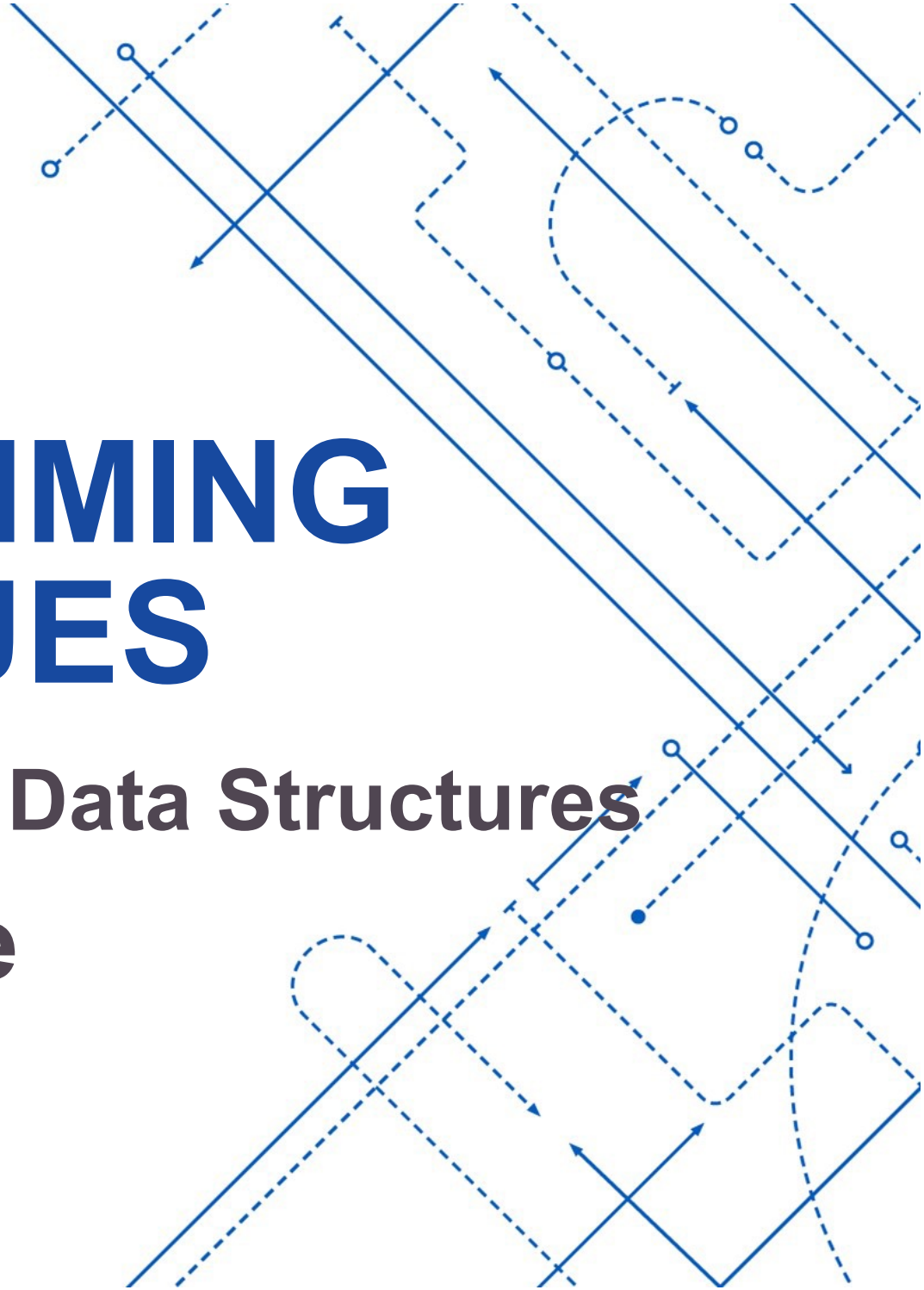
fit@hcmus

PROGRAMMING TECHNIQUES

Week 5: Dynamic Data Structures

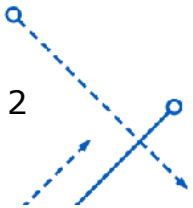
Stack & Queue

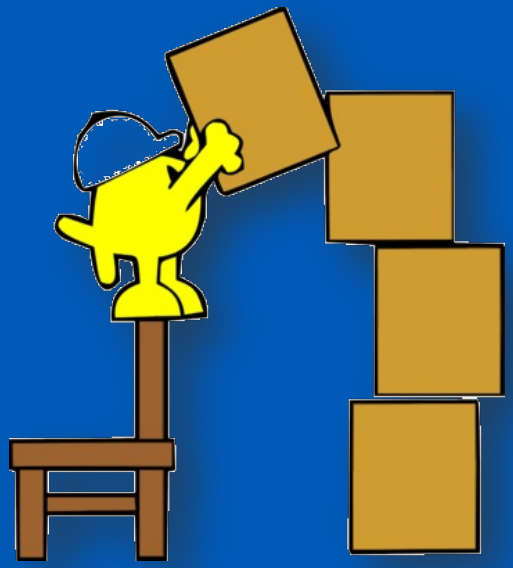
Nguyễn Hải Minh - 02/2024



Today content

- Stacks
- Queues
- Walk through examples





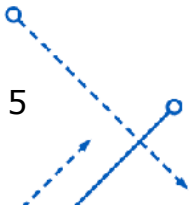
STACKS

Stacks – Introduction

- Popular images:

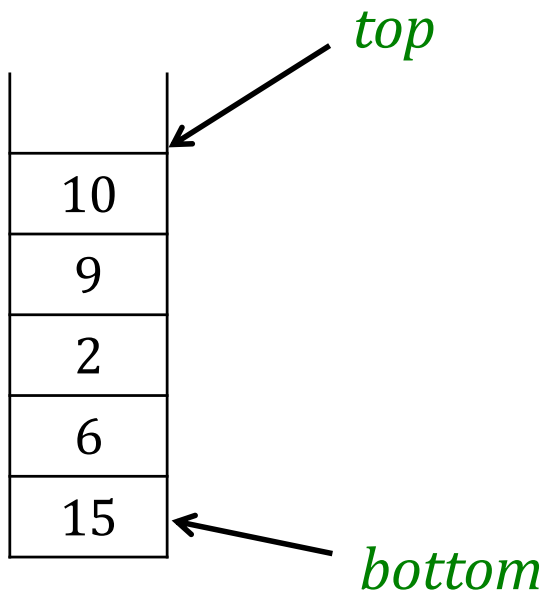
Stacks – Introduction

- We have talked about lists of data much of the term.
- One common use of a list is to represent a stack abstraction.
- Stacks allow us to add data and remove data at only one end (called the top).
- We can **push** data onto the top and **pop** data out of the top.



Stacks – Definition

- Stack is a dynamic set in which the element removed from the set is the one **most recently inserted**.
- Last-in, first out (LIFO) policy.



Implement a Stack

- Using an array:
  - An array of at most *n* elements: *S*[0…*n*-1]
  - An attribute *top* that indexes the most recently inserted element.
  - Empty stack: *top* = -1

```
struct stack{
        video* S; //array that hold the elements
        int top; //index of the last element
        int n;   //size of the array
};
```

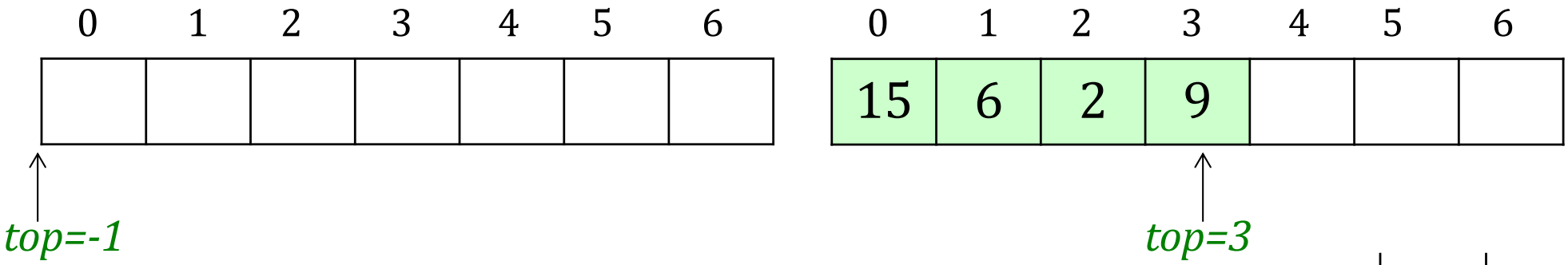
Implement a Stack

- Using a linked list
  - A pointer *pTop* points to the top of the stack
  - Empty stack: *pTop* = NULL

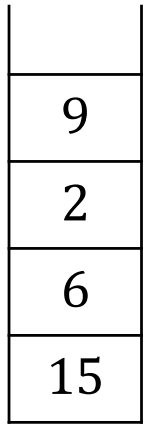
```
struct node {
    video data;
    node* pNext;
};

struct stack{
    node* pTop; //pointer that points to the top of the stack
};
```

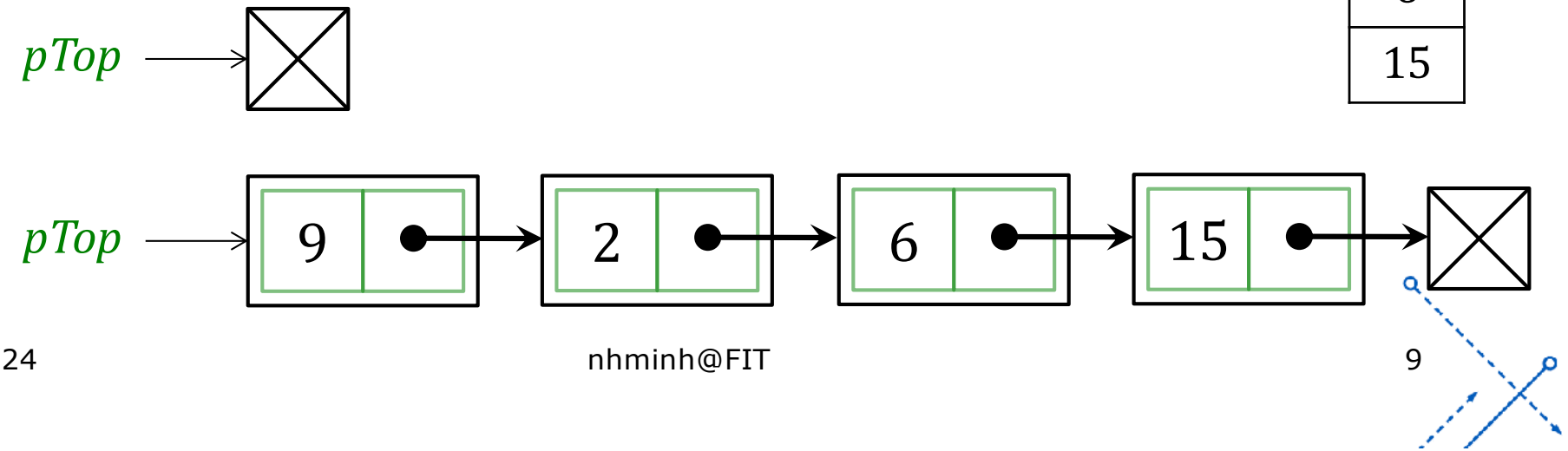
Implement a Stack

- Using an array:



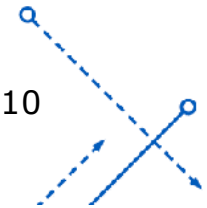
- Using a linked list:



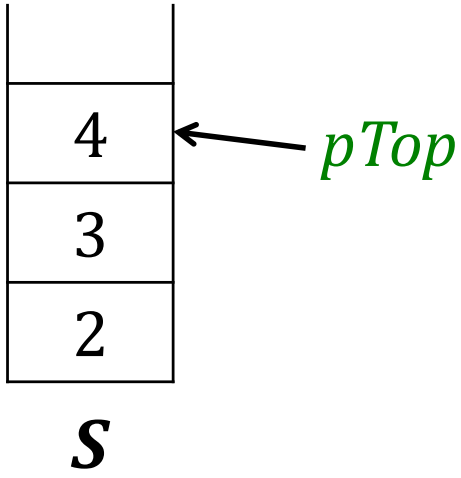


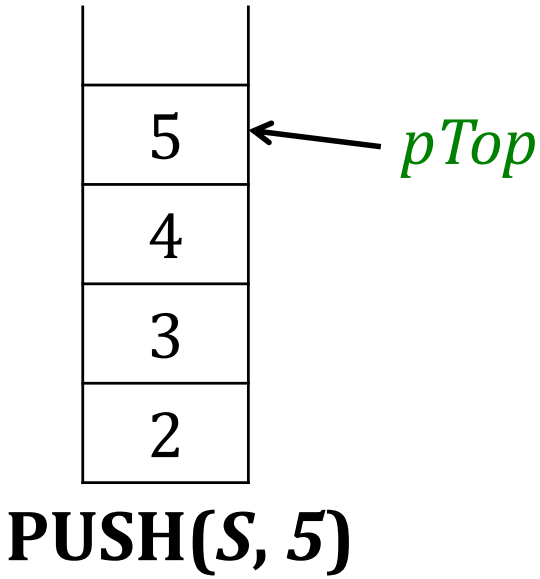
Stack Operations

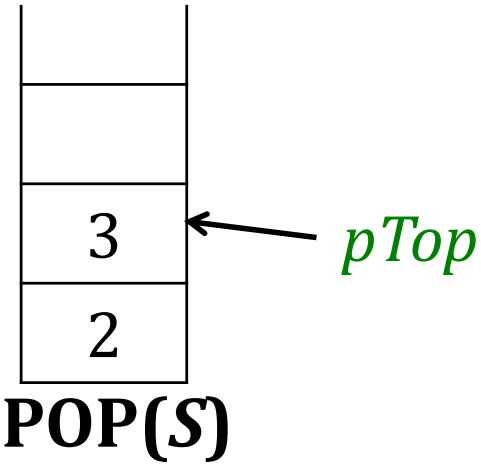
- STACK-EMPTY: Check if a stack is empty
- PUSH: Insert a new element into the stack
- POP: Delete an element in the stack
- TOP: Get the top element without delete it from the stack
- STACK-FULL: Check if a stack if full

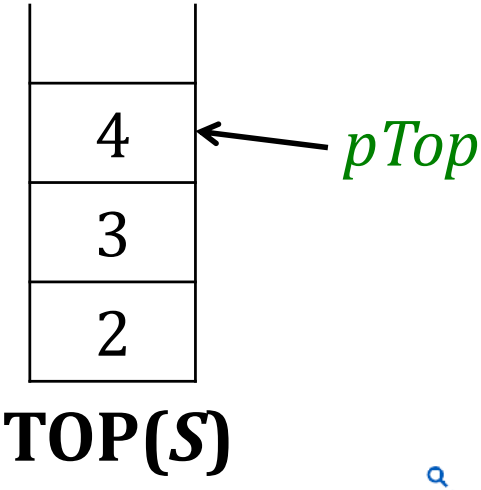


Stack Operations – Examples







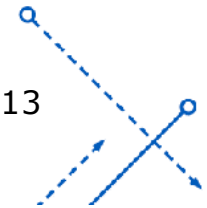


Stack Operations

```
bool IsStackEmpty(stack s);
bool IsStackFull(stack s);
int Push(stack& s, const video& data);
int Pop(stack& s, video& data);
int Top(stack s, video& data);
```

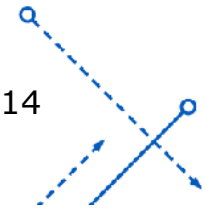
Implement a Stack – using Linked List

- Implementing Push and Pop with a linear linked list data structure
  - represent very simple insert and remove algorithms
  - in fact, push is the same as adding at the beginning of a LLL
  - and, pop is the same as removing at the beginning of a LLL
  - why wouldn't we add/remove at the end (or tail) instead?



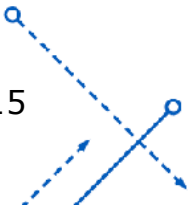
Implement a Stack – using Linked List

- Why wouldn't we add/remove at the end (or tail) instead?
  - pushing at the tail would be just as easy and efficient as pushing at the head iff we kept a tail pointer.
  - but, popping at the tail would require that we traverse to the (tail-1) node...regardless of whether or not there was a tail pointer.
  - a doubly linked list is not the answer



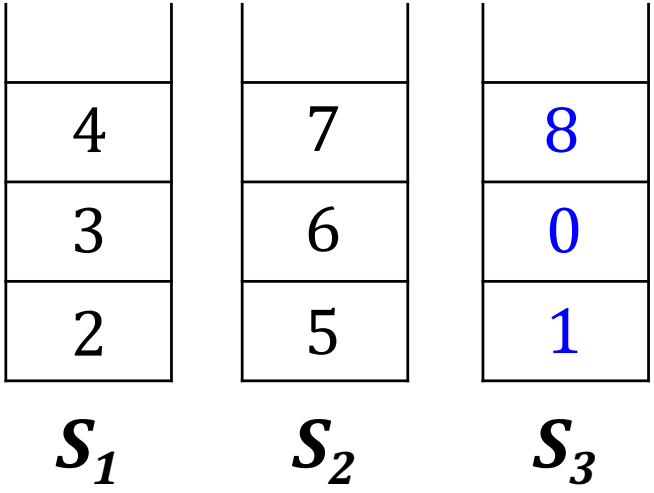
Stack Applications

- Delimiter Matching (part of any compiler)
  - C++: ( ) { } /* */ [ ]
  - Expression: $((a+b)\times((c-d)+e))$
  - Delimiters can be nested
- Idea:
  - Read the characters from left to right.
  - Whenever you see a left (opening) delimiter, push it to the stack.
  - Whenever you see a right (close) delimiter, pop the delimiter from the stack.
    - If you can't pop (because stack is empty) or they don't match, return false
    - Otherwise, continue reading characters
  - If stack is empty when we finish reading string, return true
  - Otherwise, return false.



Stack Applications

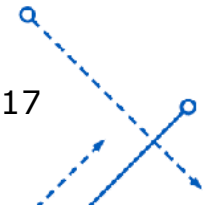
- Adding two large number
  - Treat these numbers as strings of numerals, store the numbers corresponding to these numerals on 2 stacks.
  - Perform addition by popping numbers from the stacks
- Example: 234 + 567



- $Pop(S_1) + Pop(S_2) = 4+7 = 11$
- → Add 1 to S3
- $Pop(S_1) + Pop(S_2) = 3+6+1 = 10$
- → Add 0 to S3
- $Pop(S_1) + Pop(S_2) = 2+5+1 = 8$
- → Add 8 to S3
- S_1 S_2 are empty
- → $Pop(S_3)$ and get the result: 801

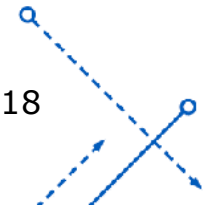
Stack Applications

- Reverse Polish Notation (RPN):
  - Every operator follows all its operands.
  - Example:
    - 3 + 4 → 3 4 +
    - (3 – 4) x 5 → 3 4 – 5 x
  - Implementations:
    - Mac OS X Calculator
    - iPhone apps: RPN Calculator
    - Android apps: RealCalc



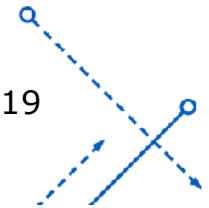
Stack Applications

- Backtracking:
  - Used in algorithms in which there are steps along some path from a starting point to some goal.
  - Example:
    - Find your way through a maze
    - Find a route from one point to another point
    - Games in which there are moves to be made (chess, tic-tac-toe)



Stack Applications

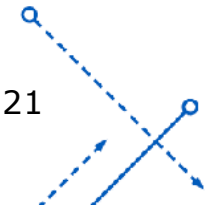
- Converting Decimal to Binary:
  - Read a number
  - Loop while number > 0
    - digit = number % 2
    - push digit to stack
    - number = number/2
  - While stack is not empty
    - pop and print the top element of stack



QUEUES

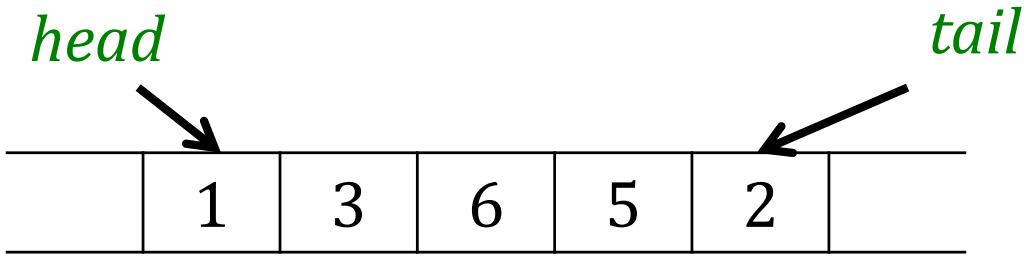
Queues – Introduction

- Another common use of a list is to represent a queue abstraction
- Queues allow us to add data at one end (the rear) and remove data at the other end (at the front)
- We can enqueue data at the rear and dequeue data at the front (FIFO)



Queue – Definition

- Queue is a dynamic set in which the element removed from the set is the one that has been inserted to the set for the longest time.
- First-in, First out (FIFO) policy.

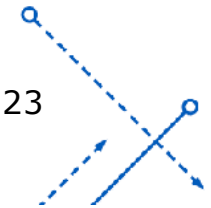


Implement a Queue

fit@hcmus

Using Array

```
struct queue {
        video* S; //list of elements
        int n;   //max size of the list
        int front;  //the first element id
        int tail;   //the last element id
};
```



Implement a Queue

- Using Linked List

```
struct node {
    video data;
    node* pNext;
};

struct queue{
    node* pFront; //the head pointer
    node* pRear; //the tail pointer
};
```

Queue Operations

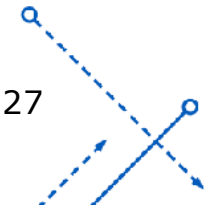
- QUEUE-EMPTY: Check if a queue is empty
- ENQUEUE: Insert a new element into the queue
- DEQUEUE: Delete an element in the queue
- QUEUE-FULL: Check if a queue is full
- FIRST: Get the head element without delete it from the queue

Queue Operations

```
bool IsQueueEmpty(queue s);
bool IsQueueFull(queue s);
int Enqueue(queue& s,const video& data);
int Dequeue(queue& s, video& data);
int First(queue s, video& data);
```

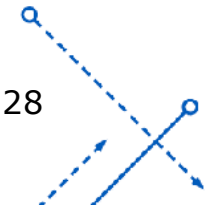
Queues

- Implementing enqueue and dequeue with a linear linked list data structure
  - are also simple
  - but, should the “rear” pointer point to the first or the last node? And, should the “front” pointer point to the first or the last node?
  - draw the pointer diagrams for either way and decide which would traverse less...



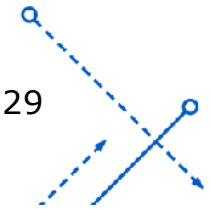
Queues

- enqueue should add at *the rear - the tail*
- dequeue should remove at *the front - the head*
- Why?
  - enqueuing at the front or rear is equally easy and efficient
  - but, dequeuing at the rear requires that we traverse to the “last-1” node (but a doubly linked list would be overkill). Luckily, dequeuing at the front is simple and efficient

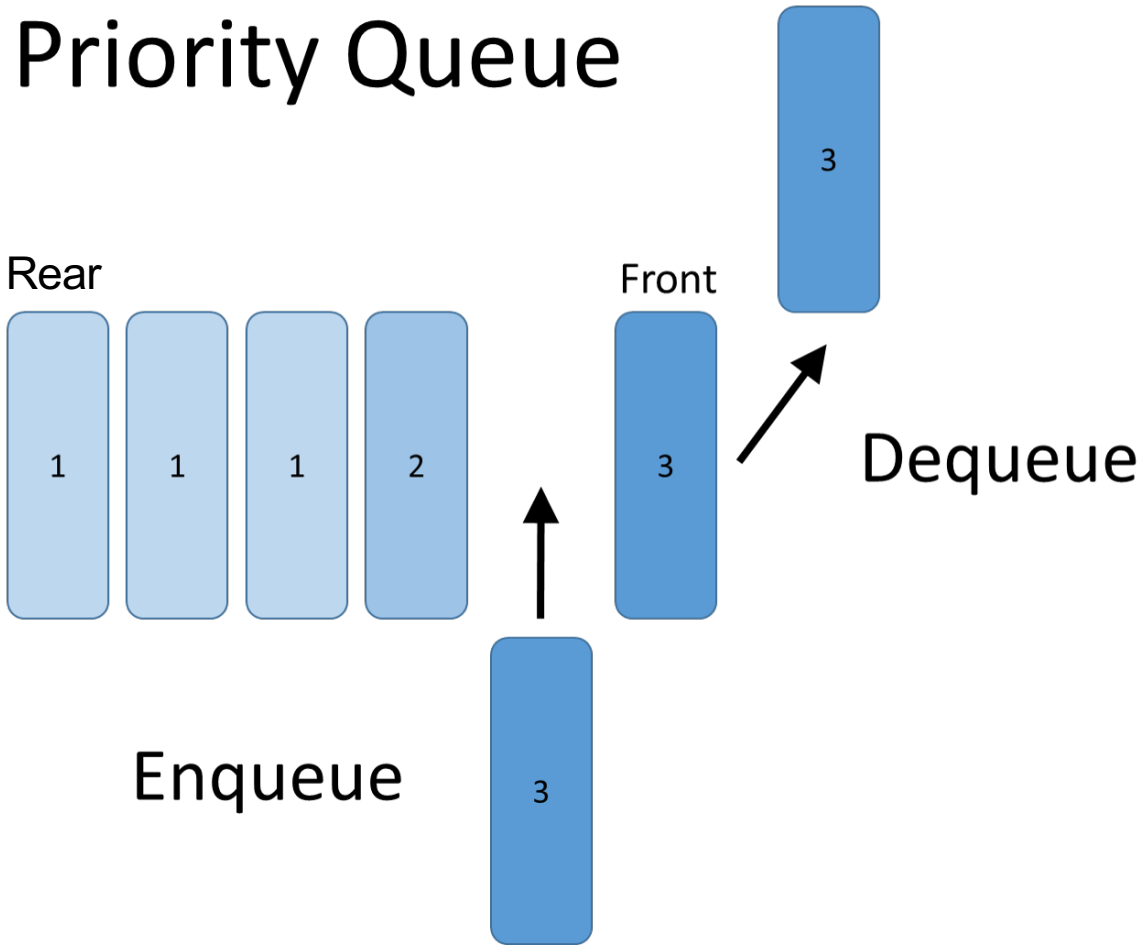


Queue Applications

- Task Scheduling
- Resource Allocation
- Message buffering
- Traffic Management
- Print jobs, procedures management
- Download jobs in browsers.
- …

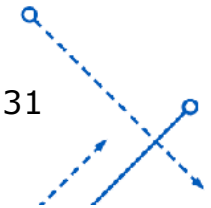


Priority Queues



Priority Queues

- Items are ordered by priority rather than temporally.
- Queue implementations are modified to acquire a priority queue.
  - Enqueue must scan for the ***correct insertion point***.
- A linked list implementation is preferred.
- A heap can be used as the underlying implementation of a priority queue
  - *(to be discussed in heapsort algorithm – Data Structures & Algorithms)*



Next week

- Midterm Examination