BM267 - Introduction to Data Structures

4. Abstract Data Types

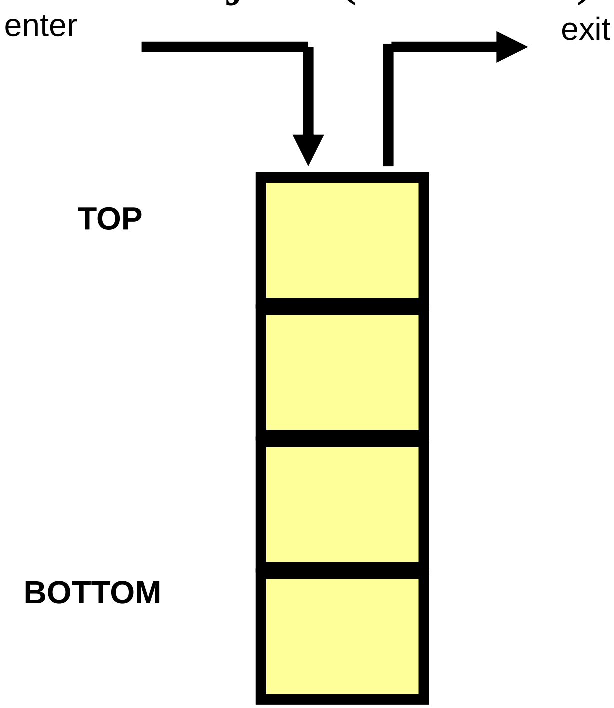
Ankara University

Computer Engineering Department

Bulent Tugrul

Pushdown Stack ADT

- A container of objects that are inserted and removed according to the last-in-first-out (**LIFO**) principle.
- Objects are inserted into a stack in at any time, but only the most recently inserted object (last one!) can be removed at any time.



Pushdown stacks in action

Web browser:

- Stores the addresses of recently visited sites on a stack.
- Each time a user visits a new site, the address of the site is **push**ed into the stack of addresses.
- Using the 'back' button the user can **pop** back to previously visited sites!

Text editors:

- Powerful text editors keep text changes in a stack.
- The user can use the undo mechanism to cancel recent editing operations

Pushdown stack operations

The fundamental operations involved in a stack are "**push**" and "**pop**".

- **push**: adds a new element on **top** of the stack
- **pop**: removes an element from the **top** of the stack

It is an error to pop an element from an **empty** stack.
It is also an error to push an elemet to a **full stack**.

Other operations

- **isEmpty**: Checks if the stack is empty
- **isFull**: checks if the stack is full (if there is an implementation dependent limit)

Pushdown stack ADT implementation

A stack can be implemented with an array of objects easily.

- The maximum size of the stack (array) must be estimated when the array is declared.
- Space is wasted if we use less elements.
- We cannot "push" more elements than the array can hold (overflow).

If the maximum stack size cannot be be estimated, use a linked list to implement the stack.

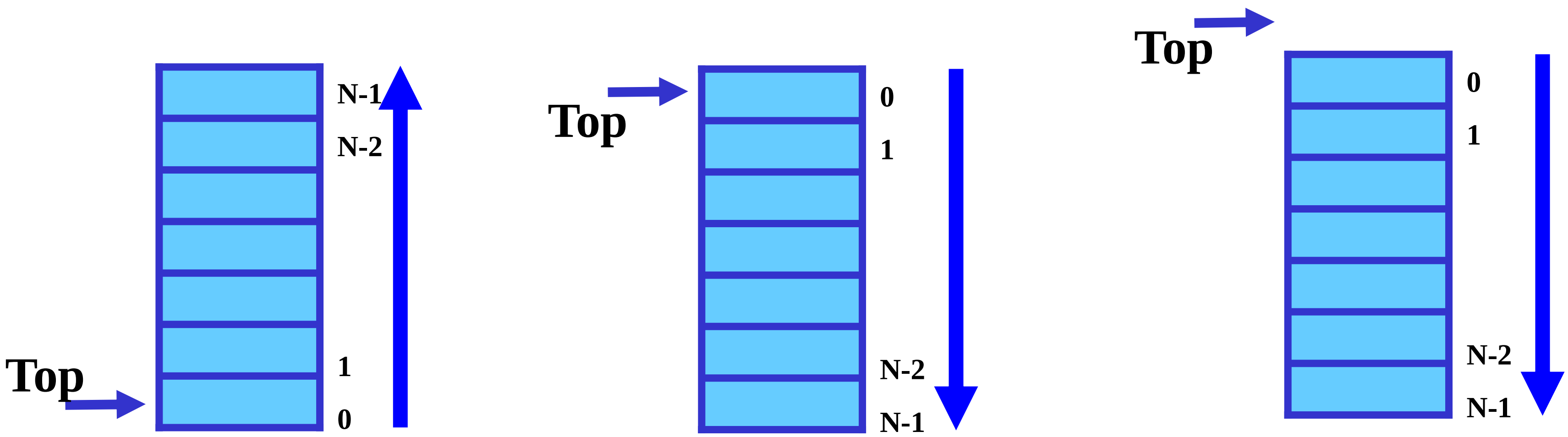
Pushdown stack ADT implementation

Interface for pushdown stack ADT for objects of type ‘Item’:

```
void STACKinit(int);
int  STACKempty();
void STACKpush(Item)
Item STACKpop();
```

Pushdown stack ADT implementation

Note that, if an array is used, you can visualize (and implement) stack in several ways.



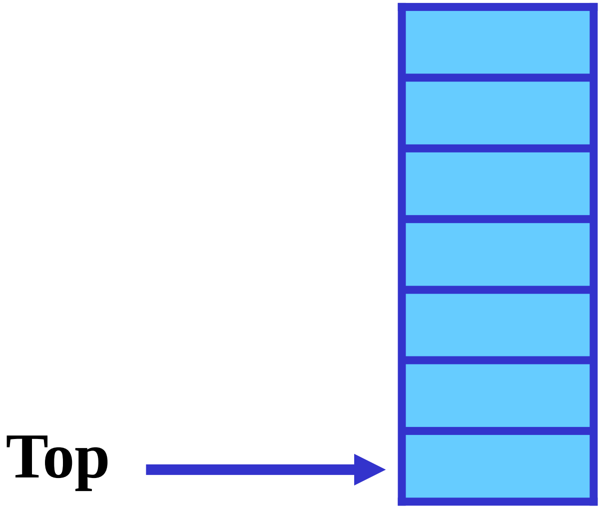
Top points to next available element

Top points to last inserted element

Pushdown stack ADT implementation

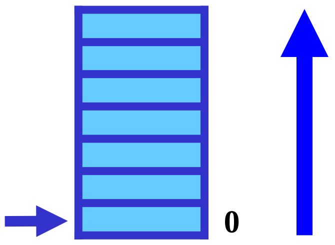
```
void STACKinit(int);
```

- Initializes an array of **Items** of specified size.
- Item (structure) is known by both the client and the implementation
- Must have a pointer (or array index) that points the next available (or last filled) slot on the stack.



Pushdown stack ADT implementation

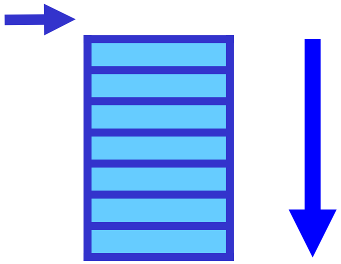
Convention:



- Need pointer initialized to (index) 0, since we adopted the convention that **top** refers to the next free place in the stack, where a push will be written.
- The last item pushed onto the stack is therefore at **top-1**.
- Delete item: We have to decrease **top** by 1 before we pop an item from the stack.
- Insert item: We have to push the item first then increment **top**.

Pushdown stack ADT implementation

Convention:



- Need pointer initialized to (index) -1, since we adopted the convention that **top** refers to the last item inserted in the stack
- The last item pushed onto the stack is therefore at **top**.
- Delete item: We have to remove the item first, then decrease **top** by 1.
- Insert item: We have to increment **top before** pushing the item.

Pushdown stack ADT implementation

`void push(Item);` Insert new item at the top of the stack.

Precondition: The stack is not full.

Postcondition: The stack has a new item at the top.

`Item pop( );` Remove the item from the top of the stack.

Precondition: The stack is not empty

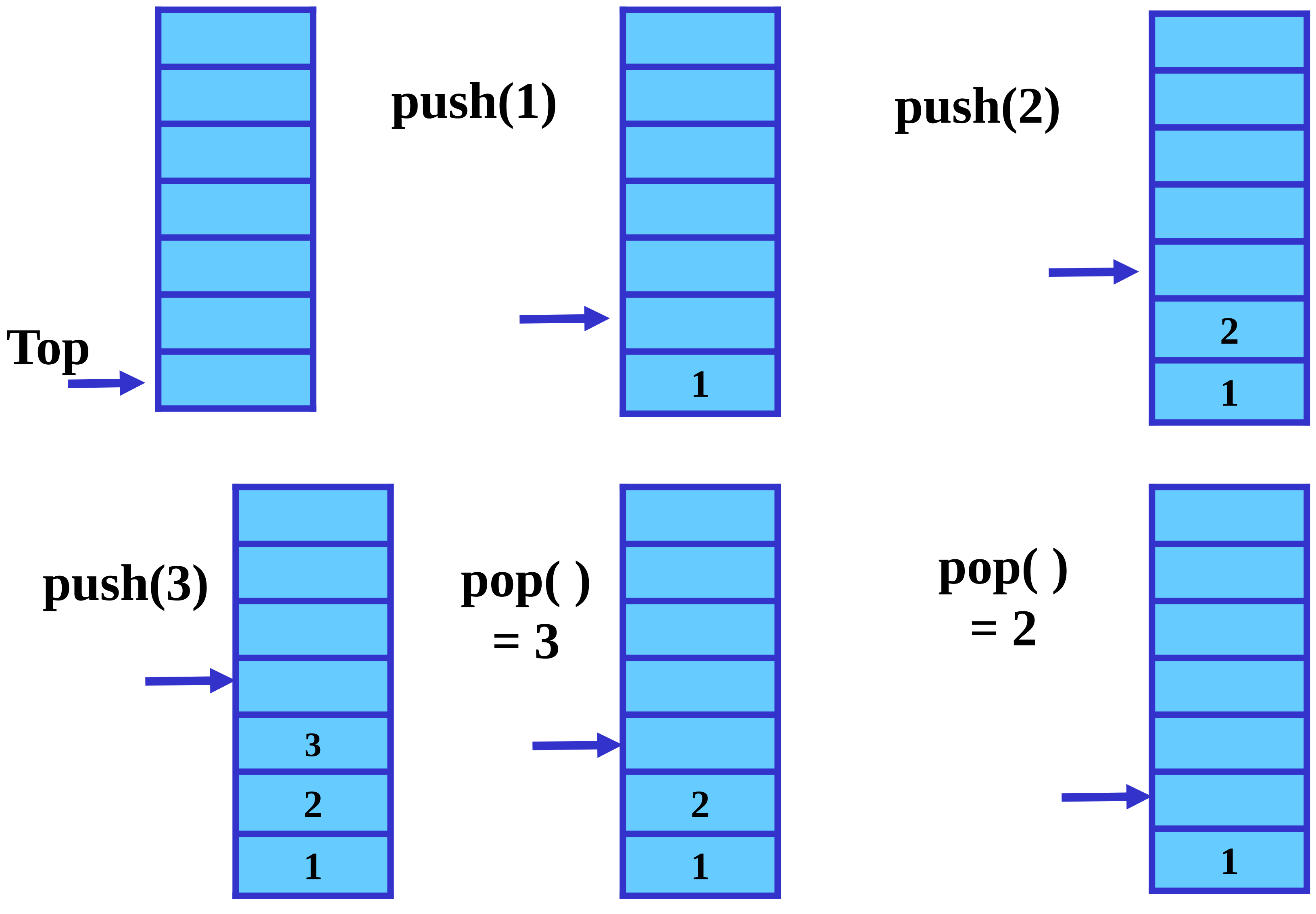
Postcondition: Either the stack is empty or the stack has a new topmost item from a previous push.

`int STACKempty();` Returns a logical value depending on the number of elements in the stack.

Precondition: The stack has $0 \le N \le Max$ elements

Postcondition: The stack has N elements.

Pushdown stack ADT implementation



Example: Using a Stack to compute a Hex Number

431 / 16 = 26	26 / 16 = 1	1 / 16 = 0
Rem: 15	Rem: 10	Rem: 1
Push (Hex 15)	Push (Hex 10)	Push (Hex 1)

Stack: Empty

'F'

'A'
'F'

'1'
'A'
'F'

pop( ) = '1'

String = 1

'1'

'A'
'F'

pop( ) = 'A'

String = 1A

'A'

'F'

pop( ) = '1'

String = 1AF

'F'

Stack: Empty

Example: Array Implementation of a Stack

```
int *s;
int Top;
void STACKinit(int maxN){
      s = (int *) malloc( maxN * sizeof(int) );
      Top = 0; }
int STACKempty(){
      return Top == 0; }
void STACKpush(int item){
      s[Top++] = item; }
int STACKpop(){
      return s[--Top]; }
```

Example: Postfix Calculation

- Suppose that we need to find the value of a simple arithmetic expression involving multiplication and addition of integers, such as

```
5 * ( ( ( 9 + 8) * ( 4 * 6 ) ) + 7)
```

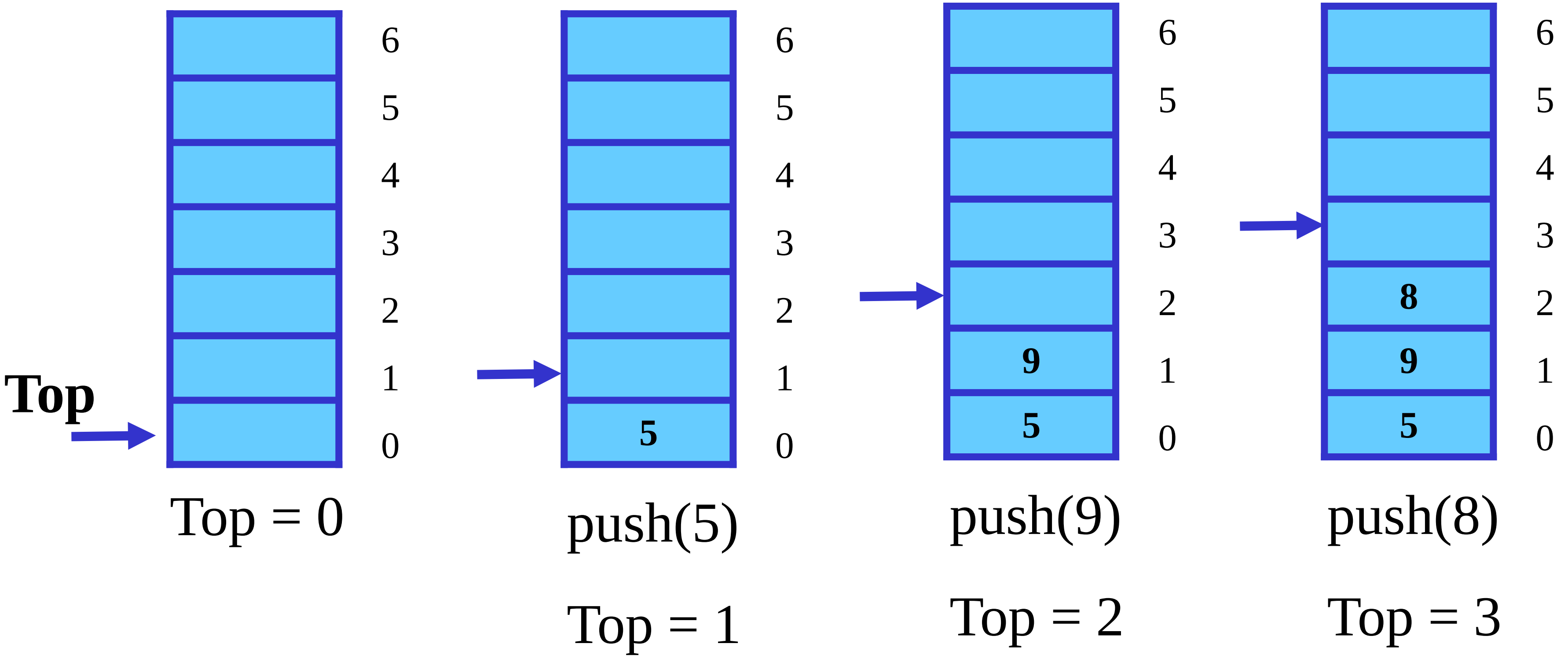
- The calculation saves intermediate results: For example, if we calculate (9 + 8), then we have to save the result.
- A pushdown stack is the ideal mechanism for saving intermediate results in a such calculation.
- We can convert to arithmetic expression into postfix representation. In postfix representation each operator appears after its two operand.

```
( 5 + 9 ) ➔ 5 9 +
```

Example: Postfix Calculation

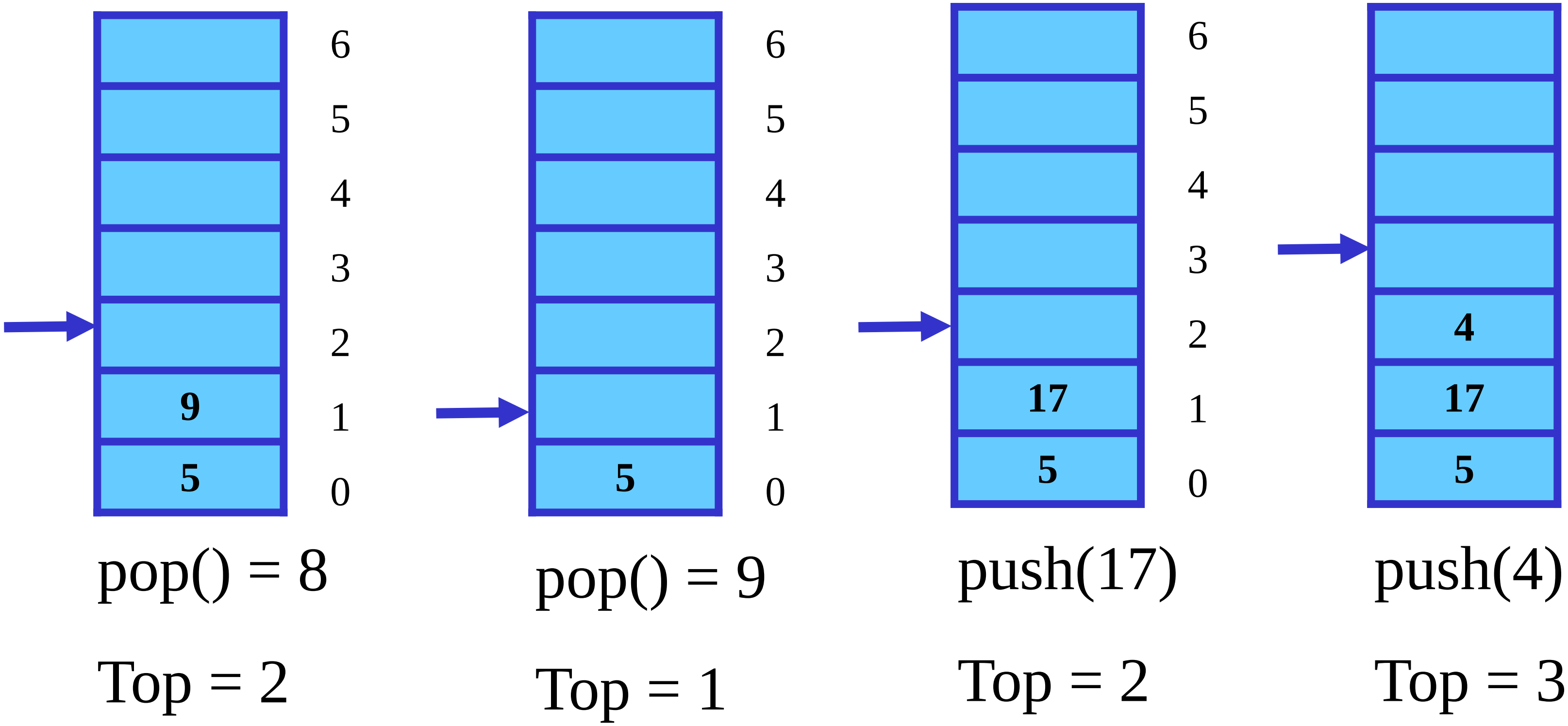
```
5 9 8 + 4 6 * * 7 + *
5 ( 9 + 8 ) 4 6 * * 7 + *
5 17 4 6 * * 7 + *
5 17 ( 4 * 6 ) * 7 + *
5 17 24 * 7 + *
5 ( 17 * 24 ) 7 + *
5 408 7 + *
5 ( 408 + 7 ) *
5 415 *
( 5 * 415 )
2075
```

Example: Postfix Calculation



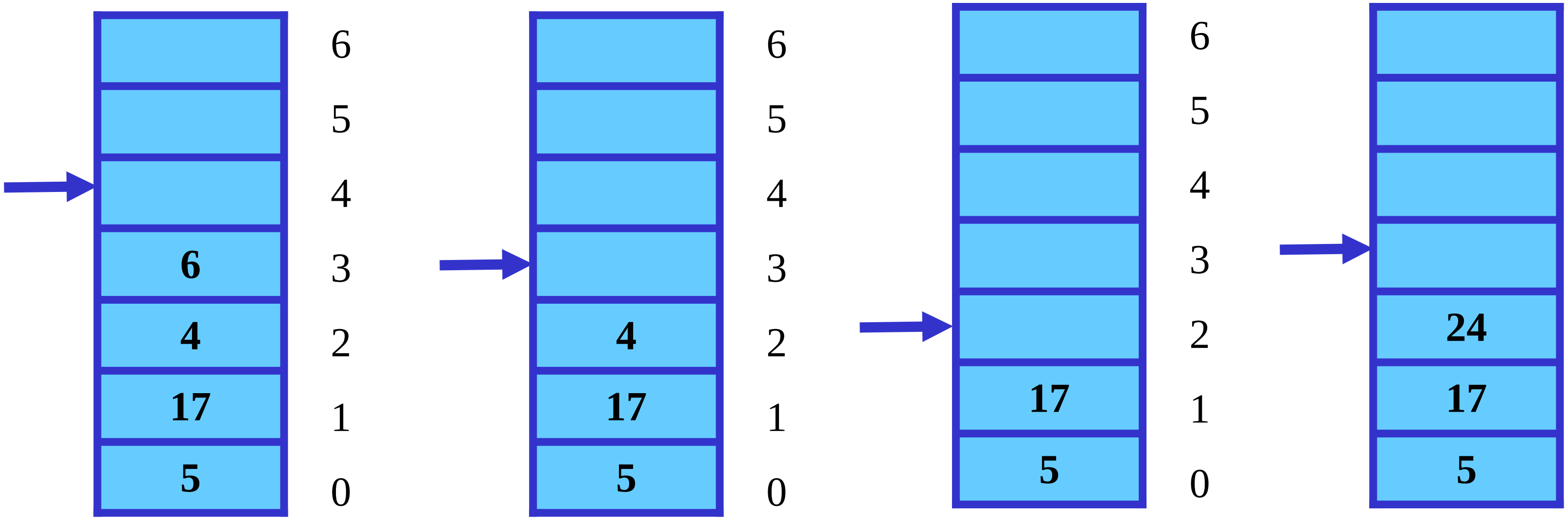
5 9 8 + 4 6 * * 7 + *

Example: Postfix Calculation



5 9 8 + 4 6 * * 7 + *

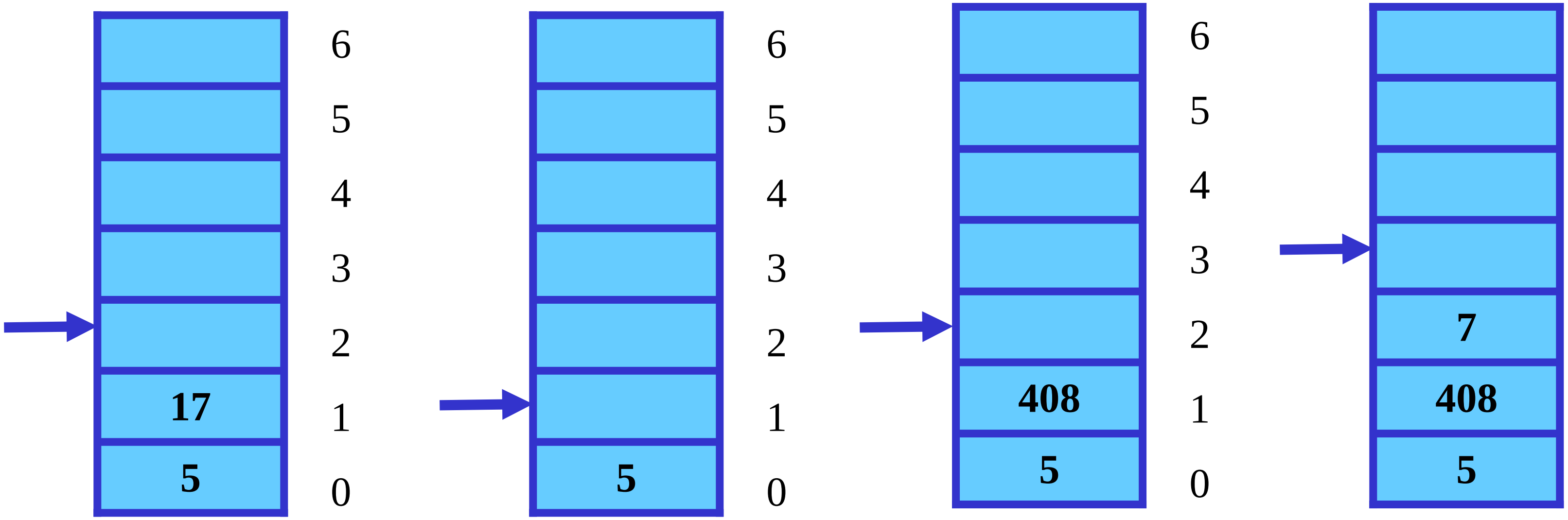
Example: Postfix Calculation



push(6)	pop() = 6	pop() = 4	push(24)
Top = 4	Top = 3	Top = 2	Top = 3

`5 9 8 + 4 6 * * 7 + *`

Example: Postfix Calculation

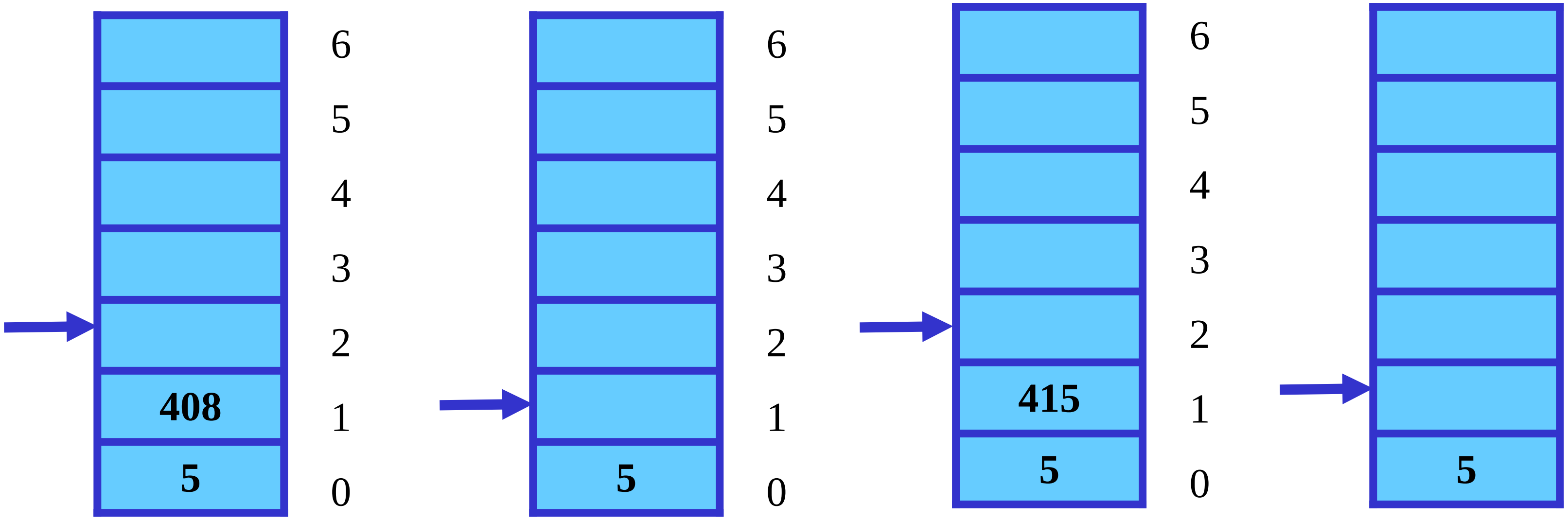


pop() = 24 | pop() = 17 | push(408) | push(7)

Top = 2 | Top = 1 | Top = 2 | Top = 3

5 9 8 + 4 6 * * 7 + *

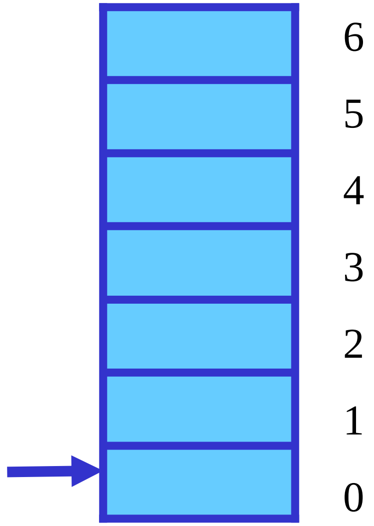
Example: Postfix Calculation



pop() = 7	pop() = 408	push(415)	pop()=415
Top = 2	Top = 1	Top = 2	Top = 1

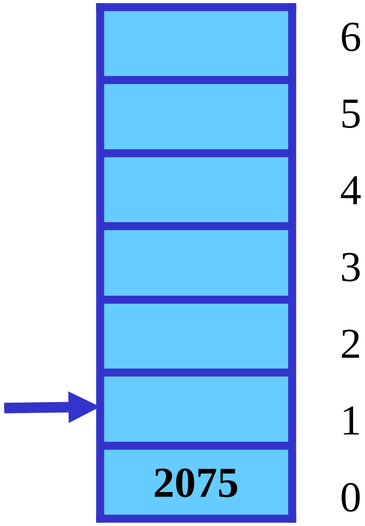
```
5 9 8 + 4 6 * * 7 + *
```

Example: Postfix Calculation



pop() = 5

Top = 0



push(2075)

Top = 1

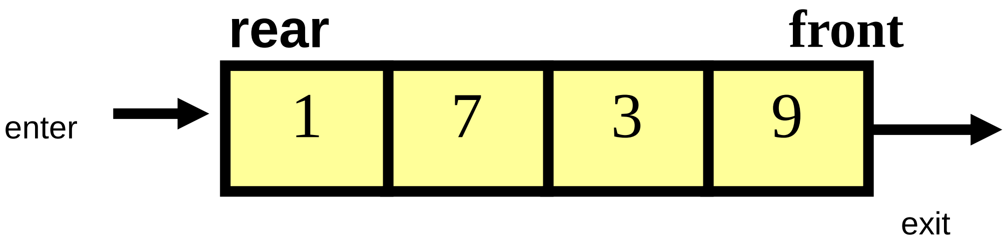
```
5 9 8 + 4 6 * * 7 + *
```

Example: Array Implementation of a Stack

```
/* Postfix-expression evaluation */
int main(int argc, char *argv[]){
        char a[] = "5 11 * 5 + 2 *";
        int i, array_size = strlen(a);
    STACKinit( array_size );
    for (i = 0; i < array_size; i++) {
        if (a[i] == '+')
          STACKpush( STACKpop()+ STACKpop() );
        if (a[i] == '*')
          STACKpush( STACKpop() * STACKpop() );
        if ( (a[i] >= '0') && ( a[i] <= '9') )
          STACKpush(0);
        while ((a[i] >= '0') && (a[i] <= '9'))
          STACKpush(10*STACKpop() + (a[i++]-'0'));   }
    printf("%d \n", STACKpop());
        return 0;}
```

Queues

- A queue is a data structure that items can be inserted only at one end (called rear) and removed at the other end (called the front).
- The item at the front end of the queue is called the first item.



Queue operations

- **put**: adds a new element at the **rear** of the queue
- Increase the number of element in the queue by 1.
- **get**: removes an element from the **front** of the queue
- Decrease the number of elements in the queue by 1

It is an error to **get** an element from an **empty** queue.

It is also an error to **put** an element to a full queue.

Other operations

- **isEmpty**: Checks if the queue is empty
- **isFull**: checks if the queue is full (if there is an implementation dependent limit)

Queue implementation

A queue can be implemented with an array of objects easily.

- The maximum size of the queue (array) must be estimated when the array is declared.
- Space is wasted if we use less elements.
- We cannot "put" more elements than the array can hold (overflow).

If the maximum queue size cannot be be estimated, use a linked list to implement the queue.

Queue ADT implementation

Interface for queue ADT for objects of type ‘Item’:

```
void QUEUEinit(int);
int  QUEUEempty();
void QUEUEput(Item)
Item QUEUEget();
```

(Compare these operations with stack operations)

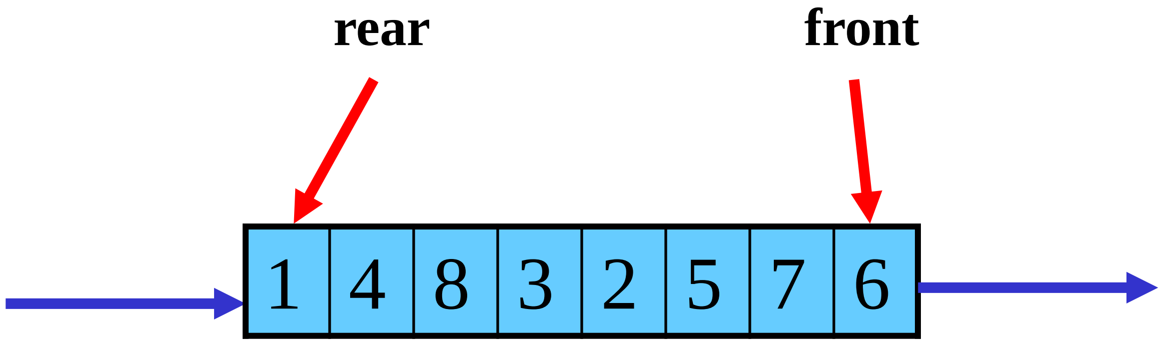
They are almost the same:

- push, put: insertion
- pop, get: removal

Queue ADT implementation

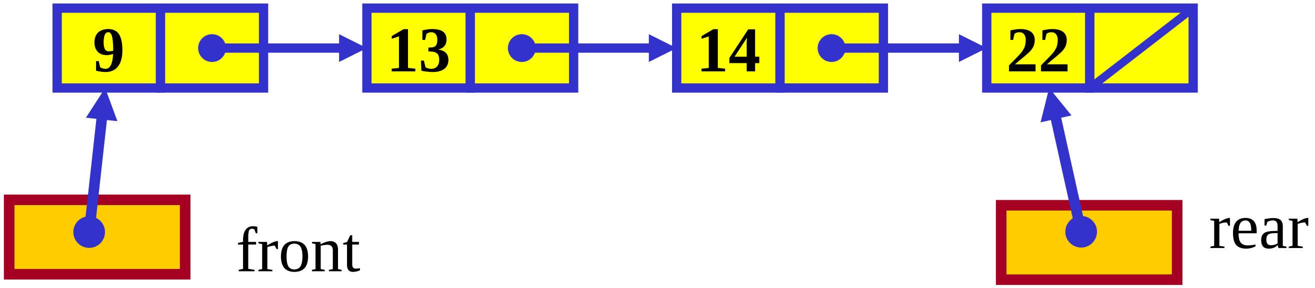
```
void QUEUEinit(int);
```

- Initializes an array of **Items** of specified size.
- **Item** (structure) is known by both the client and the implementation
- Must have two pointers (or array indices) that point to the rear and the front of the queue.

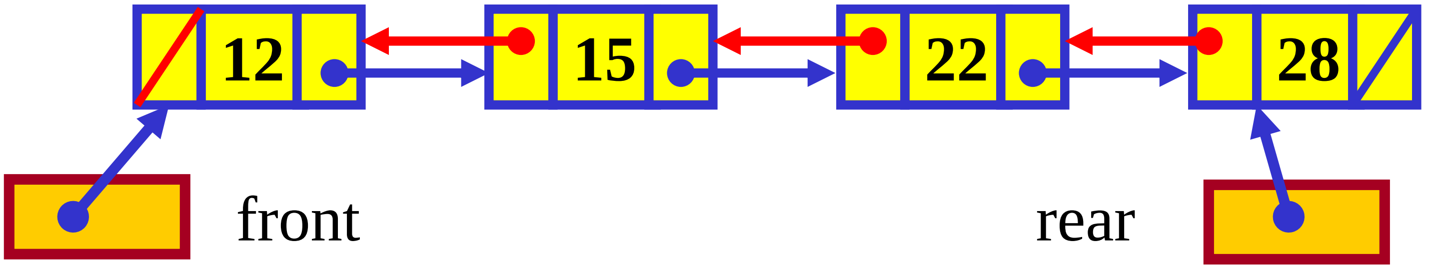


Queue Linked List implementation

- Elements can be added and removed in any order
- Therefore it is easier to use a singly-linked list as a queue, provided two extra pointers are kept.

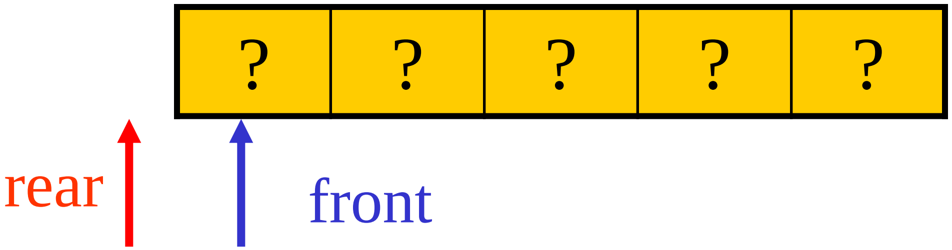


- Or better yet, use a doubly linked list, to maintain the head pointer easily.



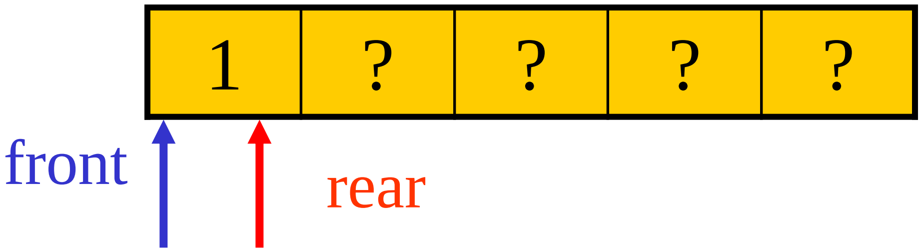
Queue Array implementation

First Approach(not efficient!!)



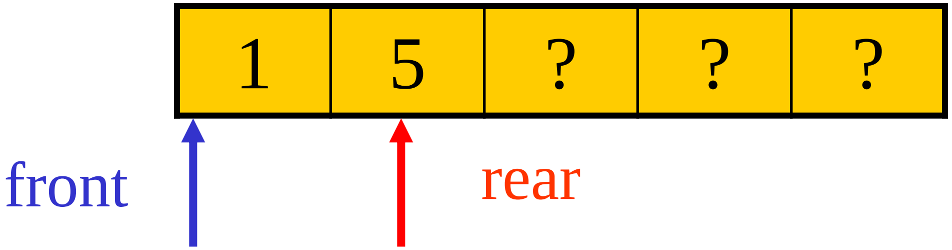
Initial state

front = 0 rear = -1



put(1)

front = 0 rear = 0



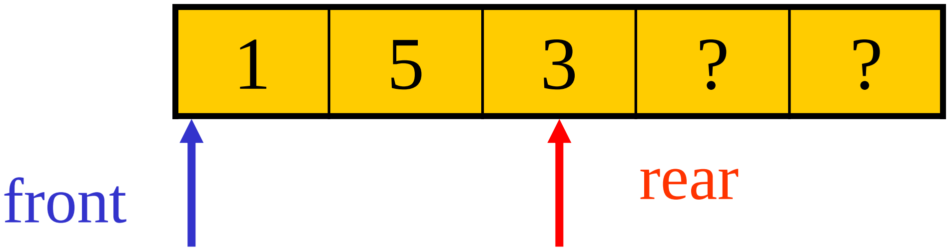
put(5)

front = 0 rear = 1

Queue Array implementation

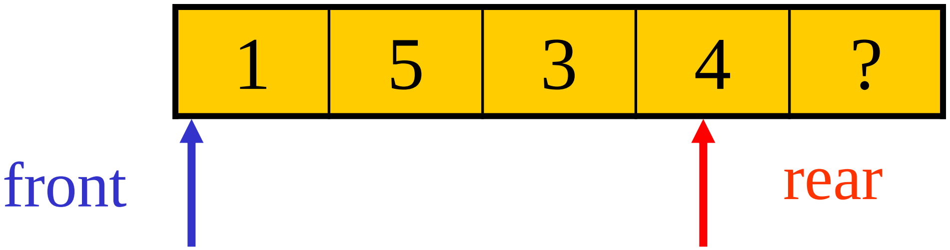
put(3)

front = 0 rear = 2



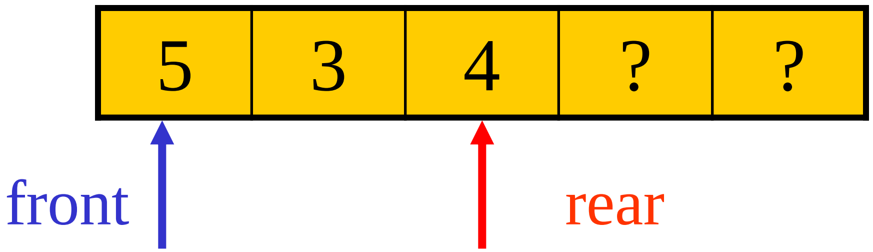
put(4)

front = 0 rear = 3

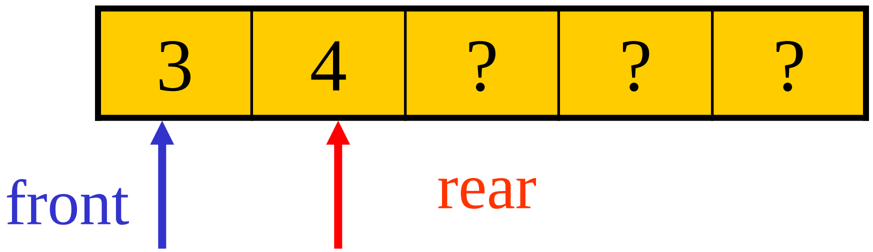


get( ) = 1

front = 0 rear = 2

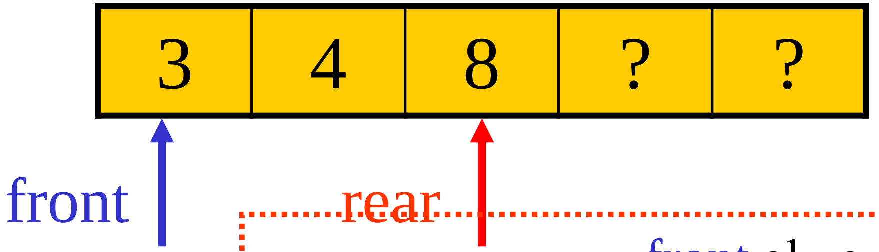


Queue Array implementation



get( ) = 5

front = 0 rear = 1



put(8)

front = 0 rear = 2

Observations:

front always “0”

rear is initially “–1” and can be at most “N-1”

put(int) adds 1 to rear

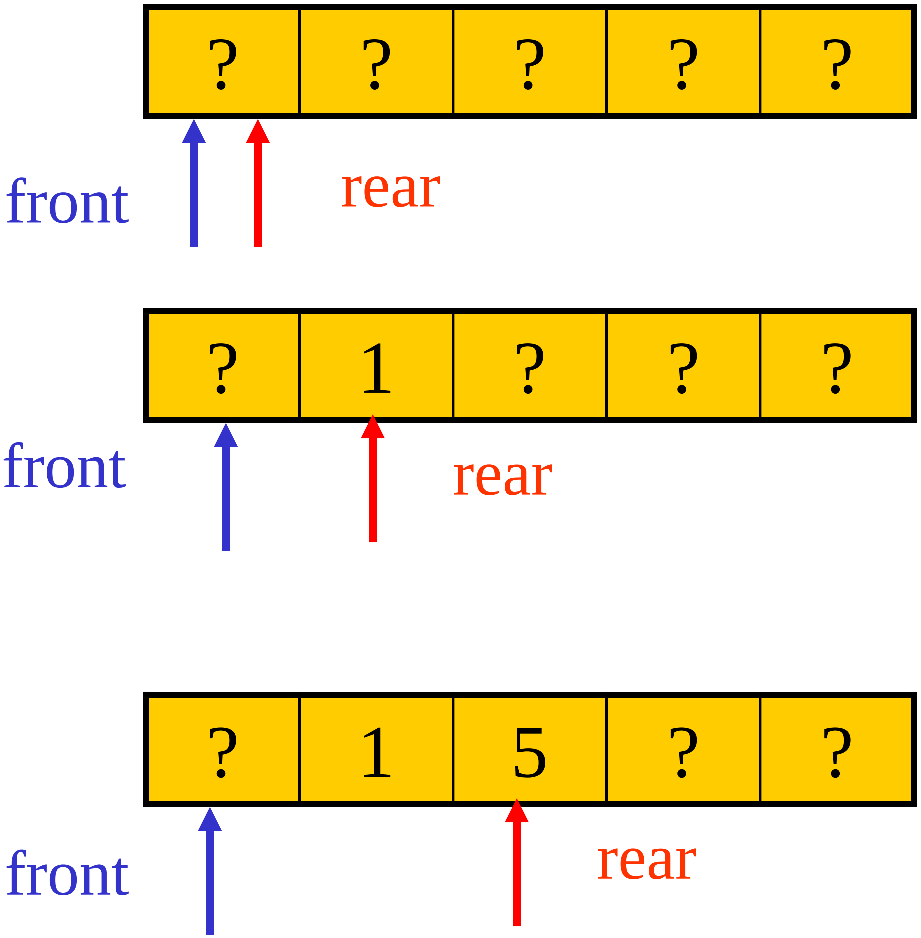
get() subtracts 1 from rear but needs some elements to be shifted.

if rear = -1 queue is empty

if rear = N-1 queue is full

Queue Array implementation

Second Approach(more efficient)



Initial state

front = 0 rear = 0

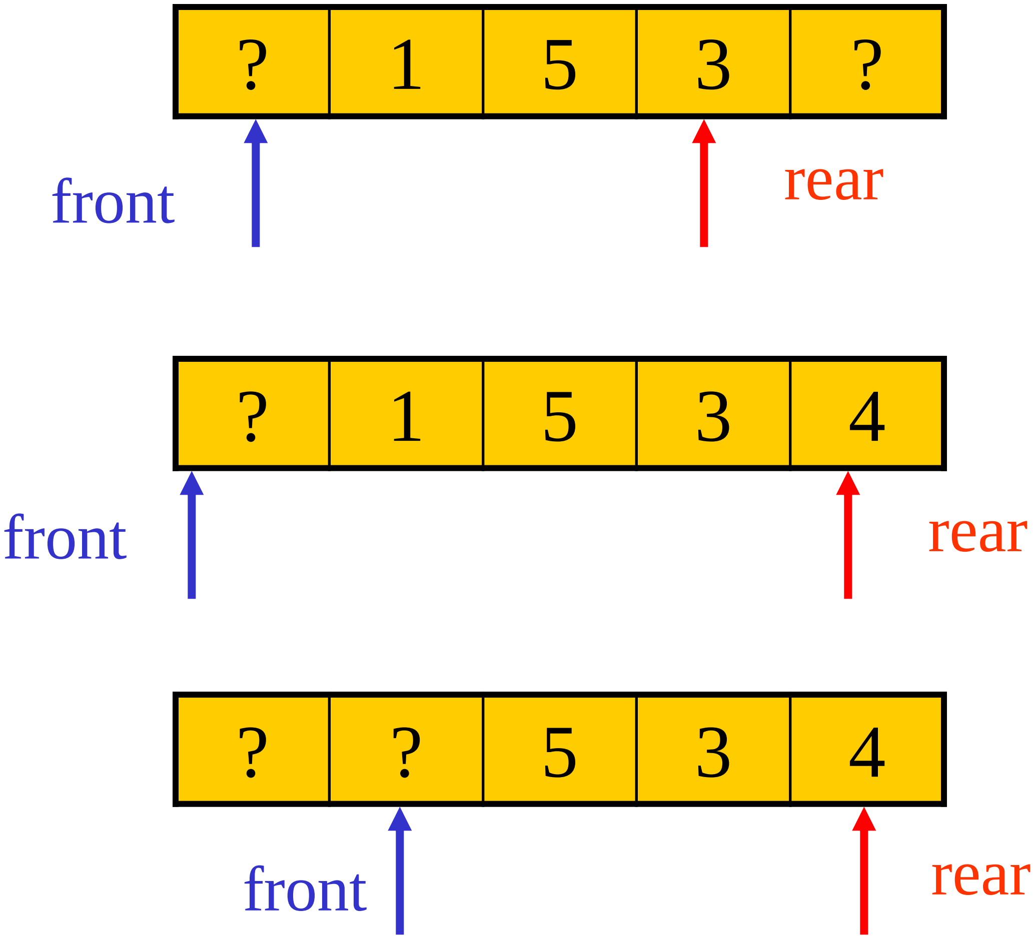
put(1)

front = 0 rear = 1

put(5)

front = 0 rear = 2

Queue Array implementation



put(3)

front = 0 rear = 3

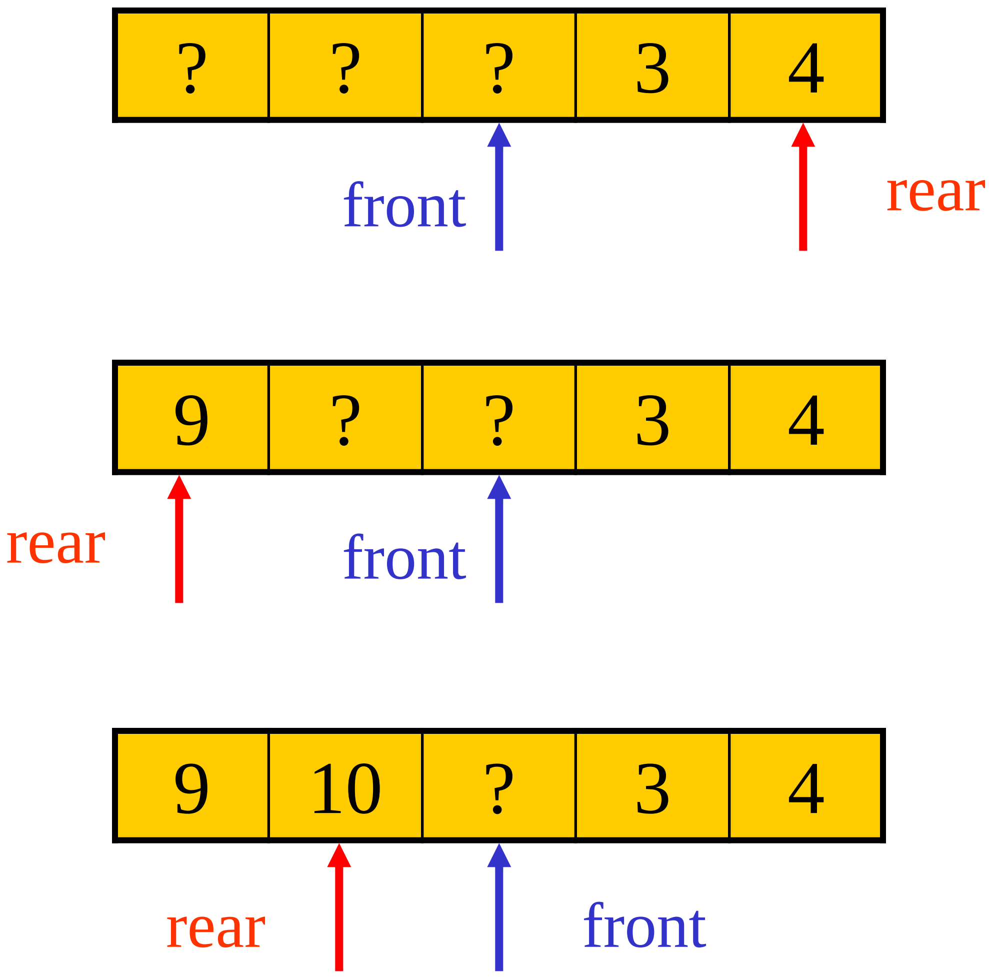
put(4)

front = 0 rear = 4

get( ) = 1

front = 1 rear = 4

Queue Array implementation



get( ) = 5

front = 2 rear = 4

put(9)

front = 2 rear = 0

put(10)

front = 2 rear = 1

Queue Array implementation

Allocate maxSize+1 element (1 for front)

Initially Queue empty front = 0 rear = 0

Observations: Put(int) adds 1 to rear and inserts to array[rear] = item

Get() adds 1 to front and then returns the item

if "rear + 1 = front" Queue is full

Queue Array implementation

```
void QUEUEinit(int maxN){
        q = ( int * )malloc((maxN+1)*sizeof(int));
    N = maxN+1;
        front = 0;
        rear= 0; }
int QUEUEempty(){
        if( rear == front  )
                return 1;
        return 0; }
int QUEUEfull(){
        if( ( ( rear + 1) % N )  == front )
                return 1;
        return 0; }
```

Queue Array implementation

```
void QUEUEput(int item){
        if( QUEUEfull() )
                printf("  Queue is full!!!");
        else   {
                rear = (rear + 1) % N;
                q[rear] = item;} }
int QUEUEget(){
        if( QUEUEempty() ){
                printf("  Queue is empty!!!");
                return -1;}
        else{
                front = (front + 1) % N;
                return q[front];} }
```

ADT observations

Pushdown stacks and FIFO queues are special instances of the **generalized queue ADT.**

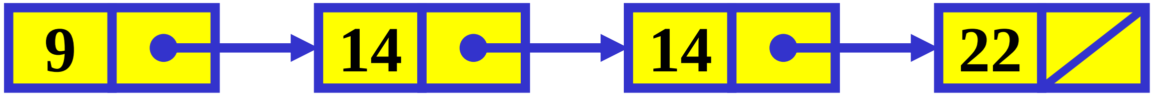
Generalized queue ADT can take many forms depending on the element insertion and removal policy.

- **Pushdown stack**: remove the last item.
- **FIFO queue**: remove the oldest item.
- **Random queue**: remove a randomly selected item.
- **Priority queue**: Remove the item with highest (lowest) value.
- **Symbol table**: remove item whose key is given.
- **De-queue** (**d**ouble-**e**nded queue): add/remove items at either end.

ADT duplicate elements

ADT's also differ in their element acceptance criteria.

"Is element duplication allowed?"



Some policies are:

- Let the client (user) decide.
- Duplicates are allowed (triplicates as well...)
- Duplicates are never allowed, new element is ignored.
- Duplicates are never allowed, new element replaces the old.
- Duplicates are never allowed, retain the more desirable element.

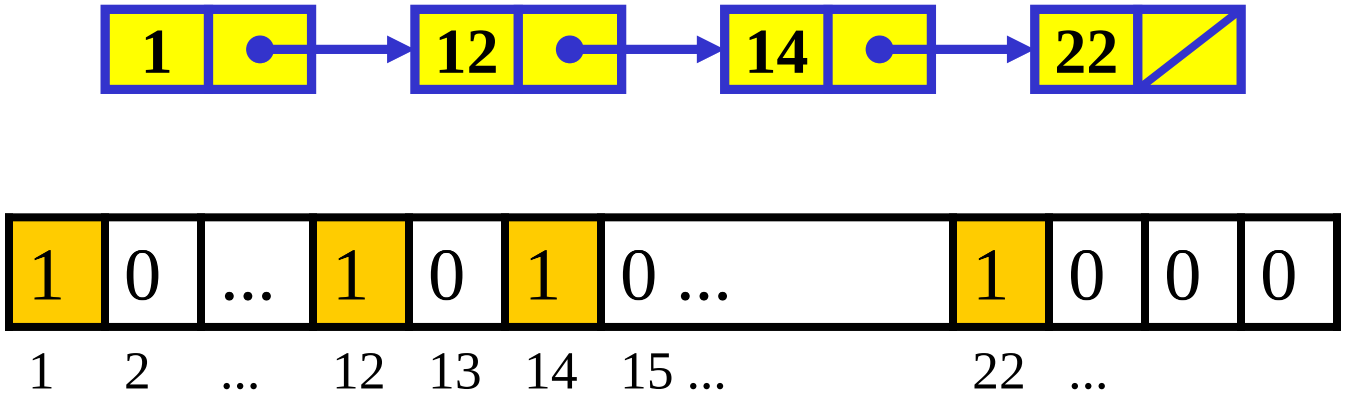
ADT duplicate elements

If duplication is not allowed,

- A test function is needed to determine item **existence** (whether an item is already in the data structure). Sometimes a second array may be used for this purpose.
- A test function is needed for testing item **equality**.

ADT duplicate elements

If the keys are unique and relatively small, use a second array:



Note that the linked list shown above is an ADT: it may actually be implemented using an array.