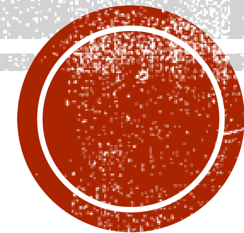
STACKS & QUEUES



STACKS & QUEUES

Two abstract data types

- Can be implemented using arrays or linked lists
- Restricted lists (only a small number of operations are performed on them)

Stacks

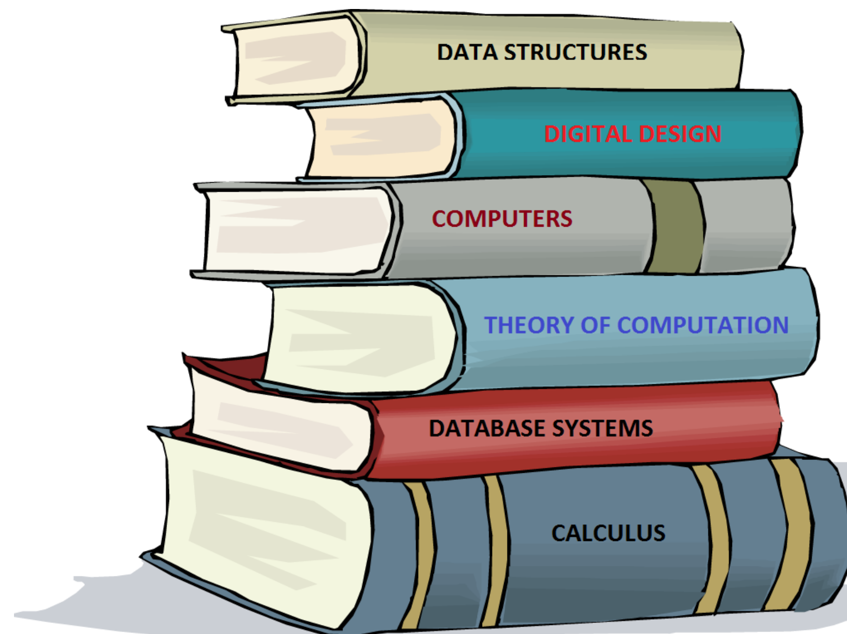
“Last In First Out” (LIFO)

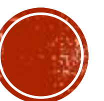
Queues

“First In First Out” (FIFO)

STACKS

A stack is an ordered lists in which insertions and deletions are made at one ende called the ***top***.





OPERATIONS ON STACKS

PUSH

Place an item on the stack

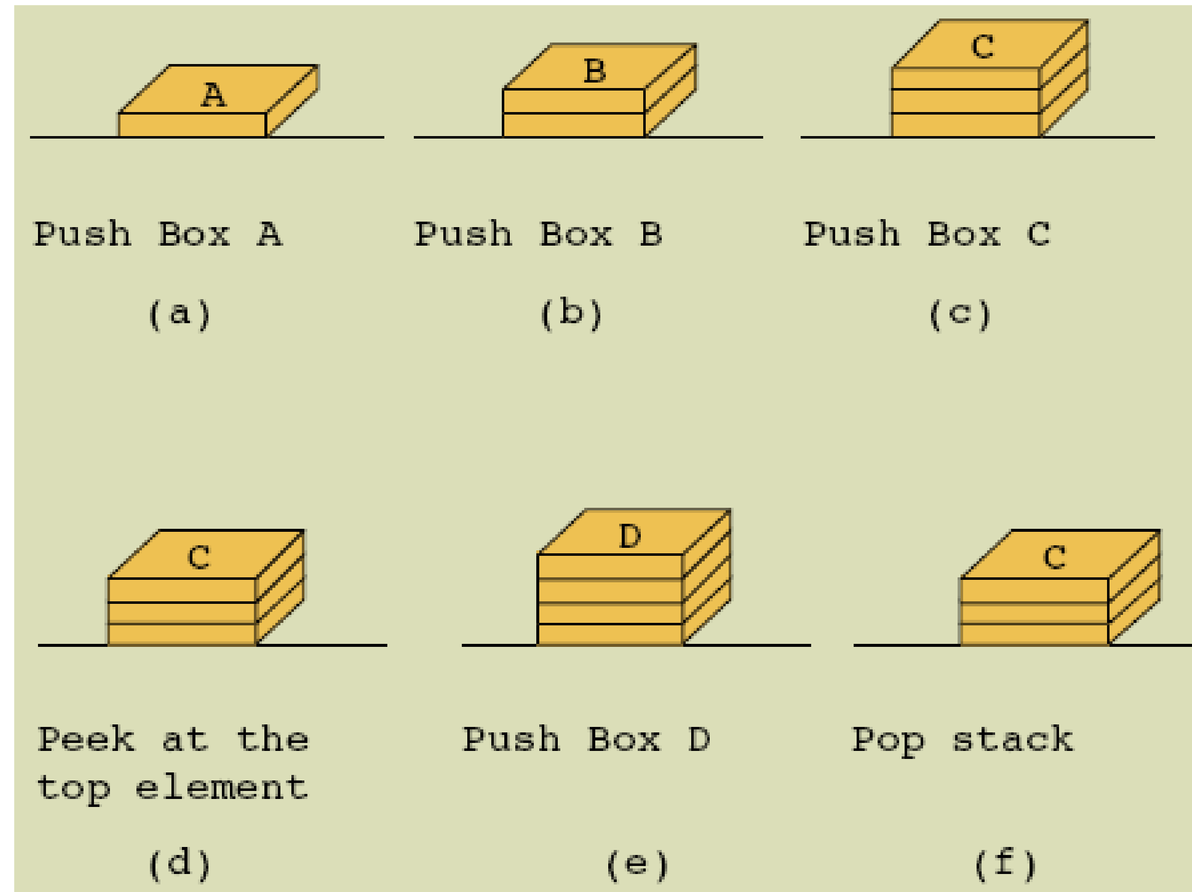
Peek

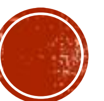
Look at the item on top of the stack, but do not remove it

POP

Look at the item on top of the stack and remove it

OPERATIONS ON STACKS





SOME APPLICATIONS

Converting a decimal number into a binary

Towers of Hanoi

Program execution

Parsing

Evaluating postfix expressions

....

IMPLEMENTATION OF STACKS USING ARRAYS

[99]

[3]

[2]

[1]

[0]

D

C

B

A

stack

maxStackSize 100

stackTop 4

list

stack elements

STACK

```
#define MAX_STACK_SIZE 100

/* stack structure*/
struct stack {
        int list[MAX_STACK_SIZE];
        int stackTop;
} st;
```

INITIALIZE STACK

```
void  initializeStack()
{
   int i;
   for (i =0 ; i < MAX_STACK_SIZE;
           st.list[i] = 0;
   st.stackTop = 0;
}
```

EMPTY STACK

```
void  isEmpty()
{
   if( st.stackTop == 0)
           return 1;
   else
           return 0;
}
```

FULL STACK

```
void  isFull()
{
   if( st.stackTop >= MAX_STACK_SIZE)
          return 1;
   else
          return 0;
}
```

PUSH STACK

```
void push(int item)
{
	if (!isFull()){
		st.list[st.stackTop] = item;
		st.stackTop++;
	}
	else
		printf("Stack Is Full!");
}
```

PEEK STACK

```
int peek()
{
        if (!isEmpty()){
                return st.list[st.stackTop-1] ;
        else
                return -1;
}
```

POP STACK

```
void pop()
{
	if (!isEmpty()){
		st.stackTop--;
		st.list[st.stackTop] = 0;
	}
	else
		printf("Stack Is Empty!");
}
```

CONVERTING A DECIMAL NUMBER TO A BINARY

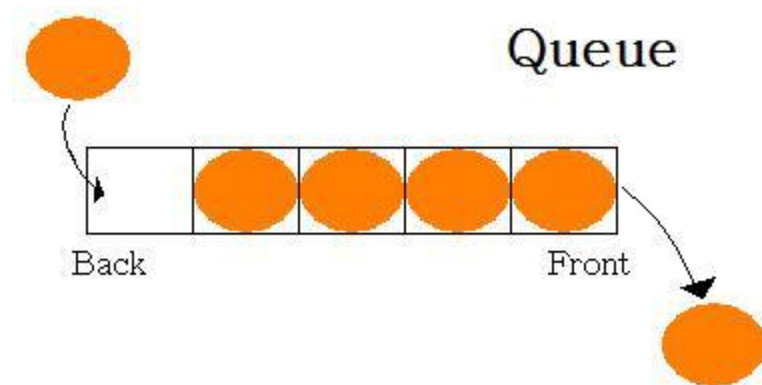
```
function outputInBinary(Integer n)
        Stack s = new Stack
        while n > 0 do
                Integer bit = n modulo 2
                s.push(bit)
                if s is full
                        then return error
                end if
                n = floor(n / 2)
                end while
        while s is not empty do
                output(s.pop())
        end while
end function
```

QUEUES

A queue is an ordered list in which all insertions take place at one end called the ***rear/back*** and all deletions take place at the opposite end called the ***front***.

A queue is a First In First Out data structure.

As in a stack, the middle elements of the queue are inaccessible.



CONCEPTUAL QUEUE

QUEUE

```
#define MAX_QUEUE_SIZE 100
/* queue structure*/
struct stack {
        int key;
} element;
element queue[MAX_QUEUE_SIZE];
int rear =  -1;
int front = -1;
```

INITIALIZE QUEUE

```
void  initializeQueue()
{
   int i;
   for (i =0 ; i < MAX_QUEUE_SIZE;
   i++)
           queue[i].key = 0;
   rear = front = -1;
}
```

EMPTY QUEUE

```
void  isEmpty()
{
   if( front == rear)
         return 1;
   else
         return 0;
}
```

FULL QUEUE

```
void  isFull()
{
   if( rear == MAX_QUEUE_SIZE)
         return 1;
   else
         return 0;
}
```

ADD QUEUE

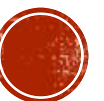
```
void addq(int *read, element item)
{
        if(*rear == MAX_QUEUE_SIZE-1)
            then return error
        queue[++*read] = item;
}
```

DELETE QUEUE

```
int deleteq(int *front, int rear)
{
        if (*front == rear){
                return empty error;
        return queue[++*front];
}
```

JOB SCHEDULING

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



CIRCULAR QUEUE

When the queue is full (the rear index is equals MAX_QUEUE_SIZE)

we should move the entire queue to the left and

recalculate the reat

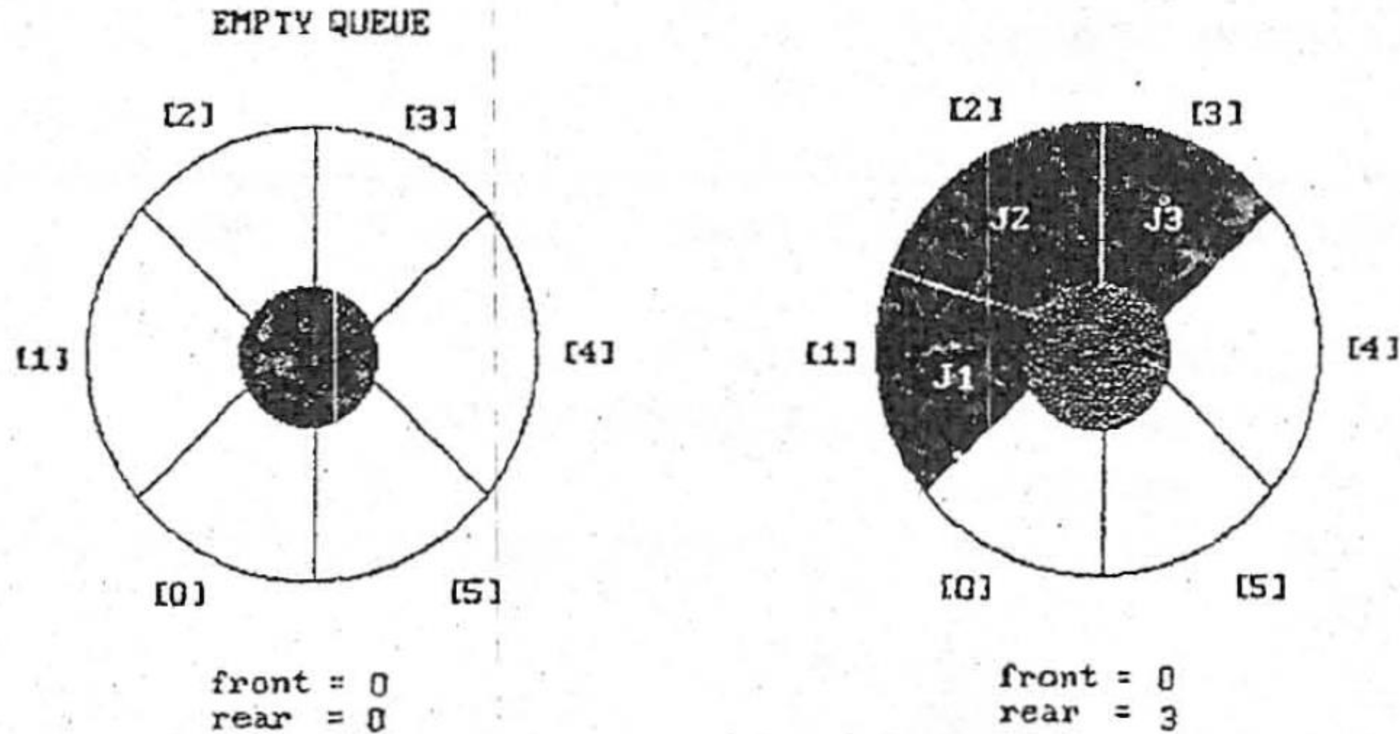
→The first element of the queue is at queue[0] and front is at -1.

Shifting an array is time-consuming

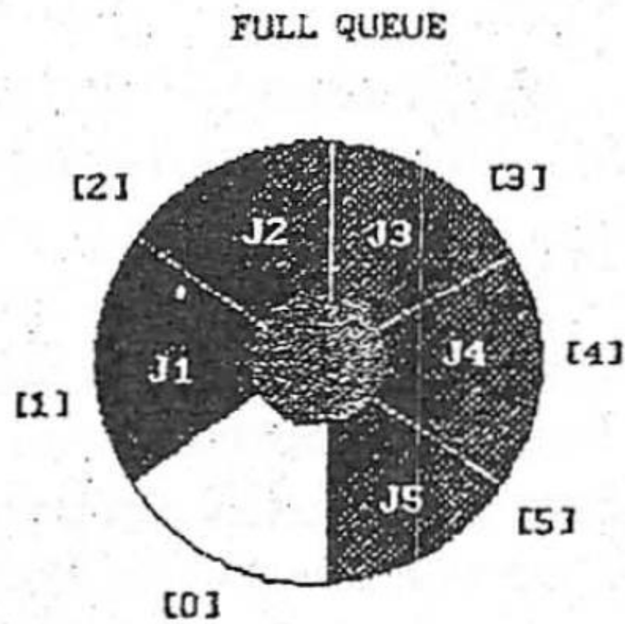
O(MAX_QUEUE_SIZE)

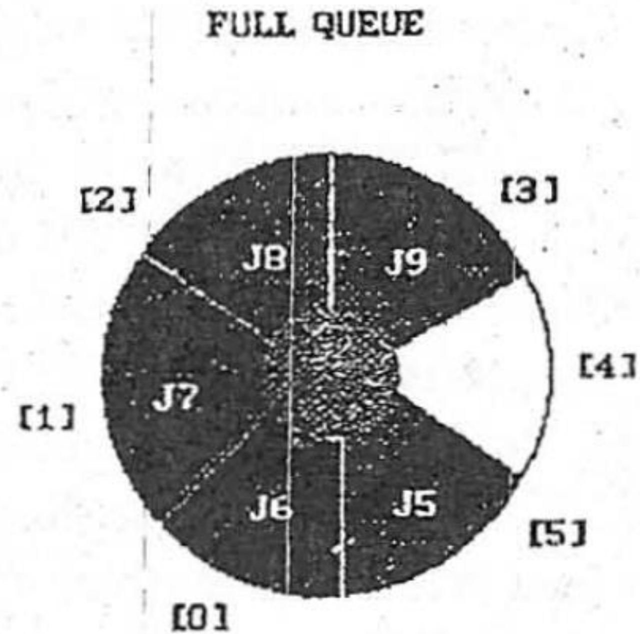
CIRCULAR QUEUE

More efficient queue representation



FULL CIRCULAR QUEUE





INITIALIZE CIRCULAR QUEUE

```
void  initializeQueue()
{
   int i;
   for (i =0 ; i < MAX_QUEUE_SIZE;
   i++)
          queue[i].key = 0;
   rear = front = 0;
}
```

EMPTY QUEUE

```
void  isEmpty()
{
   if( front == rear)
           return 1;
   else
           return 0;
}
```

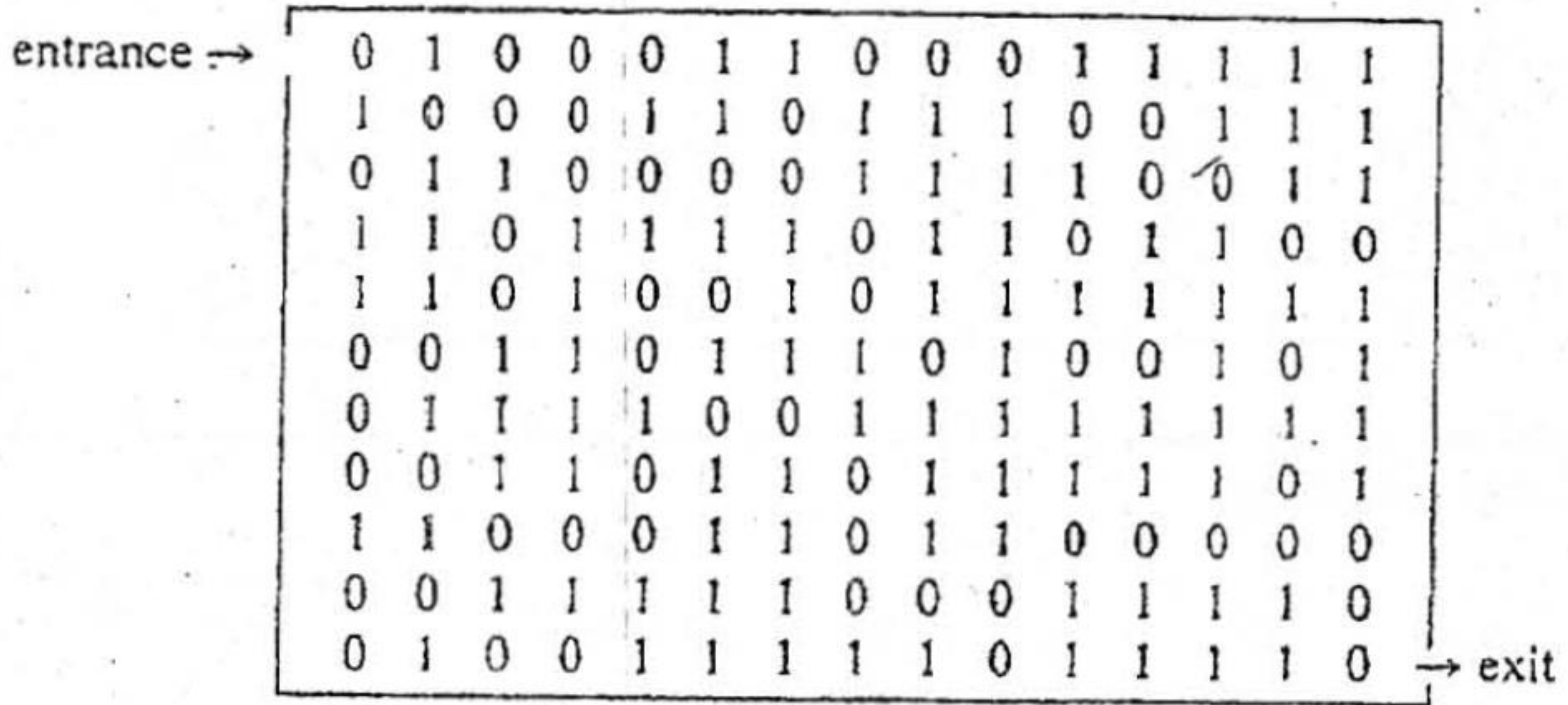
ADD CIRCULAR QUEUE

```
void addq(int front, int *read, element item)
{
          *rear = (*rear + 1) % MAX_QUEUE_SIZE
          if(front == *rear){
                    queueIsFull(rear);
                    return;
          }
          queue[*rear] = item;
}
```

DELETE CIRCULAR QUEUE

```
int deleteq(int *front, int rear)
{
        if (*front == rear){
                return empty error;
        *front = (*front + 1) % MAX_QUEUE_SIZE
        return queue[*front];
}
```

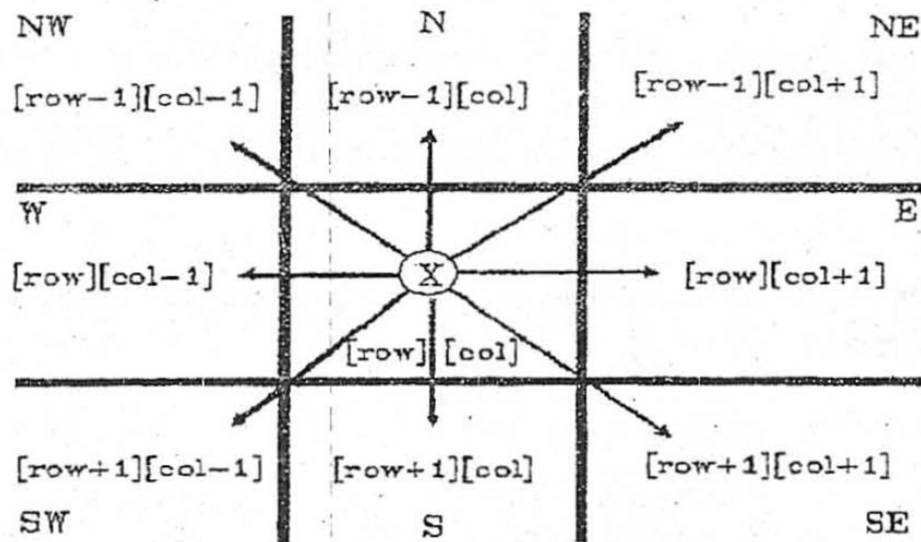
A MAZING PROBLEM



DIRECTIONS

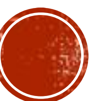
```
typedef struct {
        short int vert;
        short int horiz;
} offsets;
offsets move[8];
```

ALLOWABLE MOVES



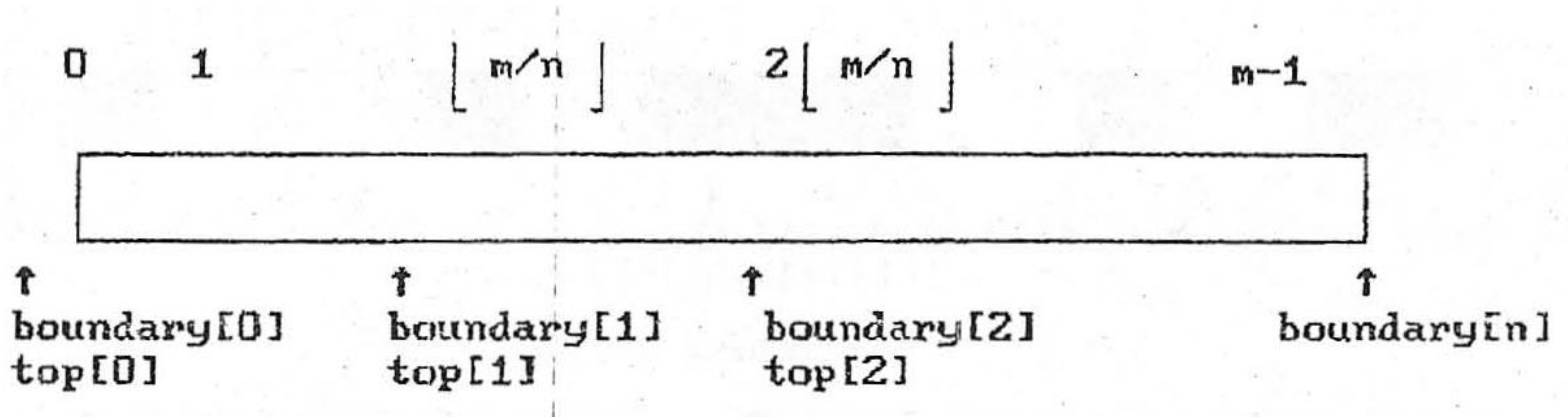
Name	Dir	*move[dir].vert*	*move[dir].horiz*
N	0	−1	0
NE	1	−1	1
E	2	0	1
SE	3	1	1
S	4	1	0
SW	5	1	−1
W	6	0	−1
NW	7	−1	−1

next_row = row + move[dir].vert;
next_col = col + move[dir].horiz;



IMPLEMENTATION

MULTIPLE STACKS & QUEUES



MULTIPLE STACKS

```
#define MEMORY_SIZE 100
#define MAX_STACKS 10
element memory[MEMORY_SIZE];
İnt top[MAX_STACKS];
İnt boundary[MAX_STACKS];
İnt n;                    //number of stacks entered by the user

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

MULTIPLE STACKS - ADD

```
void add(int i, element item)
{
        if(top[i] == boundary[i+1])
                stack_full(i);
        memory[++top[i]] = item;
}
```

MULTIPLE STACKS - DELETE

```
element delete(int i)
{
        if(top[i] == boundary[i])
                return stack_empty(i);
        return memory[top[i]--];
}
```

MULTIPLE STACKS – STACK FULL

b[0] t[0] b[1] t[1] b[i] t[i] t[i+1] t[j] b[j+1] b[n]

b[i+1] b[i+2]

b = boundary, *t* = top

MULTIPLE STACKS

1. stack_no < j < n, such that there is free space between stacks j and j+1

   move stacks stack_no+1, stack_no+2, ... j one position right.
   this creates a space between stack_no and stack_no+1.

2. 0 <= j < stack_no, such that there is free space between stacks j and j+1

   move stacks j+1, j+2, ... stack_no one position right.
   this also creates a space between stack_no and stack_no+1.

3. No j satisfying either condition 1 o condition 2, then there is no free space.

O(MEMORY_SIZE)