Python for Data Scientists
L9: Linear Data structures

Shirin Tavara

Today's Topics

- Pointers
- Arrays
- Linked Lists
- Linear Data Structure
  - Stack and Queue

Pointer

- Variables used to store the location of another variables present in the memory, i.e., pointers store memory addresses.

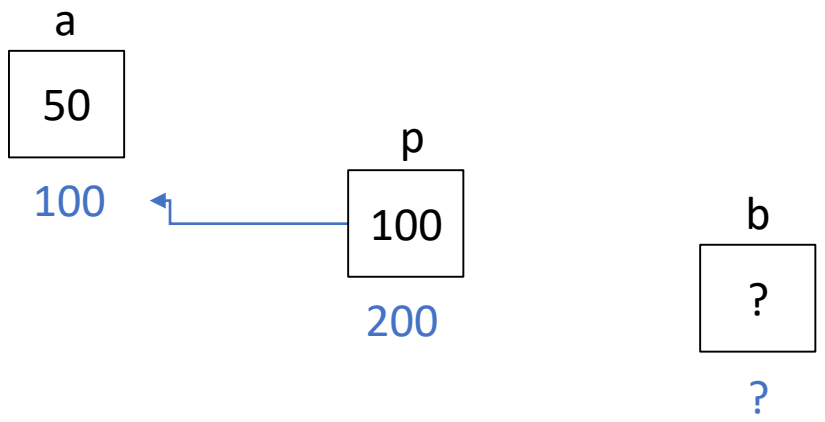
<dataType> *_name;

int *ip; → Pointer to an integer

double *dp; → Pointer to a double

```
int a = 50;
int *p;
p = &a;
int b;
b = *p;
```

→ Dereferencing



Pointer

- Variables used to store the location of another variables present in the memory, i.e., pointers store memory addresses.

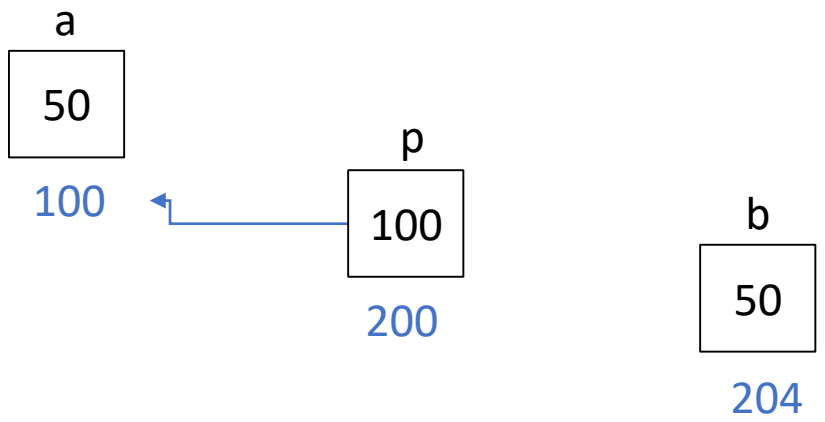
<dataType> *_name;

int *ip; → Pointer to an integer

double *dp; → Pointer to a double

```
int a = 50;
int *p;
p = &a;
int b;
b = *p;
```

→ Dereferencing

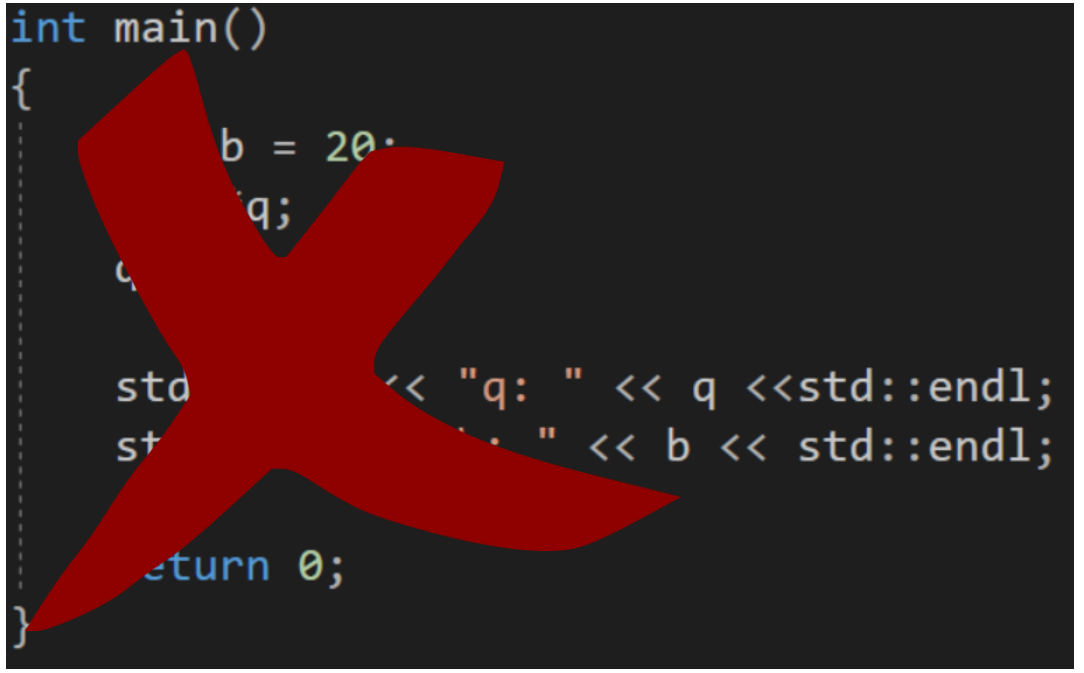


```cpp
int main()
{
    int a = 20;
    int *p;
    p = &a;

    std::cout << "p: " << p<<std::endl;
    std::cout<< "a: " << a << std::endl;

    return 0;
}
```

>> p: 012FFE5C
a: 20



'=': cannot convert from 'int' to 'int *'

Poll

What will be the result of the following code?

```
int main()
{
    int b = 20;
    int *q= &b;

    cout << "Nr1: " << b <<endl;
    cout << "Nr2: " << *q << endl;
    cout << "Nr3: " << &b << endl;
    cout << "Nr4: " << q << endl;
}
```

Pointers in Python? (poll)

id(): returns a unique IDentity (ID) for the object

The ID is the object's memory address

```
a=20
print(id(a))
```

>> 140736418723248

(1)

```
a=20
print(id(a))

a +=10
print(id(a))
```

>> 140736418723248
140736418723568

immutable

(2)

```
L=[1,2,3,4]
print(id(L))

L.append(5)
print(id(L))
```

>> 1826345481736
1826345481736

mutable

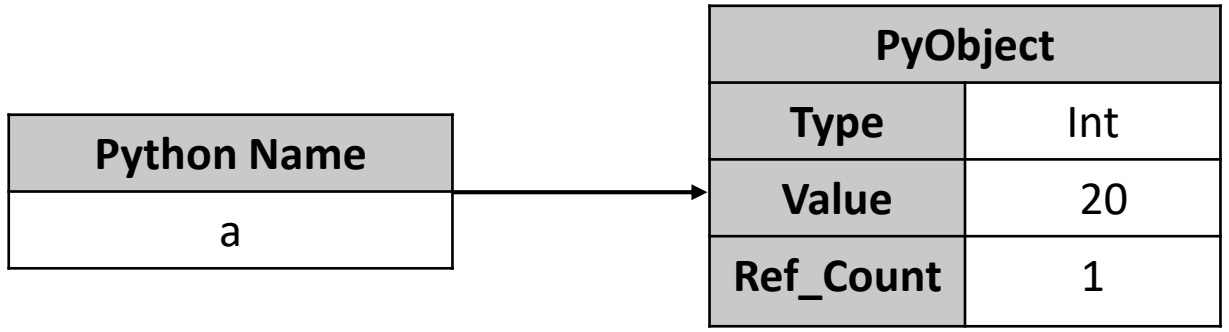
```
L=[1,2,3,4]
print(id(L))

L[0]=4
print(id(L))
```

>> 2534266549000
>> 2534266549000

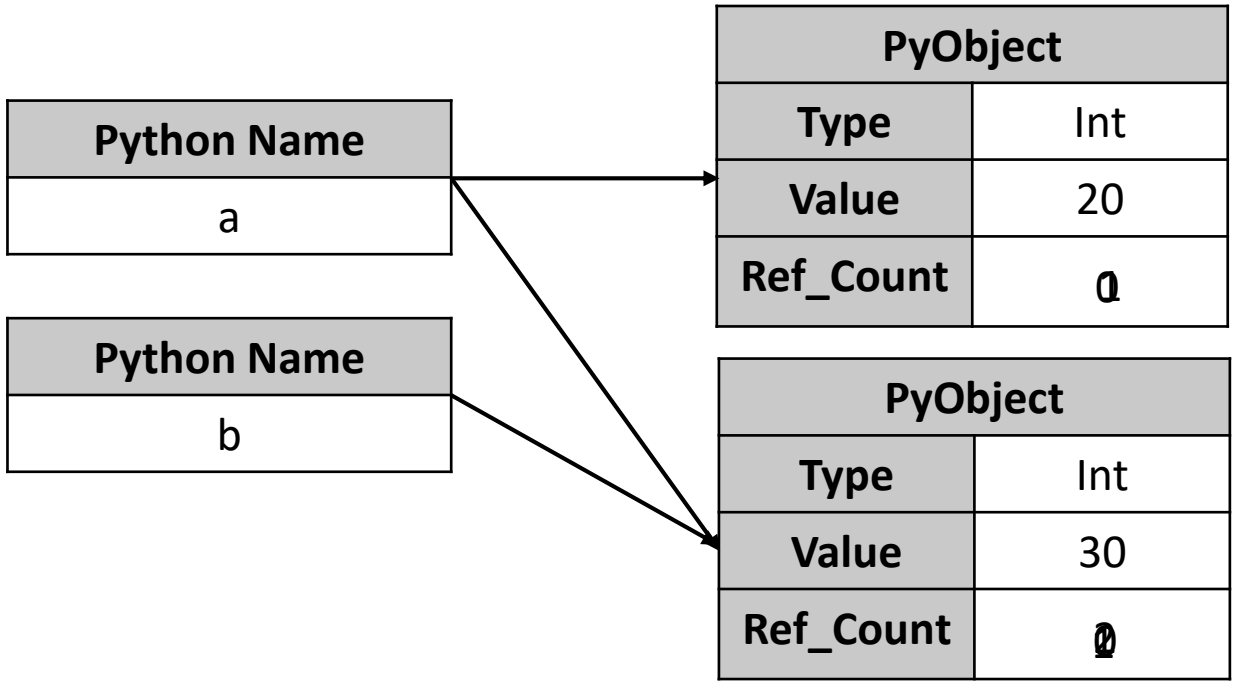
Pointers in Python?

```
>>> a= 20
```




```
>>> a += 10

>>> b = a
```



C++

```
int a = 50;
cout << "value of a: " << a << ", addr_a before : " << &a << endl;
a += 10;
cout << "value of a: " << a << ", addr_a after : " << &a << endl;
```

>> value of a: 50, addr_a before : 0052FDD8

a	
address	0052FDD8
Value	50

>>> value of a: 60, addr_a after : 0052FDD8

a	
address	0052FDD8
Value	60

Why to use pointers?

- Improve time complexity
- Control program flow

Linear Data Structure

- In a linear data structure, the data items are organized sequentially (or linearly).

- The data items are traversed serially one after another and all the data items in a linear data structure can be traversed during a single run.

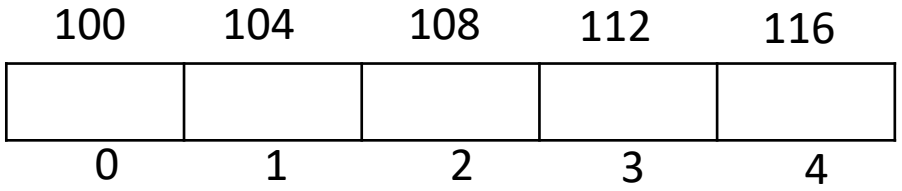
Arrays

- Contiguous area of memory
- Equal-size elements
- Indexes are contiguous integers

0	1	2	3	4	Zero-based indexing
1	2	3	4	5	One-based indexing

Key point-Arrays

- Constant-time access



- Access an element of array in index “i”:

$$array\_address + element\_size \times (i - first\_index)$$

$$100 + 4 \times (1 - 0) = 104$$
$$100 + 4 \times (2 - 0) = 108$$
$$100 + 4 \times (3 - 0) = 112$$
$$100 + 4 \times (4 - 0) = 116$$

- Reading and writing an element *O*(1)

Multi-Dimensional Arrays

(0,0)				
			(2,3)	

Multi-Dimensional Arrays

(0,0)				
			(2,3)	

- Skip rows of elements before the element of the interest:
  2 rows here, i.e., (2-0) × 5

Multi-Dimensional Arrays

(0,0)				
			(2,3)	

- Skip rows of elements before the element of the interest:
  2 rows here, i.e., (2-0) × 5
- Skip the columns before the elements
  3 columns here, i.e., (3-0)

- The address of the element of interest:

$$array\_address + element\_size \times ((2-0) \times 5 + (3-0))$$

Array lay-outs

- Row-major ordering

(0,0)
(0,1)
(0,2)
(0,3)
(1,0)
(1,1)

- Column-major ordering

(0,0)
(1,0)
(2,0)
(3,0)
(0,1)
(1,1)

Operations in Array

	Add	Remove	Read	Write
Start	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Middle	$O(n)$	$O(n)$	$O(1)$	$O(1)$
End	$O(1)$	$O(1)$	$O(1)$	$O(1)$

Linked List

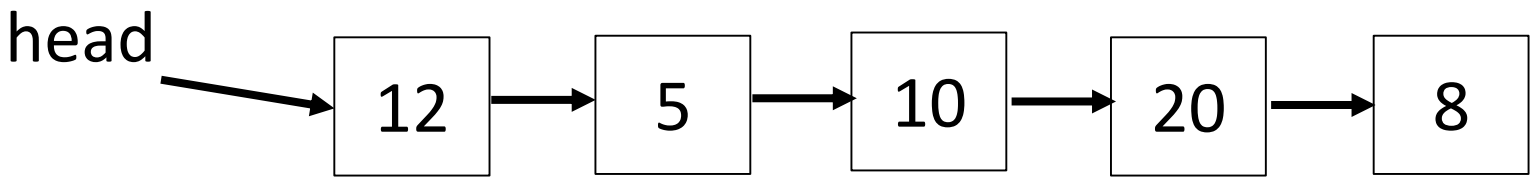
- It is a linear data structure

- Elements in linked list are not stored in contiguous location, they are linked with pointers.

Linked List

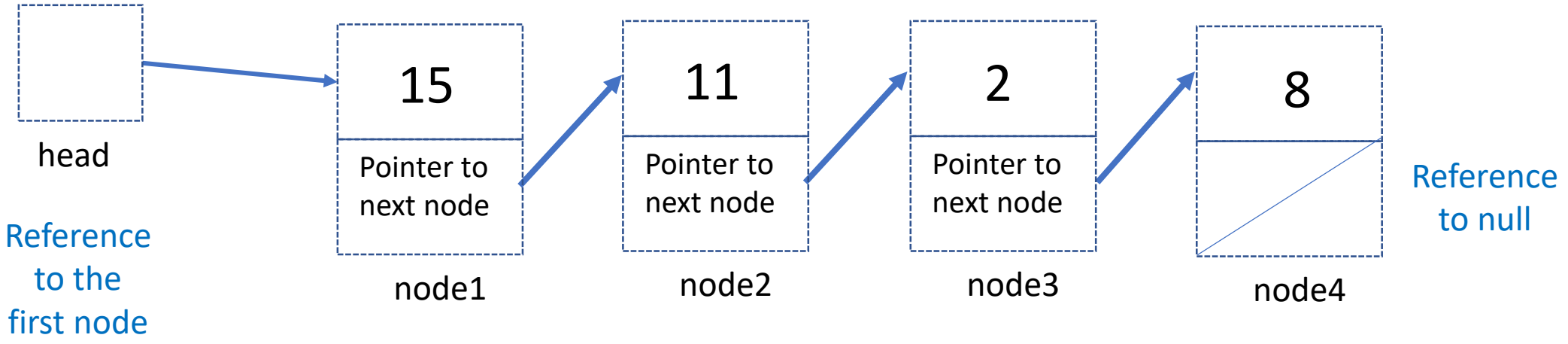
- It is a linear data structure → Like Arrays

- Elements in linked list are not stored in contiguous location, they are linked with pointers. → Unlike Arrays



Linked Lists: Singly-Linked Lists

- Head
- Node
  - Key or data
  - Pointer to the next node



Linked List Operations

Operation	Description
Boolean Empty()	Is the list empty?
PushFront(Key)	Add to front
Key TopFront()	Return front item
PopFront()	Remove front item
PushBack(Key)	Add to back, it is known as Append
Key TopBack()	Return back item
PopBack()	Remove back item
AddAfter(Node,Key)	Adds key after node
AddBefore(Node,Key)	Adds key before node
Erase(Key)	Remove key from list

Question?

We have an empty list what will be the result after the following operations?

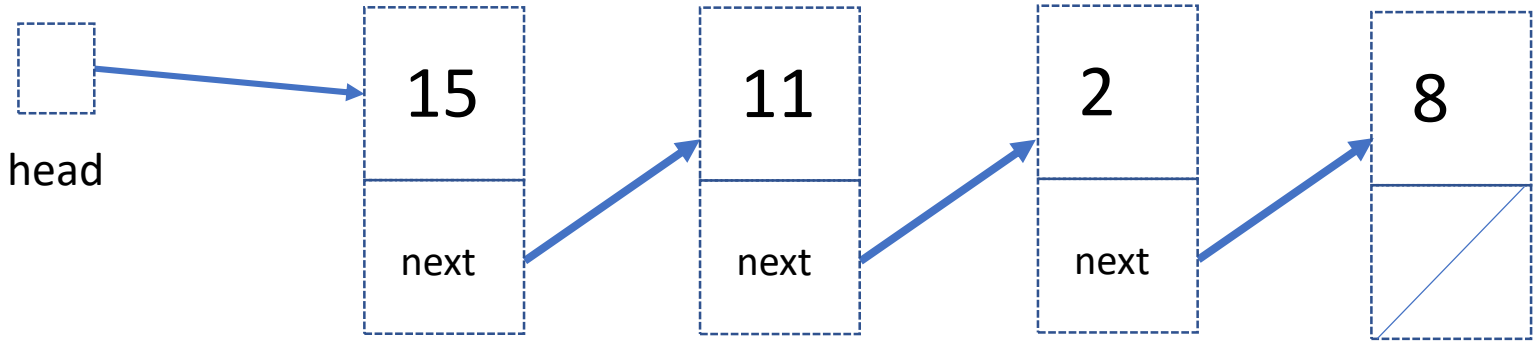
```
PushFront(T)
PushBack(H)
PushBack(O)
PushFront(Y)
PushBack(N)
PushFront(P)
PopBack()
```

Question?

We have an empty list what will be the result after the following operations?

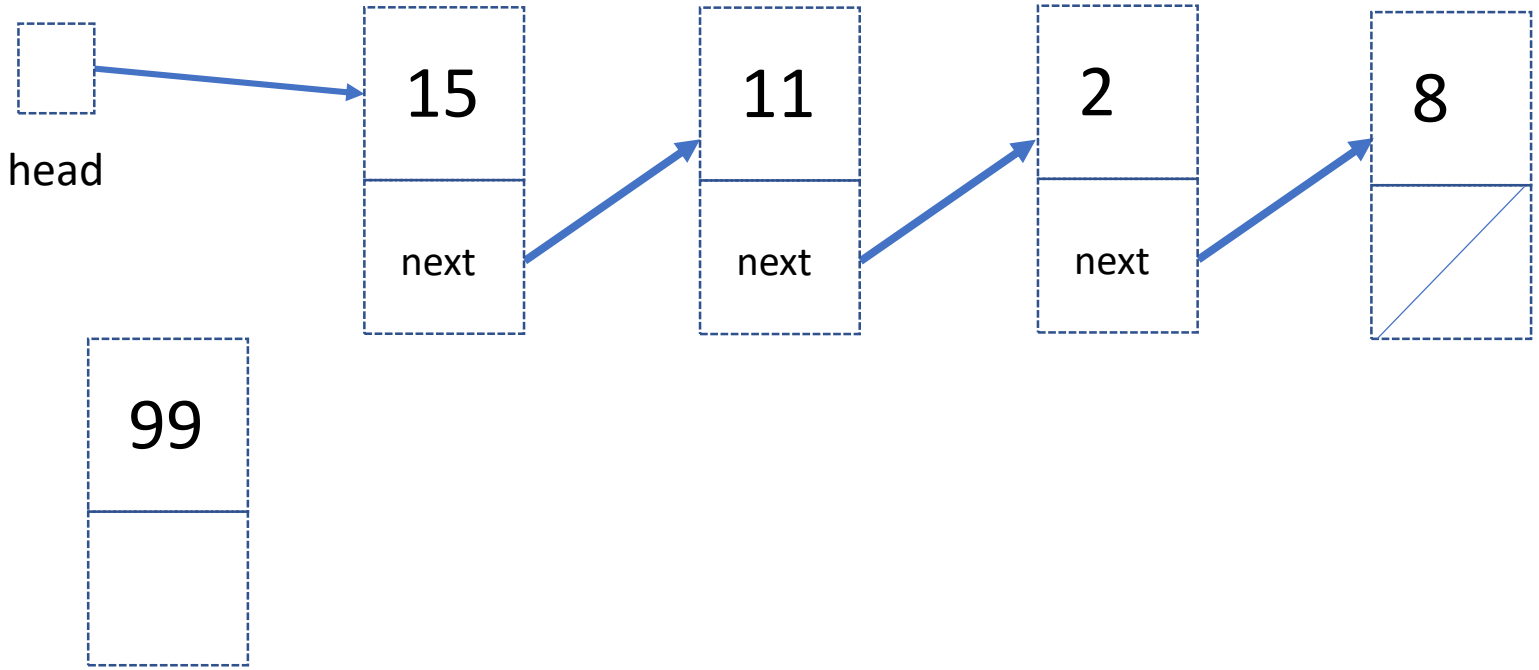
```
PushFront(T) -> T
PushBack(H)  -> T, H
PushBack(O)  -> T, H, O
PushFront(Y) -> Y, T, H, O
PushBack(N)  -> Y, T, H, O, N
PushFront(P) -> P, Y, T, H, O, N
PopBack()    -> P, Y, T, H, O
```

Closer look to operations- PushFront



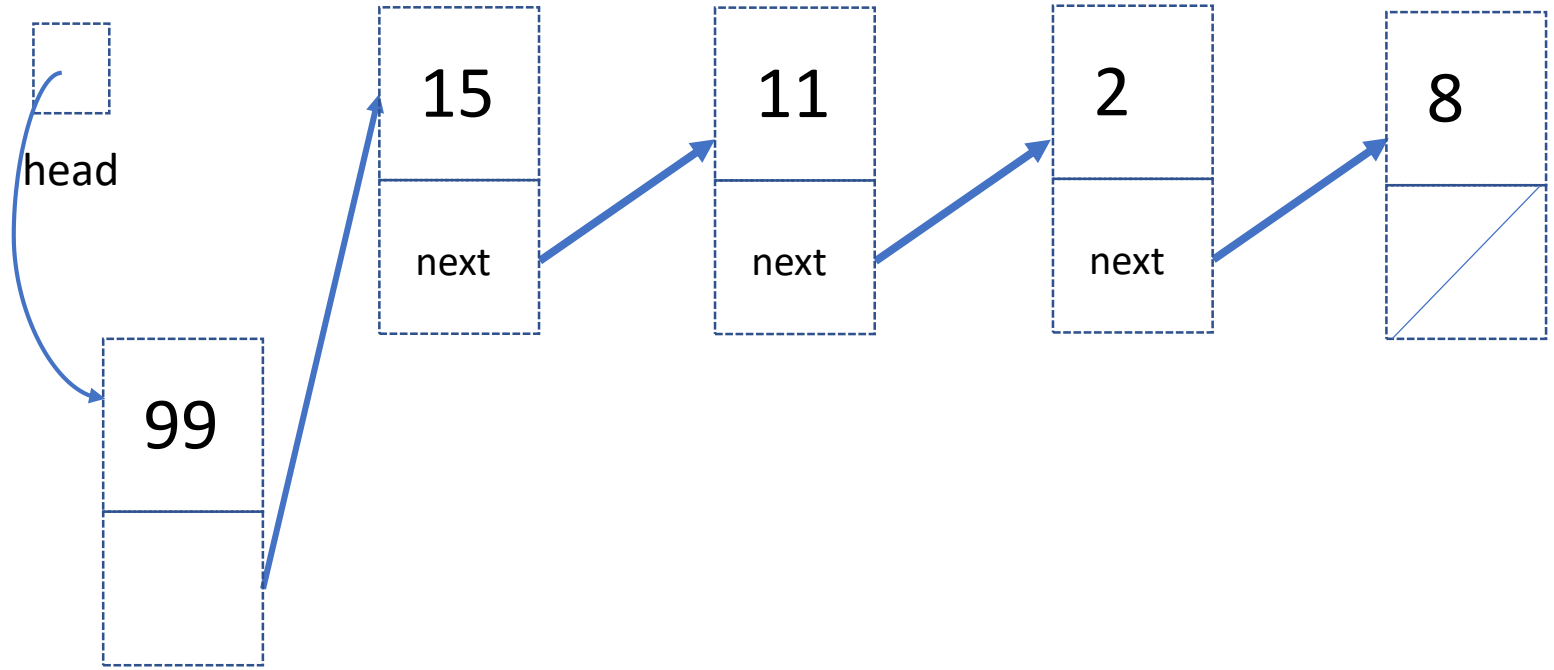
Closer look to operations- PushFront

1. Create a node



Closer look to operations- PushFront

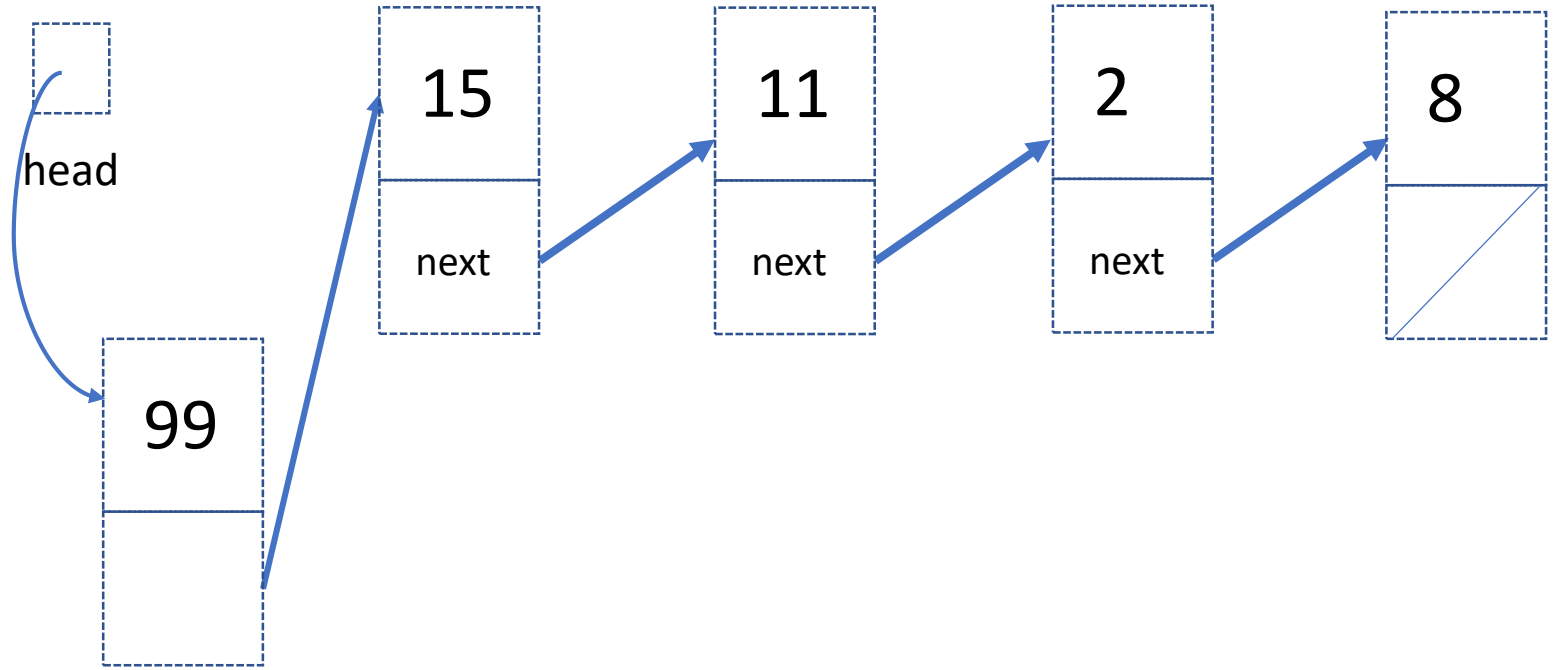
1. Create a node
2. Update the corresponding pointers



Closer look to operations- PushFront

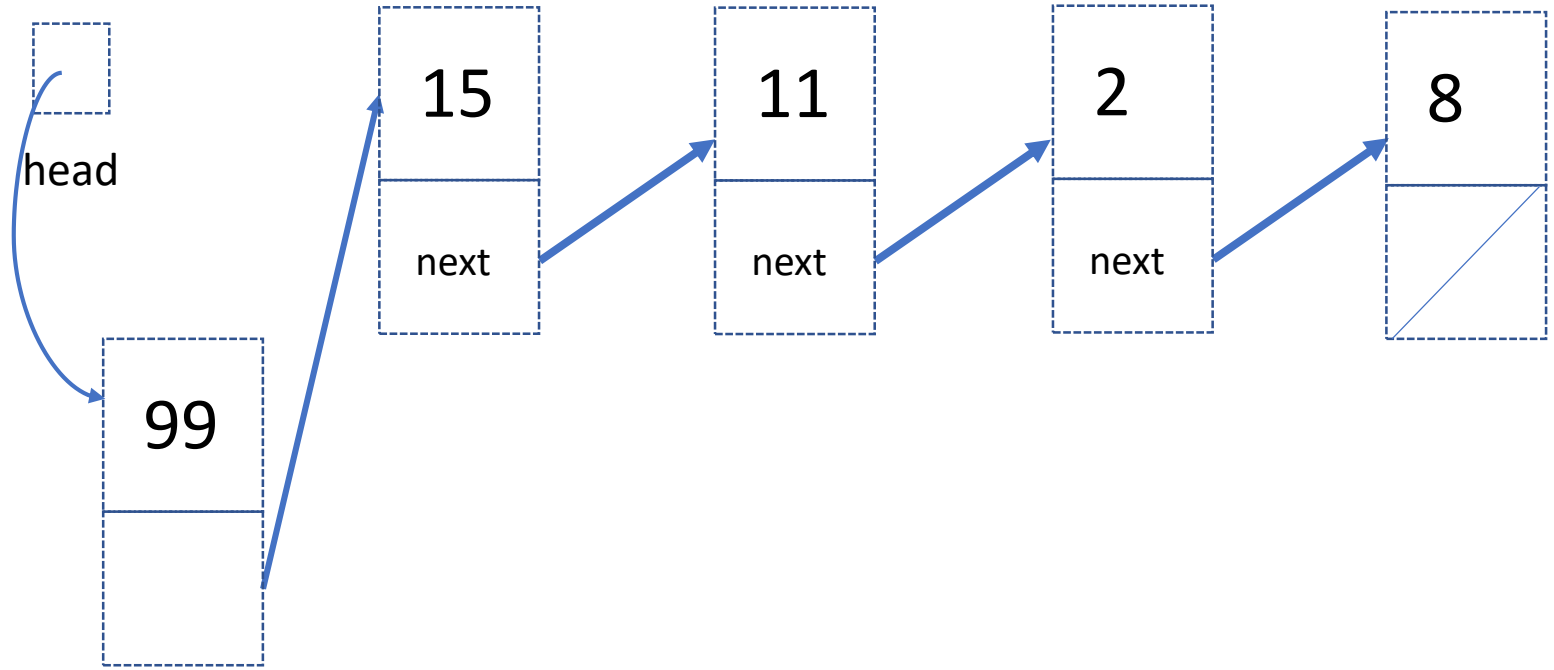
1. Create a node
2. Update the corresponding pointers

$O(1)$



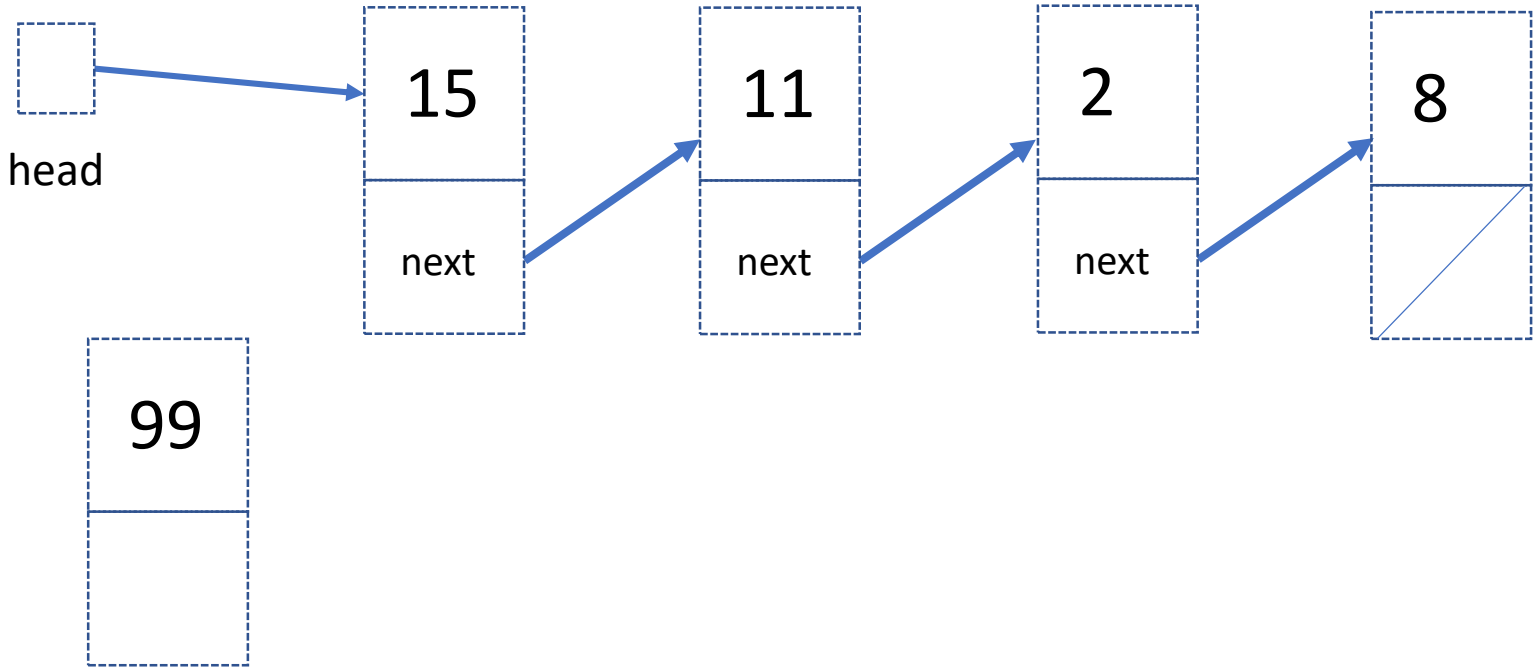
Closer look to operations- PopFront

Finding the first element is cheap



Closer look to operations- PopFront

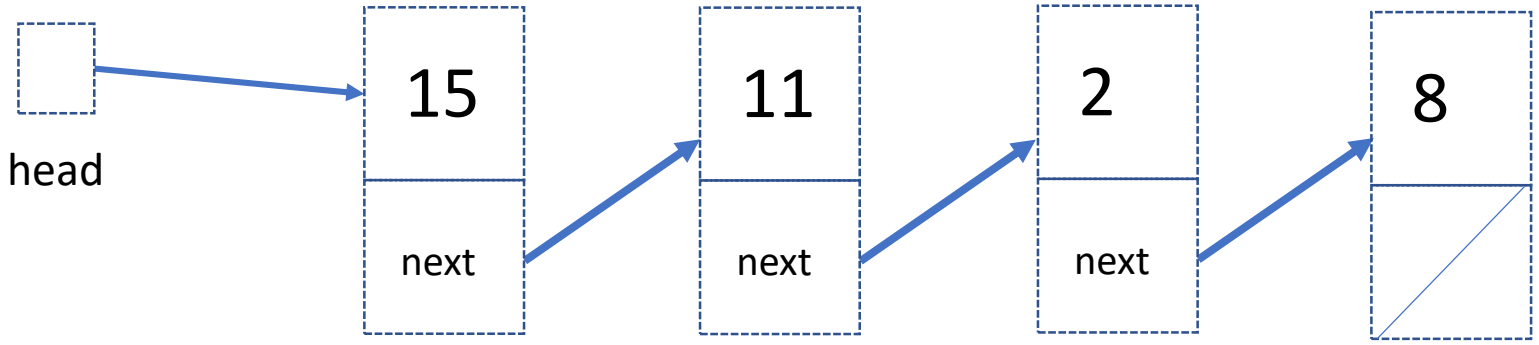
1. Change the head pointer



Closer look to operations- PopFront

1. Change the head pointer
2. Remove the node

$O(1)$



Similar approach for TopFront(), $O(1)$

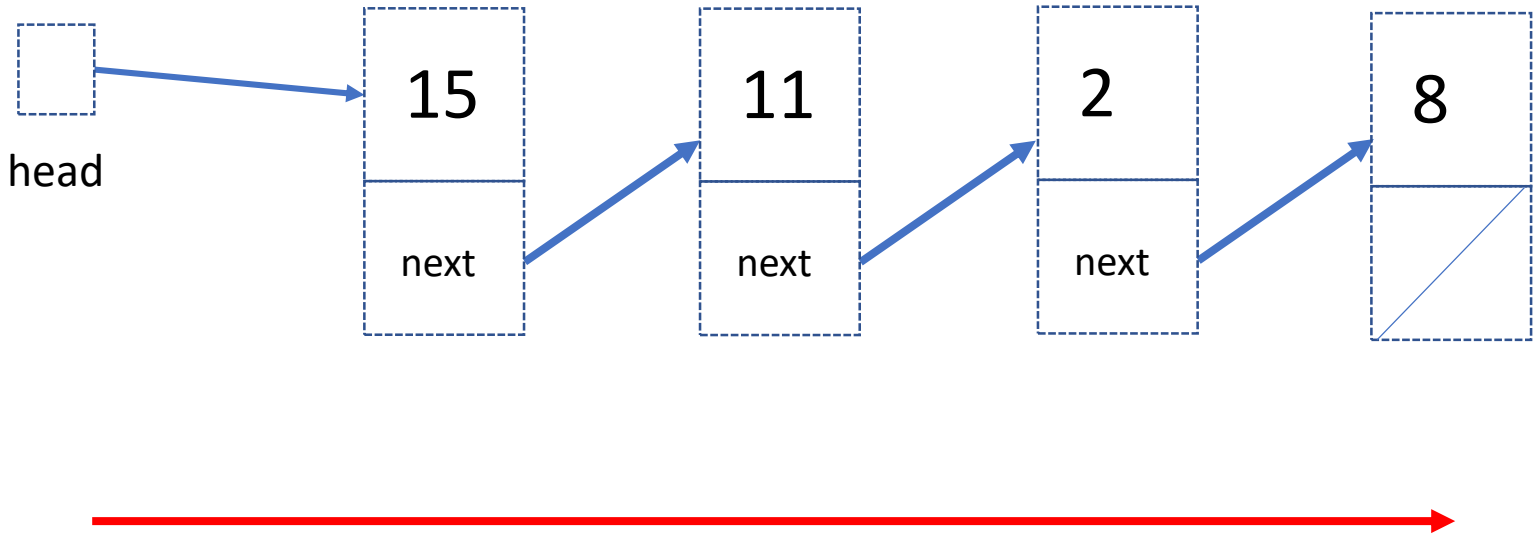
Closer look to operations- PushBack

What about PushBack? $O(?)$

Closer look to operations- PushBack

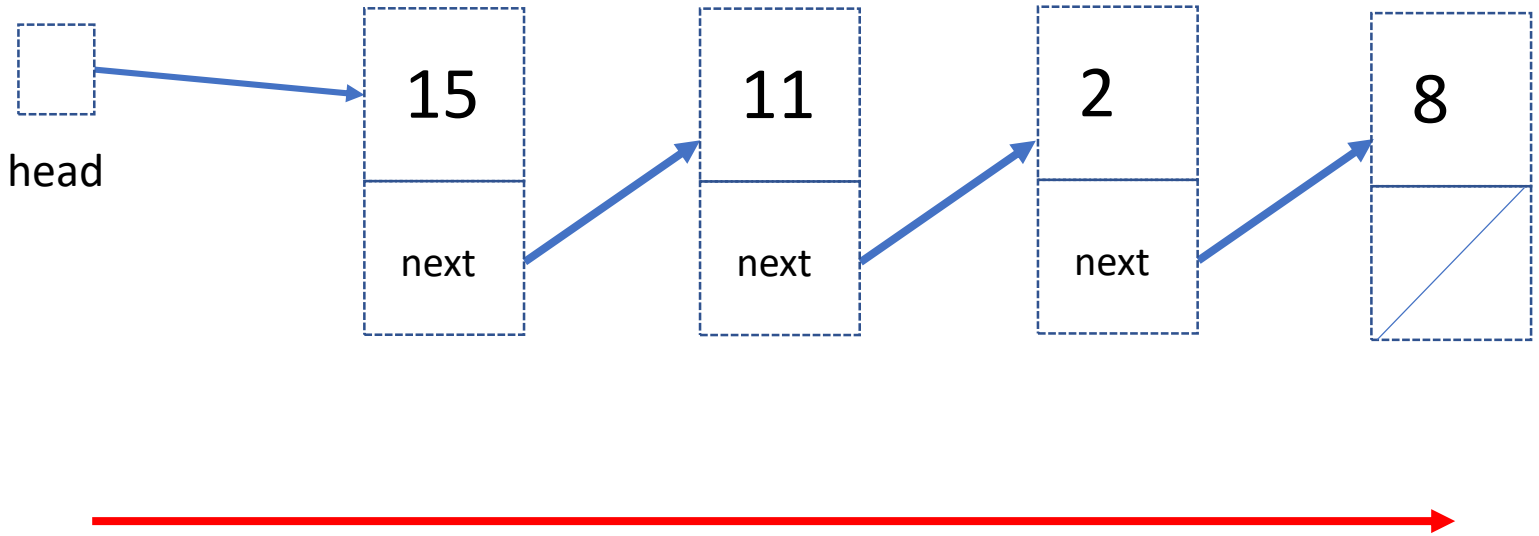
Start from the head and move to the end

$O(n)$



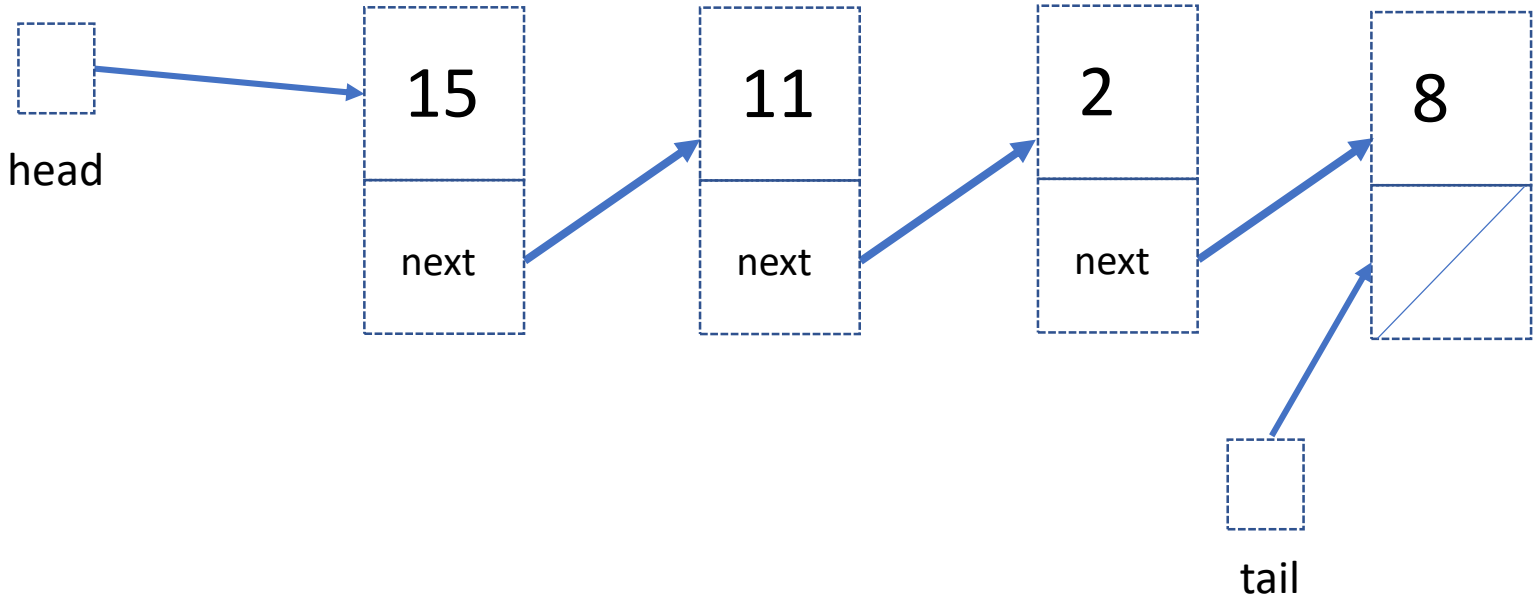
Closer look to operations- PopBack() & TopBack()

Similar to PushBack, *O*(n)



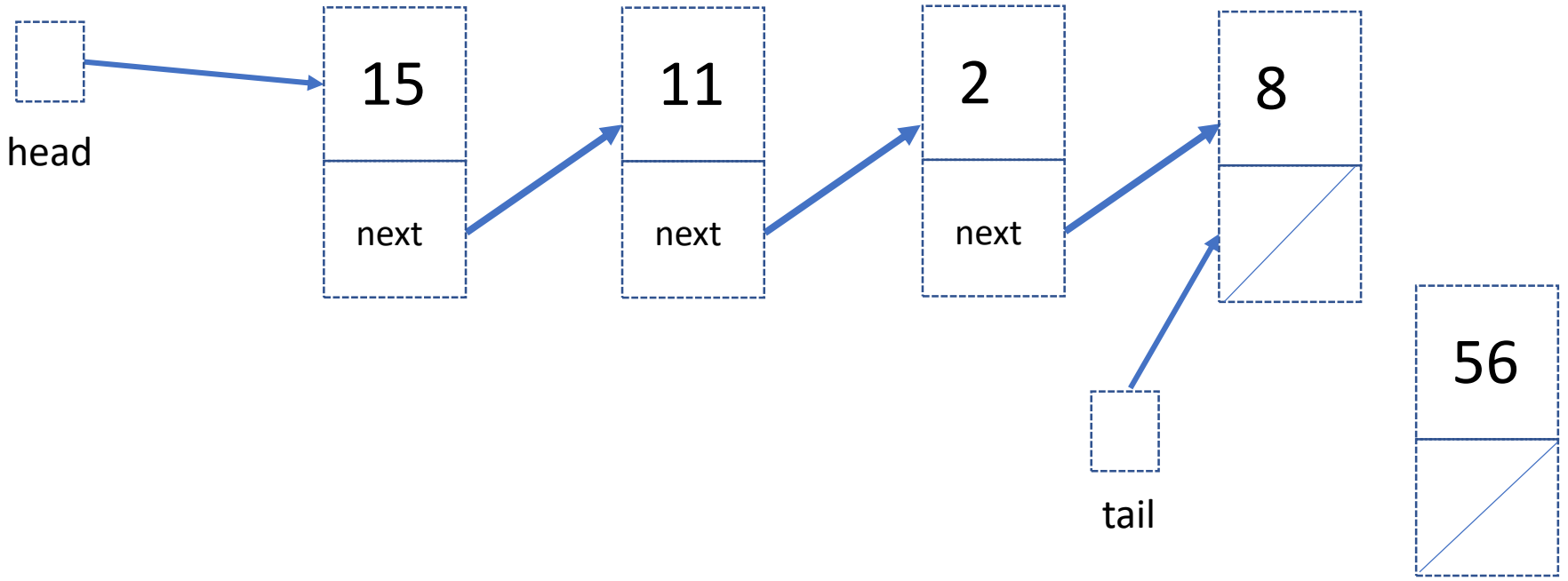
Linked List with tail pointer

Getting the first and the last elements are cheap



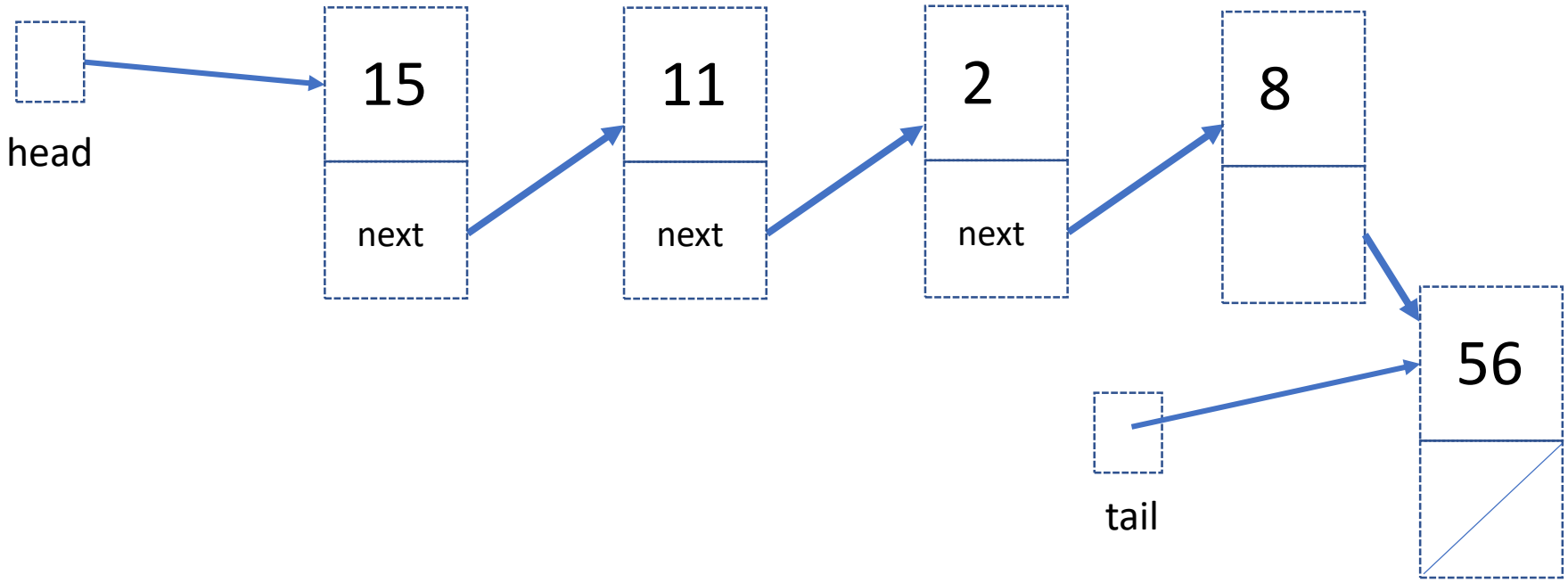
Linked List with tail pointer- PushBack()

Now what will be cost of PushBack?



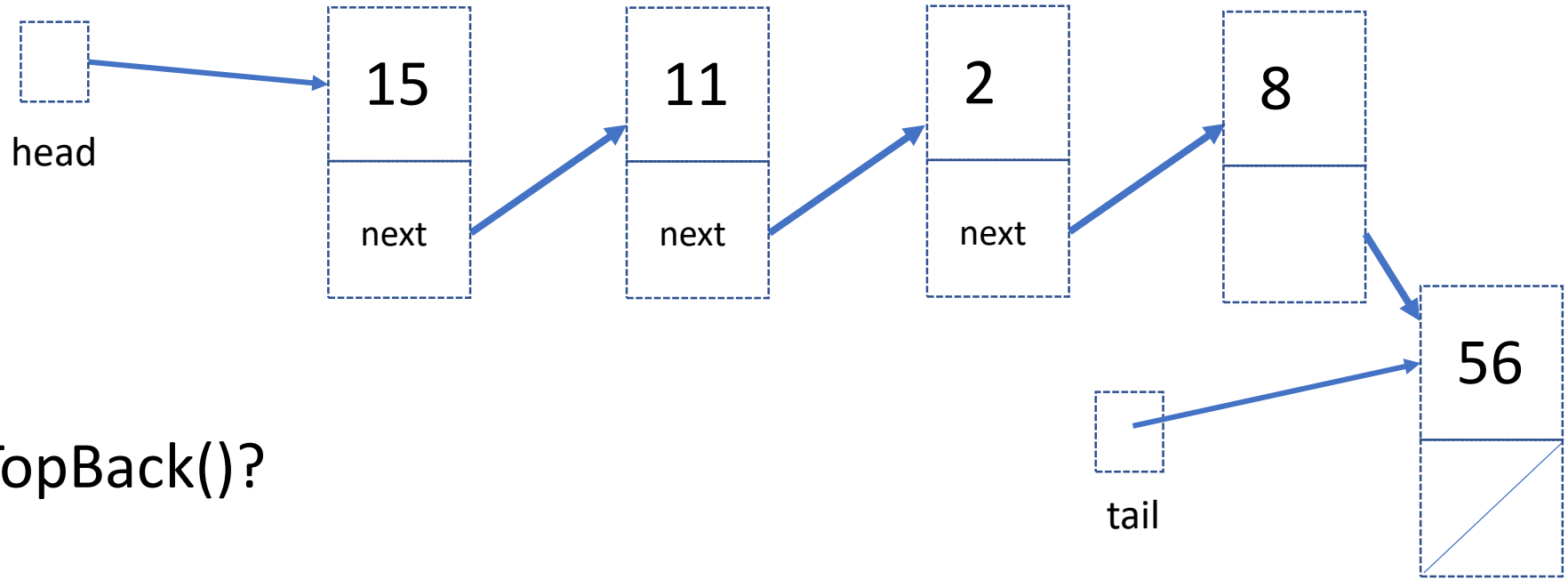
Linked List with tail pointer- PushBack()

Update the corresponding pointers



Linked List with tail pointer-PushBack()

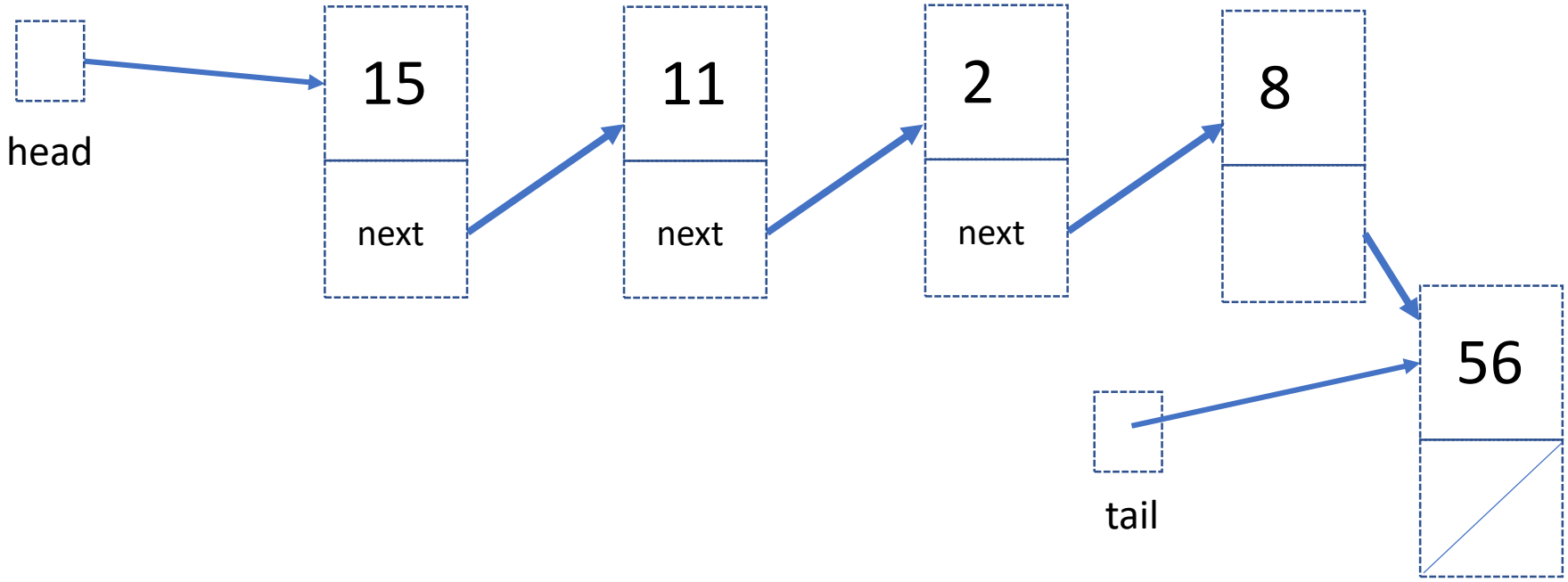
Update the corresponding pointers, $O(1)$



What about TopBack()?
$O(1)$

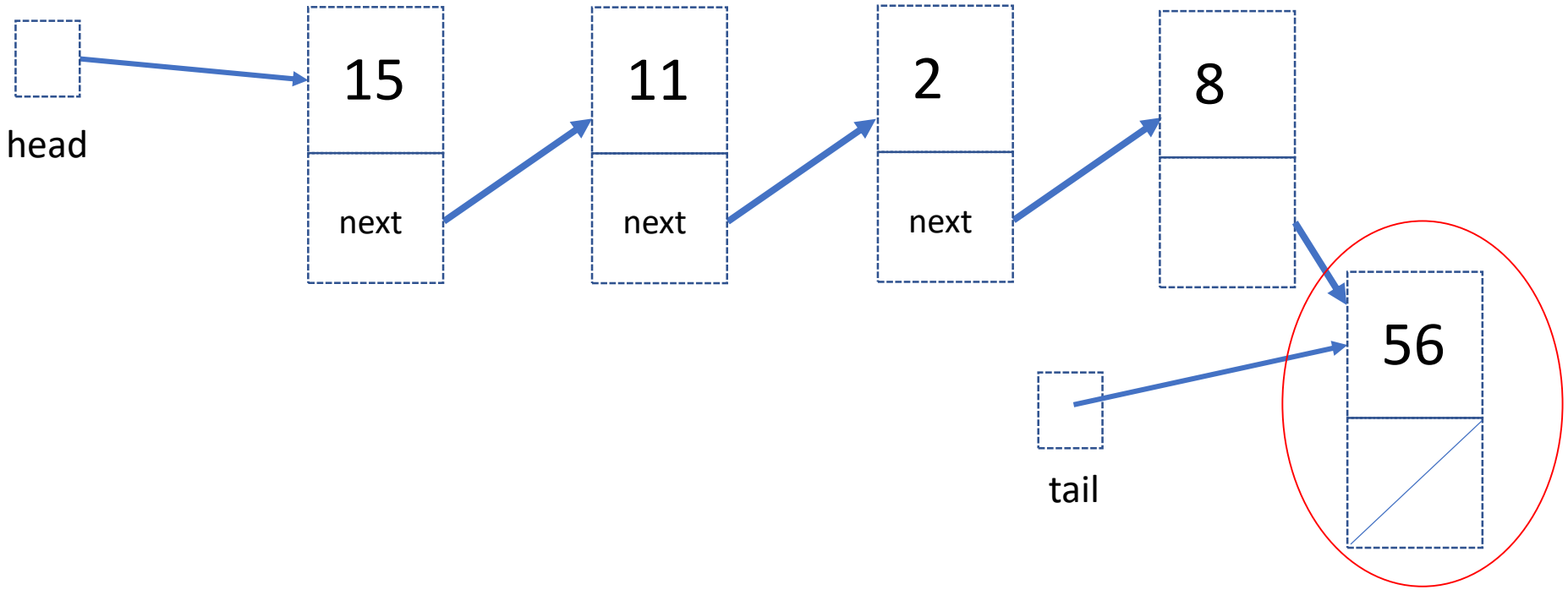
Linked List with tail pointer-PopBack()

What about PopBack()? *O*(?)



Linked List with tail pointer-PopBack()

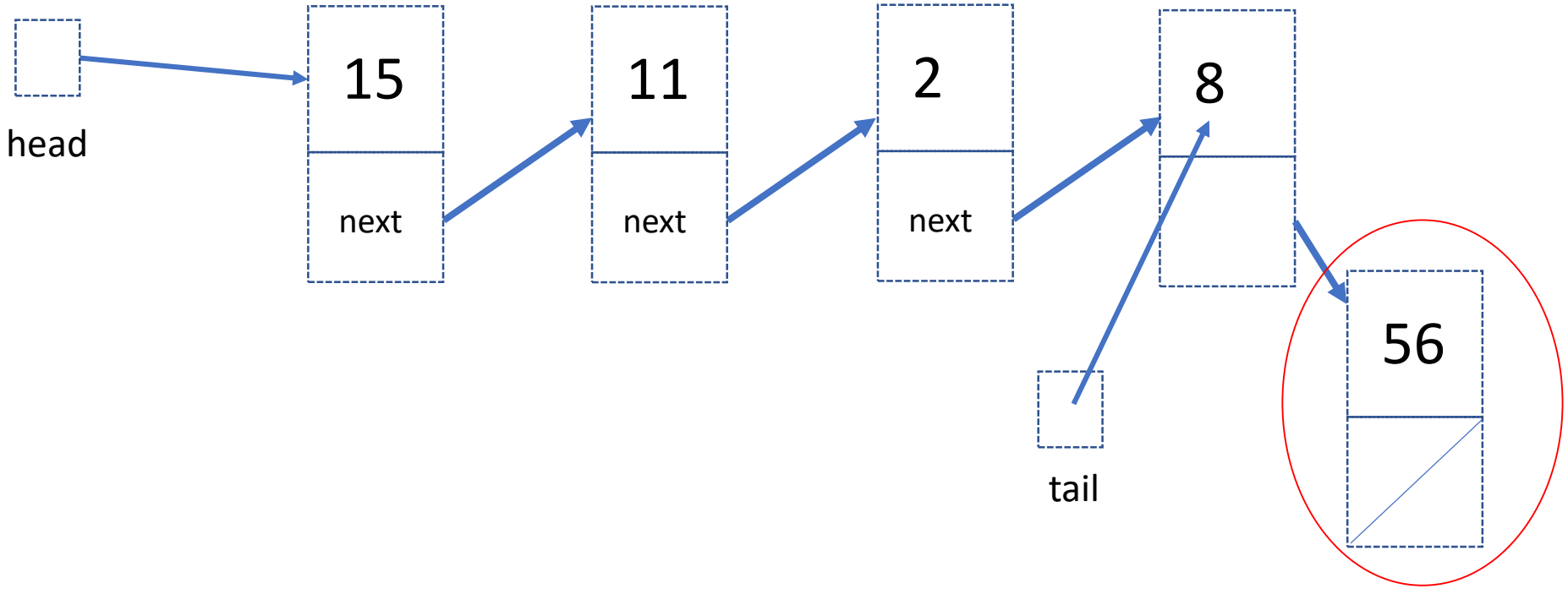
We can access the last node via the tail pointer and remove the last node



Linked List with tail pointer-PopBack()

We need to update the tail to the node with element 8, but how?

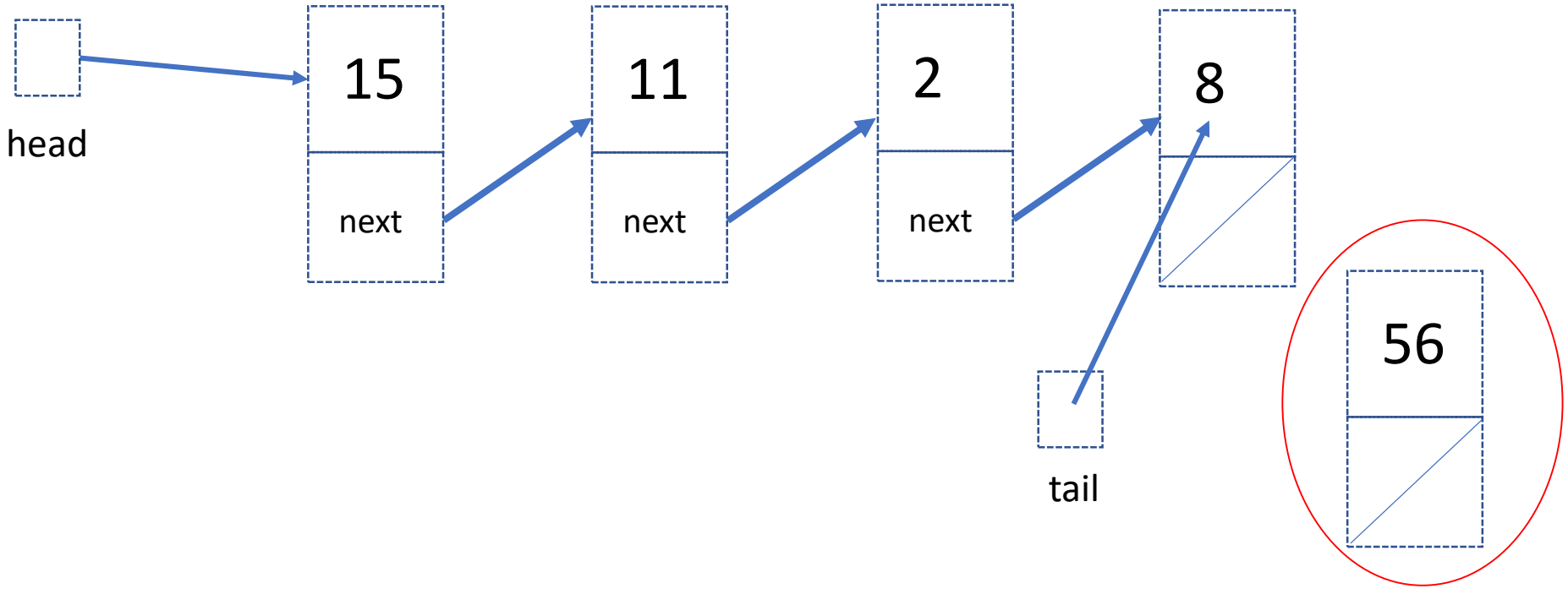
We cannot walk back?



Linked List with tail pointer-PopBack()

We need to update the tail to the node with element 8, but how?

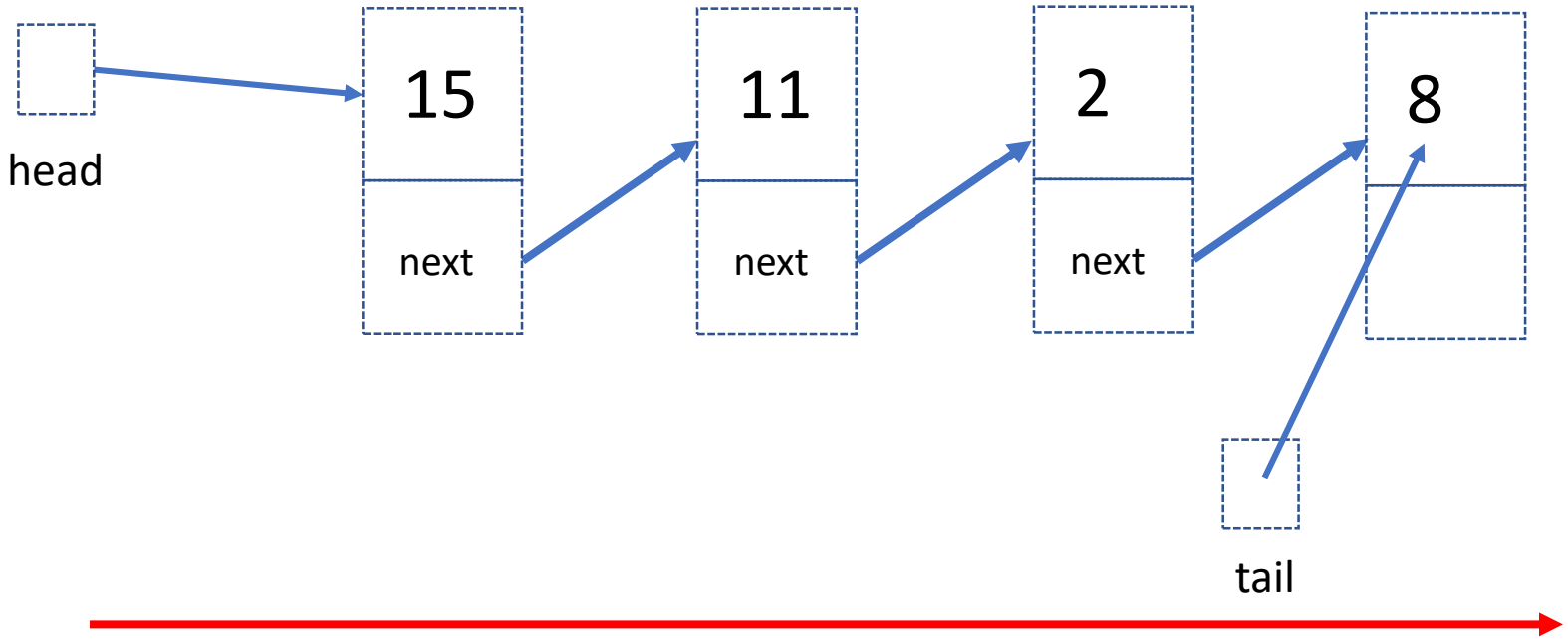
We cannot walk back?



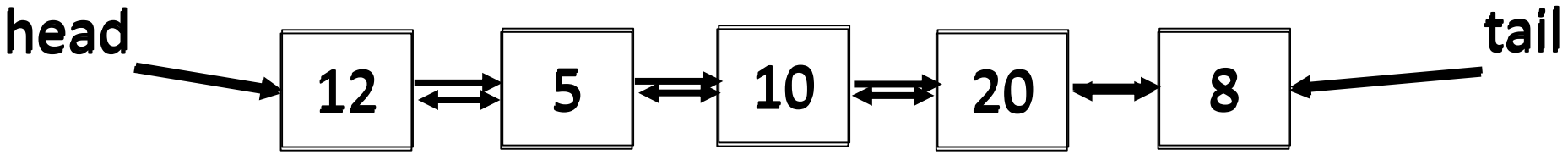
Linked List with tail pointer-PopBack()

We need to walk from beginning

$O(n)$



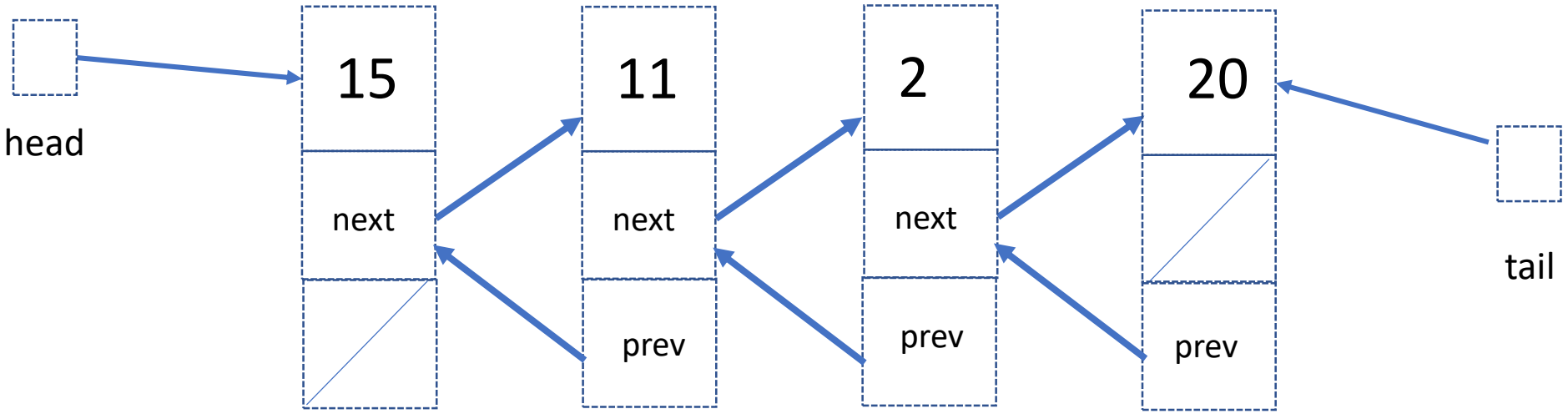
Doubly-Linked Lists



Doubly-Linked Lists- Implementation

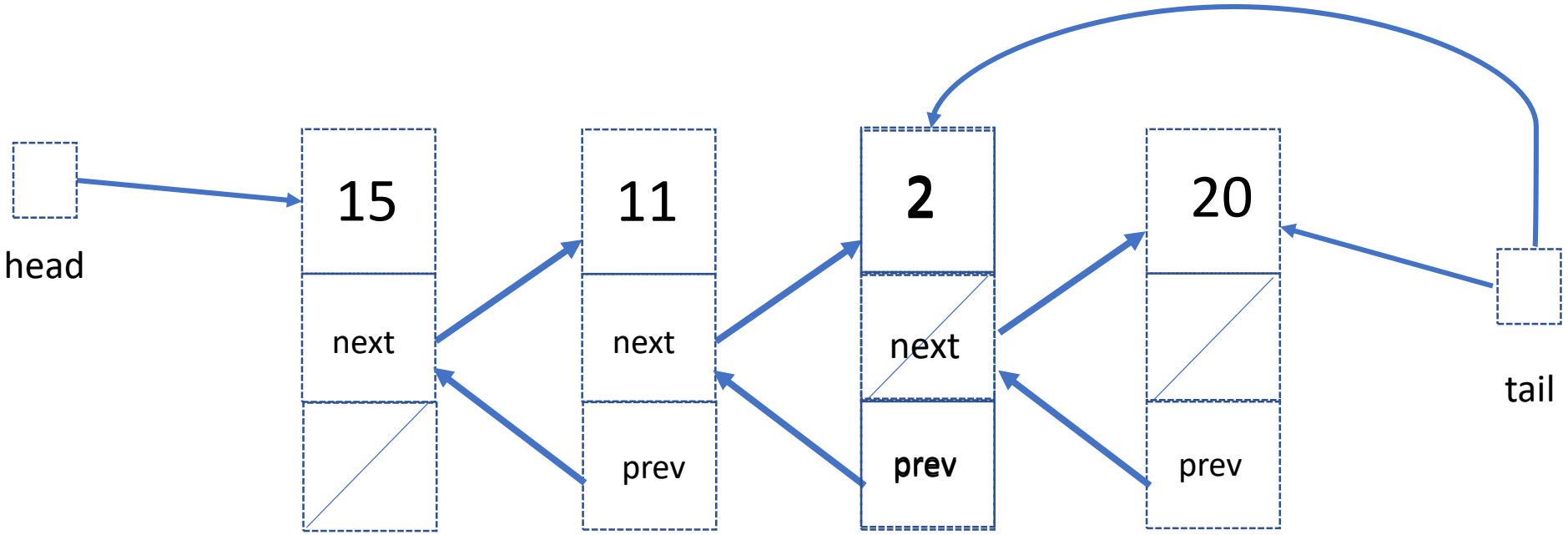
Node contains;

- Data or key
- A Pointer to the next node
- A pointer to the previous node



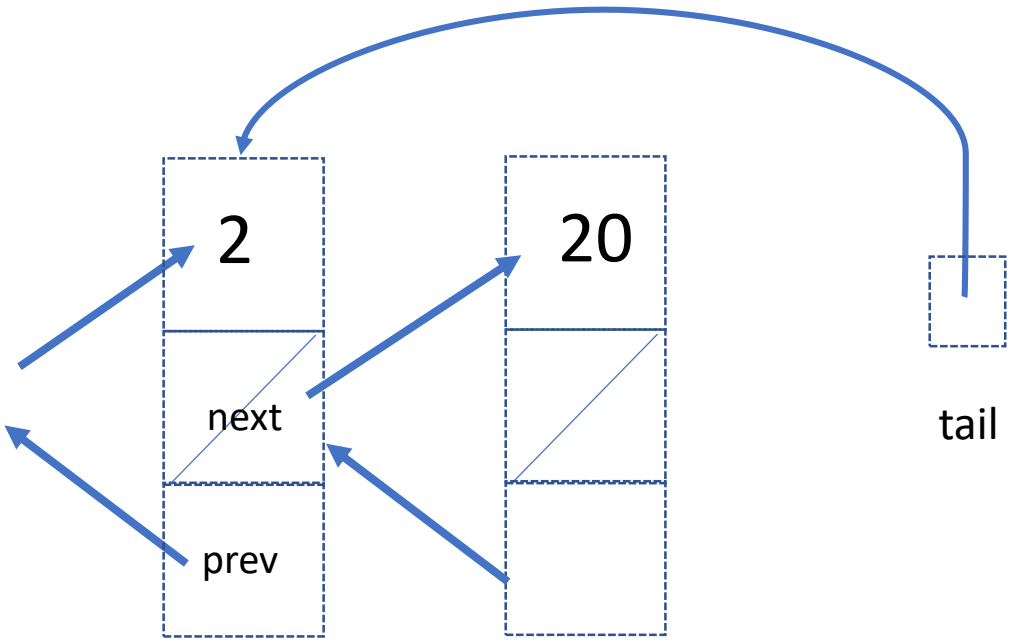
Doubly-Linked Lists- PopBack()

- $O(1)$



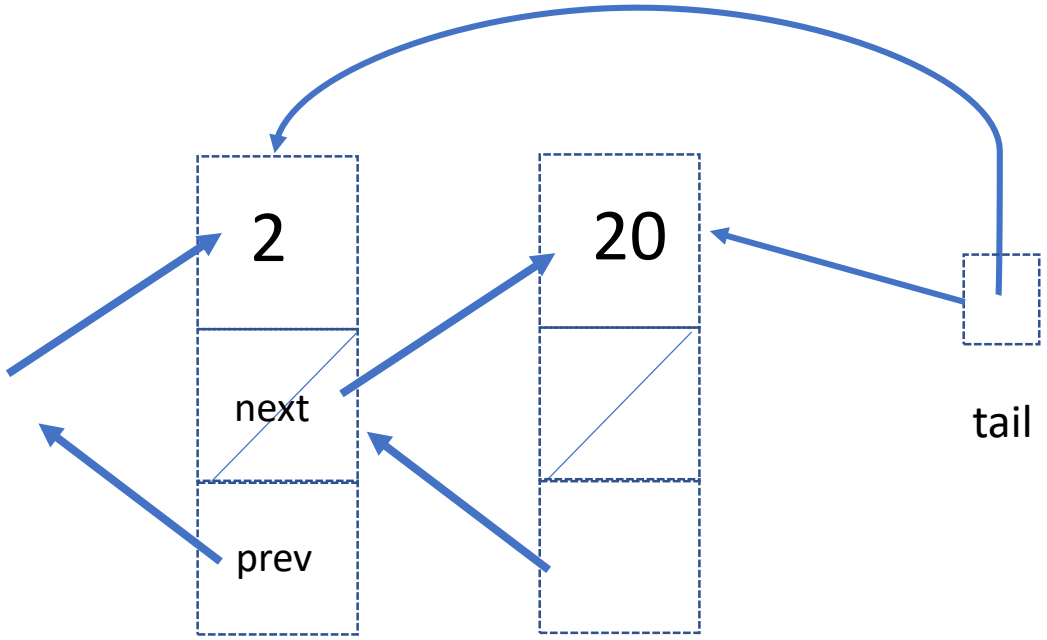
Doubly-Linked Lists- PushBack

PushBack(key)



Doubly-Linked Lists- PopBack()

```
PopBack()
if head= Null:
    ERROR: empty List
```



Operations	Singly-Linked List		Doubly-Linked List
	Without tail	With tail	
PushFront(key)	*O*(1)		*O(1)*
TopFront()	*O(1)*		*O(1)*
PopFront()	*O(1)*		*O(1)*
PushBack(key)	*O(n)*	*O(1)*	*O(1)*
TopBack()	*O(n)*	*O(1)*	*O(1)*
PopBack()	*O(n)*		*O(1)*
Find(key)	*O(n)*		*O(n)*
Erase(key)	*O(n)*		*O(n)*
AddBefore(Node, key)	*O(n)*		*O(1)*
AddAfter(Node, key)	*O(1)*		*O(1)*

Operations	Singly-Linked List		Doubly-Linked List
	Without tail	With tail	
PushFront(key)	*O*(1)		*O(1)*
TopFront()	*O(1)*		*O(1)*
PopFront()	*O(1)*		*O(1)*
PushBack(key)	*O(n)*	*O(1)*	*O(1)*
TopBack()	*O(n)*	*O(1)*	*O(1)*
PopBack()	*O(n)*		*O(1)*
Find(key)	*O(n)*		*O(n)*
Erase(key)	*O(n)*		*O(n)*
AddBefore(Node, key)	*O(n)*		*O(1)*
AddAfter(Node, key)	*O(1)*		*O(1)*

Recap

Array

- Random access to an element within
- Costly to insert at or remove from the front
- If the size exceeds, then must be resized
- Removing elements leaves empty spaces
- Only need to save data
- List of data with the known size

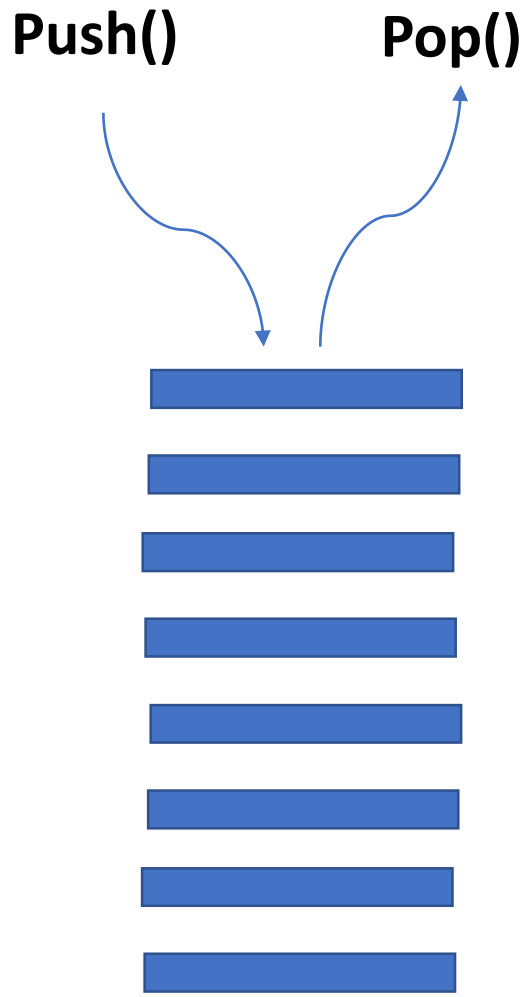
Linked List

- Sequential access to an element within
- Constant time to insert at or remove from the front
- Can always add elements
- Constant time to insert at or remove from the back with tail and doubly-linked
- Need to save both data and a pointer to the next node
- For large list of data where the size of the list can change

Stack

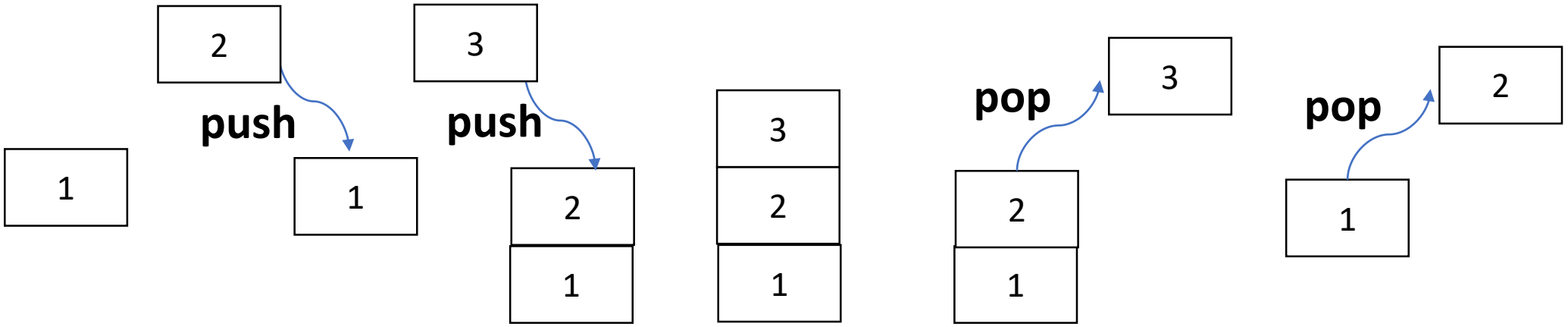
It stores items in a Last-In-First-Out (LIFO) manner

- A new element is added at one end/top and an element is removed from the same end
- The *insert/add* and *delete* operations are often called *Push()* and *Pop()*



Stack - Operations

- **Key Top():** returns most recently-added key
- **Push(key):** adds key at the top of the stack
- **Key Pop():** removes and returns most recently-added key
- **Boolean Empty():** Are there any elements in the stack?



Matching Brackets Problem

Non-Matched brackets:

“( ) ([ ] [ ]))”

“[( ]]”

“[[ ))]]”

Matched brackets:

“( ) ([ ] [ ])”

“[( )]”

“[[(( ))]]”

Matching Brackets Problem

Can Stack be a good solution?

```
IsMatched(myString)
    Stack stack
    for str in myString:
        if str in ["[", "(" ]:
            stack.Push(str)
        else:
            if stack.Empty():
                return False
            top <- stack.Pop()
            if (top = "(" and str != ")") or (top = "[" and str != "]"):
                return False
    return stack.Empty()
```

Poll

Given the string “[][()”, which character will be on the top of stack when for loop is finished?

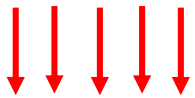
1. “(“
2. “]”
3. “)”
4. “[“

```
IsMatched(myString)
    Stack stack
    for str in myString:
        if str in ["[", "(" ]:
            stack.Push(str)
        else:
            if stack.Empty():
                return False
            top <- stack.Pop()
            if (top = "(" and str != ")") or (top = "[" and str != "]"):
                return False
    return stack.Empty()
```

Poll

Given the string “[ ] [( ) ”, which character will be on the top of stack when for loop is finished?

1. “(“
2. “]”
3. “)”
4. “[“



Reading “[ ] [( )“

Stack Implementation using Arrays

NrElements: 4

n

ERROR

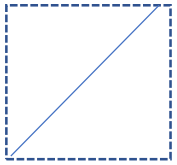
Push(d), Push(n), Top(), Push(c), Pop(), Push(z), Push(m), Push(k)

Empty(), Pop(), Pop(), Pop(), Pop(), Empty()

False

True

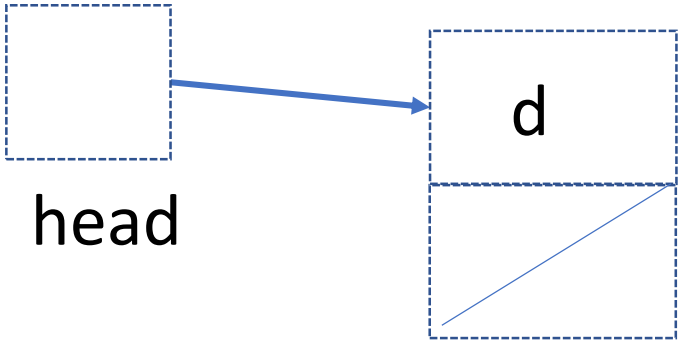
Stack Implementation using Linked Lists



head

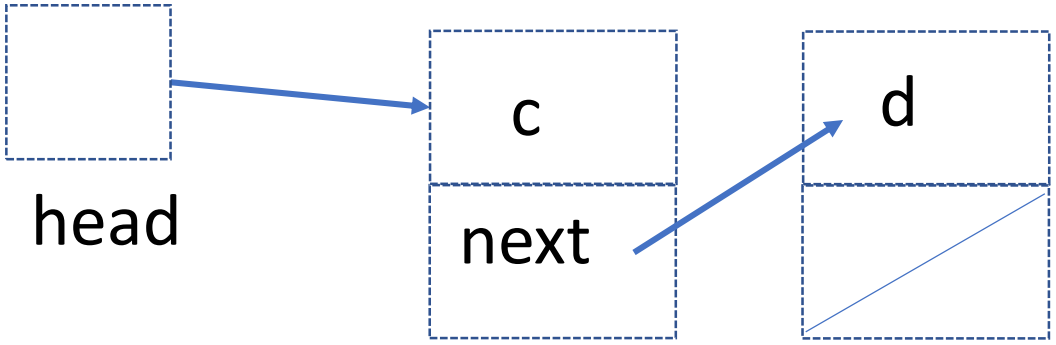
Push(d)

Stack Implementation using Linked Lists



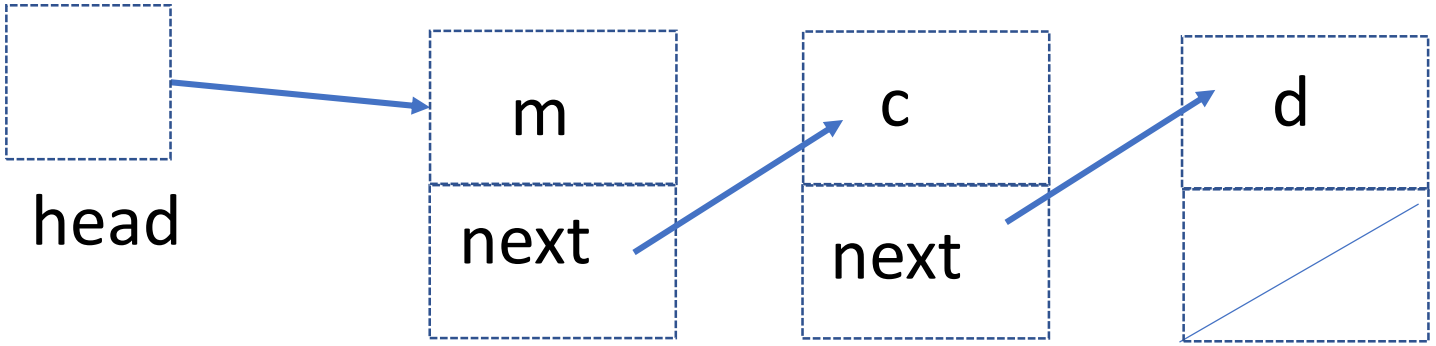
Push(d) , Push(c)

Stack Implementation using Linked Lists



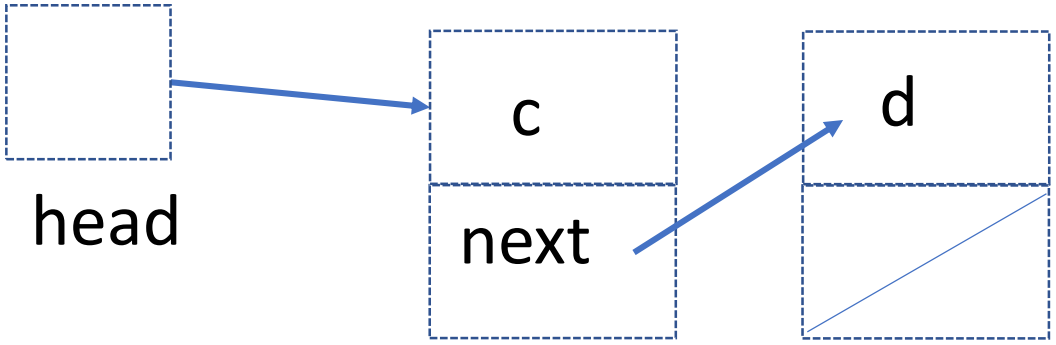
Push(d) , Push(c), Push(m)

Stack Implementation using Linked Lists



Push(d) , Push(c), Push(m), Pop()

Stack Implementation using Linked Lists



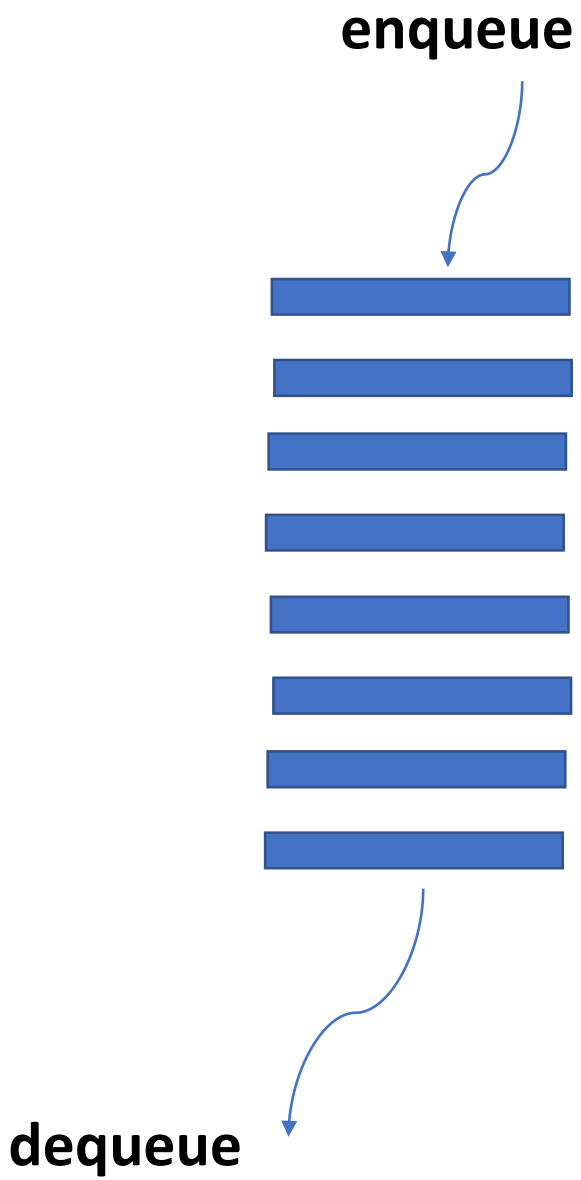
Push(d) , Push(c), Push(m) , Pop() , IsEmpty()

No, if head is not Null

Queue

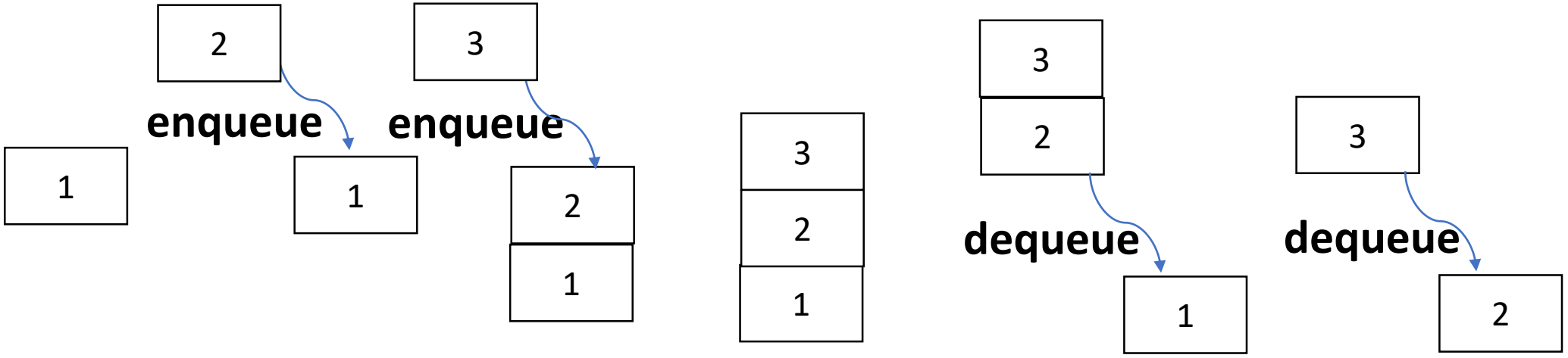
It stores items in a First-In-First-Out (FIFO) manner

- The least recently added item is removed first
- the *insert/add* and *delete* operations are often called *enqueue()* and *dequeue()*

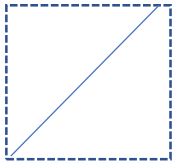


Queue - Operations

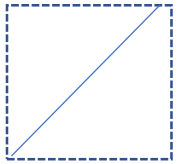
- **Enqueue (key):** Adds key to the queue/collection
- **Key Dequeue():** returns and removes least recently-added key from the queue
- **Boolean Empty():** are there any elements in the queue?



Queue Implementation using Linked List



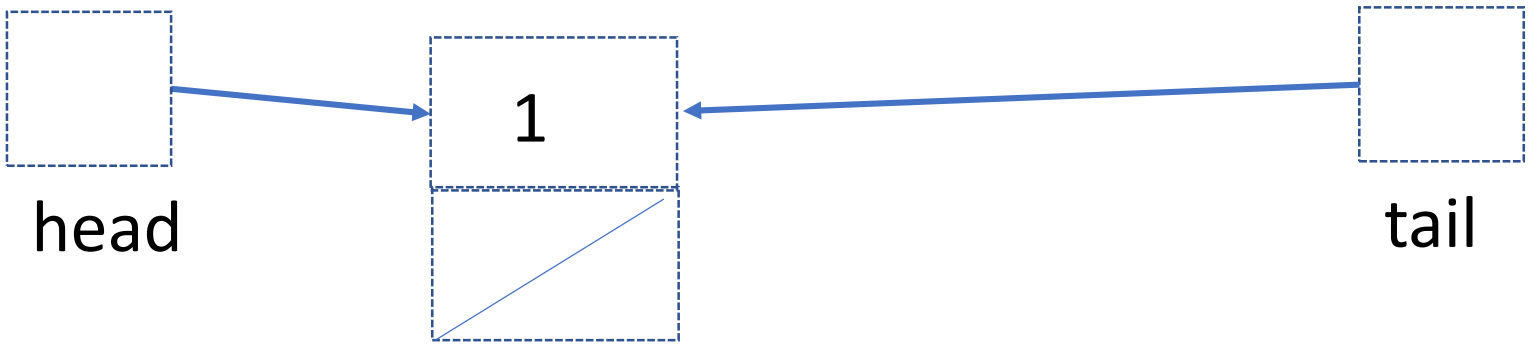
head



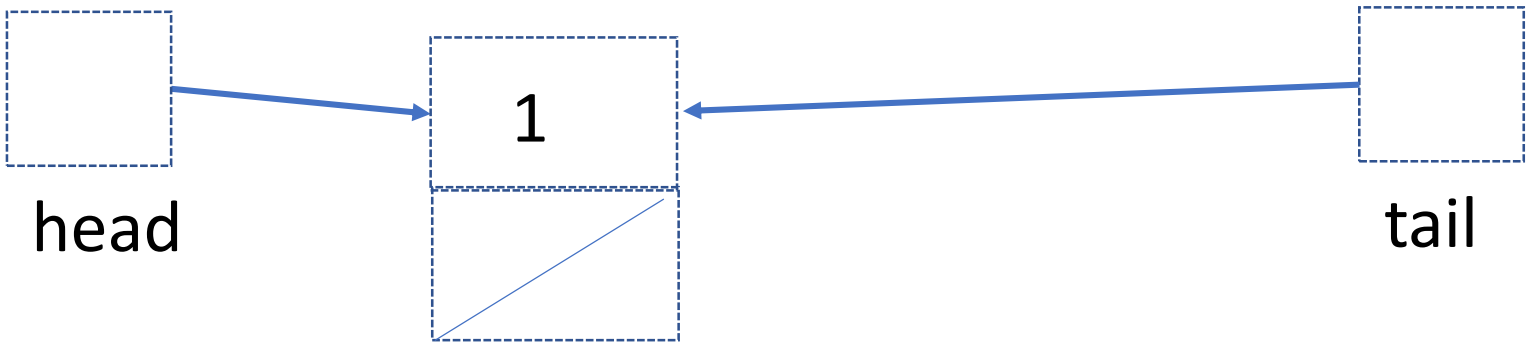
tail

Enqueue (1)

Queue Implementation using Linked List

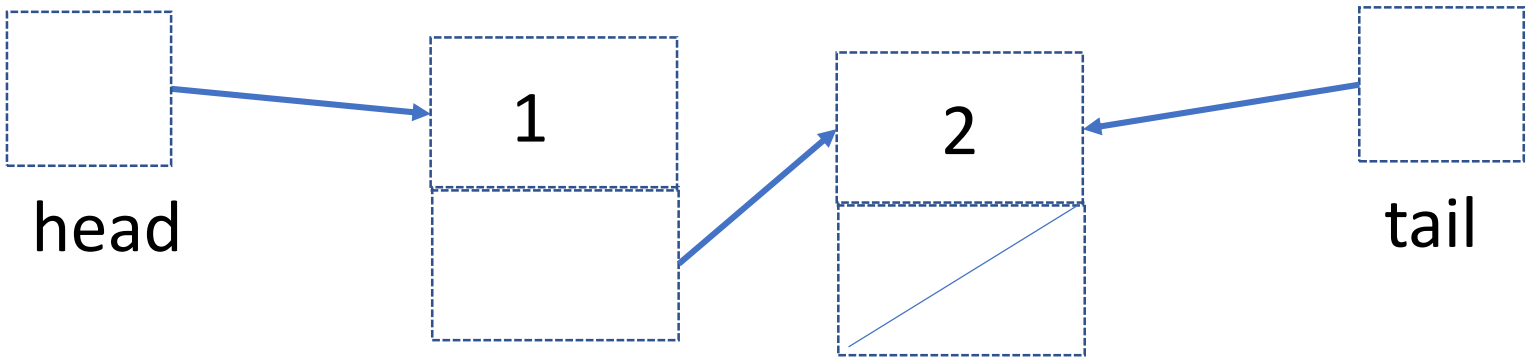


Queue Implementation using Linked List



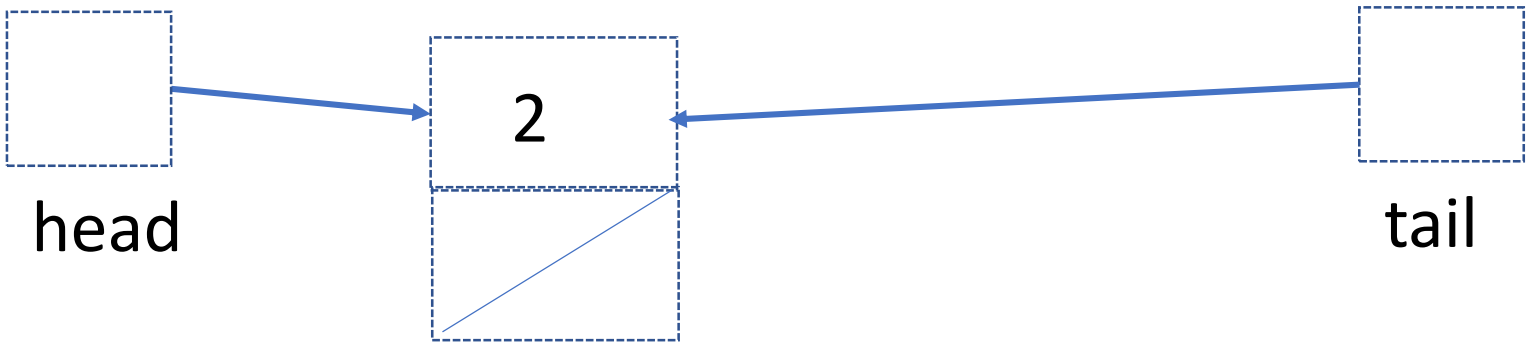
Enqueue (2)

Queue Implementation using Linked List



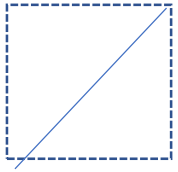
Enqueue (2)

Queue Implementation using Linked List

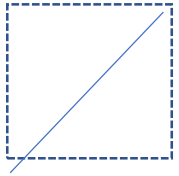


Dequeue ()

Queue Implementation using Linked List



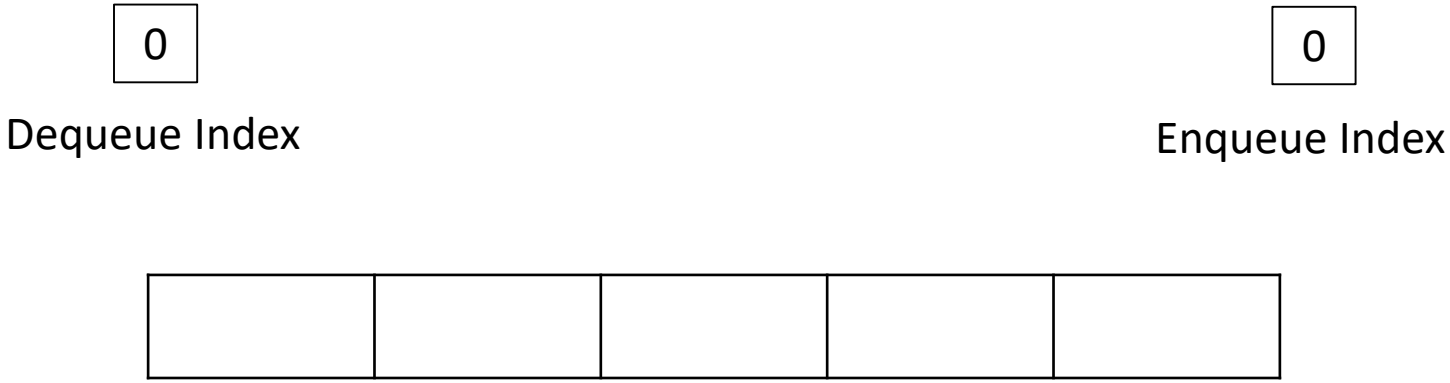
head



tail

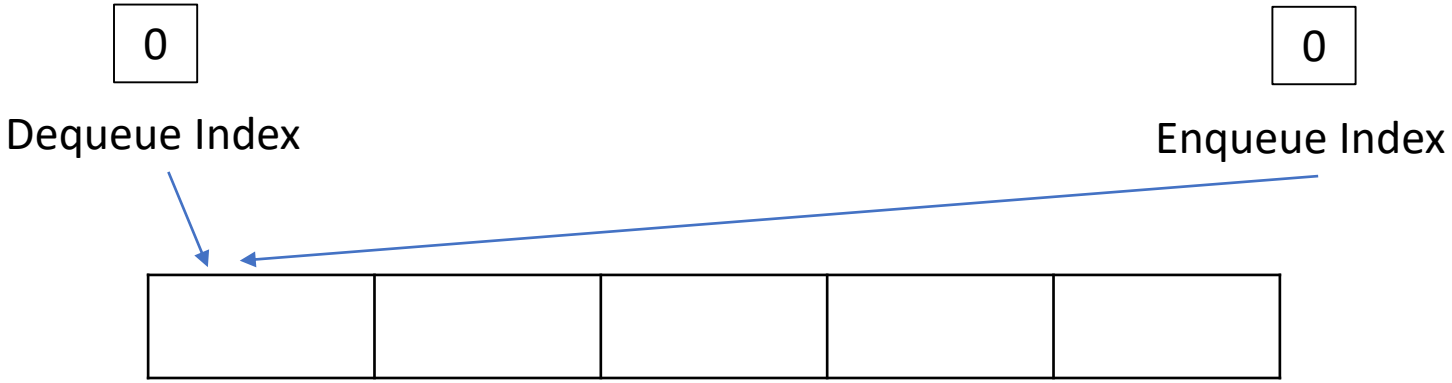
Dequeue ()

Queue Implementation using Array

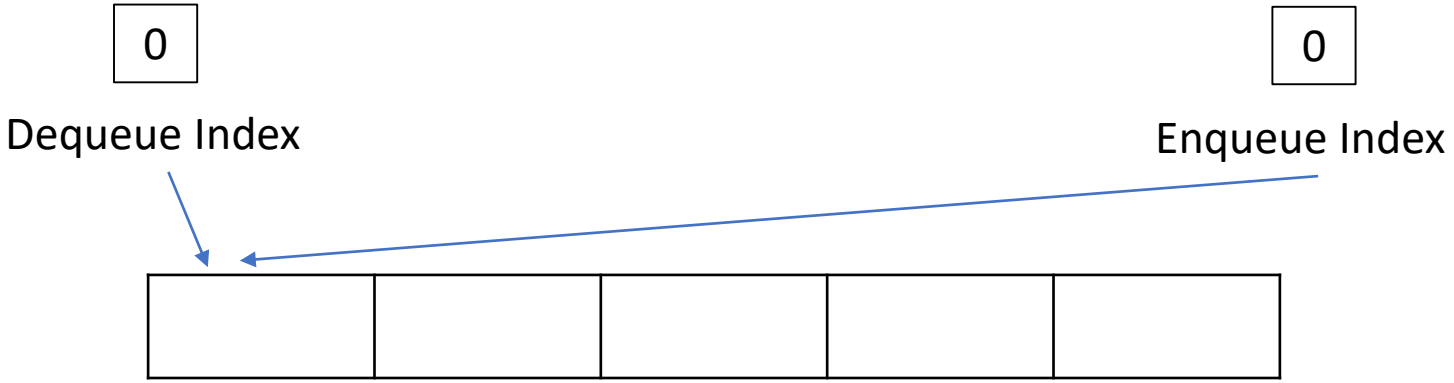



```
If Enqueue Index = Dequeue Index
        queue is empty
```

Queue Implementation using Array

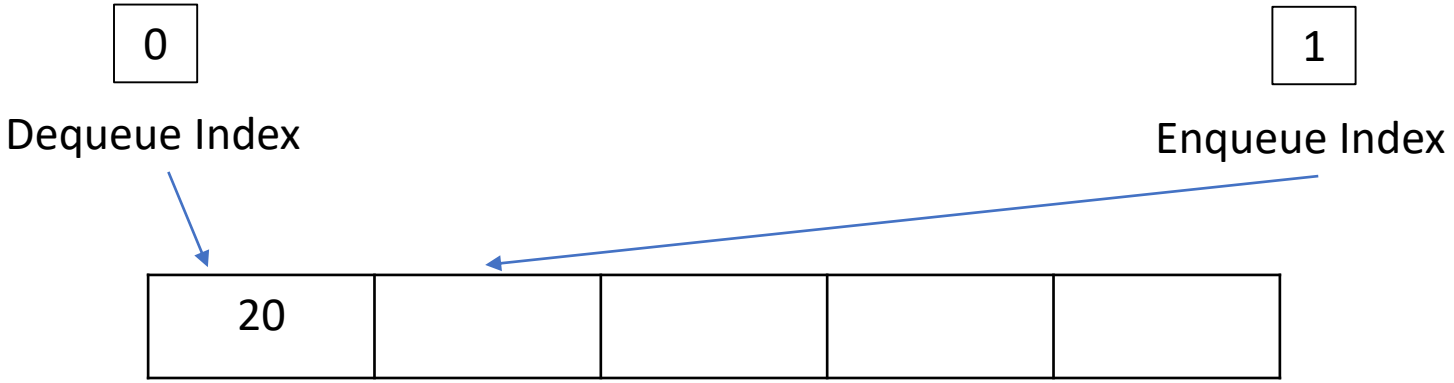


Queue Implementation using Array



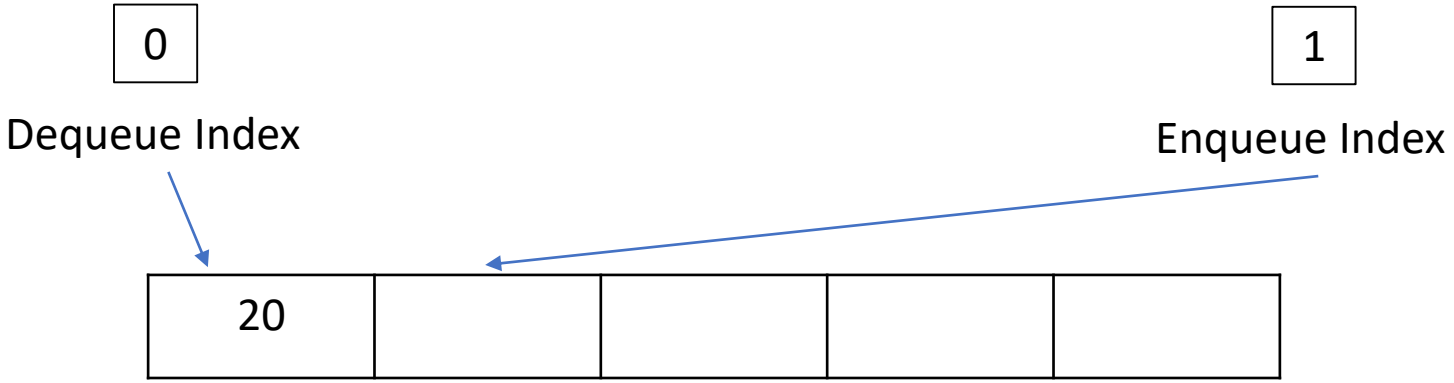
Enqueue(20)

Queue Implementation using Array



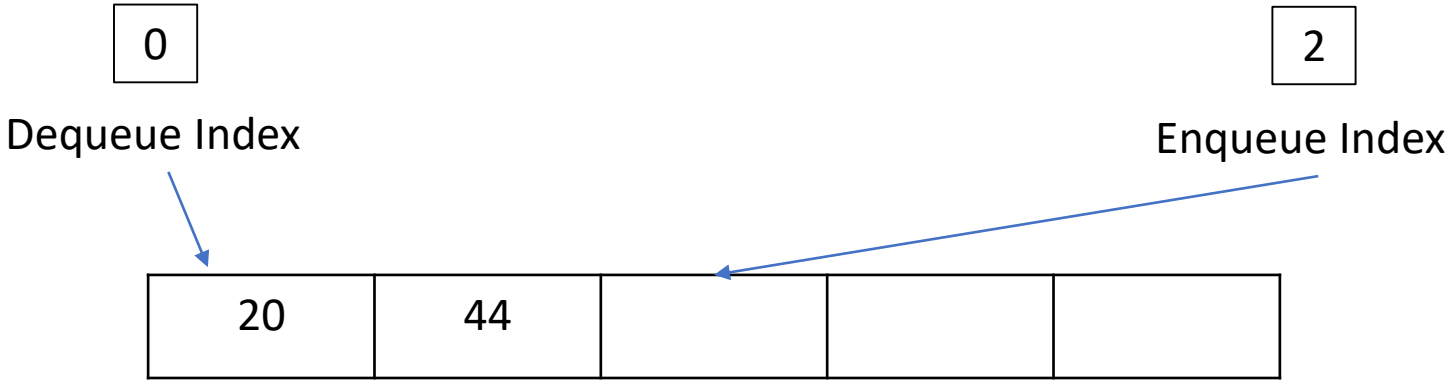
Enqueue(20)

Queue Implementation using Array



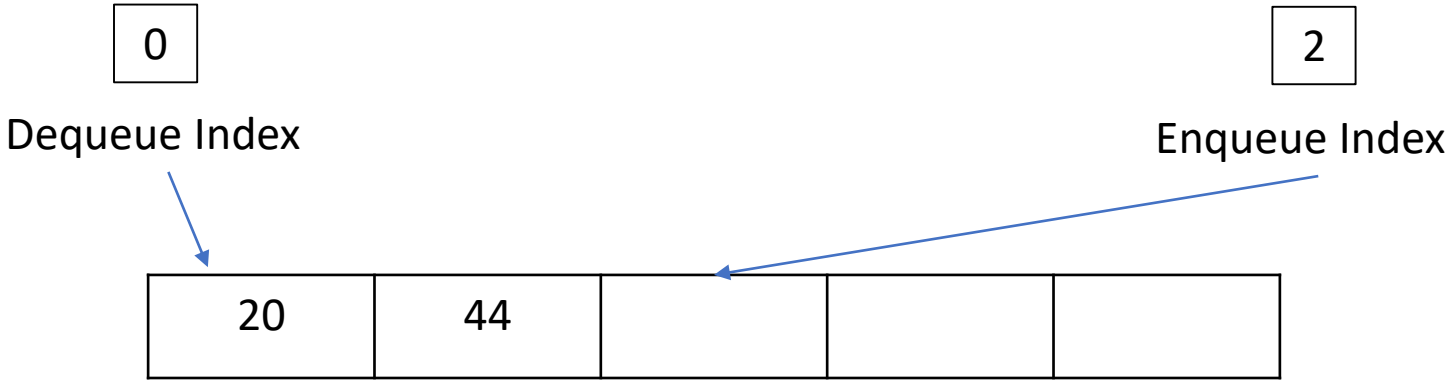
Enqueue(44)

Queue Implementation using Array



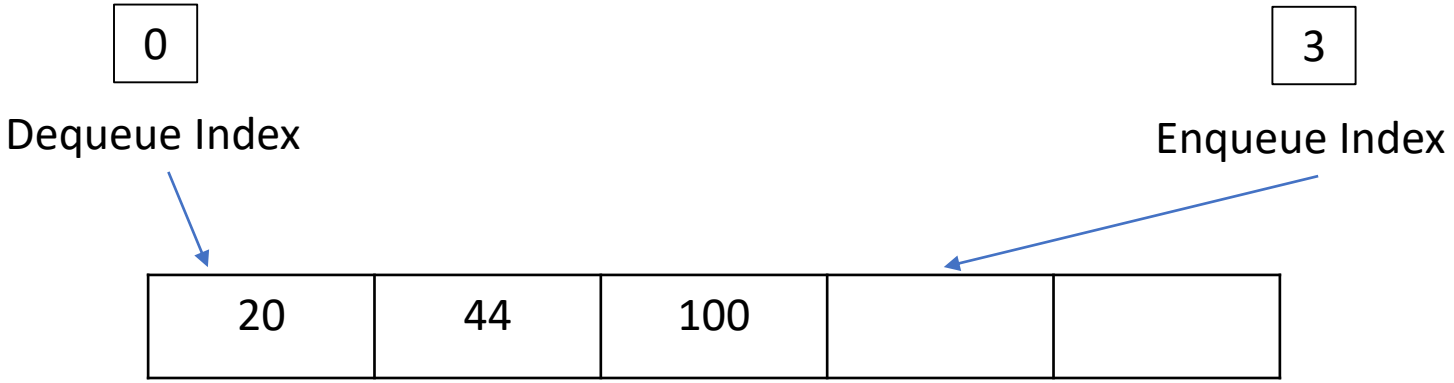
Enqueue(44)

Queue Implementation using Array



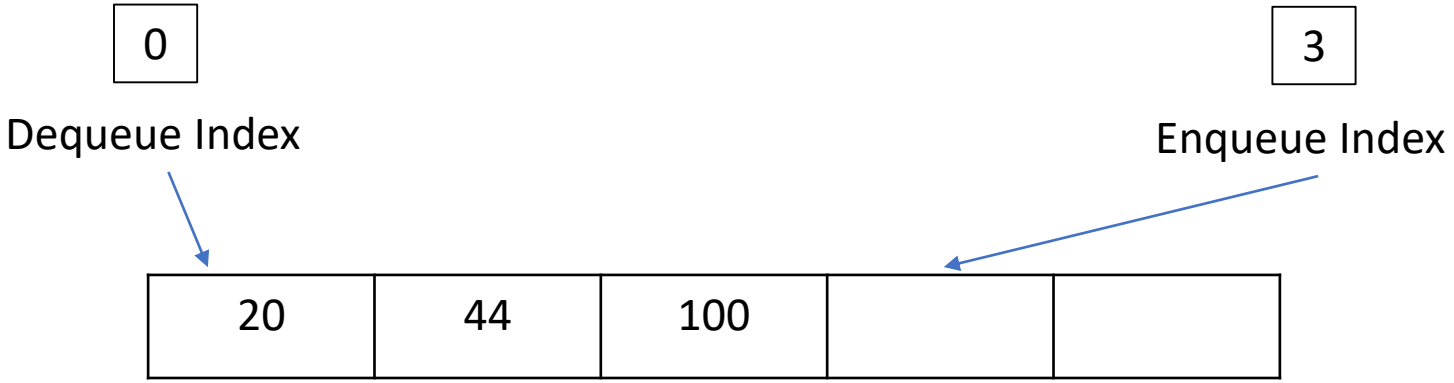
Enqueue(100)

Queue Implementation using Array



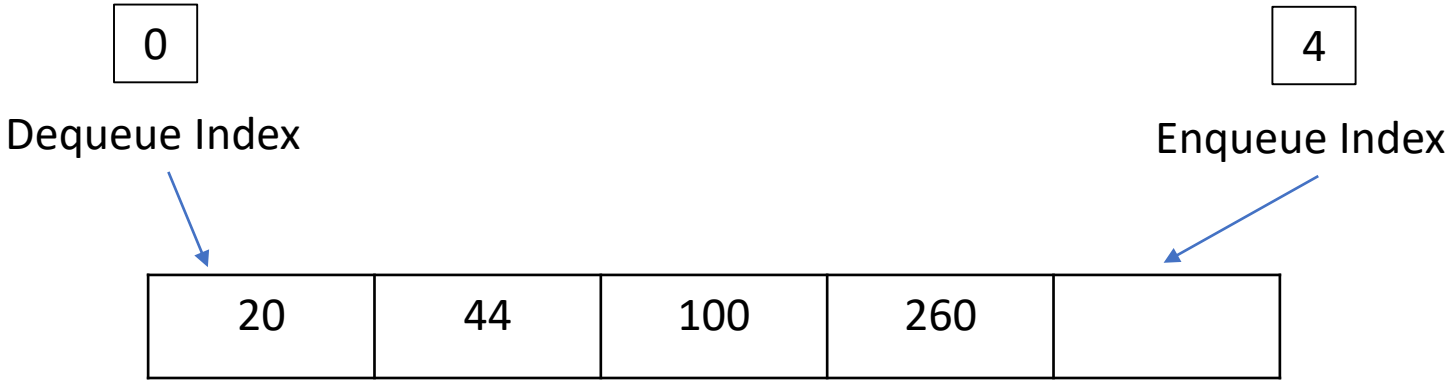
Enqueue(100)

Queue Implementation using Array



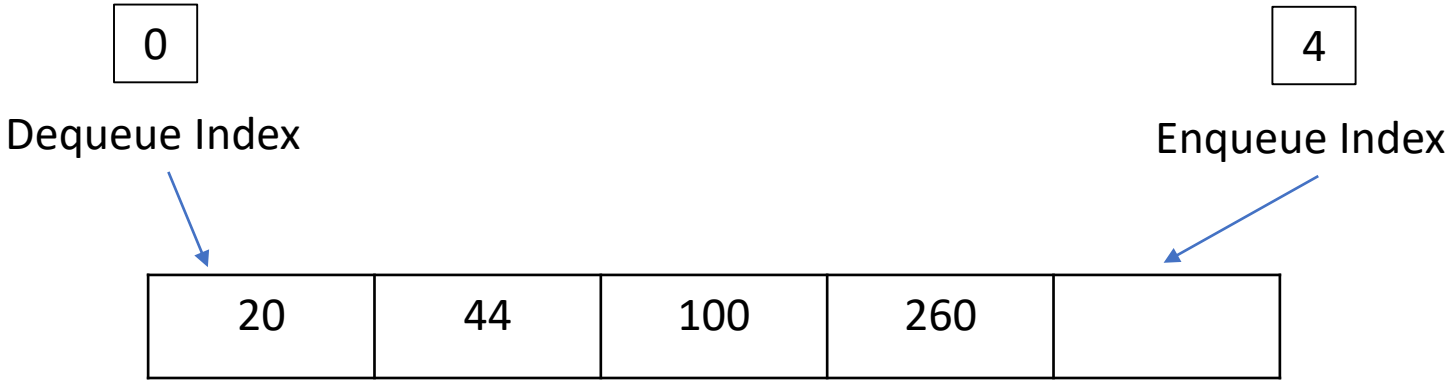
Enqueue(260)

Queue Implementation using Array



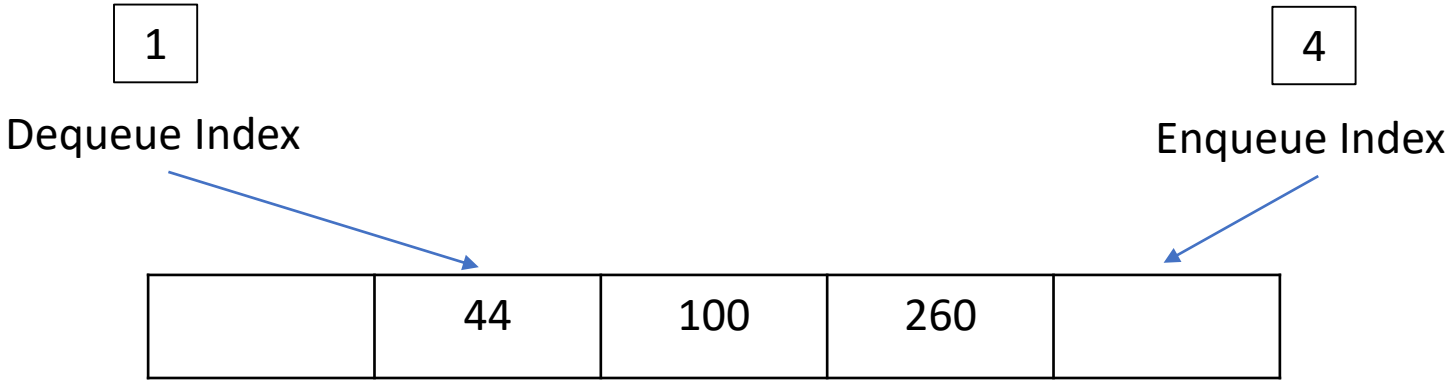
Enqueue(260)

Queue Implementation using Array



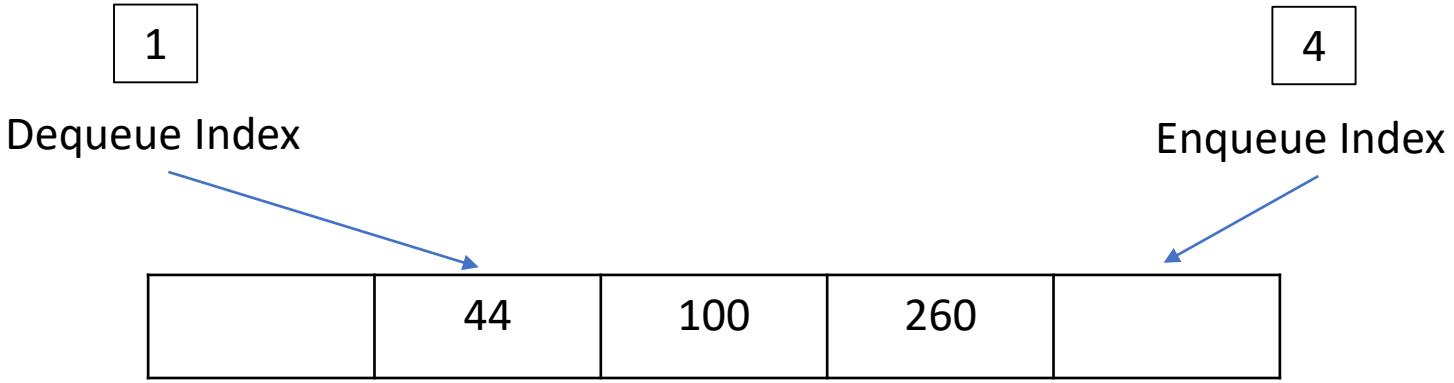
Dequeue()

Queue Implementation using Array



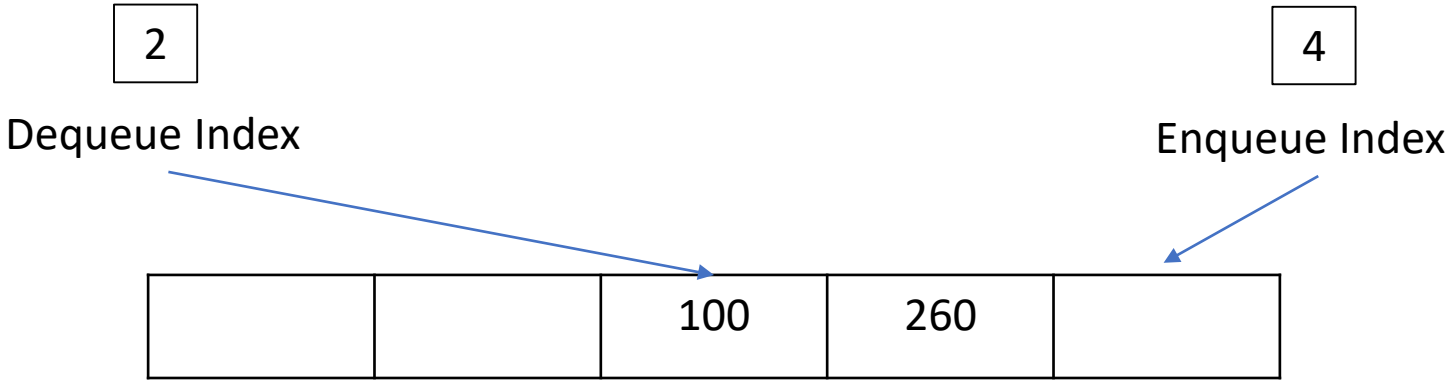
Dequeue()

Queue Implementation using Array

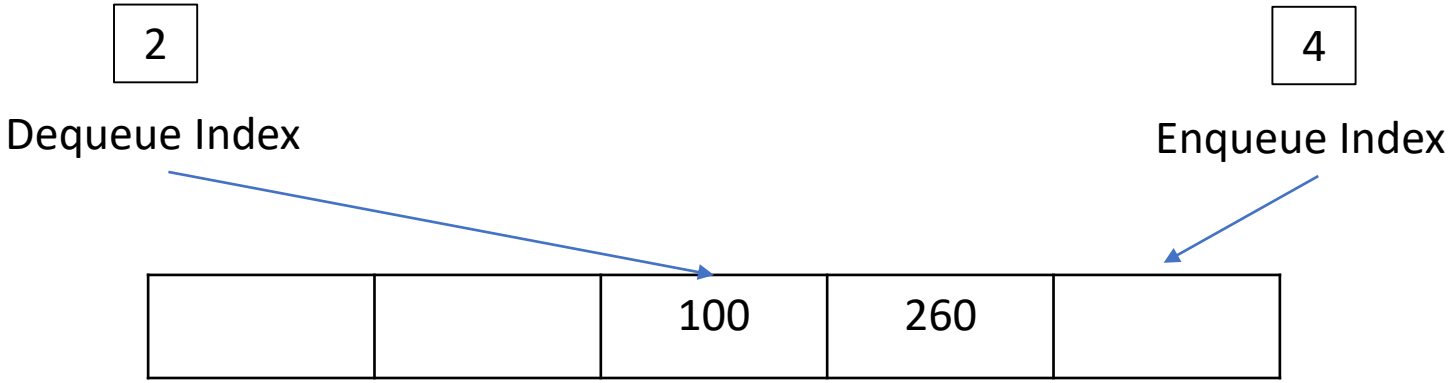


Dequeue()

Queue Implementation using Array

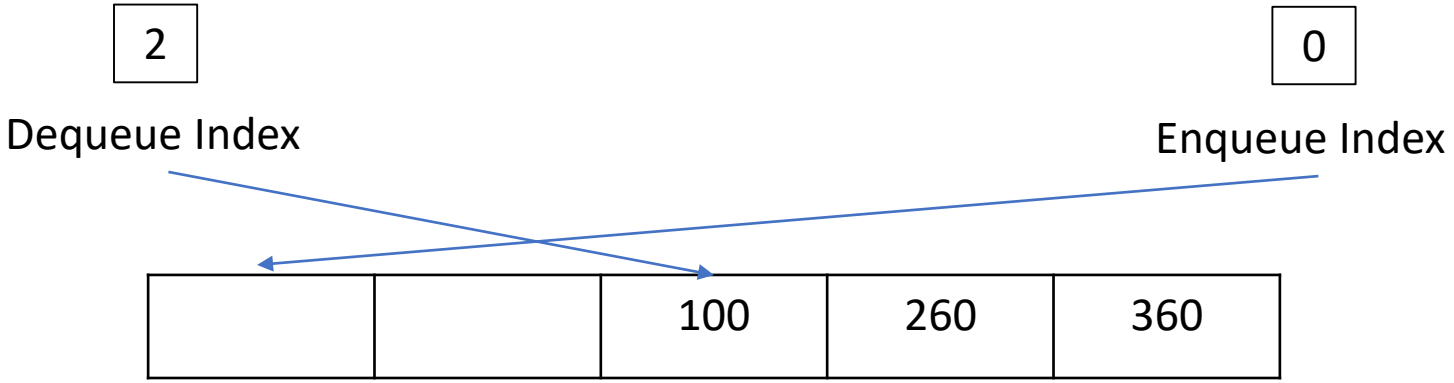


Queue Implementation using Array



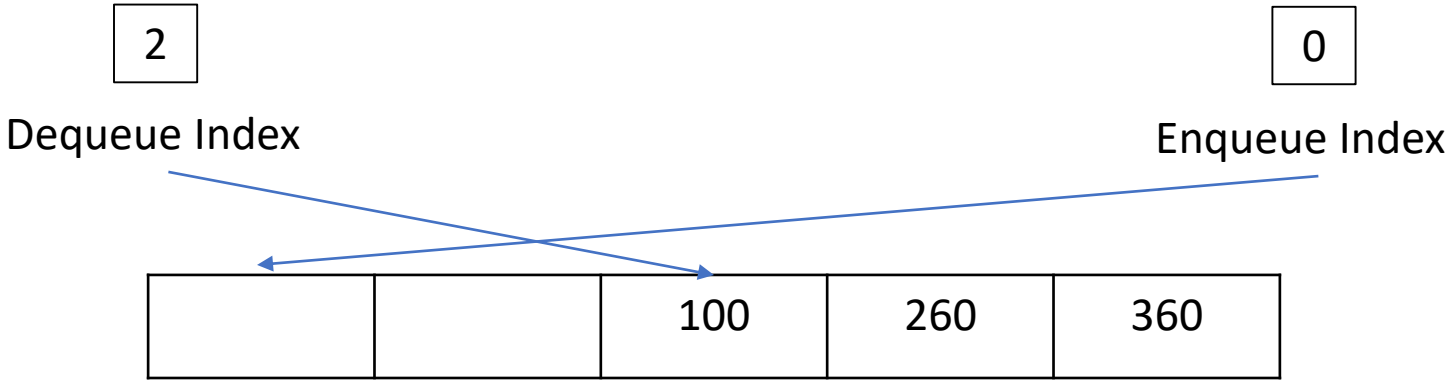
Enqueue(360)

Queue Implementation using Array



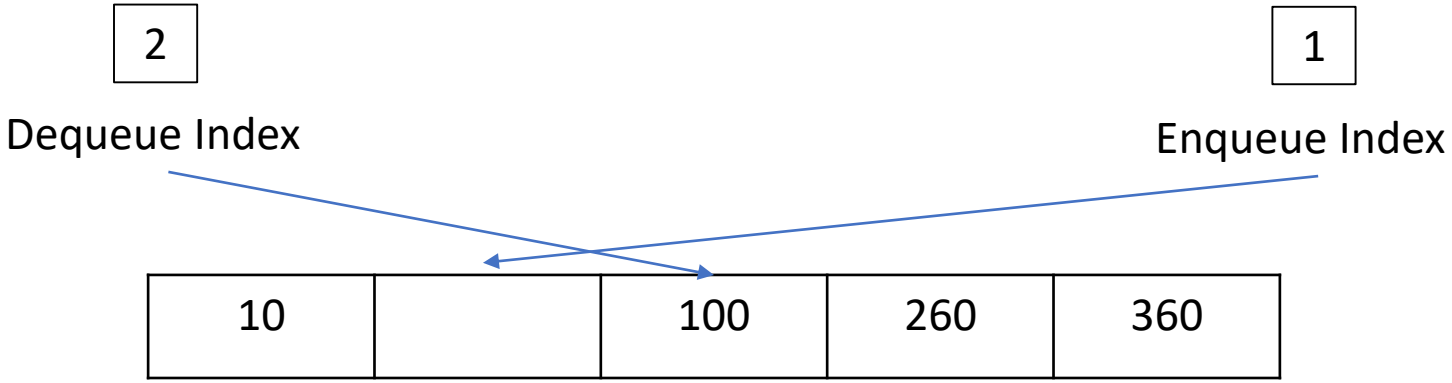
Enqueue(360)

Queue Implementation using Array



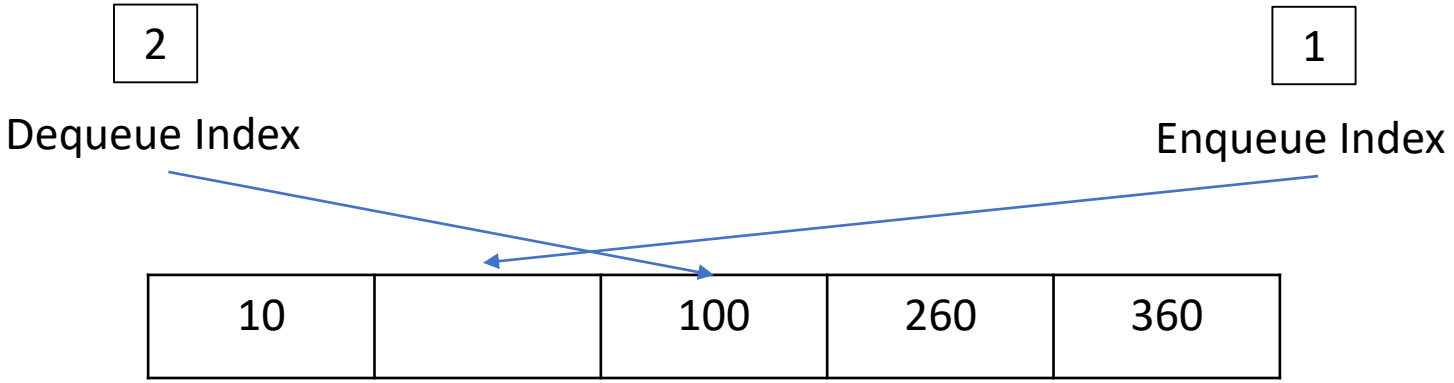
Enqueue(10)

Queue Implementation using Array



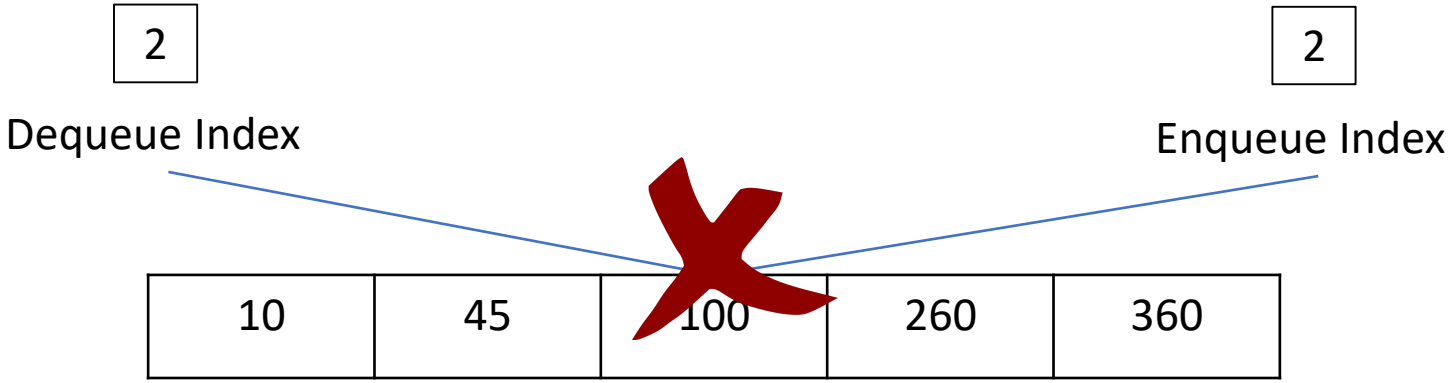
Enqueue(10)

Queue Implementation using Array



Enqueue(45)

Queue Implementation using Array



Enqueue(45)

ERROR

Recap - Stack

- Stacks are known as LIFO
- Implementation of a stack:
  - With an array
  - With a linked list

- Push, Pop, Top and Empty run in $O(1)$

Recap - Queue

- Queues are known as FIFO
- Implementation of a queue:
  - With an array
  - With a linked list with tail pointer

- Enqueue, Dequeue, and Empty run in O(1)

Next week lecture …

Non-Linear Data Structure