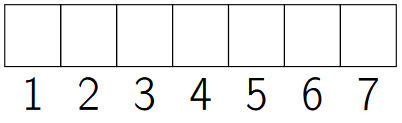
# Arrays and Linked Lists.

## Arrays.

1. https://www.coursera.org/learn/data-structures/lecture/OsBSF/arrays

**Array**: contiguous area of memory consisting of equal-size elements indexed by contiguous integers.

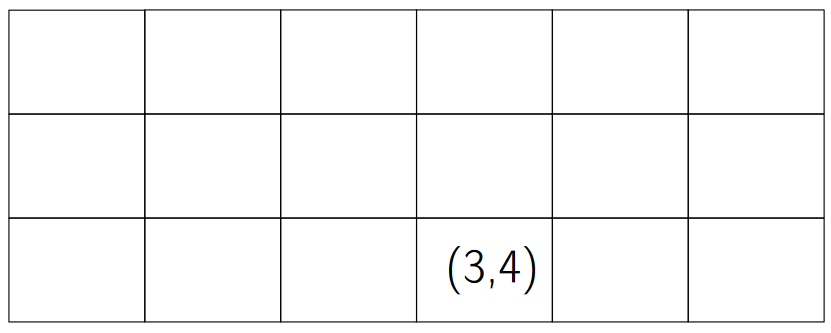


**Major advantage**: constant time access to read/write elements on any index.

*Constant-time access*

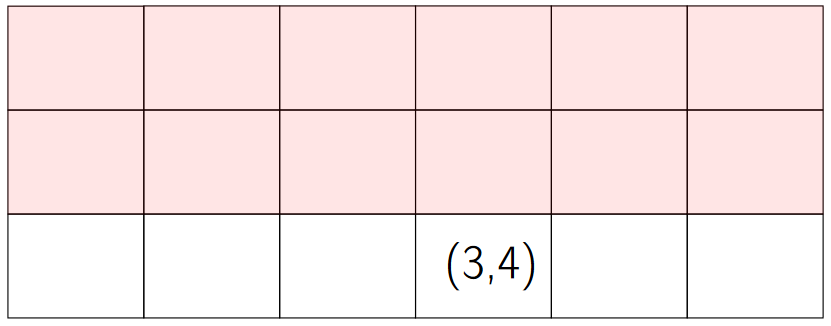
*array\_addr+elem\_size×(i−first\_index)*

### Multi-dimensional Arrays

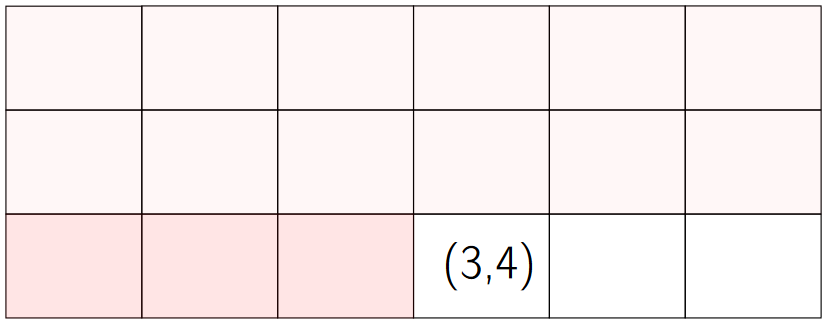


To find the address:

1. Skip the rows that aren’t being used.

(3−1)×6

1. Skip the elements before the index we’re searching for.

(3−1)×6+ (4−1)

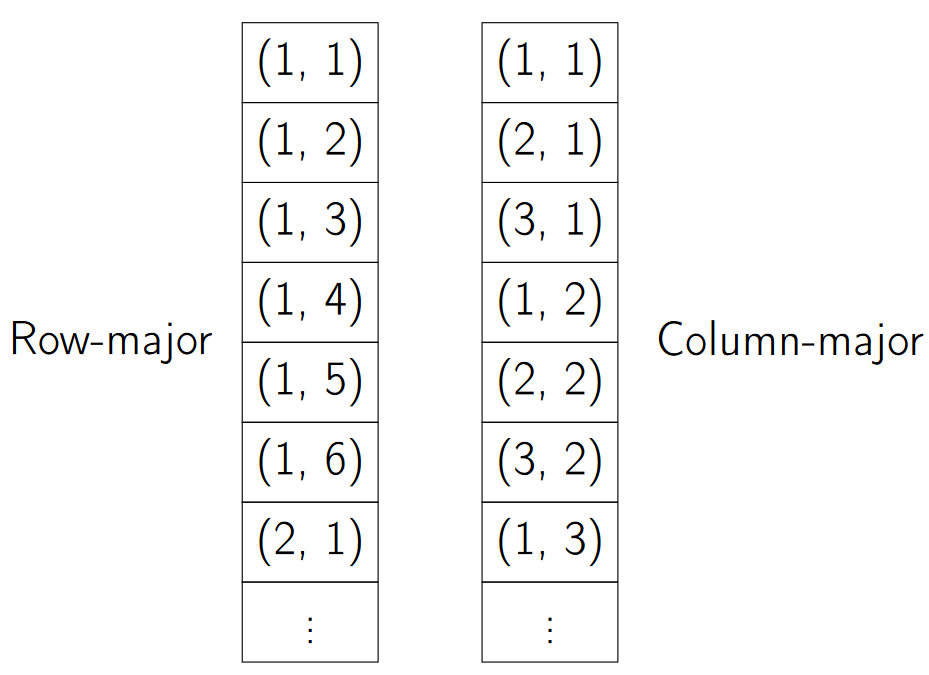
*elem\_size×((3−1)×6+ (4−1))*

1. Final .

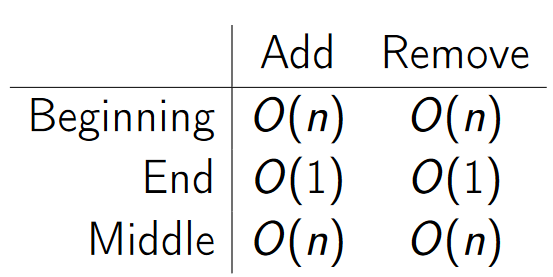
*array\_addr+elem\_size×((3−1)×6+ (4−1))*

### Ordering

1. **Row-major ordering**: the column number changes move rapidly.
2. **Column-major ordering**: the row number changes more rapidly.



### Times for common operations



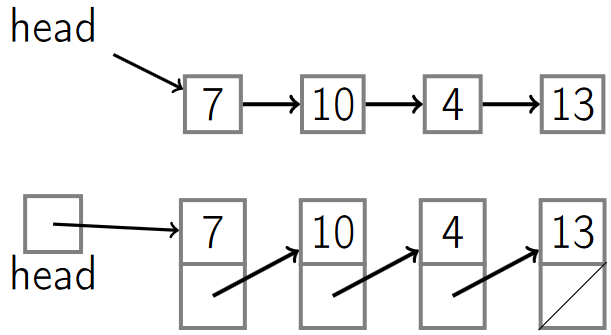
|  |  |  |  |
| --- | --- | --- | --- |
|  | **Add** | **Remove** | **Note** |
| **Beginning** | O (n) | O (n) | Constant time: to remove the first element, we remove it (the index becomes empty) and then we have to move the rest of the elements one index down. |
| **End** | O (1) | O (1) | Linear time: we only have to add and remove the last index. |
| **Middle** | O (n) | O (n) | Constant time. |

Note: arrays are great if we have to add/remove to the end, but it gets expensive if it’s in the beginning or in the middle.

## Singly-Linked Lists

1. https://www.coursera.org/learn/data-structures/lecture/kHhgK/singly-linked-lists

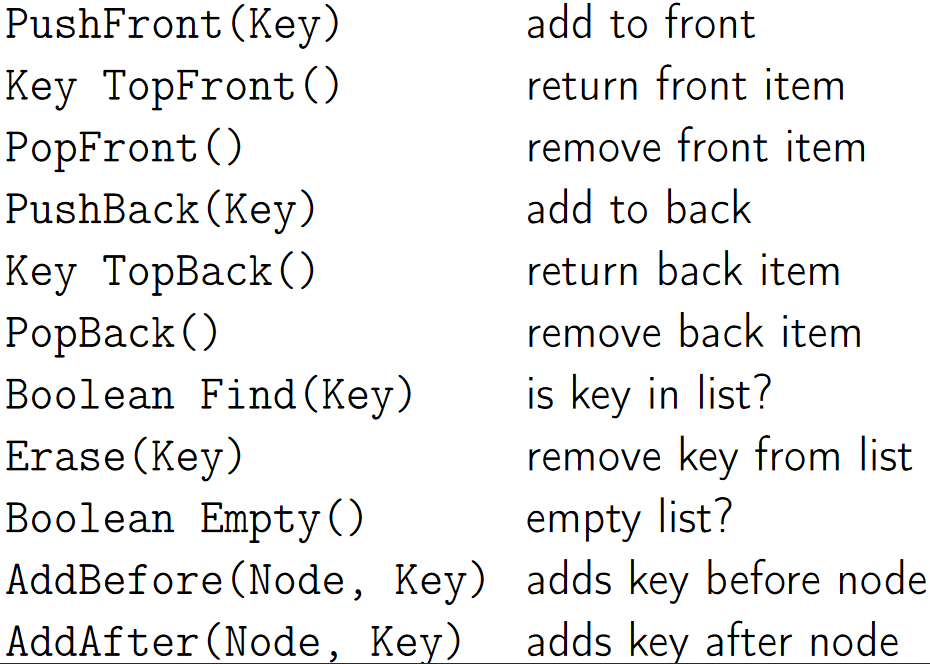
**Singly-linked lists**: they look like links in a chain, with a *head pointer* that points to a node, that then points to another node and so on.



Each node contains:

1. A key: the value (like an integer).
2. The next pointer.

### List operations

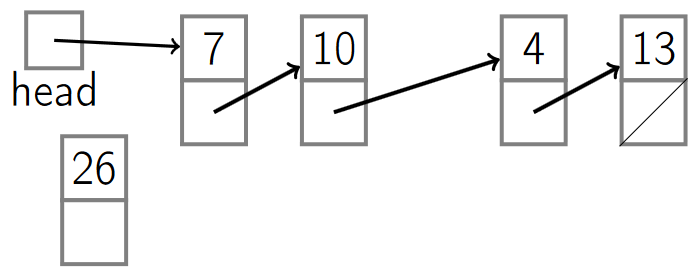


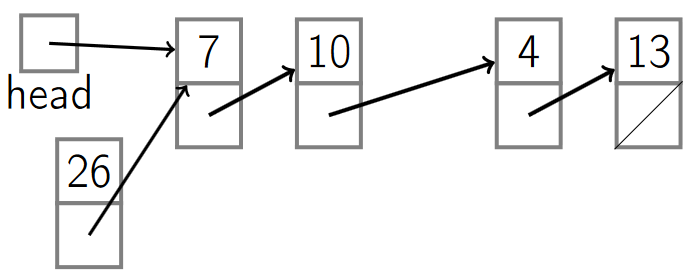
|  |  |  |
| --- | --- | --- |
| Push Front(key) | Add to front |  |
| Key TopFront() | Return front item |  |
| PopFront() | Remove front item |  |
| PushBack(key) | Add to back | Also known as *append* |
| Key TopBack() | Return back item |  |
| PopBack() | Remove back item |  |
| Boolean Find(key) | Check if key is in list |  |
| Erase(key) | Remove key from list |  |
| Boolean Empty() | Check if the list is empty |  |
| AddBefore(Node, key) | Adds key before a node |  |
| AddAfter(Node, key) | Adds key after a node |  |

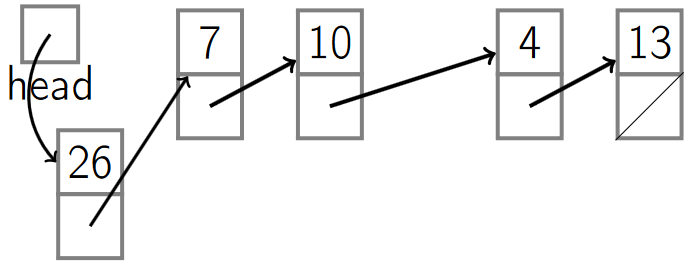
### Times for some operations

#### PushFront O(1)

1. Create the node with the new key.
2. Update the pointer of the new node to point to the head.
3. Update the head pointer to point to the new node.

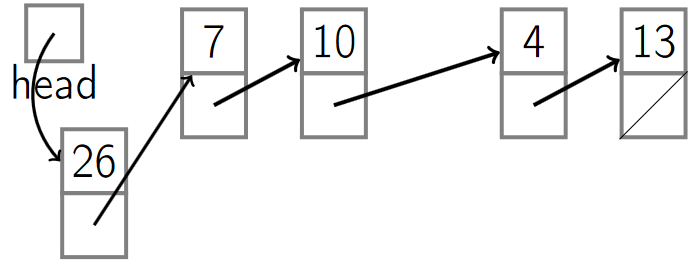


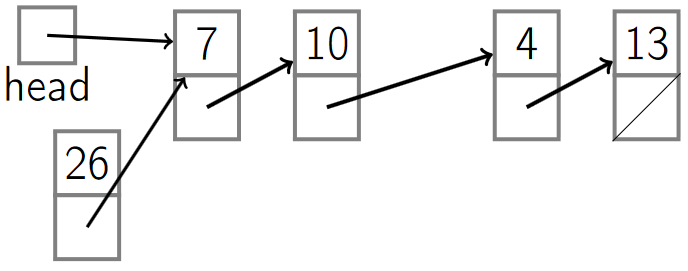


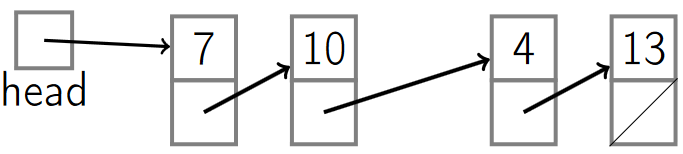


#### PopFront O(1)

1. Update the head pointer.
2. Remove the node.







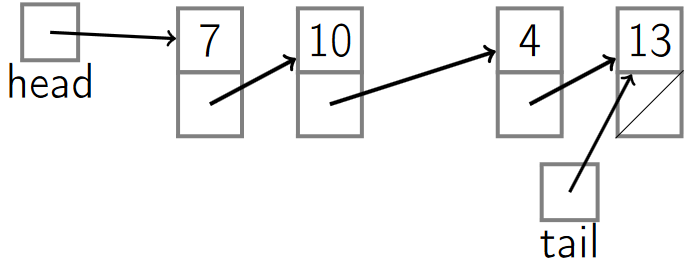
#### PushBack O(n) (no tail)

1. Start at the head and walk down the list unitl we get to the end.
2. Add the node to the end of the list.

#### PopBack O(n) (no tail)

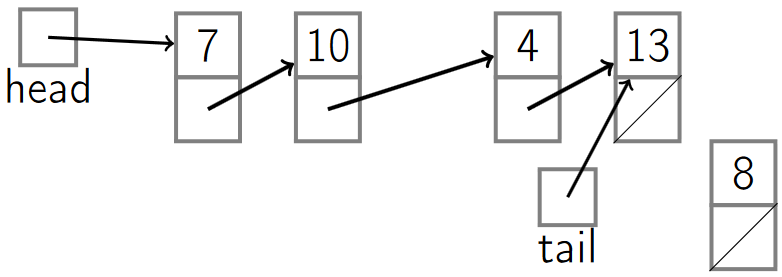
1. Start at the head and walk to the end of the list.
2. Remove the last element.

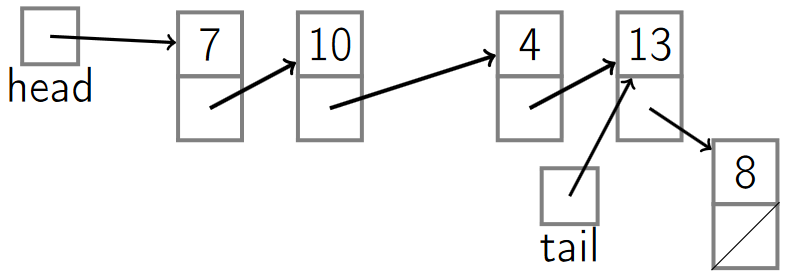
**Note**: adding a *tail pointer* is like having a *head pointer* but it points to the end of the list.

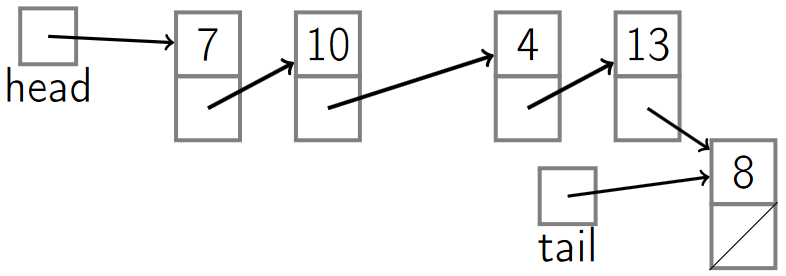


#### PushBack O(1) (with tail)

1. Create a node with a new key and its pointer set to *null* (because it will be the last element).
2. Update the next pointer of the current tail to point to the new tail.
3. Update the tail pointer.

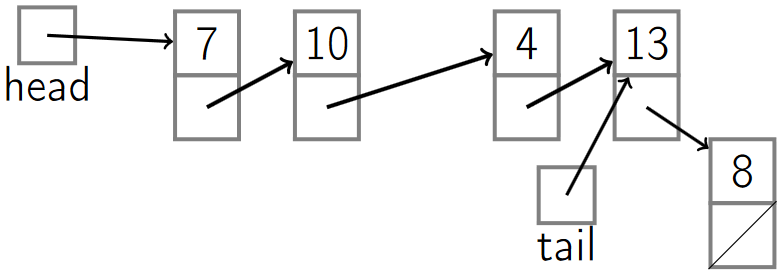


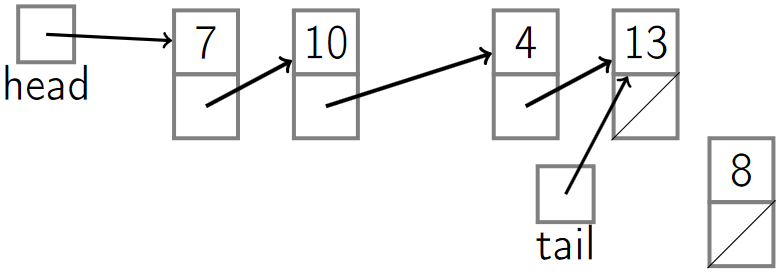


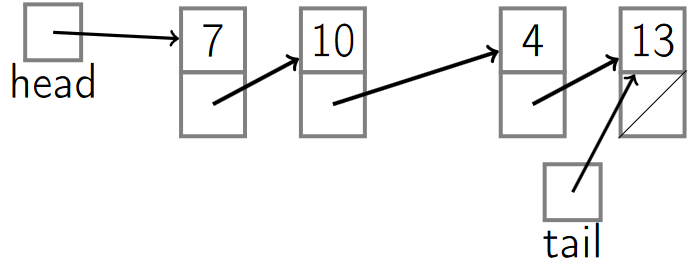


#### PopBack O(n) (with tail)

1. Go to the tail.
2. Update the tail to point to the penultimate element of the list.
   1. Because we have a pointer from the penultimate element to the last one, but we don’t have one from the last one to the penultimate element, we have to start at the beginning of the list and walk all the way down until we find the penultimate element (expensive) that points to the current tail, and then update the tail.
3. Update the pointer of the new tail (last element) to poin to null.
4. Remove the element.







### Singly-Linked List Pseudocode

#### PushFront(key)

|  |  |
| --- | --- |
| node←new node  node.key←key | *Create a new node with its key.* |
| node.next←head | *Set the next element of the newly created node to point to the old head.* |
| head←node | *Update the current head pointer.* |
| if tail=nil:  tail←head | *If the tail isn’t null, that means that before the insertion it was an empty list, so we need to update its tail.* |

#### PopFront()

|  |  |
| --- | --- |
| if head = nil:  ERROR: empty list | *Check if the list is empty.* |
| head←head.next | *Update the head to point now to the next head.* |
| if head = nil:  tail←nil | *If there was only one element in the list, then there’s no more elements.*  *Check if the new head is null and if so, update the tail to be null.* |

#### PushBack(key)

|  |  |
| --- | --- |
| node←new node  node.key←key | *Create a new node with its key.* |
| node.next = nil | *Set its next pointer to null.* |
| if tail = nil:  head←tail←node | *Check the current tail. If it’s null, then it’s an empty list so we update the tail and the head to point to the new node.* |
| else:  tail.next←node | *Otherwise, update the old tail’s next pointer to point to the new node.* |
| tail←node | *Update the tail to point to the new node.* |

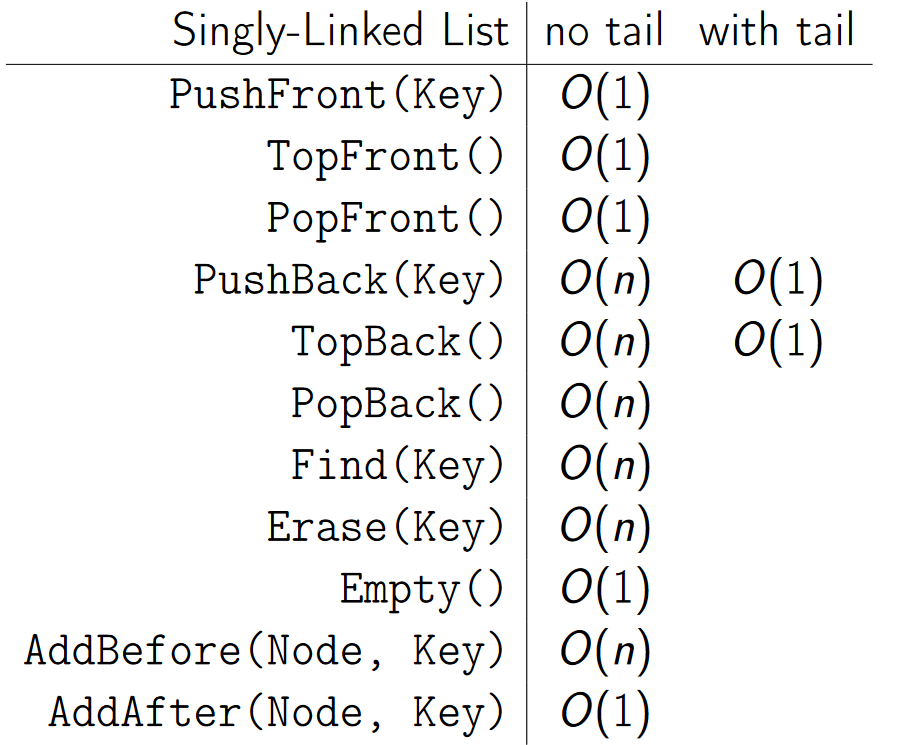
#### PopBack()

|  |  |
| --- | --- |
| if head= nil:  ERROR: empty list | *If the list is empty throw an error.* |
| if head=tail:  head←tail←nil | *If the head is equal to the tail, then that means there’s only one element on the list, so we update the tail and head to null.* |
| else: | *Otherwise, it means there’s more than one element.* |
| p←head | *Start at the head (p).* |
| while p.next.next ̸= nil:  p←p.next | *Work our way down, trying to find the next to last element. When we exit the loop, ‘p’ will be the next to last element.* |
| p.next←nil | *Update p’s next pointer to null.* |
| tail←p | *Set the tail equal to p.* |

#### AddAfter (*node*, *key*)

|  |  |
| --- | --- |
| node2←new node  node2.key←key | *Create a new node with its key.* |
| node2.next = node.next | *Set the new node’s next pointer to whatever node we’re adding after (the previous node).* |
| node.next = node2 | *Update the node pointer of the one we’re adding after so that it points to the new node.* |
| if tail = node:  tail←node2 | *In case the node we’re adding after was the tail, update the tail to point to the new node.* |

### Time cost of operations

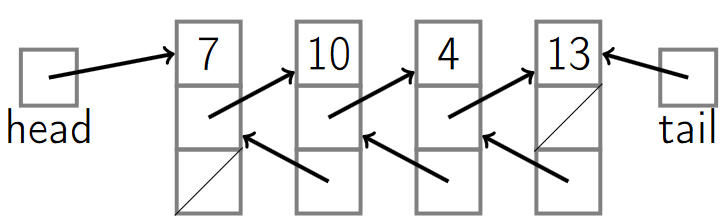


|  |  |  |
| --- | --- | --- |
| **Singly-Linked List** | **no tail** | **with tail** |
| PushFront(Key) | O(1) |  |
| TopFront() | O(1) |  |
| PopFront() | O(1) |  |
| PushBack(Key) | O(n) | O(1) |
| TopBack() | O(n) | O(1) |
| PopBack() | O(n) |  |
| Find(Key) | O(n) |  |
| Erase(Key) | O(n) |  |
| Empty() | O(1) |  |
| AddBefore(Node, Key) | O(n) |  |
| AddAfter(Node, Key) | O(1) |  |

## Doubly-Linked Lists

1. https://www.coursera.org/learn/data-structures/lecture/jpGKD/doubly-linked-lists

**Doubly-linked lists**: adds a way to get go back on a list (by adding another pointer pointing to the previous node).



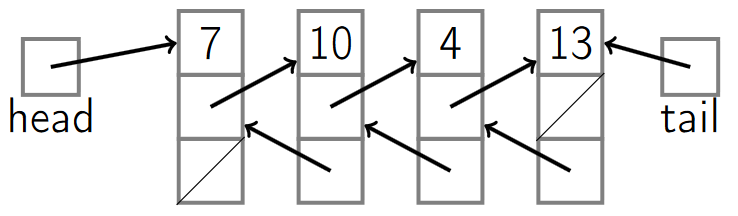
Each node contains:

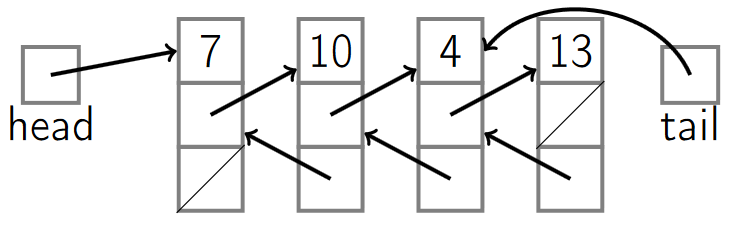
1. A key: the value (like an integer).
2. The next pointer.
3. The previoust pointer.

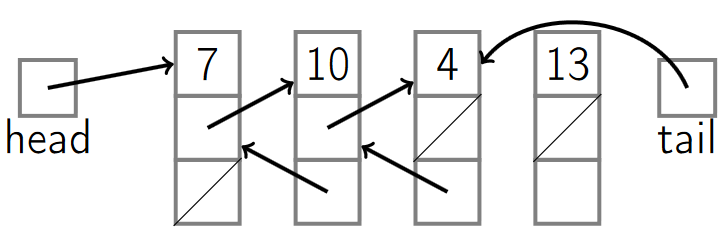
### Doubly-Linked List Pseudocode.

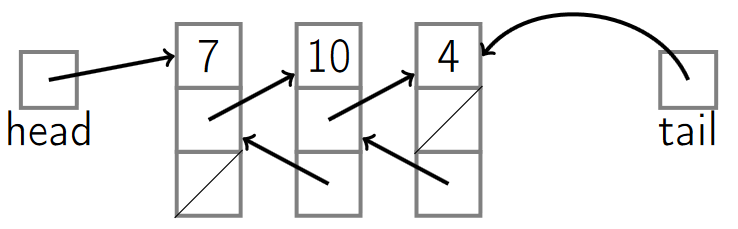
#### PopBack O(1)

1. Update the tail pointer to point to the previous element.
2. Update its next pointer to be null.
3. Remove the last element.



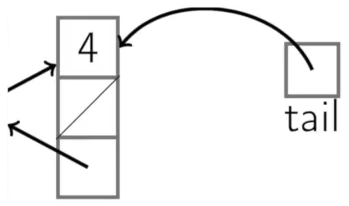
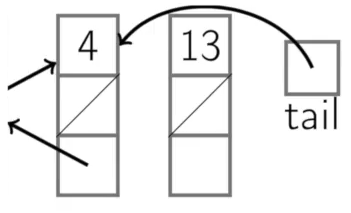
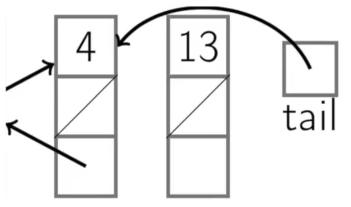






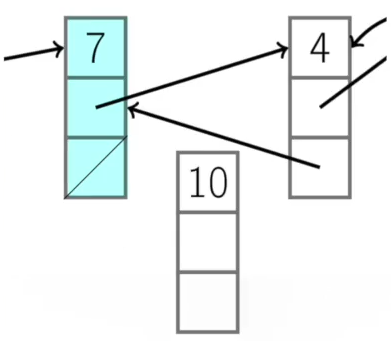
|  |  |
| --- | --- |
| if head= nil:  ERROR: empty list | *Check if the list is empty.* |
| if head=tail:  head←tail←nil | *If there’s only one element on the list, set the tail and head to null.* |
| else: | *Otherwise, there’s more than one element.* |
| tail←tail.prev | *Update the tail to be the next to last element.* |
| tail.next←nil | *Update the next pointer of the tail to null.* |

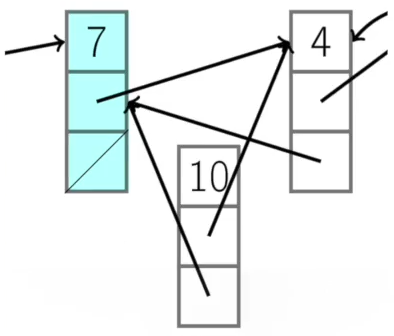
#### PushBack(key) O(1)

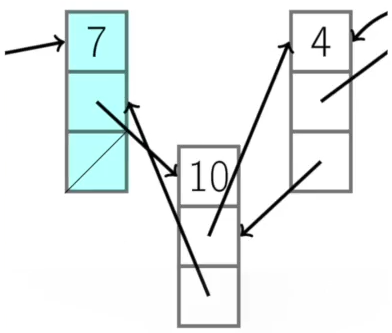
 

|  |  |
| --- | --- |
| node←new node  node.key←key | *Create a new node with its key.* |
| node.next = nil | *Set its next pointer to null.* |
| if tail = nil:  head←tail←node  node.prev←nil | *Check the current tail. If it’s null, then it’s an empty list so we update the tail and the head to point to the new node.* |
| else: | *List is not empty.* |
| tail.next←node | *Update the tail’s next pointer to the new node.* |
| node.prev←tail | *Update the previous pointer of the node to point to the old tail.* |
| tail←node | *Update the tail to point to the new node.* |

#### AddAfter (*node*, *key*)

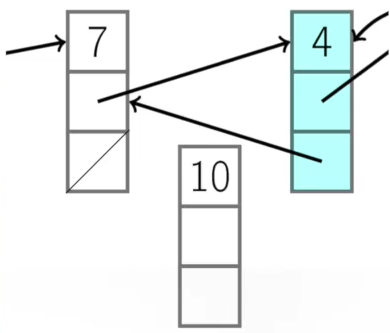


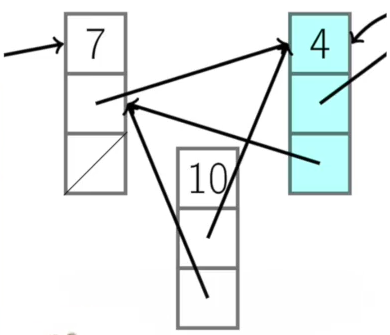


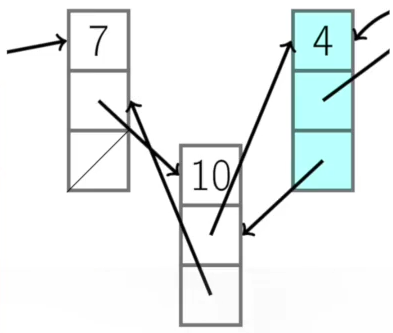


|  |  |
| --- | --- |
| node2←new node  node2.key←key | *Create a new node with its key.* |
| node2.next←node.next  node2.prev = node | *Update pointers.* |
| node.next = node2  if node2.nex t̸= nil:  node2.next.prev←node2 | *Maintain the previous pointer.* |
| if tail = node: |  |
| tail←node2 |  |

#### AddBefore (*node*, *key*)

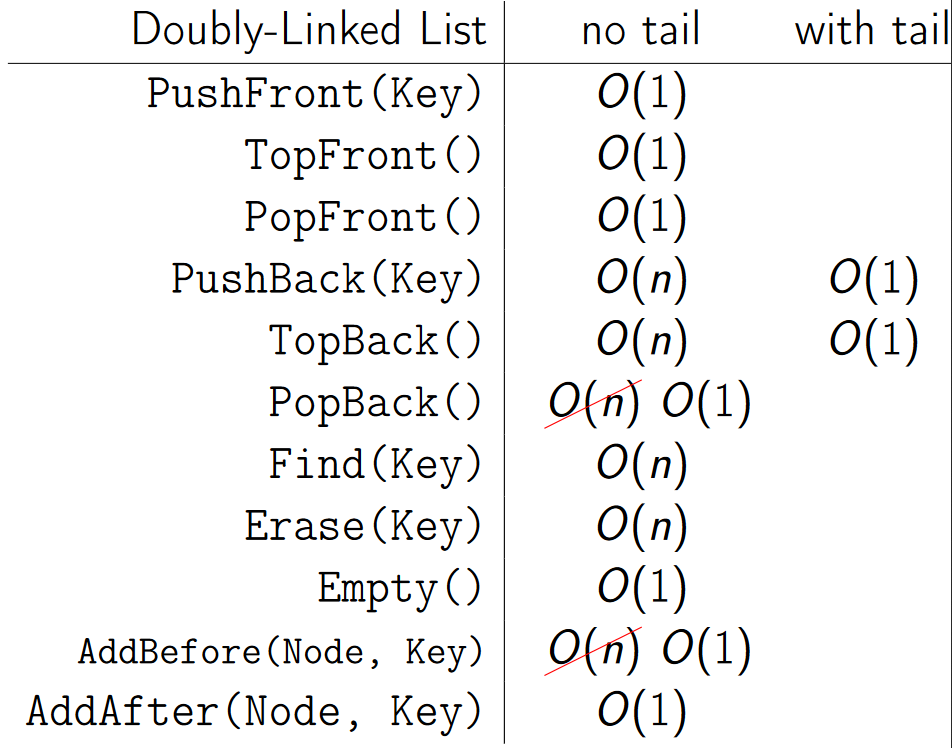






|  |  |
| --- | --- |
| node2←new node  node2.key←key | *Create a new node with its key.* |
| node2.next←node  node2.prev←node.prev | *Update pointers.* |
| node.prev←node2  if node2.prev ̸= nil:  node2.prev.next←node2 | *Maintain the previous pointer.* |
| if head = node: |  |
| head←node2 |  |

### Time cost of operations



|  |  |  |
| --- | --- | --- |
| **Doubly -Linked List** | **no tail** | **with tail** |
| PushFront(Key) | O(1) |  |
| TopFront() | O(1) |  |
| PopFront() | O(1) |  |
| PushBack(Key) | O(n) | O(1) |
| TopBack() | O(n) | O(1) |
| PopBack() | ~~O(n)~~ O(1) |  |
| Find(Key) | O(n) |  |
| Erase(Key) | O(n) |  |
| Empty() | O(1) |  |
| AddBefore(Node, Key) | ~~O(n)~~ O(1) |  |
| AddAfter(Node, Key) | O(1) |  |

## Arrays vs. Linked-Lists

Find:

* Array is constant time to access any element (binary search is very efficient).
* Linked-list is linear time, finding an element is expensive (we have to start at the head or the tail).

Insert/remove at the front:

* Arrays take linear time to insert (we have to move the remaining elements).
* Linked-lists are constant time to insert/remove at the front.

Insert/remove at the back:

* Arrays take constant time.
* Doubly-Linked-lists with tail take constant time.

Insert/remove arbitrary elements:

* Arrays take linear time (find, remove and move the remaining elements).
* Doubly-linked lists take constant time to insert between nodes or remove a node.

Contiguous:

* Arrays need to be contigous in memory.
* List elements don’t need to be contiguous (they can be in separatedly allocated locations in memory and pointer pointing to the location).

# Stacks and Queues

## Stacks

1. https://www.coursera.org/learn/data-structures/lecture/UdKzQ/stacks

**Stack**:

* ***LIFO*** (*Last In First Out*) data structure (also known as *LIFO* queues) (abstract data type).
* Can be implemented with either an **array** or a **linked list**.
* Each stack operation is **O(1)**: Push,Pop, Top, Empty.
* Example: a stack of books, we can take the top element, and put another element at the top, but we can’t take the element at the bottom.

It has the following operations.

* **Push(Key)**: adds key to collection.
* **Key Top()**: returns mostrecently-added key.
* **Key Pop()**: removes and returns most recently-added key.
* **Boolean Empty()**: are there any elements?

### Example

A stack allows to check if an expression has balanced brackets:

* Input: A string str consisting of ‘(‘, ‘)’, ‘[‘,‘]’ characters.
* Output: Return whether or not the string’s parentheses and square brackets are balanced.

Example:

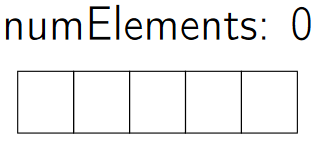
* Balanced:“([])[]()”,“((([([])]))())”
* Unbalanced:“([]]()”“][”

#### IsBalanced(str)

|  |  |
| --- | --- |
| Stack *stack* | *Create a stack.* |
| for *char* in *str*: | *Go through every character in the string.* |
| if *char* in [‘(‘, ‘[‘]:  *stack*.Push(*char*) | *If we have an opening character,*  *push the character to the stack.* |
| else: | *If it isn’t an opening character,* |
| if *stack*.Empty():  return False | *Check if the stack is empty, if it is then it isn’t balanced and we return False.* |
| *top*←*stack*.Pop() | *Pop the top element off,* |
| if (*top* = ‘[‘ and *char* != ‘]’) or (*top* = ‘(‘ and *char* != ‘)’):  return False | *Check if the top element matches the character that we got (so if the top is a ‘[’ and we read a ‘]’ then it matches) and we continue.* |
| return *stack*.Empty() | *Once we’ve run through all of the characters, we check again if the stack is empty.* |

### Stack Implementation with Array.

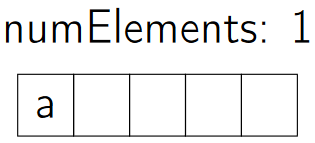
Created by allocating an array of some maximum stack size (example: 5).



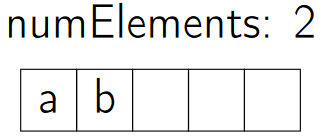
* **Advantage of using an array**:
  + All operations are done in constant time *O(1)*.
* **Disadvantages of using an array**:
  + We have a maximum size based on the array we initially allocated.
  + Potential wasted space (unused indices).

#### Push O(1)

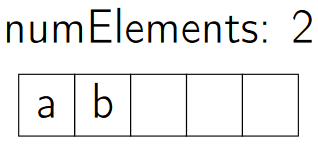
1. When we *push* an element (example: a), we put it at the end of the array we got so far.



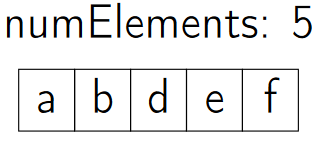
* Pushing (or *appending*) another element (b) would put it at the end of the array.



1. The stack would like this:

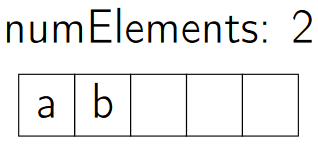


**Note**: pushing more elements than the original array allows would throw an error.



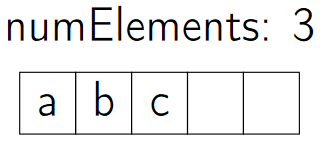
#### Top O(1)

1. Returns the top element of the array (would return b) but it *doesn’t remove the element*.



#### Pop O(1)

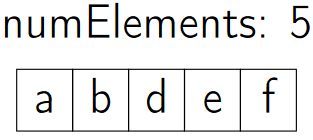
1. Returns and removes the last element of the stack (example: c).

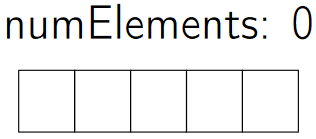


**Note**: popping when the array is empty would return an error.

#### Empty O(1)

1. Check whether the number of elements is greater than zero.



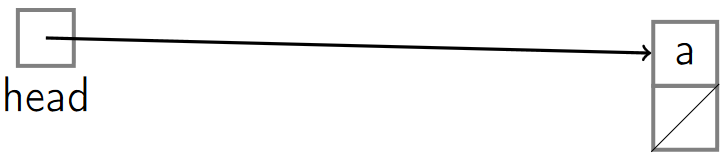


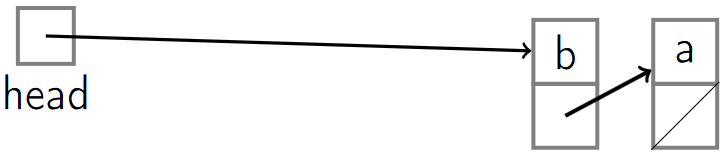
### Stack Implementation with Linked-List.

* **Advantage of using a linked list**:
  + All operations are done in constant time *O(1)*.
  + There’s no limit to the amount of elements that can be added (as long as there’s available memory).
  + No wasted space because only the used space is allocated.
* **Disadvantages of using a linked list**:
  + Using array, each element size is just big enough to store the key, using a linked-list we also have to store a pointer.

#### Push O(1)

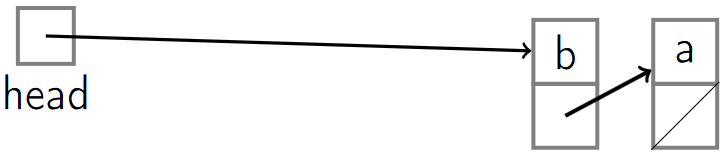
1. Pushing an elements pushes it to the front (*PushFront()*).
2. The head is updated to point to the newly pushed element.





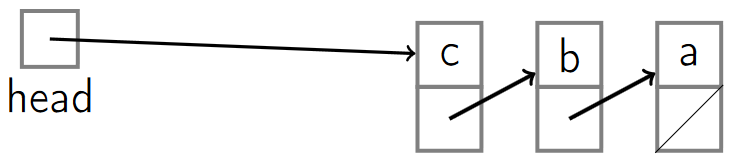
#### Top O(1)

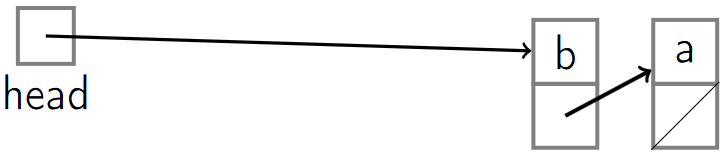
1. Returns the top element of the array (would return b) but it *doesn’t remove the element*. (*TopFront()*).



#### Pop O(1)

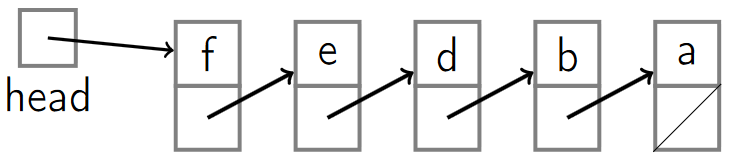
1. Returns and removes the last element of the stack (example: c). (*TopFront(), PopFront()*).

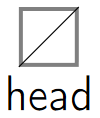




#### Empty O(1)

1. Check if the head points to *null*.





## Queues

1. https://www.coursera.org/learn/data-structures/lecture/EShpq/queues

**Queue**:

* ***FIFO*** (*First In First Out*) data structure (abstract data type).
* Each queue operation is **O(1)**: Enqueue, Dequeue, Empty.
* Can be implemented with either a linked list (with tail pointer) or an array (with a read and a write indices).
  + An array implementation:
    - Has a maximum size that the queue can grow to.
    - Any unused indices are wasted space.
  + A linked list implementation:
    - Can get as large as needed as long as there’s available memory.
    - More space is needed, because we have to allocate space for both the key and the pointer.
* Example: waiting in line at the supermarket.

It has the following operations.

* **Enqueue(Key)**: adds key to collection
* **Key Dequeue()**: removes and returns least recently-added key
* **Boolean Empty()**: are there any elements?

### Queue Implementation with Linked-List.

Operations:

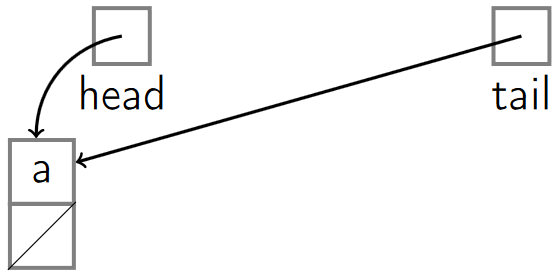
* **Enqueue**: uses *List.PushBack*
* **Dequeue**: uses *List.TopFront* and *List.PopFront*
* **Empty**: uses *List.Empty*

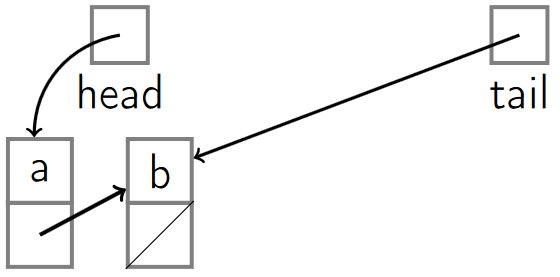
#### Enqueue O(1)

1. Push to the back of the linked list (uses *List.PushBack*)



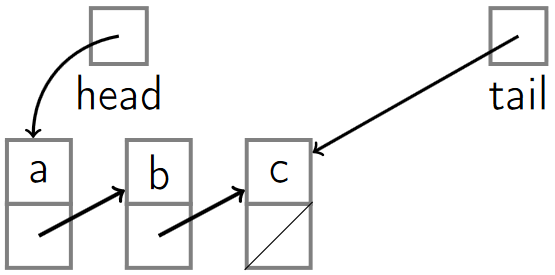
2. If there’re no elements in the queue, both the tail and the head’s pointer are set to the newly enqueued node.

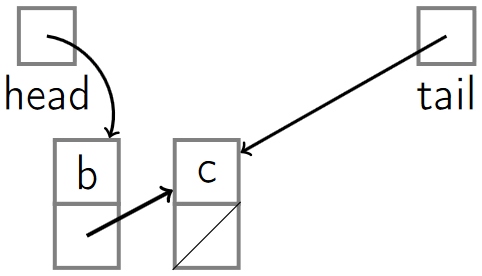




#### Dequeue O(1)

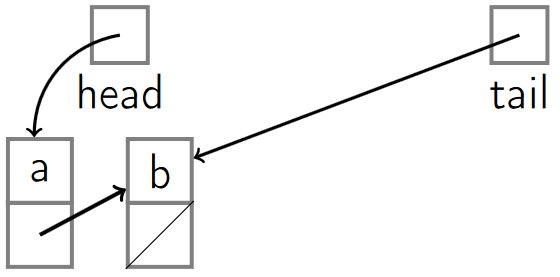
1. Removes the first element of the list, where the head points to (uses *List.TopFront* and *List.PopFront*).





#### Empty O(1)

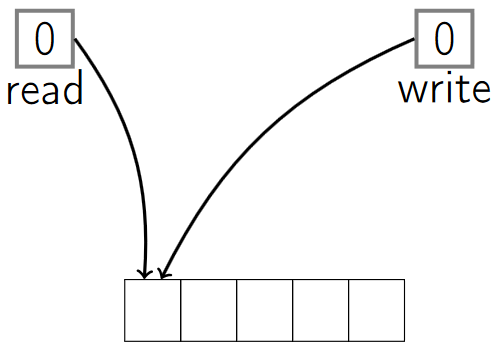
1. Check if the head and tail both point to *null* (uses *List.Empty*).





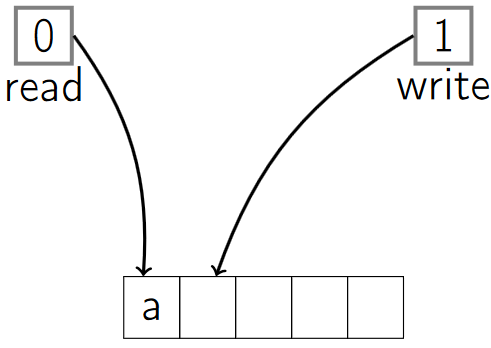
### Queue Implementation with Array.

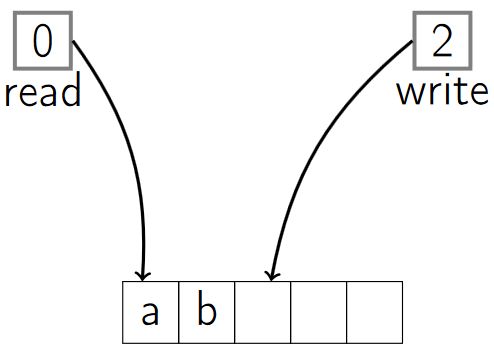
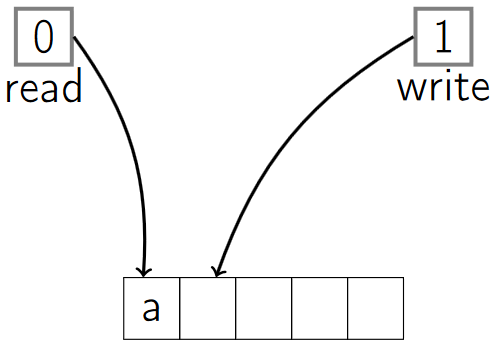
We need to keep track of the array as a *circular array* using a *read and write indices*.

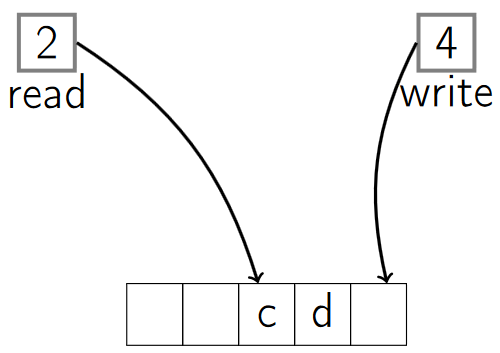
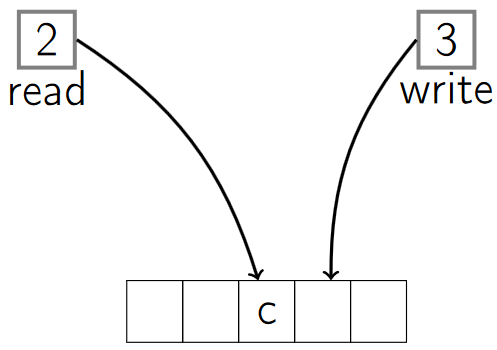


#### Enqueue O(1)

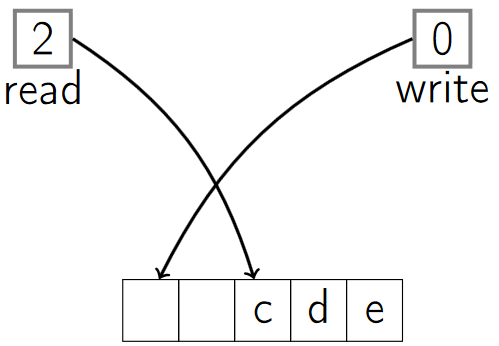
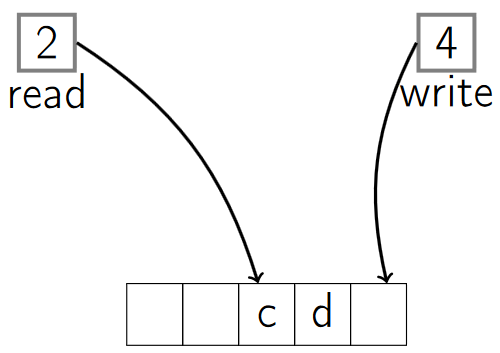
1. The *write index* tells us where the next *Enqueue* operation should happen.
2. When *enqueuing*, we read what’s at the *write index* and then we increment the *write index* by 1.



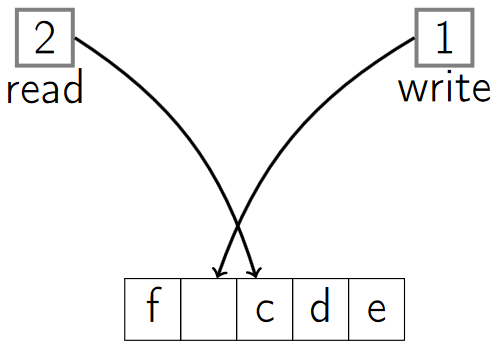




1. When the *write index* reaches the length of the array, we restart it to the first index (0).

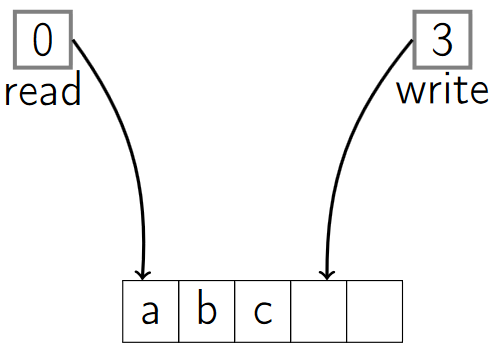


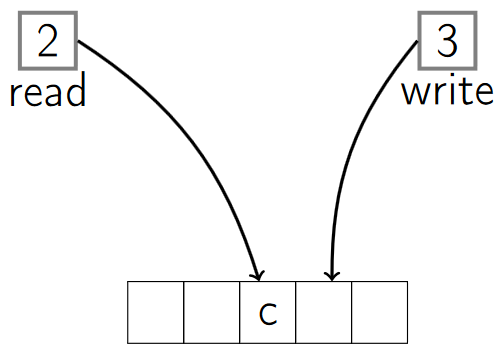
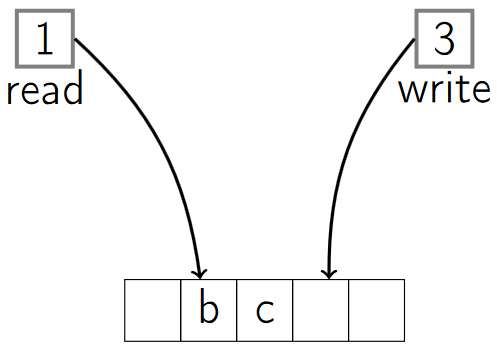
1. The *read* and *write indices* can’t be both the same number (we have a buffer of at least one element that can’t be written to, to make sure read and write are separate and distinct if the queue’s not empty).



#### Dequeue O(1)

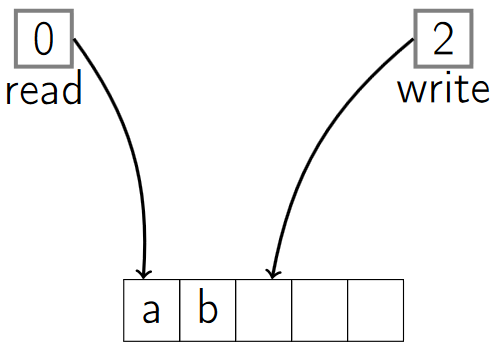
1. Expensive operation, we would need to move the elements any time the top is removed.
2. The *read index* tells us where the next *Dequeue* operation should happen.
3. When *dequeueing*, we read what’s at the *read index* and then increment the *read index* by 1.

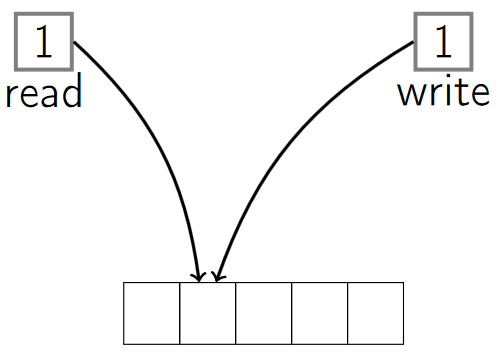




#### Empty O(1)

1. We check whether the *write index* is equal to the *read index* , if not then there’s something to *dequeue* that has been *enqueued*.





# Trees

## Walking a tree