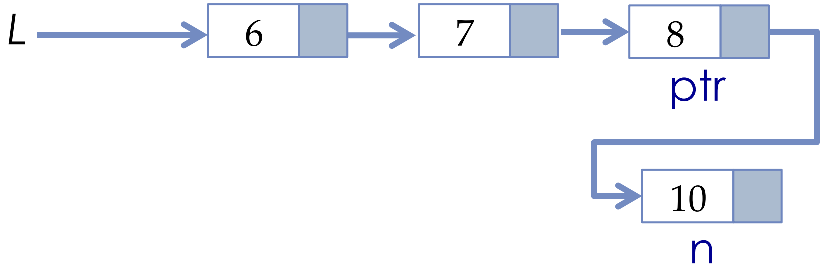
Insert a new node 10 after the node that holds 8

1. Find the target 8



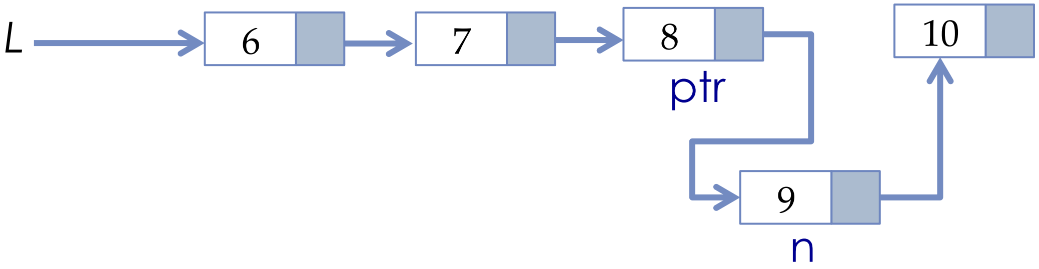
```
Node ptr = L;
while (ptr != null){
   if (ptr.data == target) {
          break;
   }
   ptr = ptr.next;
}
```

2. Create the new node and Insert after target 8



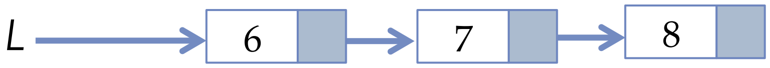
```
if (ptr != null){
   Node n = new Node(10,null);
   n.next = ptr.next;
   ptr.next = n;
}
```

Insert a new node 9 after the node that holds 8

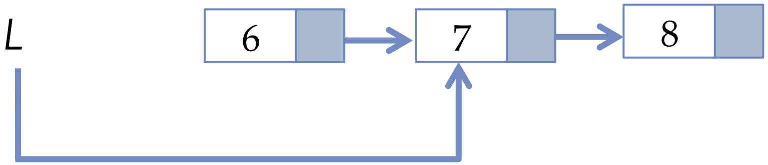


LL: Remove Front

To remove the first node from the list simply make the reference pointing to the first node point to the second node



```
L = L.next;
```



What happens to the node with the integer value of 6?

- It is garbage collected: since there are no references to this node, its memory is freed by the garbage collector

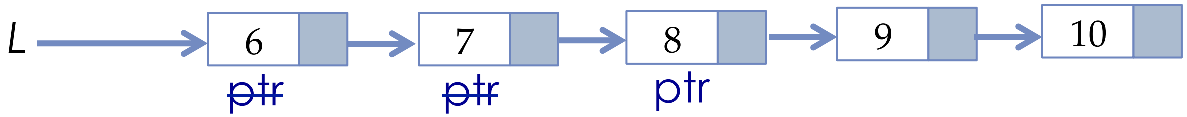
What is the running time?

- O(1): one assignment

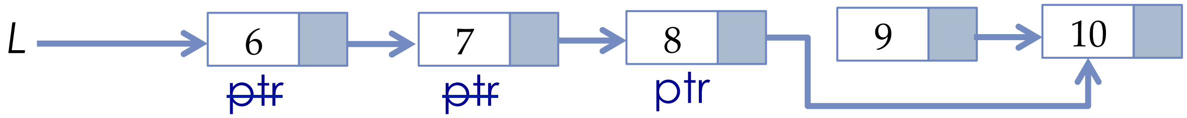
LL: Delete After Target

Delete the node after 8

1. Find target the 8



2. Remove the node after 8



```
ptr.next = ptr.next.next;
```

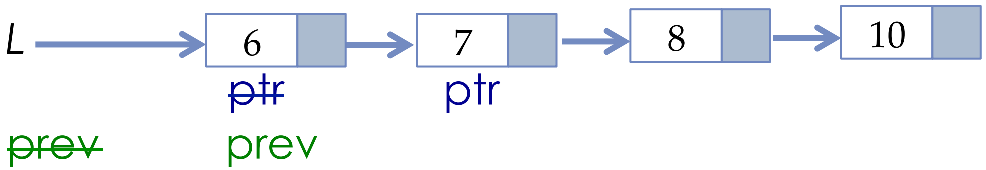
What is the running time?

- Best case if target is the first node: O(1)
- Worst case if target is the last node: O(n)

LL: Remove Target

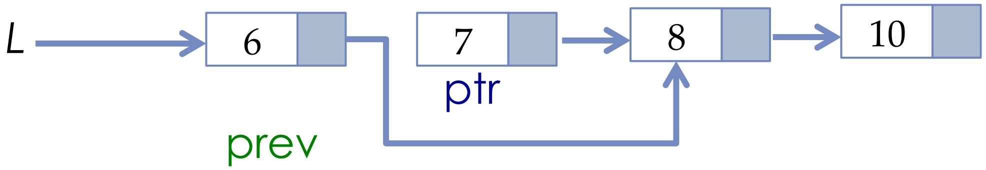
Remove the node that holds 7

1. Find the target 7. Need a handle to the node just before 7 (previous)



```
Node ptr = L, prev = null;
while (ptr != null){
   prev = ptr;
   ptr = ptr.next;
}
```

2. Make the node just before 7 (previous) point to the node just after 7



```
prev.next = ptr.next;
```

Three cases:

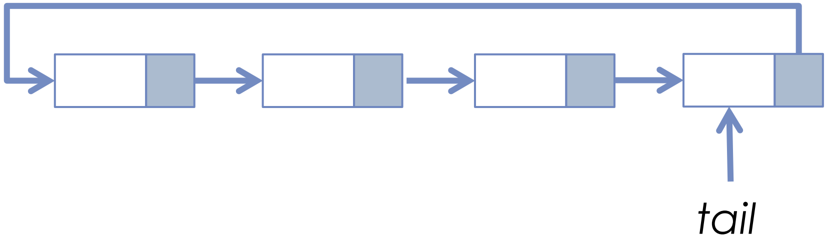
1. target not found (ptr == null)
2. target is the front of the list (ptr == L)
3. target is found and is not the front of the list

Running time

- Worst: O(n), Best: O(1)

Circular Linked List

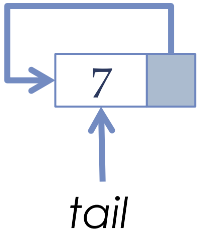
A linked list where the last node refers back to the first



By keeping a pointer to the last entry we have access to the *first* and *last* entry in constant time.
last: `tail`
first: `tail.next`

Add to front: two cases

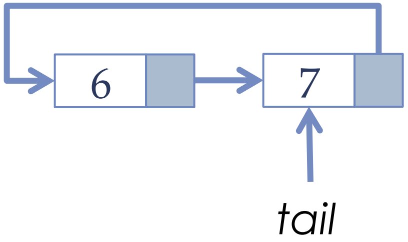
1. List is empty



```
Node n = new Node(7, null);
n.next = n;
tail = n;
```

Running time: O(1)

2. List is not empty



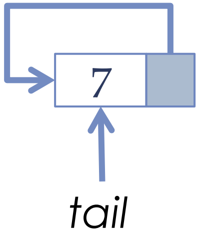
```
Node n = new Node(6, null);
n.next = tail.next;
tail.next = n;
```

CLL: Remove Front

Deletes the first element of the Circular Linked List

Three cases:

1. List is empty
2. One element

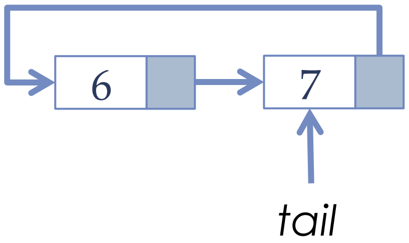


```
tail = null;
```

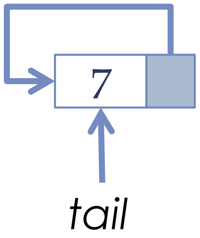
What is the running time?

- O(1)

3. More than one element



```
tail.next = tail.next.next;
```

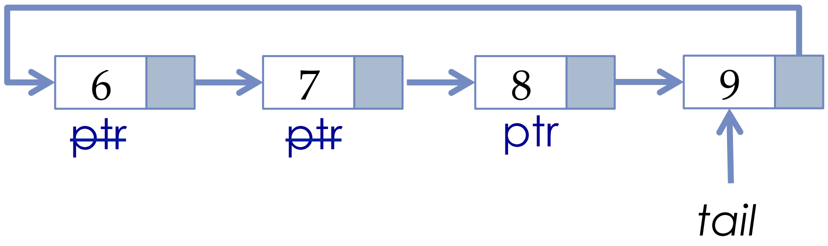


CLL: Search For a Target

To search in a CLL: say target is 8

1. Start a pointer at the front
2. Advance pointer until target is found or the beginning of the list is reached again.

```
Node ptr = tail.next;
do{
   if (ptr.data == target) {
          break;
   }
   ptr = ptr.next;
} while (ptr != tail.next);
```



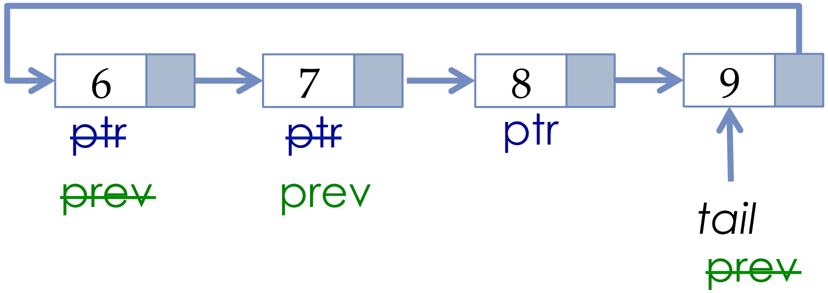
What is the running time?

- Worst: O(n)
- Best: O(1)

CLL: Delete Target

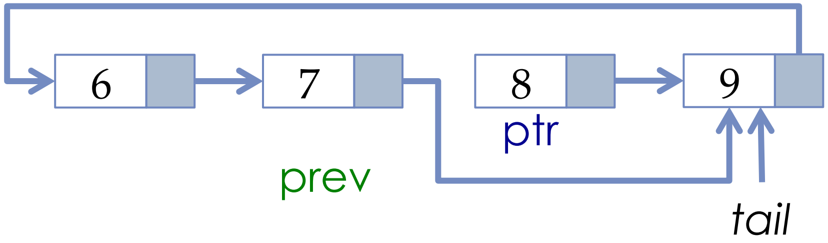
Removes the node with the target value

1. Find target



```
Node ptr = tail.next;
Node prev = tail;
do{
   if (ptr.data == target) {
          break;
   }
   prev = ptr;
   ptr = ptr.next;
} while (ptr != tail.next);
```

2. Delete target



```
prev.next = ptr.next;
```

What is the running time?

- Worst if removing the tail: O(n)
- Best if removing the front: O(1)

Three cases:

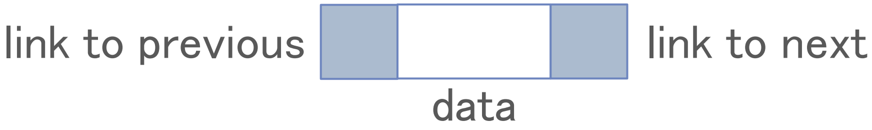
1. target not found (ptr == null)
2. target is the tail of the list (ptr == tail)
3. target is found and is not the tail of the list

Doubly Linked List

A linked list where every node refers to its previous and next nodes

Each node has three parts:

- a data part
- a link to the previous node
- a link to the next node

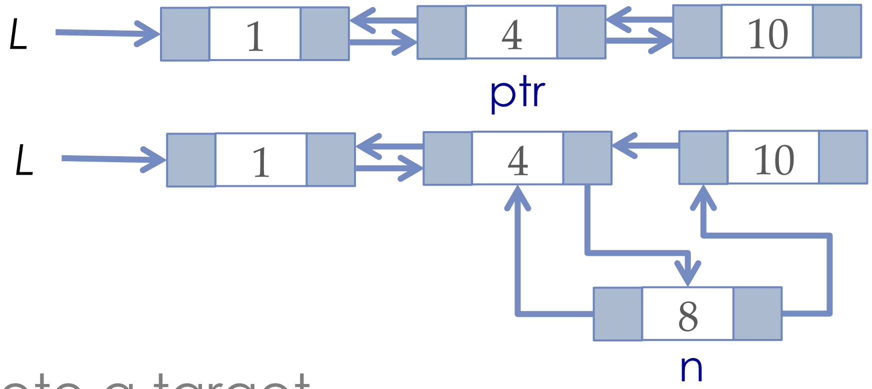


```
class Node {
   int data;
   Node prev;
   Node next;
   Node (int data, Node prev, Node next) {
         this.data = data;
         this.prev = prev;
         this.next = next;
   }
}
```

Doubly Linked List

Insert after a target

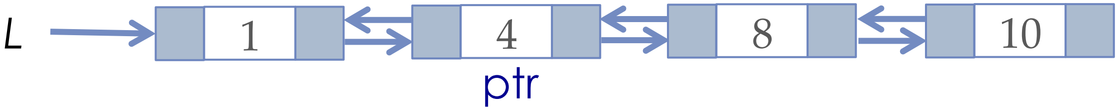
- Create the new node 8 and insert it after 4



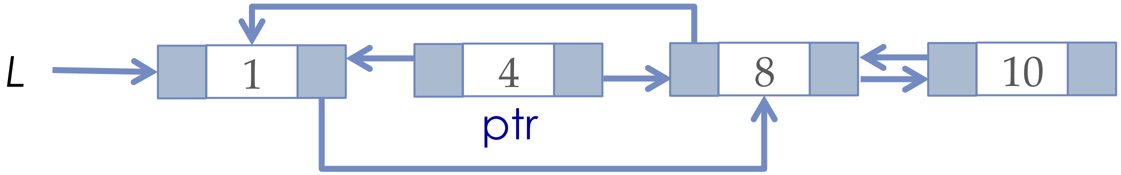
```
Node n = new Node(8,ptr,ptr.next);
ptr.next = n;
n.next.prev = n;
```

Delete a target

- Delete node 4



```
ptr.prev.next = ptr.next;
ptr.next.prev = ptr.prev;
```



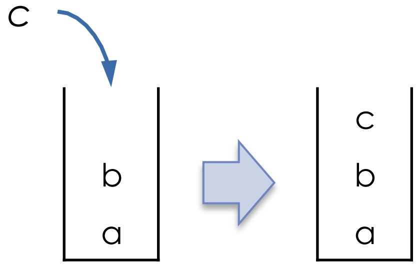
Stack

A stack is a collection with LIFO (Last In First Out) behavior

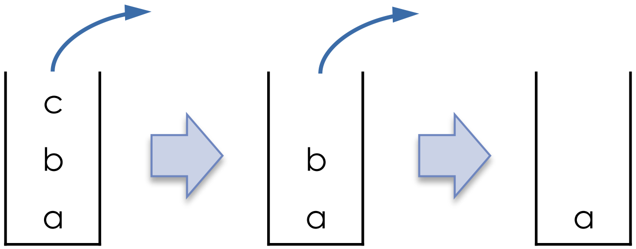
- It memorizes things and recalls in the reverse order (undo operation of the text editor)

Operations

- Push and pop are allusions to physical stacks. The order in which items are popped from the stack is the reverse of the order in which they were pushed onto the stack
- **Push**: add an entry to the top of the stack



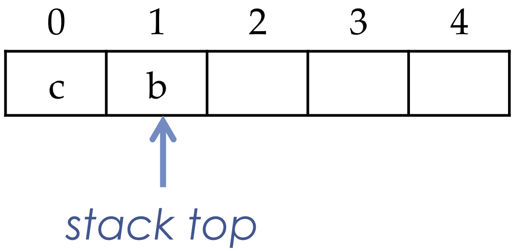
- **Pop**: removes an entry from the top of the stack



Stack Implementation

There are many ways to implement a stack, here are two:

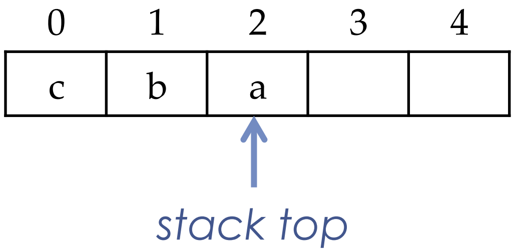
Array



0	1	2	3	4
c	b			

stack top

push a

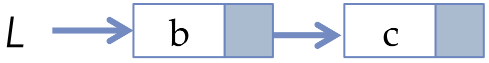


0	1	2	3	4
c	b	a		

stack top

Have to keep the index of the top of the stack

Linked List

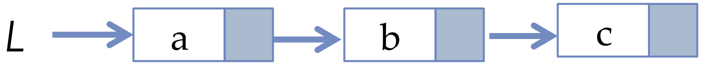


L

b

c

push a



L

a

b

c

Use addToFront to push into the stack and RemoveFront to pop the stack

What is the running time to find how many items in the stack?

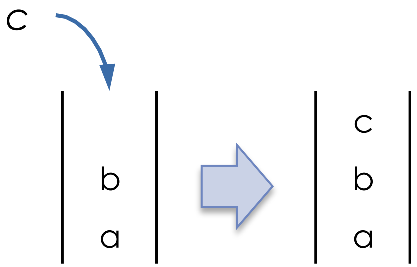
Queue

A queue is a collection with FIFO (First In First Out) behavior.

- It memorizes things and recalls them in the order they were inserted. It has a front and an end (tail).

Operations

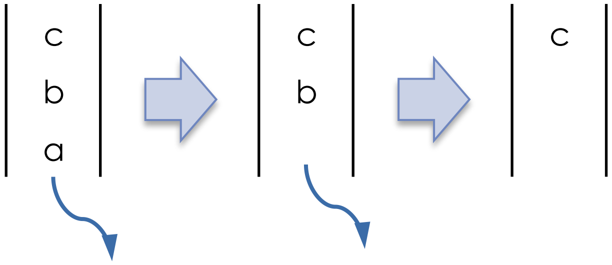
- **Enqueue**: add an entry to the end of the queue



We are going to see three ways to implement a queue:

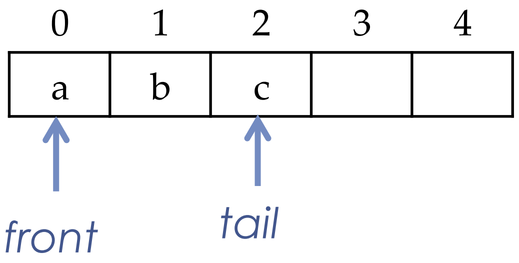
1. Array
2. Circular bounded array
3. Circular linked list

- **Dequeue**: remove an entry from the front of the queue



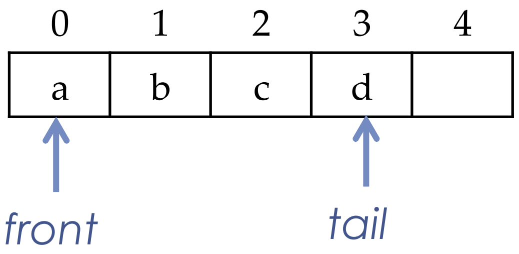
Queue Implementation: Array

Enqueue



0	1	2	3	4
a	b	c		

enqueue d

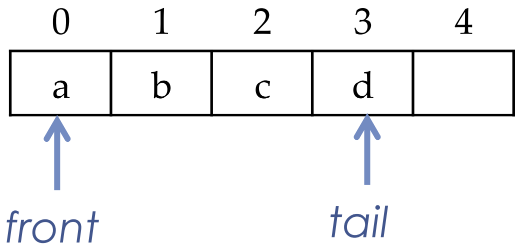


0	1	2	3	4
a	b	c	d	

How many items in the array?

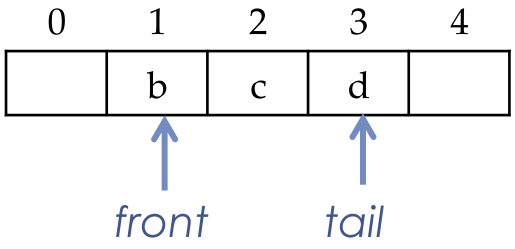
tail – front + 1

Dequeue



0	1	2	3	4
a	b	c	d	

dequeue



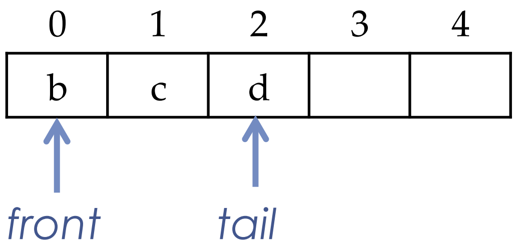
0	1	2	3	4
	b	c	d	

How to fill the empty spot?

Queue Implementation: Array

How to fill the empty spot after a dequeue?

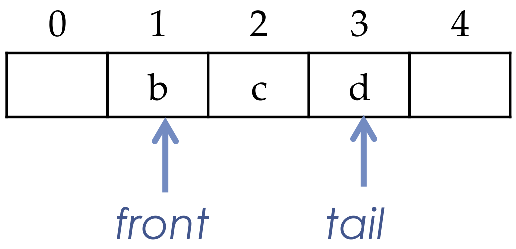
a. Move all the entries over by 1 positions



0	1	2	3	4
b	c	d		

Each dequeue would take linear time: very inefficient

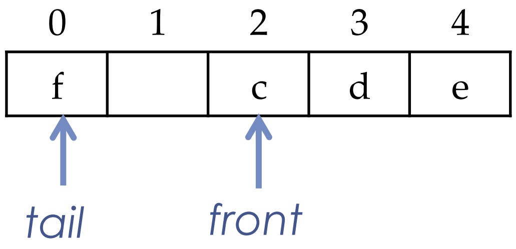
b. Leave the space empty



0	1	2	3	4
	b	c	d	

Each dequeue would take constant time but space is wasted

c. Use a circular bounded array (wraps around)



0	1	2	3	4
f		c	d	e

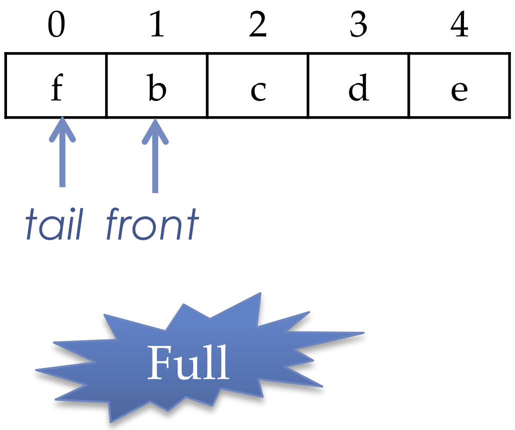
Leaves empty spaces but reclaims it

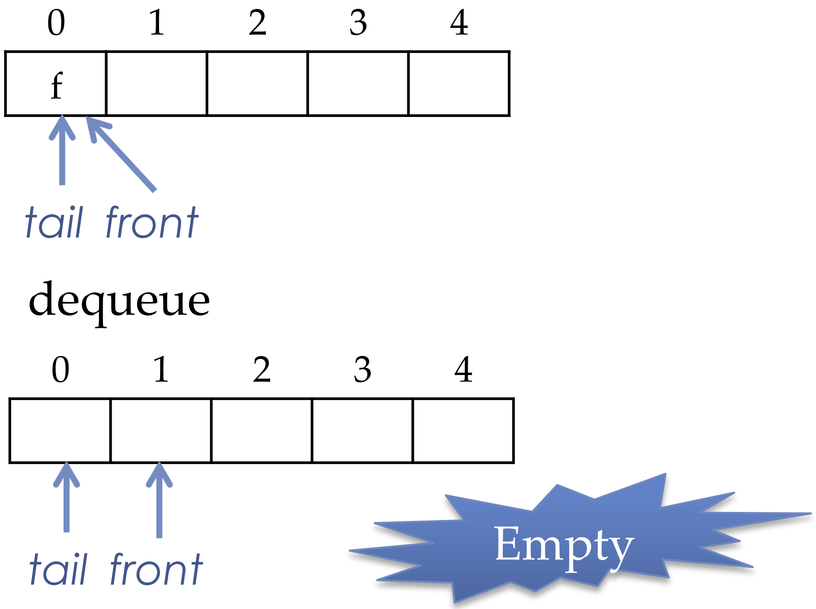
Queue Implementation: Array

Circular bounded array

- How many items?
  - Before: tail – front + 1
  - Now we can have tail < font index
    - tail < front: tail – front + 1
    - tail > front: size – (front – tail – 1)
- Is the queue empty or full?

The only way to know if it is empty or full is to keep the size of the queue.





Queue Implementation: CLL

Linked lists are more attractive when implementing queues

- Enqueue
  - add an element to the tail of the CLL
- Dequeue
  - remove the front item of the CLL