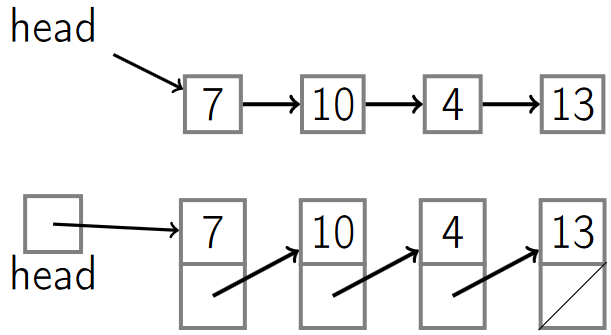
## Singly-Linked Lists

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

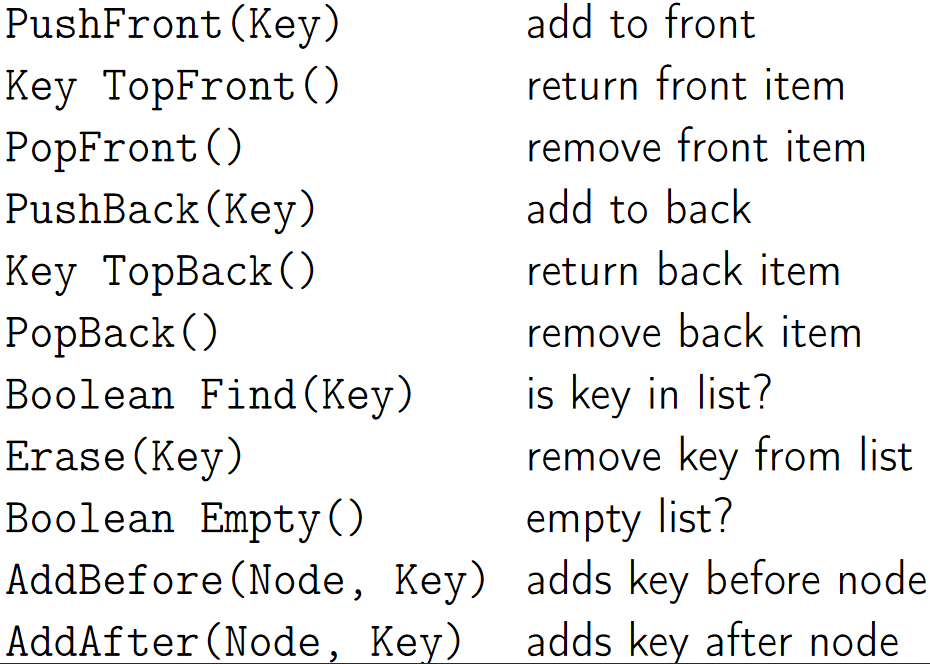
**Singly-linked lists**: they look like links in a chain, with a *head pointer* that points to a node, that then points to another node and so on.



Each node contains:

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

### List operations

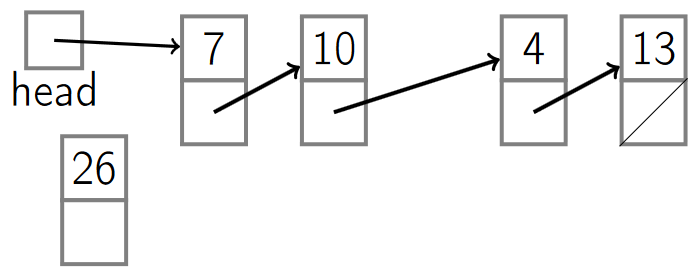


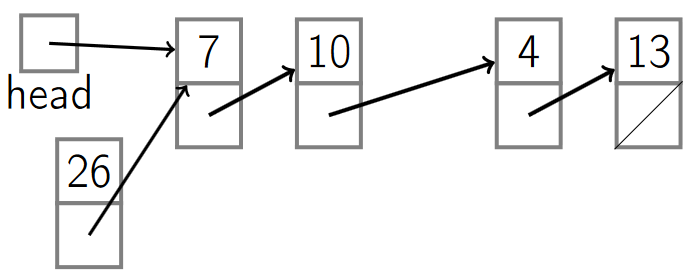
|  |  |  |
| --- | --- | --- |
| Push Front(key) | Add to front |  |
| Key TopFront() | Return front item |  |
| PopFront() | Remove front item |  |
| PushBack(key) | Add to back | Also known as *append* |
| Key TopBack() | Return back item |  |
| PopBack() | Remove back item |  |
| Boolean Find(key) | 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 |  |

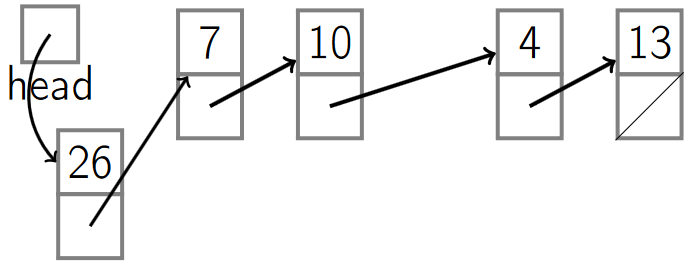
### Times for some operations

#### 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.

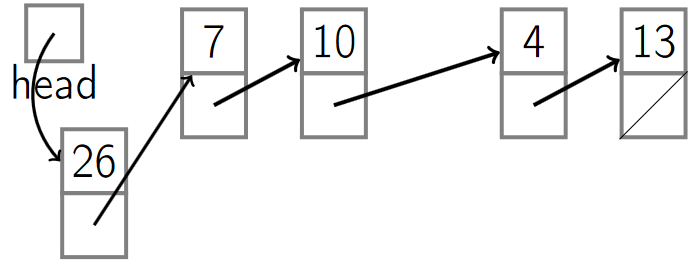


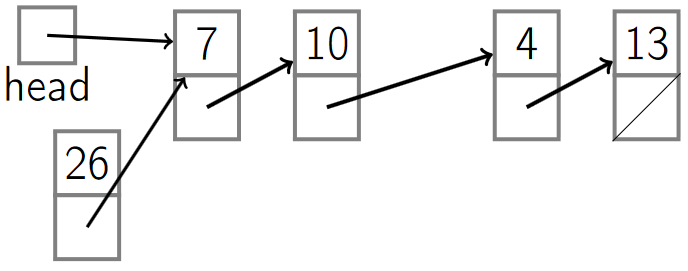


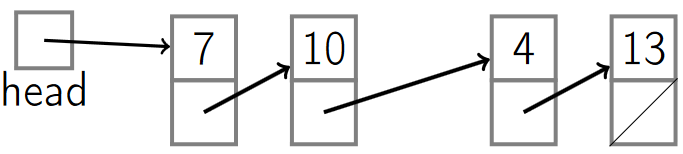


#### PopFront O(1)

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







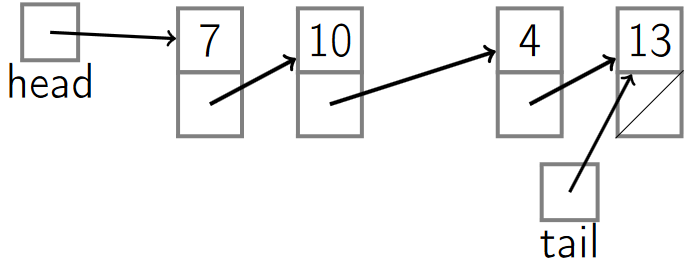
#### PushBack O(n) (no tail)

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)

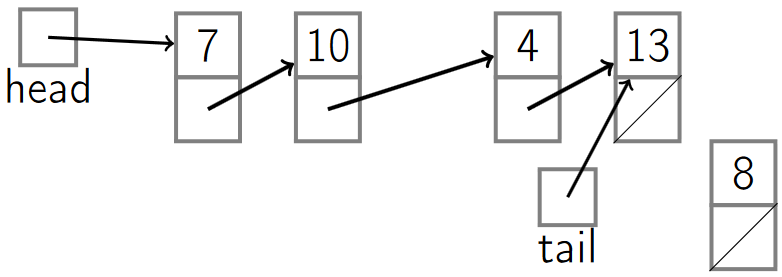
1. Start at the head and walk to the end of the list.
2. Remove the last element.

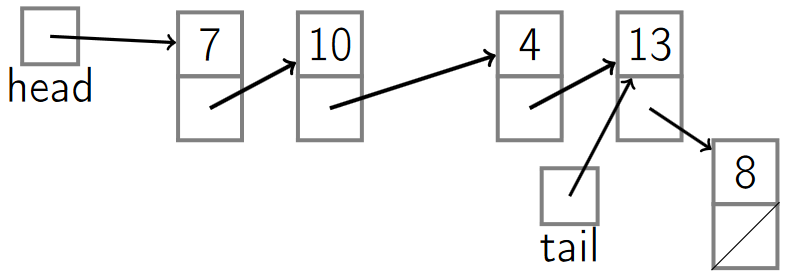
**Note**: adding a *tail pointer* is like having a *head pointer* but it points to the end of the list.

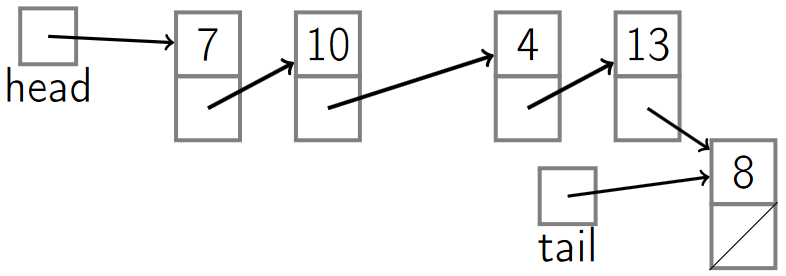


#### PushBack O(1) (with tail)

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.

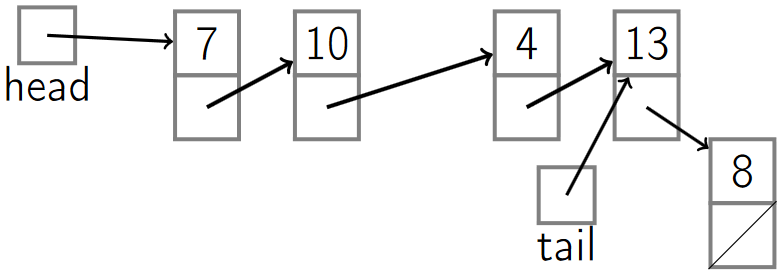


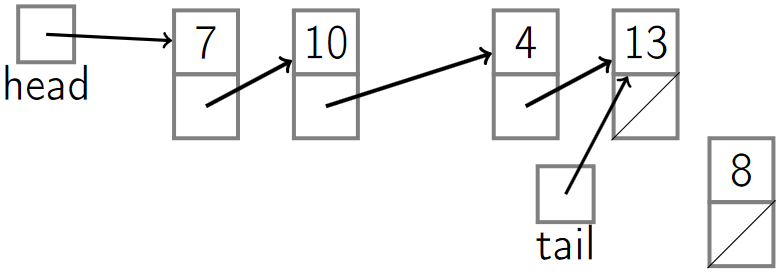


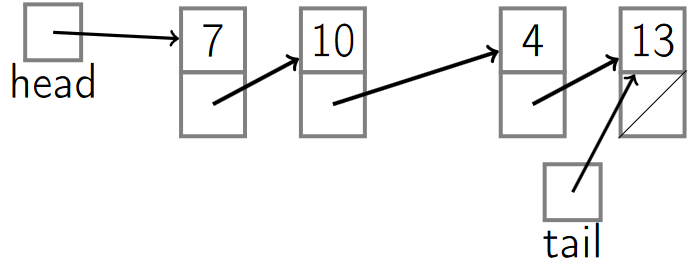


#### PopBack O(n) (with tail)

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.* |

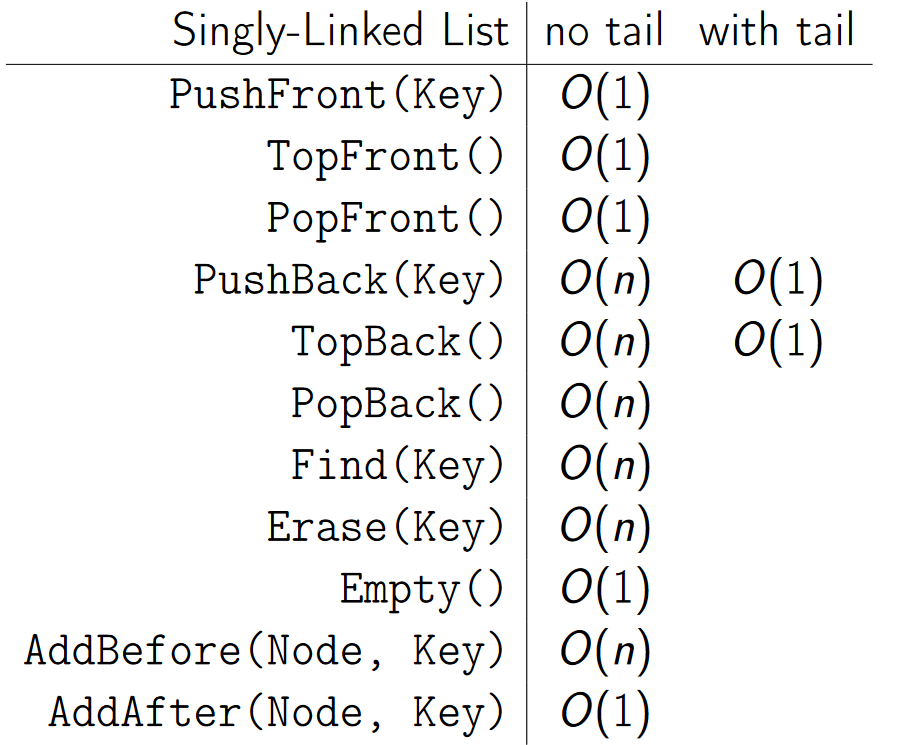
#### PopBack()

|  |  |
| --- | --- |
| 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

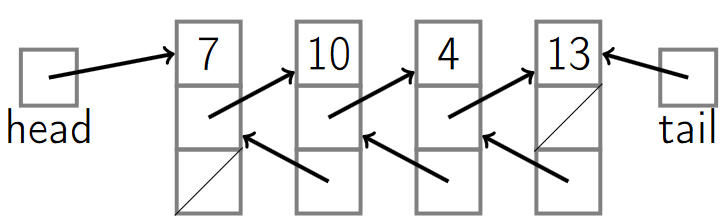


|  |  |  |
| --- | --- | --- |
| **Singly-Linked List** | **no tail** | **with tail** |
| PushFront(Key) | O(1) |  |
| TopFront() | O(1) |  |
| PopFront() | O(1) |  |
| PushBack(Key) | O(n) | O(1) |
| TopBack() | O(n) | O(1) |
| PopBack() | O(n) |  |
| Find(Key) | O(n) |  |
| Erase(Key) | O(n) |  |
| 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).



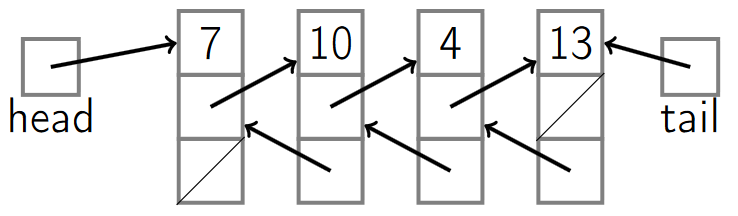
Each node contains:

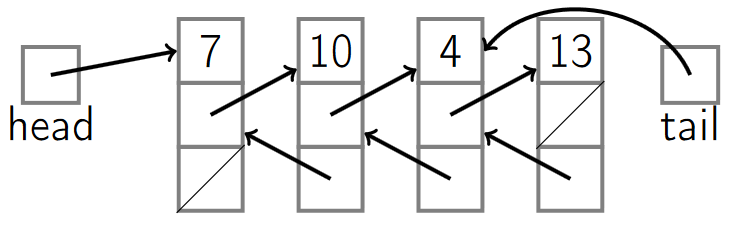
1. A key: the value (like an integer).
2. The next pointer.
3. The previoust pointer.

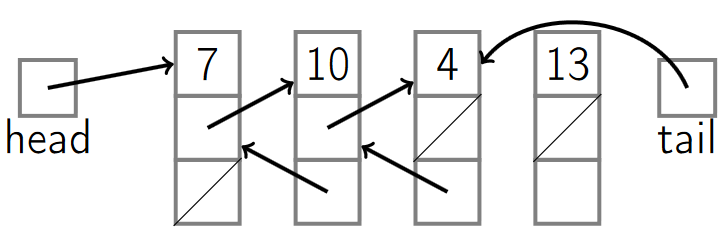
### Doubly-Linked List Pseudocode.

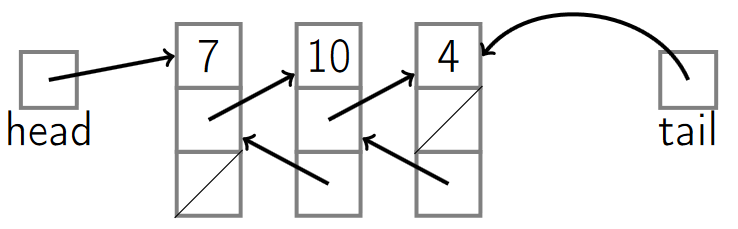
#### PopBack O(1)

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



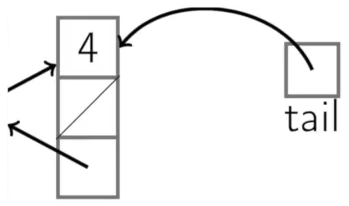
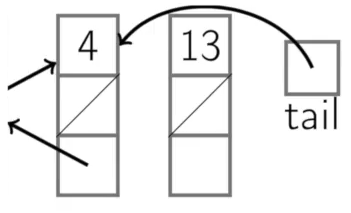
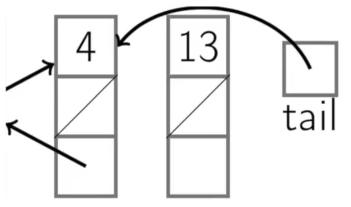






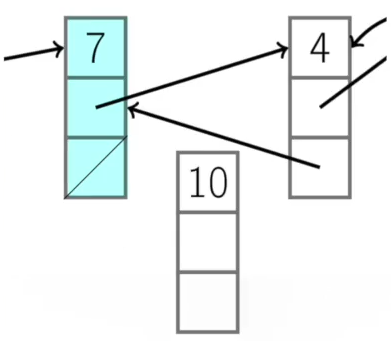
|  |  |
| --- | --- |
| if head= nil:  ERROR: empty list | *Check if the list is empty.* |
| if head=tail:  head←tail←nil | *If there’s only one element on the list, set the tail and head to null.* |
| else: | *Otherwise, there’s more than one element.* |
| tail←tail.prev | *Update the tail to be the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

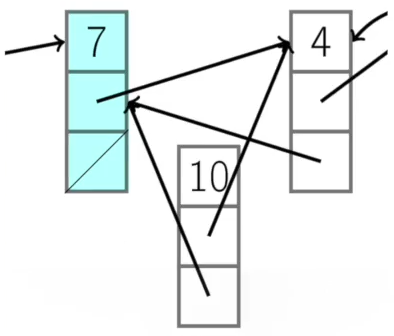
#### PushBack(key) O(1)

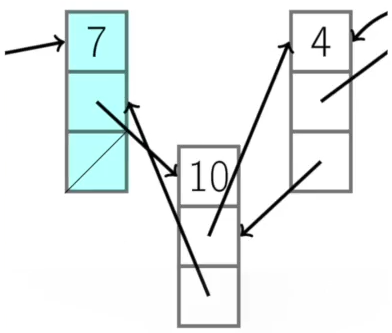
 

|  |  |
| --- | --- |
| node←new node  node.key←key | *Create a new node with its key.* |
| node.next = nil | *Set its next pointer to null.* |
| if tail = nil:  head←tail←node  node.prev←nil | *Check the current tail. If it’s null, then it’s an empty list so we update the tail and the head to point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

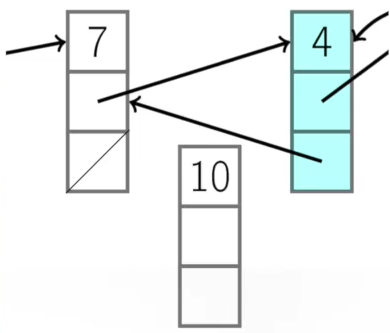


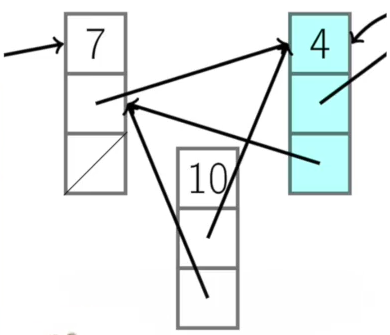


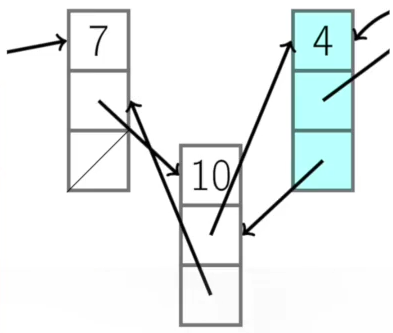


|  |  |
| --- | --- |
| node2←new node  node2.key←key | *Create a new node with its key.* |
| node2.next←node.next  node2.prev = node | *Update pointers.* |
| node.next = node2  if node2.nex t̸= nil:  node2.next.prev←node2 | *Maintain the previous pointer.* |
| if tail = node: |  |
| tail←node2 |  |

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

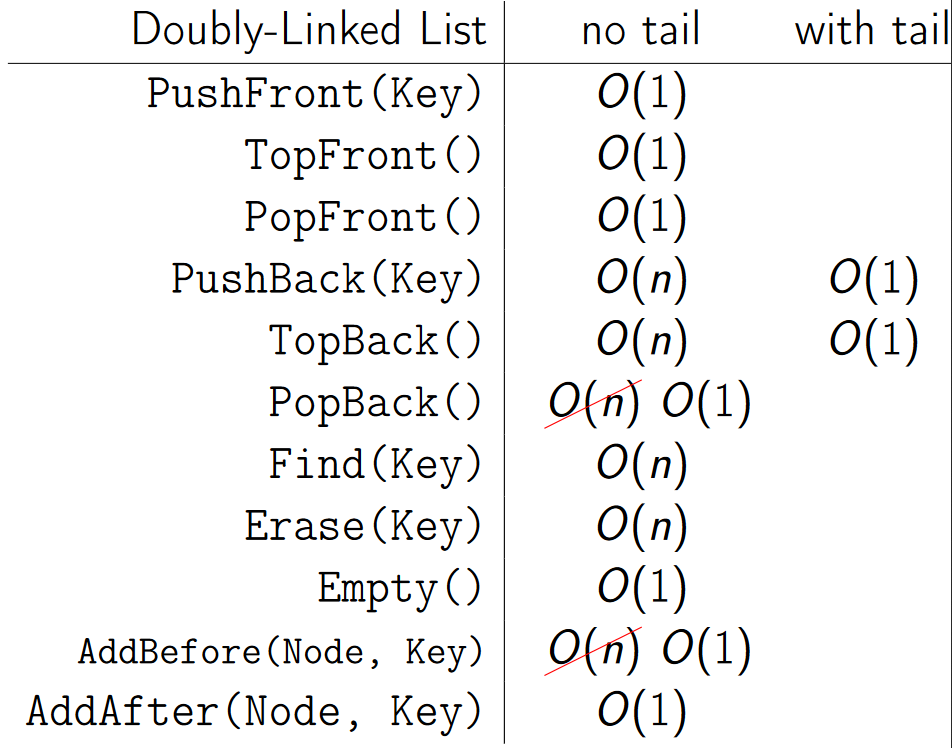






|  |  |
| --- | --- |
| node2←new node  node2.key←key | *Create a new node with its key.* |
| node2.next←node  node2.prev←node.prev | *Update pointers.* |
| node.prev←node2  if node2.prev ̸= nil:  node2.prev.next←node2 | *Maintain the previous pointer.* |
| if head = node: |  |
| head←node2 |  |

### 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).