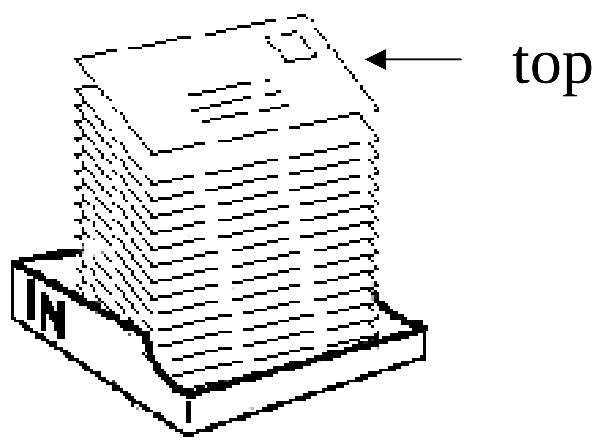
Stacks and Queues

Topics to be covered :

- What are ‘stacks’ and ‘queues’?
- Terminology
- How are they implemented?
- Example uses of stacks and queues

Stacks

A stack is a list in which all insertions and deletions are made at one end, called the top. The last element to be inserted into the stack will be the first to be removed. Thus stacks are sometimes referred to as ***L***ast ***I***n ***F***irst ***O***ut (LIFO) lists.

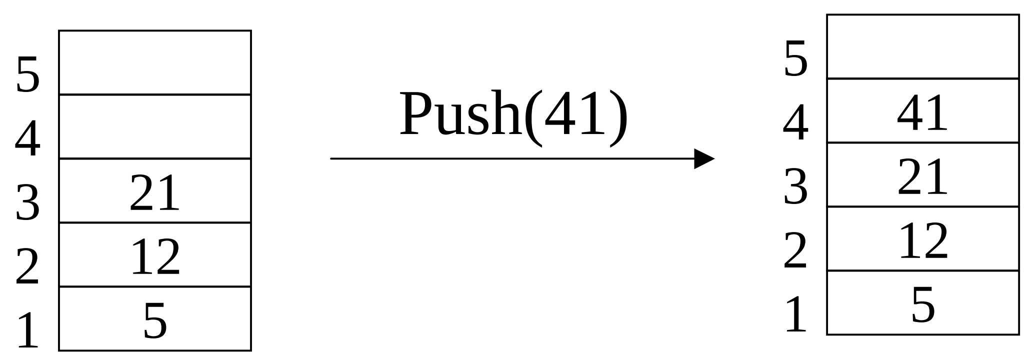


Stack Interface

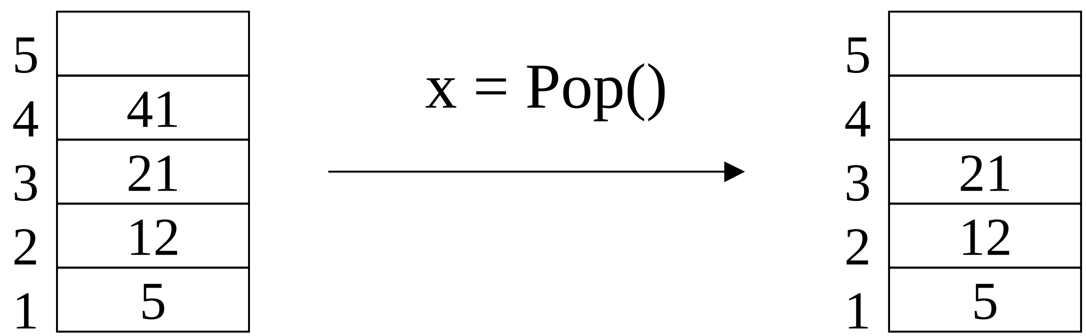
The following operations can be applied to a stack:

InitStack(Stack):	creates an empty stack
Push (Item):	pushes an item on the stack
Pop(Stack):	removes the first item from the stack
Top(Stack):	returns the first item from the stack w/o removing it
isEmpty(Stack):	returns true is the stack is empty

Push



Pop



Stack Implementation using Arrays (quick and dirty)

```
int StackArray[50]; // StackArray can contain
                    //  up to 50 numbers
int top=-1;         // index of the top element of the stack
                    // -1 used to indicate an empty stack
void Push(int elem)
{
   top++;
   StackArray[top] = elem;
}
int Pop()
{
   int elem = StackArray[top];
   top--;
   return elem;
}
```

Problem

The previous solution works on a fixed array. What if we want to have multiple stacks in a program? Copy code?

```
int StackArray2[50];// a second stack
int top2=-1;        // index of the top element of the stack

void Push2(int elem)
{
   top2++;
   StackArray[top2] = elem;
}
int Pop(){
 …
}
```

Bad idea!

How do we make this more efficient?

Abstract Data Type

Definition:

- An Abstract Data Type is some sort of data together with a set of functions (interface) that operate on the data.

- Access is only allowed through that interface.

- Implementation details are ‘hidden’ from the user.

The Stack-ADT

```
#define STACKSIZE 50

struct Stack
{
   int item[STACKSIZE];
   int top;
};

void InitStack(Stack &st);
void Push(Stack &st, int elem);
int  Pop (Stack &st);
int  Top (Stack st);
bool isEmpty(Stack st);
```

stack.h

Stack specification

Only defines the interface!

Using the Stack ADT

```
#include "stack.h"

void main()
{
   Stack st1, st2;  // declare 2 stack variables

   InitStack(st1);  // initialise them
   InitStack(st2);

   Push(st1, 13);   // push 13 onto the first stack
   Push(st2, 32);   // push 32 onto the second stack

   int i = Pop(st2); // now popping st2 into i
   int j = Top(st1); // returns the top of st1 to j
                     // without removing  element
};
```

Application of Stacks

e.g.
Evaluation of arithmetic expressions:
Usually, arithmetic expressions are written in *infix* notation, e.g.

A+B*C

An expression can as well be written in *postfix* notation (also called *reverse polish notation*):

A+B	becomes	AB+
A*C	becomes	AC*
A+B*C	becomes	ABC*+
(A+B)*C	becomes	AB+C*

Evaluating expressions

Given an expression in postfix notation. Using a stack they can be evaluated as follows:

- Scan the expression from left to right

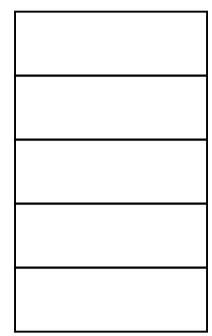
- When a value (operand) is encountered, push it on the stack

- When an operator is encountered, the first and second element from the stack are popped and the operator is applied

- The result is pushed on the stack

Evaluating Expressions (2)

Example: 7 1 3 + - 4 *



Stack

Another Stack Example

- Are stacks only useful for making sense of postfix notation expressions?

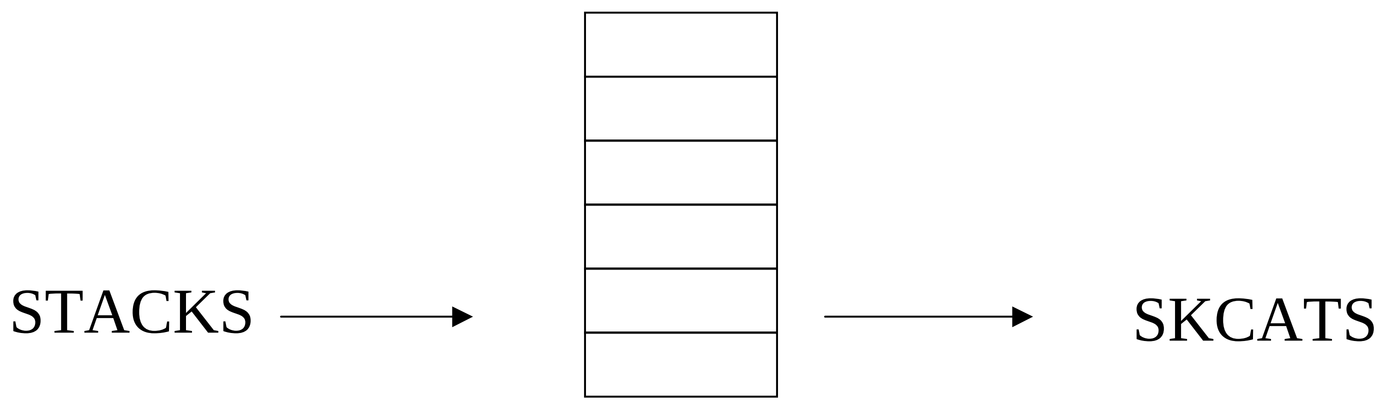
- Not so, Stacks have many uses!

- Another e.g. : Reversing word order

STACKS → SKCATS

- Simply push each letter onto the stack, then pop them back off again and hey presto!

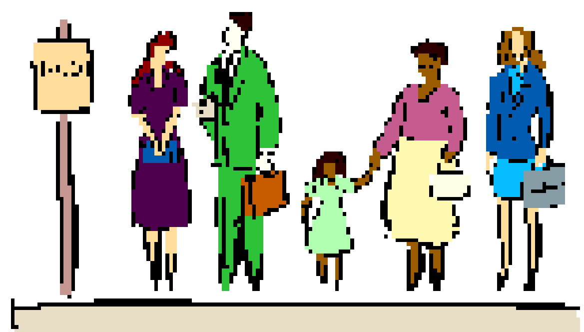
Another Stack Example(2)



Queues

Definition:

A Queue is an ordered collection of items from which items may be deleted at one end (called the *front* of the queue) and into which items may be inserted at the other end (the *rear* of the queue).



Queue Interface

The following operations can be applied to a queue:

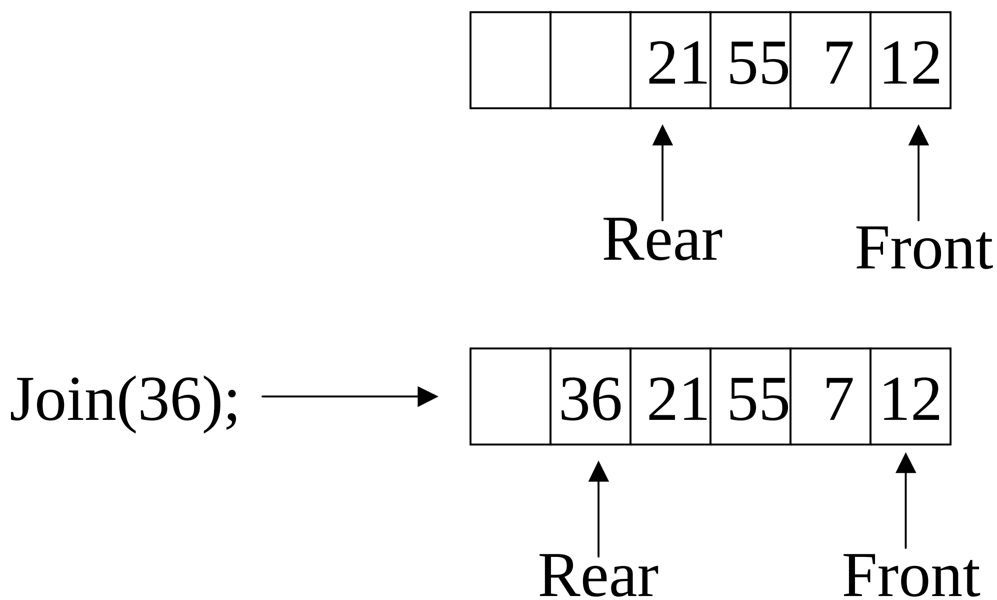
InitQueue(Queue): creates an empty queue

Join (Item): inserts an item to the rear of the queue

Leave(Queue): removes an item from the front of the queue

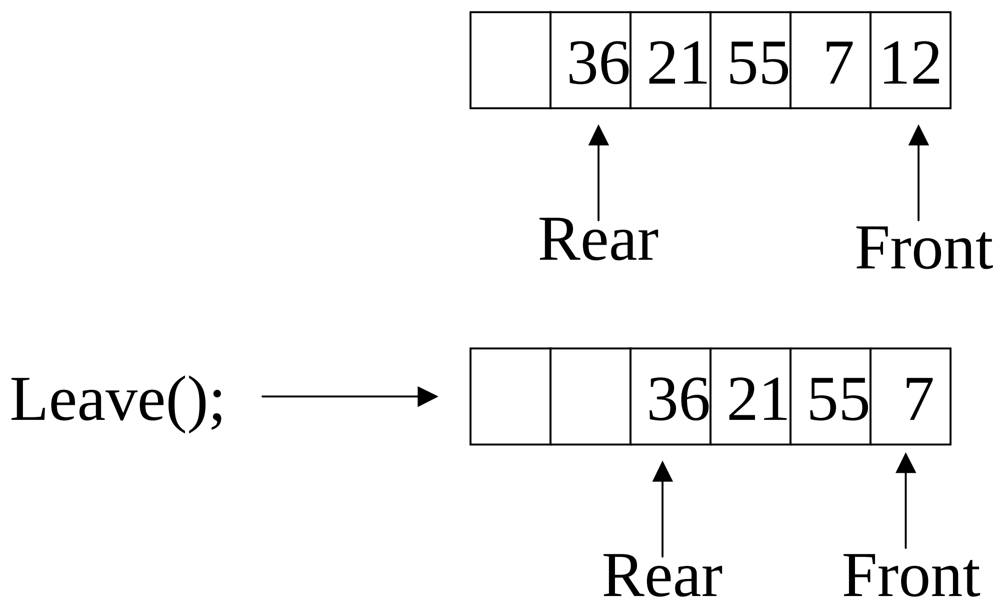
isEmpty(Queue): returns true is the queue is empty

Queues (FIFO-lists)



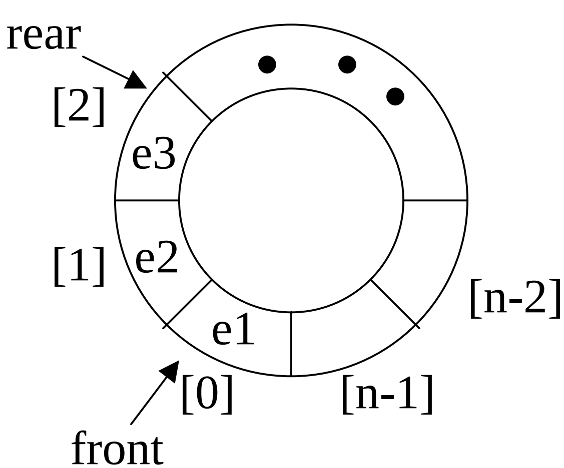
Elements can <u>only</u> be added to the rear of the queue and removed from the front of the queue.

Queues (contd.)



Implementation of Queues

Removing an element from the queue is an expensive operation because all remaining elements have to be moved by one position. A more efficient implementation is obtained if we consider the array as being 'circular':



Problem:
How do we know if queue is full/empty?

Joining the Queue

Initially, the queue is empty, i.e. `front == rear`. If we add an element to the queue we

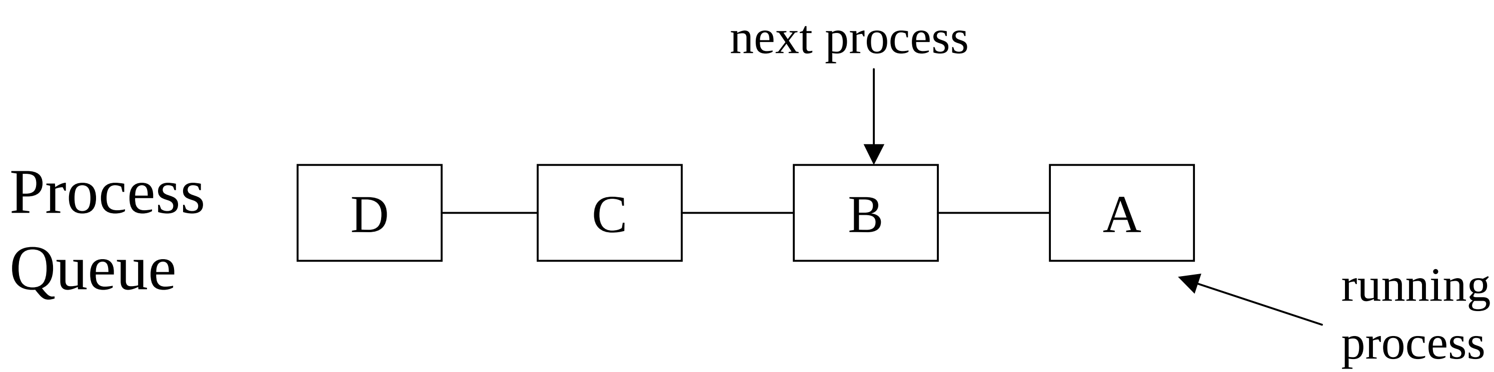
1) check if the queue is not full
2) store the element at the position indicated by `rear`
3) increase `rear` by one, wrap around if necessary (in this case `rear` always points to the last item in the queue – the rear item)

Adding one element:

```
  if (rear == QSIZE -1)
       rear = 0;
   else
       rear = rear+1;
       add an element to the queue
```

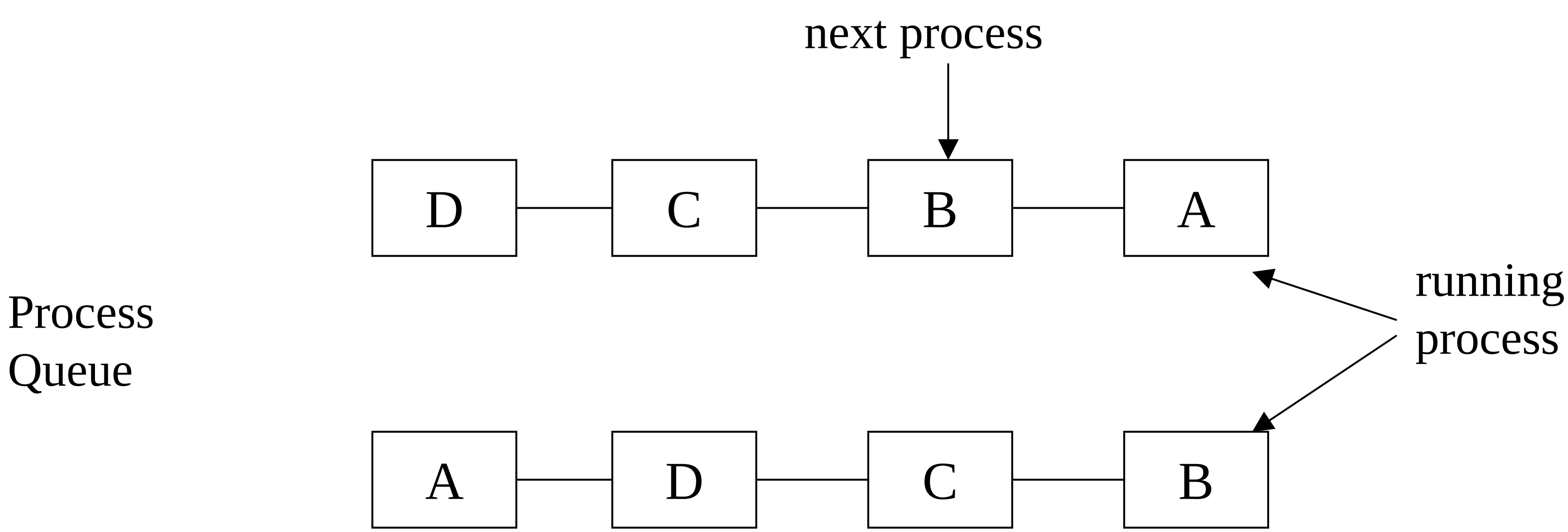
Application of Queues

In a multitasking operating system, the CPU time is shared between multiple processes. At a given time, only one process is running, all the others are 'sleeping'. The CPU time is administered by the scheduler. The scheduler keeps all current processes in a queue with the active process at the front of the queue.



Round-Robin Scheduling

Every process is granted a specific amount of CPU time, its 'quantum'. If the process is still running after its quantum run out, it is suspended and put towards the end of the queue.



The Queue-ADT

```
#define QSIZE 50

struct Queue
{
   int items[QSIZE];
   int rear;
   int front;
};

void InitQueue(Queue &q);
void Join(Queue &q, int elem);
int  Leave(Queue &q);
bool isEmpty(Queue q);
```

queue.h