Abstract Data Types II - Stacks & Queues

Doan Nhat Quang

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

Linked Lists

Linear Abstract Data Types:

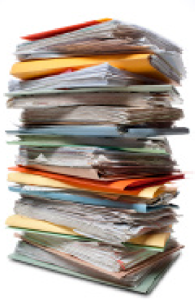
- Lists
  - Definition: is a collection with finite number of data objects (same type) and has a finite size.
  - List ADT: Array-based Lists, Linked Lists
  - List Operations
- Stacks
- Queues

Today Objectives

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

Plan

Stacks



Stack of books

Stack of coins

Stacks

General Definition

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

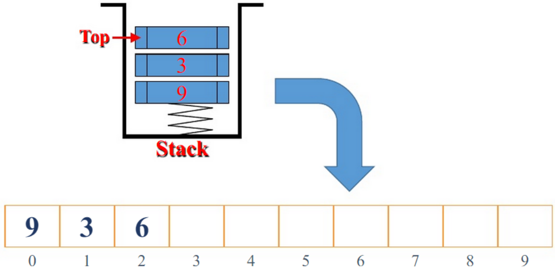
Programming Definition

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

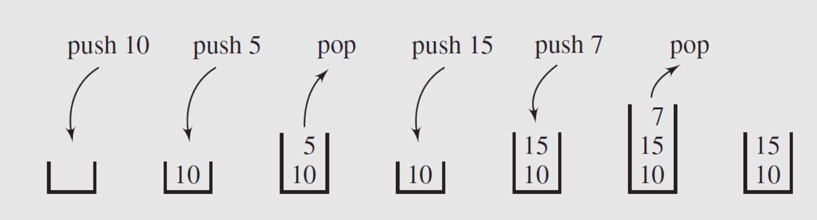
Stacks

ADT Stacks

- A linear data structure 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 follows 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 in which there are moves to be made (checkers, chess, sudoku).
- undo-mechanism in an editor

Stacks

Stack Application

Aerithmetic Expression:

- infix - operation between operands

  $$(3+5)*10$$

- prefix - operation before operands

  $$* + 3\ 5\ 10$$

- postfix - operation after operands

  $$3\ 5 + 10\ *$$

Stacks

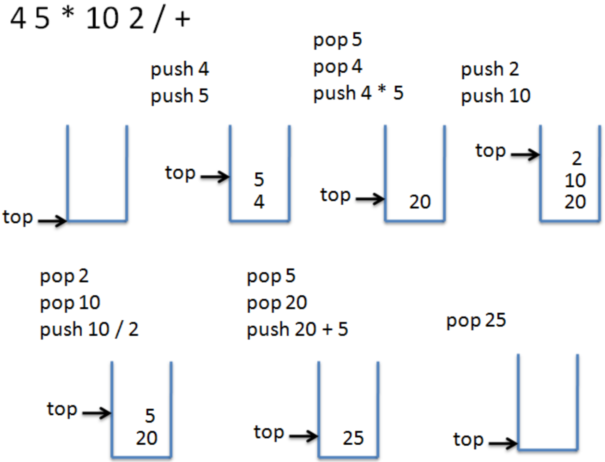
Stack Application

Aerithmetic 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 / +
push 4
push 5
pop 5
pop 4
push 4 * 5
push 2
push 10
top
top
5
4
top
20
top
2
10
20
pop 2
pop 10
push 10 / 2
pop 5
pop 20
push 20 + 5
pop 25
top
5
20
top
25
top

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

ADT Stacks

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

- 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

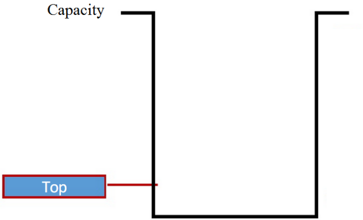
The idea is to store a stack in a fixed size static array for simple implementation.

```
struct _Stack {
    <DataType> data[CAPACITY];
    int top;
};
typedef struct _Stack Stack;
```

Static Array-Based Stacks

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

```c
void init (Stack *s) {
    l->top = -1;
}
```



Static Array-Based Stacks

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

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

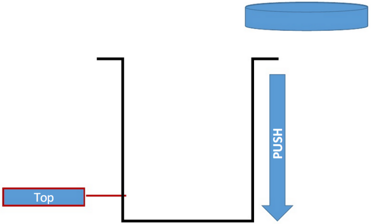
- size(): this function returns the stack size.

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

Static Array-Based Stacks

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

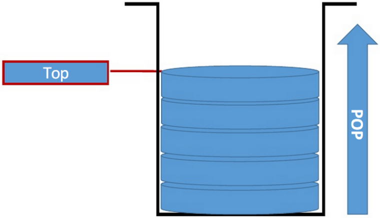
```
void push(Stack *s, int val){
 if (isFull(*s))
     printf(''Stack is full!'');
 else{
     s->top ++;
     s->data[s->top] = val;
 }
}
```



Static Array-Based Stacks

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

```
void pop(Stack *s){
 if (isEmpty(*s))
    printf(‘‘Stack empty!’’);
 else{
    s->top --;
 }
}
```



Dynamic Array-Based Stacks

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

```
struct _Stack {
    int top;
    int capacity
   <DataType> *data;
};
typedef  struct _Stack Stack;
```

Dynamic Array-Based Stacks

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

```
void init (Stack *s, int N) {
    s->top = 0;
    sl->capacity = N;
    s->array = malloc(sizeof(<DataType>)*s->capacity);
}
```

Array-Based Stacks

Array-based stack implementation:

Pros

- Simple to understand and to implement.
- Stack is assserted 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:

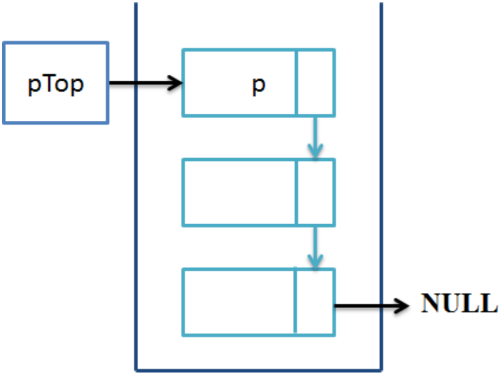
- A data part stores an element value of the stack.
- A next part contains a link (or pointer) that indicates the location of the node containing the next element.

Stack Implementation with Linked Lists

Implenenting a stack as a linked list:

```
struct _Node{
    <DataType> data;
    struct _Node *next;
};
typedef struct _Node Node;
struct _Stack{
    int size;
    Node *pTop;
};
typedef struct _Stack Stack;
```

Stacks



Stack Implementation with Linked Lists

Push operation is adapted to the new declaration:

```
int Push(<DataType> newData, Stack *st){
    Node *p;
    p=(Node *)malloc(sizeof(Node));
    if (p == NULL)
      return 0;
    p->data = newData;
    //insert at the begining of the list
    p->next = st->pTop->next;
    st->pTop = p;
    st->size++;
    return 1;
}
```

Stack Implementation with Linked Lists

Pop operation is adapted to the new declaration:

```
int Pop(stack *st){
    Node *p;
    if (isEmpty(*st))
        return 0; // fail to pop
    p = st->pTop;
    st->pTop = st->pTop->next;
    st->size--;
    free(p);
    return 1;
}
```

Complexity

Comparisons of complexity for different stack implementations

	push()	pop()	top()
Array-based List	O(1)	O(1)	O(1)
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, to find the top, we have to traverse all the elements in the stack.

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.

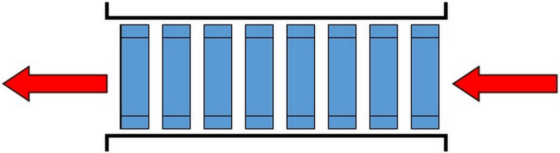
Programming Definition

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

ADT Queues

- A special data structure of lists used to store data in a particular order.
- Basic operations are done in both end: insert at one end (back/rear) and remove at the other end (front/head).
- Access of items in a Queue is restricted, it follows 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 that are ready to execute or that are waiting 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

ADT Queues

There are several solutions to the queue implementation using different declaration.

- 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.

```
struct _Queue {
    <DataType> data[CAPACITY];
    int front, back;
};
typedef struct _Queue Queue;
```

Static Array-Based Queues

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

```
void init(Queue *q){
    q->front = 0;
    q->back = 0;
}
```

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

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

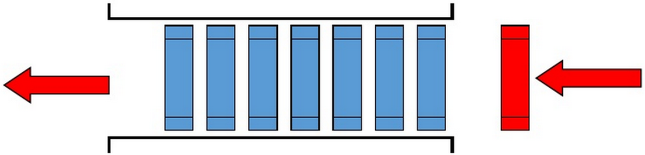
Static Array-Based Queues

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

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

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 new item into a queue.



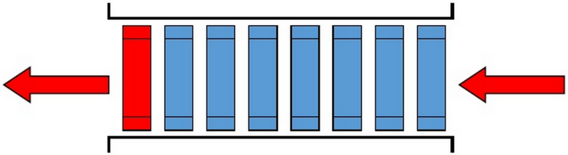
Static Array-Based Queues

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

```c
int enqueue(Queue *q, <DataType> newData){
    if (length(q) == CAPACITY){
        printf("Queue is full!");
        return 0;
    }
    if (isEmpty(q)){
        q->val[0] = newData;
    } else {
        int idx = q->back;
        q->val[idx] = newData;
    }
    q->back++;
    return 1;
}
```

Static Array-Based Queues

Due to the FIFO order, if we want to remove items from a queue, this action will be proceeded 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.

```
int dequeue(Queue *q){
    if (isEmpty(q))
        return 0;
    else {
        if (length(q) > 1){
            for (int i = 1; i <length(q); i++)
                q->val[i-1] = q->val[i];
        }
        q->back = q->back-1;
    }
    return 1;
}
```

Dynamic Array-Based Queues

Dynamic array-based Queue improves the static array-based implementation.

```
struct _Queue{
        int front,back;
        int capacity;
       <DataType> *val;
        };
typedef struct _Queue Queue;
```

Dynamic Array-Based Queues

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

```
void init(Queue *q, int N){
    q->back = 0;
    q->front = 0;
    q->capacity = N;
    q->val = (int *)malloc(q->capacity);
}
```

Array-Based Queues

Array-based queue implementation:

Pros

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

Cons

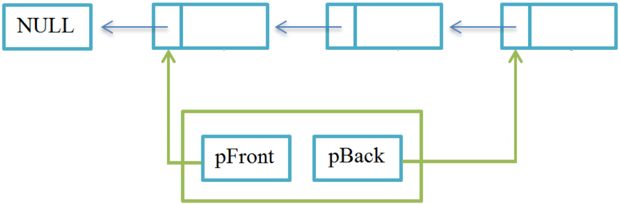
- Only the first element is accessible.
- All the elements have to be shifted ($O(n)$ time for a queue with n elements) after a dequeue.

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 location of the node containing the next element.
- The front element points to NULL.



Queue Implementation with Linked Lists

Queue implementation using a linked list

```
typedef struct _Node {
   <DataType> data;
    struct _Node *next;
} Node;
typedef struct _Queue {
      Node *pFront, *pBack;
      int size;
}Queue;
```

Queue Implementation with Linked Lists

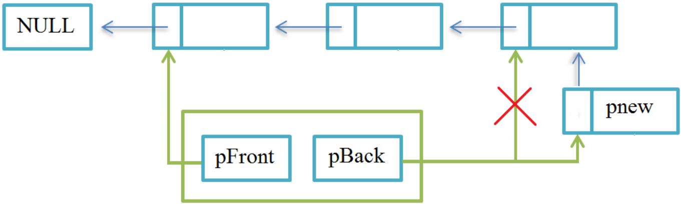
Several basic operations are re-written to adapt to the new use of queue implementation.

```
void init(Queue *q){
    q->size = 0;
    q->pFront = q->pBack = NULL;
}
int isEmpty(Queue q){
  return (q->qFront == NULL);
}
```

Queue Implementation with Linked Lists

Enqueue opertation:

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



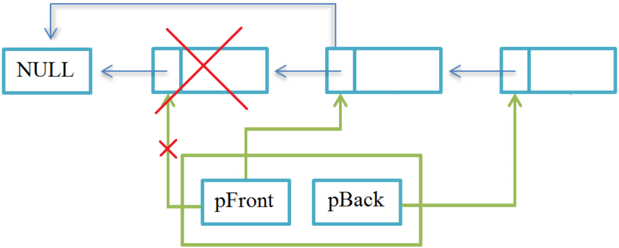
Queue Implementation with Linked Lists

```
void enqueue (Queue *q, <DataType> val){
  Node *p = (Node *)malloc(sizeof(Node));
  p->name = val;
  p->next = NULL;
  if ( q->pFront == NULL)
      q->pFront = q->pBack = p;

  else{
      p->next = q->pBack;
      q->pBack = p;

  }
  q->size++;
}
```

Queue Implementation with Linked Lists

Dequeue opteration:

- 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

```
void dequeue(Queue *q){
if (isEmpty(*q))
        return 0;
    else {
        if (q->size == 1){
            q->pFront = q->pBack = NULL;
            q->size --;
        }
        else{
            Passenger *p = q->pBack;
            while (p->next !=q->pFront)
                    p = p->next;
            q->pFront = p;
            q->pFront->next = NULL;
            q->size --;
        }
    }
    return 1;
}
```

Complexity

Comparisons of complexity for different queue implementations

	enqueue()	dequeue()	front()
Array-based List	O(1)	O(n)	O(1)
Linked List (with pFront, pBack)	O(1)	O(n)	O(1)
Linked List (without pFront)	O(1)	O(n)	O(n)

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