Module 2

Linear Data Structures and sequential representation

Stacks

A stack is a data structure where elements are inserted from one end and elements are deleted from the other end.(ie same end).ie LAST IN FIRST OUT(LIFO).We can represent a stack as

S={10,20,30,40}

The elements are inserted in the order 10,20,30,40,….

The element 10 will be at the bottom of the stack and 40 will be the top of the stack.The deleteion will be in the order 40,30,20,10

Operations

1.Insert 2.Delete 3.Overflow 4.Underflow 5.Create

Insert

An element is inserted from the top end. This operation is also called as PUSH

Delete(POP)

An element is deleted from the top of the stack. This operation is also called as POP operation

Overflow

This operation will check if the stack is full or not

Underflow

This function will check if the stack is empty or not

Create

This function creates a empty stack whose maximum size is STACK_SIZE

Implementation

As a 1-D array

STACK_SIZE is constant. Hence

#define STACk_SIZE 10;

Define S as an array of STACK_SIZE

int s[STACK_SIZE];

Variable top is associated with the array s and holds the index of the top element. If item is the element to be inserted for the first time then

s[0]=item;

Note : top=0 for the first insertion. Hence top should be equal to -1

ie top=-1 before the first insertion

Hence

top= -1;//stack empty

Isfull()

Inserting a element on to a stack is called as push operation.An item can be inserted only if the stack is NOT FULL. This condition is called overflow.

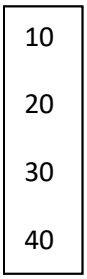
Consider a stack of STACK_SIZE 4

#define STACK_SIZE 4

Int top=-1;

Int s[STACK_SIZE];

After inserting 4 elements the stack would be



Note that the stack is full and it is not possible to insert. Ie such a condition is

Top=3

ie top=STACK_SIZE-1

Hence the condition is

```
if (top==STACK_SIZE-1)
return 1;//Stack Full
else
return 0;//Stack not full
```

PUSH()

Before inserting a element we need to check if the stack is full or not. Hence

```
if(top==STACK_SIZE-1)
{
printf("Stack Full");
return;
}
```

If the condition fails then we can insert

ie

```
top=top+1;
s[top]=item;
```

Therefore the function push would be

```
Void push()
{
if(top==STACK_SIZE-1)
{
printf("overflow");
return;
}
top=top+1;
s[top]=item;
}
```

Isempty()

An empty stack condition is indicated by this function.

```
if(top==-1)
return 1;
else
return 0;
```

Hence the function IsEmpty()

```
Int IsEmpty()
{
if(top==-1)
return 1;
else
return 0;
}
```

POP ()

Deleting a element from the stack is called POP.Only one item can be deleted from the top.Each time a item is deleted the top is decremented by 1 and when the stack is empty top=-1. When a attempt is made to delete from a empty stack it results in underflow. Hence

```
Int pop()
{
if(top==-1)
return -1;
return s[top];
top--;
}
```

Display()

We cannot display a empty stack and we need to display from the bottom. Hence we have

```
for(i=0;i<=top;i++)
{
printf("%d\n",s[i]);
}
```

ie

```
Void disp()
{
int i;
if(top==-1)
{
printf("empty stack");
return;
}
```

```
printf("elements are");
for(i=0;i<=top;i++)
{
printf("%d",s[i]);
}
}
```

Applications of stacks

Major applications of stack are

1.Conversion of expressions from infix to postfix

2.Evaluation of of expressions either as prefix or postfix

3.Recurssion

4.Determine if a given string is a palindrome or not

Expressions

A expression is a sequence of operators and operands which further reduces to a single value. Here the operands are constants and variables and operators are +,-,*,/. The expressions could be

Infix

Prefix

Postfix

Infix expression

Operator is between two operands. It can be parenthesized or unparenthesized. Example could be

A+b ---→it is a infix expression

Postfix expression

It is also called as suffix or reverse polish expression. Here the operator follows the operands. In such a expression parenthesis is NOT allowed. Example is

Ab+

Prefix expression

Here the operator precedes the operator. The expression is also called as polish expression. example is

+ab;

Precedence

In C language each operator is associated with priority. Based on the priority the expressions are evaluated.

Operator	precedence
( )	15
*,/,%	13
+ -	12

Conversion of infix to postfix

Infix expressions are evaluated based on the precedence and associativity which is time consuming. Evaluating postfix and prefix expressions are easy- No precedence or associativity is required

Program to convert from infix to postfix

```
#include<stdio.h>

#include<process.h>

#include<conio.h>


#define SIZE 50

char s[SIZE];

int top=-1;
```

```
void push(char elem)
{
s[++top]=elem;
}

char pop()
{
return (s[top--]);
}

int pr(char elem)
{
switch(elem)
{
case '#' :return 0;
case '(' :return 1;
case '+': return 2;
case '-' :return 2;
case '*':return 3;
case '/' : return 3;
case '%': return 3;
case '^' :return 4;
}
}
```

```
void main()
{
char infx[50],pofx[50],ch,elem;
int i=0;
int k=0;
printf("\n read the infix");
scanf("%s",infx);
push('#');
while((ch=infx[i++])!='\0')
        {
if(ch=='(')
push(ch);
else if(isalnum(ch))
pofx[k++]=ch;
else if(ch==')')
        {
while (s[top]!='(')
pofx[k++]=pop();
elem =pop();
        }
else
        {
while(pr(s[top])>=pr(ch))
pofx[k++]=pop();
```

```
push(ch);
        }
                }
while(s[top]!='#')
pofx[k++]=pop();
pofx[k]='\0';
printf("\n\n given infix %s postfix is %s\n",infx,pofx);
getch();
}
```

Evaluation of suffix expressions

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<math.h>
#include<stdlib.h>
#define MAX 50
int stack[MAX];
char post[MAX];
int top=-1;
void pushstack(int tmp);
void calculator(char c);
```

```
void main()
{
int i;
int  num;
printf("\n insert a postfix notation");
scanf("%s",post);
printf("postfix is %s",post);
for(i=0;i<strlen(post);i++)
{
if(post[i]>='0' &&  post[i]<='9')
{
pushstack(i);
}
if(post[i]=='+'||post[i]=='-'||post[i]=='*'||post[i]=='/'||post[i]=='^')
{
calculator(post[i]);
}
}
printf("\n\n result %d",stack[top]);
getch();

        printf("\n enter the num of disks");
        scanf("%d",&num);
        printf("sequence of moves are");
```

```
        towers(num,'A','C','B');
        getch();
        //return 0;

}
void pushstack(int tmp)
{
top++;
stack[top]=(int)(post[tmp]-48);
}


void calculator(char c)
{
int a,b,ans;
a=stack[top];
stack[top]='\0';
top--;
b=stack[top];
stack[top]='\0';
top--;
switch(c)
{
case '+':ans=b+a;
        break;
```

```
case '-': ans=b-a;
break;
case '*':ans =b*a;
break;
case '/': ans=b/a;
break;
case '^': ans=pow(b,a);
break;
default:ans=0;
}
top++;
stack[top]=ans;
}
```

Recursion

Recursion is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function.

```
void recursion() {
  recursion(); /* function calls itself */
}

int main() {
  recursion();
}
```

The C programming language supports recursion, i.e., a function to call itself. But while using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go into an infinite loop.

Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, etc.

Number Factorial

The following example calculates the factorial of a given number using a recursive function −

```
#include <stdio.h>

int factorial(unsigned int i) {

   if(i <= 1) {
      return 1;
   }
   return i * factorial(i - 1);
}

int  main() {
   int i = 15;
   printf("Factorial of %d is %d\n", i, factorial(i));
   return 0;
}
```

Fibonacci Series

The following example generates the Fibonacci series for a given number using a recursive function −

```
#include <stdio.h>

int fibonaci(int i) {

   if(i == 0) {
      return 0;
   }

   if(i == 1) {
      return 1;
   }
   return fibonaci(i-1) + fibonaci(i-2);
}

int  main() {

   int i;
```

```
  for (i = 0; i < 10; i++) {
    printf("%d\t\n", fibonaci(i));
  }

  return 0;
}
```

GCD of two numbers

```
#include <stdio.h>
int main()
{
   int n1, n2, i, gcd;

   printf("Enter two integers: ");
   scanf("%d %d", &n1, &n2);

   for(i=1; i <= n1 && i <= n2; ++i)
   {
      // Checks if i is factor of both integers
      if(n1%i==0 && n2%i==0)
         gcd = i;
   }

   printf("G.C.D of %d and %d is %d", n1, n2, gcd);

   return 0;
}
```

In this program, two integers entered by the user are stored in variable *n1* and *n2*.Then, for loop is iterated until *i* is less than *n1* and *n2*.

In each iteration, if both *n1* and *n2* are exactly divisible by *i*, the value of *i* is assigned to *gcd*.

When the for loop is completed, the greatest common divisor of two numbers is stored in variable *gcd*.

GCD of Two Numbers using Recursion

```
#include <stdio.h>
int hcf(int n1, int n2);
int main()
{
  int n1, n2;
```

```
  printf("Enter two positive integers: ");
  scanf("%d %d", &n1, &n2);
  printf("G.C.D of %d and %d is %d.", n1, n2, hcf(n1,n2));
  return 0;
}
int hcf(int n1, int n2)
{
   if (n2!=0)
      return hcf(n2, n1%n2);
   else
      return n1;
}
```

Tower of Hanoi problem

Consider 3 needles say A,B,C. Different diameter discs are placed one above the other through needle A. The smaller disc is on the top.The two needles B and C are empty. All the discsare to be transferred to C using B with the following conditions

Only one disc can be moved at one time

Smaller disc is always at the top of the larger disc

Only one needle can be used temporarily

BASE case

This occurs when there are no discs. Hence

return when n=0;

GENERAL

This occurs when one or more discs have to be moved from source to destination. If there are N discs then

Move n-1 discs recursively to temp

Move nth disc from source to destination

Move n-1 discs from temp to destination

```
void towers(int num,char frompeg,char topeg,char auxpeg)
{
if(num==1)
{
printf("\n move disk 1 from peg %c to peg %c ",frompeg,topeg);
return;
}
towers(num-1,frompeg,auxpeg,topeg);
printf("\n move disk %d from peg %c to peg %c",num,frompeg,topeg);
towers(num-1,auxpeg,topeg,frompeg);
}
```

Queues

A queue is a data structure where elements are inserted from one end and deleted from the other end. Insertion is from rear end and deletion from front end (FIFO). There are different types of queues.

Linear queue

Circular queue

Double ended queue

Priority queue

Linear queue

Such a queue is also called as queue. It is implemented as a linear list. One of the methods of representing a queue is a 1D array with two variables front and rear. Variable rear is used as an index to the first element. Variable front is used as an index to the last element. Elements are inserted in FIFO order.

Operations on Queue

Insert

Delete

Overflow

Underflow

Create

Create()

This function can be implemented using a 1-D array

Q[QUE_SIZE]

Where QUE_SIZE is a constant defined as

#define QUE_SIZE 5

Variables front and rear are associated with q and holds the index for the first and last element.

If the item is the element to be inserted first time then

Q[0]=item;

Note that front and rear will be zero

Since we are inserting from the rear end value of

Rear=-1; and

Front=0;//indicates queue empty

Hence create() is implemented as

```
#define QUE_SIZE 5
Int q[QUE_SIZE];
Int front=0;
Int rear =-1;
```

Isfull()

In order to insert elements on to a queue we need to so at the rear end.

Consider

#define QUE_SIZE 4

When 4 elements are inserted we cannot insert any further. Such a condition is called queue overflow. Such a condition is checked by

Rear=QUE_SIZE-1

Then Queue is full

Ie

If rear==QUE_SIZE-1

retrun 1;

else

return 0;

Hence we have

```
Int ISfull()
{
return rear==QUE_SIZE-1
}
```

InserQ()

Before inserting we need to check if Queue is full or not.

```
If rear==QUE_SIZE-1
{
printf("Queue full")
return;
}
```

To insert

```
rear=rear+1;
```

and

```
Q[rear]=item;
```

Hence we have

```
Void InsertQ()
{
if (rear==QUE_SIZE-1)
{
printf("overflow");
return;
}
rear=rear+1;
Q[rear]=item;
}
```

ISempty()

When the Queue is empty rear=-1 and front=0;

ie

if(front)>rear

return 1;

else

return 0;

Hence we have

Int Isempty()

{

return (front>rear);

}

DeleteQ()

Element is always deleted from the front.

No elements in the queue is indicated by

if(front>rear)

return -1;

In order to return we execute

return q[front+1];

Hence we have

Int DeleteQ()

{

if (front>rear)

retrun -1;

return[front+1];

Display()

In order to display the elements of the queue we do so by displaying from the front end. Hence we have

```
for(i=front;i<=rear;i++)

{

printf("%d\n",q[i]);

}
```

Disadvantages of queue

Consider the queue in which front is < rear

Q[f]=30 and q[r]=50

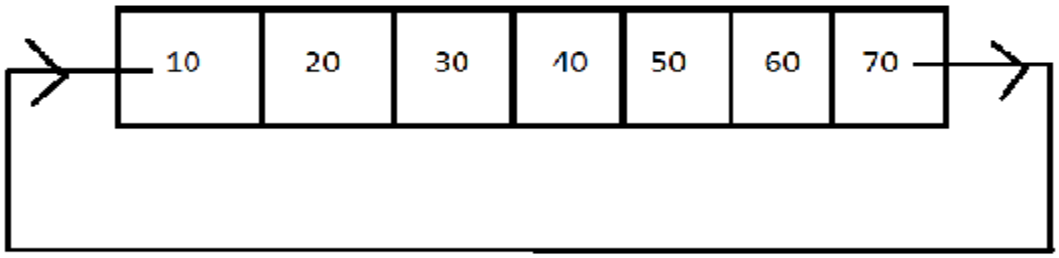
	30	40	50

If we try to insert say 60 we still get "QUEUE OVERFLOW" although space is available. This is because before insertion we check if rear==QUE_SIZE-1 and if so we display "overflow".Hence even though space is available we are still not able to use the same. This is the main disadvantage of a QUEUE

CIRCULAR QUEUE

In a standard queue data structure re-buffering problem occurs for each dequeue operation. To solve this problem by joining the front and rear ends of a queue to make the queue as a circular queue
Circular queue is a linear data structure. It follows FIFO principle.

- In circular queue the last node is connected back to the first node to make a circle.
- Circular linked list fallow the First In First Out principle
- Elements are added at the rear end and the elements are deleted at front end of the queue
- Both the front and the rear pointers points to the beginning of the array.
- It is also called as "Ring buffer".
- Items can inserted and deleted from a queue in O(1) time.



Circular Queue

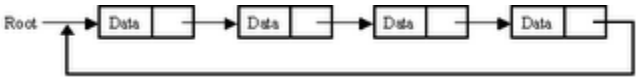
Circular Queue can be created in three ways they are

- Using single linked list
- Using double linked list
- Using arrays

Using single linked list:

It is an extension for the basic single linked list. In circular linked list Instead of storing a Null value in the last node of a single linked list, store the address of the 1st node (root) forms a circular linked list. Using circular linked list it is possible to directly traverse to the first node after reaching the last node.

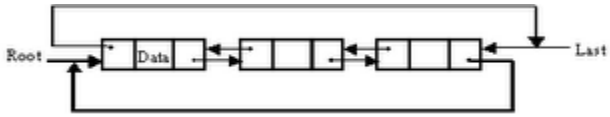
The following figure shows circular single linked list:



Using double linked list

In double linked list the right side pointer points to the next node address or the address of first node and left side pointer points to the previous node address or the address of last node of a list. Hence the above list is known as circular double linked list.

The following figure shows Circular Double linked list :-



Algorithm for creating circular linked list :-

Step 1) start

Step 2) create anode with the following fields to store information and the address of the next node.

Structure node

begin

int info

pointer to structure node called next

end

Step 3) create a class called clist with the member variables of pointer to structure nodes called root, prev, next and the member functions create ( ) to create the circular linked list and display ( ) to display the circular linked list.

Step 4) create an object called 'C' of clist type

Step 5) call C. create ( ) member function

Step 6) call C. display ( ) member function

Step 7) stop

Algorithm for create ( ) function:-

Step 1) allocate the memory for newnode

newnode = new (node)

Step 2) newnode->next=newnode. // circular

Step 3) Repeat the steps from 4 to 5 until choice = 'n'

Step 4) if (root=NULL)

root = prev=newnode // prev is a running pointer which points last node of a list

else

newnode->next = root

prev->next = newnode

prev = newnode

Step 5) Read the choice

Step 6) return

Algorithm for display ( ) function :-

Step 1) start
Step 2) declare a variable of pointer to structure node called temp, assign root to temp
temp = root
Step 3) display temp->info
Step 4) temp = temp->next
Step 5) repeat the steps 6 until temp = root
Step 6) display temp info
Step 7) temp=temp->next
Step 8) return

Using array

In arrays the range of a subscript is 0 to n-1 where n is the maximum size. To make the array as a circular array by making the subscript 0 as the next address of the subscript n-1 by using the formula subscript = (subscript +1) % maximum size. In circular queue the front and rear pointer are updated by using the above formula.

Algorithm for Enqueue operation using array

```
Step 1. start
Step 2. if (front == (rear+1)%max)
            Print error "circular queue overflow "
Step 3. else
        { rear = (rear+1)%max
          Q[rear] = element;
           If (front == -1 ) f = 0;
        }
Step 4. stop
```

Algorithm for Dequeue operation using array

```
Step 1. start
        Step 2. if ((front == rear) && (rear == -1))
                    Print error "circular queue underflow "
        Step 3. else
                { element = Q[front]
                  If (front == rear) front=rear = -1
                  Else
                      Front = (front + 1) % max
                }
        Step 4. stop
```

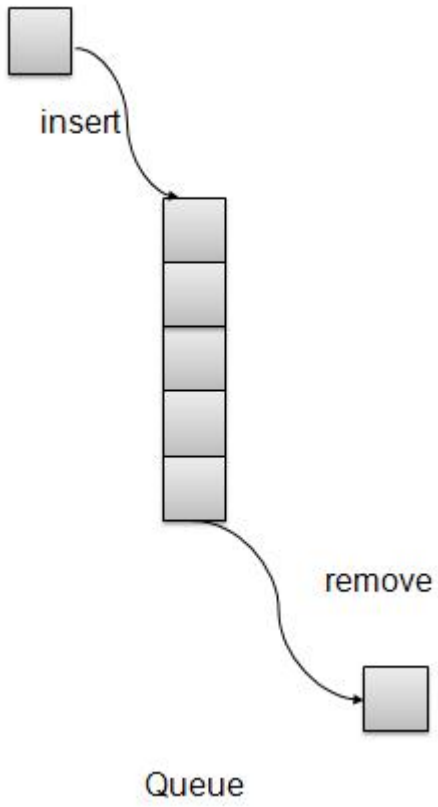
Priority queue

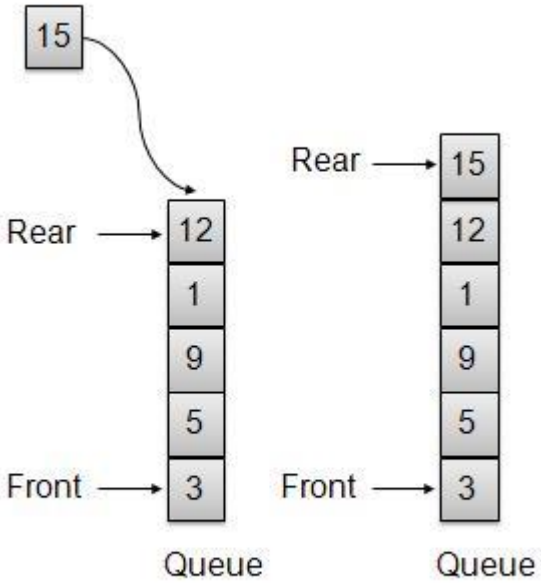
Priority Queue is more specialized data structure than Queue. Like ordinary queue, priority queue has same method but with a major difference. In Priority queue items are ordered by key value so that item with the lowest value of key is at front and item with the highest value of key is at rear or vice versa. So we're assigned priority to item based on its key value. Lower the value, higher the priority. Following are the principal methods of a Priority Queue.

Basic Operations

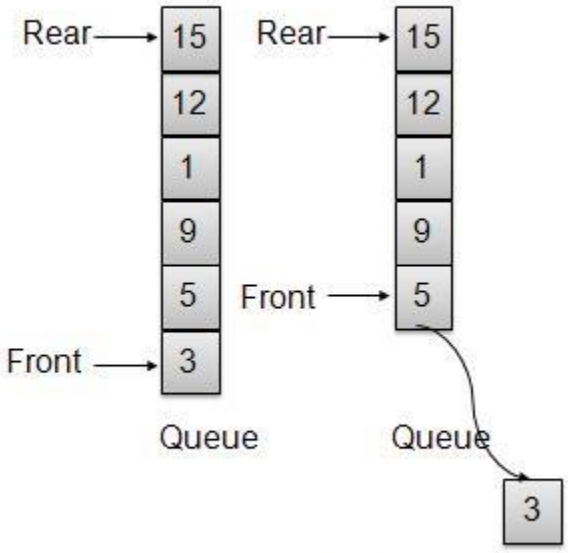
- **insert / enqueue** − add an item to the rear of the queue.
- **remove / dequeue** − remove an item from the front of the queue.

Priority Queue Representation





One item inserted at rear end



One Item removed from front

Double-Ended Queue

A double-ended queue is an abstract data type similar to an simple queue, it allows you to insert and delete from both sides means items can be added or deleted from the front or rear end.

Algorithm for Insertion at rear end

Step -1: [Check for overflow]

if(rear==MAX)

Print("Queue is Overflow");

return;

Step-2: [Insert element]

else

rear=rear+1;

q[rear]=no;

[Set rear and front pointer]

if rear=0

rear=1;

```
                if front=0
                        front=1;
```

Step-3: return

Implementation of insertion at rear end

```
void add_item_rear()
{
        int num;
        printf("\n Enter Item to insert : ");
        scanf("%d",&num);
        if(rear==MAX)
        {
                printf("\n Queue is Overflow");
                return;
        }
        else
        {
                rear++;
                q[rear]=num;
                if(rear==0)
                        rear=1;
                if(front==0)
                        front=1;
        }
```

```
}
```

Algorithm for Insertion at font end

Step-1 : [Check for the front position]

```
if(front<=1)
        Print (“Cannot add item at front end”);
return;
```

Step-2 : [Insert at front]

```
else
        front=front-1;
        q[front]=no;
```

Step-3 : Return

Implementation of Insertion at font end

```
void add_item_front()
{
    int num;
    printf("\n Enter item to insert:");
    scanf("%d",&num);
    if(front<=1)
    {
        printf("\n Cannot add item at front end");
        return;
    }
```

```
        else
        {
                front--;
                q[front]=num;
        }
}
```

Algorithm for Deletion from front end

Step-1 [Check for front pointer]

```
        if front=0
                print(" Queue is Underflow");
        return;
```

Step-2 [Perform deletion]

```
        else
                no=q[front];
                print("Deleted element is",no);
                [Set front and rear pointer]
        if front=rear
                front=0;
                rear=0;
        else
                front=front+1;
```

Step-3 : Return

Implementation of Deletion from front end

```
 void delete_item_front()
{
    int num;
    if(front==0)
    {
        printf("\n Queue is Underflow\n");
        return;
    }
    else
    {
        num=q[front];
        printf("\n Deleted item is %d\n",num);
        if(front==rear)
        {
                front=0;
                rear=0;
        }
        else
        {
                front++;
        }
    } }
```

Algorithm for Deletion from rear end

Step-1 : [Check for the rear pointer]

if rear=0

print(“Cannot delete value at rear end”);

return;

Step-2: [perform deletion]

else

no=q[rear];

[Check for the front and rear pointer]

if front= rear

front=0;

rear=0;

else

rear=rear-1;

print(“Deleted element is”,no);

Step-3 : Return

Implementation of Deletion from rear end

```
void delete_item_rear()
{
    int num;
    if(rear==0)
    {
        printf("\n Cannot delete item at rear end\n");
        return;
    }
    else
    {
        num=q[rear];
        if(front==rear)
        {
                front=0;
                rear=0;
        }
        else
        {
                rear--;
                printf("\n Deleted item is %d\n",num);
        }
```

```
    }

}
```

Applications of Queue

Queue, as the name suggests is used whenever we need to have any group of objects in an order in which the first one coming in, also gets out first while the others wait for there turn, like in the following scenarios :

1. Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
2. In real life, Call Center phone systems will use Queues, to hold people calling them in an order, until a service representative is free.
3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive, First come first served.