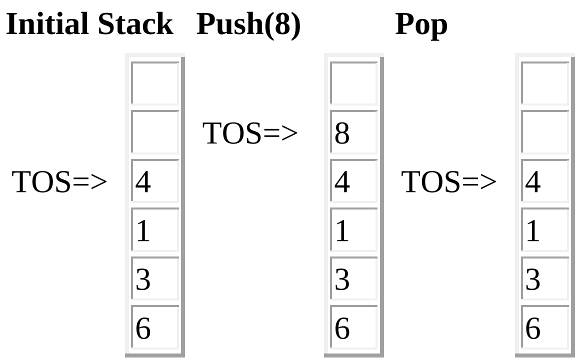
4. Stacks

A simple data structure, in which insertion and deletion occur at the same end, is termed (called) a stack. It is a LIFO (Last In First Out) structure.

The operations of insertion and deletion are called PUSH and POP

Push - push (put) item onto stack

Pop - pop (get) item from stack



Our Purpose:
To develop a stack implementation that does not tie us to a particular data type or to a particular implementation.

Implementation:
Stacks can be implemented both as an array (contiguous list) and as a linked list. We want a set of operations that will work with either type of implementation: i.e. the method of implementation is hidden and can be changed without affecting the programs that use them.

The Basic Operations:

Push()
```
{
    if there is room {
        put an item on the top of the stack
    else
        give an error message
    }
}
```

Pop()
```
{
    if stack not empty {
            return the value of the top item
            remove the top item from the stack
                        }
        else {
            give an error message
            }
}
```

CreateStack()

```
{
remove existing items from the stack
initialise the stack to empty
}
```

4.1. Array Implementation of Stacks: The PUSH operation

Here, as you might have noticed, addition of an element is known as the PUSH operation. So, if an array is given to you, which is supposed to act as a STACK, you know that it has to be a STATIC Stack; meaning, data will overflow if you cross the upper limit of the array. So, keep this in mind.

Algorithm:

Step-1: Increment the Stack TOP by 1. Check whether it is always less than the Upper Limit of the stack. If it is less than the Upper Limit go to step-2 else report -"Stack Overflow"
Step-2: Put the new element at the position pointed by the TOP

Implementation:

```
static int stack[UPPERLIMIT];
int top= -1; /*stack is empty*/
..
..
main()
{
..
..
push(item);
..
..
}

push(int item)
{
   top = top + 1;
   if(top < UPPERLIMIT)
```

```
            stack[top] = item; /*step-1 & 2*/
        else
            cout<<"Stack Overflow";
}
```

Note:- In array implementation,we have taken TOP = -1 to signify the empty stack, as this simplifies the implementation.

4.2. Array Implementation of Stacks: the POP operation

POP is the synonym for delete when it comes to Stack. So, if you're taking an array as the stack, remember that you'll return an error message, "Stack underflow", if an attempt is made to Pop an item from an empty Stack. OK.

Algorithm

```
Step-1: If the Stack is empty then give the alert "Stack underflow" and quit; or else go to step-2
Step-2: a) Hold the value for the element pointed by the TOP
        b) Put a NULL value instead
        c) Decrement the TOP by 1
```

Implementation:

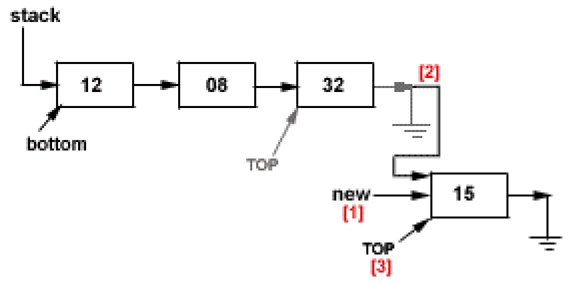
```
static int stack[UPPPERLIMIT];
int top=-1;
..
..
main()
{
..
..
poped_val = pop();
..
..
}

int pop()
{
int del_val = 0;
if(top == -1)
            cout<<"Stack underflow"; /*step-1*/
  else
     {
      del_val = stack[top];  /*step-2*/
      stack[top] = NULL;
      top = top -1;
```

```
    }
return(del_val);
}
```

Note: - Step-2:(b) signifies that the respective element has been deleted.

4.3. Linked List Implementation of Stacks: the PUSH operation

It's very similar to the insertion operation in a dynamic singly linked list. The only difference is that here you'll add the new element only at the end of the list, which means addition can happen only from the TOP. Since a dynamic list is used for the stack, the Stack is also dynamic, means it has no prior upper limit set. So, we don't have to check for the Overflow condition at all!



In Step [1] we create the new element to be pushed to the Stack.
In Step [2] the TOP most element is made to point to our newly created element.
In Step [3] the TOP is moved and made to point to the last element in the stack, which is our newly added element.

Algorithm

Step-1: If the Stack is empty go to step-2 or else go to step-3
Step-2: Create the new element and make your "stack" and "top" pointers point to it and quit.
Step-3: Create the new element and make the last (top most) element of the stack to point to it
Step-4: Make that new element your TOP most element by making the "top" pointer point to it.

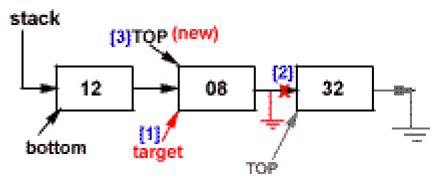
Implementation:

```
struct node{
    int item;
    struct node *next;
   }
struct node *stack = NULL; /*stack is initially empty*/
struct node *top = stack;
main()
{
```

```
..
..
push(item);
..
}

push(int item)
{
    if(stack == NULL)  /*step-1*/
      {
       newnode = new node  /*step-2*/
       newnode -> item = item;
       newnode -> next = NULL;
       stack = newnode;
       top = stack;
      }
    else
      {
       newnode = new node; /*step-3*/
       newnode -> item = item;
       newnode -> next = NULL;
       top ->next = newnode;
       top = newnode;   /*step-4*/
      }
}
```

4.4. Linked List Implementation of Stacks: the POP Operation

This is again very similar to the deletion operation in any Linked List, but you can only delete from the end of the list and only one at a time; and that makes it a stack. Here, we'll have a list pointer, "target", which will be pointing to the last but one element in the List (stack). Every time we POP, the TOP most element will be deleted and "target" will be made as the TOP most element.



In step[1] we got the "target" pointing to the last but one node. In step[2] we freed the TOP most element. In step[3] we made the "target" node as our TOP most element.

Supposing you have only one element left in the Stack, then we won't make use of "target" rather we'll take help of our "bottom" pointer. See how...

Algorithm:
Step-1: If the Stack is empty then give an alert message "Stack Underflow" and quit; or else

proceed
Step-2: If there is only one element left go to step-3 or else step-4
Step-3: Free that element and make the "stack", "top" and "bottom" pointers point to NULL and quit
Step-4: Make "target" point to just one element before the TOP; free the TOP most element; make "target" as your TOP most element

Implementation:

```
struct node
{
 int nodeval;
 struct node *next;
}
struct node *stack = NULL; /*stack is initially empty*/
struct node *top = stack;

main()
{
int newvalue, delval;
..
push(newvalue);
..
delval = pop();   /*POP returns the deleted value from the stack*/
}

int pop( )
{
int pop_val = 0;
struct node *target = stack;
    if(stack == NULL)  /*step-1*/
     cout<<"Stack Underflow";
    else
      {
       if(top == bottom)  /*step-2*/
         {
          pop_val = top -> nodeval;   /*step-3*/
          delete top;
         stack = NULL;
         top = bottom = stack;
         }
        else   /*step-4*/
         {
         while(target->next != top) target = target ->next;
         pop_val = top->nodeval;
         delete top;
         top = target;
         target ->next = NULL;
         }
      }
return(pop_val);
}
```

4.5. Applications of Stacks

4.5.1. Evaluation of Algebraic Expressions

e.g. **4 + 5 * 5**

simple calculator: 45

scientific calculator: 29 (correct)

Question:

Can we develop a method of evaluating arithmetic expressions without having to ‘look ahead’ or ‘look back’? ie consider the quadratic formula:

x = (-b+(b^2-4*a*c)^0.5)/(2*a)

where ^ is the power operator, or, as you may remember it :

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

In it’s current form we cannot solve the formula without considering the ordering of the parentheses. i.e. we solve the innermost parenthesis first and then work outwards also considering operator precedence. Although we do this naturally, consider developing an algorithm to do the same . . . . . . possible but complex and inefficient. Instead . . . .

Re-expressing the Expression

Computers solve arithmetic expressions by restructuring them so the order of each calculation is embedded in the expression. Once converted an expression can then be solved in one pass.

Types of Expression

The normal (or human) way of expressing mathematical expressions is called infix form, e.g. **4+5*5**. However, there are other ways of representing the same expression, either by writing all operators before their operands or after them,

e.g.: **4 5 5 * +**

+ 4 * 5 5

This method is called Polish Notation (because this method was discovered by the Polish mathematician Jan Lukasiewicz).

When the operators are written before their operands, it is called the **prefix** form e.g. **+ 4 * 5 5**

When the operators come after their operands, it is called **postfix** form (**suffix** form or **reverse polish notation**)

e.g. **4 5 5 * +**

The valuable aspect of RPN (Reverse Polish Notation or postfix)

- Parentheses are unnecessary
- Easy for a computer (compiler) to evaluate an arithmetic expression Postfix (Reverse Polish Notation)

Postfix notation arises from the concept of post-order traversal of an expression tree (see Weiss p. 93 - this concept will be covered when we look at trees).

For now, consider postfix notation as a way of redistributing operators in an expression so that their operation is delayed until the correct time.

Consider again the quadratic formula:
x = (-b+(b^2-4*a*c)^0.5)/(2*a)
In postfix form the formula becomes:
x b @ b 2 ^ 4 a * c * - 0.5 ^ + 2 a * / =

where @ represents the unary - operator.

Notice the order of the operands remain the same but the operands are redistributed in a non-obvious way (an algorithm to convert infix to postfix can be derived).

Purpose

The reason for using postfix notation is that a fairly simple algorithm exists to evaluate such expressions based on using a stack.

Postfix Evaluation

Consider the postfix expression :
6 5 2 3 + 8 * + 3 + *

Algorithm

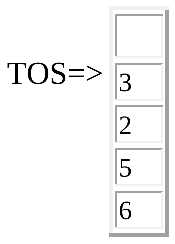
```
initialise stack to empty;
while (not end of postfix expression) {
   get next postfix item;
   if(item is value)
      push it onto the stack;
   else if(item is binary operator) {
      pop the stack to x;
      pop the stack to y;
      perform y operator x;
      push the results onto the stack;
   } else if (item is unary operator) {
      pop the stack to x;
      perform operator(x);
      push the results onto the stack
   }
}
The single value on the stack is the desired result.
```

Binary operators: +, -, *, /, etc.,

Unary operators: **unary minus**, **square root**, **sin**, **cos**, **exp**, etc.,

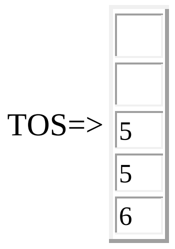
So for **6 5 2 3 + 8 * + 3 + ***

the first item is a value (6) so it is pushed onto the stack
the next item is a value (5) so it is pushed onto the stack
the next item is a value (2) so it is pushed onto the stack
the next item is a value (3) so it is pushed onto the stack
and the stack becomes

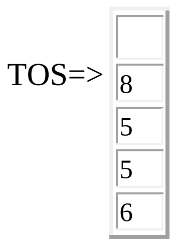


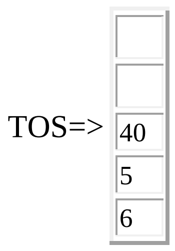
the remaining items are now: + 8 * + 3 + *

So next a '+' is read (a binary operator), so 3 and 2 are popped from the stack and their sum '5' is pushed onto the stack:



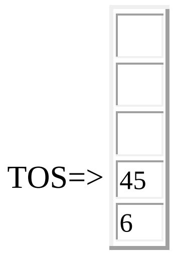
Next 8 is pushed and the next item is the operator *:

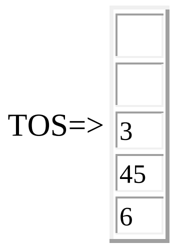




(8, 5 popped, 40 pushed)

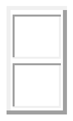
Next the operator + followed by 3:

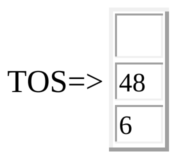




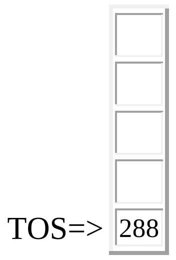
(40, 5 popped, 45 pushed, 3 pushed)

Next is operator +, so 3 and 45 are popped and 45+3=48 is pushed





Next is operator *, so 48 and 6 are popped, and 6*48=288 is pushed



Now there are no more items and there is a single value on the stack, representing the final answer 288.

Note the answer was found with a single traversal of the postfix expression, with the stack being used as a kind of memory storing values that are waiting for their operands.

4.5.2. Infix to Postfix (RPN) Conversion

Of course postfix notation is of little use unless there is an easy method to convert standard (infix) expressions to postfix. Again a simple algorithm exists that uses a stack:

Algorithm

```
initialise stack and postfix output to empty;
while(not end of infix expression) {
  get next infix item
  if(item is value) append item to pfix o/p
  else if(item == ‘(‘) push item onto stack
  else if(item == ‘)’) {
    pop stack to x
    while(x != ‘(‘)
      app.x to pfix o/p & pop stack to x
  } else {
    while(precedence(stack top) >= precedence(item))
      pop stack to x & app.x to pfix o/p
    push item onto stack
  }
}
while(stack not empty)
  pop stack to x and append x to pfix o/p
```

Operator Precedence (for this algorithm):

4 : ‘**(**‘ - only popped if a matching ‘**)**’ is found

3 : All unary operators

2 : ***/ ****

1 : + -

The algorithm immediately passes values (operands) to the postfix expression, but remembers (saves) operators on the stack until their right-hand operands are fully translated.

eg., consider the infix expression **a+b*c+(d*e+f)*g**

	Stack	**Output**
TOS=>	+	**ab**
TOS=>	* / +	**abc**
TOS=>	+	**abc*+**
TOS=>	* / (/ +	**abc*+de**
TOS=>	+ / (/ +	**abc*+de*f**
TOS=>	+	**abc*+de*f+**
TOS=>	* / +	**abc*+de*f+g**
empty		**abc*+de*f+g*+**

4.5.3. Function Calls

When a function is called, arguments (including the return address) have to be passed to the called function.

If these arguments are stored in a fixed memory area then the function cannot be called recursively since the 1st return address would be overwritten by the 2nd return address before the first was used:

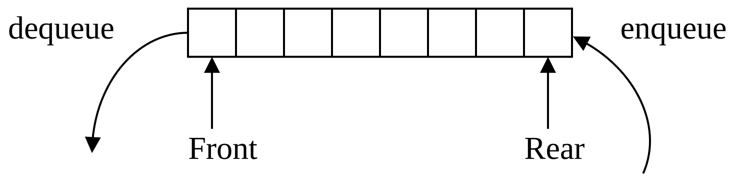
```
10 call function abc(); /* retadrs = 11 */
11 continue;
   ...
90 function abc;
91 code;
92 if (expression)
93    call function abc(); /* retadrs = 94 */
94 code
95 return /* to retadrs */
```

A stack allows a new instance of retadrs for each call to the function. Recursive calls on the function are limited only by the extent of the stack.

```
10 call function abc(); /* retadrs1 = 11 */
11 continue;
   ...
90 function abc;
91 code;
92 if (expression)
93    call function abc(); /* retadrs2 = 94 */
94 code
95 return /* to retadrsn */
```

5.Queue

- A data structure that has access to its data at the front and rear.
- Operates on FIFO (First In First Out) basis.
- Uses two pointers/indices to keep tack of information/data.
- has two basic operations:
    - enqueue - inserting data at the rear of the queue
    - dequeue – removing data at the front of the queue



Example:

Operation	Content of queue
Enqueue(B)	B
Enqueue(C)	B, C
Dequeue()	C
Enqueue(G)	C, G
Enqueue (F)	C, G, F
Dequeue()	G, F
Enqueue(A)	G, F, A
Dequeue()	F, A

5.1. Simple array implementation of enqueue and dequeue operations

Analysis:

Consider the following structure: int Num[MAX_SIZE];

We need to have two integer variables that tell:

- the index of the front element
- the index of the rear element

We also need an integer variable that tells:

- the total number of data in the queue

```
int FRONT =-1,REAR =-1;
int QUEUESIZE=0;
```

- To enqueue data to the queue
    - check if there is space in the queue

```
REAR<MAX_SIZE-1 ?
Yes:  - Increment REAR
      - Store the data in Num[REAR]
      - Increment QUEUESIZE
        FRONT = = -1?
            Yes: - Increment FRONT
No:   - Queue Overflow
```

- To dequeue data from the queue
    - check if there is data in the queue

```
QUEUESIZE > 0 ?
Yes:  - Copy the data in Num[FRONT]
      - Increment FRONT
      - Decrement QUEUESIZE
No:   - Queue Underflow
```

<u>Implementation</u>:

```
const int MAX_SIZE=100;
int FRONT =-1, REAR =-1;
int QUEUESIZE = 0;

void enqueue(int x)
{
        if(Rear<MAX_SIZE-1)
        {
             REAR++;
             Num[REAR]=x;
             QUEUESIZE++;
             if(FRONT = = -1)
                      FRONT++;
        }
        else
             cout<<"Queue Overflow";
}
int dequeue()
{
        int x;
        if(QUEUESIZE>0)
        {
             x=Num[FRONT];
             FRONT++;
             QUEUESIZE--;

        }
        else
             cout<<"Queue Underflow";
        return(x);
}
```

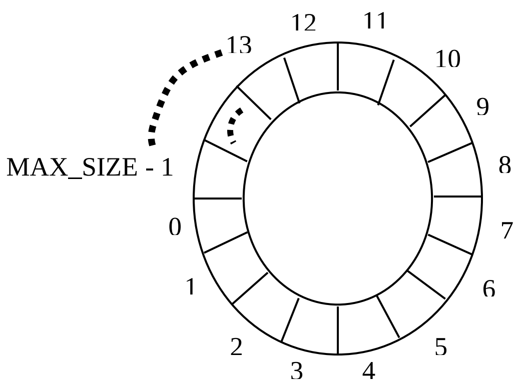
5.2. Circular array implementation of enqueue and dequeue operations

A problem with simple arrays is we run out of space even if the queue never reaches the size of the array. Thus, simulated circular arrays (in which freed spaces are re-used to store data) can be used to solve this problem.

Example: Consider a queue with MAX_SIZE = 4

Operation	Simple array							Circular array						
	Content of the array				Content of the Queue	QUEUE SIZE	Message	Content of the array				Content of the queue	QUEUE SIZE	Message
Enqueue(B)	B				B	1		B				B	1	
Enqueue(C)	B	C			BC	2		B	C			BC	2	
Dequeue()		C			C	1			C			C	1	
Enqueue(G)		C	G		CG	2			C	G		CG	2	
Enqueue (F)		C	G	F	CGF	3			C	G	F	CGF	3	
Dequeue()			G	F	GF	2				G	F	GF	2	
Enqueue(A)			G	F	GF	2	Overflow	A		G	F	GFA	3	
Enqueue(D)			G	F	GF	2	Overflow	A	D	G	F	GFAD	4	
Enqueue(C)			G	F	GF	2	Overflow	A	D	G	F	GFAD	4	Overflow
Dequeue()				F	F	1		A	D		F	FAD	3	
Enqueue(H)				F	F	1	Overflow	A	D	H	F	FADH	4	
Dequeue ()					Empty	0		A	D	H		ADH	3	
Dequeue()					Empty	0	Underflow		D	H		DH	2	
Dequeue()					Empty	0	Underflow			H		H	1	
Dequeue()					Empty	0	Underflow					Empty	0	
Dequeue()					Empty	0	Underflow					Empty	0	Underflow

The circular array implementation of a queue with MAX_SIZE can be simulated as follows:



Analysis:

Consider the following structure: int Num[MAX_SIZE];

We need to have two integer variables that tell:

- the index of the front element
- the index of the rear element

We also need an integer variable that tells:

- the total number of data in the queue

```
int FRONT =-1,REAR =-1;
int QUEUESIZE=0;
```

- To enqueue data to the queue
    - check if there is space in the queue

```
QUEUESIZE<MAX_SIZE ?
Yes:  - Increment REAR
            REAR = = MAX_SIZE ?
                  Yes:   REAR = 0
      - Store the data in Num[REAR]
      - Increment QUEUESIZE
        FRONT = = -1?
            Yes: - Increment FRONT
No:   - Queue Overflow
```

- To dequeue data from the queue
    - check if there is data in the queue

```
QUEUESIZE > 0 ?
Yes:  - Copy the data in Num[FRONT]
      - Increment FRONT
            FRONT = = MAX_SIZE ?
                  Yes:   FRONT = 0
      - Decrement QUEUESIZE
No:   - Queue Underflow
```

Implementation:

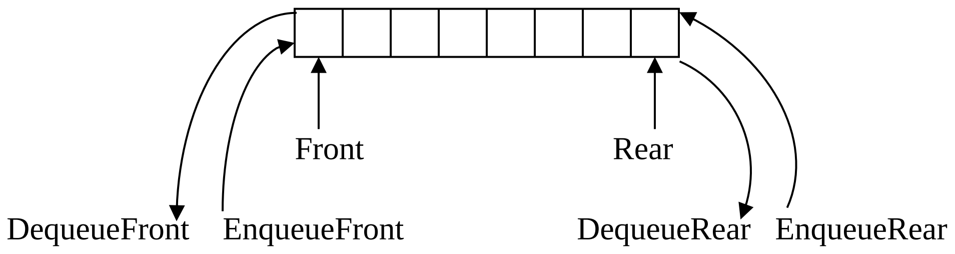
```
const int MAX_SIZE=100;
int FRONT =-1, REAR =-1;
int QUEUESIZE = 0;

void enqueue(int x)
{
      if(QUEUESIZE<MAX_SIZE)
      {
          REAR++;
          if(REAR = = MAX_SIZE)
                 REAR=0;
          Num[REAR]=x;
          QUEUESIZE++;
          if(FRONT = = -1)
                 FRONT++;
      }
      else
          cout<<"Queue Overflow";
}
int dequeue()
{
        int x;
        if(QUEUESIZE>0)
        {
              x=Num[FRONT];
              FRONT++;
              if(FRONT = = MAX_SIZE)
                          FRONT = 0;
              QUEUESIZE--;

        }
        else
              cout<<"Queue Underflow";
        return(x);
}
```

5.3. Linked list implementation of enqueue and dequeue operations

Enqueue- is inserting a node at the end of a linked list
Dequeue- is deleting the first node in the list

5.4. Deque (pronounced as Deck)

- is a Double Ended Queue
- insertion and deletion can occur at either end
- has the following basic operations
    - EnqueueFront – inserts data at the front of the list
    - DequeueFront – deletes data at the front of the list
    - EnqueueRear – inserts data at the end of the list
    - DequeueRear – deletes data at the end of the list
- implementation is similar to that of queue
- is best implemented using doubly linked list



5.5. ***Priority Queue***

- is a queue where each data has an associated key that is provided at the time of insertion.
- Dequeue operation deletes data having highest priority in the list
- One of the previously used dequeue or enqueue operations has to be modified

Example: Consider the following queue of persons where females have higher priority than males (gender is the key to give priority).

Abebe	Alemu	Aster	Belay	Kedir	Meron	Yonas
Male	Male	Female	Male	Male	Female	Male

Dequeue()- deletes Aster

Abebe	Alemu	Belay	Kedir	Meron	Yonas
Male	Male	Male	Male	Female	Male

Dequeue()- deletes Meron

Abebe	Alemu	Belay	Kedir	Yonas
Male	Male	Male	Male	Male

Now the queue has data having equal priority and dequeue operation deletes the front element like in the case of ordinary queues.

Dequeue()- deletes Abebe

Alemu	Belay	Kedir	Yonas
Male	Male	Male	Male

Dequeue()- deletes Alemu

Belay	Kedir	Yonas
Male	Male	Male

Thus, in the above example the implementation of the dequeue operation need to be modified.

5.5.1. Demerging Queues

- is the process of creating two or more queues from a single queue.
- used to give priority for some groups of data

Example: The following two queues can be created from the above priority queue.

Aster	Meron
Female	Female

Abebe	Alemu	Belay	Kedir	Yonas
Male	Male	Male	Male	Male

<u>Algorithm</u>:

```
create empty females and males queue
while (PriorityQueue is not empty)
{
        Data=DequeuePriorityQueue(); // delete data at the front
        if(gender of Data is Female)
                EnqueueFemale(Data);
        else
                EnqueueMale(Data);
}
```

5.5.2. Merging Queues

- is the process of creating a priority queue from two or more queues.
- the ordinary dequeue implementation can be used to delete data in the newly created priority queue.

Example: The following two queues (females queue has higher priority than the males queue) can be merged to create a priority queue.

Aster	Meron
Female	Female

Abebe	Alemu	Belay	Kedir	Yonas
Male	Male	Male	Male	Male

Aster	Meron	Abebe	Alemu	Belay	Kedir	Yonas
Female	Female	Male	Male	Male	Male	Male

<u>Algorithm</u>:

```
create an empty priority queue
while(FemalesQueue is not empty)
        EnqueuePriorityQueue(DequeueFemalesQueue());
while(MalesQueue is not empty)
        EnqueuePriorityQueue(DequeueMalesQueue());
```

It is also possible to merge two or more priority queues.

Example: Consider the following priority queues and suppose large numbers represent high priorities.

ABC	CDE	DEF	FGH	HIJ
52	41	35	16	12

BCD	EFG	GHI	IJK	JKL
47	32	13	10	7

Thus, the two queues can be merged to give the following priority queue.

ABC	BCD	CDE	DEF	EFG	FGH	GHI	HIJ	IJK	JKL
52	47	41	35	32	16	13	12	10	7

5.6. Application of Queues

i. Print server- maintains a queue of print jobs

```
Print()
{
    EnqueuePrintQueue(Document)
}
EndOfPrint()
{
    DequeuePrintQueue()
}
```

ii. Disk Driver- maintains a queue of disk input/output requests

iii. Task scheduler in multiprocessing system- maintains priority queues of processes

iv. Telephone calls in a busy environment –maintains a queue of telephone calls

v. Simulation of waiting line- maintains a queue of persons