NANYANG TECHNOLOGICAL UNIVERSITY

# CE1007/CZ1007 DATA STRUCTURES

## Lecture 06: **Advanced Stacks & Queue**

Dr. Owen Noel Newton Fernando

**College of Engineering**

School of Computer Science and Engineering

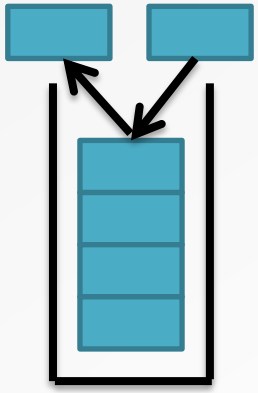
# STACK IMPLEMENTATION USING LINKED LISTS

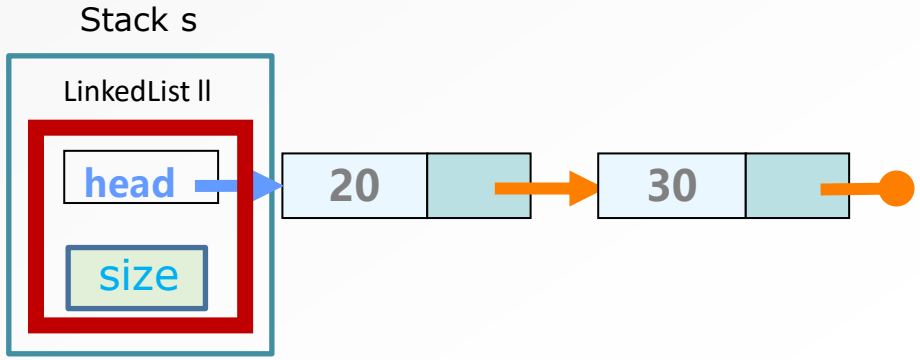
- Stack structure

```
typedef struct _stack{
    LinkedList ll;
} Stack;
```

Notice this is a LinkedList, not a LinkedList *

- Basically wrap up a linked list and use it for the actual data storage
- Just need to ensure we control where elements are added/removed
- Notice that the LinkedList already takes care of little things like keeping track of number of nodes, etc.



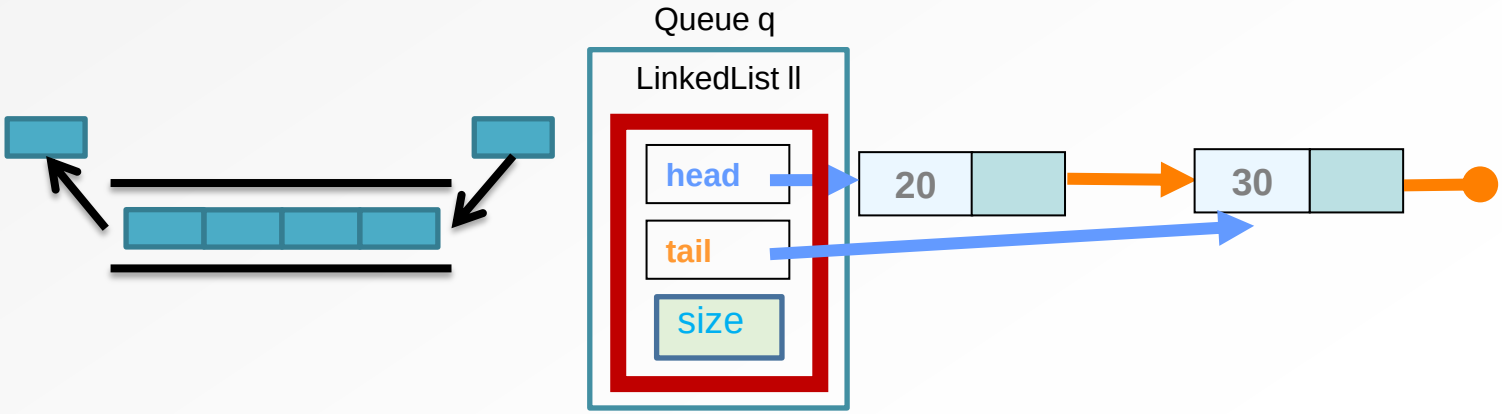


# QUEUE IMPLEMENTATION USING LINKED LISTS

- Queue structure

```
typedef struct _queue{
     LinkedList ll;
} Stack;
```

- Again, wrap up a linked list and use it for the actual data storage
- Notice that the LinkedList already takes care of little things like keeping track of number of nodes, etc.
- Need to use a tail pointer to make the operation efficient



# QUEUE AND STACK IMPLEMENTATION USING LINKED LISTS

```
typedef struct _listnode{
    int num;
    struct _listnode *next;
}ListNode;

typedef struct _linkedlist{
    ListNode *head;
    ListNode *tail;
    int size;
}LinkedList;


typedef struct _queue {
    LinkedList ll;
}Queue;
```

```
typedef struct _listnode{
    int num;
    struct _listnode *next;
}ListNode;

typedef struct _linkedlist{
    ListNode *head;
    int size;
}LinkedList;


typedef struct _stack {
    LinkedList ll;
}Stack;
```

# REVIEW OF THE POINTER

# YOU SHOULD FIGURE OUT WHICH ONE IS POINTER, WHICH ONE IS NOT

When should I use “->” or ”.” ?
pointer->
Non-pointer.

# YOU SHOULD FIGURE OUT WHICH ONE IS POINTER, WHICH ONE IS NOT

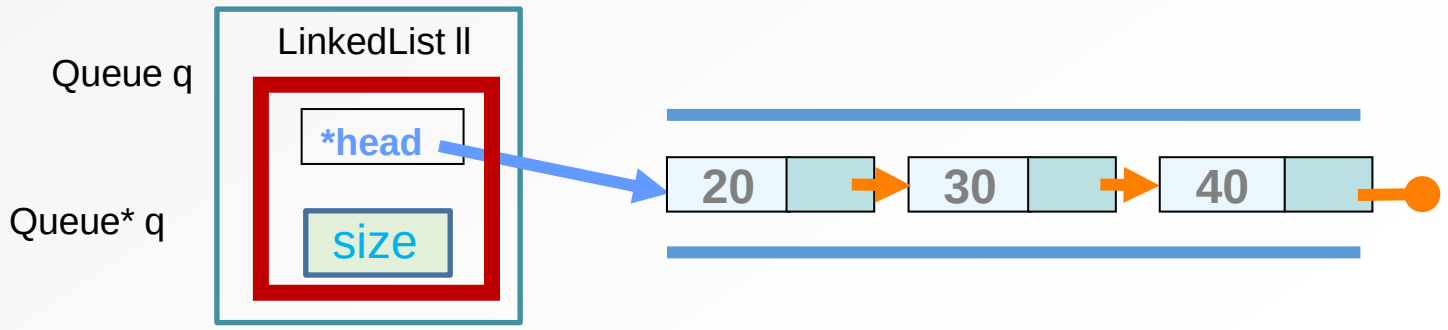
When should I use "->" or "." ?
pointer->
Non-pointer.

- Queue structure

```
typedef struct _queue {
   LinkedList ll;
}Queue;
```

**Queue q;**
**q.ll** **LinkedList**
**q.ll.head** ListNode pointer
**q.ll.head->num** **integer**

**Queue *q;** Is a pointer
**q->ll** **LinkedList**
**q->ll.head** ListNode pointer
**q->ll.head->num** **integer**



# YOU SHOULD FIGURE OUT WHICH ONE IS POINTER, WHICH ONE IS NOT

```
Queue *q;
q->ll.head->num=20;
```

- When you define Queue, you need to define LinkedList and ListNode first;
- In Queue typedef, you can use LinkedList *ll; (q->ll->head->num)
- But head is always declared as a pointer in textbooks.

```
typedef struct _listnode{
    int num;
    struct _listnode *next;
}ListNode;

typedef struct _linkedlist{
    ListNode *head;
    ListNode *tail;
    int size;
}LinkedList;


typedef struct _queue {
    LinkedList ll;
}Queue;
```

```
void enqueue (Queue *q, int item)
{
      if (q->ll.tail==NULL){
        insertNode(&(q->ll.head), 0,
      item);
        q->ll.tail=q->ll.head;
      }
      else {
        q->ll.tail->next=malloc(…);
        q->ll.tail=q->ll.tail->next;
        q->ll.tail->num = item;
        q->ll.tail->next=NULL;
        q->ll.size ++;
      }
}
```

# OUTLINE

- Array-based Queue Implementation
- Array-based Stack Implementation

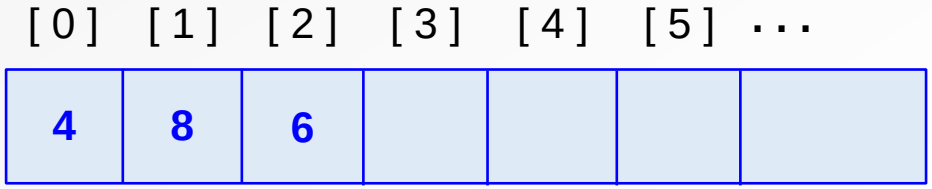
# OUTLINE

- **Array-based Queue Implementation**
- Array-based Stack Implementation

# ARRAY IMPLEMENTATION

Because we can only add an item to the back and remove an item from the front, we can:

- Implement the queue with an array
- For example, this queue contains the integers 4 (at the front), 8 and 6 (at the rear).



# ARRAY IMPLEMENTATION

- New C structure:

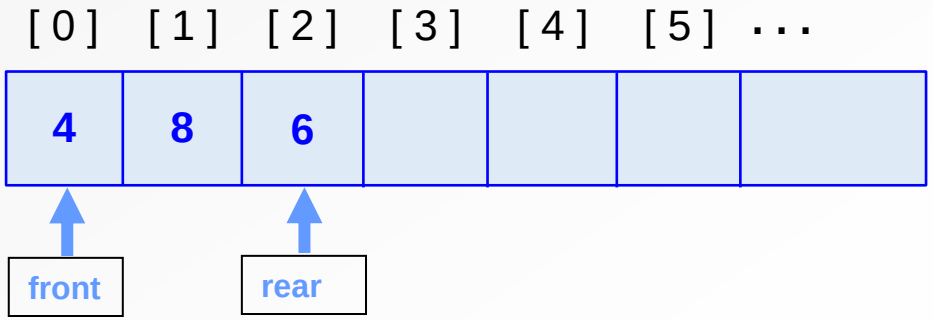
```
typedef struct {

  int num[MAX];
  int front;
  int rear;
  int size;
} queue;
```

The array can store other type of data, such as char, string, etc.

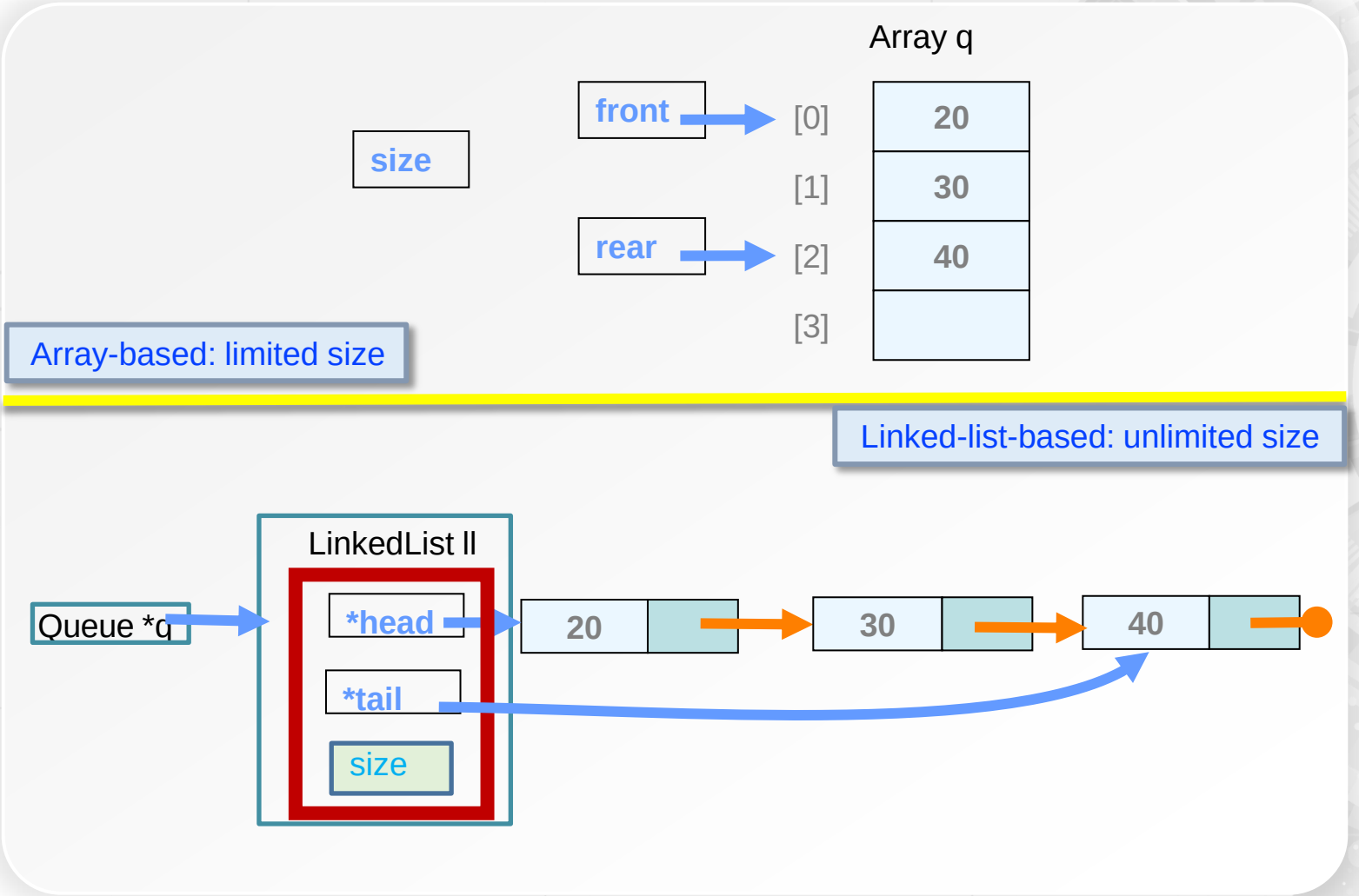
The array is of fixed size: a queue of maximum MAX elements

Need to maintain track of both **front** and **rear**

- Functions: enqueue(), dequeue(), peek(), isEmptyQueue(), isFullQueue()



# QUEUE: ARRAY-BASED VS. LINKED-LIST-BASED

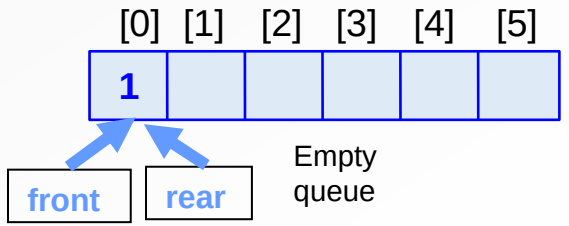


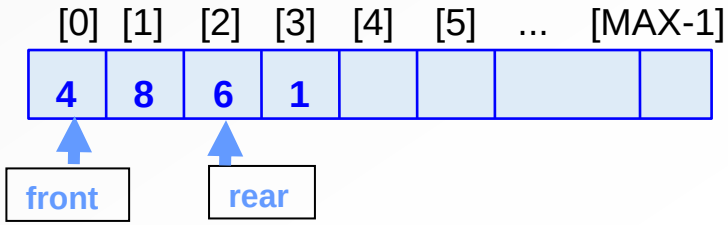
# isFullQueue()

```
int isFullQueue(queue *q ) {
    if (q->size == MAX) return 1;
    else return 0;
}
```

# enqueue()

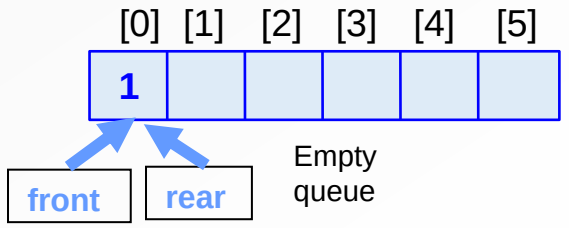
```
int enqueue(queue *q, int item)
{
    if (q->rear<MAX-1) { //available elements at the rear
       if (q->size==0)   //empty queue
           q->num[q->rear]=item;
       else {            //non-empty queue
           q->rear ++;
           q->num[q->rear]=item;}
       q->size++;
       return 1;
    }
    else return -1;
}
```

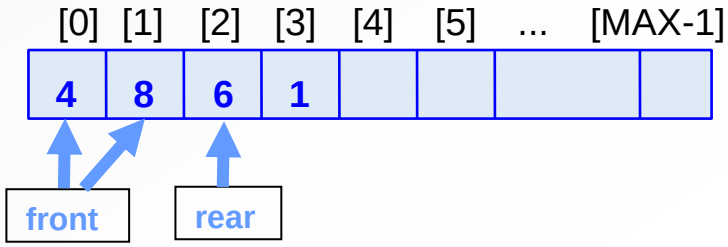


# dequeue()

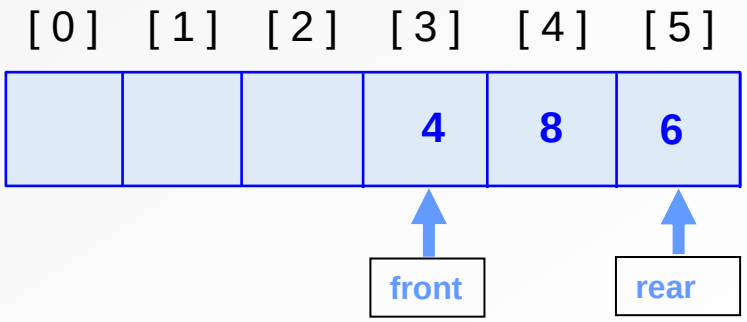
```
int dequeue(queue *q)
{    int temp;
     if (!isEmptyQueue(*q))
     {
          temp= q->num[q->front];
          q->size--;
          if (q->size>0) q->front ++;
          return temp;
     }
     else
      return NULL_VALUE;
}
```



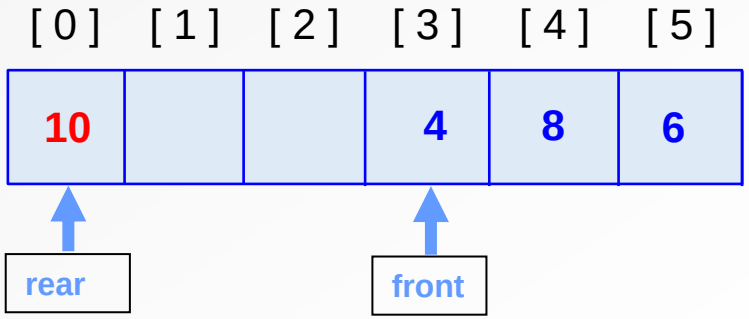
# CAN WE ADD AN ITEM TO THE QUEUE?

- The size of the array is MAX=6;
- Rear = 5 = MAX-1, so we can't add one ☹
- But the array has 3 available elements… it is a waste!
- Wrap the array → circular queue

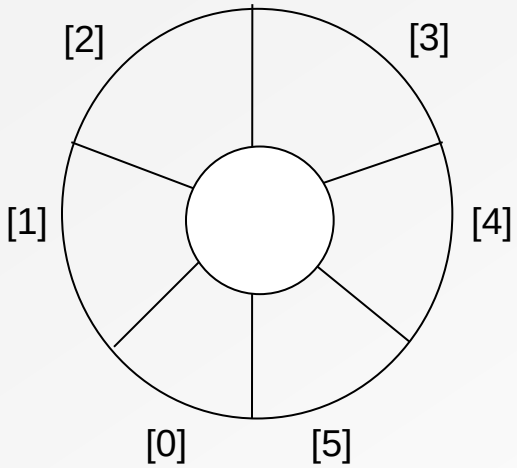


# CAN WE ADD AN ITEM TO THE QUEUE?

- The size of the array is MAX=6;
- Rear = 5 = MAX-1, so we can't add one ☹
- But the array has 3 available elements… it is a waste!
- Wrap the array → circular queue



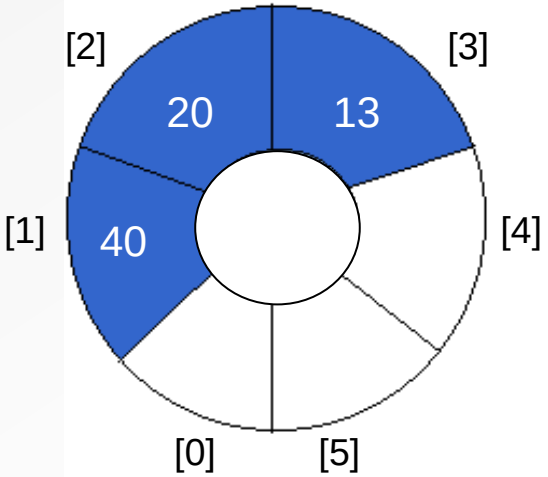
# CIRCULAR ARRAY FOR QUEUE




**front = 0**
**rear = 0**

EMPTY QUEUE

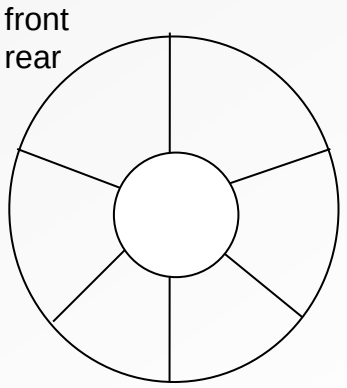
if (size==0 )

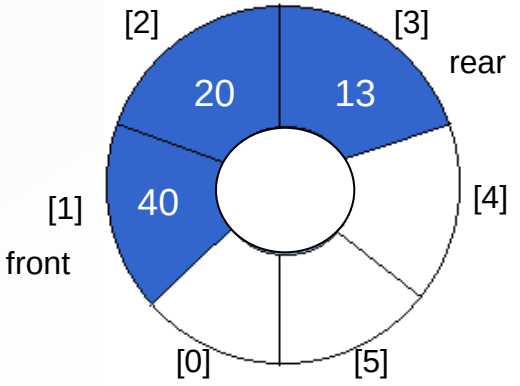



**front = 1**
**rear = 3**

# enqueue() FOR CIRCULAR ARRAY

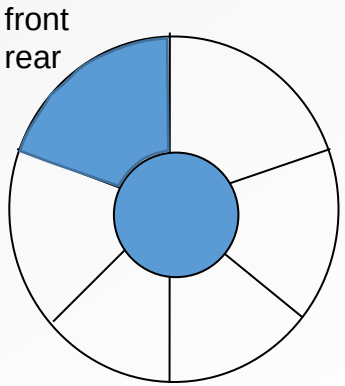
```
int enqueue(queue *q, int item)
{
    if (q->size <MAX) {    //isFullQueue()
        if (q->size==0)    //empty queue
            q->num[q->rear]=item;
        else {             //non-empty queue
            q->rear = (q->rear + 1)% MAX ;
            q->num[q->rear]=item; }
            q->size++;
            return 1;
     }
     else return -1;
}
```

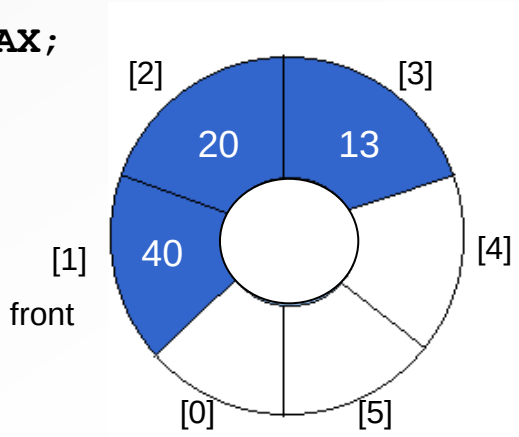


# dequeue() FOR CIRCULAR ARRAY

```
int dequeue(queue *q)
{    int temp;
     if (!isEmptyQueue(*q))
     {
        temp= q->num[q->front];
        q->size--;
        if (q->size >0)
           q->front = (q->front +1) % MAX;
        return temp;
     }
     else return NULL_VALUE;
}
```



# C STRUCTURE FOR ARRAY-BASED QUEUE

- C structure for array-based queue:

```
typedef struct {
    int num[MAX];
    int front;
    int rear;
    int size;
}queue;
```

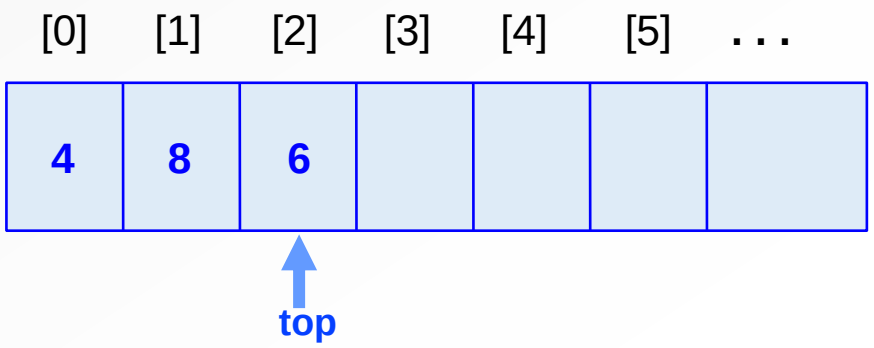
- Functions: enqueue(), dequeue(), peek(), isEmptyQueue(), isFullQueue()

# OUTLINE

- Array-based Queue Implementation
- **Array-based Stack Implementation**

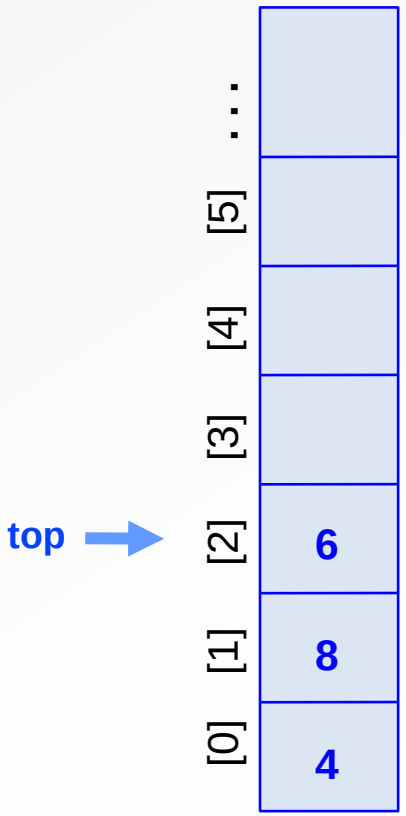
# ARRAY-BASED STACK IMPLEMENTATION

- A stack can be implemented with an array because we can only add/remove from the top.
- For example, this stack contains the integers 6 (at the top), 8 and 4(at the bottom).
- Array: **very natural and simple** implementation for stacks



# ARRAY-BASED STACK IMPLEMENTATION

- A stack can be implemented with an array because we can only add/remove from the top.
- For example, this stack contains the integers 6 (at the top), 8 and 4(at the bottom).
- Array: **very natural and simple** implementation for stacks



# ARRAY IMPLEMENTATION

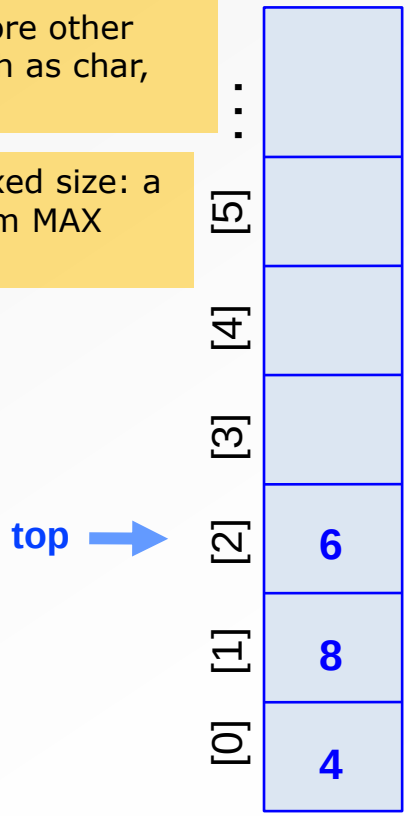
- New C structure:

```
typedef struct {
    int num[MAX];
    int top;
} stack;
```

The array can store other type of data, such as char, string, etc.

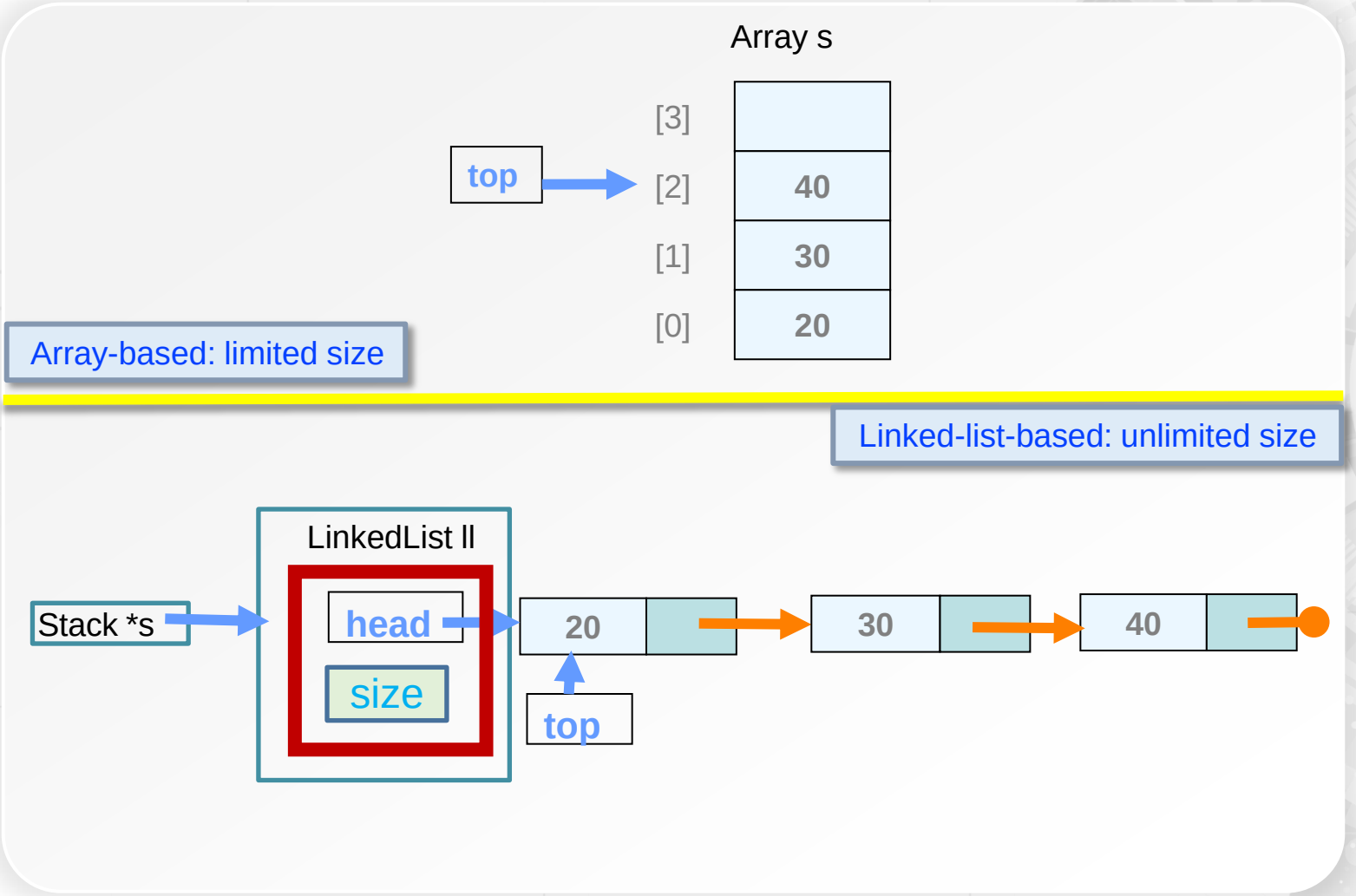
The array is of fixed size: a stack of maximum MAX elements

  - Bottom stack item stored at element 0
  - Last index in the array is the *top*
  - Increment *top* when one item is push(), decrement after pop()
  - size is not necessary here




- Functions: push(), pop(), Peek(), isEmptyStack(), isFullStack()

# STACK: ARRAY-BASED VS. LINKED-LIST-BASED

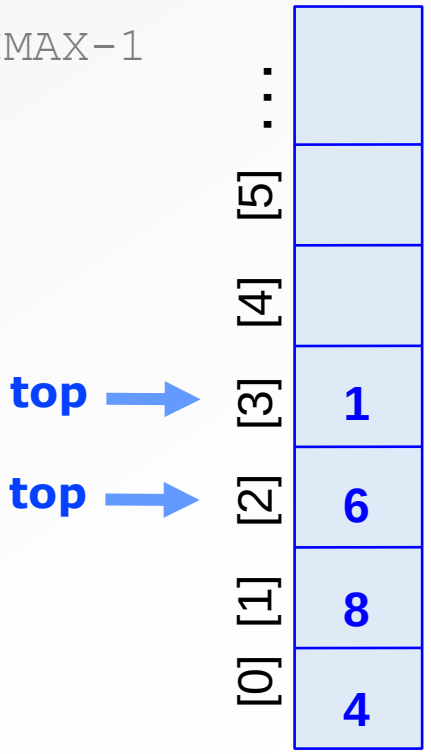


# isFullStack()

```
int isFullStack(stack *s) {
    if (s->top == MAX-1) return 1;
    else return 0;
}
```

# push()

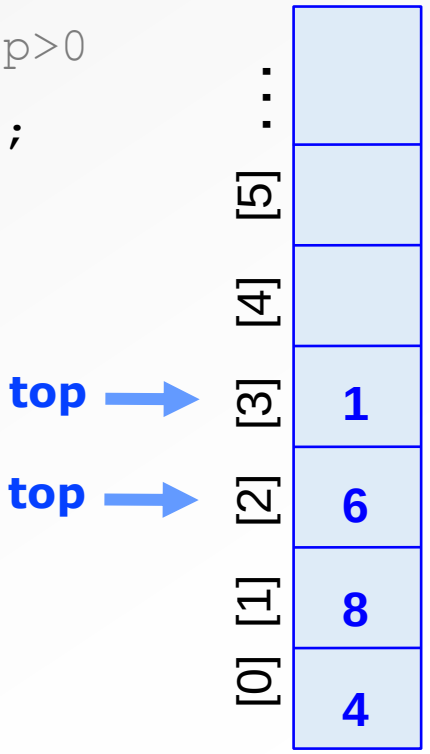
```
void push(stack *s, int item)
{
      if (!isFullStack(s)) //q->top<MAX-1
        s->num[++(s->top)]=item;
    return;
}
```




For empty stack: top = -1

# pop()

```
int pop(stack *s)
{
    if (!isEmptyStack(*s)) // q->top>0
       return (s->num[(s->top) --]);

    else return NULL_VALUE;
}
```



# C STRUCTURE FOR ARRAY-BASED STACK

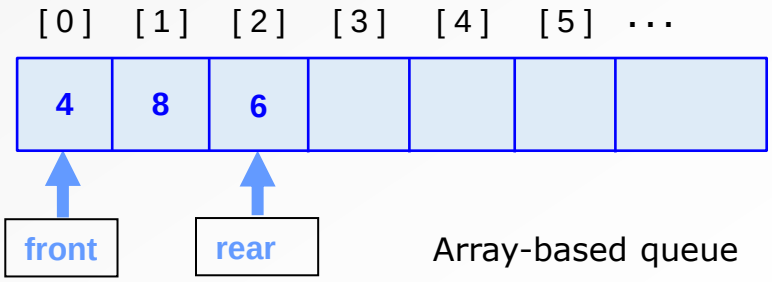
- New C structure:

```
typedef struct {
    int num[MAX];
    int top;
}stack;
```

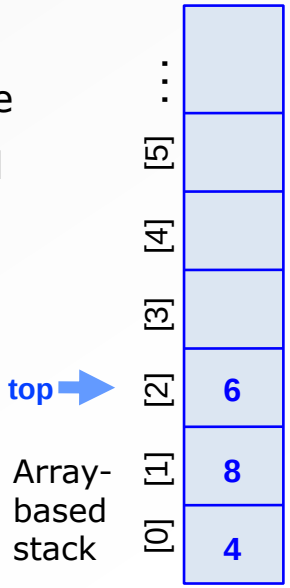
- Functions: push(), pop(), peek(), isEmptyStack(), isFullStack()

# REMARKS ON ARRAY-BASED IMPLEMENTATION

- **Easy to implement**, simple coding
- **For memory usage**
  - Save memory: If size of the queue is predetermined, no extra space for pointers.
  - Waste of memory: if we use less elements.
  - Cannot add(enqueue/push) more elements than the array can hold. it has a limited capacity with a fixed array



Array-based queue



Array-based stack