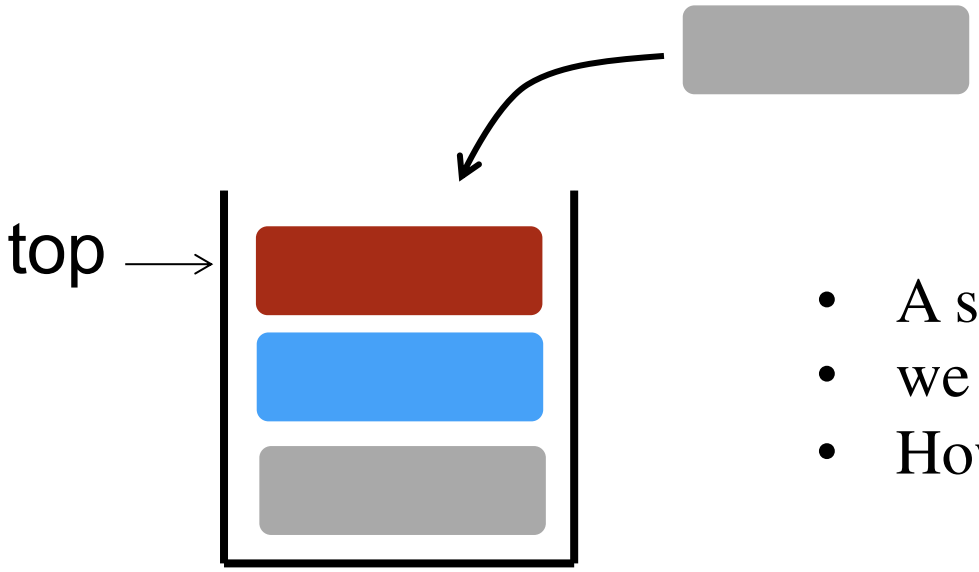
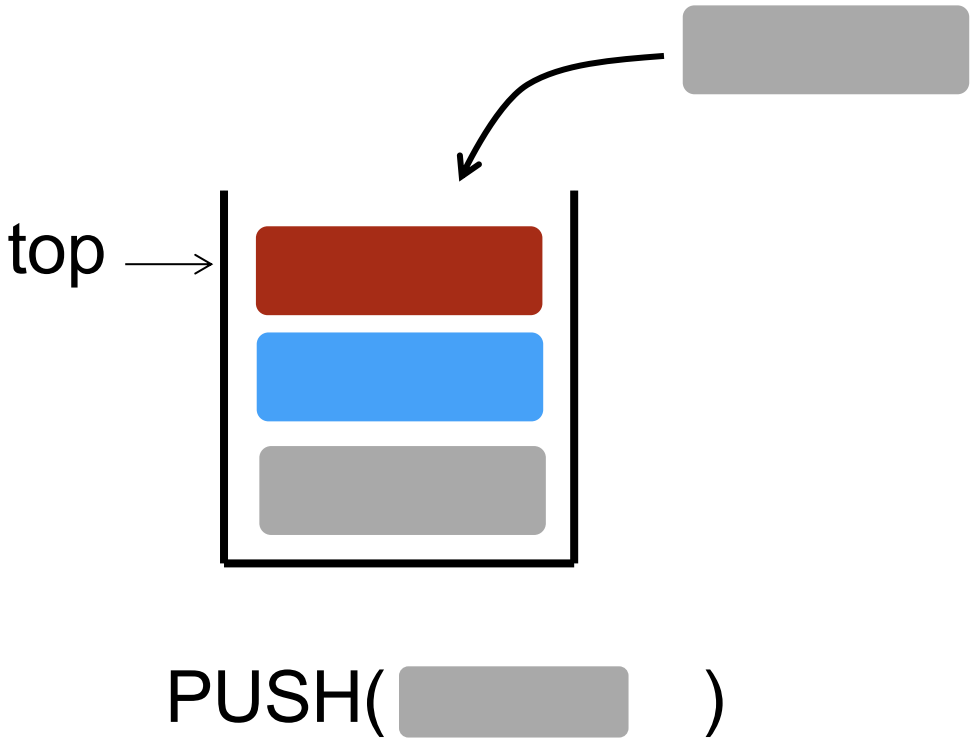
Stacks intro

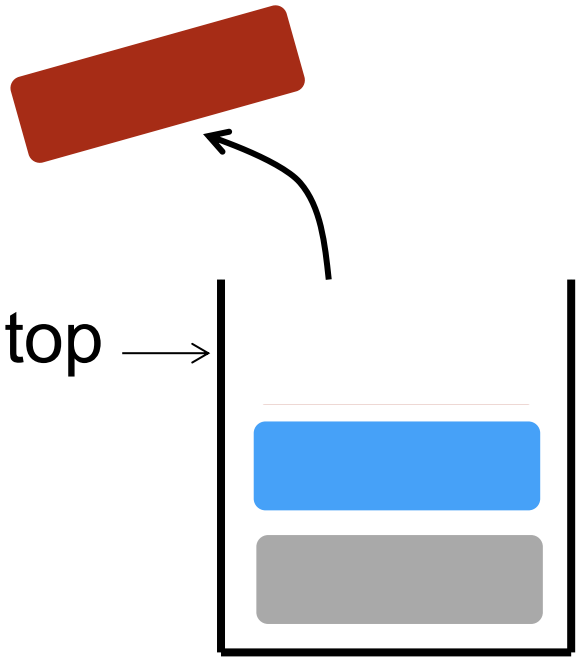
- Definition a stack data structure
- Stack operation
  - PUSH()
  - POP()
  - PEEK()
  - ISEMPTHY()



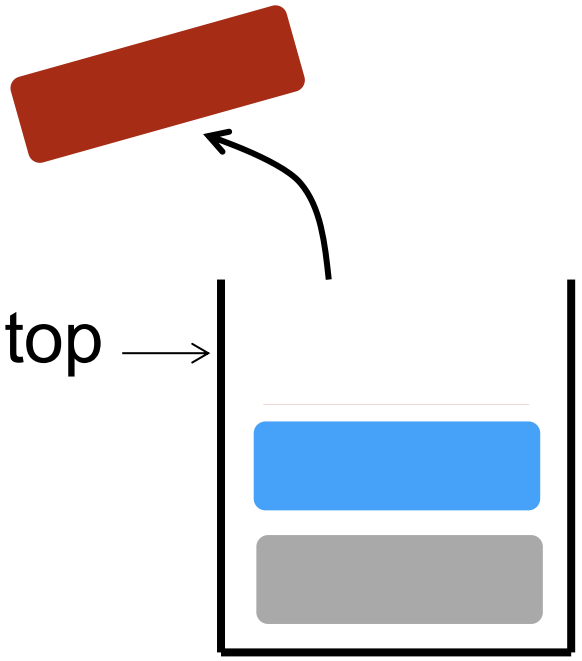
- A stack data structure works works like a physical stack
- we can insert elements at any position of a linked list
- However, elements can only inserted at the top of the stuck.



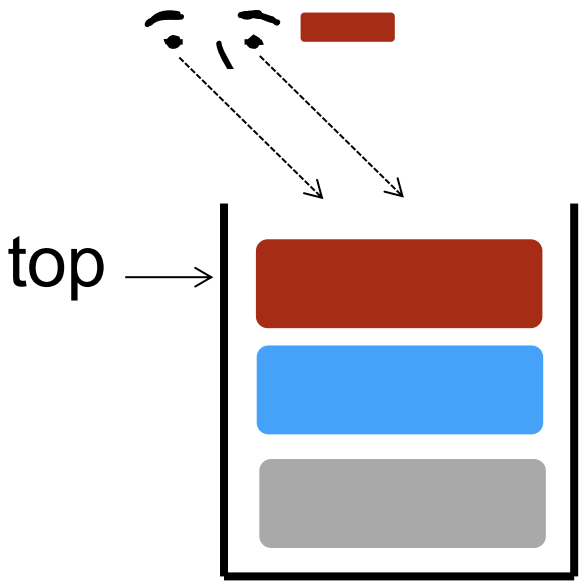
The insertion operation is called PUSH



Objects can only be removed the top of the stack.



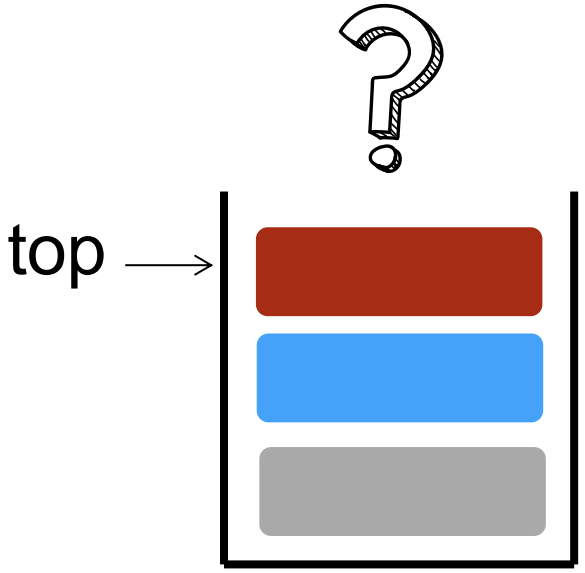
The operation to remove an object from a stack is called POP



PEEK( )

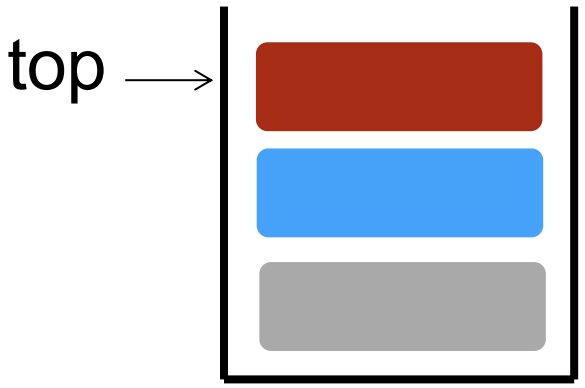
The top of the stack element is the only element that can be accessed

The operation PEEK() returns the value of the element of the stack without removing it from the stack.



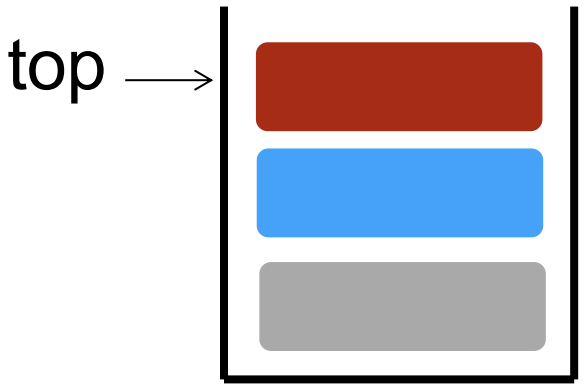
ISEMPTY( )

The final operation is **ISEMPTHY(), which returns true if the the stack empty and false otherwise.**



A Stack is linear data structure with constraints on how data is accessed, inserted and removed in it.

- You can only insert, remove or access the element stored at the top of the stack



Stacks stack data structure has a number of applications

- Verification of pairs of element such as parenthesis.
- Undoing actions in inverse chronological order

1st application

Verification of pairs of elements such

```
class Node
{
public:
          int data;
          shared_ptr<Node> next;

          Node(int d) {
             data = d;
           next = nullptr;
          }
//error due to missing parenthesis here
```

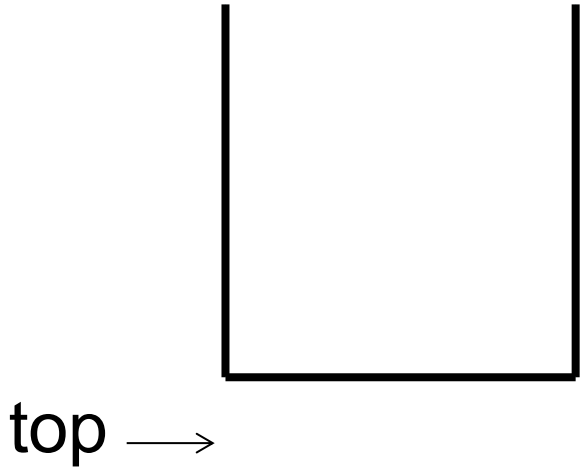
For example: When your compiler alerts that there is a missing closing bracket in your code

The compiler may use a stack to carry on the parenthesis varication task.

```
class Node
{
public:
          int data;
          shared_ptr<Node> next;

          Node(int d) {
             data = d;
           next = nullptr;
          }
//error due to missing parenthesis here
```

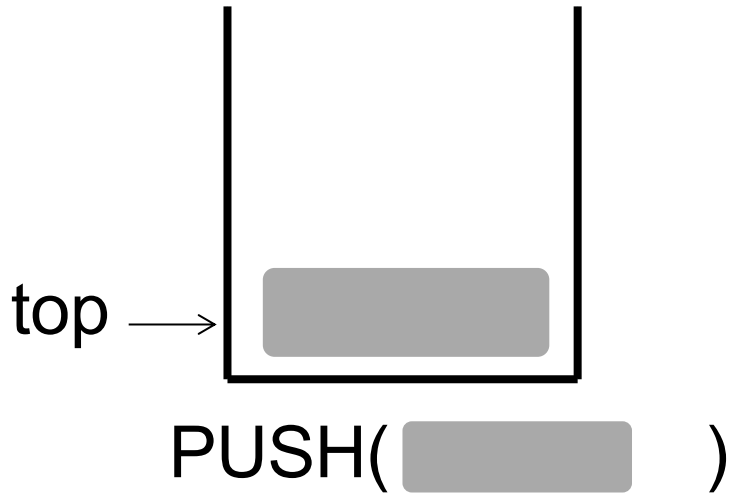
No parenthesis here

- Every time an opening bracket is found an element is pushed in the stack
- Every time a closing bracket is found and element is removed from the stack



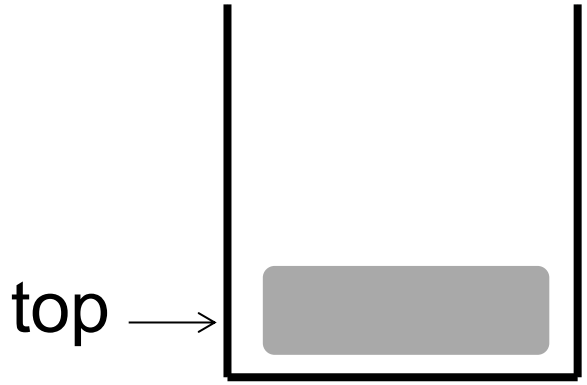
```
class Node
{
public:
          int data;
          shared_ptr<Node> next;

          Node(int d) {
             data = d;
           next = nullptr;
          }
//error due to missing parenthesis here
```

Opening parenthesis found, PUSH

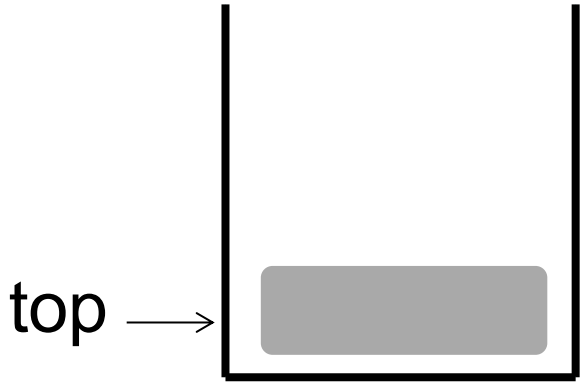


```
class Node
{
public:
          int data;
          shared_ptr<Node> next;

          Node(int d) {
             data = d;
           next = nullptr;
          }
//error due to missing parenthesis here
```

No parenthesis here

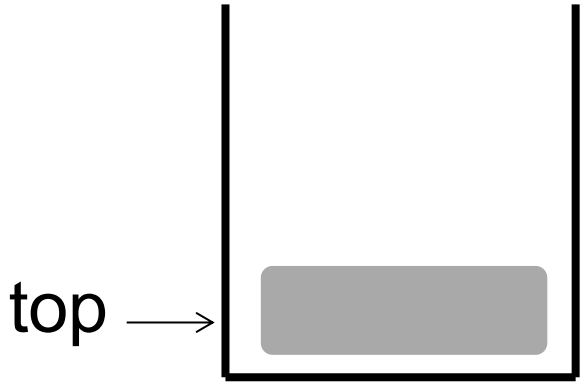


```
class Node
{
public:
          int data;
          shared_ptr<Node> next;

          Node(int d) {
             data = d;
           next = nullptr;
          }
//error due to missing parenthesis here
```



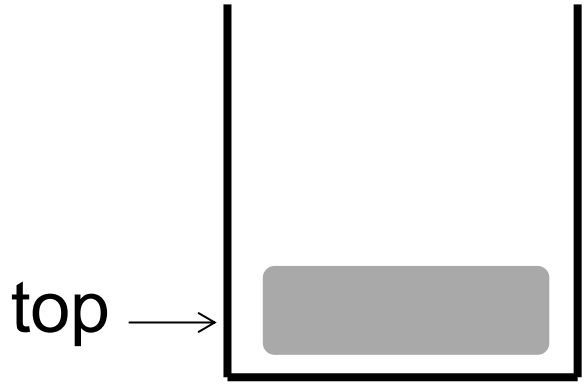
```
class Node
{
public:
          int data;
          shared_ptr<Node> next;

          Node(int d) {
             data = d;
           next = nullptr;
          }
//error due to missing parenthesis here
```

No parenthesis here



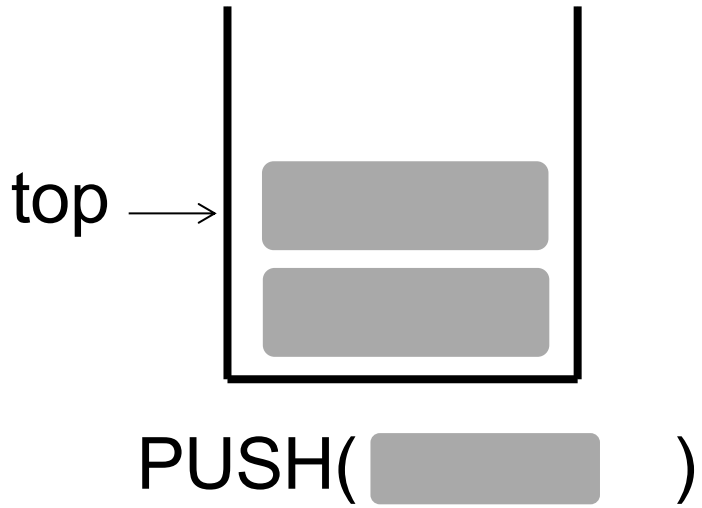
```
class Node
{
public:
          int data;
          shared_ptr<Node> next;

          Node(int d) {
             data = d;
           next = nullptr;
          }
//error due to missing parenthesis here
```

No parenthesis here



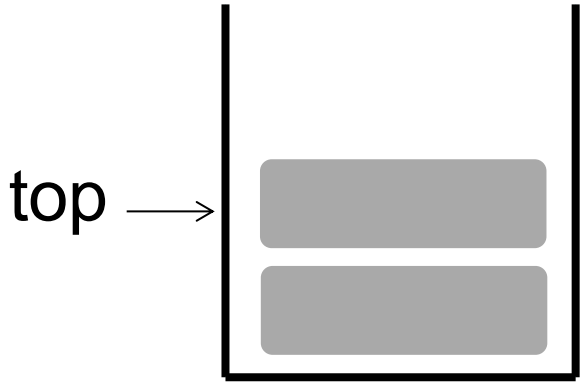
```
class Node
{
public:
          int data;
          shared_ptr<Node> next;

          Node(int d) {
             data = d;
           next = nullptr;
          }
//error due to missing parenthesis here
```

Opening parenthesis found, PUSH



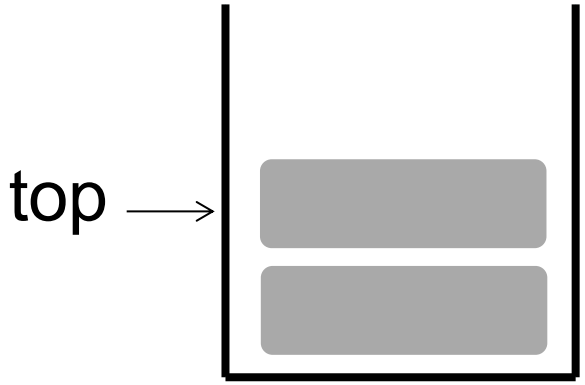
```
class Node
{
public:
          int data;
          shared_ptr<Node> next;

          Node(int d) {
             data = d;
           next = nullptr;
          }
//error due to missing parenthesis here
```

No parenthesis here



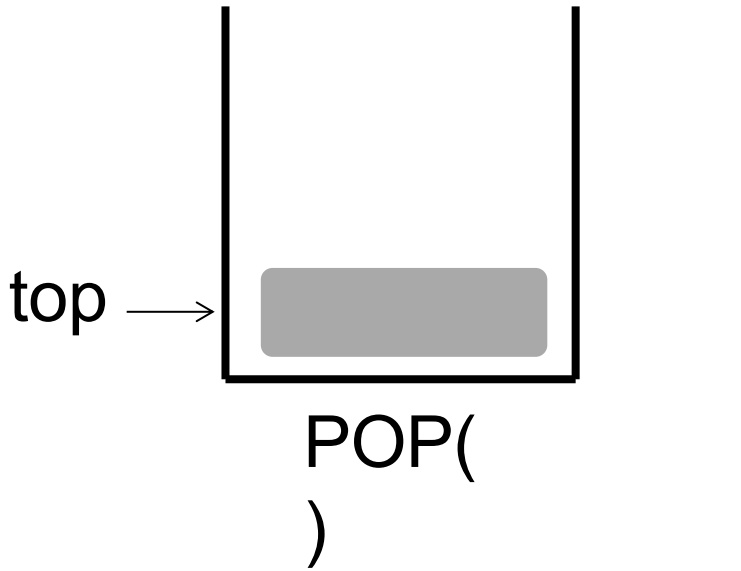
```
class Node
{
public:
          int data;
          shared_ptr<Node> next;

          Node(int d) {
             data = d;
           next = nullptr;
          }
//error due to missing parenthesis here
```

No parenthesis here



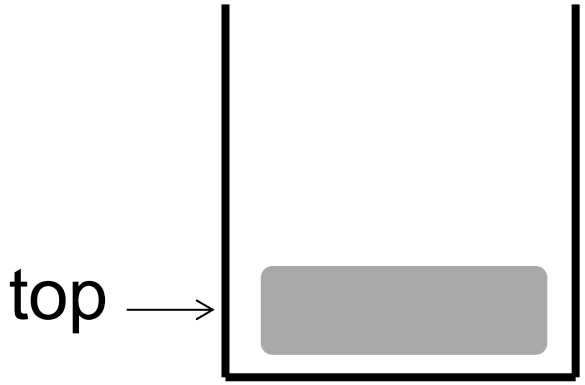
```
class Node
{
public:
          int data;
          shared_ptr<Node> next;

          Node(int d) {
             data = d;
           next = nullptr;
          }
//error due to missing parenthesis here
```

Closing parenthesis found, POP()



```
class Node
{
public:
          int data;
          shared_ptr<Node> next;

          Node(int d) {
             data = d;
           next = nullptr;
          }
```

No parenthesis here

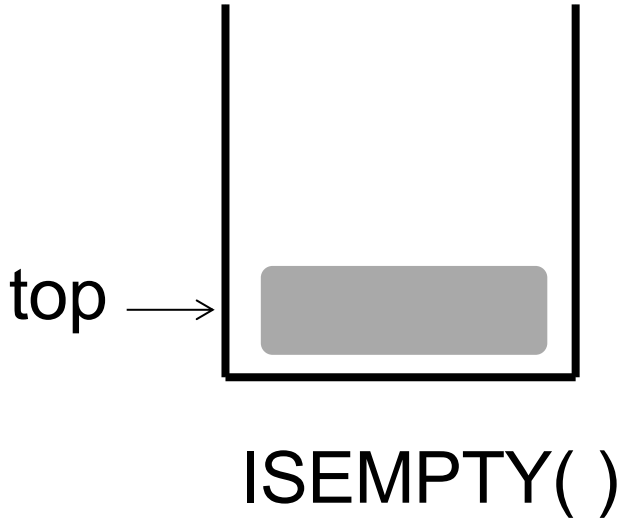


```
class Node
{
public:
          int data;
          shared_ptr<Node> next;

          Node(int d) {
             data = d;
           next = nullptr;
          }
```

ISEMPTY() is executed at the end!

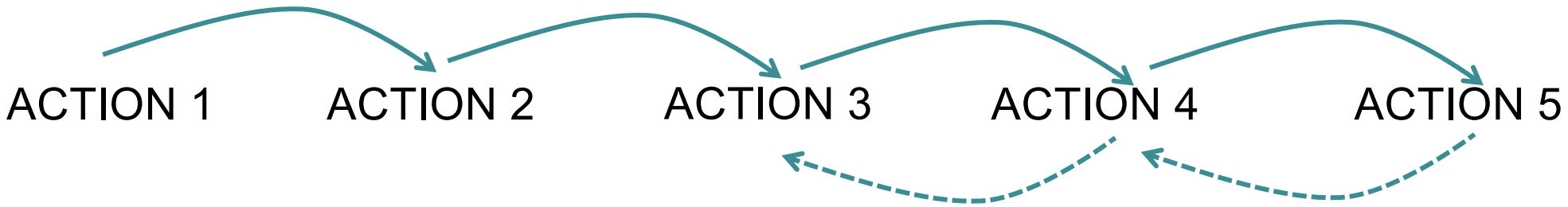
- If it returns true then brackets are balanced
- If if returns false, then there is a missing parenthesis

In this case it will return false to signal there is a missing closing bracket.



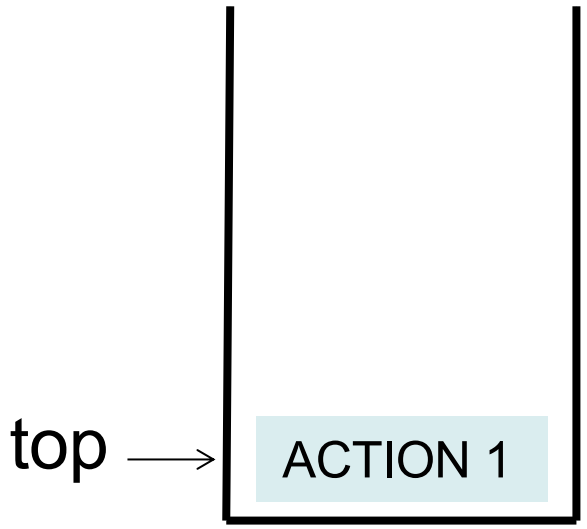
2nd application of stacks: undoing action in reverse chronological order.

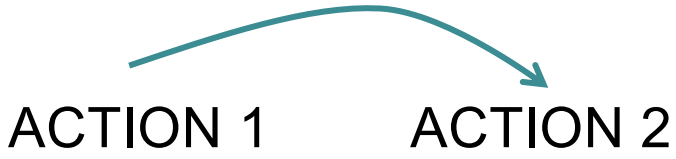
For example: Undo the last change in a text editor

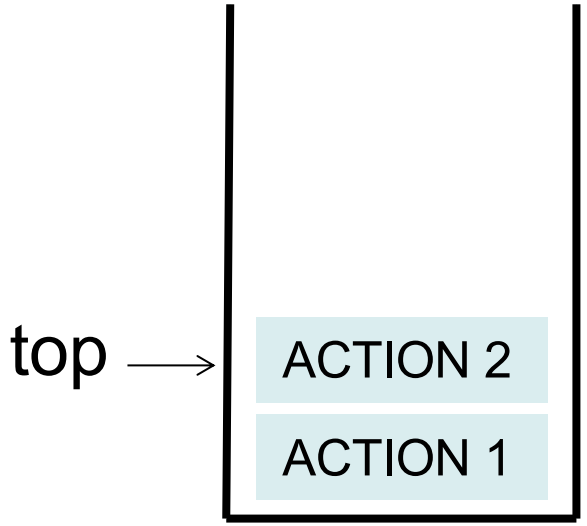


The system keeps a stack where the actions are stored.

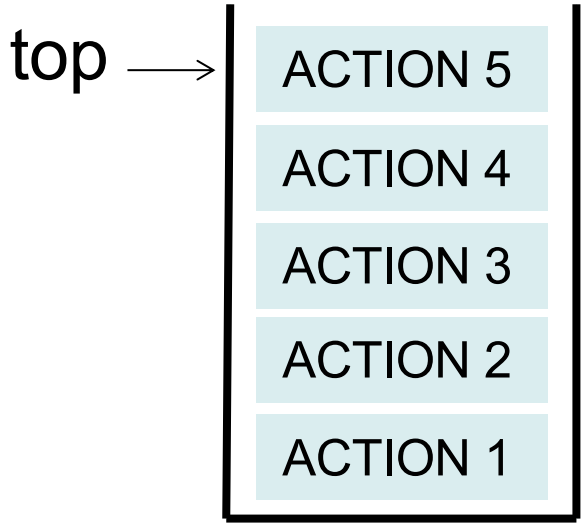
ACTION 1

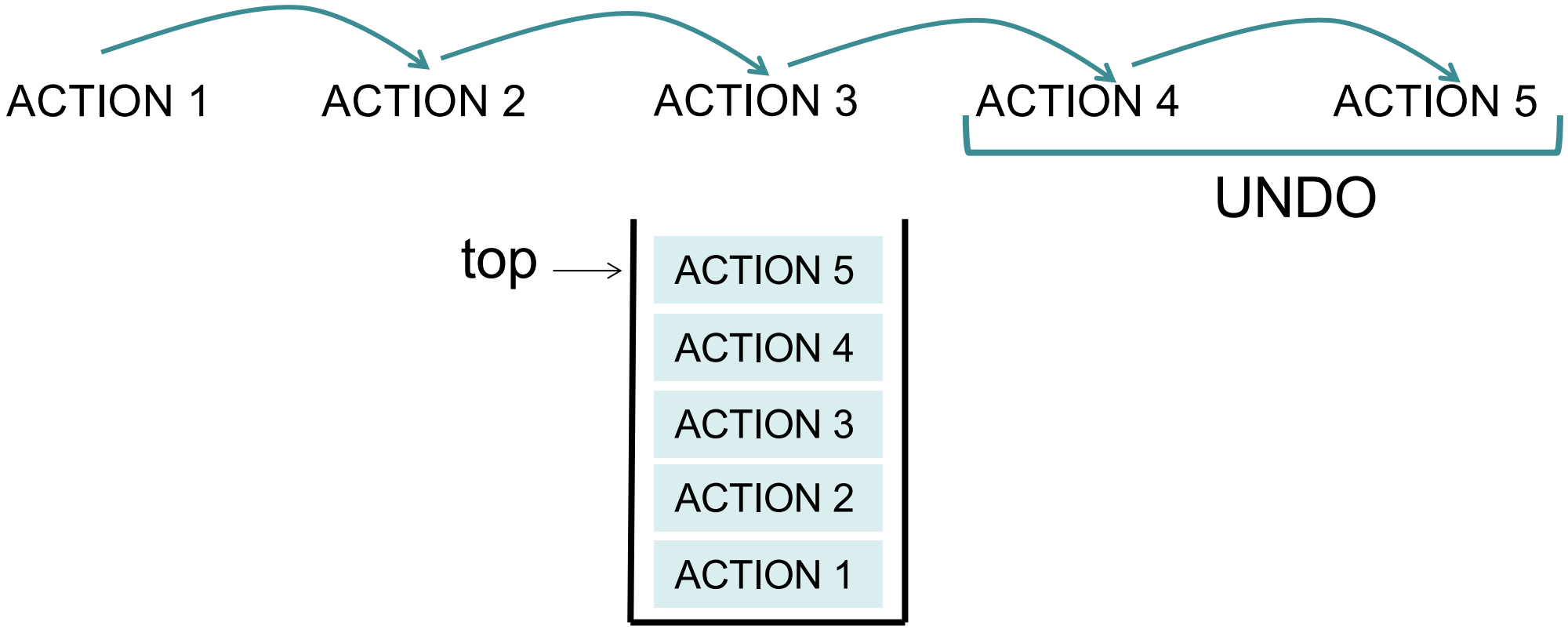


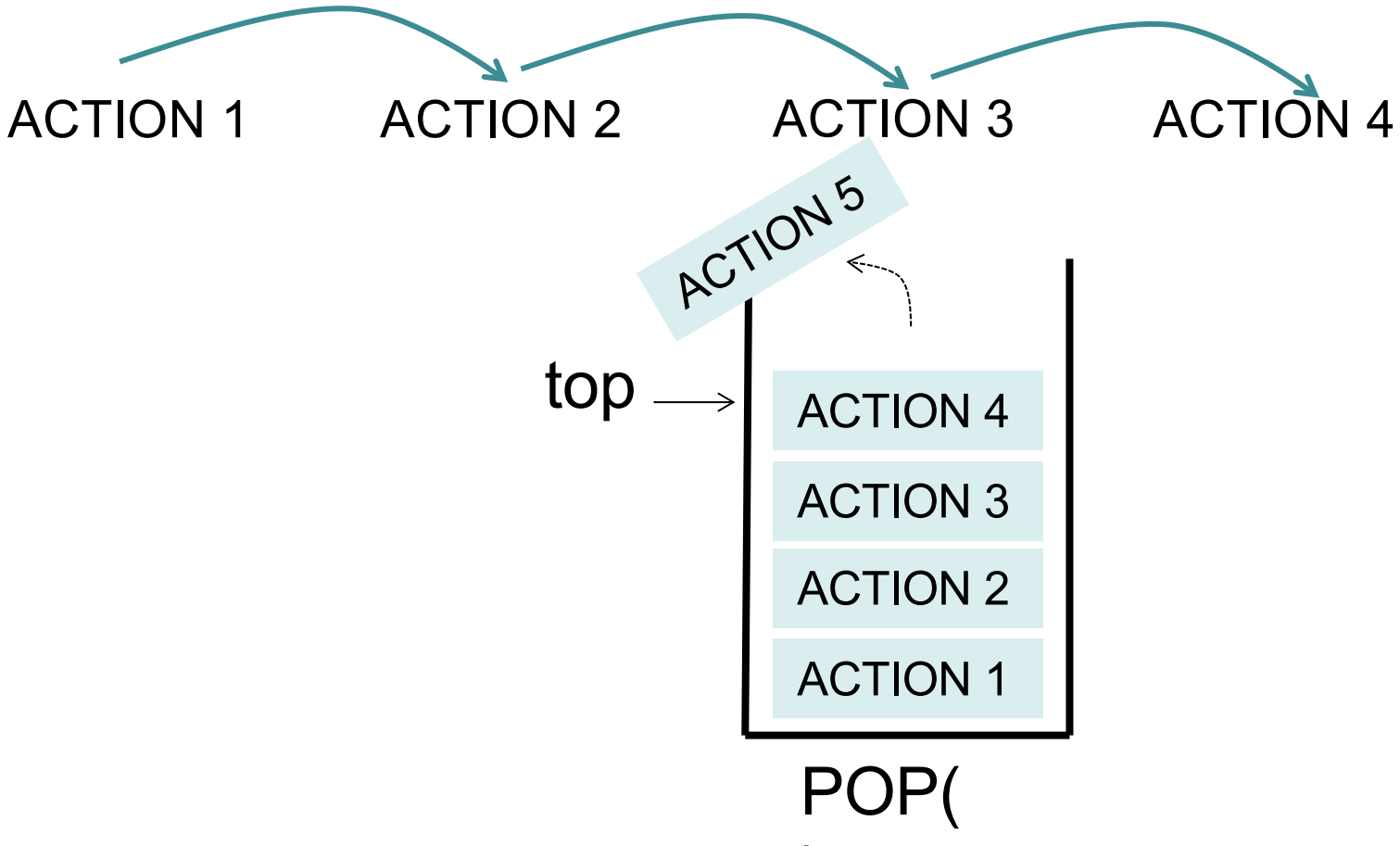

ACTION 1
ACTION 2

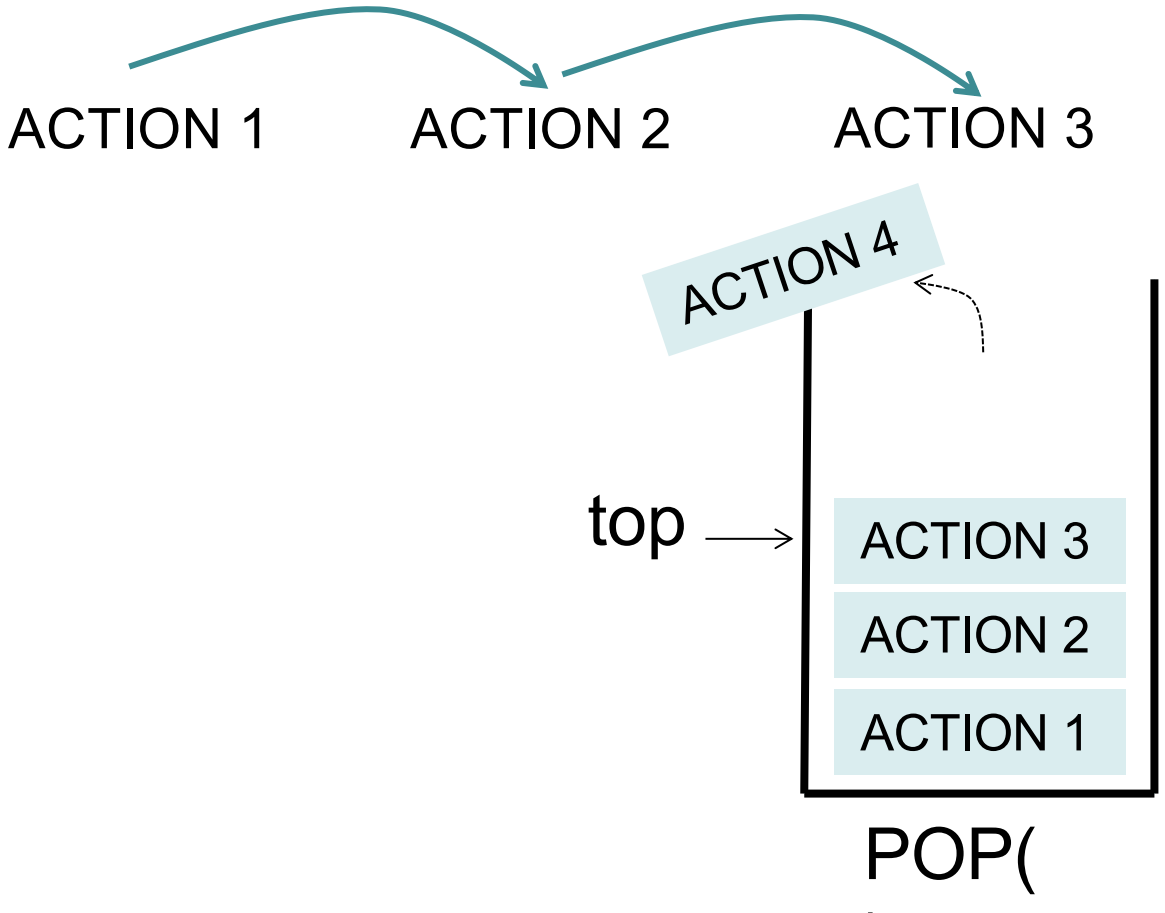

top
ACTION 2
ACTION 1

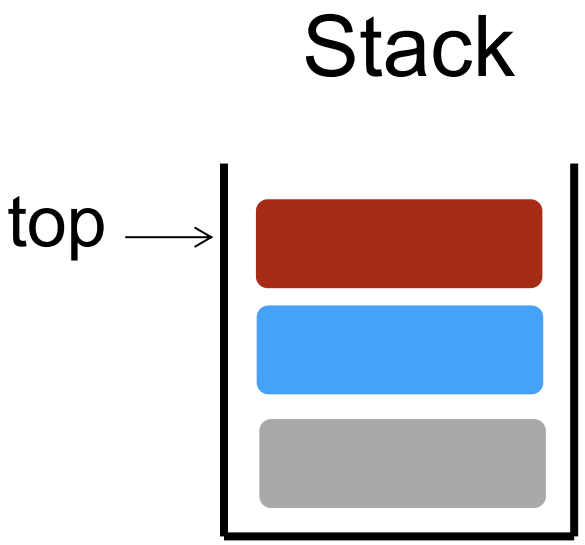

ACTION 1
ACTION 2
ACTION 3
ACTION 4
ACTION 5


top
ACTION 5
ACTION 4
ACTION 3
ACTION 2
ACTION 1


ACTION 1
ACTION 2
ACTION 3
ACTION 4
ACTION 5
UNDO
top
ACTION 5
ACTION 4
ACTION 3
ACTION 2
ACTION 1


ACTION 1
ACTION 2
ACTION 3
ACTION 4
ACTION 5
top
ACTION 4
ACTION 3
ACTION 2
ACTION 1
POP(


ACTION 1
ACTION 2
ACTION 3
ACTION 4
top
ACTION 3
ACTION 2
ACTION 1
POP(



Summary:

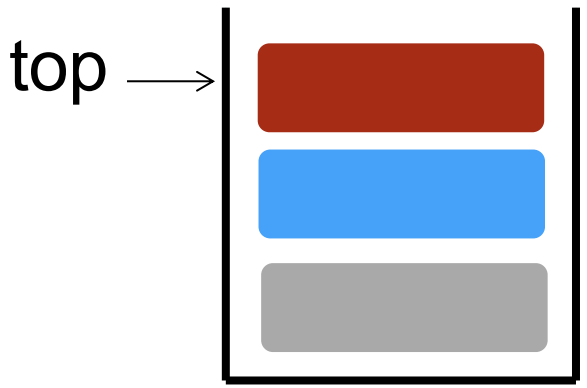
- Stack definition
- Stack operations
- Stack applications
- Next, we will look at how to implement a stack data structure.

Stacks implementation

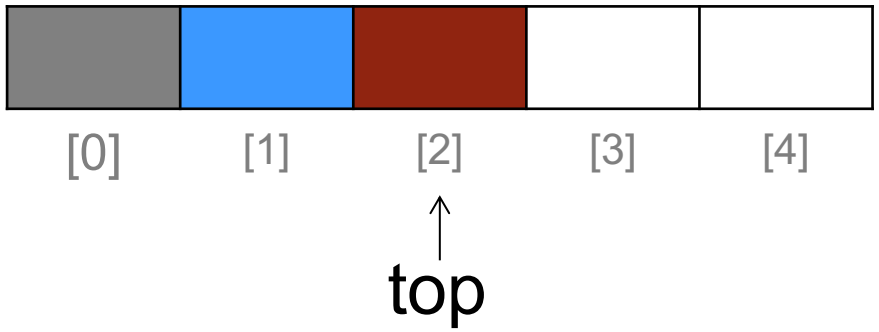
Most modern programming languages have libraries where stacks are already implemented

A stack is a linear data structure can be implemented using another linear data structure

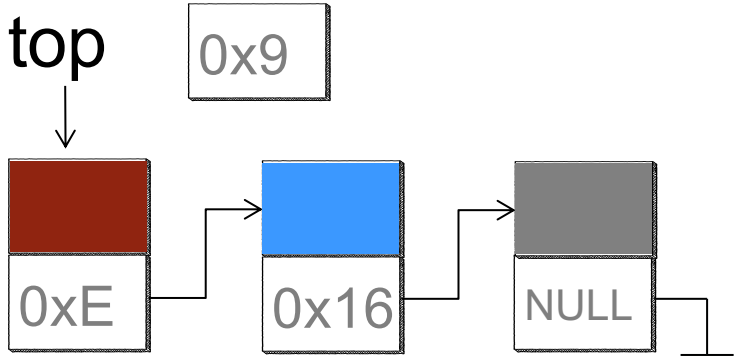
STACK

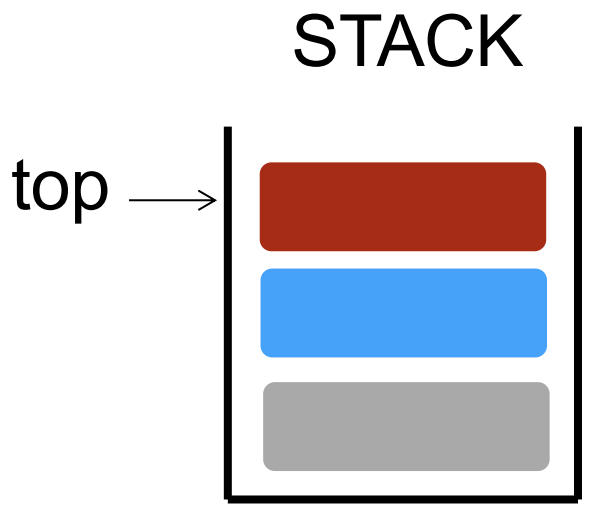


Array implementation

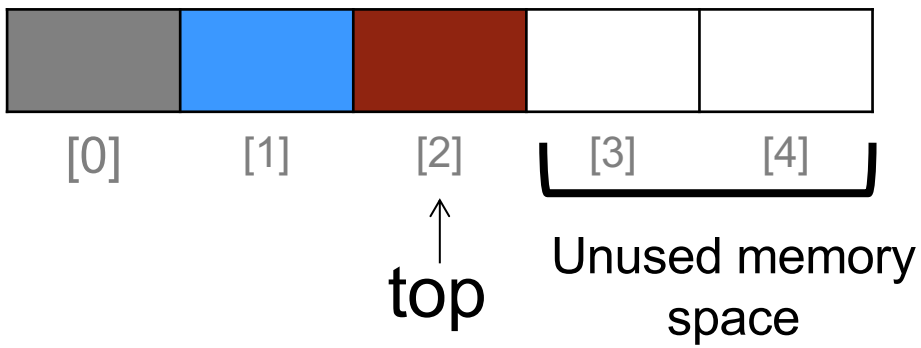


Linked list implementation

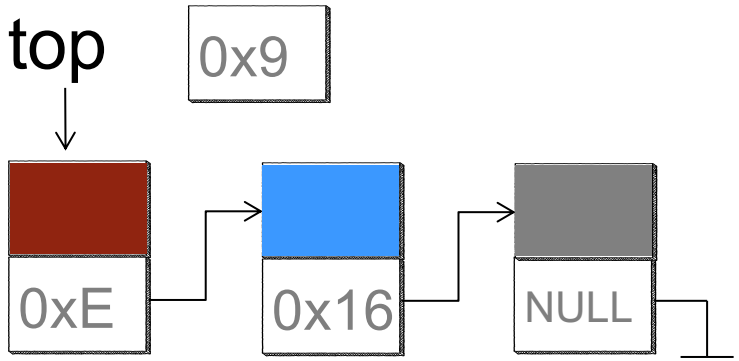




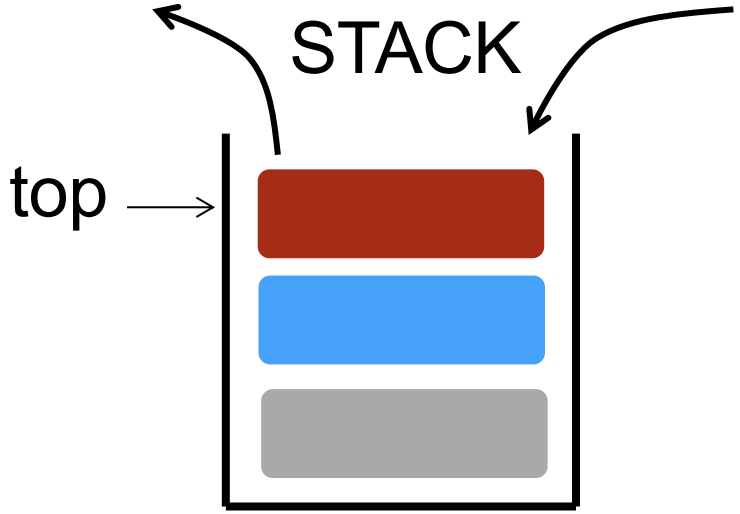
Array implementation



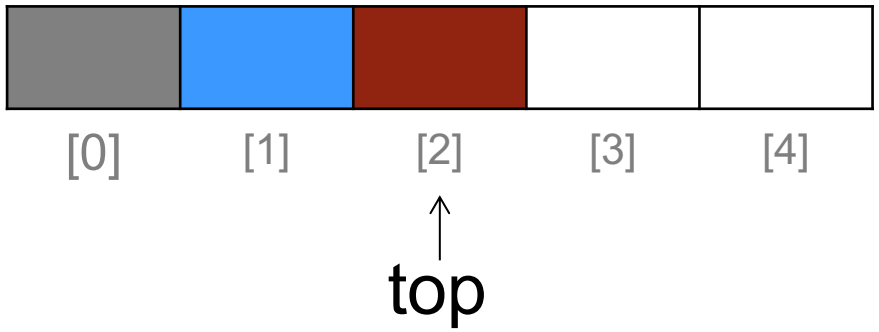
Linked list implementation



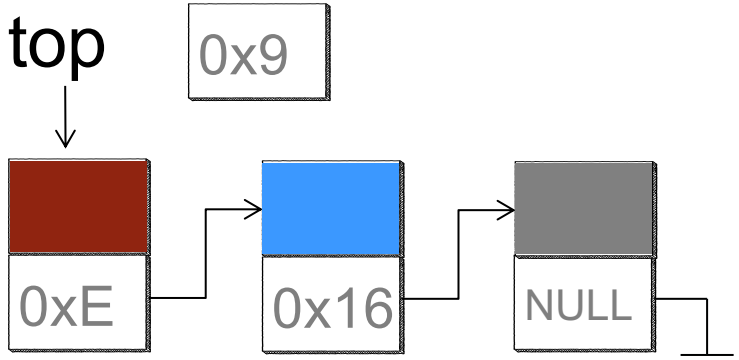
Drawbacks: unused memory or insufficient memory space

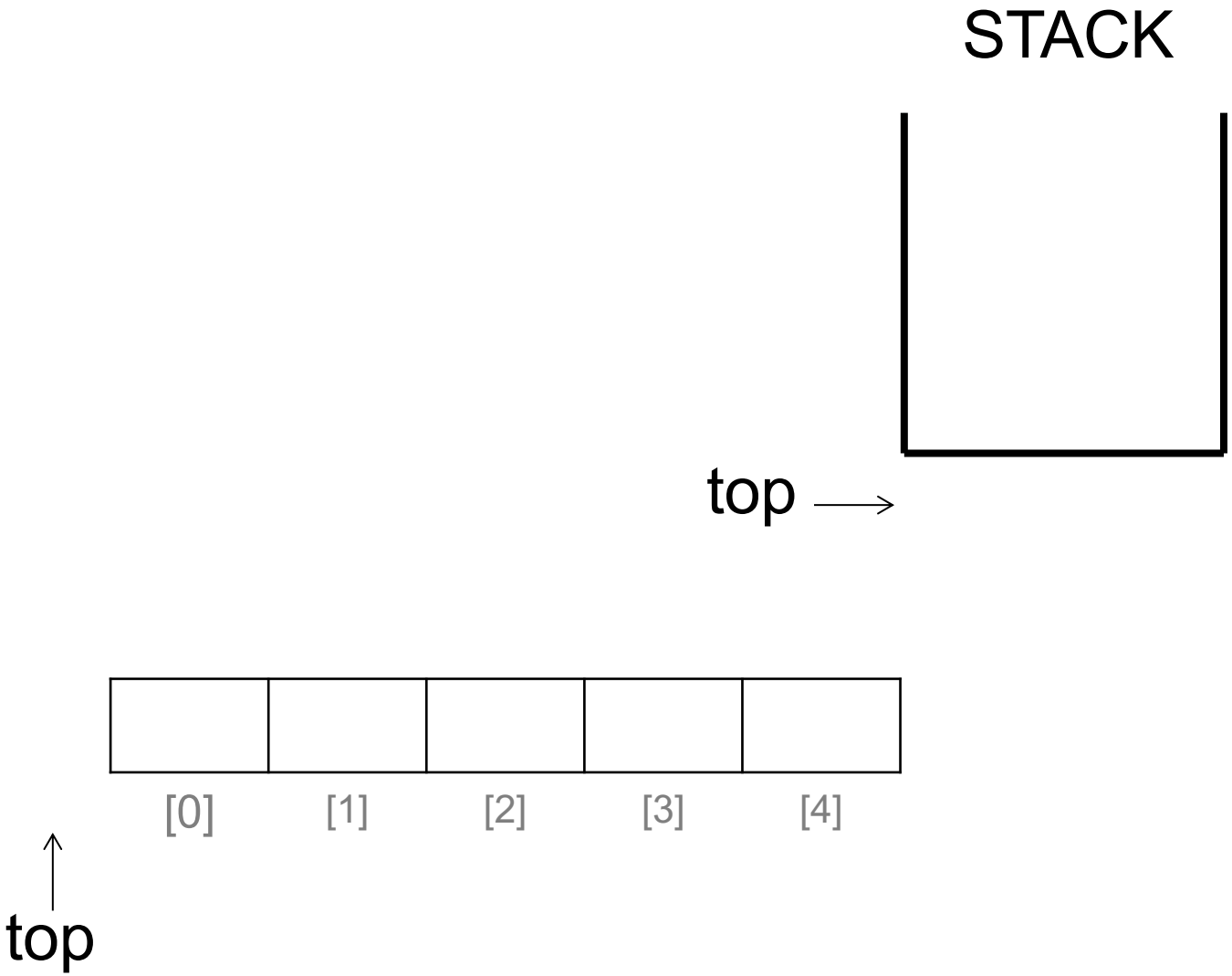


Array implementation



Linked list implementation

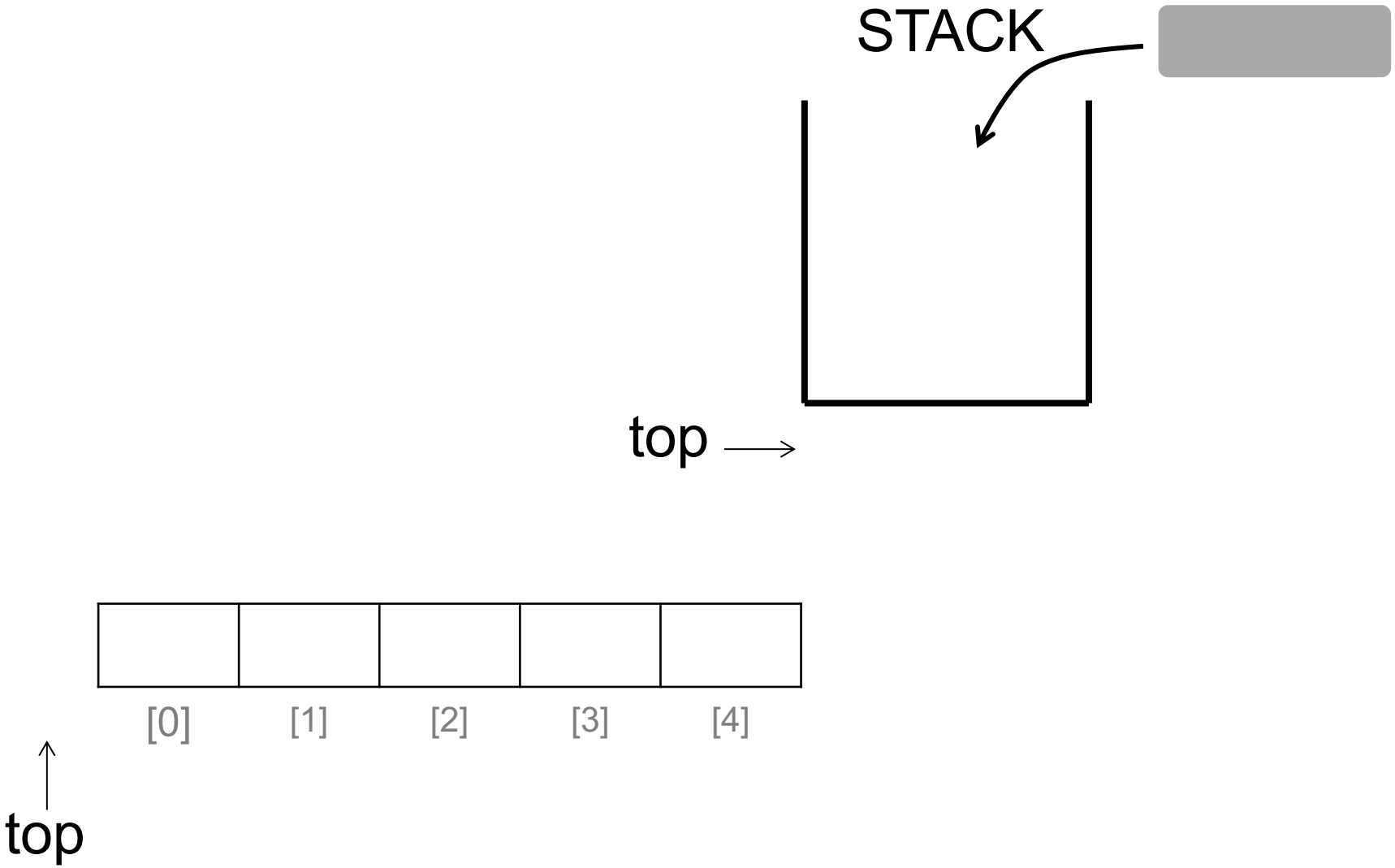




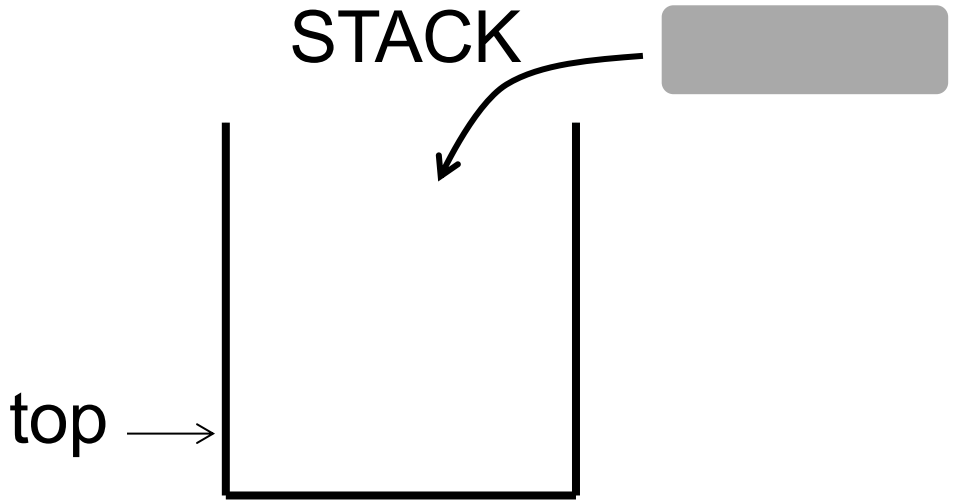
Array implementation

1st Creation of the array

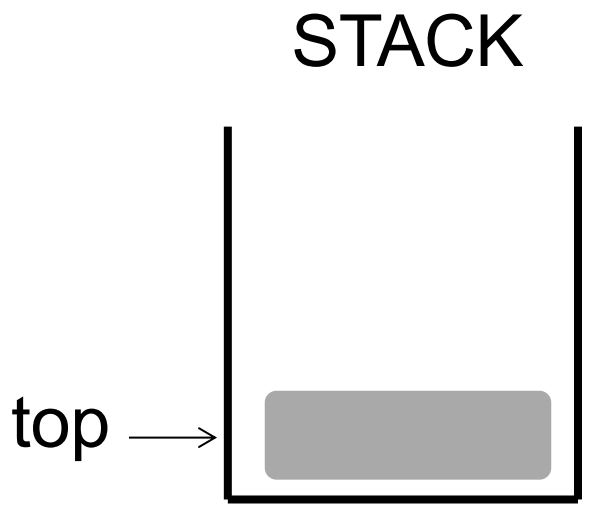
The variable top is initialised to -1 to signal that the stack is emplty



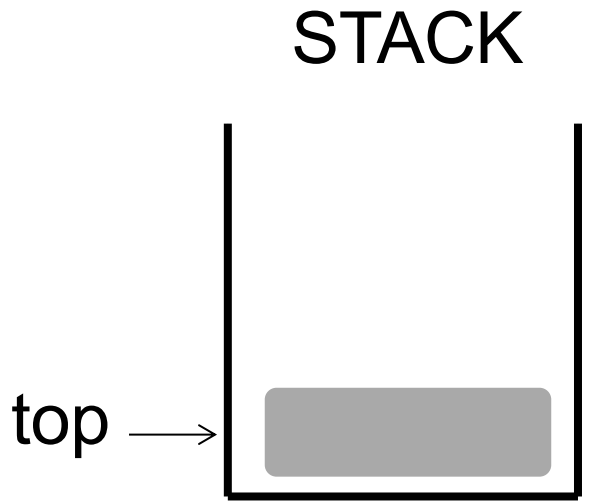
To push an element is into to the stack, we must perform two actions




```
function PUSH(x)
    top=top+1
    A[top]=x
```




```
function PUSH(x)
    top=top+1
    A[top]=x
```

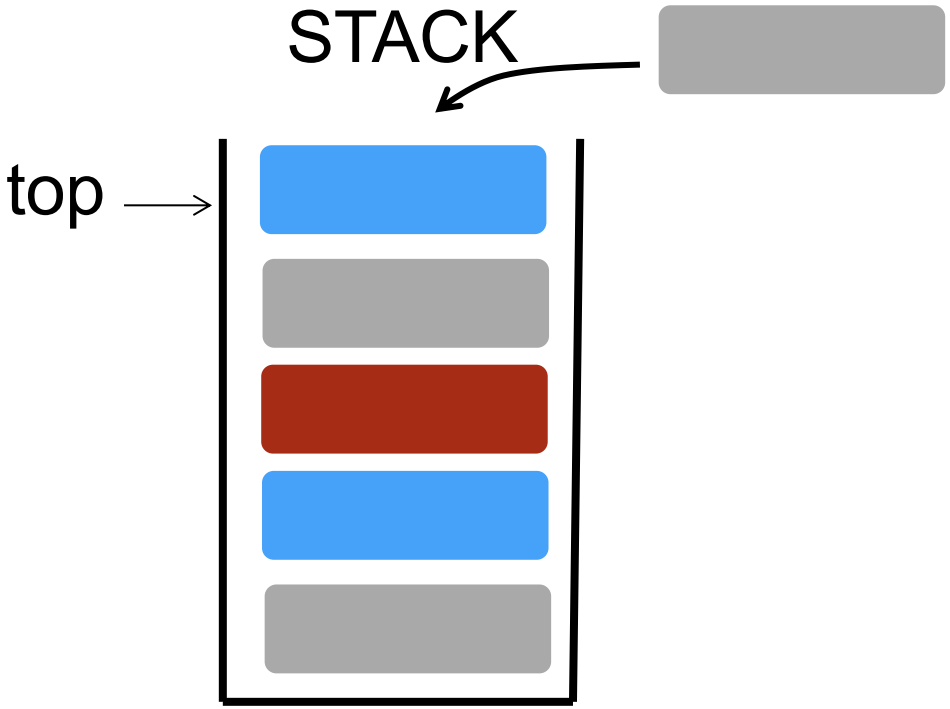


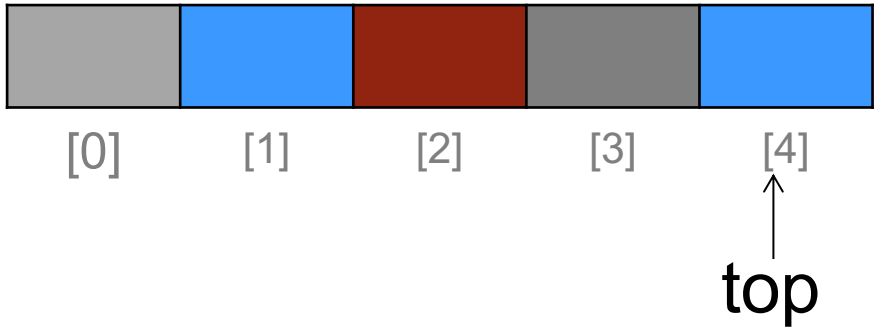

```
function PUSH(x)
    top=top+1
    A[top]=x
```

C_0

C_1

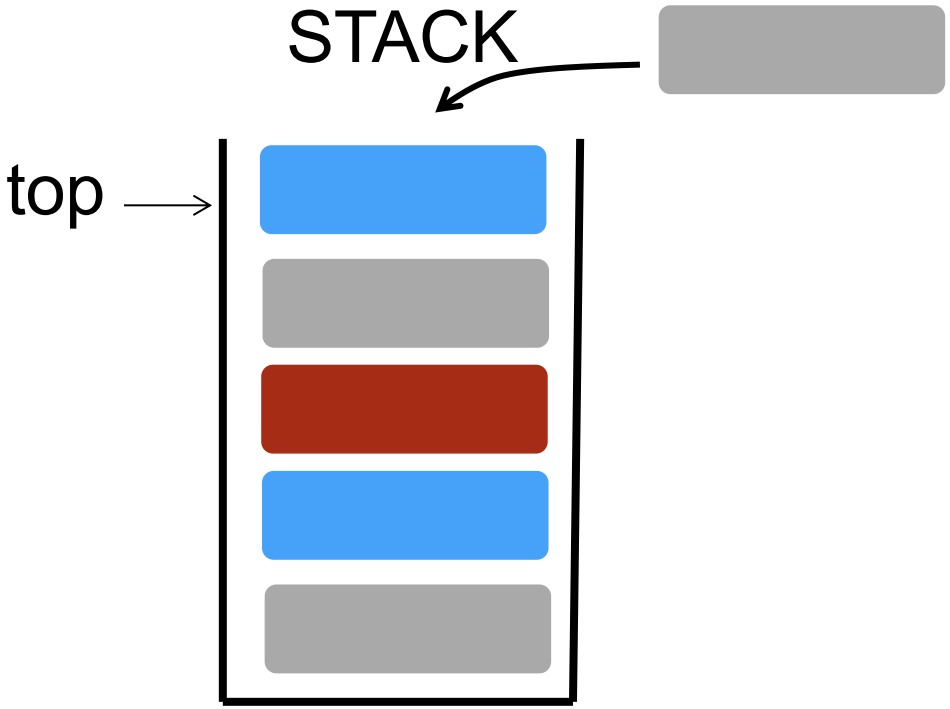
Time complexity: Theta(1)

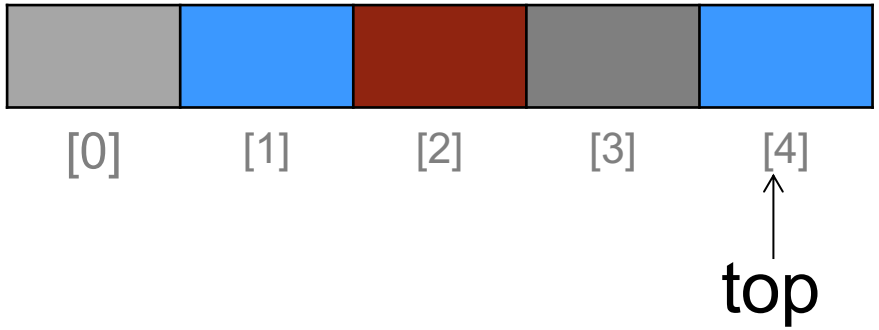





```
function PUSH(x)
    top=top+1
    A[top]=x
```

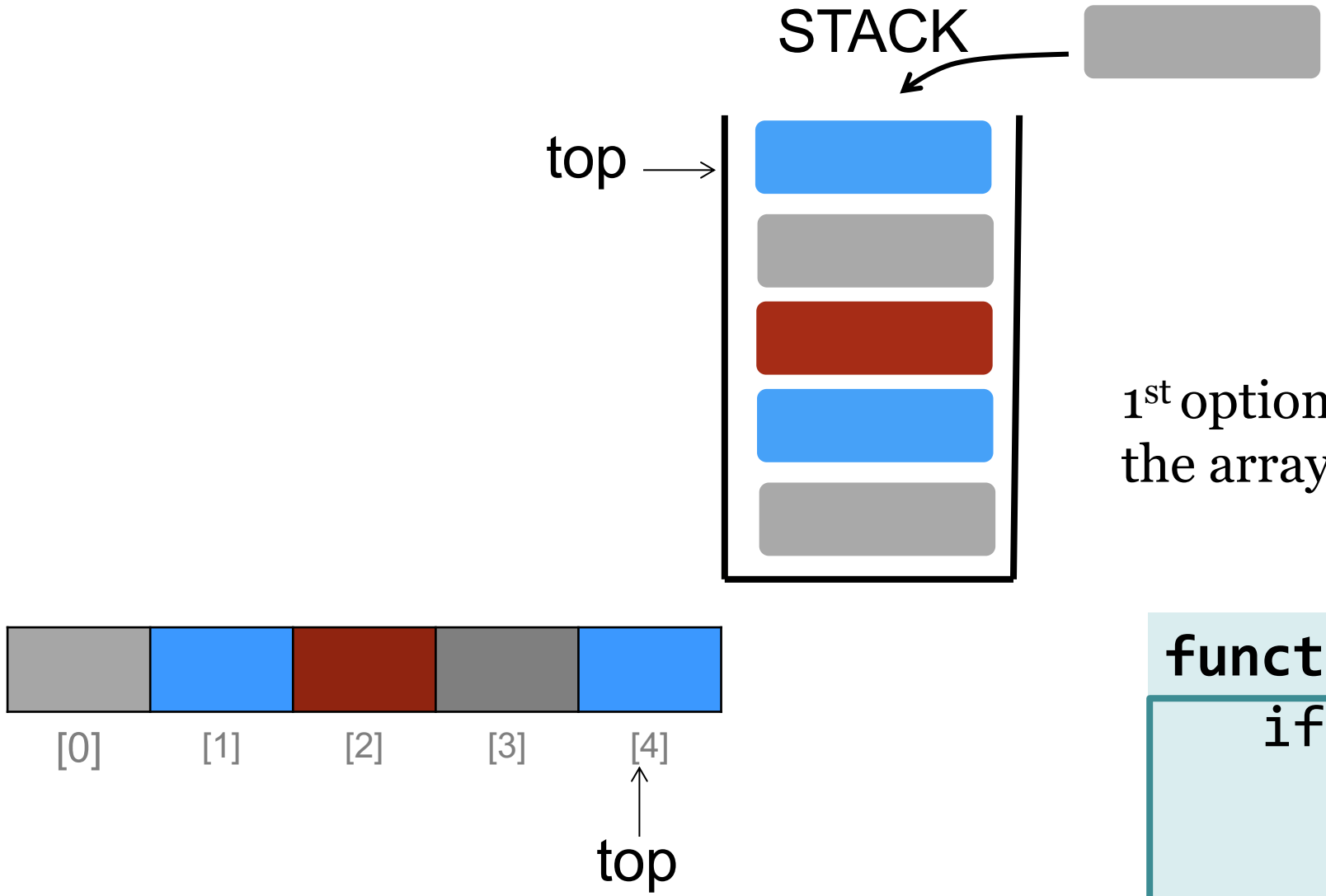
What if we run out of space in the array?





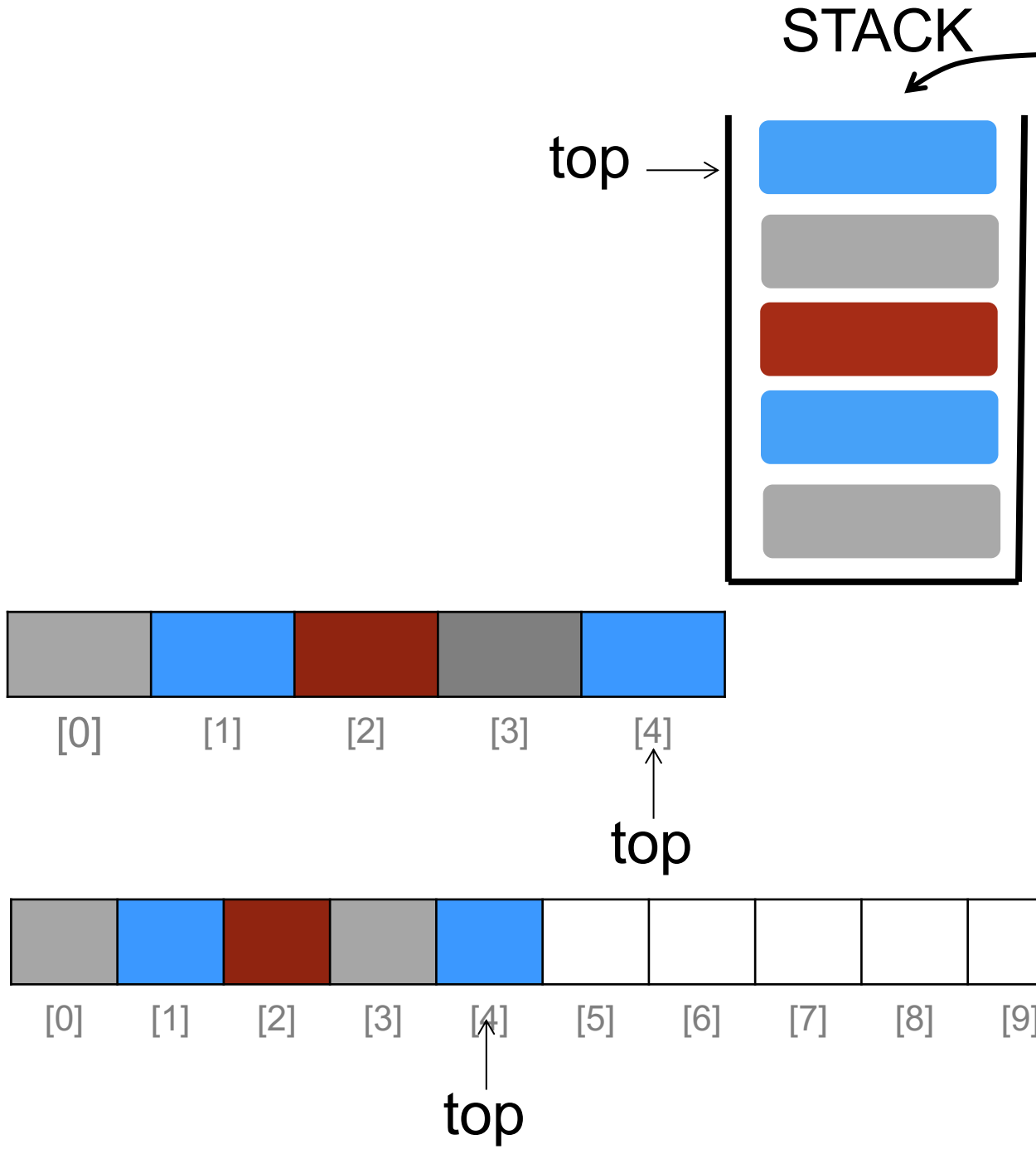

```
function PUSH(x)
    top=top+1
    A[top]=x
```

This instruction will return an error! There are two ways of solving this problem !



1st option: stop adding any element i
the array is full.

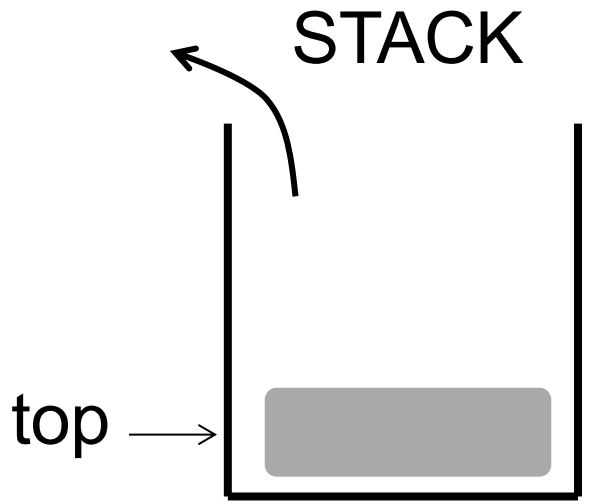
```
function PUSH(x)
    if(top==size(A)-1)
        print("stack overflow")
        return
    top=top+1
    A[top]=x
```



2nd option: dynamic array

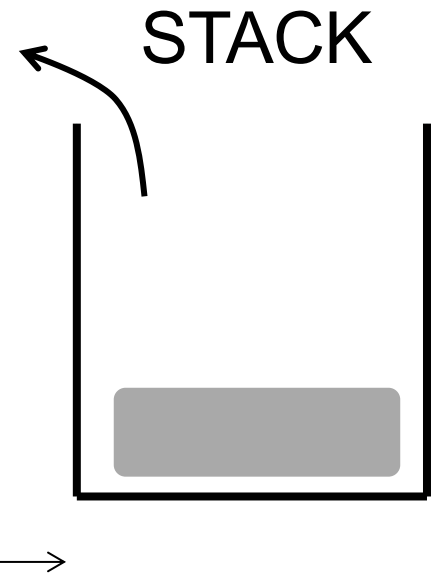
```
function PUSH(x)
    if(top==size(A)-1)
        extend&copy
    top=top+1
    A[top]=x
```

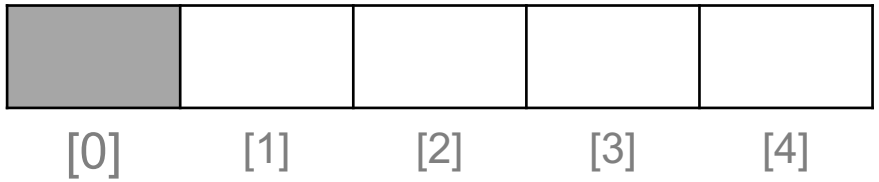
Complexity is Theta(n)




```
function POP( )
```

We assume that POP() doesn't return the element removed.

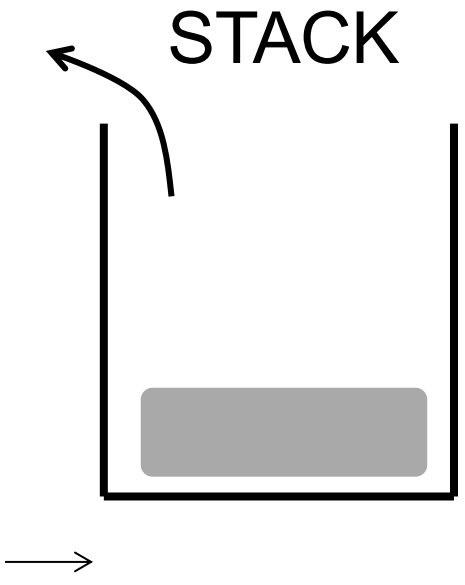


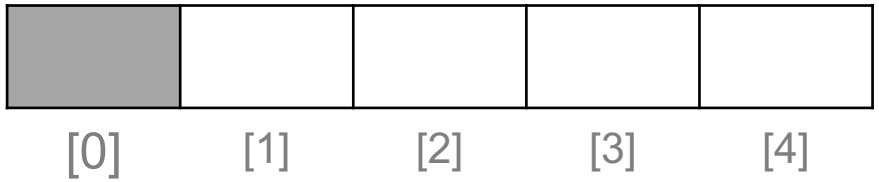


top

```
function POP( )
    top=top-1
```

To delete an element of the top we only need to decrease the value of the variable top by

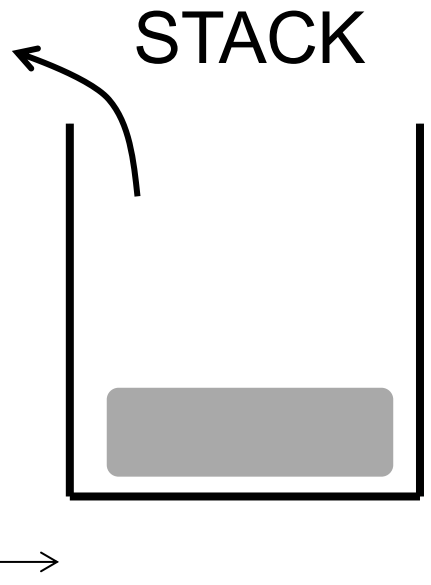




top

```
function POP( )
  if (top==-1)
    print("empty stack")
    return
  top=top-1
```

Checks if the stack is empty

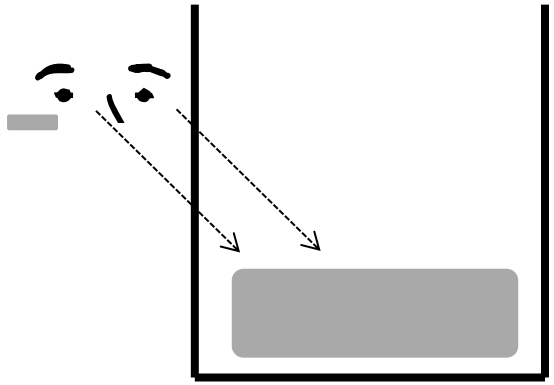



```
function POP( )
    if (top==-1)
      print("empty stack")
      return
    top=top-1
```

C_0 C_1 C_2 C_3

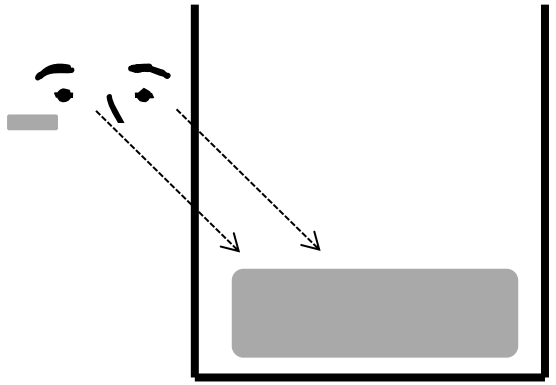
Time complexity: Theta(1)

STACK

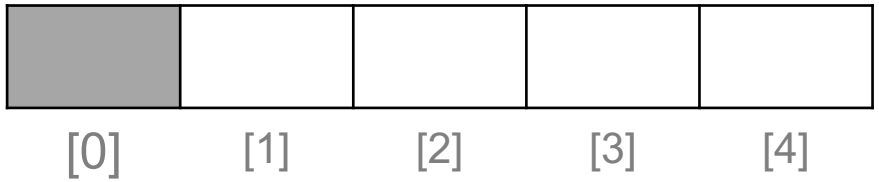



```
function PEEK( )
  if (top==-1)
    print(“empty stack”)
    return
  return A[top]
```

STACK



top

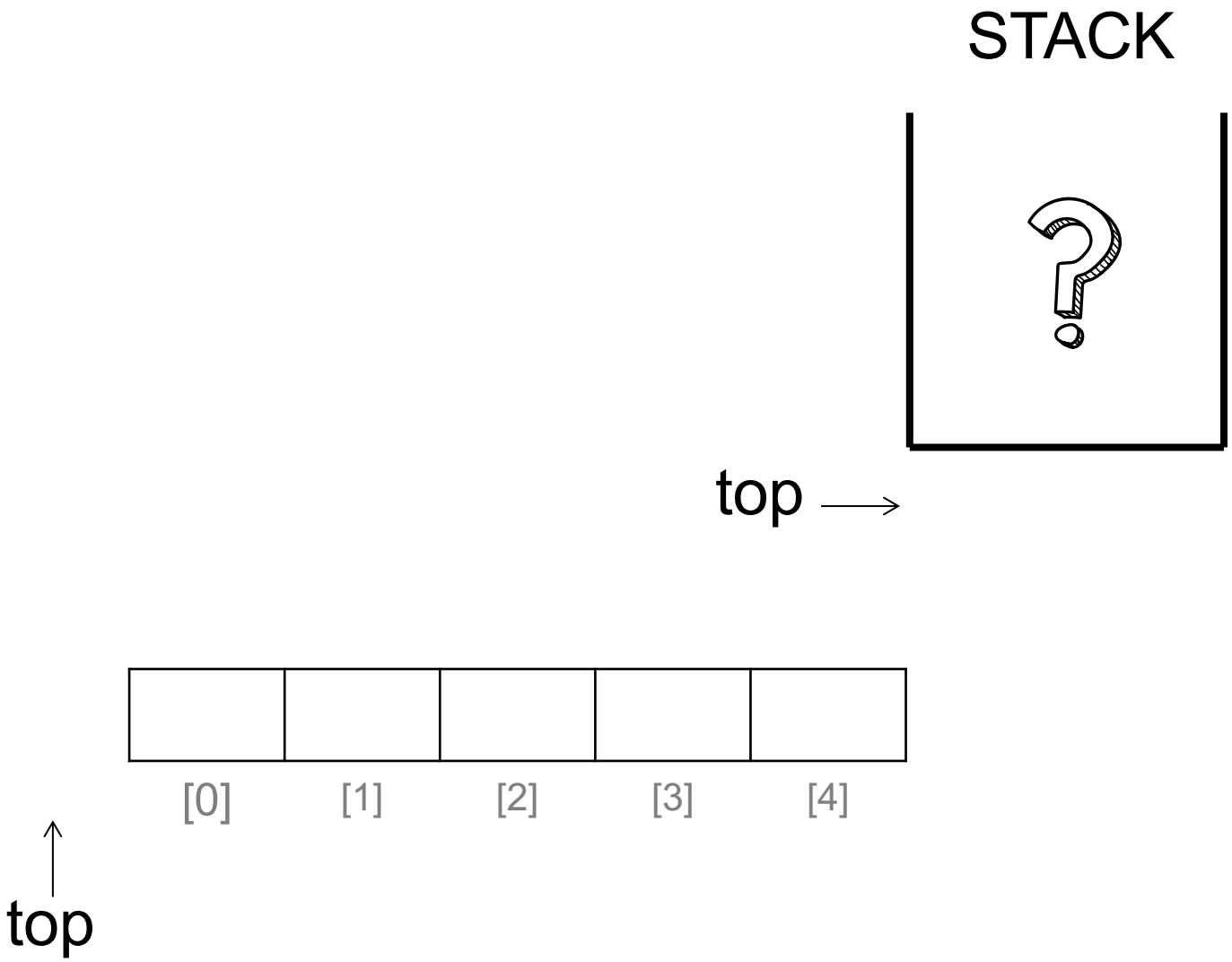


top

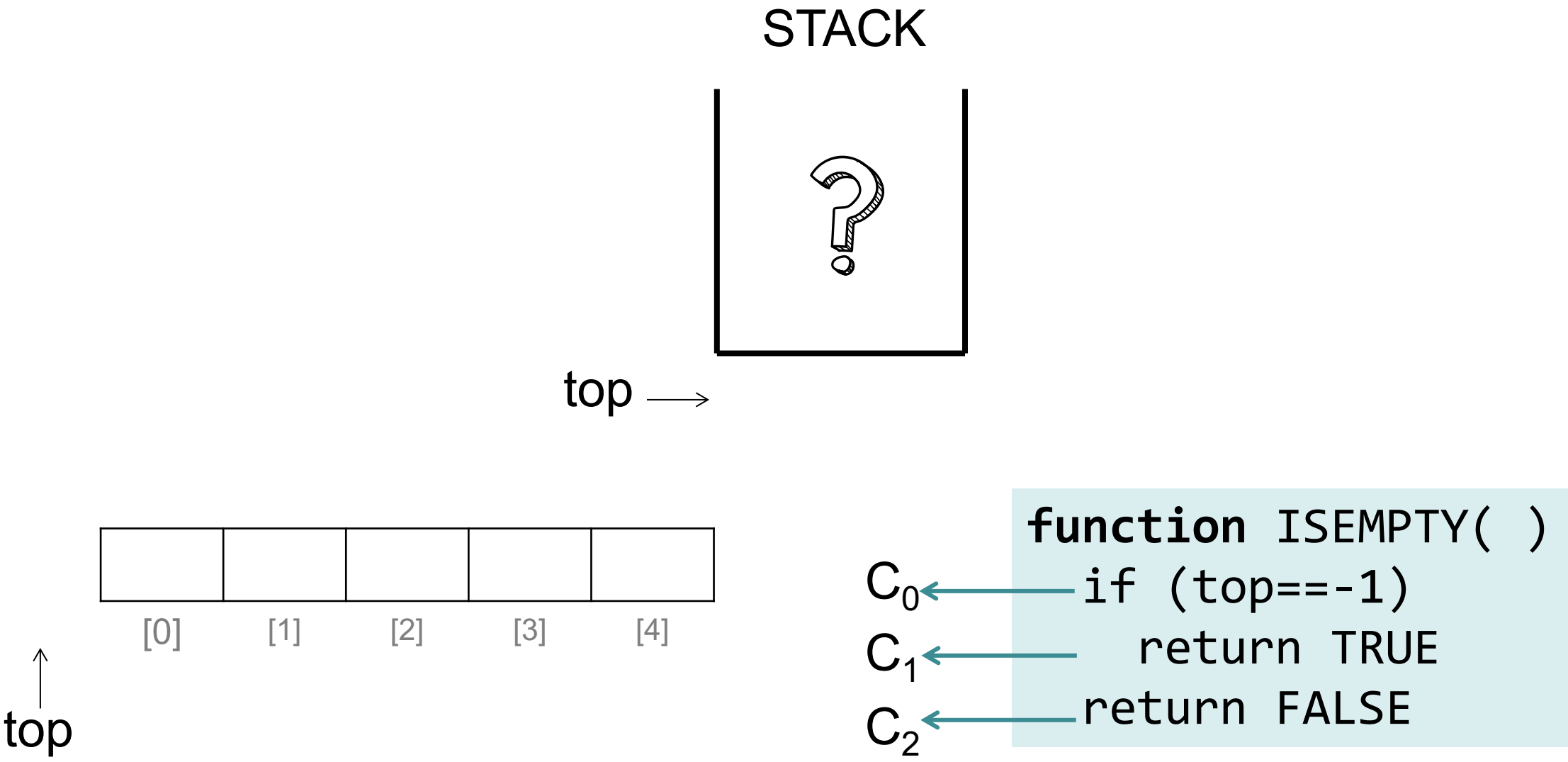
```
function PEEK( )
    if (top==-1)
      print("empty stack")
      return
    return A[top]
```

C_0 C_1 C_2 C_3

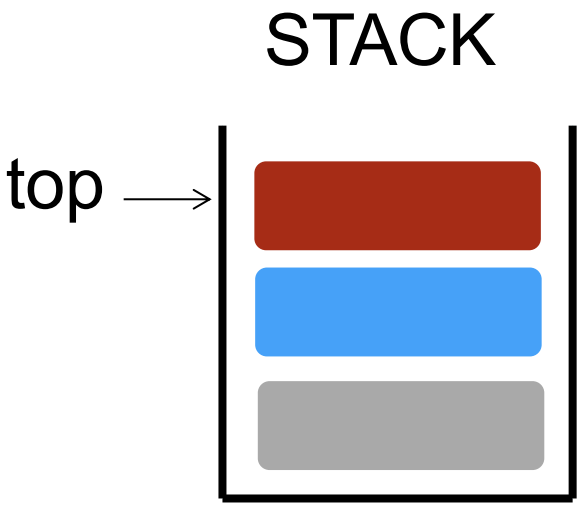
Time complexity: Theta(1)

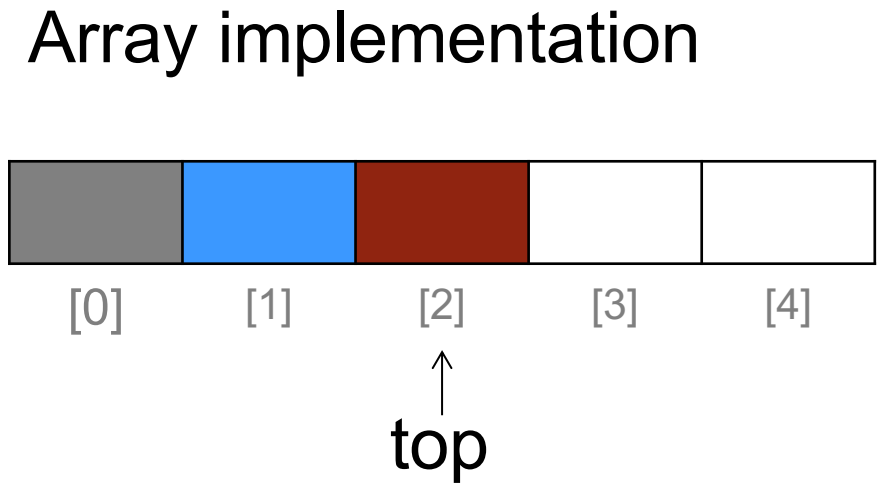



```
function ISEMPTY( )
  if (top==-1)
    return TRUE
  return FALSE
```

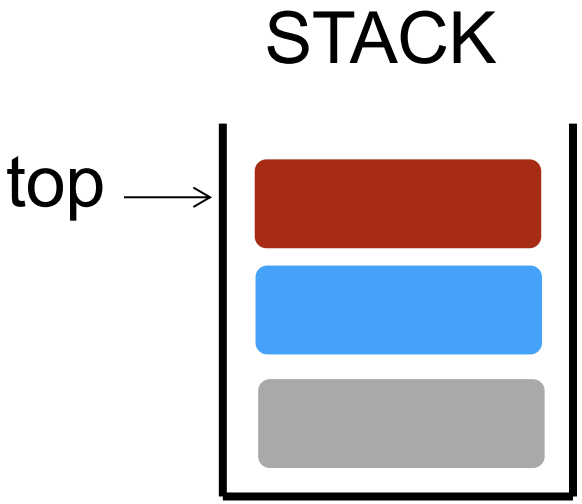


Time complexity: Theta(1)

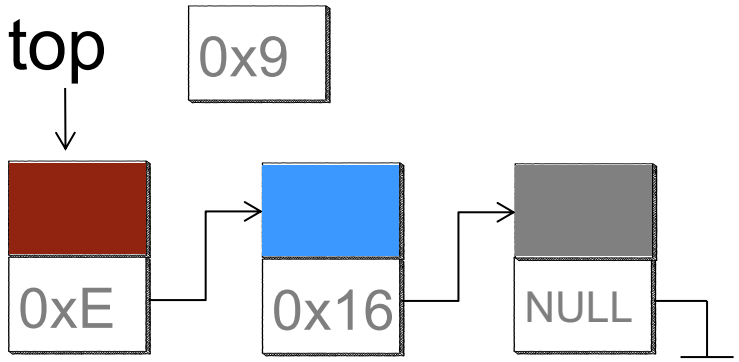


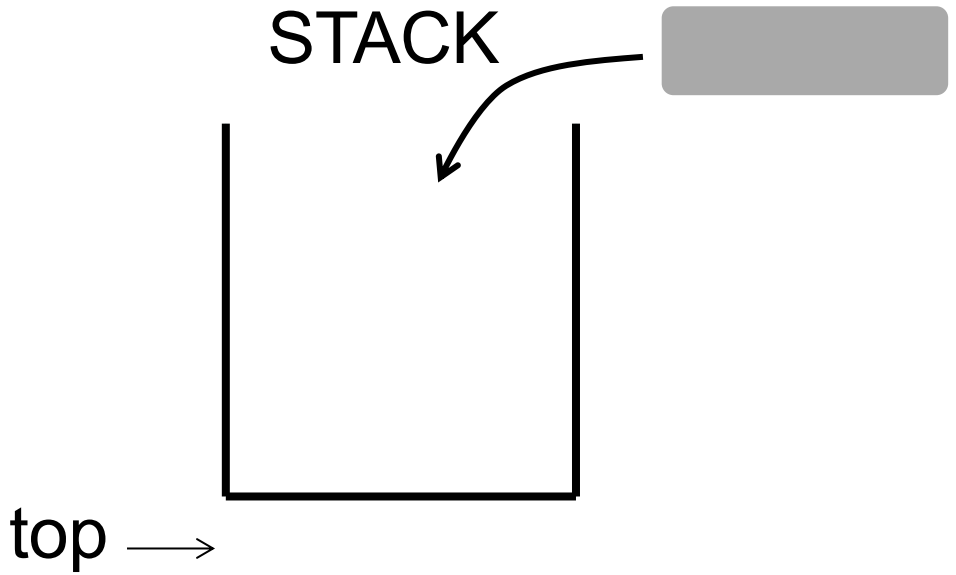


	TIME COMPLEXITY	
OPERATION	BEST CASE	WORST CASE
PUSH	Θ(1)	Θ(N)
POP	Θ(1)	Θ(1)
PEEK	Θ(1)	Θ(1)
ISEMPTY	Θ(1)	Θ(1)



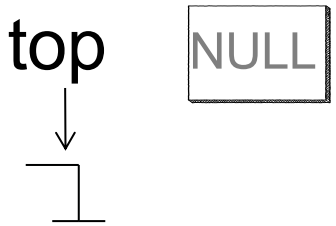
Linked list implementation

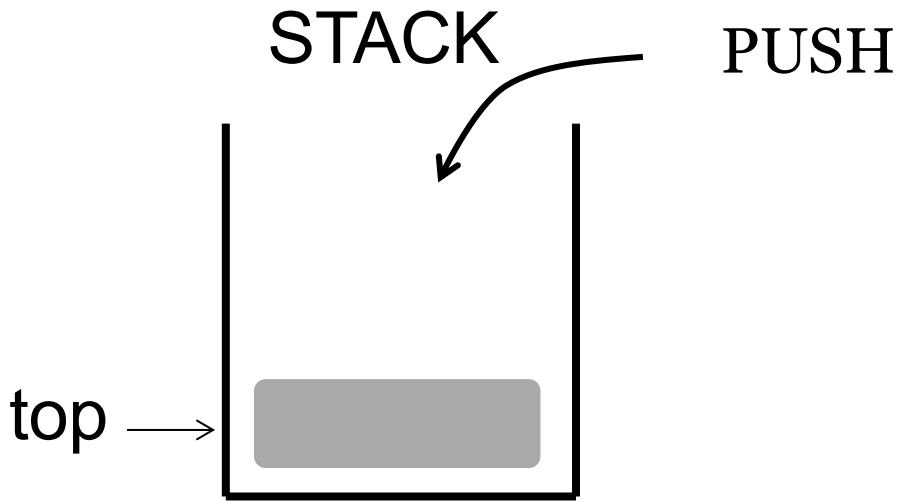




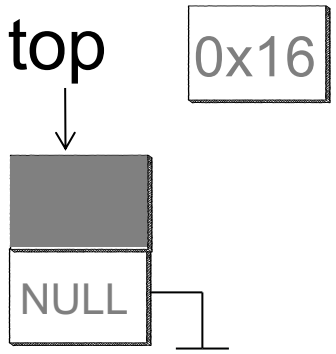
Linked list implementation

We start with an empty stack, top points to NULL

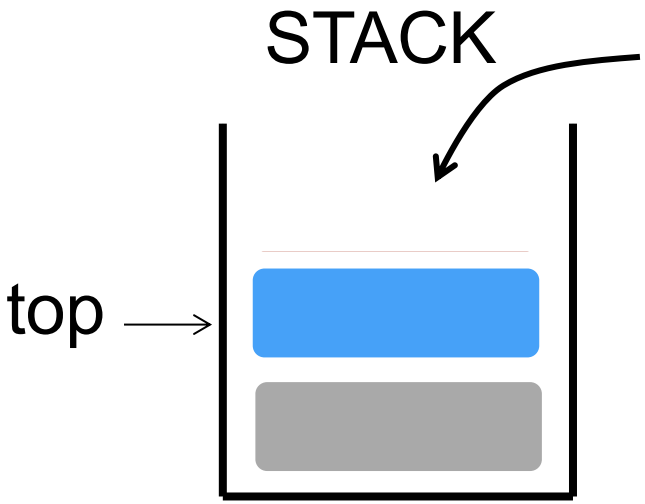




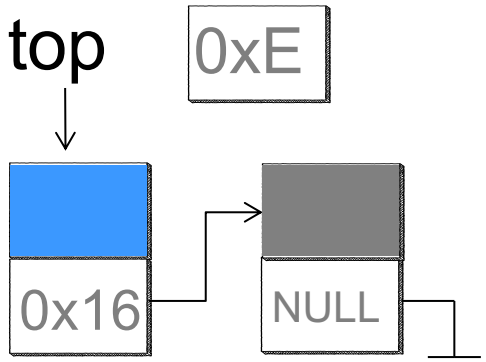
Linked list implementation



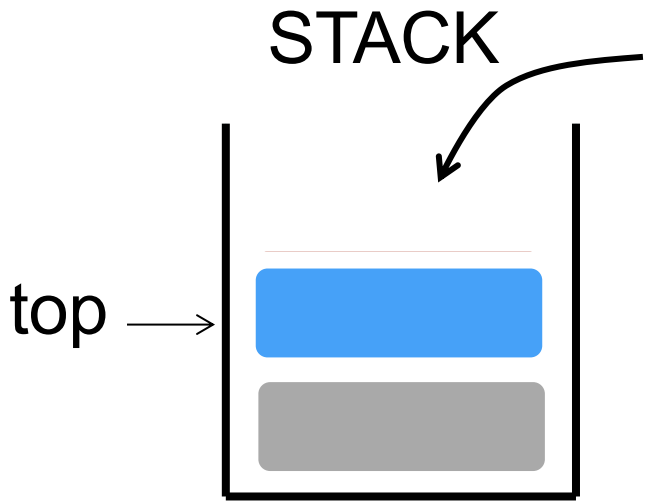
top point to the address of the new element



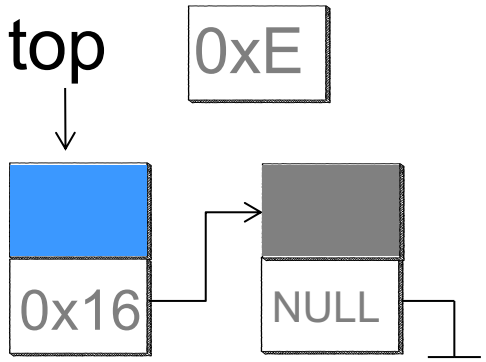
Linked list implementation



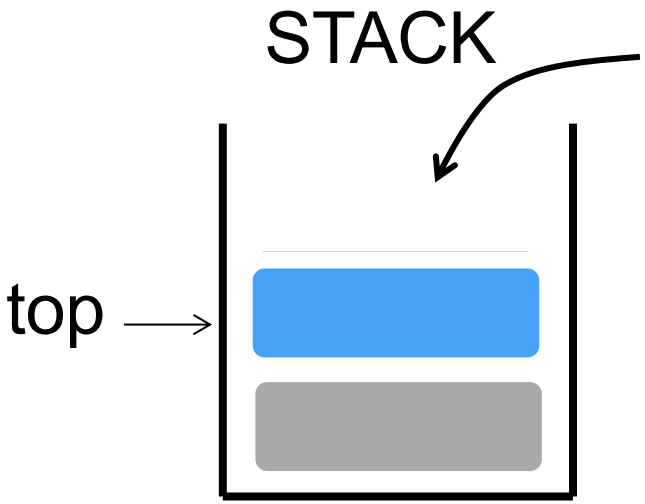
If we execute another push operation we just insert a new element at beginning of the list!



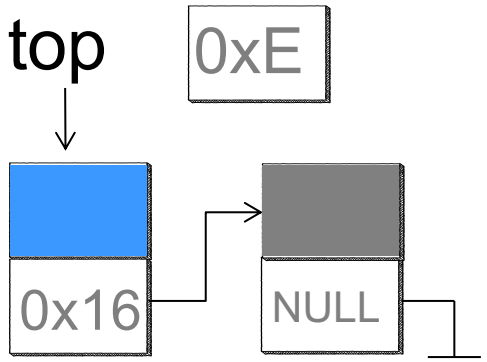
Linked list implementation




```
function PUSH( )
   newNode=new Node(data)
   newNode->next=top
   top=newNode
```



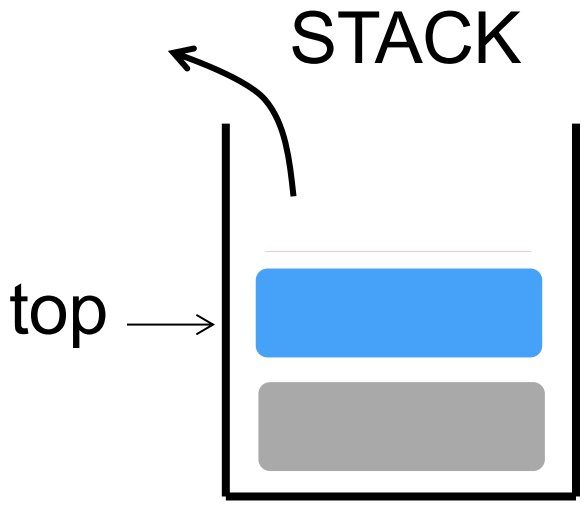
Linked list implementation

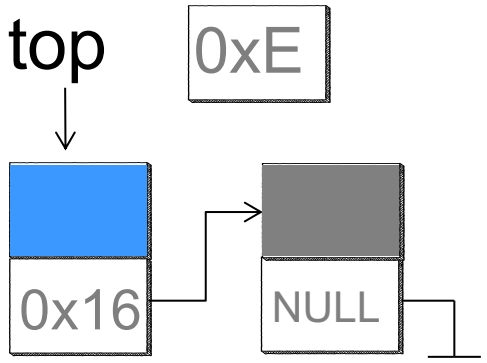



```
function PUSH( )
    newNode=new Node(data)
    newNode->next=top
    top=newNode
```

C_0 C_1 C_2

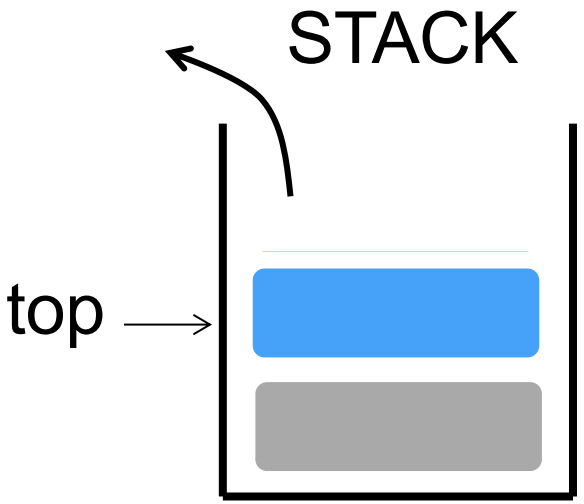
Time complexity: Theta(1)

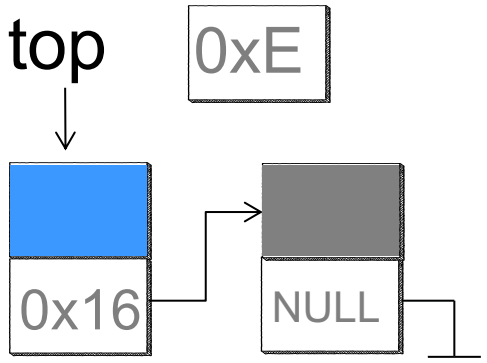




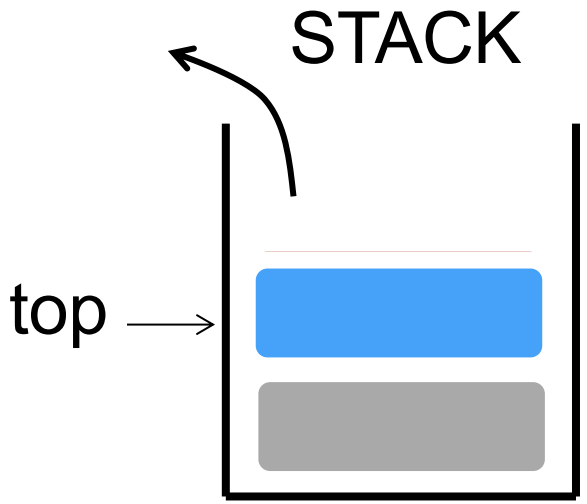

```
function POP( )
```

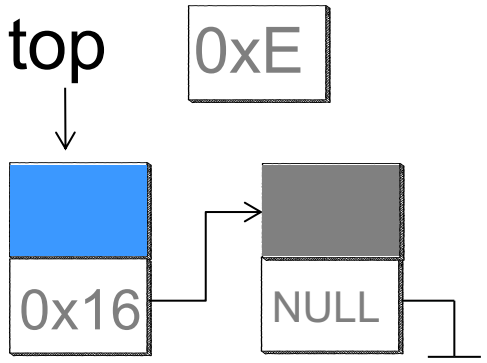
This is the same as removing the 1st element of the list!



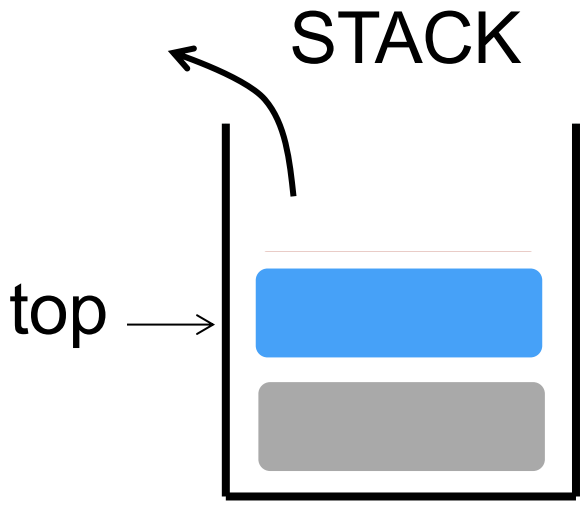


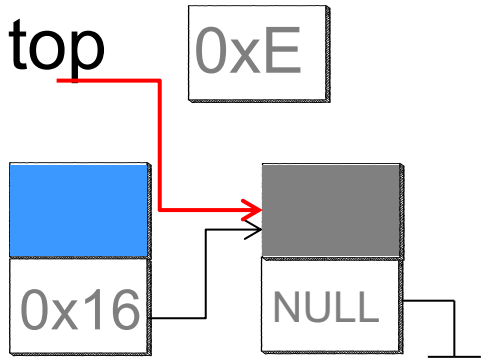

```
function POP( )
  if(top==NULL)
    print(“empty list”)
    return
  top=top->next
```



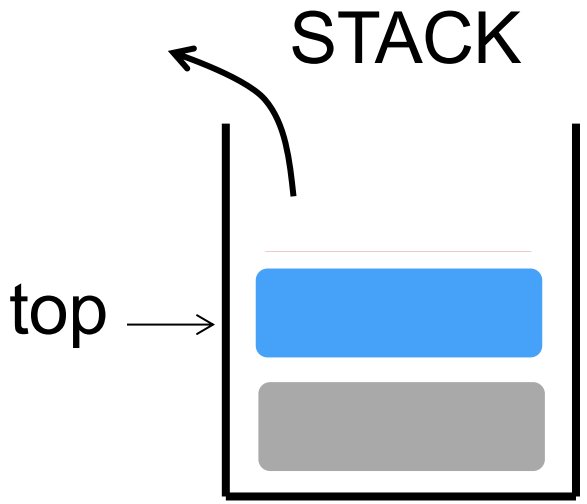


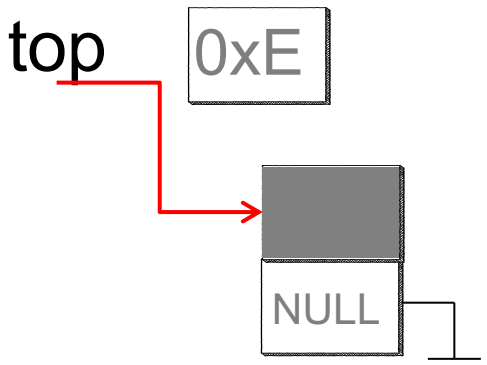

```
function POP( )
   if(top==NULL)
      print(“empty list”)
      return
   top=top->next
```



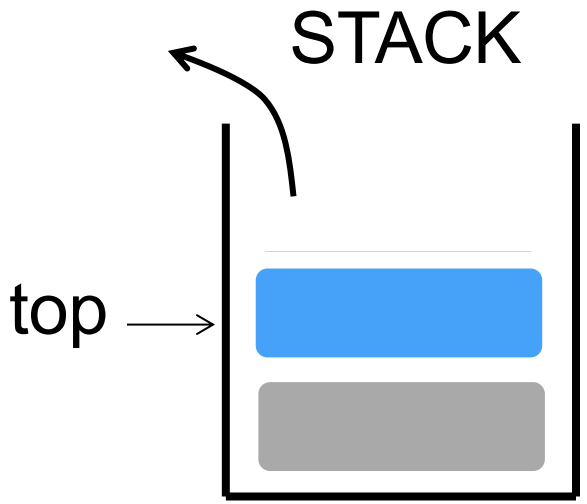


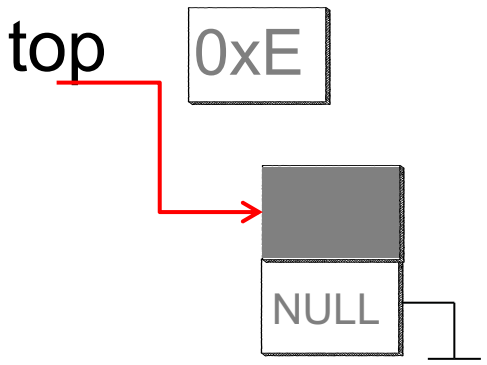

```
function POP( )
  if(top==NULL)
    print(“empty list”)
    return
  top=top->next
```






```
function POP( )
  if(top==NULL)
    print(“empty list”)
    return
  top=top->next
```



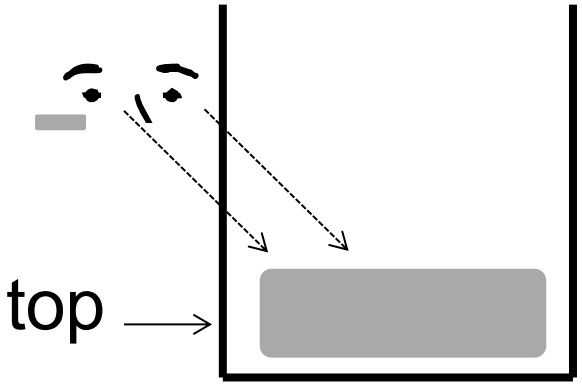


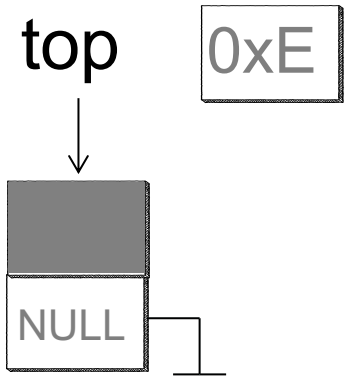

```
function POP( )
    if(top==NULL)
      print(“empty list”)
      return
    top=top->next
```

C_0 C_1 C_2 C_3

Time complexity: Theta(1)

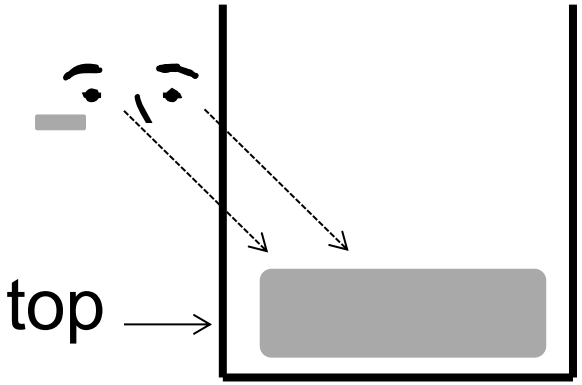
STACK

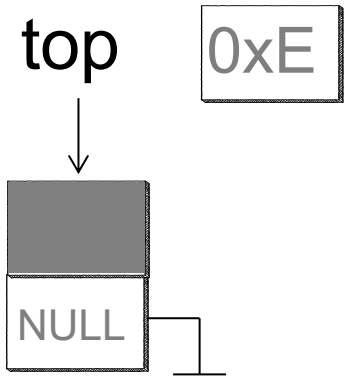





```
function PEEK( )
  if(top==NULL)
    print(“empty list”)
    return
  print(top->data)
```

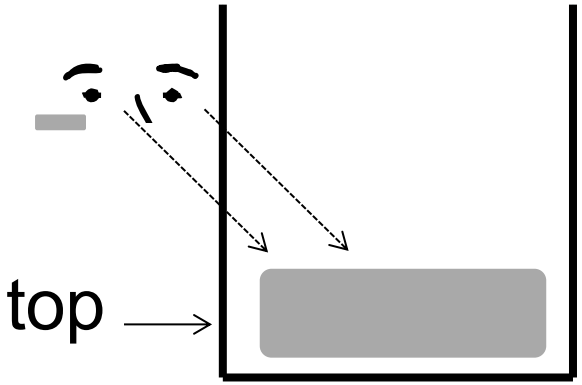
STACK

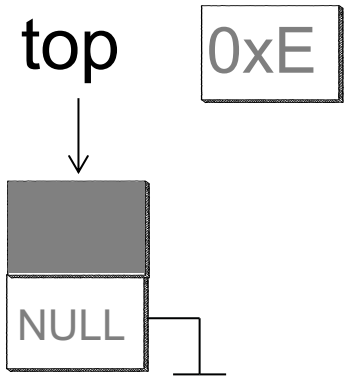




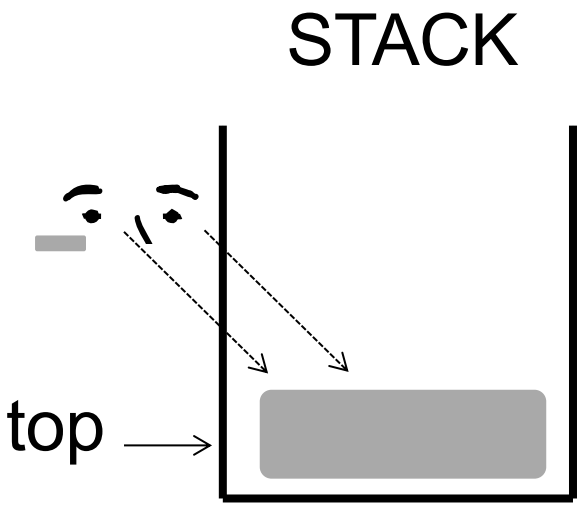

```
function PEEK( )
   if(top==NULL)
      print(“empty list”)
      return
   print(top->data)
```

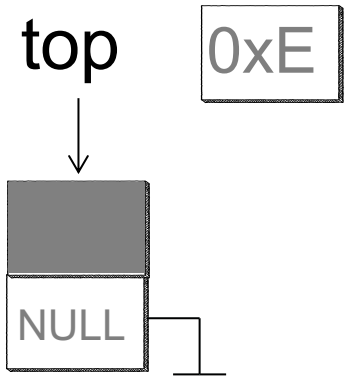
STACK






```
function PEEK( )
  if(top==NULL)
    print(“empty list”)
    return
  print(top->data)
```

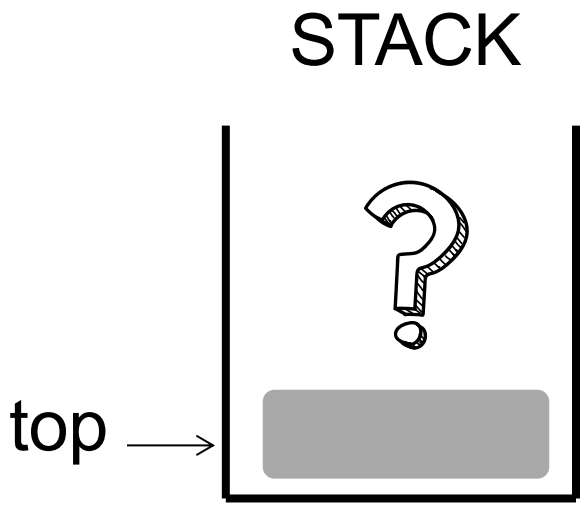


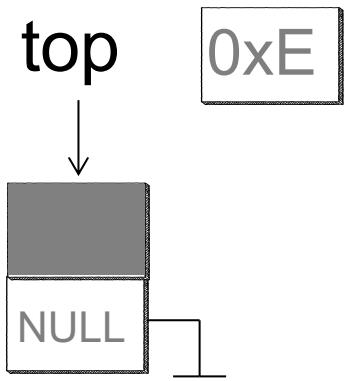


C_0 ← `if(top==NULL)`
C_1 ← `  print("empty list")`
C_2 ← `  return`
C_3 ← `print(top->data)`

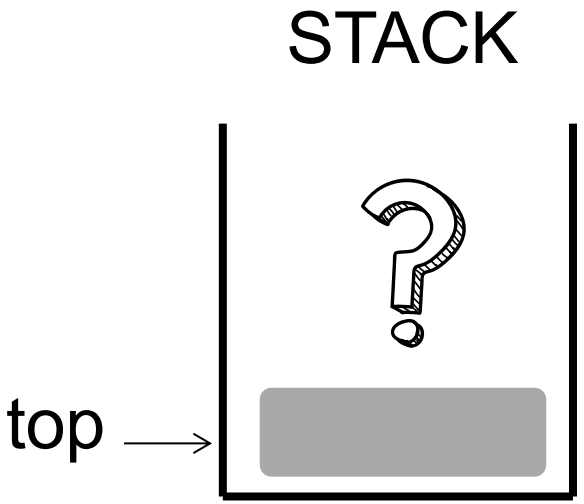
```
function PEEK( )
    if(top==NULL)
      print("empty list")
      return
    print(top->data)
```

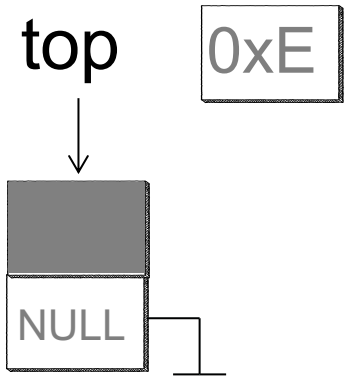
Time complexity: Theta(1)






```
function ISEMPTY( )
  if(top==NULL)
    return TRUE
  return FALSE
```



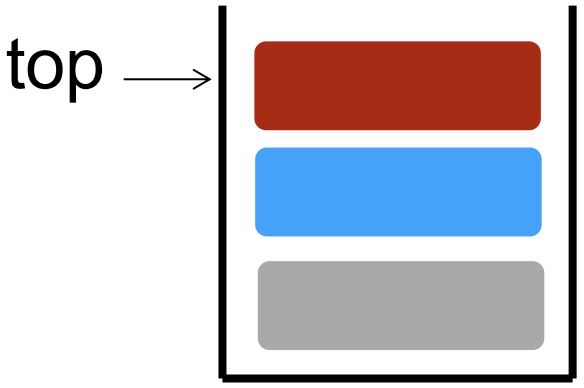



```
function ISEMPTY( )
    if(top==NULL)
      return TRUE
    return FALSE
```

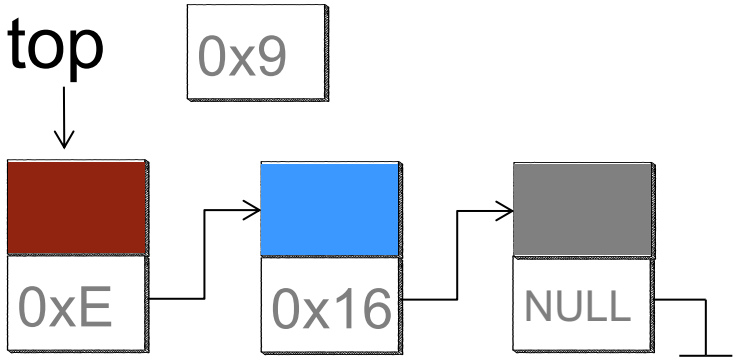
C_0 C_1 C_2

Time complexity: Theta(1)

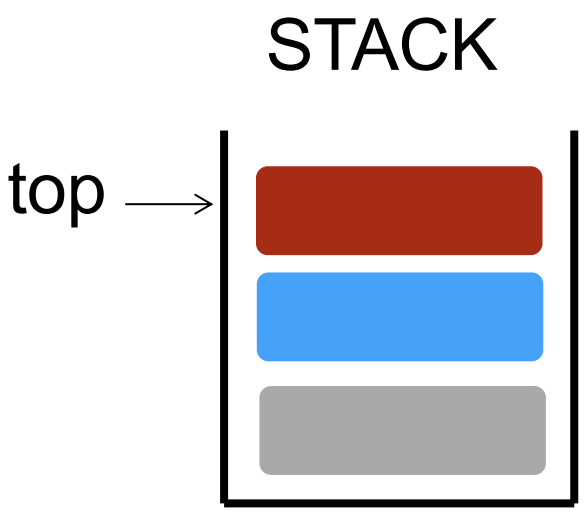
STACK



Linked list implementation



OPERATION	TIME COMPLEXITY
PUSH	Θ(1)
POP	Θ(1)
PEEK	Θ(1)
ISEMPTY	Θ(1)



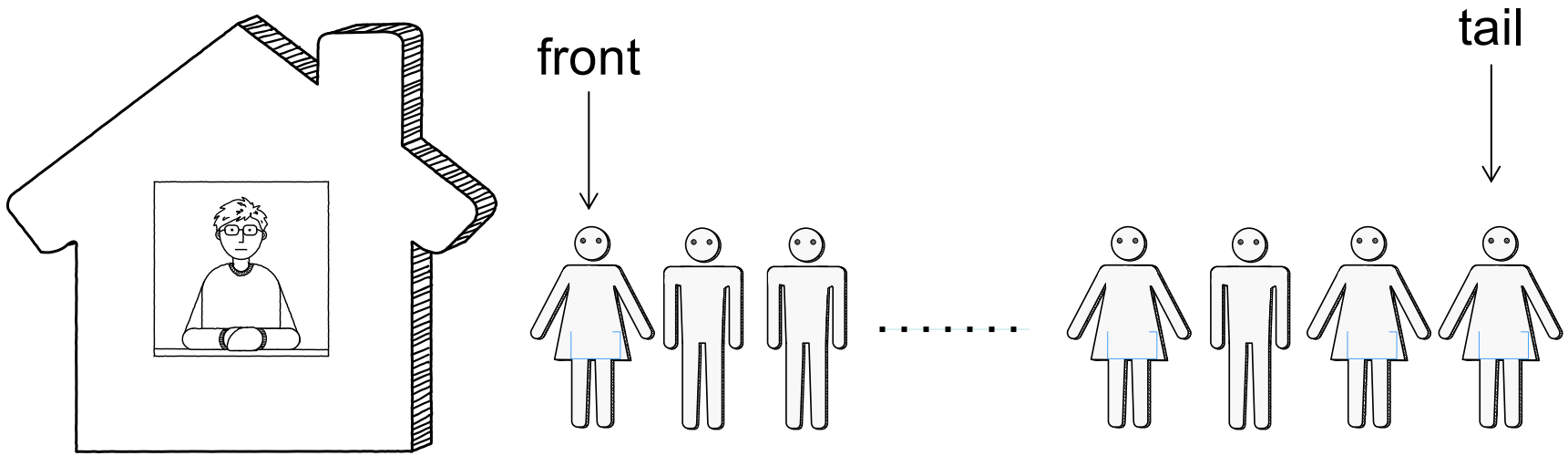
Summary

- Stack is linear data structure
- Stack operations
  - PUSH()
  - POP()
  - PEEK()
  - ISEMPTY
- Implementation
  - ARRAYS
  - LINKED LISTS
- Time complexity

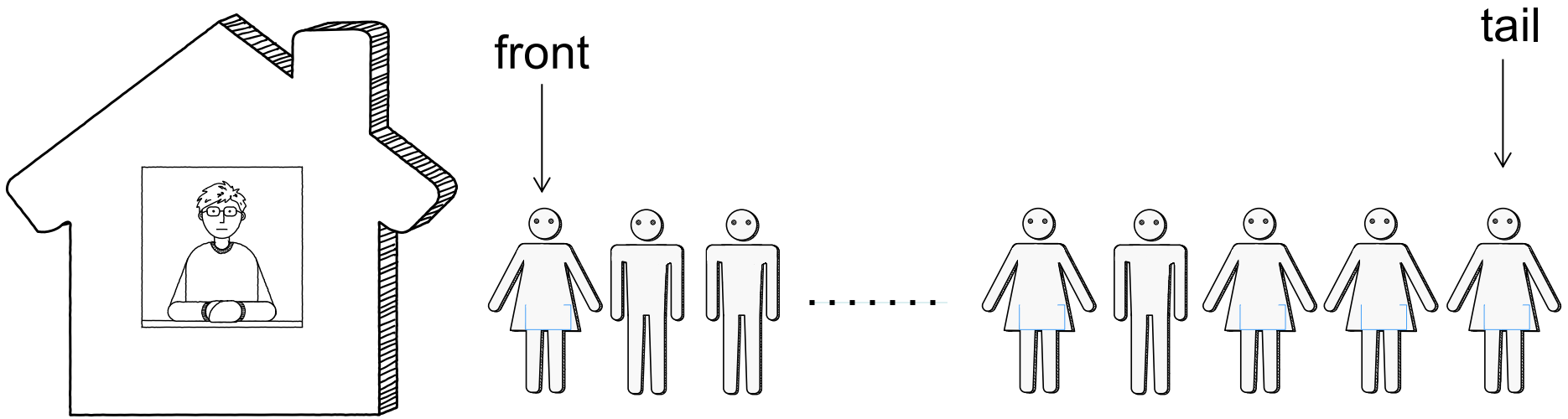
Queues

Queue intro

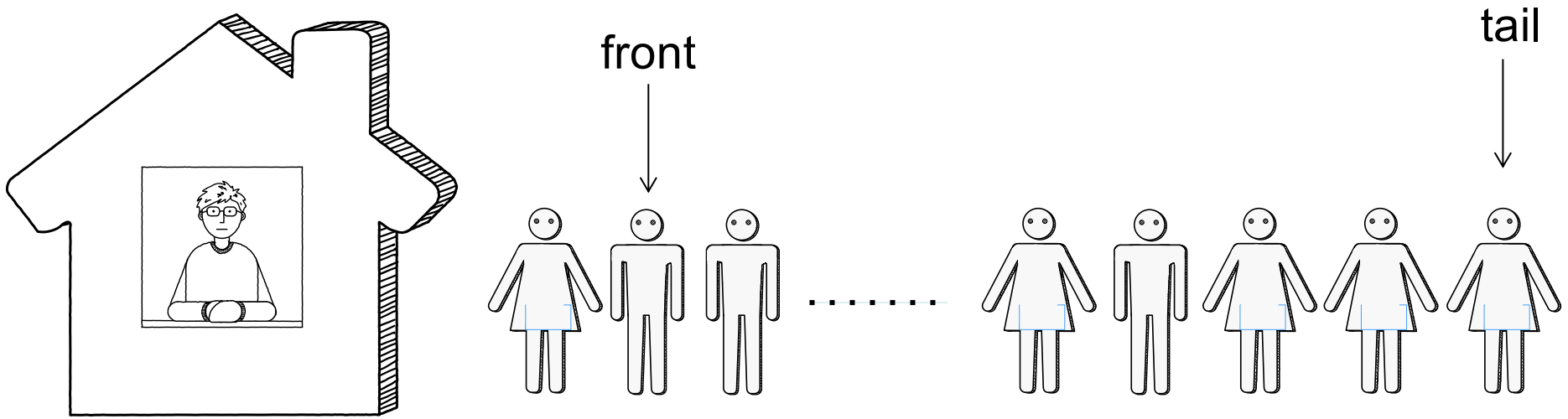
- Queue definition
- Queue operations



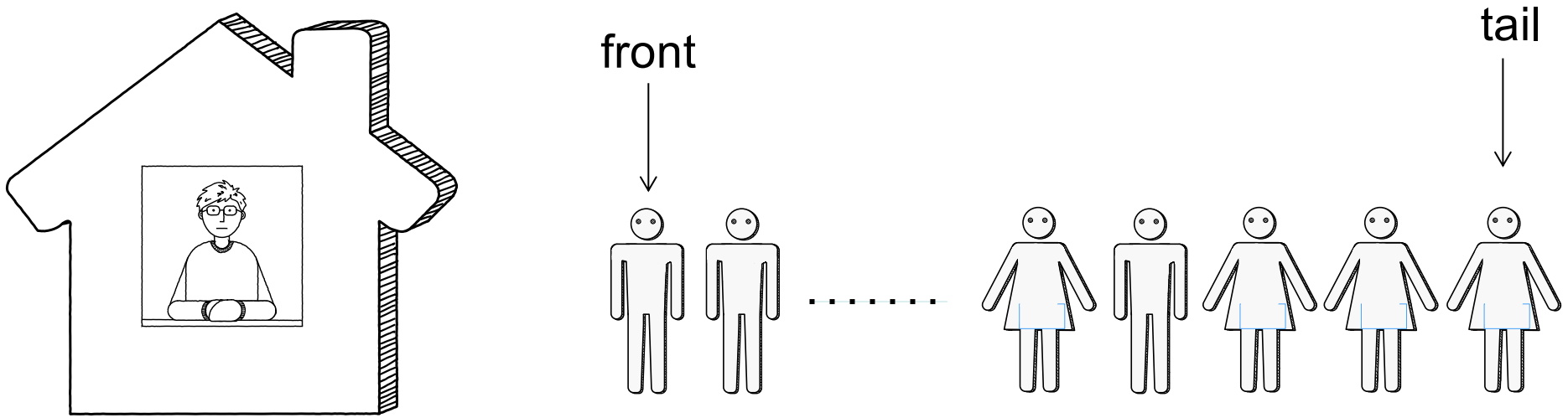
A queue data structure works in the same way as a queue in real life



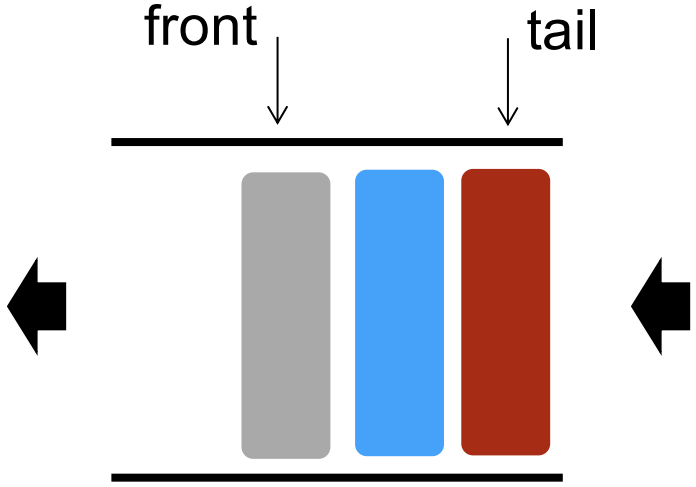
A person joins a queue at the tail of the queue

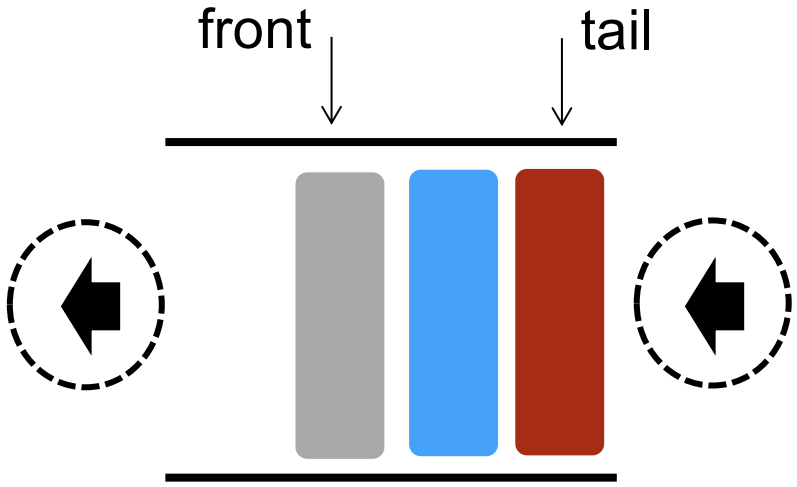


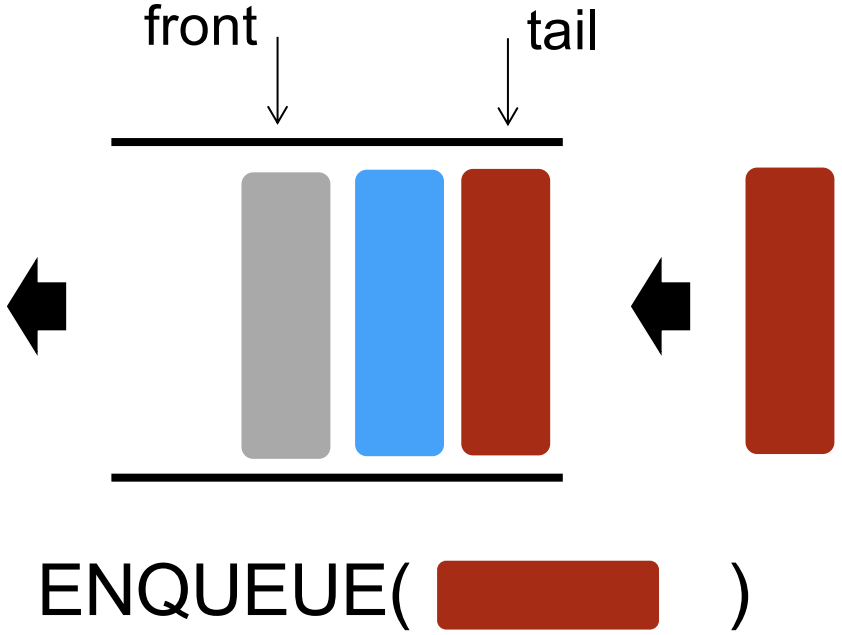
- The person at the front of the queue is served first
- Once person is served they leave the queue.



FIFO: First In – First Out

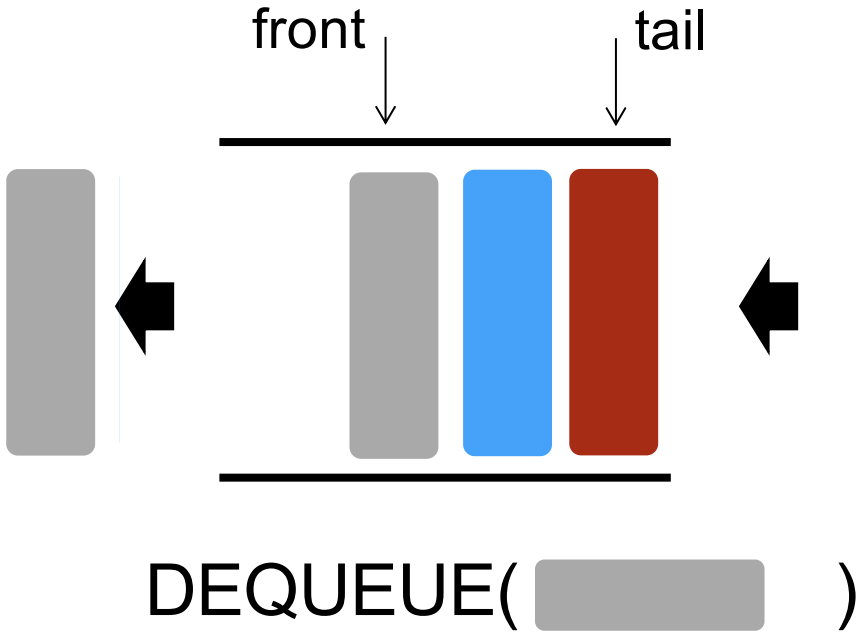

front
tail


front
tail



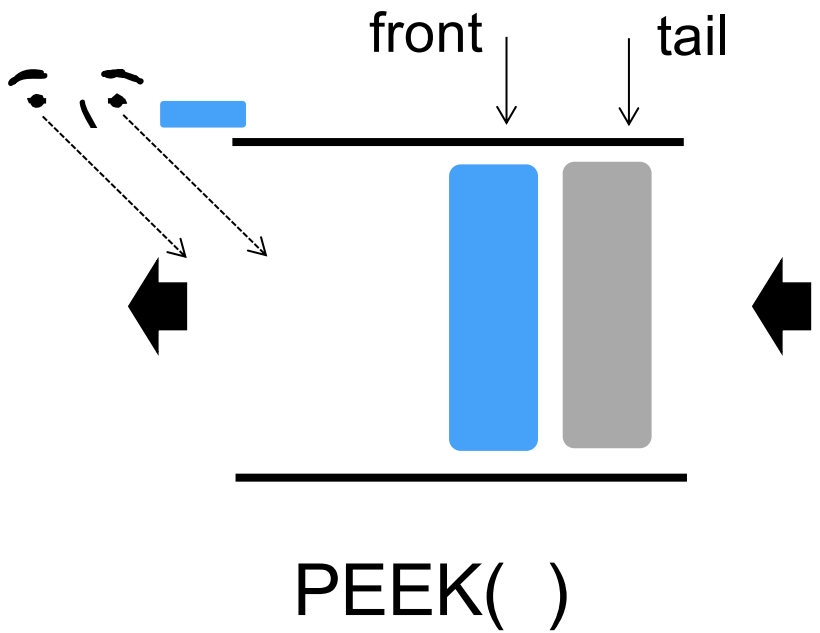
The insertion of a new element into the queue is called **ENQUEUE**

A New element is inserted a the tail of the queue



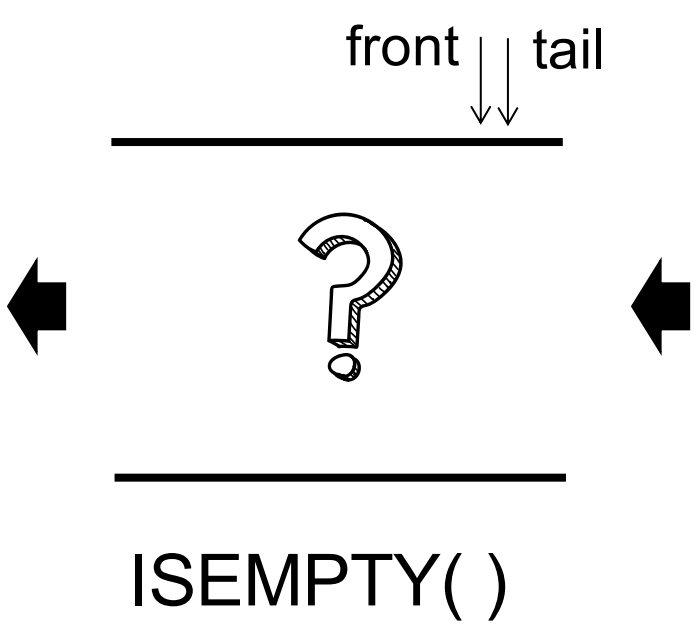
The removal of an element from the queue is called **DEQUEUE**

DEQUEUE operation removes the first element of the queue.

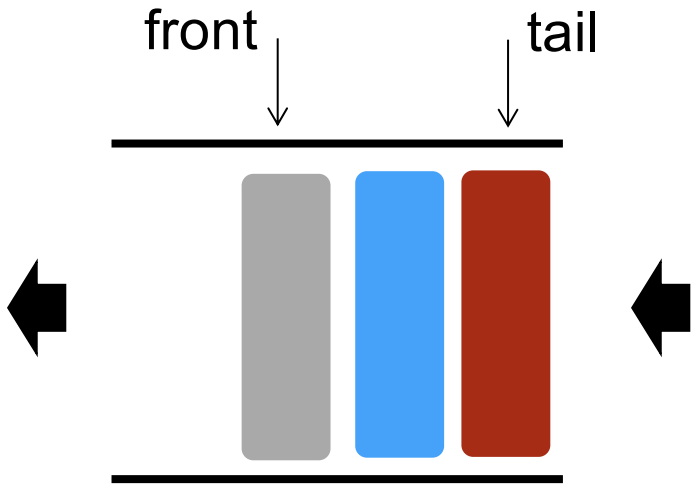


If you want to know the content of the queue, you can only at the front of queue

- this operation is call PEEK()
- PEEK() return the value of the element at the front of the queue.

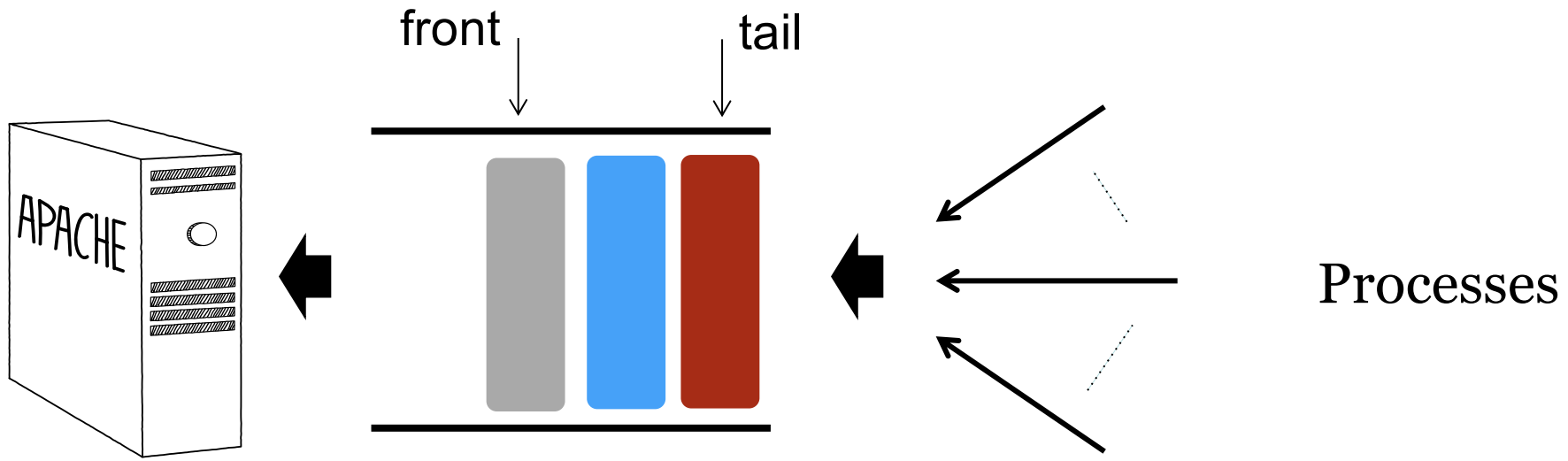


ISEMPTY( )

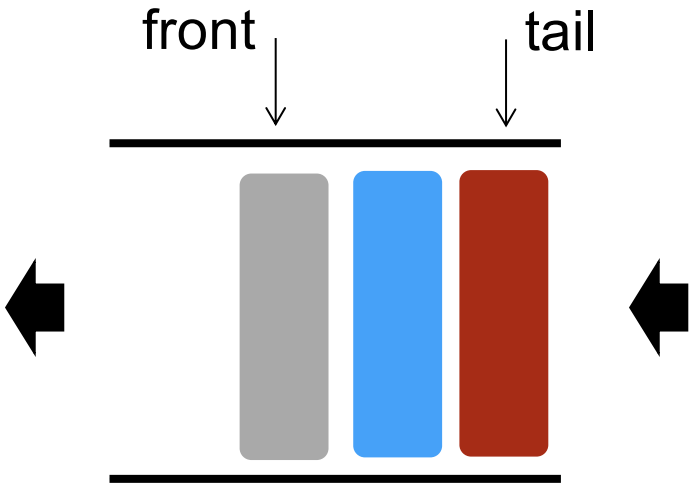


A queue a data structure were elements are inserted in one end and removed from the the other end.

One application of queue



Server can process one process at the time.



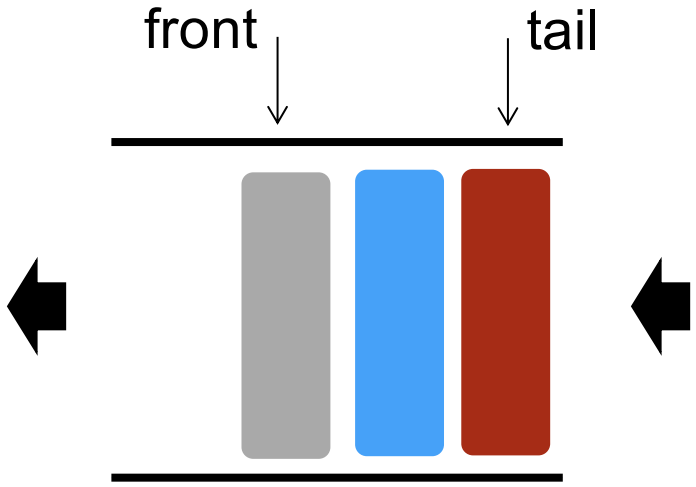
Summary

A queue is a linear data structure

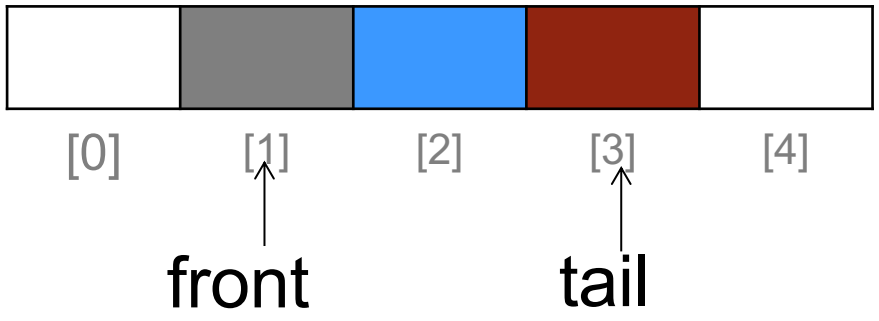
- ENQUEUE
- DEQUEUE
- PEEK()
- ISEMPTY()

Queue implementation

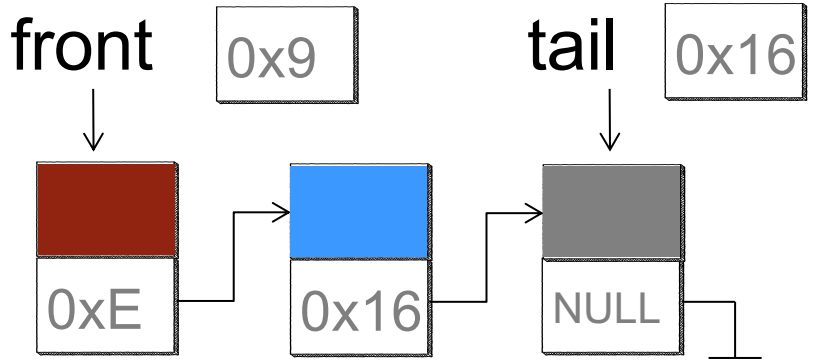
Queues
implementation arrays

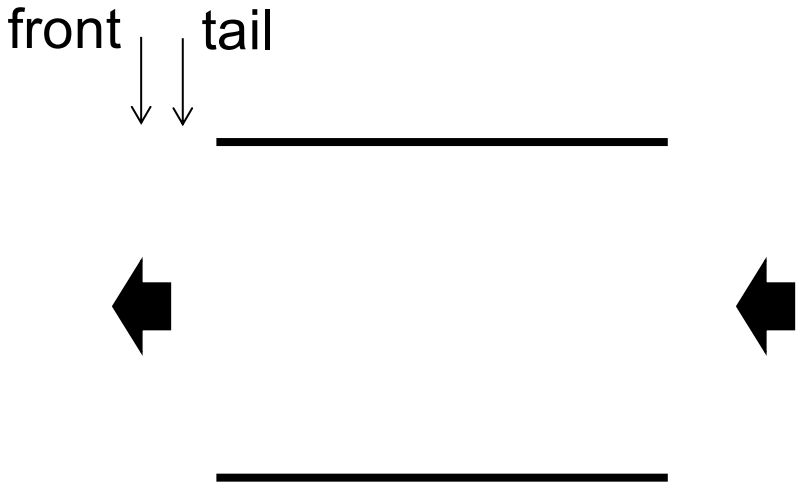


Array implementation

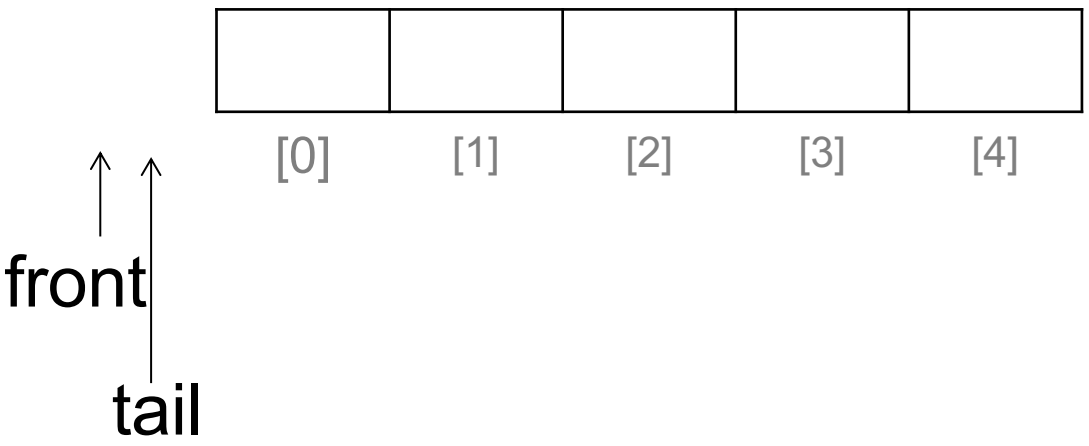


Linked list implementation

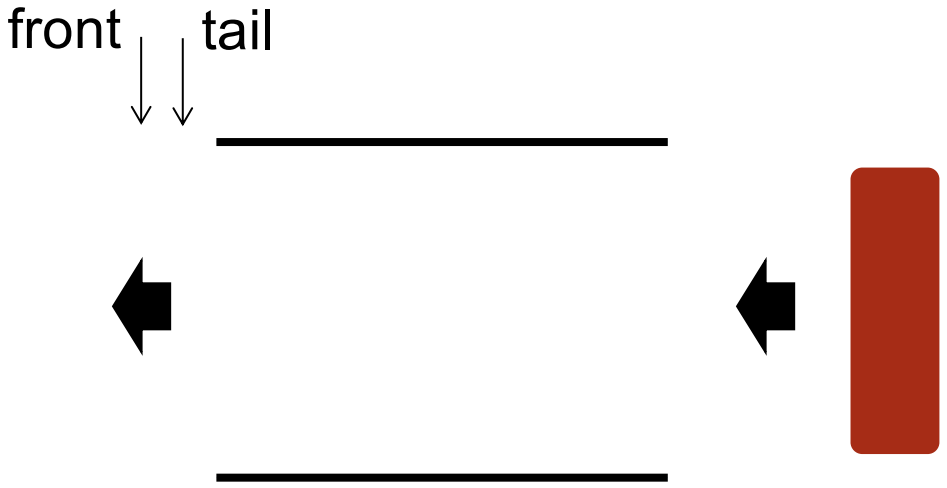




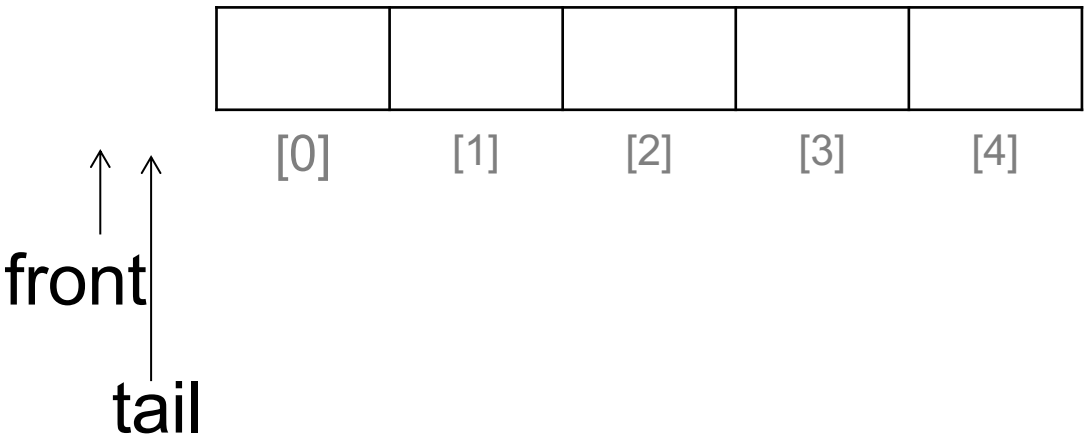
Array implementation



- We first create an array
- initialise front and tail to -1

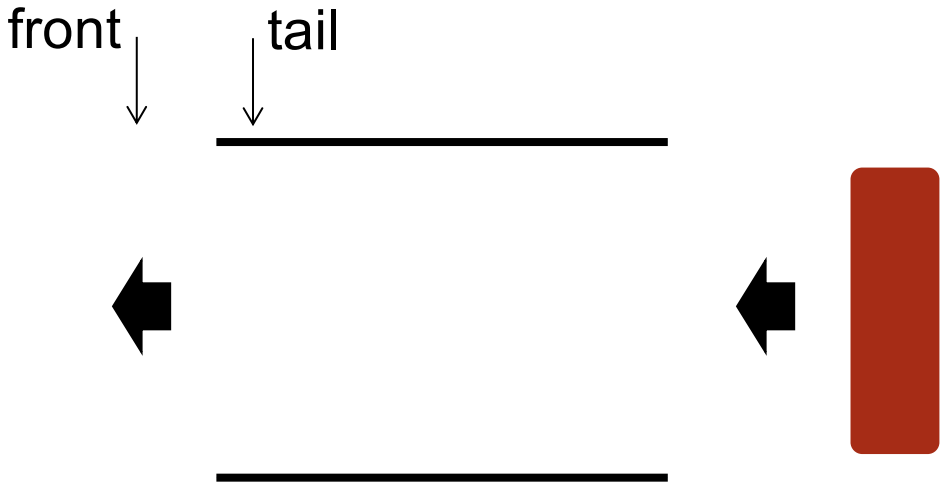


Array implementation



To enqueue one element into the list:

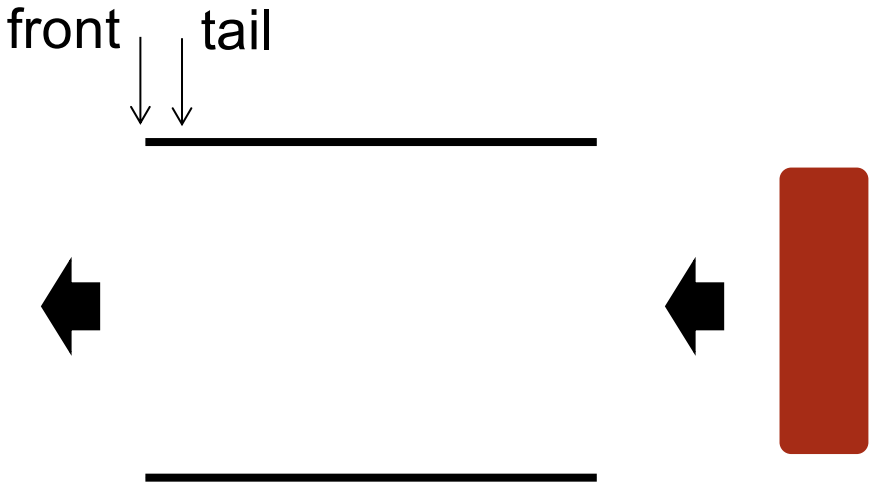
We first need to check if the queue is empty



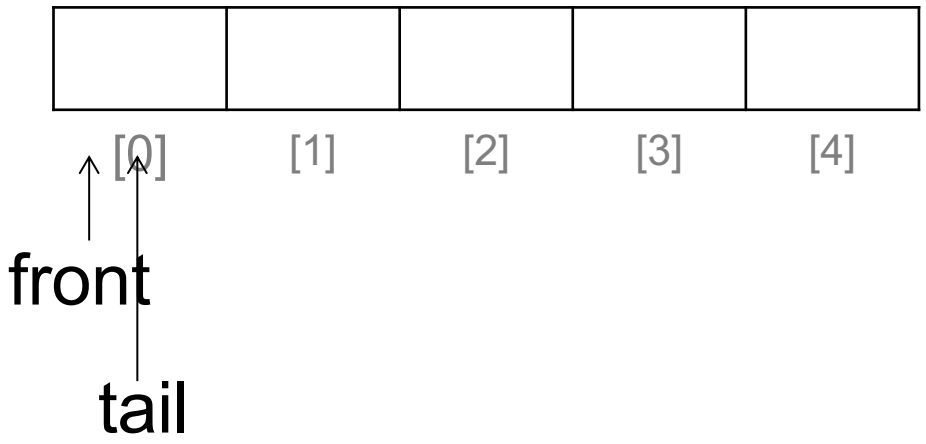
Array implementation

If the queue is empty:

- Update tail to point o

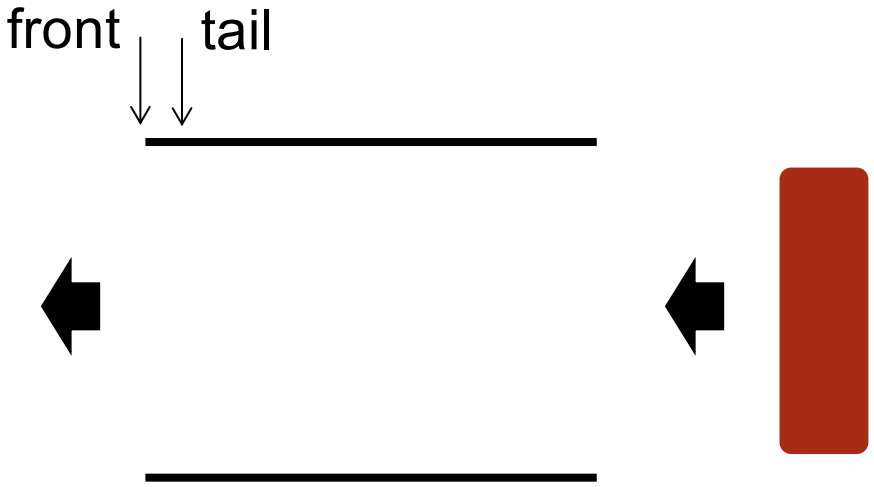


Array implementation



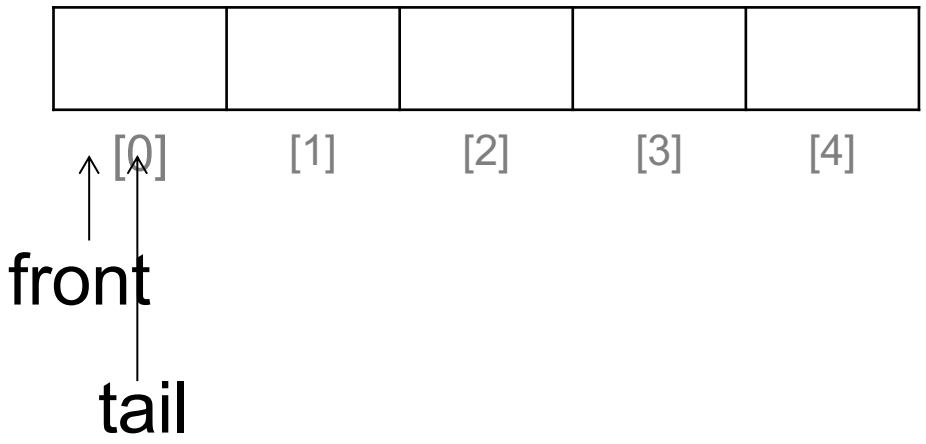
If the queue is empty:

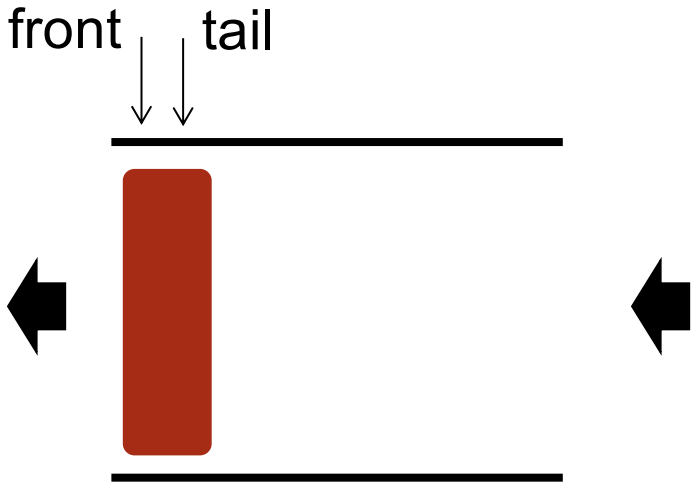
- Update tail to point 0
- Update front to point 0



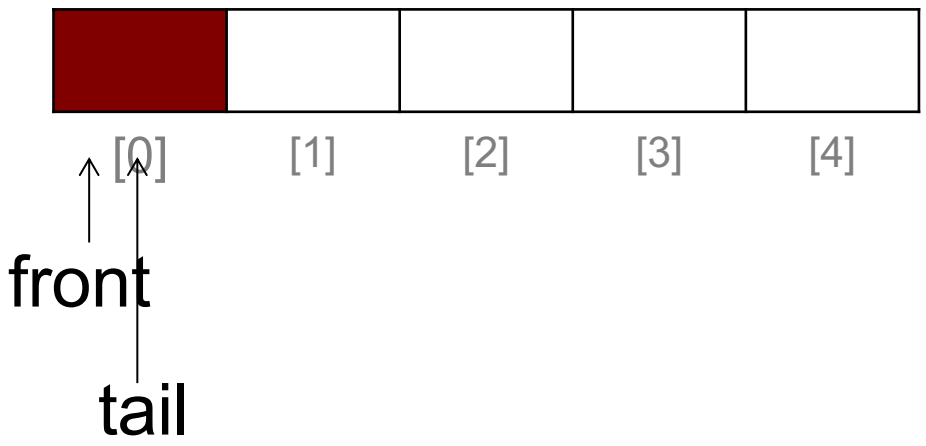

```
function ENQUEUE(x)
    if (ISEMPTY())
      front=0
      tail=0
```

Array implementation

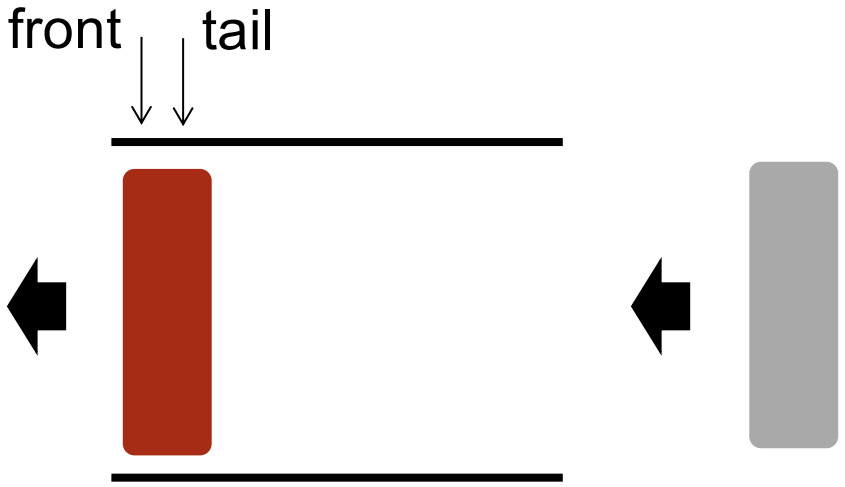




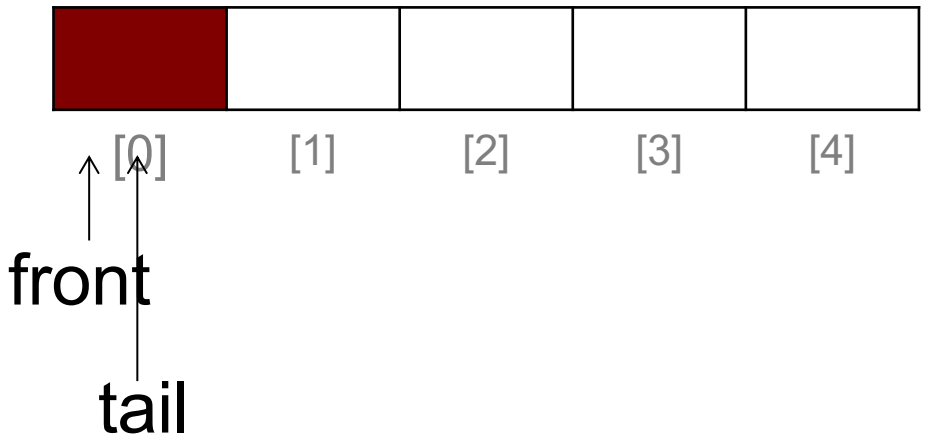
Array implementation



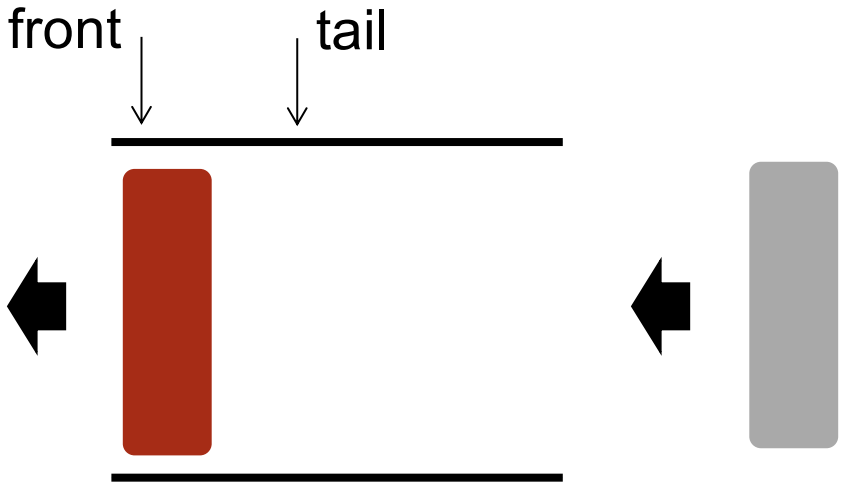

```
function ENQUEUE(x)
    if (ISEMPTY())
      front=0
      tail=0
      A[tail]=x
```



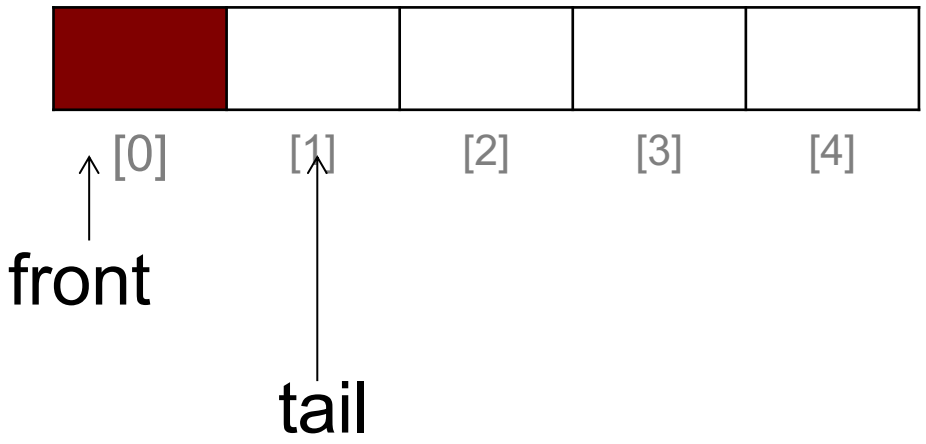
Array implementation



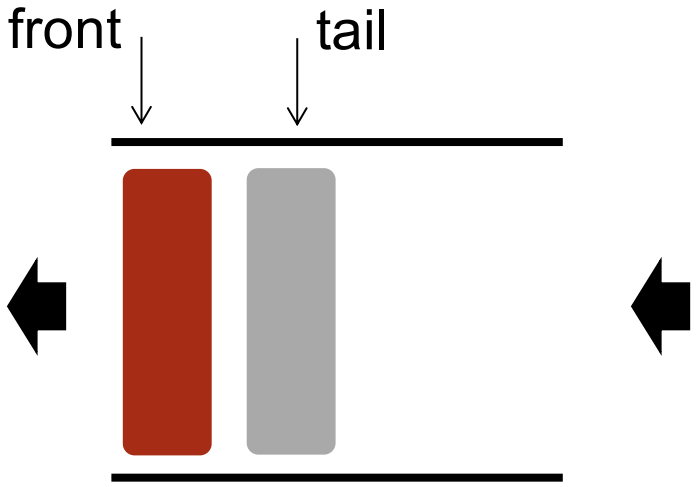

```
function ENQUEUE(x)
    if (ISEMPTY())
      front=0
      tail=0
      A[tail]=x
```



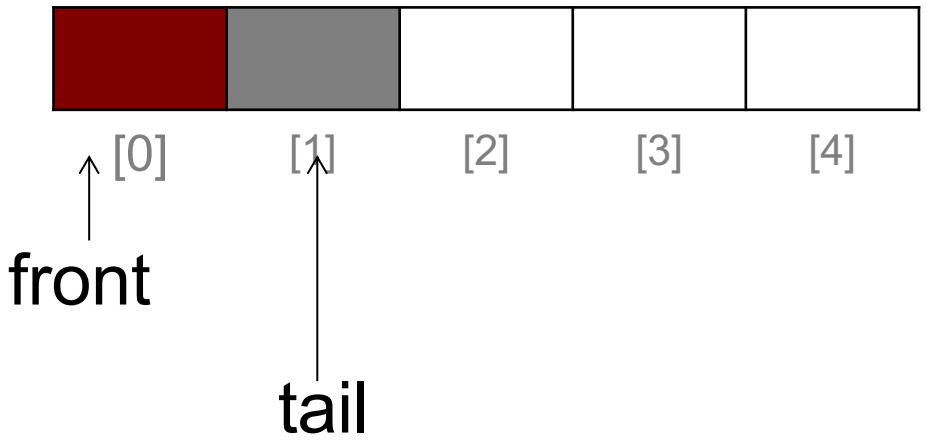
Array implementation



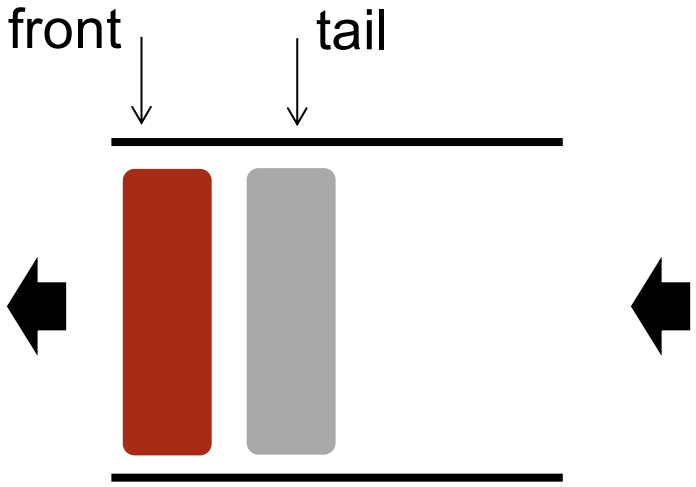

```
function ENQUEUE(x)
    if (ISEMPTY())
        front=0
        tail=0
        A[tail]=x
    else
        tail=tail+1
```



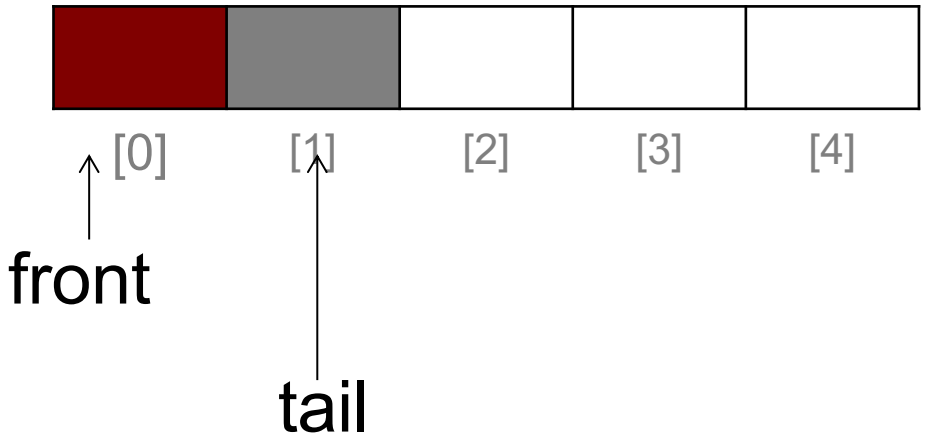
Array implementation



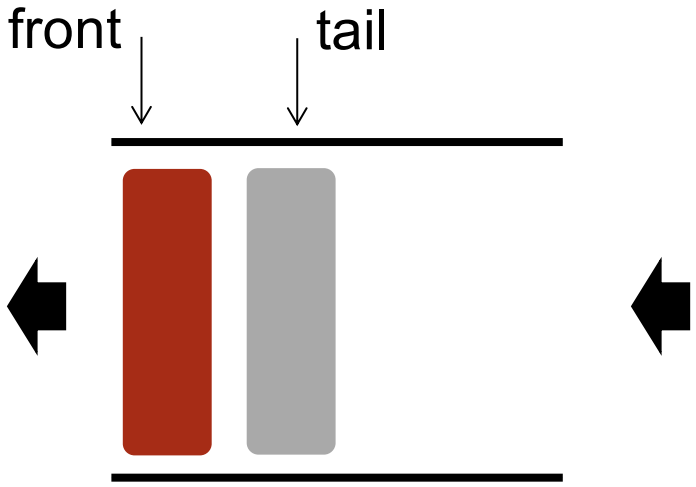

```
function ENQUEUE(x)
    if (ISEMPTY())
        front=0
        tail=0
        A[tail]=x
    else
        tail=tail+1
        A[tail]=x
```



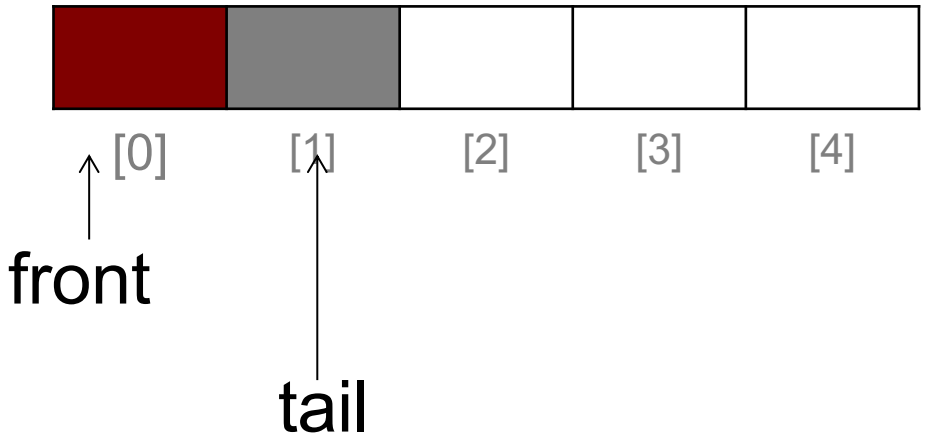
Array implementation



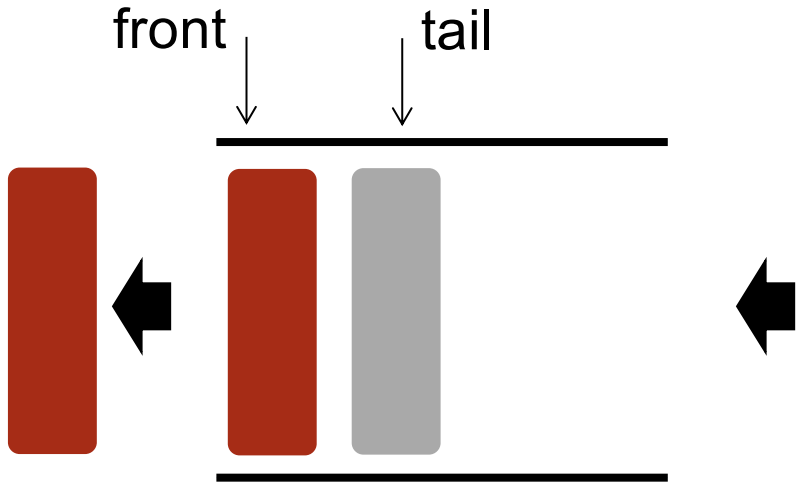

```
function ENQUEUE(x)
    if (ISEMPTY())
       front=0
       tail=0
       A[tail]=x
  else
       tail=tail+1
       A[tail]=x
```



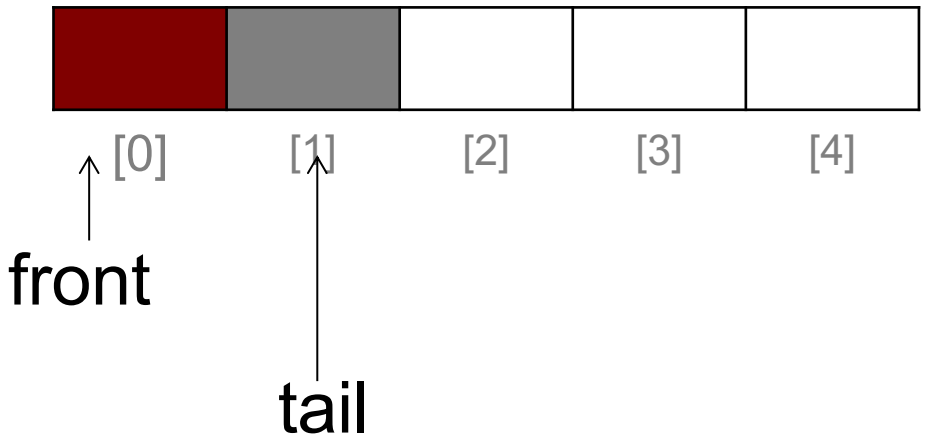
Array implementation

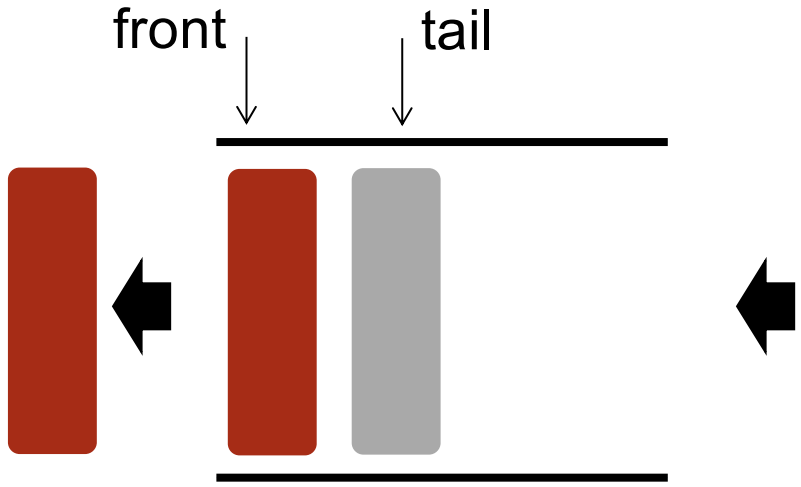



```
function ENQUEUE(x)
    if (ISEMPTY())
        front=0
        tail=0
    else
        tail=tail+1
    A[tail]=x
```

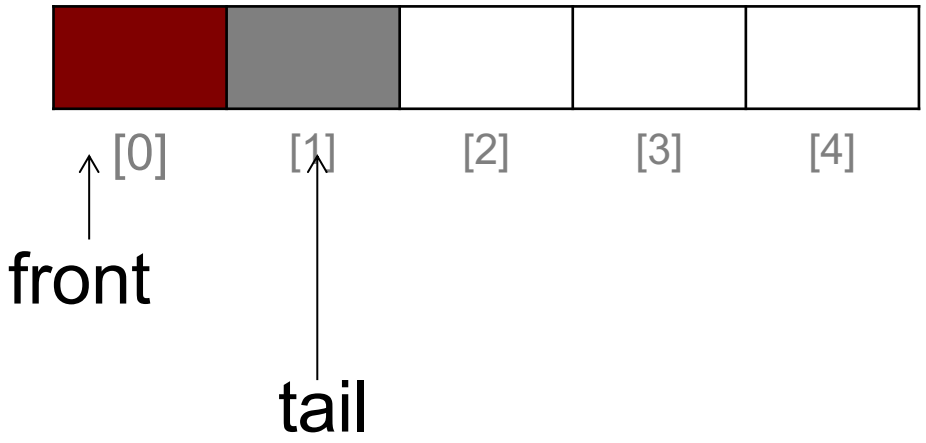


Array implementation

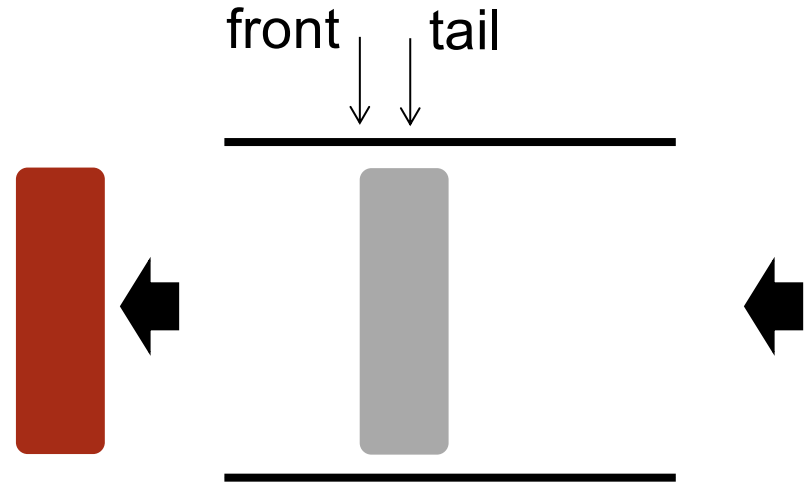




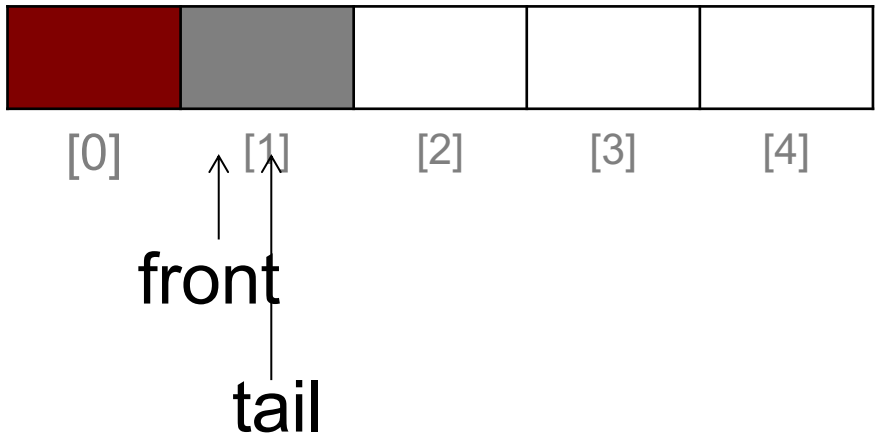
Array implementation



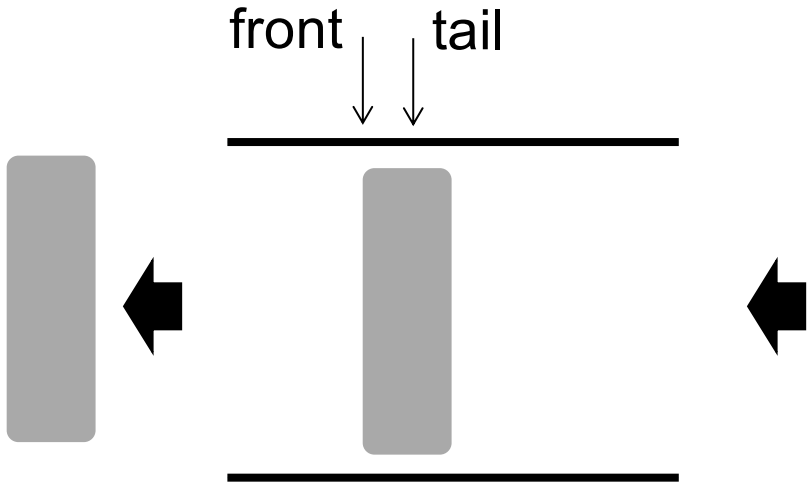

```
function DEQUEUE()
    if (ISEMPTY())
        print("Queue is empty")
        return
```



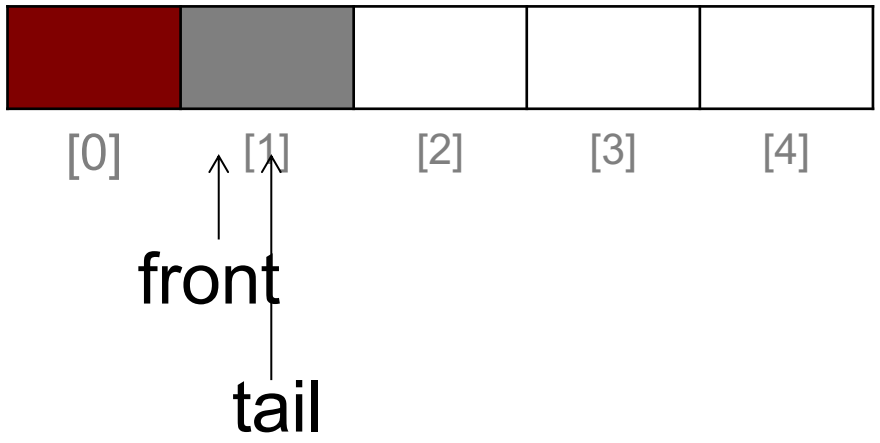
Array implementation



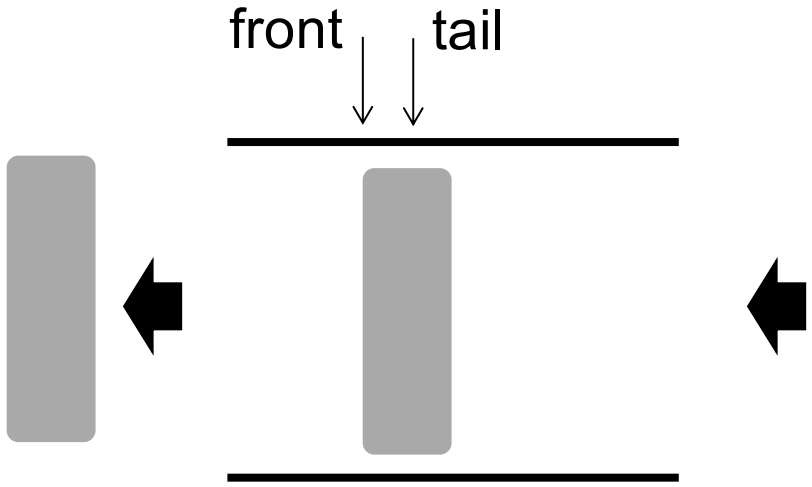

```
function DEQUEUE()
    if (ISEMPTY())
        print("Queue is empty")
        return
   front=front+1
```



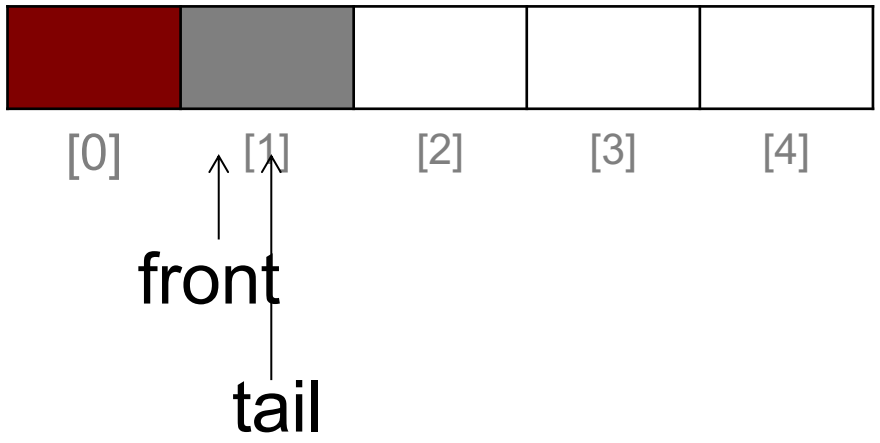
Array implementation



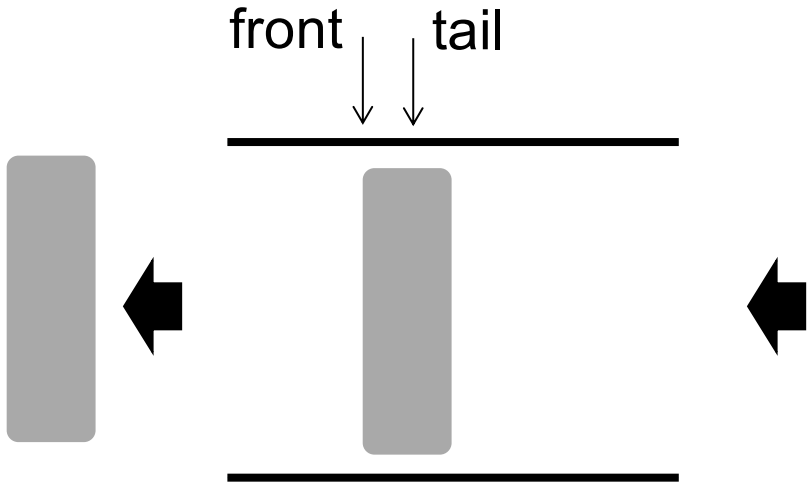

```
function DEQUEUE()
    if (ISEMPTY())
        print("Queue is empty")
        return
   front=front+1
```



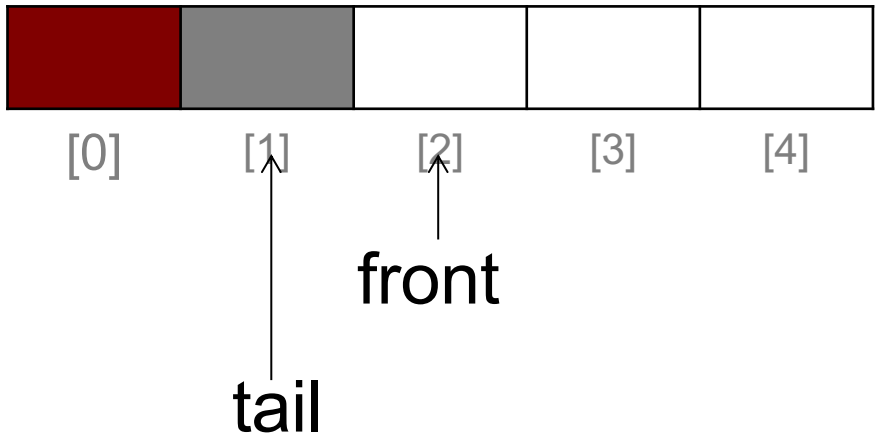
Array implementation



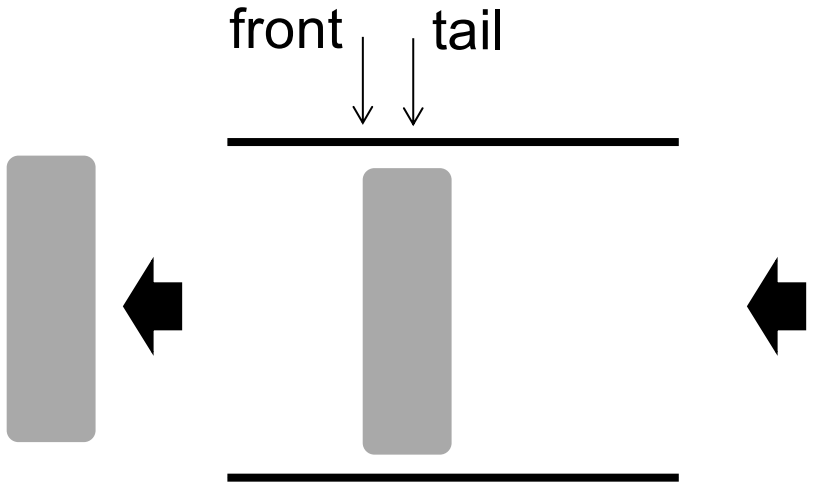

```
function DEQUEUE()
    if (ISEMPTY())
        print("Queue is empty")
        return
  front=front+1
```



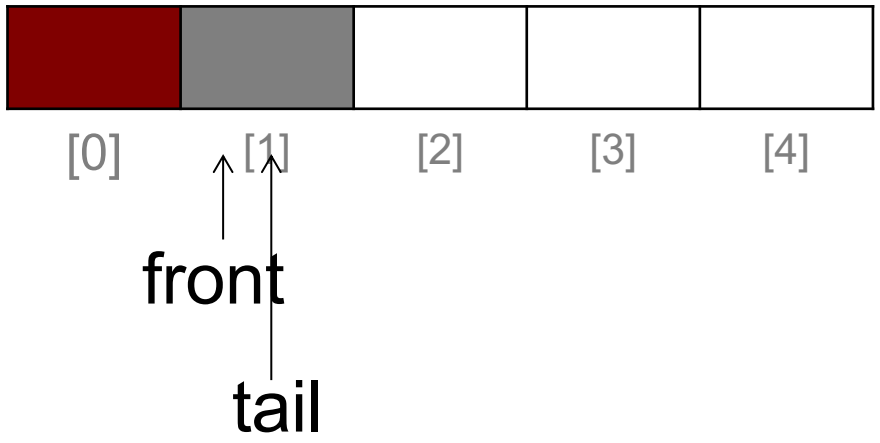
Array implementation



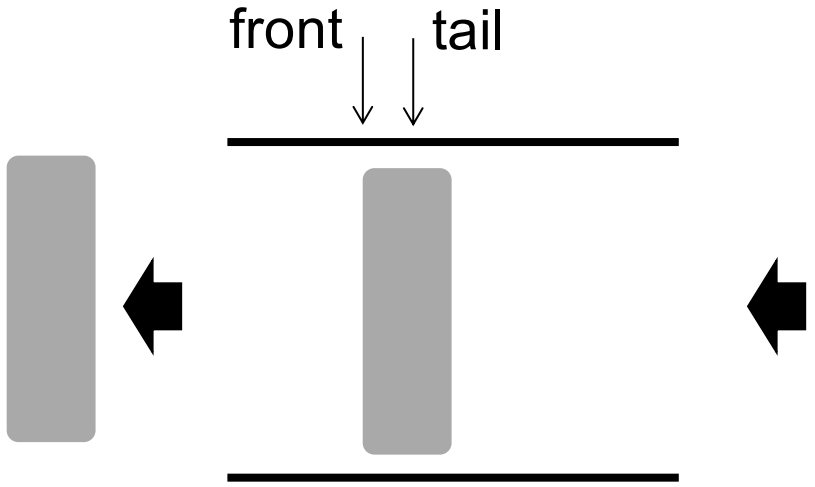

```
function DEQUEUE()
    if (ISEMPTY())
        print("Queue is empty")
        return
   front=front+1
```



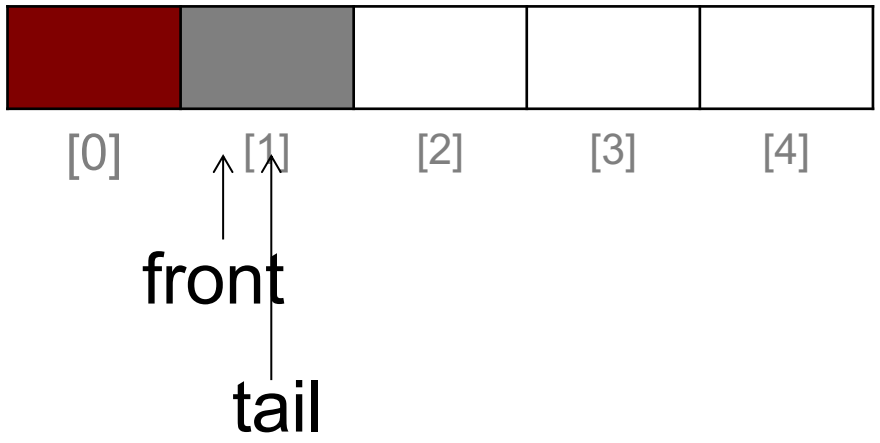
Array implementation



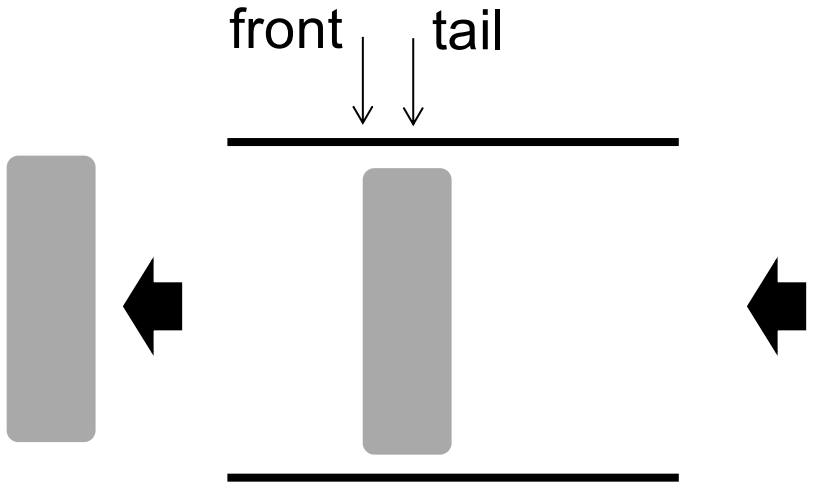

```
function DEQUEUE()
    if (ISEMPTY())
        print("Queue is empty")
        return
  if (front==tail)
        front=-1
        tail=-1
  else
        front=front+1
```



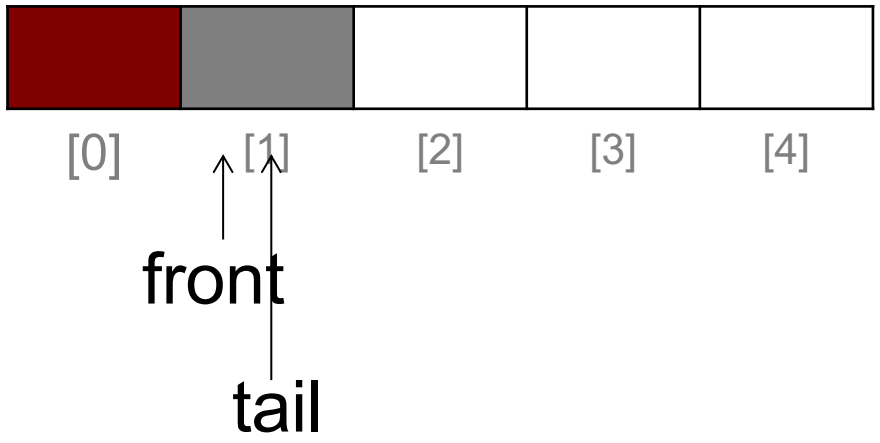
Array implementation



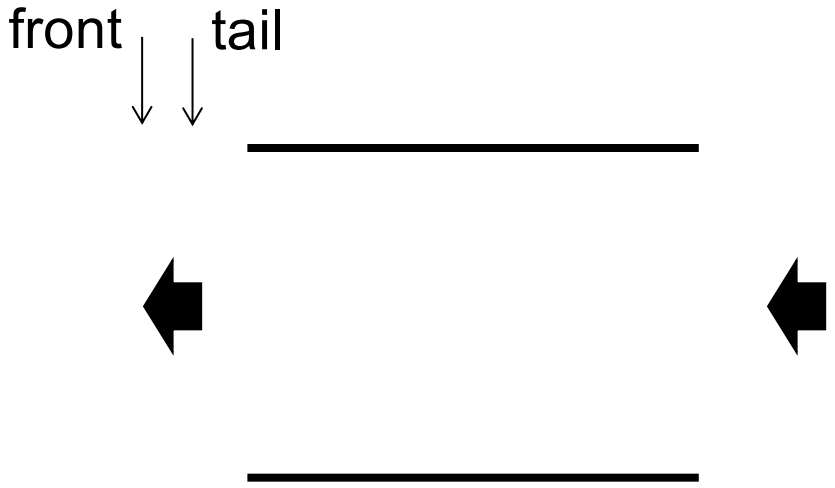

```
function DEQUEUE()
    if (ISEMPTY())
        print("Queue is empty")
        return
  if (front==tail)
      front=-1
      tail=-1
  else
      front=front+1
```



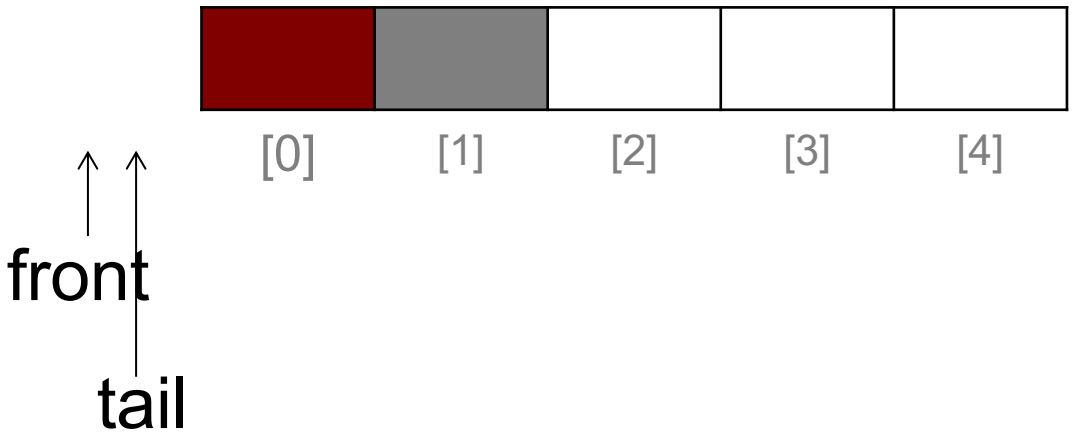
Array implementation



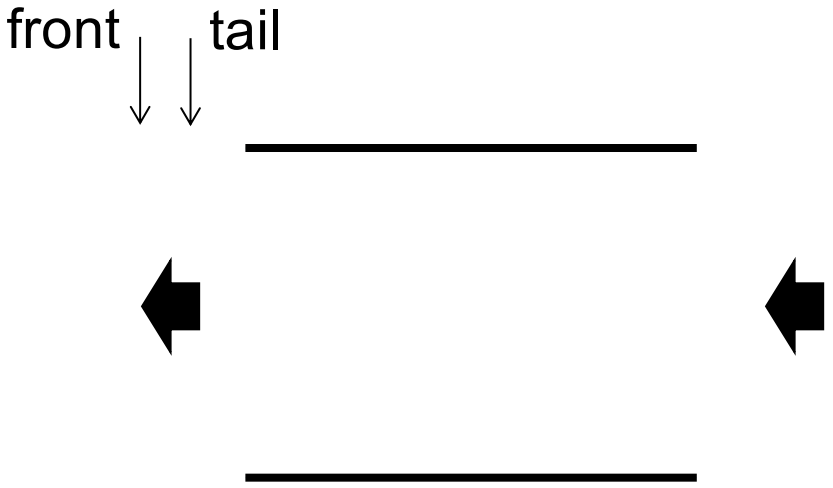

```
function DEQUEUE()
    if (ISEMPTY())
        print("Queue is empty")
        return
   if (front==tail)
        front=-1
        tail=-1
   else
        front=front+1
```



Array implementation



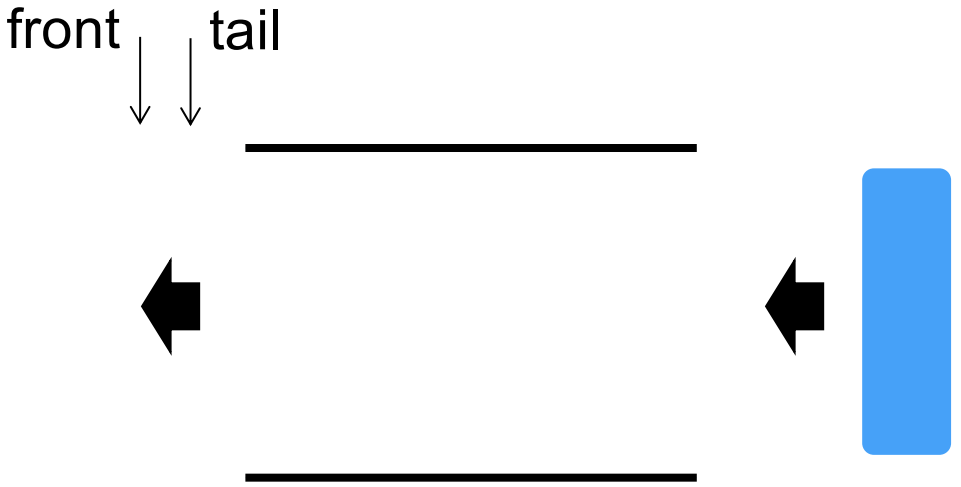

```
function DEQUEUE()
    if (ISEMPTY())
        print("Queue is empty")
        return
   if (front==tail)
        front=-1
        tail=-1
   else
        front=front+1
```




```
function ENQUEUE(x)
   if (ISEMPTY())
      front=0
      tail=0
   else
      tail=tail+1
   A[tail]=x
```

Array implementation

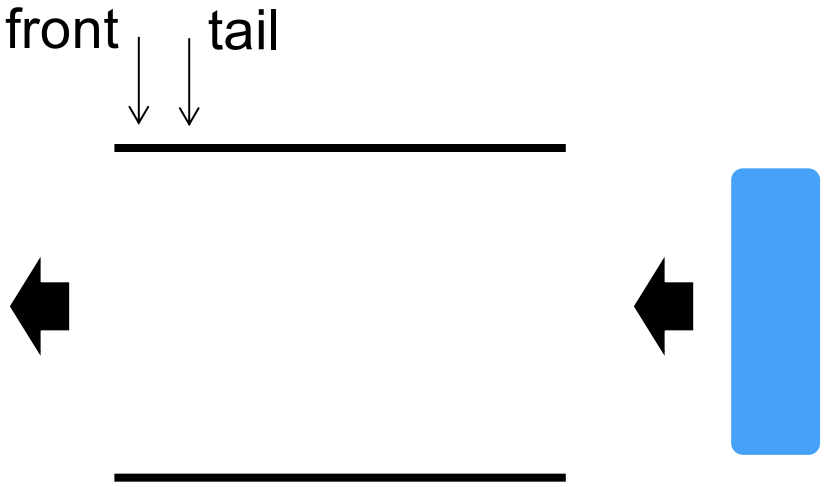

```
function DEQUEUE()
   if (ISEMPTY())
      print("Queue is empty")
      return
  if (front==tail)
      front=-1
      tail=-1
  else
      front=front+1
```




```
function ENQUEUE(x)
    if (ISEMPTY())
        front=0
        tail=0
    else
        tail=tail+1
    A[tail]=x
```

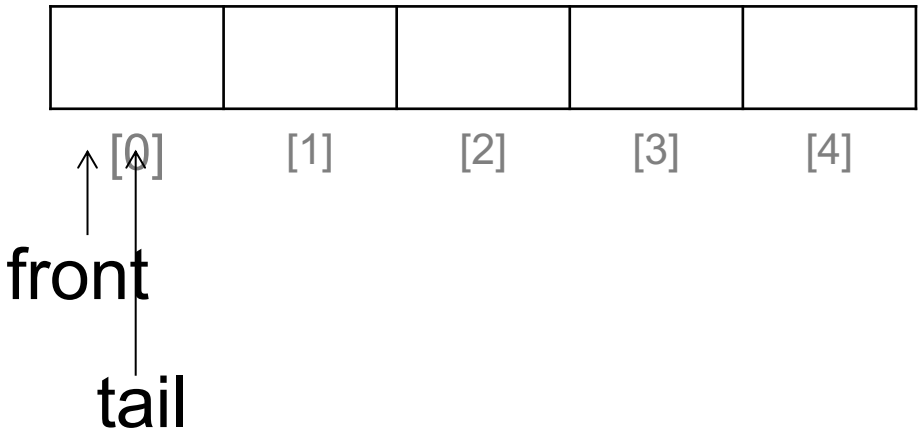
Array implementation


```
function DEQUEUE()
    if (ISEMPTY())
        print("Queue is empty")
        return
  if (front==tail)
        front=-1
        tail=-1
  else
        front=front+1
```

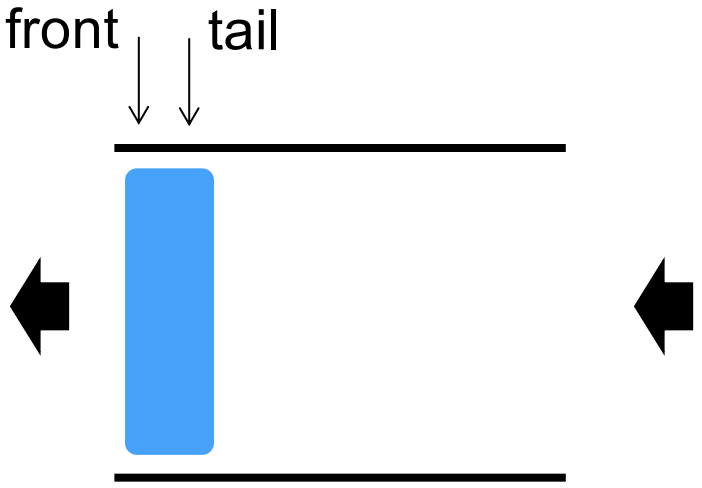



```
function ENQUEUE(x)
    if (ISEMPTY())
        front=0
        tail=0
    else
        tail=tail+1
    A[tail]=x
```

Array implementation

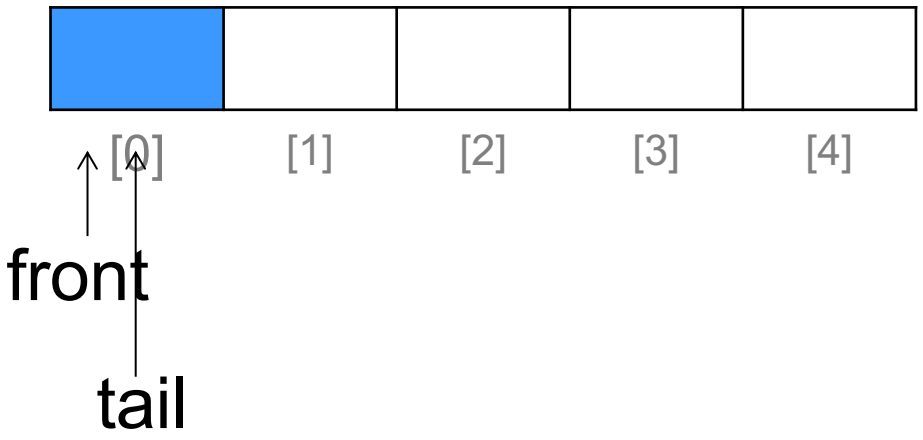



```
function DEQUEUE()
    if (ISEMPTY())
        print("Queue is empty")
        return
  if (front==tail)
        front=-1
        tail=-1
  else
        front=front+1
```

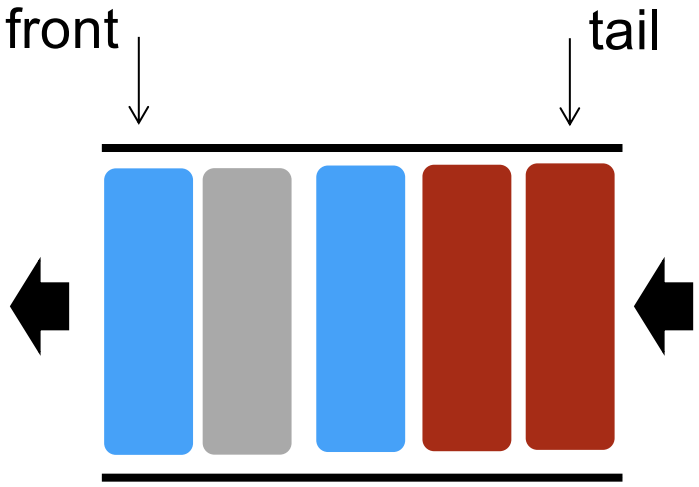



```
function ENQUEUE(x)
    if (ISEMPTY())
        front=0
        tail=0
    else
        tail=tail+1
    A[tail]=x
```

Array implementation

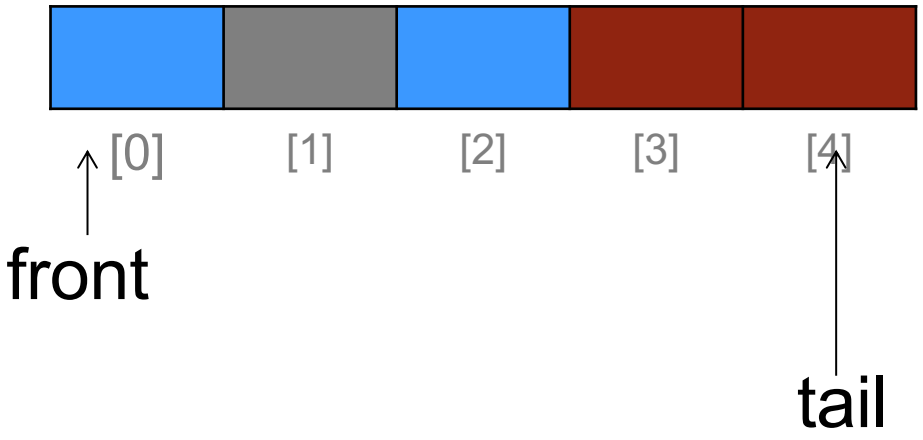



```
function DEQUEUE()
    if (ISEMPTY())
        print("Queue is empty")
        return
  if (front==tail)
        front=-1
        tail=-1
  else
        front=front+1
```

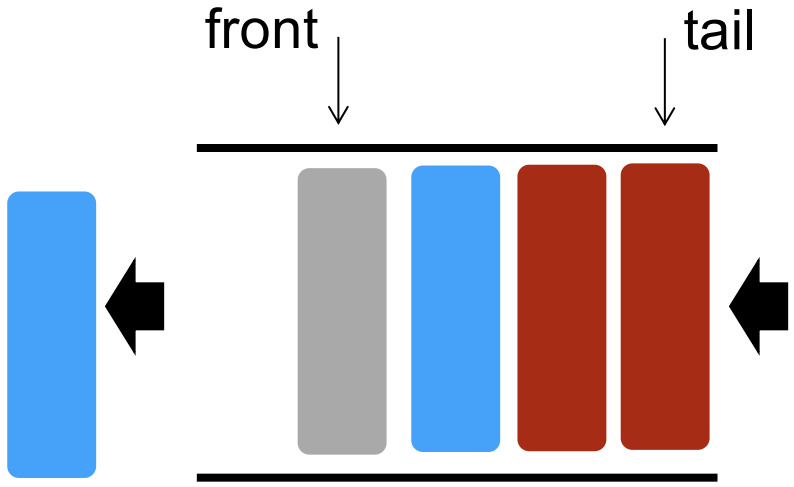



```
function ENQUEUE(x)
    if (ISEMPTY())
        front=0
        tail=0
    else
        tail=tail+1
    A[tail]=x
```

Array implementation

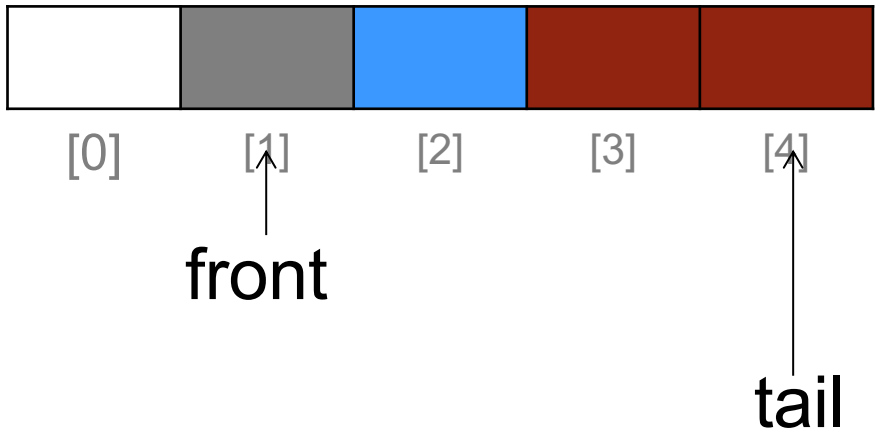



```
function DEQUEUE()
    if (ISEMPTY())
        print("Queue is empty")
        return
  if (front==tail)
        front=-1
        tail=-1
  else
        front=front+1
```

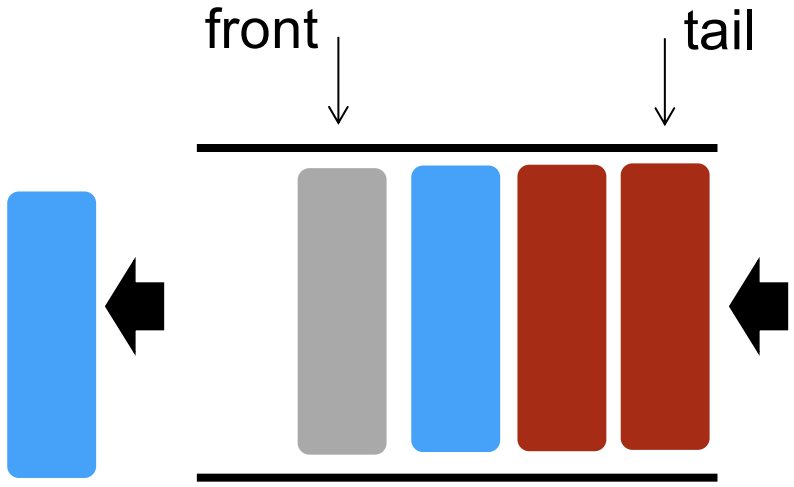



```
function ENQUEUE(x)
    if (ISEMPTY())
        front=0
        tail=0
    else
        tail=tail+1
    A[tail]=x
```

Array implementation

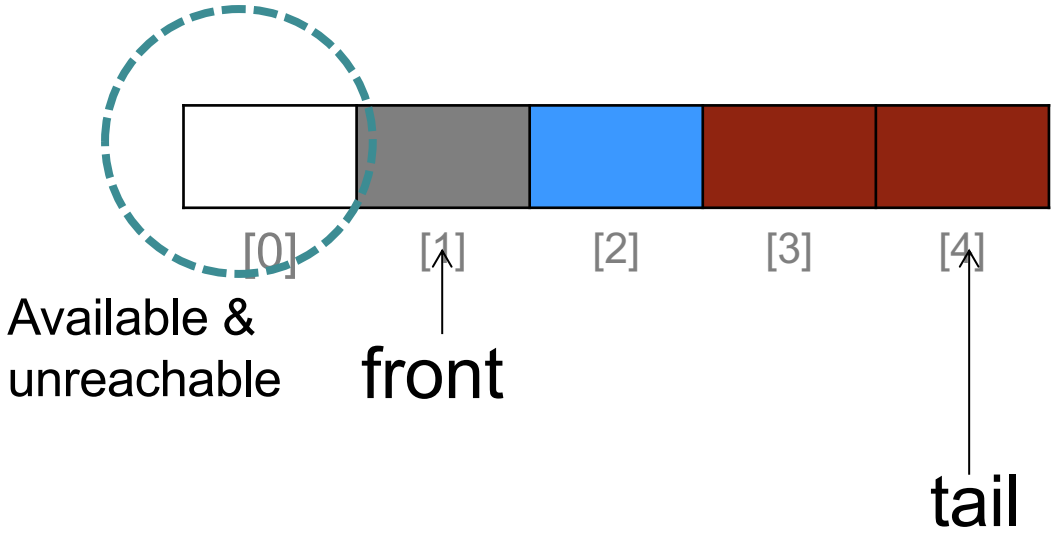



```
function DEQUEUE()
    if (ISEMPTY())
        print("Queue is empty")
        return
  if (front==tail)
        front=-1
        tail=-1
  else
        front=front+1
```




```
function ENQUEUE(x)
    if (ISEMPTY())
        front=0
        tail=0
    else
        tail=tail+1
    A[tail]=x
```

Array implementation

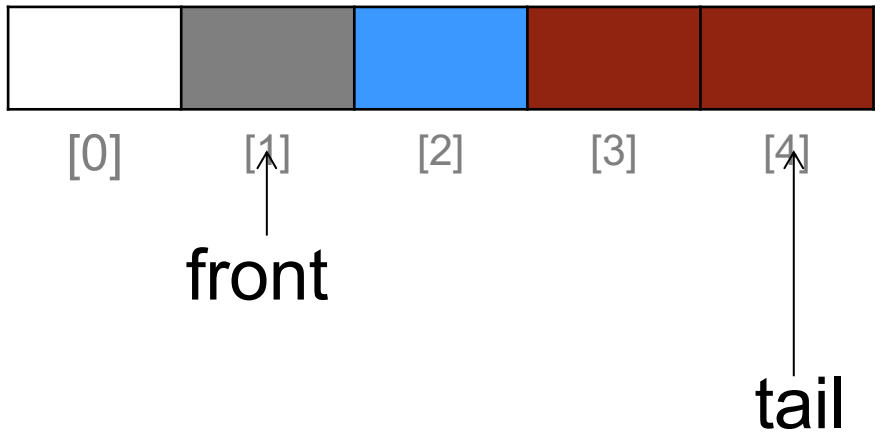



```
function DEQUEUE()
    if (ISEMPTY())
        print("Queue is empty")
        return
  if (front==tail)
        front=-1
        tail=-1
  else
        front=front+1
```

```
function ENQUEUE(x)
    if (ISEMPTY())
        front=0
        tail=0
    else
        tail=(tail+1)%N
    A[tail]=x
```

We need both pointer to traverse the array in a circular way
Once **front** or **tail** reach the end of the array, they should be able to jump to the first position array.

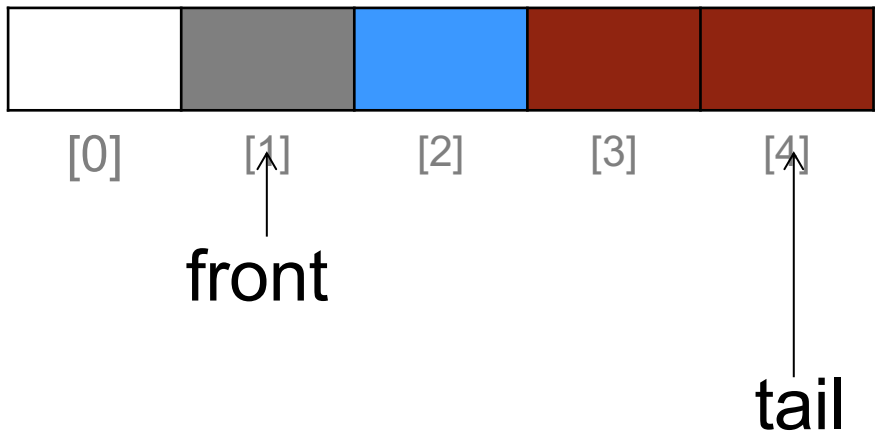
Array implementation



We also need to check if the queue is full

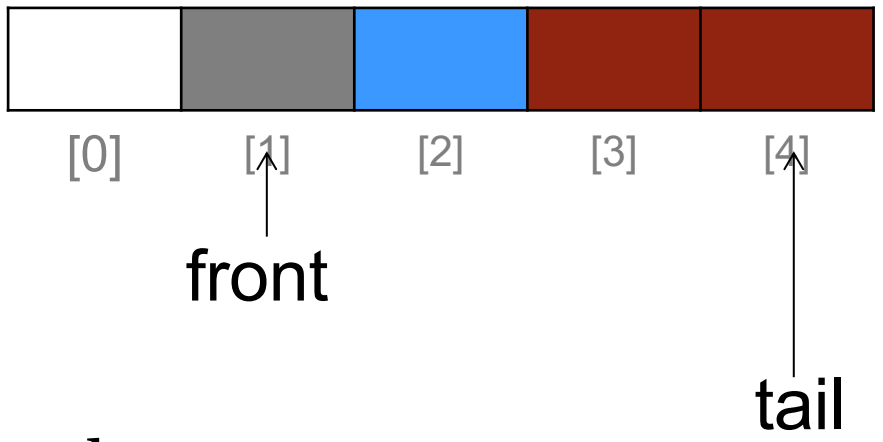
```
function ENQUEUE(x)
    if((tail+1)%N==front)
         print("Queue full")
         return
    if (ISEMPTY())
       front=0
       tail=0
    else
       tail=(tail+1)%N
    A[tail]=x
```

Array implementation



```
function ENQUEUE(x)
   if((tail+1)%N==front)
       print(“Queue full”)
       return
   if (ISEMPTY())
      front=0
      tail=0
   else
      tail=(tail+1)%N
   A[tail]=x
```

Array implementation

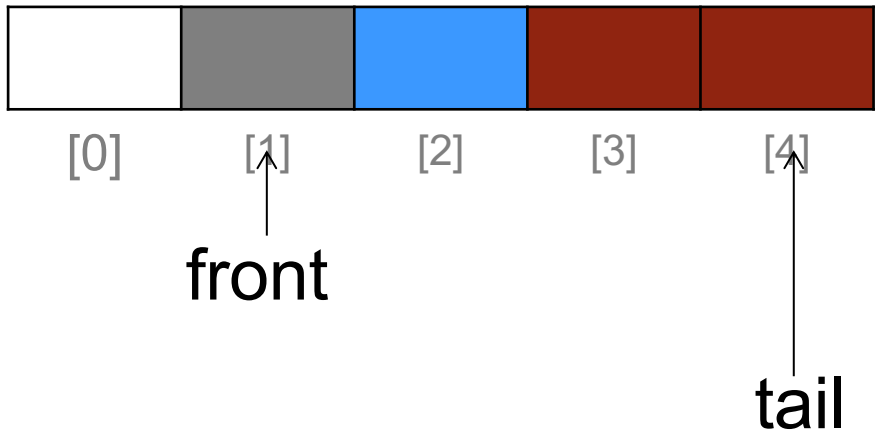


This way now the tail traverse the array in a circular way

We also need to make the front of the queue traverse the array in circular way

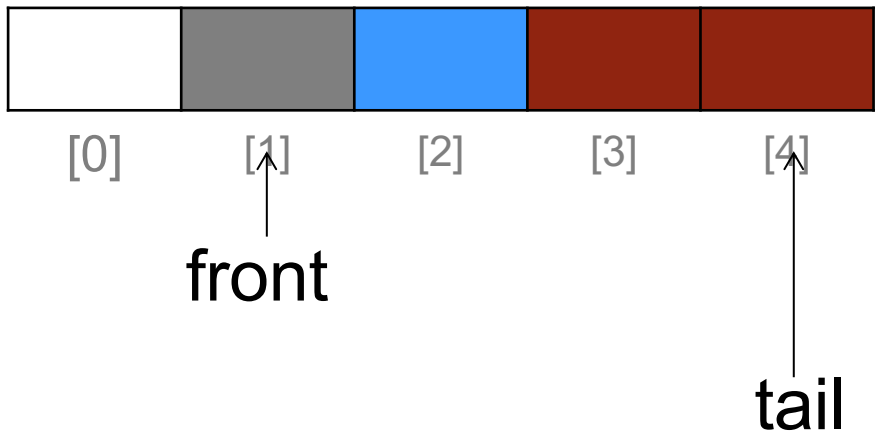
```
function DEQUEUE()
   if (ISEMPTY())
      print("Queue is empty")
      return
  if (front==tail)
      front=-1
      tail=-1
  else
      front=(front+1)%N
```

Array implementation



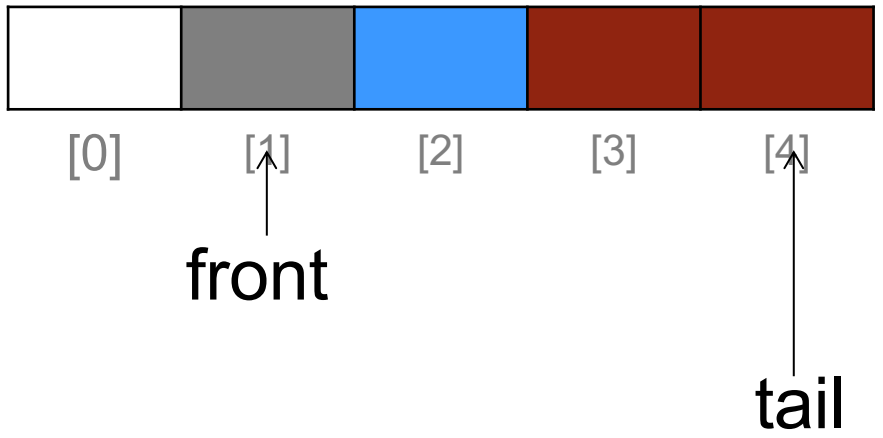
```
function PEEK()
    if(front==-1)
        print(“empty queue”)
        return
    return A[front]
```

Array implementation



```
function ISEMPTY()
   if(front==-1)
      return TRUE
   return FALSE
```

Array implementation

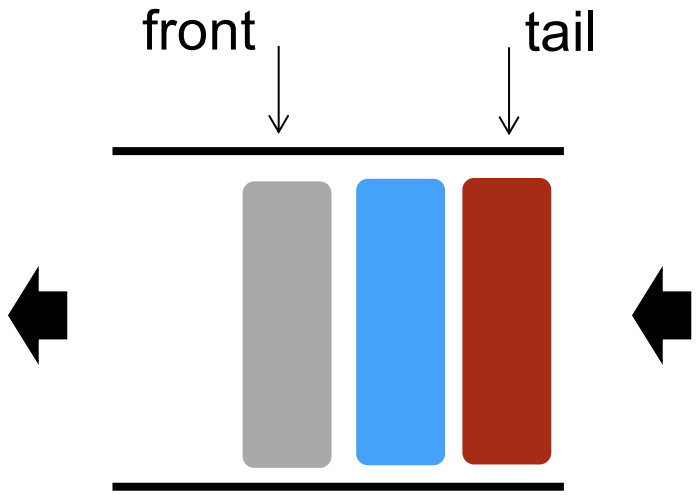


```
function ENQUEUE(x)
   if((tail+1)%N==front)
       print(“Queue full”)
       return
   if (ISEMPTY())
      front=0
      tail=0
   else
      tail=(tail+1)%N
   A[tail]=x
```

```
function PEEK()
   if(front==-1)
      print(“empty queue”)
      return
   return A[front]
```

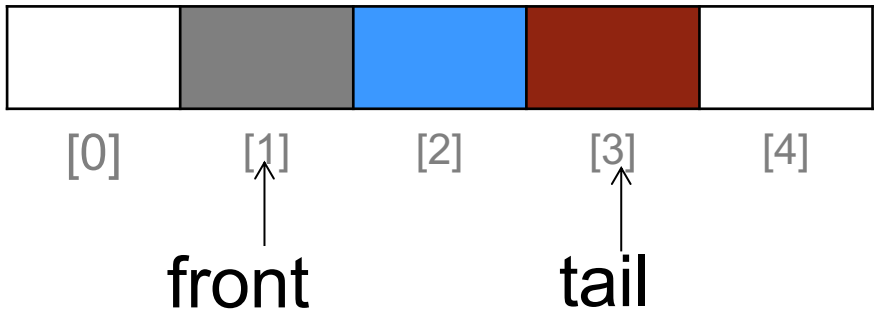
```
function DEQUEUE()
   if (ISEMPTY())
      print(“Queue is empty”)
      return
  if (front==tail)
      front=-1
      tail=-1
  else
      front=(front+1)%N
```

```
function ISEMPTY()
   if(front==-1)
      return TRUE
   return FALSE
```



Summary:
Queue implementation using arrays
Pseudocode
Both front and tail traverse array in a circular

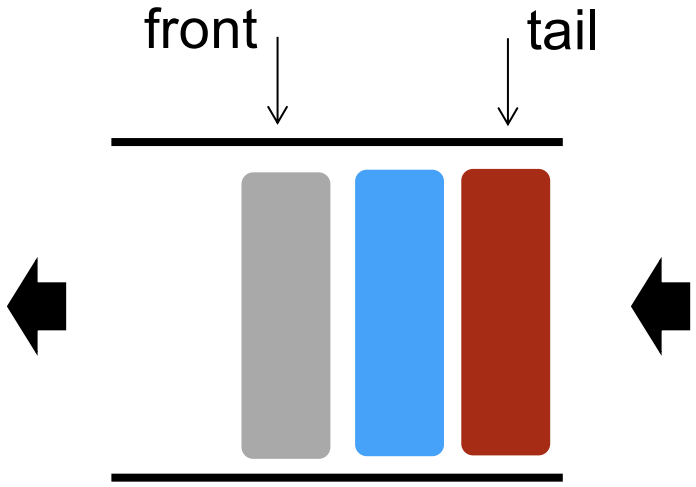
Array implementation



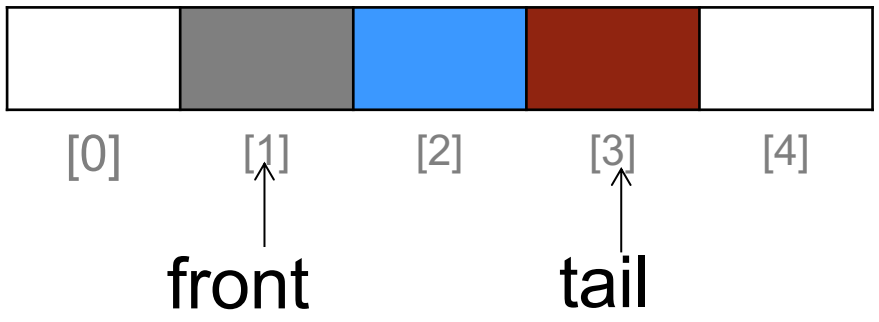
Queues implementation lists

Array implementation drawbacks:

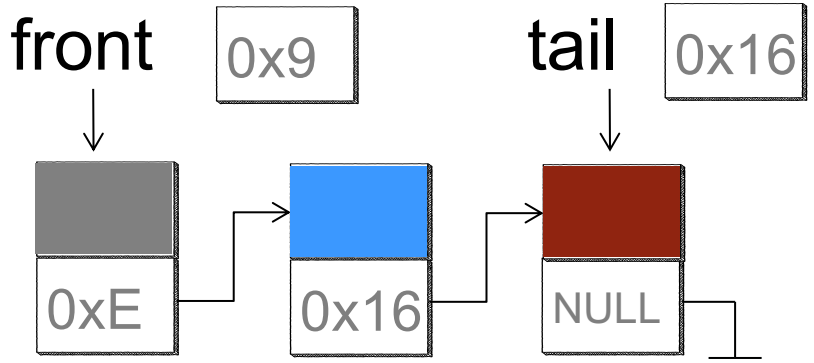
- Unused memory locations

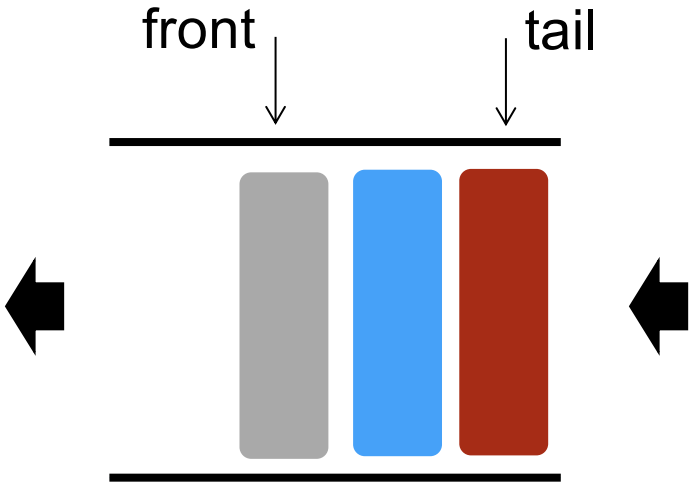


Array implementation

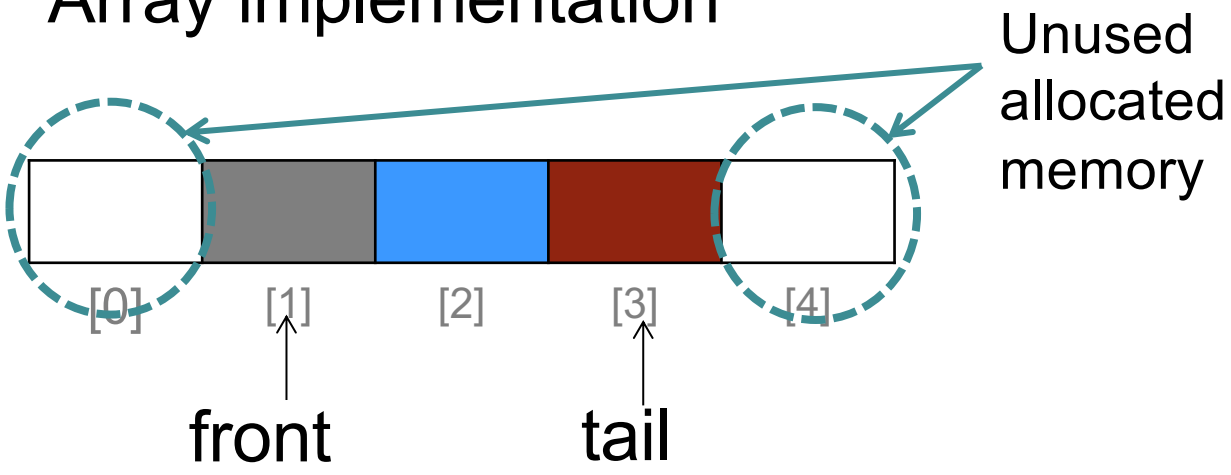


Linked list implementation

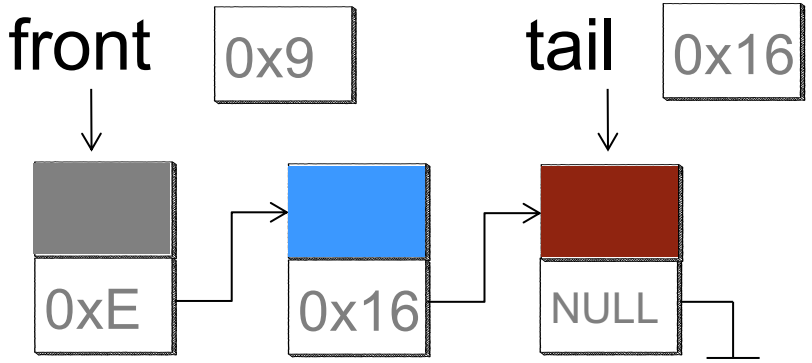


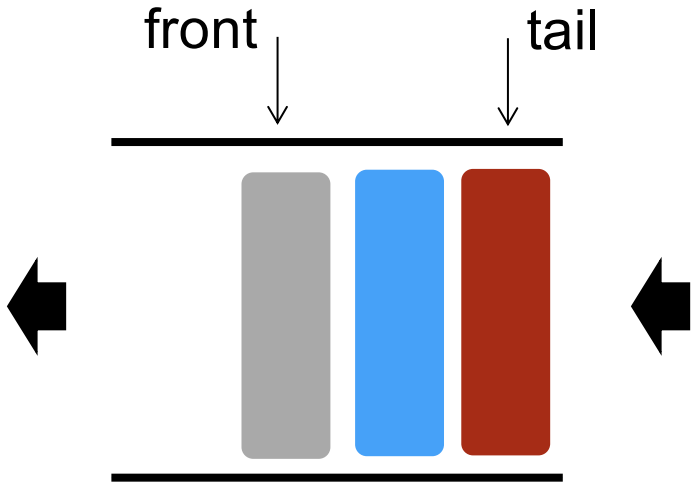


Array implementation

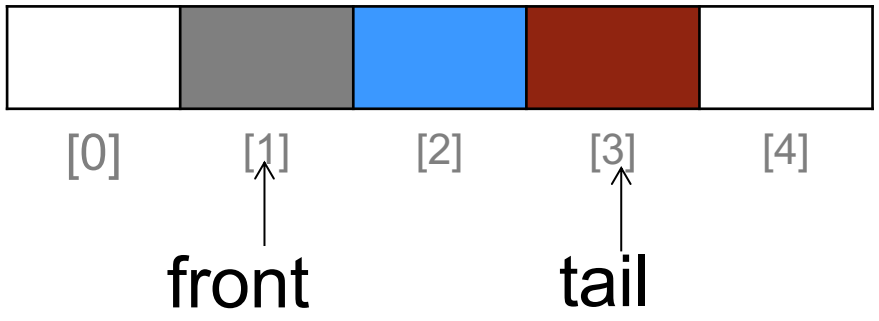


Linked list implementation

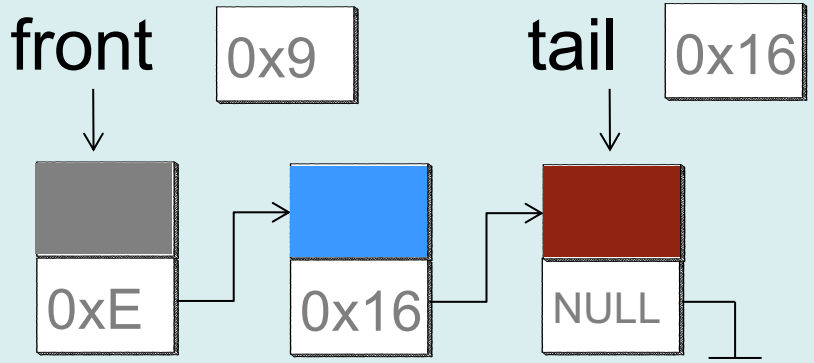


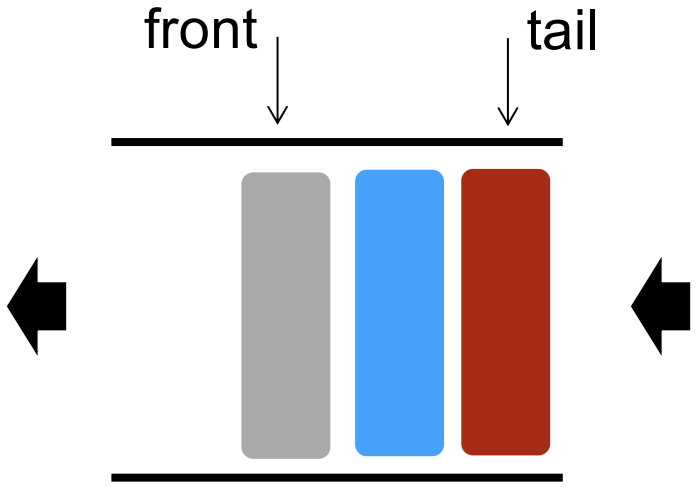


Array implementation

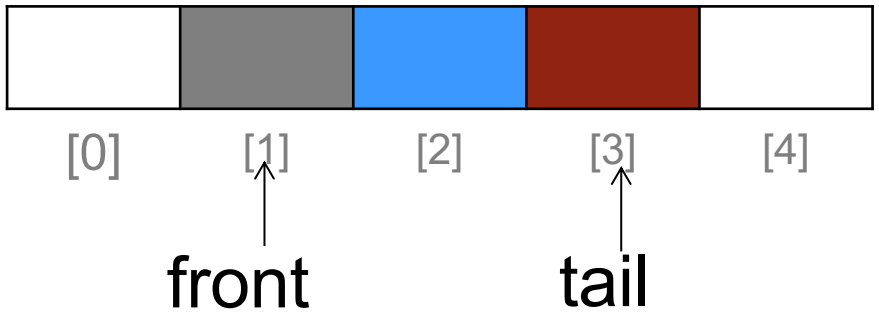


Linked list implementation

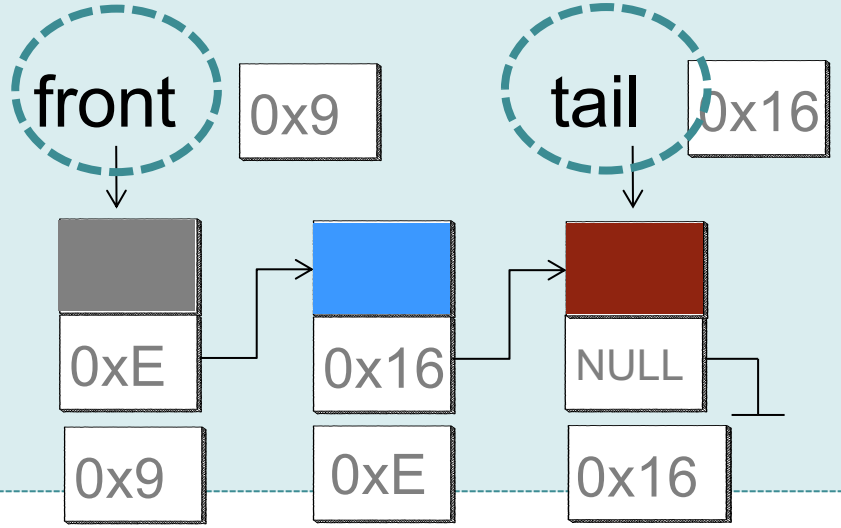


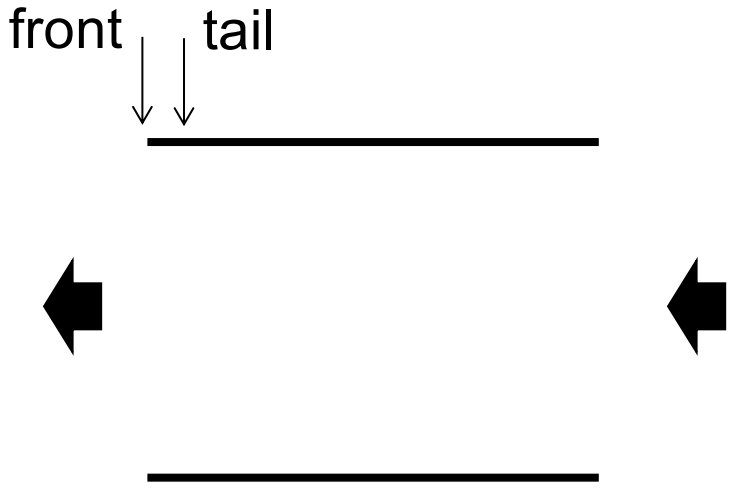


Array implementation

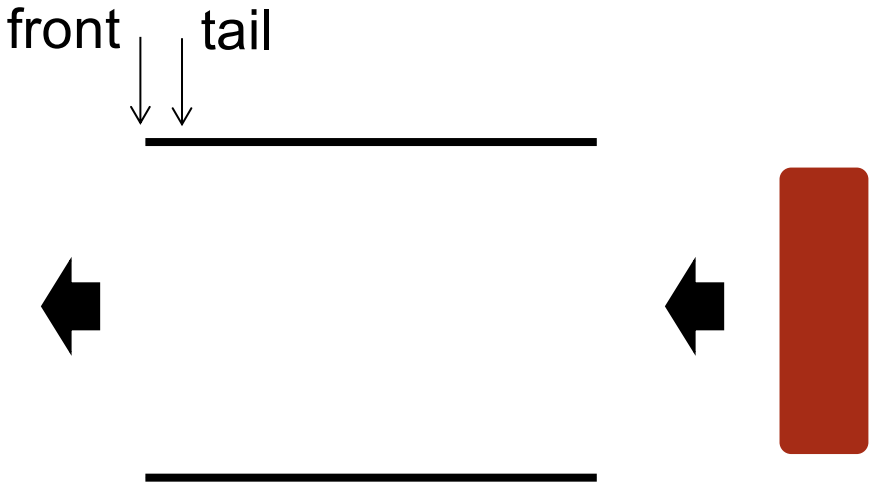


Linked list implementation

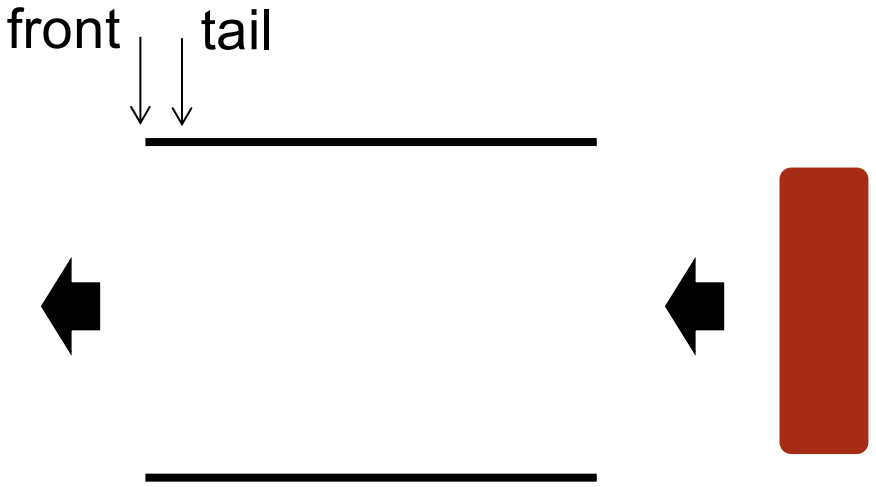




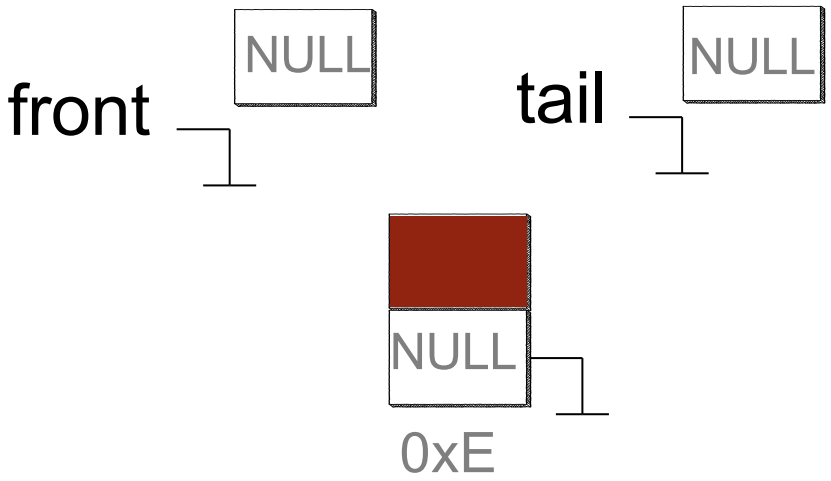
List implementation

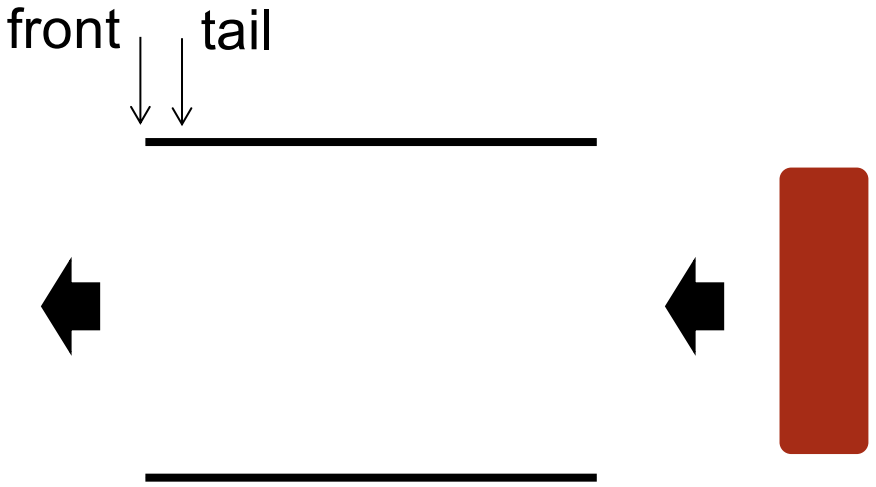


List implementation

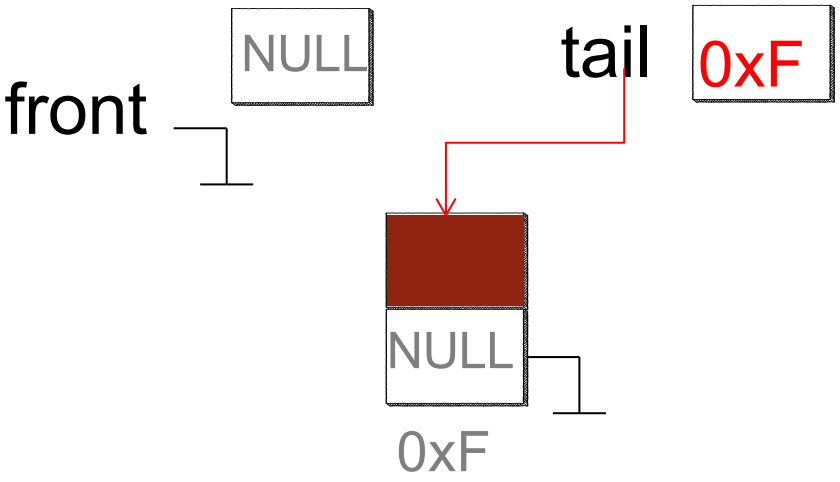


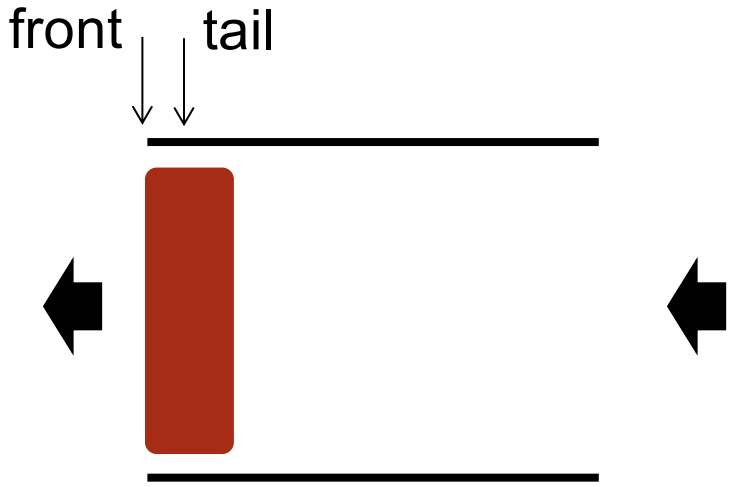
List implementation



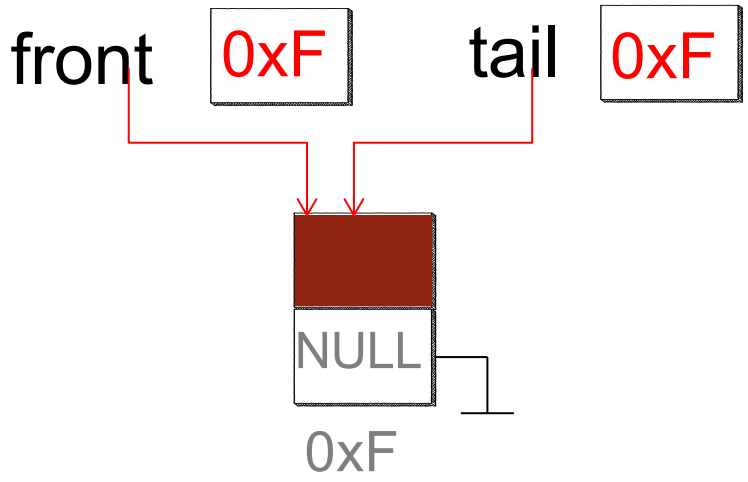


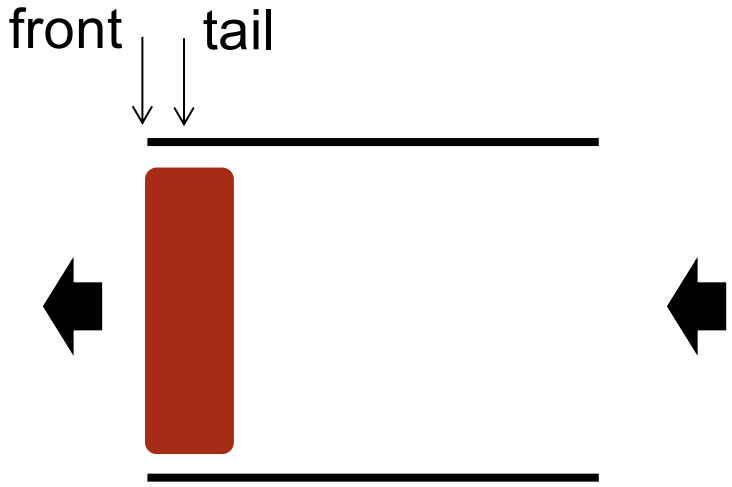
List implementation



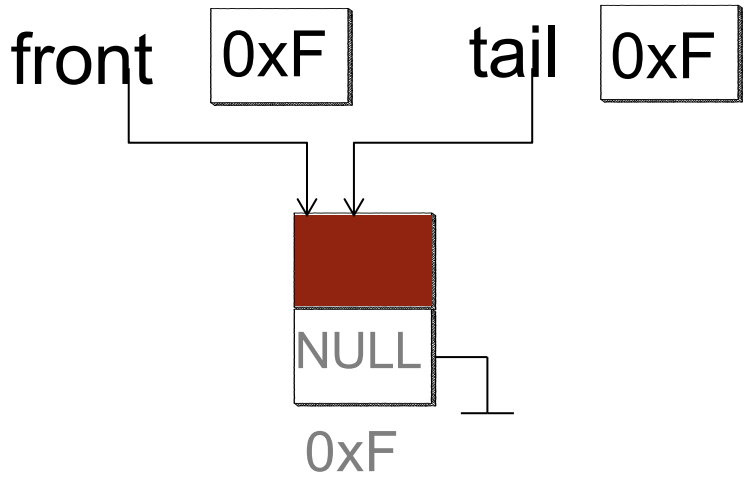


List implementation

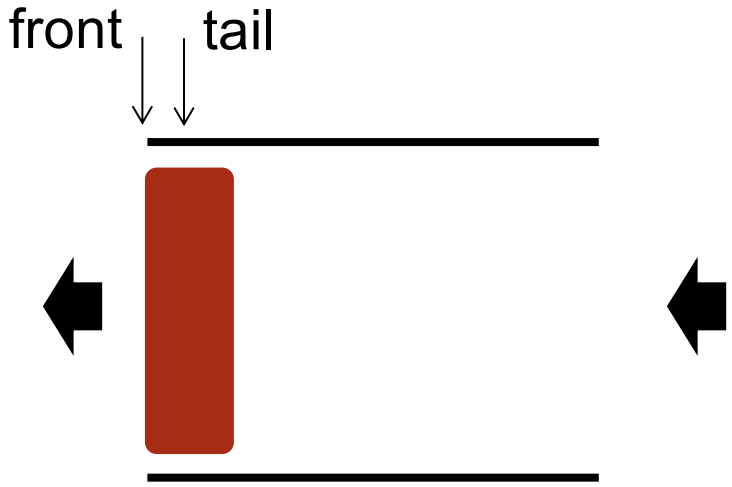




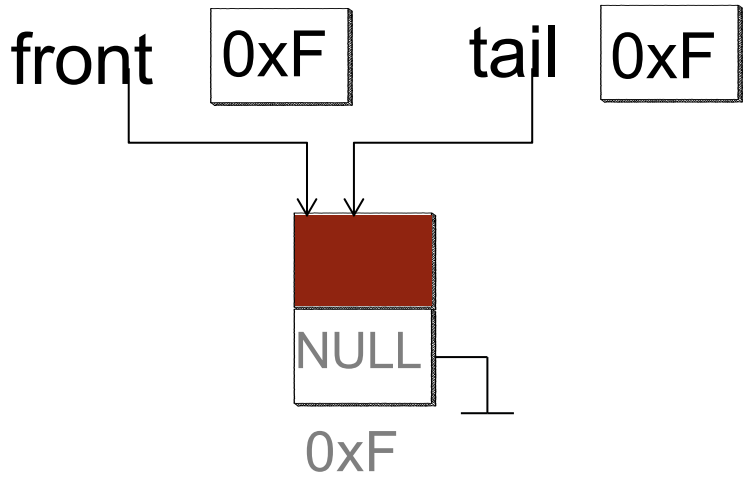
List implementation



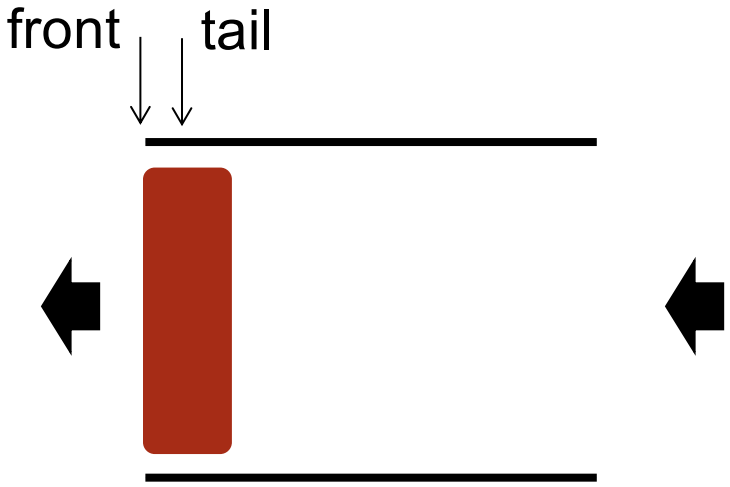

```
function ENQUEUE(x)
    newNode=new Node(x)
    if (front==NULL and tail==NULL)
      front=newNode
      tail=newNode
    else
      …
```



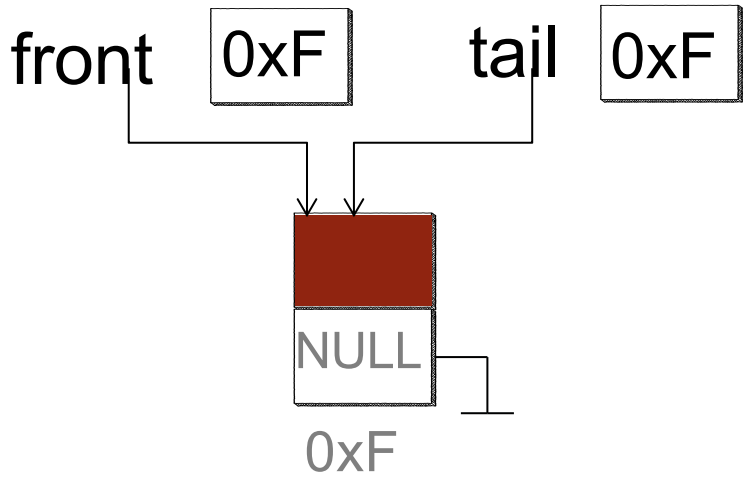
List implementation



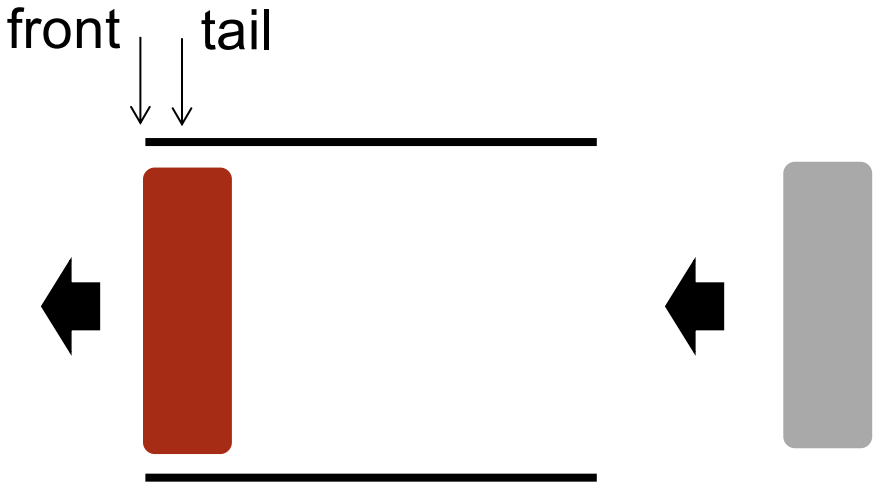

```
function ENQUEUE(x)
    newNode=new Node(x)
    if (front==NULL and tail==NULL)
      front=newNode
      tail=newNode
    else
      …
```



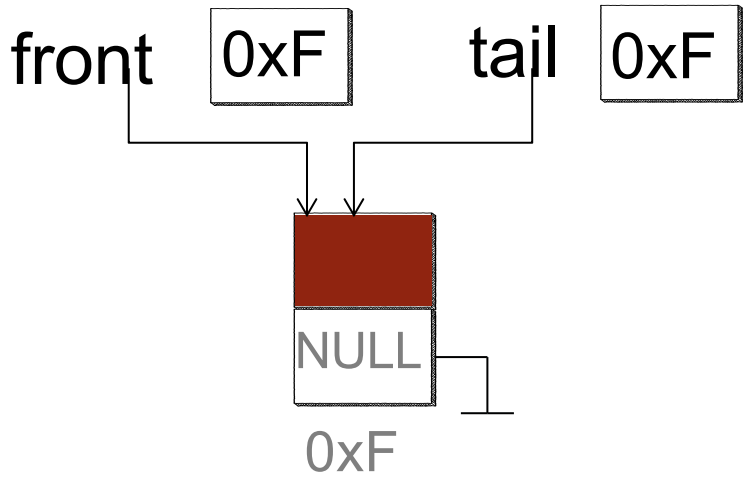
List implementation



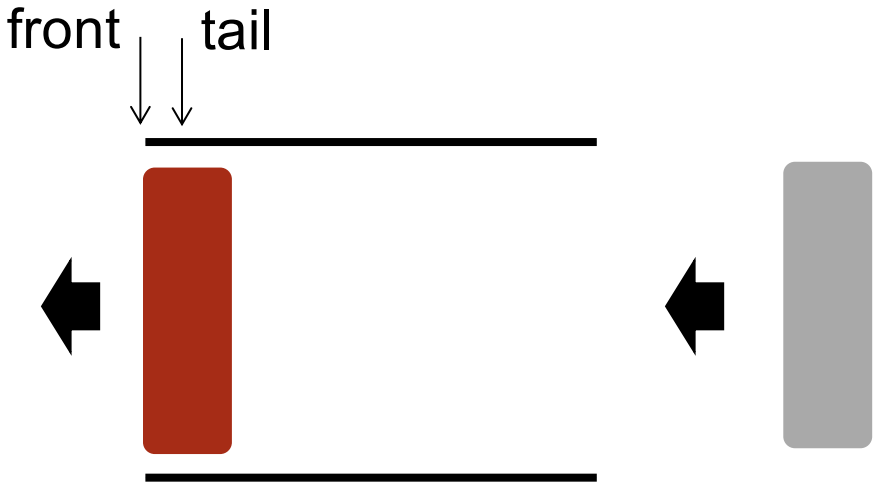

```
function ENQUEUE(x)
    newNode=new Node(x)
    if (front==NULL and tail==NULL)
      front=newNode
      tail=newNode
    else
      …
```



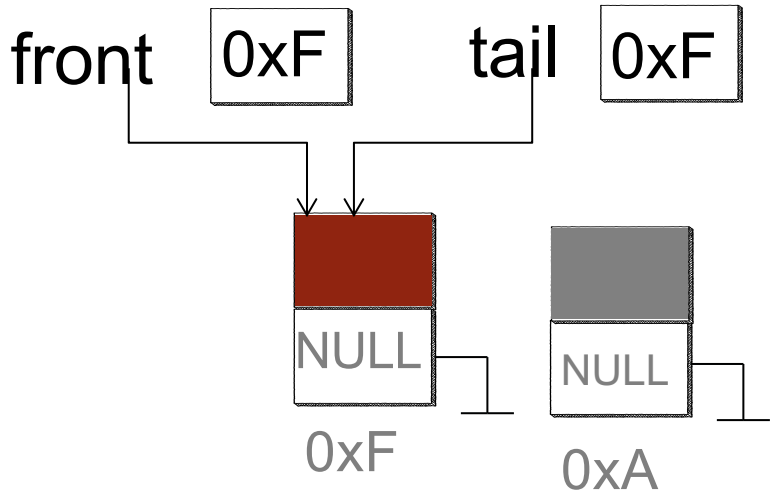
List implementation



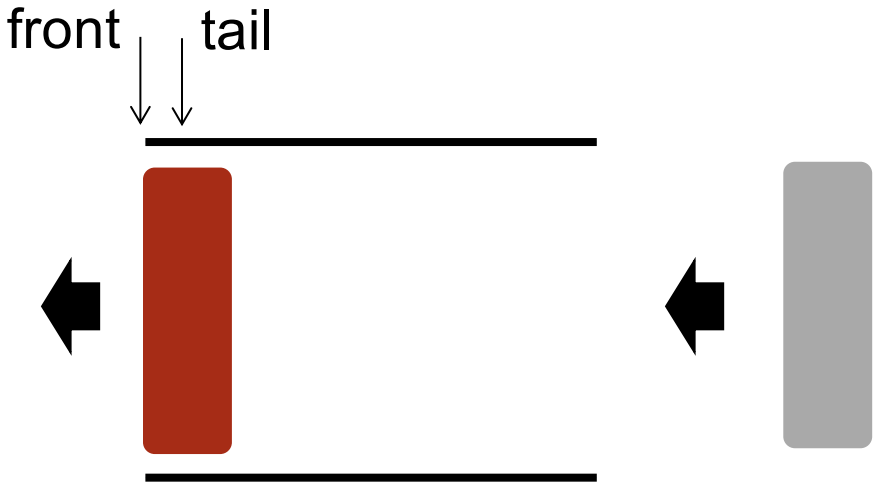

```
function ENQUEUE(x)
    newNode=new Node(x)
    if (front==NULL and tail==NULL)
      front=newNode
      tail=newNode
    else
      …
```



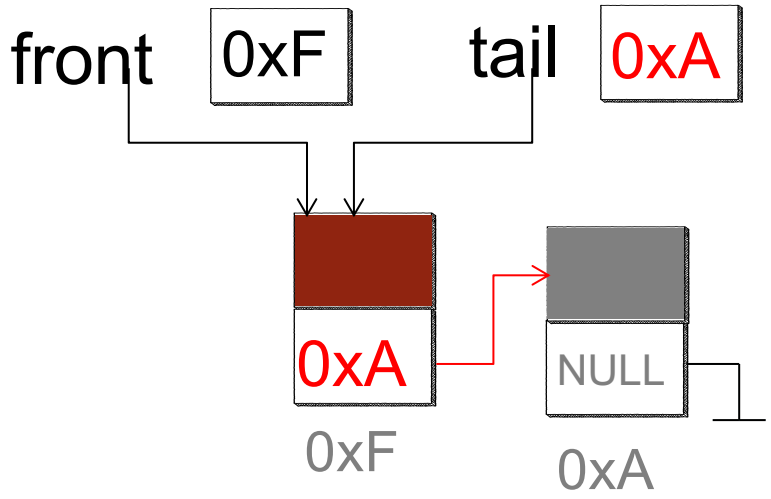
List implementation



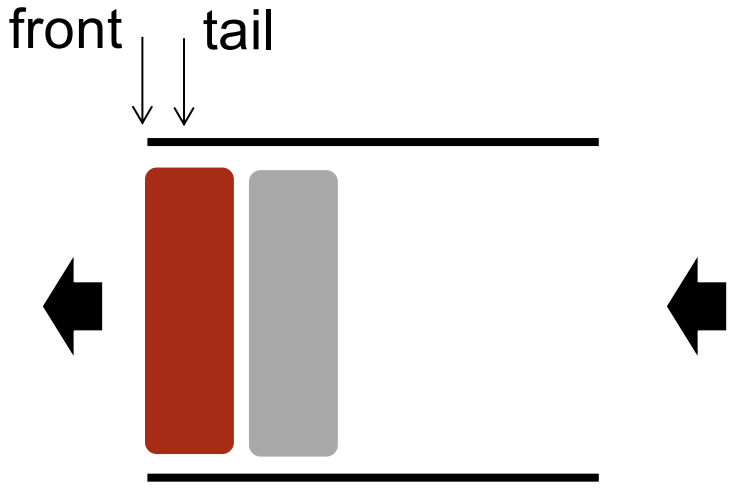

```
function ENQUEUE(x)
    newNode=new Node(x)
    if (front==NULL and tail==NULL)
      front=newNode
      tail=newNode
    else
      …
```



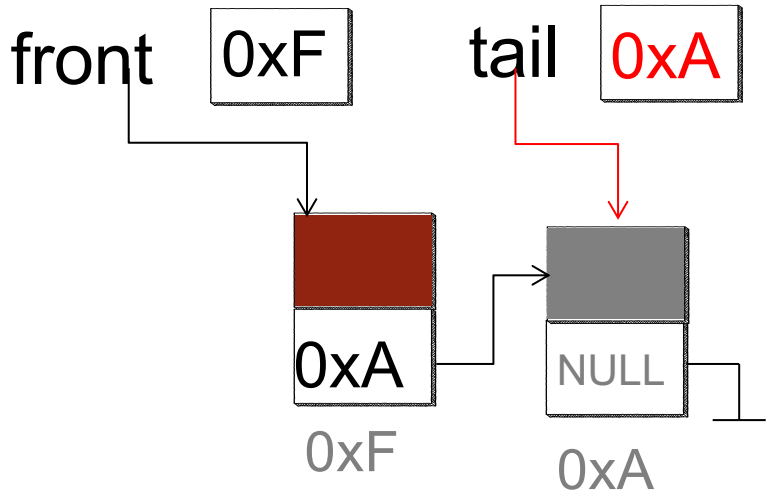
List implementation



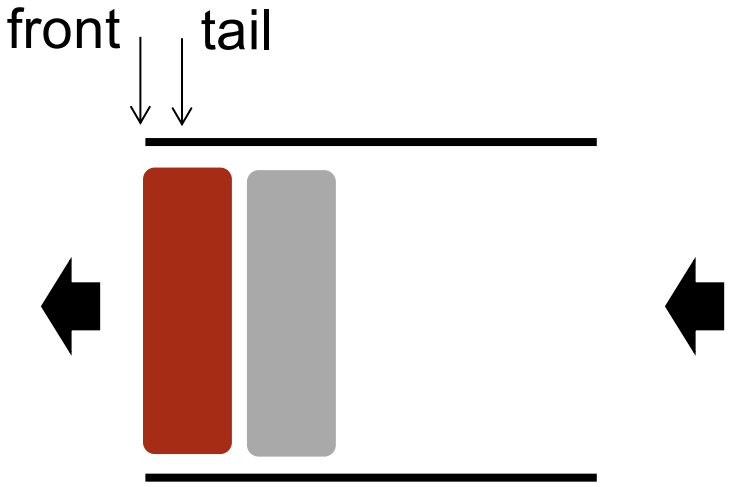

```
function ENQUEUE(x)
    newNode=new Node(x)
    if (front==NULL and tail==NULL)
      front=newNode
      tail=newNode
    else
      tail->next=newNode
```



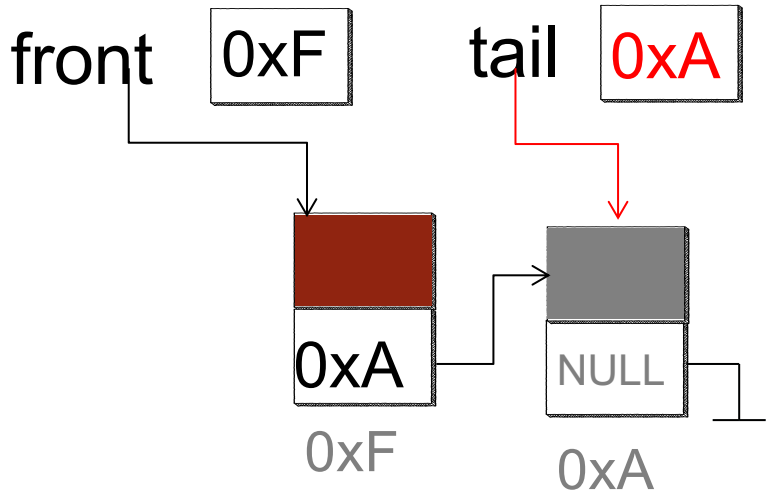
List implementation



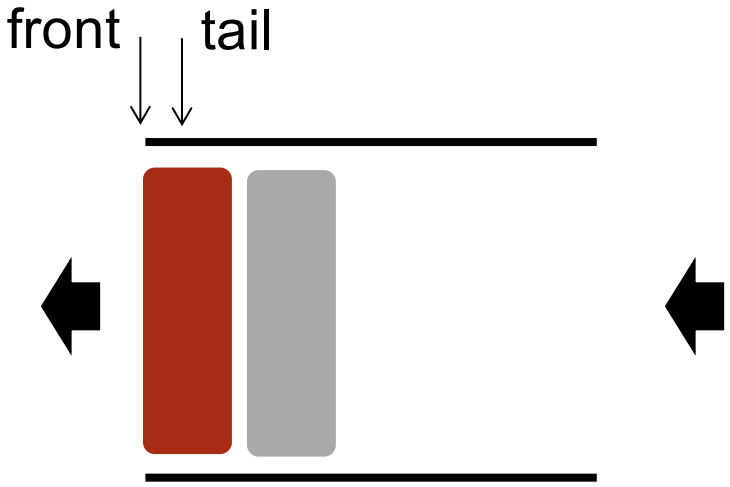

```
function ENQUEUE(x)
    newNode=new Node(x)
    if (front==NULL and tail==NULL)
      front=newNode
      tail=newNode
    else
      tail->next=newNode
      tail=newNode
```



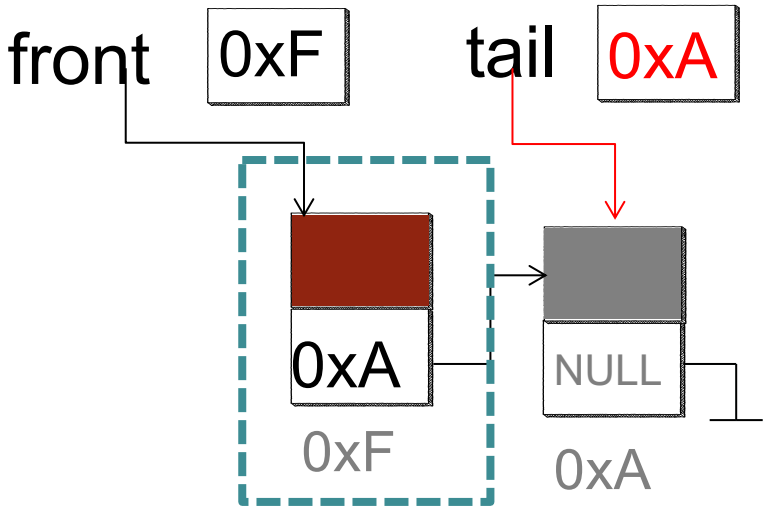
List implementation



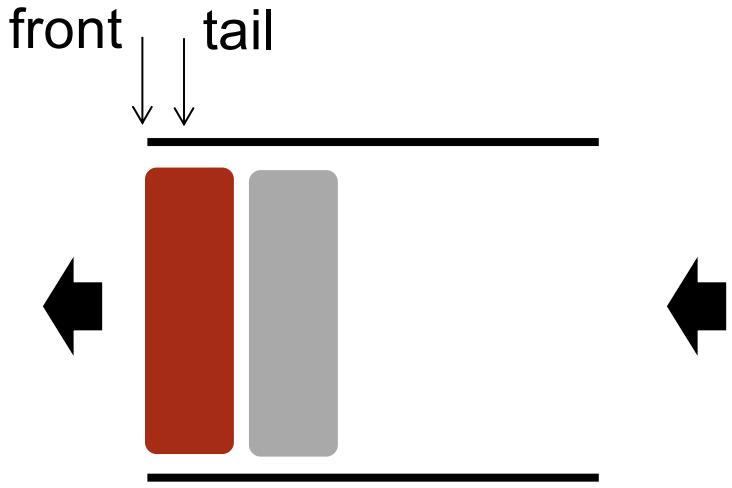

```
function DEQUEUE()
    if(front==NULL and tail==NULL)
      print(“empty queue”)
      return
    else
    …
```



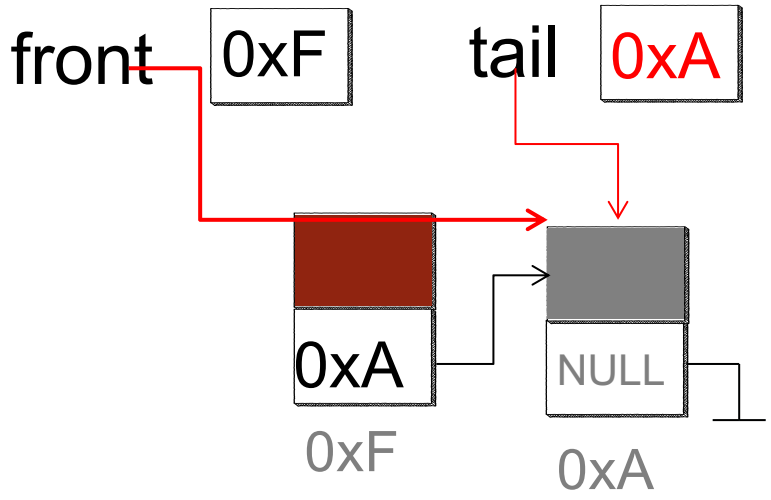
List implementation



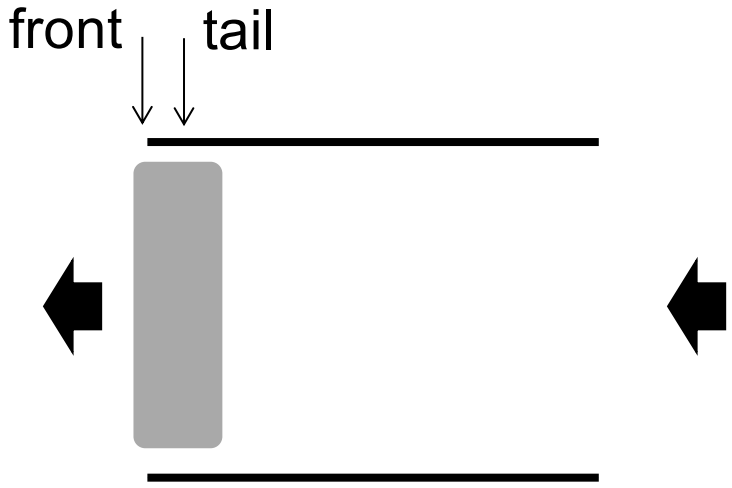

```
function DEQUEUE()
    if(front==NULL and tail==NULL)
      print("empty queue")
      return
    else
    …
```



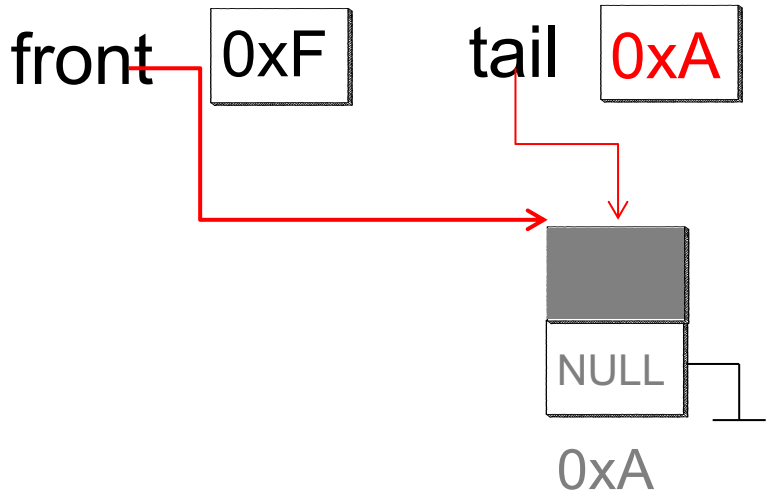
List implementation



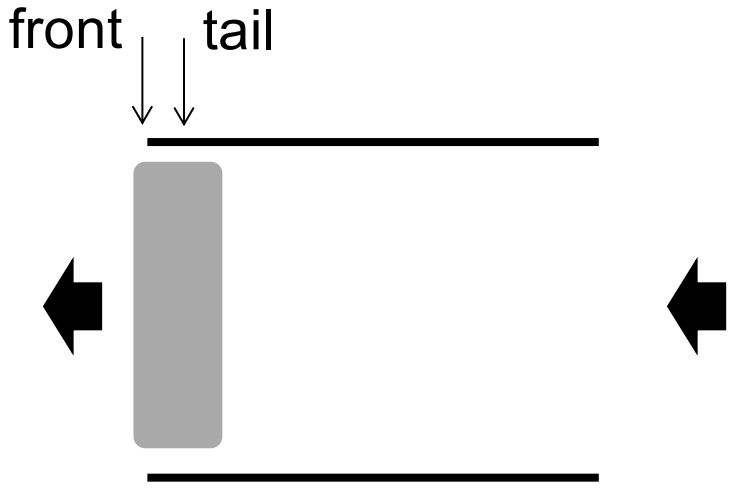

```
function DEQUEUE()
    if(front==NULL and tail==NULL)
      print("empty queue")
      return
    else
      front=front->next
```



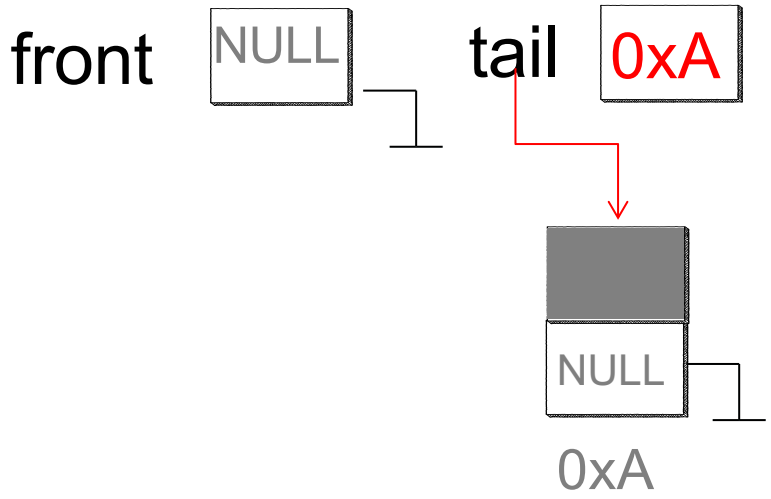
List implementation



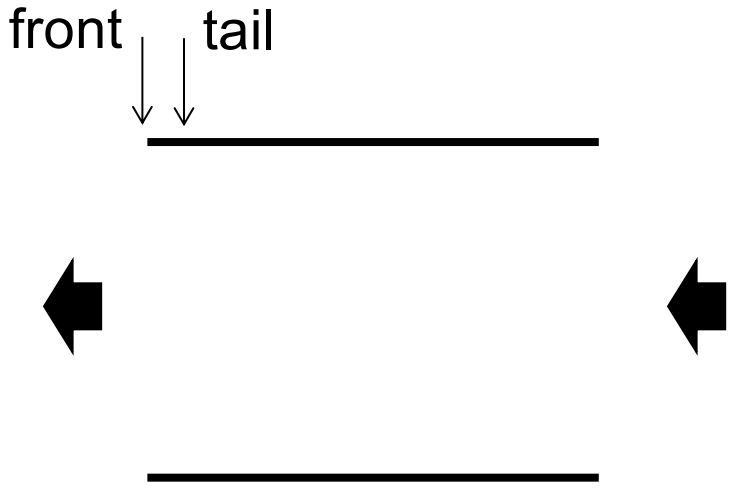

```
function DEQUEUE()
    if(front==NULL and tail==NULL)
      print("empty queue")
      return
    else
      front=front->next
```



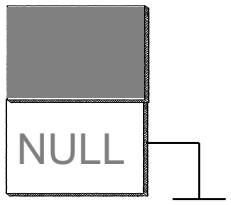
List implementation



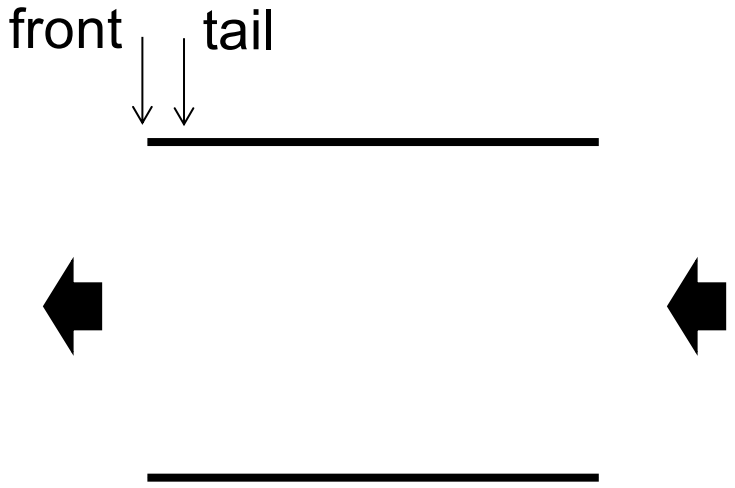

```
function DEQUEUE()
    if(front==NULL and tail==NULL)
      print(“empty queue”)
      return
    else
      front=front->next
```



List implementation

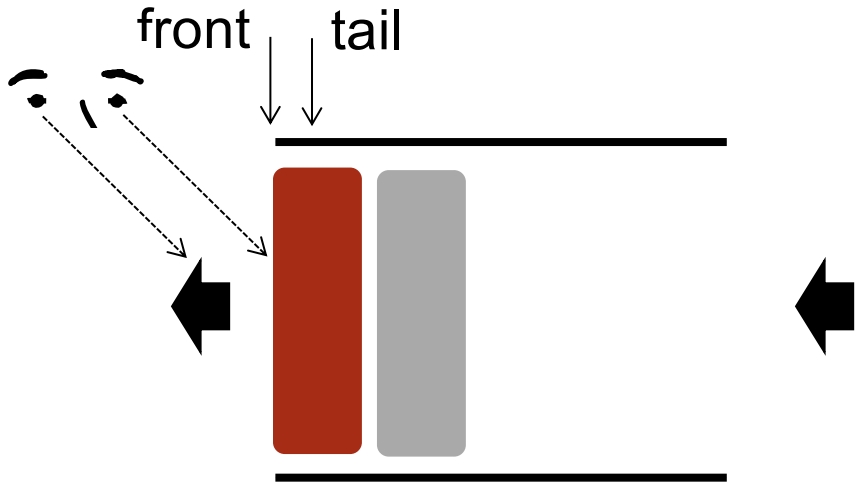



```
function DEQUEUE()
    if(front==NULL and tail==NULL)
      print("empty queue")
      return
    if(front==tail)
       front=NULL
       tail=NULL
    else
      front=front->next
```

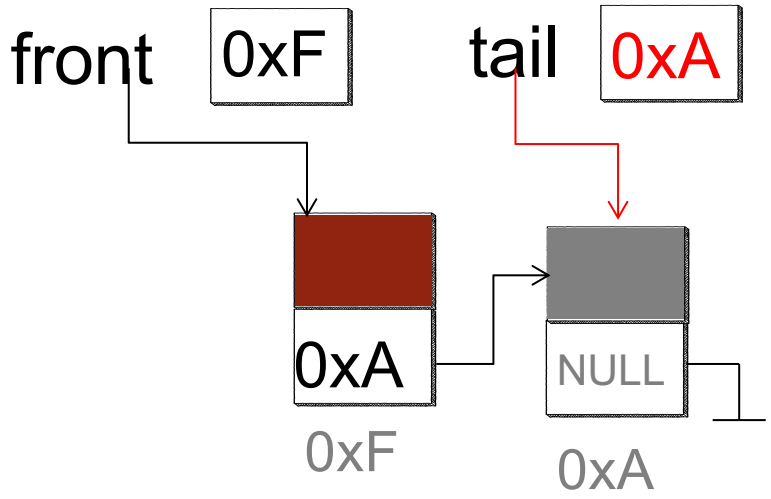


List implementation

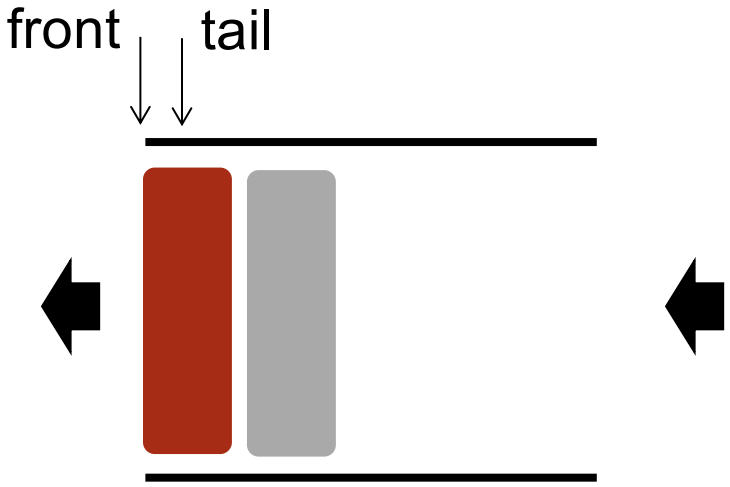

```
function DEQUEUE()
    if(front==NULL and tail==NULL)
      print(“empty queue”)
      return
    if(front==tail)
       front=NULL
       tail=NULL
    else
      front=front->next
```



List implementation

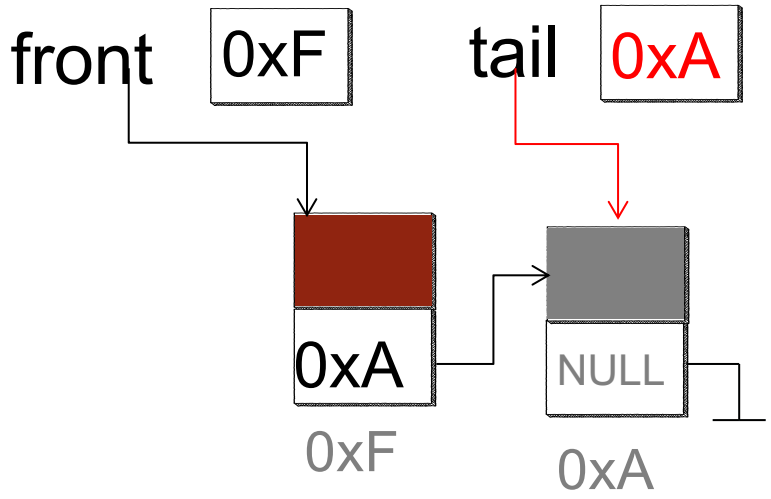



```
function PEEK()
    if(front==NULL and tail==NULL)
      print("empty queue")
      return
    else
      return front->data
```




```
function ISEMPTY()
    if(front==NULL and tail==NULL)
        return TRUE
    else
      return FALSE
```

List implementation



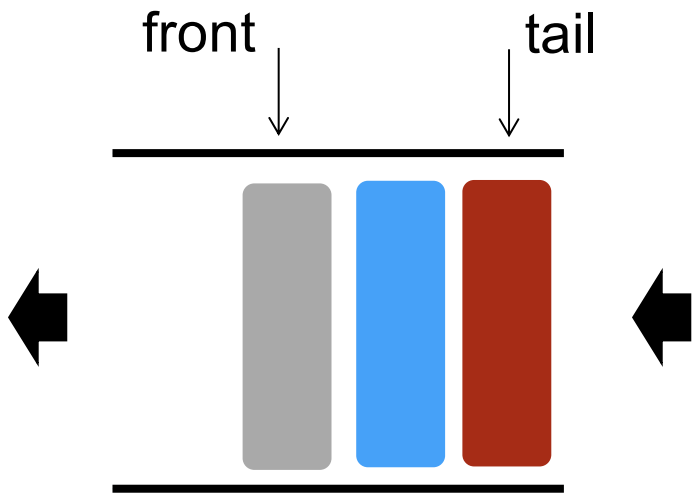
```
function ENQUEUE(x)
   newNode=new Node(x)
   if (front==NULL and tail==NULL)
     front=newNode
     tail=newNode
   else
     tail->next=newNode
     tail=newNode
```

```
function ISEMPTY()
   if(front==NULL and tail==NULL)
      return TRUE
   else
     return FALSE
```

```
function DEQUEUE()
   if(front==NULL and tail==NULL)
     print(“empty queue”)
     return
   if(front==tail)
      front=NULL
      tail=NULL
   else
     front=front->next
```

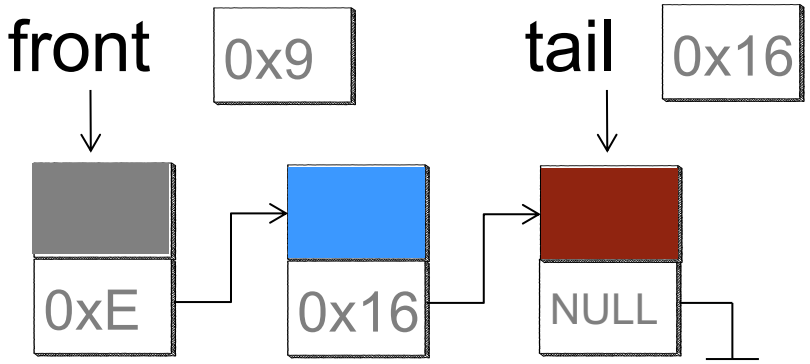
```
function PEEK()
   if(front==NULL and tail==NULL)
     print(“empty queue”)
     return
   else
     return front->data
```

Complexity: Theta(1)



Summary:
Implementation of queue using lists
Pseudocode
Theta(1)

Linked list implementation



Stacks & Queues

Definition of stacks and queues
Implementation of stacks using an array
Implementation of stacks using a linked list
Implementation of queues using an array
Implementation of queues using a linked list
Complexity theta(1)