

MODULE 2: STACKS AND QUEUES

STACKS

DEFINITION

“A **stack** is an **ordered list** in which insertions (pushes) and deletions (pops) are made at one end called the **top**.”

Given a stack $S = (a_0, \dots, a_{n-1})$, where a_0 is the bottom element, a_{n-1} is the top element, and a_i is on top of element a_{i-1} , $0 < i < n$.

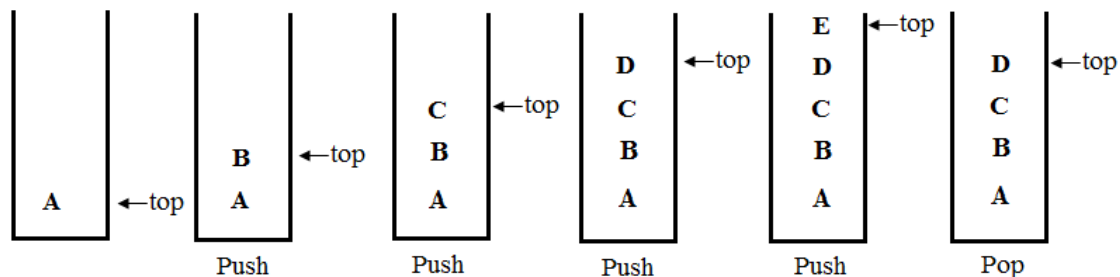


Figure: Inserting and deleting elements in a stack

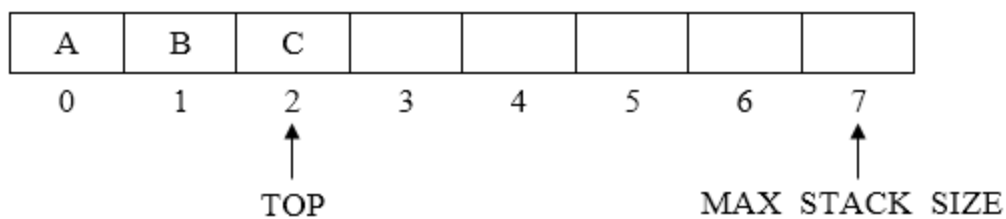
As shown in above figure, the elements are added in the stack in the order A, B, C, D, E, then **E** is the first element that is deleted from the stack and the last element is deleted from stack is **A**. Figure illustrates this sequence of operations.

Since the last element inserted into a stack is the first element removed, a stack is also known as a **Last-In-First-Out (LIFO)** list.

ARRAY REPRESENTATION OF STACKS

- Stacks may be represented in the computer in various ways such as one-way linked list (Singly linked list) or linear array.
- Stacks are maintained by the two variables such as TOP and MAX_STACK_SIZE.
- TOP which contains the location of the top element in the stack. If TOP = -1, then it indicates stack is empty.
- MAX_STACK_SIZE which gives maximum number of elements that can be stored in stack.

Stack can be represented using linear array as shown below



STACK OPERATIONS

Implementation of the stack operations as follows.

1. Stack Create

```
Stack CreateS(maxStackSize )::=
    #define MAX_STACK_SIZE 100 /* maximum stack size*/
    typedef struct
    {
        int key;
        /* other fields */
    } element;
    element stack[MAX_STACK_SIZE];
    int top = -1;
```

The **element** which is used to insert or delete is specified as a structure that consists of only a **key** field.

2. Boolean IsEmpty(Stack)::= top < 0;

3. Boolean IsFull(Stack)::= top >= MAX_STACK_SIZE-1;

The **IsEmpty** and **IsFull** operations are simple, and is implemented directly in the program push and pop functions. Each of these functions assumes that the variables **stack** and **top** are global.

4. Push()

Function **push** checks whether stack is full. If it is, it calls stackFull(), which prints an error message and terminates execution. When the stack is not full, increment top and assign item to stack [top].

```
void push(element item)
{ /* add an item to the global stack */
    if (top >= MAX_STACK_SIZE-1)
        stackFull();
    stack[++top] = item;
}
```

5. Pop()

Deleting an element from the stack is called pop operation. The element is deleted only from the top of the stack and only one element is deleted at a time.

```
element pop ( )
{ /*delete and return the top element from the stack */
    if (top == -1)
        return stackEmpty(); /*returns an error key */
    return stack[top--];
}
```

6. stackFull()

The **stackFull** which prints an error message and terminates execution.

```
void stackFull()
{
    fprintf(stderr, "Stack is full, cannot add element");
    exit(EXIT_FAILURE);
}
```

STACKS USING DYNAMIC ARRAYS

The array is used to implement stack, but the bound (MAX_STACK_SIZE) should be known during compile time. The size of bound is impossible to alter during compilation hence this can be overcome by using dynamically allocated array for the elements and then increasing the size of array as needed.

Stack Operations using dynamic array

1. Stack CreateS()::=

```
typedef struct
{
    int key;      /* other fields */
} element;
element *stack;
MALLOC(stack, sizeof(*stack));
int capacity= 1;
int top= -1;
```
2. Boolean IsEmpty(Stack)::= $top < 0$;
3. Boolean IsFull(Stack)::= $top \geq capacity - 1$;

4. push()

Here the MAX_STACK_SIZE is replaced with **capacity**

```
void push(element item)
{
    /* add an item to the global stack */
    if (top >= capacity-1)
        stackFull();
    stack[++top] = item;
}
```

5. pop()

In this function, no changes are made.

```
element pop ( )
{
    /* delete and return the top element from the stack */
    if (top == -1)
        return stackEmpty(); /* returns an error key */
    return stack[top--];
}
```

6. stackFull()

The new code shown below, attempts to increase the **capacity** of the array **stack** so that new element can be added into the stack. Before increasing the capacity of an array, decide what the new capacity should be.

In array doubling, array capacity is doubled whenever it becomes necessary to increase the capacity of an array.

```
void stackFull()
{
    REALLOC (stack, 2*capacity*sizeof(*stack));
    capacity *= 2;
}
```

Stack full with array doubling

Analysis

In the **worst case**, the realloc function needs to allocate **2*capacity*sizeof (*stack)** bytes of memory and copy **capacity *sizeof (*stack))** bytes of memory from the old array into the new one. Under the assumptions that memory may be allocated in O(1) time and that a stack element can be copied in O(1) time, the time required by array doubling is O(capacity).

Initially, capacity is 1.

Suppose that, if all elements are pushed in stack and the capacity is $2k$ for some $k, k > 0$, then the total time spent over all array doublings is $O(\sum_{i=1}^k 2i) = O(2^{k+1}) = O(2^k)$.

Since the total number of pushes is more than $2k-1$, the total time spent in array doubling is $O(n)$, where n is the total number of pushes. Hence, even with the time spent on array doubling added in, the total run time of push over all n pushes is $O(n)$.

STACK APPLICATIONS: POLISH NOTATION

Expressions: It is sequence of operators and operands that reduces to a single value after evaluation is called an expression.

$$X = a / b - c + d * e - a * c$$

In above expression contains operators (+, -, /, *) operands (a, b, c, d, e).

Expression can be represented in in different format such as

- Prefix Expression or Polish notation
- Infix Expression
- Postfix Expression or Reverse Polish notation

Infix Expression: In this expression, the binary operator is placed in-between the operand. The expression can be parenthesized or un- parenthesized.

Example: $A + B$

Here, A & B are operands and $+$ is operand

Prefix or Polish Expression: In this expression, the operator appears before its operand.

Example: $+ A B$

Here, A & B are operands and $+$ is operand

Postfix or Reverse Polish Expression: In this expression, the operator appears after its operand.

Example: $A B +$

Here, A & B are operands and $+$ is operand

Precedence of the operators

The first problem with understanding the meaning of expressions and statements is finding out the order in which the operations are performed.

Example: assume that $a = 4, b = c = 2, d = e = 3$ in below expression

$$X = a / b - c + d * e - a * c$$

$$((4/2)-2) + (3*3)-(4*2)$$

$$= 0 + 9 - 8$$

$$= 1$$

OR

$$(4 / (2 - 2 + 3)) * (3 - 4) * 2$$

$$= (4/3) * (-1) * 2$$

$$= -2.66666$$

The first answer is picked most because division is carried out before subtraction, and multiplication before addition. If we wanted the second answer, write expression differently using parentheses to change the order of evaluation

X= ((a / (b – c + d)) * (e – a) * c

In C, there is a precedence hierarchy that determines the order in which operators are evaluated. Below figure contains the precedence hierarchy for C.

Token	Operator	Precedence	Associativity
() [] →	function call array element struct or union member	17	left-to-right
-- ++	Increment, Decrement	16	left-to-right
--++ ! ~ -+ & * sizeof	decrement, increment logical not one's complement unary minus or plus address or indirection size (in bytes)	15	right-to-left
(type)	type cast	14	right-to-left
* / %	Multiplicative	13	left-to-right
+ -	binary add or subtract	12	left-to-right
<< >>	shift	11	left-to-right
> >= < <=	relational	10	left-to-right
= = ! =	equality	9	left-to-right
&	Bitwise and	8	left-to-right
^	bitwise exclusive or	7	left-to-right
	Bitwise or	6	left-to-right
&&	logical and	5	left-to-right
	logical or	4	left-to-right
?:	conditional	3	right-to-left
= += -= /= *= %= <<= >>= &= ^= =	assignment	2	right -to-left
,	comma	1	left-to-right

- The operators are arranged from highest precedence to lowest. Operators with highest precedence are evaluated first.
- The associativity column indicates how to evaluate operators with the same precedence. For example, the multiplicative operators have left-to-right associativity. This means that the expression **a * b / c % d / e** is equivalent to **((((a * b) / c) % d) / e)**
- Parentheses are used to override precedence, and expressions are always evaluated from the innermost parenthesized expression first

INFIX TO POSTFIX CONVERSION

An algorithm to convert infix to a postfix expression as follows:

1. Fully parenthesize the expression.
2. Move all binary operators so that they replace their corresponding right parentheses.
3. Delete all parentheses.

Example: Infix expression: $a/b - c + d * e - a * c$

Fully parenthesized : $((((a/b) - c) + (d * e)) - a * c)$

: $a\ b / e - d\ e * + a\ c *$

Example [Parenthesized expression]: Parentheses make the translation process more difficult because the equivalent postfix expression will be parenthesis-free.

The expression $a * (b + c) * d$ which results $abc + * d *$ in postfix. Figure shows the translation process.

Token\	Stack			Top	Output
	[0]	[1]	[2]		
a				-1	a
*	*			0	a
(*	(1	a
b	*	(1	ab
+	*	(+	2	ab
c	*	(+	2	abc
)	*			0	abc+
*	*			0	abc +*
d	*			0	abc +*d
eos	*			0	abc +*d*

- The analysis of the examples suggests a precedence-based scheme for stacking and unstacking operators.
- The left parenthesis complicates matters because it behaves like a low-precedence operator when it is on the stack and a high-precedence one when it is not. It is placed in the stack whenever it is found in the expression, but it is unstacked only when its matching right parenthesis is found.
- There are two types of precedence, **in-stack precedence (isp)** and **incoming precedence (icp)**.

The declarations that establish the precedence's are:

/* isp and icp arrays-index is value of precedence lparen rparen, plus, minus, times, divide, mod, eos */

int isp[] = {0,19,12,12,13,13,13,0};

int icp[] = {20,19,12,12,13,13,13,0};

```
void postfix(void)
{
    char symbol;
    precedence token;
    int n = 0, top = 0; /* place eos on stack */
    stack[0] = eos;
    for (token = getToken(&symbol, &n); token != eos; token =
        getToken(&symbol, &n))
    {
        if (token == operand)
            printf("%c", symbol);
        else if (token == rparen)
        {
            while (stack[top] != lparen)
                printToken(pop());
            pop();
        }
        else{
            while(isp[stack[top]] >= icp[token])
                printToken(pop());
            push(token);
        }
    }
    while((token = pop()) != eos)
        printToken(token);
    printf("\n");
}
```

Program: Function to convert from infix to postfix

Analysis of postfix: Let **n** be the number of tokens in the expression. $\Theta(n)$ time is spent extracting tokens and outputting them. Time is spent in the **two while loops**, is $\Theta(n)$ as the number of tokens that get stacked and unstacked is linear in **n**. So, the complexity of function postfix is $\Theta(n)$.

EVALUATION OF POSTFIX EXPRESSION

- The evaluation process of postfix expression is simpler than the evaluation of infix expressions because there are no parentheses to consider.
- To evaluate an expression, make a single **left-to-right** scan of it. Place the operands on a stack until an operator is found. Then remove from the stack, the correct number of operands for the operator, perform the operation, and place the result back on the stack and continue this fashion until the end of the expression. We then remove the answer from the top of the stack.

```
int eval(void)
{
    precedence token;
    char symbol;
    int opl,op2, n=0;
    int top= -1;
    token = getToken(&symbol, &n);
    while(token! = eos)
    {
        if (token == operand)
            push(symbol-'0'); /* stack insert */
        else {
            op2 = pop(); /* stack delete */
            opl = pop();
            switch(token) {
                case plus:    push(opl+op2);
                             break;
                case minus:   push(opl-op2);
                             break;
                case times:   push(opl*op2);
                             break;
                case divide:  push(opl/op2);
                             break;
                case mod:     push(opl%op2);

            }
        }
        token = getToken(&symbol, &n);
    }
    return pop(); /* return result */
}
```

Program: Function to evaluate a postfix expression

```
precedence getToken(char *symbol, int *n)
{
    *symbol = expr[(*n)++];
    switch (*symbol)
    {
        case '(': return lparen;
        case ')': return rparen;
        case '+': return plus;
        case '-': return minus;
        case '/': return divide;
        case '*': return times;
        case '%': return mod;
        case ' ': return eos;
        default: return operand;
    }
}
```

Program: Function to get a token from the input string

- The function **eval ()** contains the code to evaluate a postfix expression. Since an operand (symbol) is initially a character, convert it into a single digit integer.
- To convert use the statement, **symbol-'0'**. The statement takes the ASCII value of **symbol** and subtracts the ASCII value of '0', which is 48, from it. For example, suppose **symbol = '1'**. The character '1' has an ASCII value of 49. Therefore, the statement **symbol-'0'** produces as result the number 1.
- The function **getToken()**, obtain tokens from the expression string. If the token is an operand, convert it to a number and add it to the stack. Otherwise remove two operands from the stack, perform the specified operation, and place the result back on the stack. When the end of expression is reached, remove the result from the stack.

RECURSION

A recursive procedure

Suppose P is a procedure containing either a Call statement to itself or a Call statement to a second procedure that may eventually result in a Call statement back to the original procedure P. Then P is called a recursive procedure. So that the program will not continue to run indefinitely, a recursive procedure must have the following two properties:

1. There must be certain criteria, called base criteria, for which the procedure does not call itself.
2. Each time the procedure does call itself (directly or indirectly), it must be closer to the base criteria.

Recursive procedure with these two properties is said to be well-defined.

A recursive function

A function is said to be recursively defined if the function definition refers to itself. A recursive function must have the following two properties:

1. There must be certain arguments, called **base values**, for which the function does not refer to itself.
2. Each time the function does refer to itself, the argument of the function must be closer to a **base value**

A recursive function with these two properties is also said to be well-defined.

Factorial Function

“The product of the positive integers from 1 to n, is called "n factorial" and is denoted by n!”

$$n! = 1 * 2 * 3 \dots (n - 2) * (n - 1) * n$$

It is also convenient to define $0! = 1$, so that the function is defined for all nonnegative integers.

Definition: (Factorial Function)

- a) If $n = 0$, then $n! = 1$.
- b) If $n > 0$, then $n! = n * (n - 1)!$

Observe that this definition of $n!$ is recursive, since it refers to itself when it uses $(n - 1)!$

- (a) The value of $n!$ is explicitly given when $n = 0$ (thus 0 is the base value)
- (b) The value of $n!$ for arbitrary n is defined in terms of a smaller value of n which is closer to the base value 0.

The following are two procedures that each calculate n factorial.

1. Using for loop: This procedure evaluates N! using an iterative loop process

Procedure: FACTORIAL (FACT, N)

This procedure calculates N! and returns the value in the variable FACT.

1. If $N = 0$, then: Set FACT: = 1, and Return.
2. Set FACT: = 1. [Initializes FACT for loop.]
3. Repeat for $K = 1$ to N .
Set FACT: = $K * \text{FACT}$.
[End of loop.]
4. Return.

2. Using recursive function: This is a recursive procedure, since it contains a call to itself

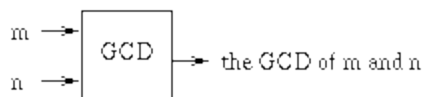
Procedure: FACTORIAL (FACT, N)

This procedure calculates N! and returns the value in the variable FACT.

1. If $N = 0$, then: Set FACT: = 1, and Return.
2. Call FACTORIAL (FACT, $N - 1$).
3. Set FACT: = $N * \text{FACT}$.
4. Return.

GCD

The **greatest common divisor** (GCD) of two integers m and n is the greatest integer that divides both m and n with no remainder.



$$\text{For } m \geq n \geq 0, \text{gcd}(m, n) = \begin{cases} n & \text{If } n \text{ divides } m \text{ with no minimum} \\ \text{gcd}(n, \text{remainder of } \frac{m}{n}) & \text{Otherwise} \end{cases}$$

Procedure: GCD (M, N)

1. If $(M \% N) = 0$, then set $\text{GCD} = N$ and RETURN
2. Call GCD (N, $M \% N$)
3. Return

Fibonacci Sequence

The Fibonacci sequence (usually denoted by F_0, F_1, F_2, \dots) is as follows:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55

That is, $F_0 = 0$ and $F_1 = 1$ and each succeeding term is the sum of the two preceding terms.

Definition: (Fibonacci Sequence)

- a) If $n = 0$ or $n = 1$, then $F_n = n$
- b) If $n > 1$, then $F_n = F_{n-2} + F_{n-1}$

Here

(a) The base values are 0 and 1

(b) The value of F_n is defined in terms of smaller values of n which are closer to the base values.

A procedure for finding the n^{th} term F_n of the Fibonacci sequence follows.

Procedure: FIBONACCI (FIB, N)

This procedure calculates F_N and returns the value in the first parameter FIB.

1. If $N = 0$ or $N = 1$, then: Set FIB: = N, and Return.
2. Call FIBONACCI (FIBA, $N - 2$).
3. Call FIBONACCI (FIBB, $N - 1$).
4. Set FIB: = FIBA + FIBB.
5. Return.

Tower of Hanoi

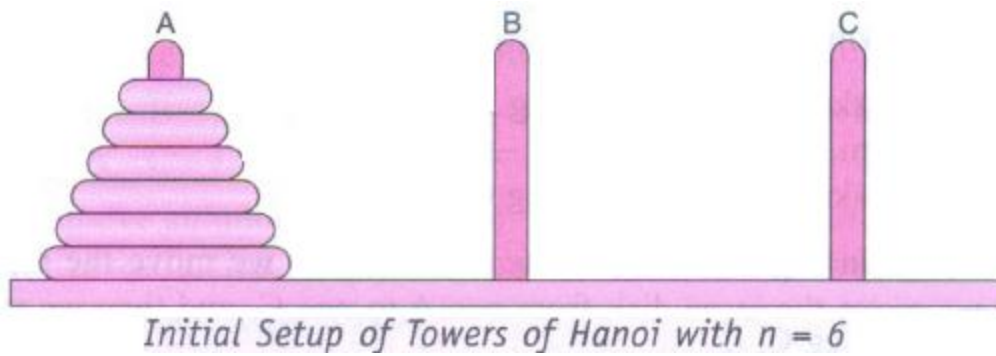
Problem description

Suppose three pegs, labeled A, B and C, are given, and suppose on peg A a finite number n of disks with decreasing size are placed.

The objective of the game is to move the disks from peg A to peg C using peg B as an auxiliary.

The rules of the game are as follows:

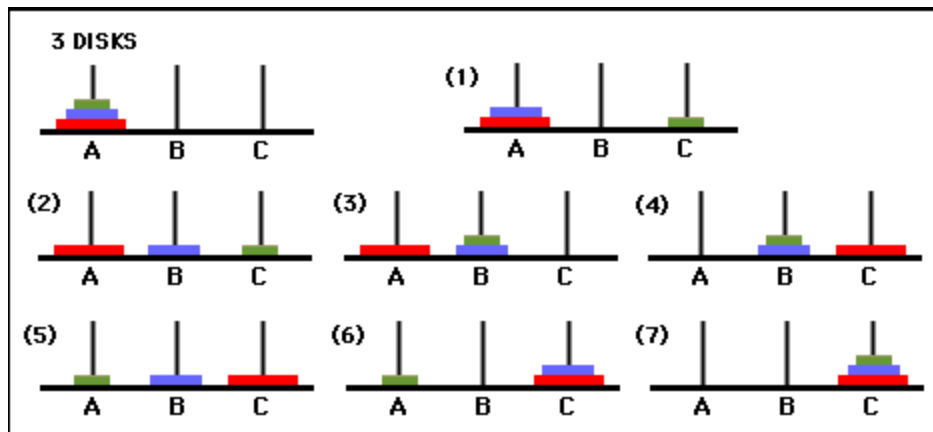
1. Only one disk may be moved at a time. Only the top disk on any peg may be moved to any other peg.
2. At no time can a larger disk be placed on a smaller disk.



We write $A \rightarrow B$ to denote the instruction "Move top disk from peg A to peg B"

Example: Towers of Hanoi problem for $n = 3$.

Solution: Observe that it consists of the following seven moves



1. Move top disk from peg A to peg C.
2. Move top disk from peg A to peg B.
3. Move top disk from peg C to peg B.
4. Move top disk from peg A to peg C.
5. Move top disk from peg B to peg A.
6. Move top disk from peg B to peg C.
7. Move top disk from peg A to peg C.

In other words,

$n=3$: $A \rightarrow C$, $A \rightarrow B$, $C \rightarrow B$, $A \rightarrow C$, $B \rightarrow A$, $B \rightarrow C$, $A \rightarrow C$

For completeness, the solution to the Towers of Hanoi problem for $n = 1$ and $n = 2$

$n=1$: $A \rightarrow C$

$n=2$: $A \rightarrow B$, $A \rightarrow C$, $B \rightarrow C$

The Towers of Hanoi problem for $n > 1$ disks may be reduced to the following sub-problems:

- (1) Move the top $n - 1$ disks from peg A to peg B
- (2) Move the top disk from peg A to peg C: $A \rightarrow C$.
- (3) Move the top $n - 1$ disks from peg B to peg C.

The general notation

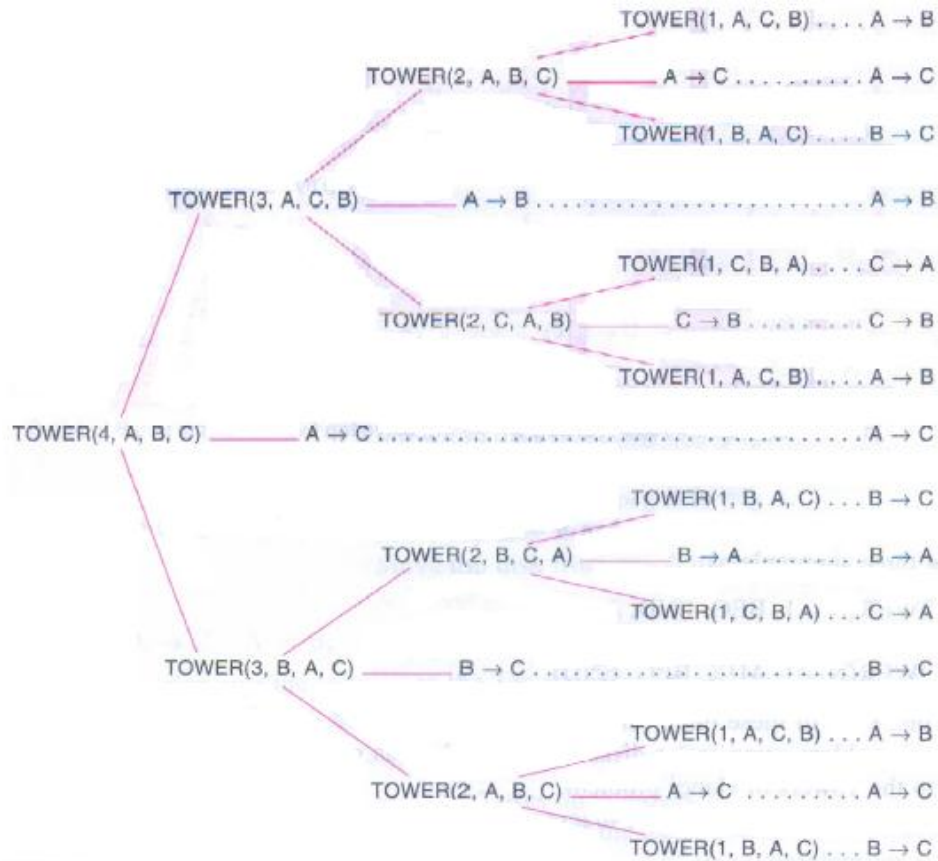
- TOWER (N, BEG, AUX, END) to denote a procedure which moves the top n disks from the initial peg BEG to the final peg END using the peg AUX as an auxiliary.
- When $n = 1$, the solution:
TOWER (1, BEG, AUX, END) consists of the single instruction $BEG \rightarrow END$
- When $n > 1$, the solution may be reduced to the solution of the following three sub-problems:
 - (a) TOWER (N - 1, BEG, END, AUX)
 - (b) TOWER (1, BEG, AUX, END) or $BEG \rightarrow END$
 - (c) TOWER (N - 1, AUX, BEG, END)

Procedure: TOWER (N, BEG, AUX, END)

This procedure gives a recursive solution to the Towers of Hanoi problem for N disks.

1. If $N=1$, then:
 - (a) Write: $BEG \rightarrow END$.
 - (b) Return.[End of If structure.]
 2. [Move $N - 1$ disks from peg BEG to peg AUX.]
Call TOWER (N - 1, BEG, END, AUX).
 3. Write: $BEG \rightarrow END$.
 4. [Move $N - 1$ disks from peg AUX to peg END.]
Call TOWER (N - 1, AUX, BEG, END).
 5. Return.
-

Example: Towers of Hanoi problem for $n = 4$



Ackermann function

The Ackermann function is a function with two arguments each of which can be assigned any nonnegative integer: 0, 1, 2,

Definition: (Ackermann Function)

- (a) If $m = 0$, then $A(m, n) = n + 1$.
- (b) If $m \neq 0$ but $n = 0$, then $A(m, n) = A(m - 1, 1)$
- (c) If $m \neq 0$ and $n \neq 0$, then $A(m, n) = A(m - 1, A(m, n - 1))$

QUEUES

DEFINITION

- “A queue is an **ordered list** in which insertions (additions, pushes) and deletions (removals and pops) take place at different ends.”
- The end at which new elements are added is called the **rear**, and that from which old elements are deleted is called the **front**.

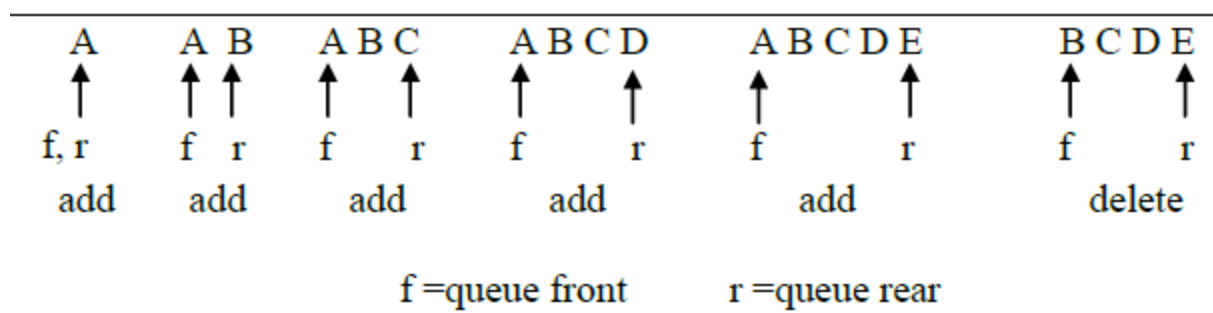
If the elements are inserted A, B, C, D and E in this order, then A is the first element deleted from the queue. Since the first element inserted into a queue is the first element removed, queues are also known as **First-In-First-Out (FIFO) lists**.

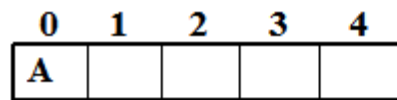
QUEUE REPRESENTATION USING ARRAY

- Queues may be represented by one-way lists or linear arrays.
- Queues will be maintained by a linear array QUEUE and two pointer variables:
FRONT-containing the location of the front element of the queue
REAR-containing the location of the rear element of the queue.
- The condition $\text{FRONT} = \text{NULL}$ will indicate that the queue is empty.

Figure indicates the way elements will be deleted from the queue and the way new elements will be added to the queue.

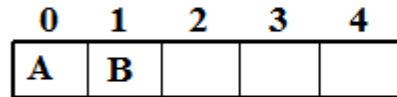
- Whenever an element is deleted from the queue, the value of FRONT is increased by 1; this can be implemented by the assignment $\text{FRONT} := \text{FRONT} + 1$
- When an element is added to the queue, the value of REAR is increased by 1; this can be implemented by the assignment $\text{REAR} := \text{REAR} + 1$





↑
f r

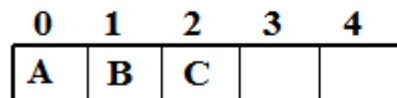
add



↑
f

↑
r

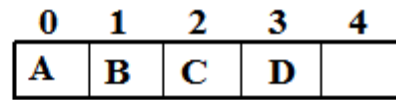
add



↑
f

↑
r

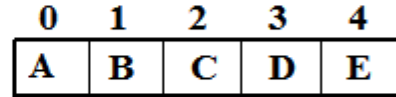
add



↑
f

↑
r

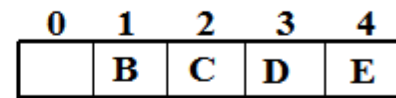
add



↑
f

↑
r

add



↑
f

↑
r

delete

QUEUE OPERATIONS

Implementation of the queue operations as follows.

1. Queue Create

```
Queue CreateQ(maxQueueSize) ::=
    #define MAX_QUEUE_SIZE 100      /* maximum queue size */
    typedef struct
    {
        int key;                      /* other fields */
    } element;
    element queue[MAX_QUEUE_SIZE];
    int rear = -1;
    int front = -1;
```

2. Boolean IsEmptyQ(queue) ::= front == rear

3. Boolean IsFullQ(queue) ::= rear == MAX_QUEUE_SIZE-1

In the queue, two variables are used which are **front** and **rear**. The queue increments **rear** in **addq()** and **front** in **delete()**. The function calls would be **addq (item);** and **item =delete();**

4. addq(item)

```

void addq(element item)
{
    /* add an item to the queue */
    if (rear == MAX_QUEUE_SIZE-1)
        queueFull();
    queue [++rear] = item;
}

```

Program: Add to a queue

5. deleteq()

```

element deleteq()
{
    /* remove element at the front of the queue */
    if (front == rear)
        return queueEmpty( );          /* return an error key */
    return queue[++front];
}

```

Program: Delete from a queue

6. queueFull()

The **queueFull** function which prints an error message and terminates execution

```

void queueFull()
{
    fprintf(stderr, "Queue is full, cannot add element");
    exit(EXIT_FAILURE);
}

```

Example: Job scheduling

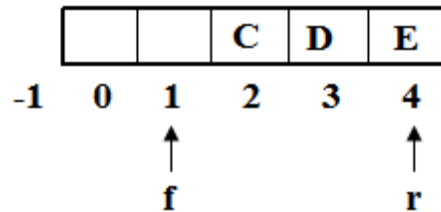
- Queues are frequently used in creation of a **job queue** by an operating system. If the operating system does not use priorities, then the jobs are processed in the order they enter the system.
- Figure illustrates how an operating system process jobs using a sequential representation for its queue.

front	rear	Q[0]	Q[1]	Q[2]	Q[3]	Comments
-1	-1					Queue is empty
-1	0	J1				Job 1 is added
-1	1	J1	J2			Job 2 is added
-1	2	J1	J2	J3		Job 3 is added
0	2		J2	J3		Job 1 is deleted
1	2			J3		Job 2 is deleted

Figure: Insertion and deletion from a sequential queue

Drawback of Queue

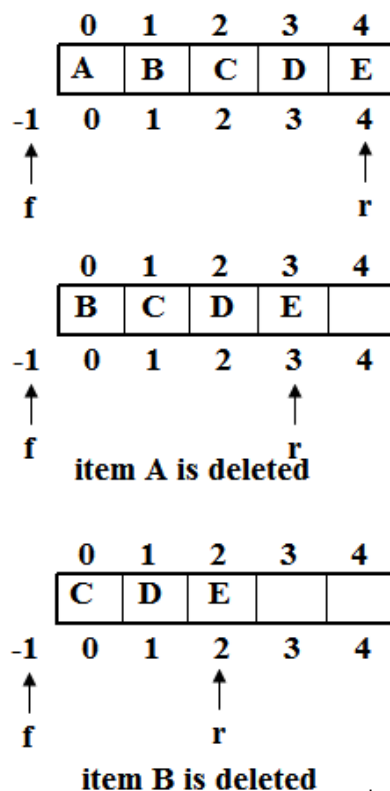
When item enters and deleted from the queue, the queue gradually shifts to the right as shown in figure.



In this above situation, when we try to insert another item, which shows that the **queue is full**. This means that the **rear** index equals to $\text{MAX_QUEUE_SIZE} - 1$. But even if the space is available at the front end, rear insertion cannot be done.

Overcome of Drawback using different methods**Method 1:**

- When an item is deleted from the queue, move the entire queue to the **left** so that the first element is again at `queue[0]` and front is at **-1**. It should also recalculate **rear** so that it is correctly positioned.
- Shifting an array is very time-consuming when there are many elements in queue & `queueFull` has worst case complexity of $O(\text{MAX_QUEUE_SIZE})$

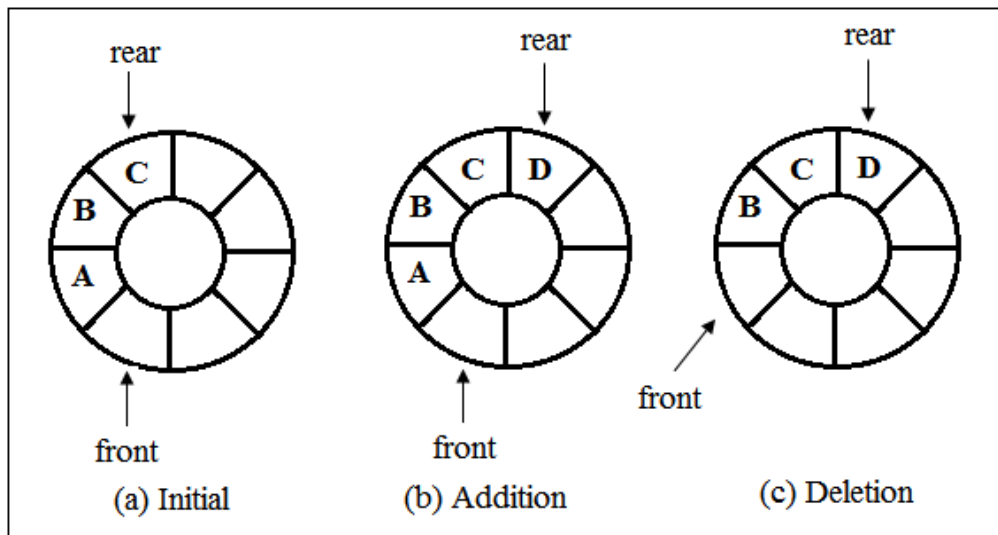


Method 2:**Circular Queue**

- It is “The queue which **wrap around** the end of the array.” The array positions are arranged in a circle.
- In this convention the variable **front** is changed. **front** variable points one position **counterclockwise** from the location of the front element in the queue. The convention for rear is unchanged.

CIRCULAR QUEUES

- It is “The queue which **wrap around** the end of the array.” The array positions are arranged in a circle as shown in figure.
- In this convention the variable **front** is changed. **front** variable points one position **counterclockwise** from the location of the front element in the queue. The convention for rear is unchanged.

**Implementation of Circular Queue Operations**

- When the array is viewed as a circle, each array position has a **next** and a **previous** position. The position next to **MAX-QUEUE-SIZE -1** is **0**, and the position that precedes **0** is **MAX-QUEUE-SIZE -1**.
- When the queue **rear** is at **MAX_QUEUE_SIZE-1**, the next element is inserted at position **0**.
- In circular queue, the variables **front** and **rear** are moved from their current position to the next position in clockwise direction. This may be done using code

```

if (rear == MAX_QUEUE_SIZE-1)
    rear = 0;
else rear++;

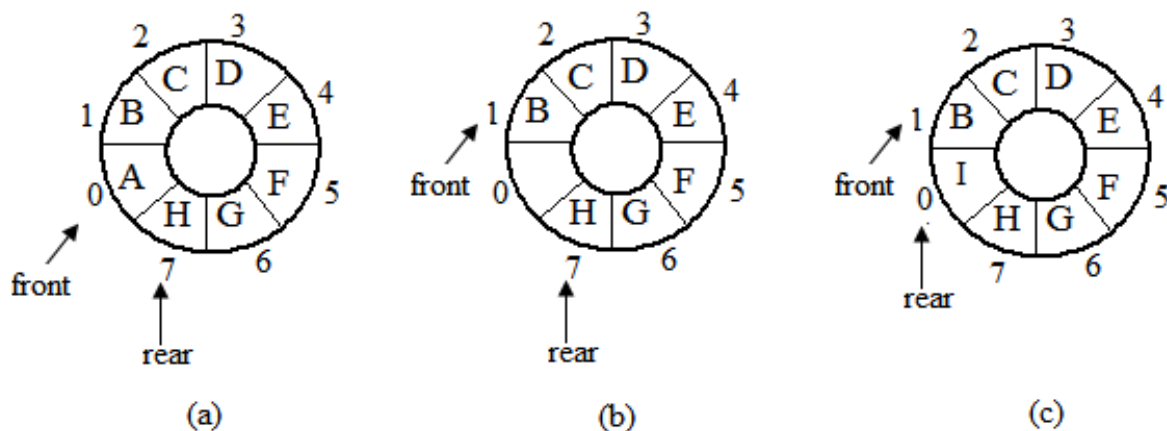
```

Addition & Deletion

- To add an element, increment **rear** one position clockwise and insert at the new position. Here the **MAX_QUEUE_SIZE** is 8 and if all 8 elements are added into queue and that can be represented in below figure (a).
- To delete an element, increment **front** one position clockwise. The element **A** is deleted from queue and if we perform 6 deletions from the queue of Figure (b) in this fashion, then queue becomes empty and that **front = rear**.
- If the element **I** is added into the queue as in figure (c), then **rear** needs to increment by 1 and the value of rear is **8**. Since queue is circular, the next position should be 0 instead of 8.

This can be done by using the modulus operator, which computes remainders.

$$(\text{rear} + 1) \% \text{MAX_QUEUE_SIZE}$$



```
void addq(element item)
{
    /* add an item to the queue */
    rear = (rear + 1) % MAX_QUEUE_SIZE;
    if (front == rear)
        queueFull();          /* print error and exit */
    queue [rear] = item;
}
```

Program: Add to a circular queue

```
element deleteq()
{
    /* remove front element from the queue */
    element item;
    if (front == rear)
        return queueEmpty( );          /* return an error key */
    front = (front+1)% MAX_QUEUE_SIZE;
    return queue[front];
}
```

Program: Delete from a circular queue

Note:

- When queue becomes empty, then **front = rear**. When the queue becomes full and **front = rear**. It is difficult to distinguish between an empty and a full queue.
- To avoid the resulting confusion, increase the capacity of a queue just before it becomes full.

CIRCULAR QUEUES USING DYNAMIC ARRAYS

- A dynamically allocated array is used to hold the queue elements. Let **capacity** be the number of positions in the array queue.
- To add an element to a **full queue**, first increase the size of this array using a function **realloc**. As with dynamically allocated stacks, **array doubling** is used.

Consider the **full queue** of figure (a). This figure shows a queue with seven elements in an array whose capacity is 8. A circular queue is flattened out the array as in Figure (b).

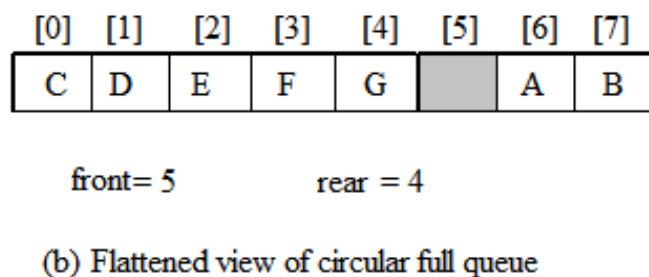
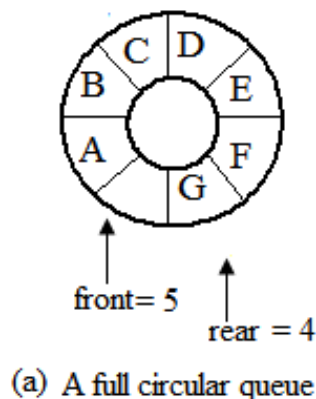
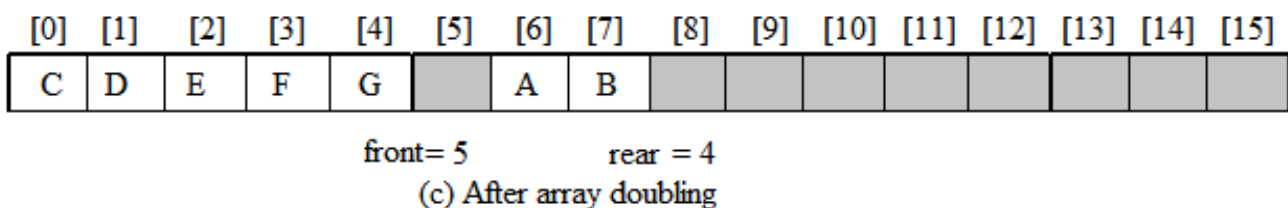
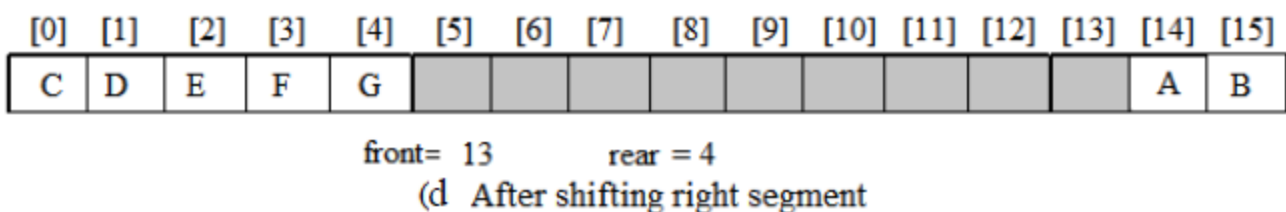


Figure (c) shows the array after array doubling by realloc



To get a proper circular queue configuration, slide the elements in the right segment (i.e., elements A and B) to the right end of the array as in figure (d)



To obtain the configuration as shown in figure (e), follow the steps

- 1) Create a new array **newQueue** of twice the capacity.
- 2) Copy the second segment (i.e., the elements queue [front +1] through queue [capacity-1]) to positions in **newQueue** beginning at 0.
- 3) Copy the first segment (i.e., the elements queue [0] through queue [rear]) to positions in newQueue beginning at **capacity – front – 1**.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
A	B	C	D	E	F	G									

front= 15 rear = 6
(e) Alternative configuration

Below program gives the code to add to a circular queue using a dynamically allocated array.

```
void addq( element item)
{
    /* add an item to the queue
    rear = (rear +1) % capacity;
    if(front == rear)
        queueFull( );      /* double capacity */
    queue[rear] = item;
}
```

Below program obtains the configuration of **figure (e)** and gives the code for queueFull. The function copy (a,b,c) copies elements from locations **a** through **b-1** to locations beginning at **c**.

```
void queueFull( )
{
    /* allocate an array with twice the capacity */
    element *newQueue;
    MALLOC ( newQueue, 2 * capacity * sizeof(* queue));
    /* copy from queue to newQueue */

    int start = ( front + ) % capacity;
    if ( start < 2)      /* no wrap around */
        copy( queue+start, queue+start+capacity-1,newQueue);
    else
    {
        /* queue wrap around */
        copy(queue, queue+capacity, newQueue);
        copy(queue, queue+rear+1, newQueue+capacity-start);
    }
}
```

```

/* switch to newQueue*/
front = 2*capacity - 1;
rear = capacity - 2;
capacity *= 2;
free(queue);
queue = newQueue;
}

```

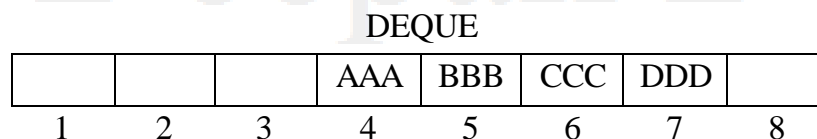
Program: queueFull

DEQUEUES OR DEQUE

A deque (double ended queue) is a linear list in which elements can be added or removed at either end but not in the middle.

Representation

- Deque is maintained by a circular array DEQUE with pointers LEFT and RIGHT, which point to the two ends of the deque.
- Figure shows deque with 4 elements maintained in an array with $N = 8$ memory locations.
- The condition $LEFT = NULL$ will be used to indicate that a deque is empty.



LEFT: 4

RIGHT: 7

There are two variations of a deque

1. **Input-restricted deque** is a deque which allows insertions at only one end of the list but allows deletions at both ends of the list
2. **Output-restricted deque** is a deque which allows deletions at only one end of the list but allows insertions at both ends of the list.

PRIORITY QUEUES

A priority queue is a collection of elements such that each element has been assigned a priority and such that the order in which elements are deleted and processed comes from the following rules:

- (1) An element of higher priority is processed before any element of lower priority.
- (2) Two elements with the same priority are processed according to the order in which they were added to the queue.

A prototype of a priority queue is a timesharing system: programs of high priority are processed first, and programs with the same priority form a standard queue.

Representation of a Priority Queue

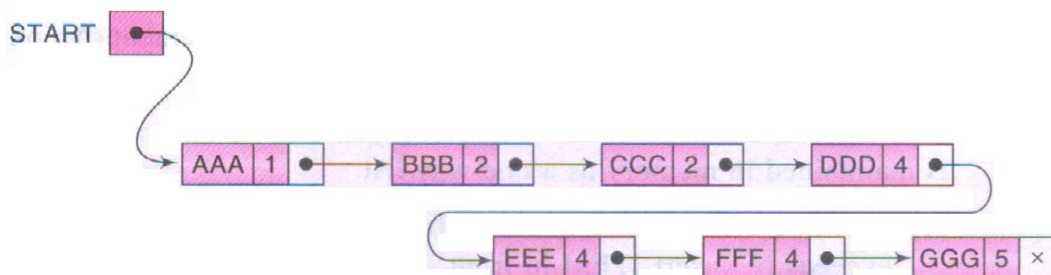
1. One-Way List Representation of a Priority Queue

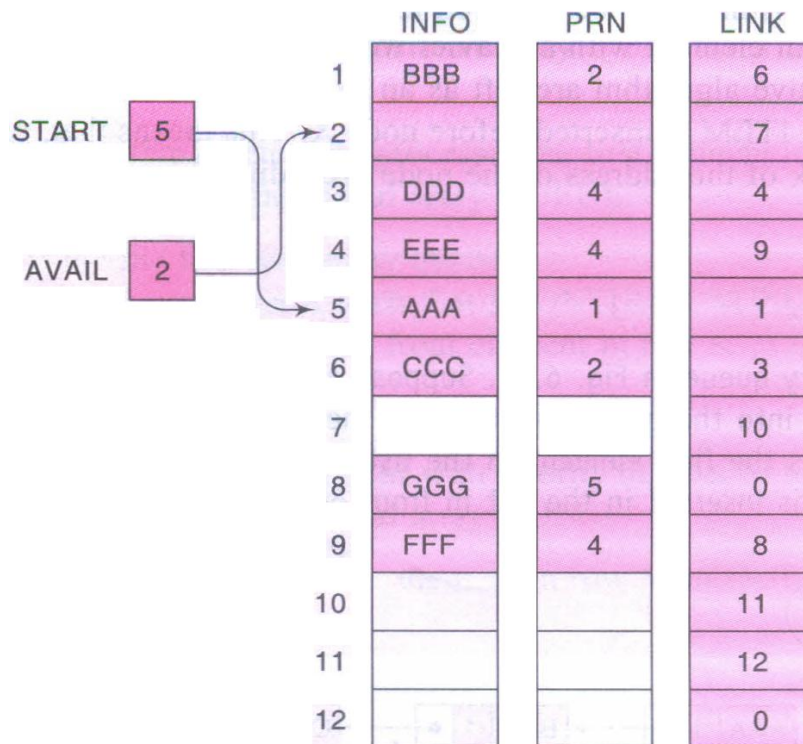
One way to maintain a priority queue in memory is by means of a one-way list, as follows:

1. Each node in the list will contain three items of information: an information field INFO, a priority number PRN and a link number LINK.
2. A node X precedes a node Y in the list
 - a. When X has higher priority than Y
 - b. When both have the same priority but X was added to the list before Y. This means that the order in the one-way list corresponds to the order of the priority queue.

Example:

- Below Figure shows the way the priority queue may appear in memory using linear arrays INFO, PRN and LINK with 7 elements.
- The diagram does not tell us whether BBB was added to the list before or after DDD. On the other hand, the diagram does tell us that BBB was inserted before CCC, because BBB and CCC have the same priority number and BBB appears before CCC in the list.





The main property of the one-way list representation of a priority queue is that the element in the queue that should be processed first always appears at the beginning of the one-way list. Accordingly, it is a very simple matter to delete and process an element from our priority queue.

Algorithm to delete and processes the first element in a priority queue

Algorithm: This algorithm deletes and processes the first element in a priority queue which appears in memory as a one-way list.

1. Set ITEM:= INFO[START] [This saves the data in the first node.]
2. Delete first node from the list.
3. Process ITEM.
4. Exit.

Algorithm to add an element to priority queue

Adding an element to priority queue is much more complicated than deleting an element from the queue, because we need to find the correct place to insert the element.

Algorithm: This algorithm adds an ITEM with priority number N to a priority queue which is maintained in memory as a one-way list.

1. Traverse the one-way list until finding a node X whose priority number exceeds N. Insert ITEM in front of node X.
2. If no such node is found, insert ITEM as the last element of the list.

The main difficulty in the algorithm comes from the fact that ITEM is inserted before node X. This means that, while traversing the list, one must also keep track of the address of the node preceding the node being accessed.

Example:

Consider the priority queue in Fig (a). Suppose an item XXX with priority number 2 is to be inserted into the queue. We traverse the list, comparing priority numbers.

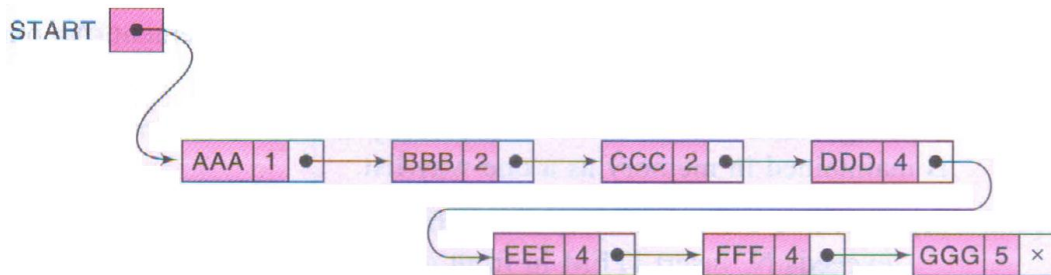
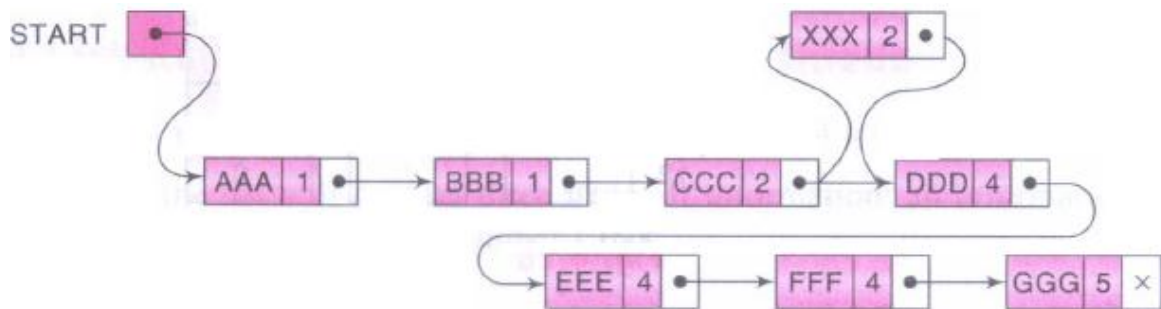


Fig (a)



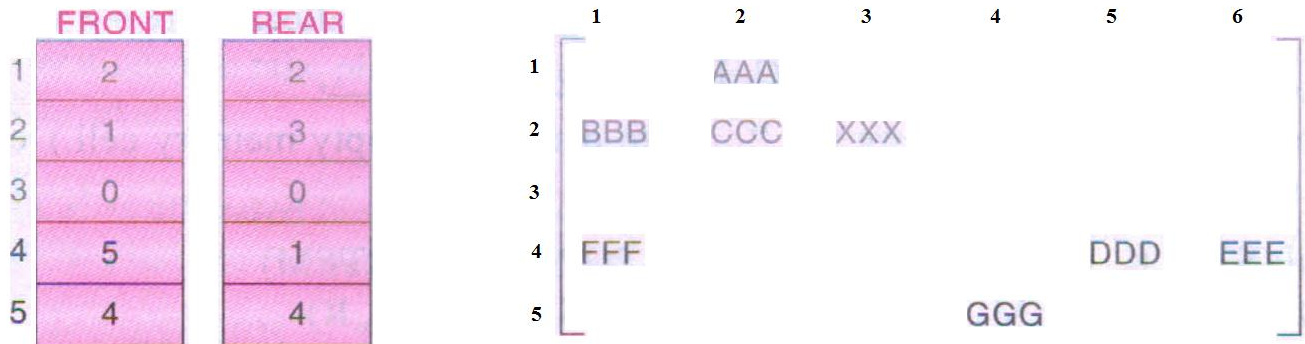
Fig(b)

Observe that DDD is the first element in the list whose priority number exceeds that of XXX. Hence XXX is inserted in the list in front of DDD, as pictured in Fig(b).

Observe that XXX comes after BBB and CCC, which have the same priority as XXX. Suppose now that an element is to be deleted from the queue. It will be AAA, the first element in the List. Assuming no other insertions, the next element to be deleted will be BBB, then CCC, then XXX, and so on.

Array Representation of a Priority Queue

- Another way to maintain a priority queue in memory is to use a separate queue for each level of priority (or for each priority number).
- Each such queue will appear in its own circular array and must have its own pair of pointers, FRONT and REAR.
- If each queue is allocated the same amount of space, a two-dimensional array QUEUE can be used instead of the linear arrays.



Observe that FRONT[K] and REAR[K] contain, respectively, the front and rear elements of row K of QUEUE, the row that maintains the queue of elements with priority number K.

The following are outlines or algorithms for deleting and inserting elements in a priority queue

Algorithm: This algorithm deletes and processes the first element in a priority queue maintained by a two-dimensional array QUEUE.

1. [Find the first non-empty queue.]
Find the smallest K such that FRONT[K] \neq NULL.
2. Delete and process the front element in row K of QUEUE.
3. Exit.

Algorithm: This algorithm adds an ITEM with priority number M to a priority queue maintained by a two-dimensional array QUEUE.

1. Insert ITEM as the rear element in row M of QUEUE.
2. Exit.

MULTIPLE STACKS AND QUEUES

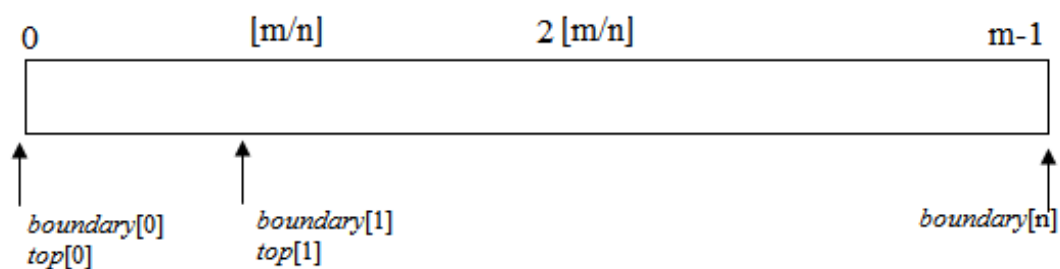
- In multiple stacks, we examine only **sequential mappings** of stacks into an array. The array is one dimensional which is **memory[MEMORY_SIZE]**. Assume n stacks are needed, and then divide the available memory into n segments. The array is divided in proportion if the expected sizes of the various stacks are known. Otherwise, divide the memory into equal segments.
- Assume that i refers to the stack number of one of the n stacks. To establish this stack, create indices for both the **bottom** and **top** positions of this stack. **boundary[i]** points to the position immediately to the left of the bottom element of stack i , **top[i]** points to the top element. Stack i is empty iff **boundary[i]=top[i]**.

The declarations are:

```
#define MEMORY_SIZE 100          /* size of memory */
#define MAX_STACKS 10           /* max number of stacks plus 1 */
element memory[MEMORY_SIZE];    /* global memory declaration */
int top [MAX_STACKS];
int boundary [MAX_STACKS] ;
int n;                          /*number of stacks entered by the user */
```

To divide the array into roughly equal segments

```
top[0] = boundary[0] = -1;
for (j= 1;j<n; j++)
    top[j] = boundary[j] = (MEMORY_SIZE / n) * j;
boundary[n] = MEMORY_SIZE - 1;
```



All stacks are empty and divided into roughly equal segments

Figure: Initial configuration for n stacks in *memory [m]*.

In the figure, n is the number of stacks entered by the user, $n < \text{MAX_STACKS}$, and $m = \text{MEMORY_SIZE}$. Stack i grow from **boundary[i] + 1** to **boundary [i + 1]** before it is full. A boundary for the last stack is needed, so set **boundary [n]** to **MEMORY_SIZE-1**.

Implementation of the add operation

```

void push(int i, element item)
{
    /* add an item to the ith stack */
    if (top[i] == boundary[i+1])
        stackFull(i);
    memory[++top[i]] = item;
}

```

Program: Add an item to the ith stack**Implementation of the delete operation**

```

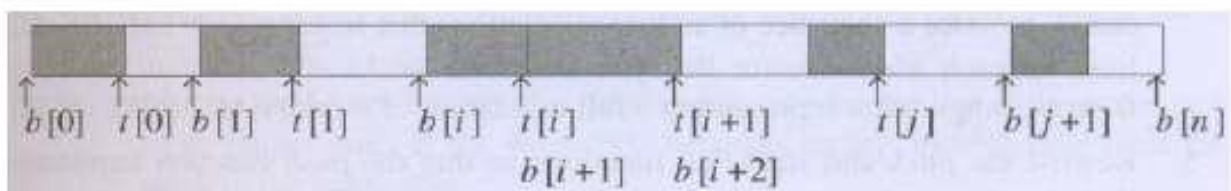
element pop(int i)
{
    /* remove top element from the ith stack */
    if (top[i] == boundary[i])
        return stackEmpty(i);
    return memory[top[i]--];
}

```

Program: Delete an item from the ith stack

The $\text{top}[i] == \text{boundary}[i+1]$ condition in push implies only that a particular stack ran out of memory, not that the entire memory is full. But still there may be a lot of unused space between other stacks in array memory as shown in Figure.

Therefore, create an error recovery function called **stackFull**, which determines if there is any free space in memory. If there is space available, it should shift the stacks so that space is allocated to the full stack.



b boundary , t=top

Method to design stackFull

- Determine the least, j , $i < j < n$, such that there is free space between stacks j and $j+1$. That is, $top[j] < boundary[j+1]$. If there is a j , then move stacks $i+1, i+2, \dots, j$ one position to the right (treating $memory[0]$ as leftmost and $memory[MEMORY_SIZE - 1]$ as rightmost). This creates a space between stacks i and $i+1$.
- If there is no j as in (1), then look to the left of stack i . Find the largest j such that $0 \leq j \leq i$ and there is space between stacks j and $j+1$ ie, $top[j] < boundary[j+1]$. If there is a j , then move stacks $j+1, j+2, \dots, i$ one space to the left. This also creates space between stacks i and $i+1$.
- If there is no j satisfying either condition (1) or condition (2), then all $MEMORY_SIZE$ spaces of memory are utilized and there is no free space. In this case *stackFull* terminates with an error message.

Deepak D