

# Data Structures II - Stacks & Queues

Doan Nhat Quang

doan-nhat.quang@usth.edu.vn  
University of Science and Technology of Hanoi  
ICT department

# Linked Lists

## Linear Data Structures:

- Lists
  - Definition: is a collection with a finite number of data objects (same type) and has a finite size.
  - List data structure: Array-based Lists, Linked Lists
  - List Operations
- Stacks
- Queues

# Today Objectives

- Introduce the basics of Stacks and Queues: declaration, initialization, and use.
- Learn different functions and operations with Stacks and Queues: add, remove, search, etc.
- Implement examples in C/C++.

# Plan

1 Stacks

2 Queues

# Stacks



Stack of books



Stack of coins

# Stacks

## General Definition

A **stack** is a pile of objects, typically one that is neatly arranged.

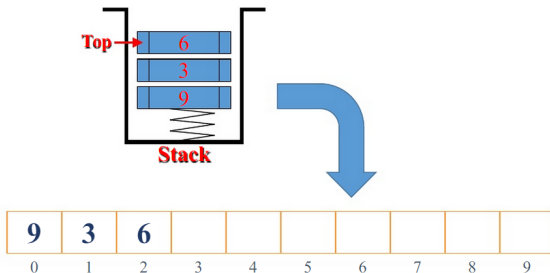
## Programming Definition

A **stack** is a container of objects inserted and removed according to the First In Last Out (FILO) principle.

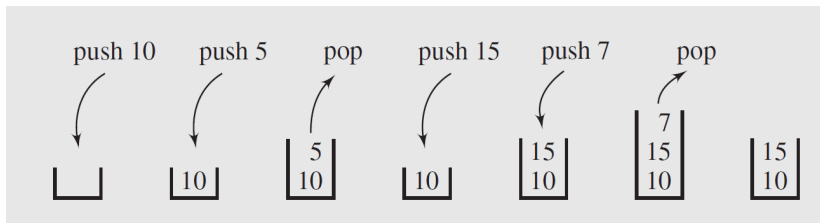
# Stacks

## Principle

- A linear data structure is used to store data in a particular order.
- Storing and retrieving data are performed only on the top: **Push** inserts an element; **Pop** removes the last element that was added.
- Access of items in a stack is restricted; it follows **First In Last Out (FILO)** order.



# Stacks



Push and pop operations follow FILO order.



# Stacks

## Stack Application

- Expression evaluation: calculate arithmetic expression.
- Backtracking: This is a process when you need to access the most recent data element in a series of elements
  - Find your way through a maze.
  - Find a path from one point in a graph (roadmap) to another point.
  - Play a game with moves to be made (checkers, chess, sudoku).
- Undo/Redo-mechanism of text editors (Back/Forward Navigation of web-browsers).
- Call stack in recursive functions.
- Data structures for Machine Learning algorithms.

# Stacks

## Stack Application

Arithmetic Expression:

- **infix** - operation **between** operands

$$(3+5)*10$$

- **prefix** - operation **before** operands

$$* + 3 5 10$$

- **postfix** - operation **after** operands

$$3 5 + 10 *$$

# Stacks

## Stack Application

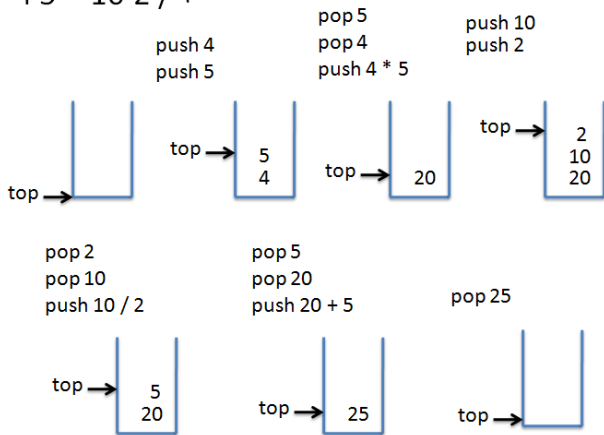
Arithmetic Expression: evaluating postfix

- repeat
  - find the first operation preceded by two operands
  - evaluate and replace
- Example:

$$\begin{aligned} & 4\ 5\ * \ 10\ 2\ /\ + \\ = & 20\ 10\ 2\ /\ + \\ = & 20\ 5\ + \\ = & 25 \end{aligned}$$

# Stacks

4 5 \* 10 2 / +



# Stacks

Stacks implementation may offer the possible operations:

- `init()`: create an empty stack.
- `isEmpty()`: check if the stack is empty.
- `push()`: add a new item at the top of a stack.
- `pop()`: remove the top item of a stack.
- `top()`: retrieve the top item of a stack.

Other operations can be possibly defined:

- `size()`: return the size of a stack.
- `isFull()`: check if the stack is full.
- `display()`: display the content of a stack.
- etc.

# Stacks

## Stack Data Structure

There are several solutions to the stack implementation using different declarations.

- **Static array-based stack**: arrays can be simply used to manipulate collections of items.
- **Dynamic array-based stack**: **malloc()** is capable of representing a stack.
- **Linked stack**: A very flexible mechanism for dynamic memory management is provided by pointers.

# Static Array-Based Stacks

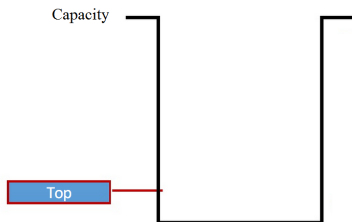
The idea is to store a stack in a fixed-size static array for simple implementation.

```
1 struct _Stack {  
2     <DataType> data [CAPACITY];  
3     int top;  
4 };  
5 typedef struct _Stack Stack;
```

# Static Array-Based Stacks

- `init()`: this function allows for creating an empty stack.

```
1 void init(Stack *st) {  
2     // st must get malloc() in main()  
3     st->top = -1;  
4 }
```

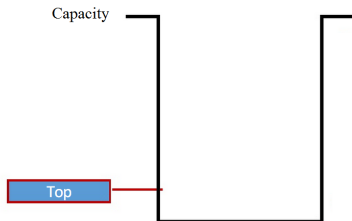




# Static Array-Based Stacks

- `init()`: this alternative function allows for creating an empty stack.

```
1 Stack * init(Stack *st) {  
2     st = (Stack*) malloc(sizeof(Stack));  
3     st->top = -1;  
4     return st;  
5 }
```



# Static Array-Based Stacks

- isEmpty(): this action allows checking if a stack is empty.

```
1 int isEmpty(Stack st){  
2     return (st.top < 0);  
3 }
```

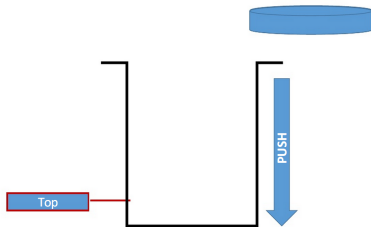
- size(): this function returns the stack size.

```
1 int size(Stack st){  
2     return st.top + 1;  
3 }
```

# Static Array-Based Stacks

- `push()`: this function allows to add a new item into a stack.

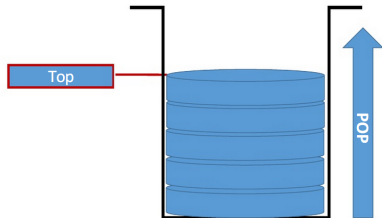
```
1 void push(Stack *s, int val){  
2     if (isFull(*s))  
3         printf('Stack is full!');  
4     else{  
5         s->top ++;  
6         s->data[s->top] = val;  
7     }  
8 }
```



# Static Array-Based Stacks

- `pop()`: this function allows to remove the top item from a stack,

```
1 void pop(Stack *s){  
2     if (isEmpty(*s))  
3         printf('Stack empty! ');\br/>4     else{  
5         s->top --;  
6     }  
7 }
```



# Dynamic Array-Based Stacks

The idea is to perform the stack implementation with a dynamic array.

```
1 struct _Stack {  
2     int top;  
3     int capacity  
4     int *data;  
5 };  
6 typedef struct _Stack Stack;
```

# Dynamic Array-Based Stacks

- `init()`: this function allows to create an empty stack.

```
1 void init (Stack *s, int N) {  
2     // s gets malloc() in the main() function  
3     s->top = 0;  
4     s->capacity = N;  
5     s->array = (int *)malloc(s->capacity);  
6 }
```

# Array-Based Stacks

Array-based stack implementation:

## Pros

- Simple to understand and implement.
- Stack is asserted at the top without changing other elements.

## Cons

- Stack size has to be manipulated.

# Stack Implementation with Linked Lists

## Definition

In this implementation, each item is placed together with the link to the next item, resulting in a simple component called **a node**:

- A data part stores an element value of the stack.
- A next part contains a link (or pointer) that indicates the node's location containing the next element.

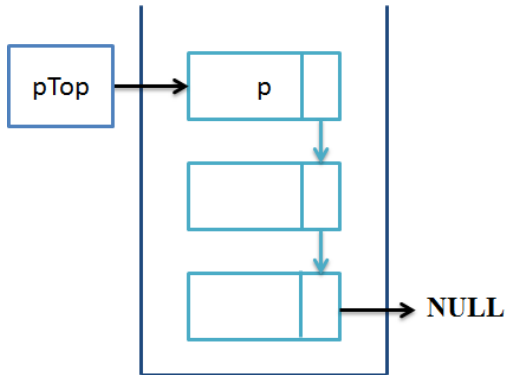


# Stack Implementation with Linked Lists

Implementing a stack as a linked list:

```
1 struct _Node{
2     int data;
3     struct _Node *next;
4 };
5 typedef struct _Node Node;
6 struct _Stack{
7     int size;
8     Node *pTop;
9 };
10 typedef struct _Stack Stack;
```

# Stacks



# Stacks Implementation with Linked Lists

Several basic operations are re-written to adapt to the new use of stack implementation.

```
1 void init(Stack *s){  
2     // s gets malloc() in the main() function  
3     s->size = 0;  
4     s->pTop = NULL;  
5 }  
6 int isEmpty(Stack st){  
7     return (st.size);  
8 }
```

# Stack Implementation with Linked Lists

Push operation is adapted to the new declaration:

```
1  int push(int newData, Stack *st){
2      Node *p;
3      pnew=(Node *)malloc(sizeof(Node));
4      if (p == NULL)
5          return 0;
6      pnew->data = newData;
7      //insert at the beginning of the list
8      pnew->next = st->pTop->next;
9      st->pTop = p;
10     st->size++;
11     return 1;
12 }
```

# Stack Implementation with Linked Lists

Pop operation is adapted to the new declaration:

```
1  int pop(stack *st){  
2      Node *p;  
3      if (isEmpty(*st))  
4          return 0; // fail to pop  
5      p = st->pTop;  
6      st->pTop = st->pTop->next;  
7      st->size --;  
8      free(p);  
9      return 1;  
10 }
```

# Complexity

Comparisons of complexity for different stack implementations

	push()	pop()	top()
Array-based Stacks	$O(1)$	$O(1)$	$O(1)$
Stacks with Linked List	$O(1)$	$O(1)$	$O(1)$

# Stack Implementation with Linked Lists

## Pros

- Stack implementation with linked lists is flexible to the size and memory.

## Cons

- If the top element is not used in the implementation, we have to traverse all the elements in the stack to find the top.

# Queues





# Queues

## General Definition

A **queue** is a line or sequence of people or vehicles awaiting their turn to be attended to or to proceed.

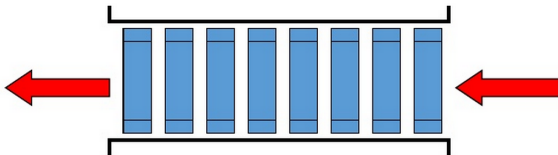
## Programming Definition

A **queue** is a container of objects (a linear collection) that are inserted and removed according to the first-in-first-out (FIFO) principle.

# Queues

## Principle

- A special data structure of lists used to store data in a particular order.
- Basic operations are done in both ends: insert at one end (back/rear) and remove at the other end (front/head).
- Access of items in a Queue is restricted; it follows the First In First Out (FIFO) order.



# Queues

## Queue Application

Typical uses of queues are in simulations and operating systems.

- Operating systems often maintain a queue of processes ready to execute or to wait for a particular event to occur.
- Anything that involves “waiting in line”: printing on the computer, seating customers at a restaurant, etc.

# Queues

Queues are an abstract data structure, and its implementation may offer the possible operations:

- `init()`: initialize an empty queue.
- `isEmpty()`: check if the queue is empty.
- `enqueue()`: add a new item at the back of the queue.
- `dequeue()`: remove the front item of the queue.

Other operations can be possibly defined:

- `length()`: return the size of a queue.
- `front()`: retrieve the front item of the queue.
- `isFull()`: check if the queue is full.
- `display()`: display the content of a queue.

# Queues

## Queues

There are several solutions to queue implementation using different declarations.

- **Static array-based queue**: arrays can be simply used to manipulate collections of items.
- **Dynamic array-based queue**: **malloc()** is capable of representing a queue.
- **Linked queue**: A very flexible mechanism for dynamic memory management is provided by pointers.

# Static Array-Based Queues

The idea is to store a queue in a fixed-size static array for simple implementation.

```
1 struct _Queue {  
2     int data[CAPACITY];  
3     int front, back;  
4     // front is optional  
5 };  
6 typedef struct _Queue Queue;
```

# Static Array-Based Queues

- `init()`: this function allows to create an empty queue.

```
1 void init(Queue *q){  
2     // q gets malloc() in the main() function  
3     q->front = 0;  
4     q->back = 0;  
5 }
```

- `isEmpty()`: this operation verifies that a queue is empty.

```
1 int isEmpty(Queue *q){  
2     return (q->back == 0);  
3 }
```

# Static Array-Based Queues

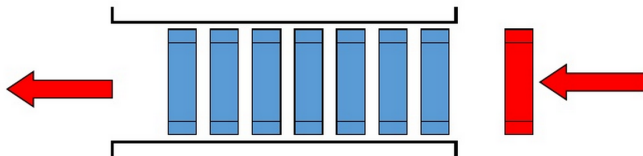
- `length()`: this operation returns the queue size.

```
1 int length(Queue *q){  
2     int l = q->back - q->front;  
3     return l;  
4 }
```



# Static Array-Based Queues

Due to the FIFO order, new items are inserted at the back of the queue. The function `enqueue()` allows to add a new item into a queue.



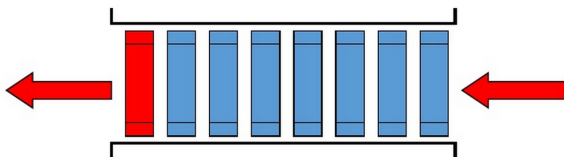
# Static Array-Based Queues

There are three cases to be proceeded for enqueue(): the queue is full, empty, and has at least one item.

```
1  int enqueue(Queue *q, <DataType> newData){
2      if (length(q) == CAPACITY){
3          printf("Queue is full!");
4          return 0;
5      }
6      if (isEmpty(q)){
7          q->val[0] = newData;
8      } else {
9          int idx = q->back;
10         q->val[idx] = newData;
11     }
12     q->back++;
13     return 1;
14 }
```

# Static Array-Based Queues

Due to the FIFO order, if we want to remove items from a queue, this action will proceed at the front of the queue. The function `dequeue()` asserts the deletion.



# Static Array-Based Queues

Two possible cases for `dequeue()` must be manipulated: when the queue is empty or not empty.

```
1  int dequeue(Queue *q){
2      if (isEmpty(q))
3          return 0;
4      else {
5          if (length(q) > 1){
6              for (int i = 1; i < length(q); i++)
7                  q->val[i-1] = q->val[i];
8          }
9          q->back = q->back - 1;
10     }
11     return 1;
12 }
```

# Dynamic Array-Based Queues

A dynamic array-based Queue improves the static array-based implementation.

```
1 struct _Queue{
2     int front , back ;
3     int capacity ;
4     int *val ;
5 };
6 typedef struct _Queue Queue ;
```

# Dynamic Array-Based Queues

- `init()`: this function allows to create an empty queue.

```
1 void init(Queue *q, int N){  
2     // q gets malloc() in the main() function  
3     q->back = 0;  
4     q->front = 0;  
5     q->capacity = N;  
6     q->val = (int *) malloc(q->capacity);  
7 }
```

# Array-Based Queues

Array-based queue implementation:

## Pros

- Simple to understand and implement.
- Enqueue is asserted at the back without shifting elements.

## Cons

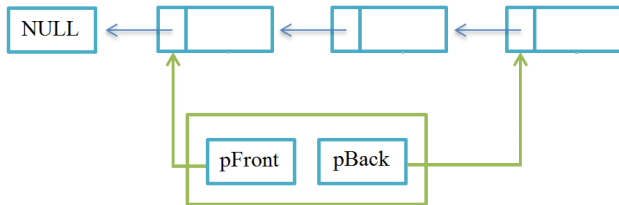
- Only the first element is accessible.
- All the elements have to be shifted ( $O(n)$  time for a queue with  $n$  elements) after a dequeue.

# Queue Implementation with Linked Lists

## Definition

In this implementation, each item is placed together with the link to the next item, resulting in a simple component called a **node**:

- A data part stores an element value of the queue.
- A next part contains a link (or pointer) that indicates the node's location containing the next element.
- The **front** element points to NULL.





# Queue Implementation with Linked Lists

Queue implementation using a linked list

```
1 typedef struct _Node {  
2     int data;  
3     struct _Node *next;  
4 } Node;  
5 typedef struct _Queue {  
6     Node *pFront, *pBack;  
7     int size;  
8 } Queue;
```

# Queue Implementation with Linked Lists

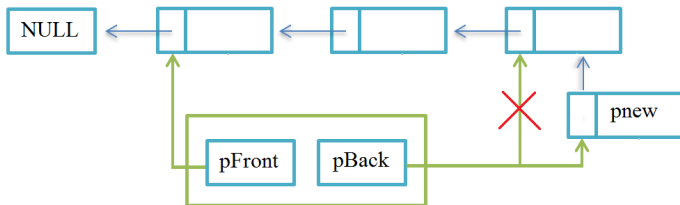
Several basic operations are re-written to adapt to the new use of queue implementation.

```
1 void init(Queue *q){  
2     // q gets malloc() in the main() function  
3     q->size = 0;  
4     q->pFront = q->pBack = NULL;  
5 }  
6 int isEmpty(Queue q){  
7     return (q->qFront == NULL);  
8 }
```

# Queue Implementation with Linked Lists

Enqueue operation:

- New items are enqueued at the back of the queue.
- The back node points to new items.



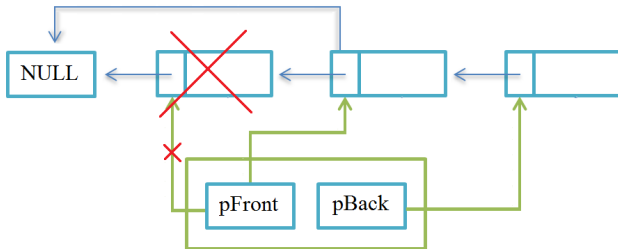
# Queue Implementation with Linked Lists

```
1 void enqueue (Queue *q, int _val){
2     Node *pnew = (Node *)malloc(sizeof(Node));
3     pnew->val = _val;
4     pnew->next = NULL;
5     if ( q->pFront == NULL)
6         q->pFront = q->pBack = pnew;
7     else{
8         pnew->next = q->pBack;
9         q->pBack = pnew;
10    }
11    q->size++;
12 }
```

# Queue Implementation with Linked Lists

Dequeue operation:

- The list should have at least one element.
- The front node points to the node that points to the first one.
- The pointer of this node points to NULL.



# Queue Implementation with Linked Lists

```
1 void dequeue(Queue *q){
2   if (isEmpty(*q))
3     return 0;
4   else {
5     if (q->size == 1){
6       q->pFront = q->pBack = NULL;
7       q->size --;
8     }
9     else{
10      Node *p = q->pBack;
11      while (p->next != q->pFront)
12        p = p->next;
13      q->pFront = p;
14      q->pFront->next = NULL;
15      q->size --;
16    }
17  }
18  return 1;
19 }
```

# Complexity

Comparisons of complexity for different queue implementations

	enqueue()	dequeue()	front()
Array-based Queues	$O(1)$	$O(n)$	$O(1)$
Queues wLL (with pFront, pBack)	$O(1)$	$O(n)$	$O(1)$
Queues wLL (without pFront)	$O(1)$	$O(n)$	$O(n)$
Queues wDLL (with pFront, pBack)	$O(1)$	$O(1)$	$O(1)$

LL - Singly Linked List; DLL - Doubly Linked List

# Queue Implementation with Linked Lists

## Pros

- Flexible to the size and memory.
- Enqueue can be done without shifting elements.

## Cons

- Have to traverse all the way to find the second element for the dequeue