Data Structures

1-1) Data Structures

It’s study of the connection way between data programmers and the relation of the information with the devices (especially, computer memory that stored the data).

Data structure are represent the way of distributed information, algorithms to access, and the way of dealing with its, like (adding, deleting, edit, create, sort, search ..., etc).

1-2) Types of Data Structure

There are many languages deals with data, and when use Pascal we use this definitions :-

X: Integer
Y: Real
A: Char
P: Boolean
S: String

when represent the data in computer memory by using simple programming like:

X := X+100
Y := Y+15.6

And by using another’s data we need to use different data structures as:-
1- Array
المصفوفة
2- Record
قيد
3- Linear structures
الهياكل الخطية
- Stack
المكدس
- Queue
الطابور
- Circular Queue
الطابور الدائري
- Linked Structures
الهياكل الموصلة
- Linked Stack
المكدس الموصل
- Linked Queue
الطابور الموصل
4- Non-Linear Structures
الهياكل غير خطية
- Graphs
المخططات
- Directed graph
المخطط المتجه
- Tree structure
هيكل الشجرة
- Undirected graph
المخطط غير المتجه

1-3) How can choice the fixed DS

For every group of data, there are many methods to presentation and put in especial DS, and are limited for some of requirements to choice suite DS:-

1- Size of data.
2- Speed and manner use data.
3- Data dynamic’s, as change and edit.
4- Size of requirement storage.
5- Time to fetch any information from DS.
6- Type of programming.
Array

2-1) **Array**

It’s a set of storage location in the memory, use and classify as:-

1- *All locations are same data type, according to the definition (real, integer, char,...).*

2- *Can random access to any location without depending on any location in Array, the requirement time to access for any location is constant.*

3- *The element’s location of Array are still steady, don’t change when dealing with any elements of Array.*

4- *Array representation as alternate locations in memory.*

2-2) **Representation of One-Dimensional Array**

In Pascal language we can defined array as

VAR X: array [ 1 ... N] of integer {or any other type}

That’s means the structure contains a set of data elements, numbered (N), for example called (X), its defined as type of element, the second type is the index type, is the type of values used to access individual element of the array, the value of index is

\[ 1 \leq I \leq N \]

By this definition the compiler limits the storage region to storing set of element, and the first location is individual element of array , and this called the Base Address, let’s be as 500 , the second address of array, is after Base Address (501) and like for the all elements and used the index I, by its value are range

\[ 1 \leq I \leq N \]

according to Base Index (500), by using this relation:

**Location ( X[I] ) = Base Address + (I-1)**
When the requirement to bounding the forth element (I=4):

Location ( X[4] ) = 500 + (4-1)

= 500 +3

= 503

So the address of forth element is 503 because the first element in 500.

When the program indicate or dealing with element of array in any instruction like (write ( X [I] ), read ( X [I] ) ), the compiler depend on going relation to bounding the requirement address.

2-3) Representation of Two Dimensional Array

There are two method to represent two dimensional array as follows:

1- Row-Wise Method
2- Column-Wise Method

The definition of two dimensional array is:

VAR A:array [ 1….M, 1… N] of integer { or any other type }

That’s mean structure name is A, consist of set of data elements as (M * N), and used two argument for access to the require element as:

1<= I <=M             to bounding row element.

1<= J <=N             to bounding column element.

For example A [3,5], where I=3, J=5, mean that the element located in third row and fifth column.

The compiler depend on one of two method to represent array:
2-3-1) **Row-Wise Method**

In this method, take all elements of the first row (I=1) of an array and store in memory from Base Address, let’s 700:

700  BA  store in A[1,1]
701  BA+1 store in A[1,2]
702  BA+2 store in A[1,3]

Then, take all elements of the second row (I=2) of the array and store in memory, from the address that’s after the last address of the first row.

Then, take all elements of the third row (I=3) of the array and store in memory, from the address that’s after the last address of the second row.

To determine the element address A[i,j]:

\[ \text{Location (A[i,j])} = \text{Base Address} + (N \times (I-1)) + (j - 1) \]

\[ \text{Total columns of array} \quad \text{Number of previous rows to address of require element} \quad \text{Number of previous columns to address of require element} \]

This relation that’s compiler used to determine the require element to handle from every instruction of program instructions.

Ex)

VAR T:array [1..5,1..7] of integer
Calculate address element T[4,6], where BA=900
Sol)
I = 4 , J = 6
M= 5  , N= 7
Location (T[4,6]) = BA + (7 x (4-1)) + (6-1)
= 900 + (7 x 3) + 5
= 900 + 21 + 5
= 926
2-3-2) Column-Wise Method

In this method take all element of first column (j=1) of array and store in memory from Base Address, lets 200:

200      BA             store in A[1,1]
201      BA+1         store in A[2,1]
202      BA+2         store in A[3,1]

Then token all elements of second column J=2 of array and store in memory, from the address that’s after last address of first column.

Then token all elements of third column J=3 of array and store in memory, from the address that’s after last address of second column.

To determine element address A[i,j]:

\[
\text{Location ( } A[ i,j ] \text{ )} = \text{Base Address} + (M \times (j - 1)) + (i - 1)
\]

this relation that’s compiler used to determine the require element to handle from every instruction of program instructions.

Ex)
VAR T:array [1...6,1...8] of integer
Calculate address element T[5,7], where BA=300
Sol)
I = 5   , J = 7
M= 6   , N= 8
Location (T [4,6]) = BA + (6 \times (7-1)) + (5-1)
   = 300+ (6 \times 6) +4
   = 300+ 36+4
   = 340
2-4) **Representation of Three & Four Dimensional Array**

By the same way we can determine address of element for three and four dimensional array:

**Three Dimensional Array**

VAR \( X: \text{array } [1...M,1...N,1...R] \) of integer

To calculate address of element \( X[i,j,k] \) as:

**Rows Method**

Location \((X[i,j,k]) = BA + MN \cdot (k-1) + N \cdot (i-1) + (j-1)\)

**Columns Method**

Location \((X[i,j,k]) = BA + MN \cdot (k-1) + M \cdot (j-1) + (i-1)\)

**Four Dimensional Array**

VAR \( Y: \text{array } [1...M,1...N,1...R,1...P] \) of integer

To calculate address of element \( X[i,j,k,l] \) as:

**Rows Method**

Location \((Y[i,j,k,l]) = BA + MNR \cdot (l-1) + MN \cdot (k-1) + N \cdot (i-1) + (j-1)\)

**Columns Method**

Location \((Y[i,j,k,l]) = BA + MNR \cdot (l-1) + MN \cdot (k-1) + M \cdot (j-1) + (i-1)\)
Ex)
TAB : array [ 1..8, 1..5, 1..7 ] of integer
Calculate address element TAB[5,3,6], by using rows & columns methods, if
BA=900?
Sol)
The dimensions of TAB :
M=8 , N=5, R=7
To compute the location of the element TAB [5,3,6]
This means the indicts
i=5, j=3, k=6

**Rows- wise**
Location (TAB[i,j,k]) = BA + MN(k-1) + N(i-1) + (j-1)
Location (TAB[5,3,6]) = 900 + 8x5(6-1) + 5(5-1) + (3-1)
= 900 + 40 x 5 + 5 x 4 + 2
= 900 + 200 + 20 + 2
= 1122

**Columns- wise**
Location (TAB[i,j,k]) = BA + MN(k-1) + M(j-1) + (i-1)
Location (TAB[5,3,6]) = 900 + 8x5(6-1) + 8(3-1) + (5-1)
= 900 + 40 x 5 + 8 x 2 + 4
= 900 + 200 + 16 + 4
= 1120
Ex)
VAR BOB : array [ 1..4, 1..9, 1..6, 1..8 ] of integer
Calculate address element BOB[3,7,4,5], by using rows &columns methods, if BA=415?
Sol)
The dimensions of BOB are:
M=4, N=9, R=6, P=8
To compute the location of the element TAB [3,7,4,5]
  i=3, j=7, k=4, l=5

**Rows- wise**
Location (BOB[i,j,k,l]) = BA + MNk(l-1) + N(k-1) + M(i-1)+ (j-1)
Location (BOB[3,7,4,5]) = 415 + 4x9x6(5-1) +4x9(4-1) + 9x(3-1)+(7-1)
  = 415 + 216 x 4 +36 x 3 + 9 x 2 + 6
  = 415 + 864 +108 +18 +6
  = 1411

**Columns- wise**
Location (BOB[i,j,k,l]) = BA + MNR(l-1) + MN(k-1) + M(i-1)+ (i-1)
Location (BOB[3,7,4,5]) = 415 + 4x9x6(5-1) +4x9(4-1) + 4x(7-1)+(3-1)
  = 415 + 216 x 4 +36 x 3 + 4 x 6 + 2
  = 415 + 864 +108 +24 +2
  = 1413
List, Stack & Queue

3-1) Linear List

Its set of data elements (item, nodes, elements) are sequential, and connect each items contiguity relation, where each item are foregone another item (except the first item don’t come any item before it, and the last item don’t come any item after it).

If each item as a node, so the list is set of nodes

\[ X[1]. X[2]. X[3] \ldots \ldots \ldots X[k-1]. X[k]. X[k+1] \ldots \ldots X[n] \]

Where first node is \( X[1] \), last node \( X[n] \).

Each set of data and information can called List.

3-1-1) Types of Linear Lists

1- Non-Linear Lists

It’s the lists don’t used pointers, and be as data sequential, used array for representation, and use this type to processing data that’s don’t change dearly, because difficult the operations of delete and add, because of the locations of memory may be busy, so can’t use for adding or deleting.

2- Linked List

It’s the list that’s used pointers, for facilitation to operations of adding, deleting and editing, where for each element has pointer, to limiting location of the next element, and by for existing pointers are cancelled neighbouring storage locations.
3-1-2) The Operations on the Linear Lists

Can implement some of operations to any data structure, when execute its data, and the following types of operations that can be implementing some or all according to the applications:

1- Search
   It's operation of search inside data structure, it's mean the access to element, according to the value of some fields, and called Key Field, that's means the searching is by the contents and not the address.

2- Additional (Inserting)
   To add new node to data structure.

3- Deletion
   It’s delete node from data structure.

4- Merge
   It’s merge two structures or more to constituting one structure.

5- Split
   It’s to divide data structure to two structures or more.

6- Counting
   It’s counting some of items or nodes in data structure.

7- Copying
   It’s copy data of data structure to another data structure.

8- Sort
   It’s sort items or nodes in data structure according to the value of the field or set of fields.

9- Access
   To access from node or item to another one may be need some of purposes to test or change...etc.
3-2) **Stack**

A stack is an order collection of items into which new items may be inserted and from which items may be deleted at one end, and the other end is closed.

![Diagram of a stack with items A, B, C, and D.](image)

Fig. 3-1 illustrate a stack contain the item A.B.C, and when insert new item D, must the insertion from the open end, they will placed on top of C, as shown below:

![Diagram showing the process of adding and deleting items in a stack.](image)

And if we want to delete item from a stack, we must use the same end (open only), and must delete the item D at the first then the item C sequential, and we can’t take item C before take item D, note that item D added at the last, so we can also defined a stack as **(Last In First Out)**
It can’t delete item from mid items stack until delete (pop) items that’s previous from open end (top), with note that the another end is close and don’t never used.

The two change that can be made to a stack are given special names. When an item added to a stack, it’s (Insertion or Push) onto the stack, and when an item is removed, it’s (Deletion or Pop) from the stack.

Ex.)
Let S is (stacking), it’s for push item to the stack, and U is (un stacking), it’s for pop item from the stack, and the set of inputs to the stack are (M, B, Y, N, R), what are the outputs from these operations:

A) SSUUSUSUSU
B) SSSUSUUUU

Sol.)
The meaning the inputs items to the stack are, chose item M first, then item B,...R. we can’t pop N before previous items.

A)
Input -----» M B Y N R
Operations -----» S S U U S U S U S U
Output -----» B M Y N R

B)
Input -----» M B Y N R
Operations -----» S S S U S U U S U U
Output -----» Y N B R M
Ex)
If the set of inputs of a stack are (1.2.3.4.5) show the right outputs shown below according to the stack operation:
A) 2. 4. 5. 3. 1
B) 4. 2. 3. 1. 5
C) 4. 5. 1. 2. 3
D) 4. 3. 5. 2. 1

Sol)
A) Outputs (2. 4. 5. 3. 1)
To pop item 2 must at first push two items 1. 2, the operation is SSU.

```
  1
```

And to pop item 4 after item 2 must at first push two items 3. 4, the operation is SSUSSU.

```
  3
  1
```

To pop item 5. After item 4, must push item 5 them pop item 5, the operation is SSUSSUSU

```
  3
  1
```

According to the state of a stack we can now pop item 3 then item 1, and the operation is now SSUSSUSUUU.
B) Outputs (4. 2. 3. 1. 5)
To pop item 4 must push item 1. 2. 3. 4, according to the operations SSSSU

And to pop item 2 from the stack must pop item 3 at the first, so this sequence of operations are wrong.

C) Outputs (4. 5. 1. 2. 3)
We can pop two items 4 & 5 by this sequence of operations

\[
\begin{array}{c}
\text{Input} \\
\text{Operations} \\
\text{Output}
\end{array}
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 \\
S & S & S & S & U & S & U \\
4 & 5
\end{array}
\]

Now the stack items are:

\[
\begin{array}{c}
3 \\
2 \\
1
\end{array}
\]

We can’t pop item 1 before two items 2 & 3, so this sequence of operations are wrong.

D) Outputs (4. 3. 5. 2. 1)
We can get these outputs when use the operations of pop & push by using this sequences

\[
\begin{array}{c}
\text{Input} \\
\text{Operations} \\
\text{Output}
\end{array}
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 \\
S & S & S & S & U & U & S & U \\
4 & 3 & 5 & 2 & 1
\end{array}
\]
3-2-1) **Array Representation of Stack**

It can applying the stack by using one dimensional array with requirement size, and with data type that store the (Real, Integer.....etc) with use independent variable called (Top), it’s used as pointer, it’s point to the location of high item in the stack (location of nearer item to the open end), at the first the value of pointer (Top=0) when the stack is empty from items, the stack defined as:

```plaintext
Const   size = 10; { or any other value}
Type    Stackelement = integer; {or any other type}
        ST = array [1....size] of stackelement;
Var      stack : ST;
        Top : integer;
```

<table>
<thead>
<tr>
<th>Top=10</th>
<th>J</th>
<th>I</th>
<th>H</th>
<th>G</th>
<th>F</th>
<th>E</th>
<th>D</th>
<th>C</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack full contains 10 elements</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Top=4</th>
<th>D</th>
<th>C</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack contains 4 elements</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Top=1</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack contains 1 elements</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Top=0</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack empty from element</td>
<td></td>
</tr>
</tbody>
</table>

**Push Operation to Stack:**

To executing push operation we must do the following steps:

1- Check the stack is not full, the pointer (Top<>size), to avoidance of over flow.

2- Modify the value of pointer (Top = Top + 1), to point to the next location.

3- Adding new element in new location (Stack [ Top ]).
**Pop Operation from Stack:**
The executing pop operation any element from the stack, must do the following steps:

1- Check the stack is not empty, the pointer (Top<>0), to avoidance of under flow.
2- Take element from location that point to Top and store it temporary as independent variable (Item = Stack [ Top ]
3- Modify the value of pointer (Top = Top - 1), to point to the location of next element for deleted element.

3-2-2) **Stack’s Algorithms**

It can design set of algorithms to coverage stack’s actions, to programming and representation it.

1- **Push Algorithm**

If Stack is full
Then Overflow $\leftarrow$ True
Else
   Overflow $\leftarrow$ False
   Top $\leftarrow$ Top + 1
   Stack [ Top ] $\leftarrow$ New element

2- **Pop Algorithm**

If Stack is Empty
Then Underflow $\leftarrow$ True
Else
   Underflow $\leftarrow$ False
   Element $\leftarrow$ Stack [ Top ]
   Top $\leftarrow$ Top – 1
3- **Stack Full Algorithm** مكدس ملء الخوارزمية
This algorithm to check if the stack is full or not, depending on the value of pointer (Top), before push operations.

If Top = Size
Then Stack full  True

4- **Stack Empty Algorithm** مكدس خلو الخوارزمية
This algorithm to check if the stack is empty or not, depending on the value of pointer (Top), before pop operations.

If Top = 0
Then Stack empty  True

5- **Clear Stack Algorithm** إخلاء المكدس الخوارزمية
This algorithm is use to making the stack and clear it from element by made the value of pointer (Top = 0)

   Top = 0

3-2-3) **Stack’s Procedures and Function**

To design procedure or function for any action or operation of stack’s operations, assist for simplifying and illustrating how programming these operations, and then grouping all in one program, this program is simple and capable for modification and development.

Assume existing this definition in the start of the program to be all subsequent programs is right

Const Size = 20; {or any other integer value}

Type Stackelement = integer; {or any other type}

   St : array [ 1 .. Size ] of stackelement;

Var Stack : ST;

   Top : integer;
   Item : stackelement;
1- **Clear Stack Procedure**  
برنامـج فرـعي لإخلاء المكدـس  
Procedure clearstack (VAR Top : integer)  
Begin  
    Top := 0  
End  
Note that not need to passing to all stack locations and made it equal to zero, and only made the pointer (Top = 0), this procedure are running in the starting work to make the stake empty.

2- **Check Full Stack Procedure**  
برنامـج فرـعي للتحقق من امـتلاء المكدـس  
Function FullStack (Top : integer) : Boolean  
Begin  
    If Top = Size  
    Then FullStack := True  
    Else FullStack := False  
End  
The input of this function is the pointer (Top), and by its value the output is Boolean of (FullStack) and it may be true when the stack is full and false when the stack is empty, this procedure are call in procedure push.

3- **Check Empty Stack Procedure**  
برنامـج فرـعي للتحقق من خـلو المكدـس  
Function EmptyStack (Top : integer) : Boolean  
Begin  
    If Top = 0  
    Then EmptyStack := True  
    Else EmptyStack := False  
End  
The input of this function is the pointer (Top), and by its value the output is Boolean of (EmptyStack) and it may be true when the stack is empty and false when the stack is full, this procedure are call in procedure pop.
4- **Add One Item to Stack Procedure**

Procedure Push (Var Stack : ST; Var Top : integer; Var Item :stackelement);
Begin
  If Fullstack (Top)
    Then writeln (‘ Error... the stack is full’)
  Else
    Begin
      Top := Top + 1;
      Stack [Top ] := Item
    End
End
This procedure add one item to the stack, and it can call in the main program with many times by using some of instructions like (For ... Do) that include reading item then run push procedure to adding to the stack.

5- **Delete One Item to Stack Procedure**

Procedure Pop (Var Stack : ST; Var Top : integer; Var Item :stackelement);
Begin
  If Emptystack (Top)
    Then writeln (‘ Error... the stack is Empty’)
  Else
    Begin
      Item := Stack [Top ];
      Top := Top + 1
    End
End
This procedure take item that’s point to top and copy in the variable (item), to used it later by some processing to satisfy the object of pop this item from the stack.
And to delete or pop more items sequentially from the stack, this procedure will be run many times and in any location in main program.
3-2-4) **Record Implementation of Stack**

In the last application are used array and used definition pointer (Top) as independent variable from array that’s represent a stack, in this application we use (Record) to represented with gather as one data structure, where the record are divided as two part, first part represent a stack, this stack as an array, and the second is field represent pointer (Top), the definition in Pascal language as:

```pascal
Const   Size = 20;  {or any other value}
Type    Stelement = integer;  {or any other type}
        St = Record
        Element : array [1... size] of Stelement;
        Top : 0 ... size
End;

Var   Stack : ST;
       Item : Stelement
```

![Stack Diagram](image)

To add new item to this stack we flow the following steps:

1- Modify the pointer value (Top), where is a field in the record (stack) to be (4)
   
   ```pascal
   Stack . Top = Stack . Top + 1
   ```

2- Add new item (D) in the new location (4)
   
   ```pascal
   Stack . element [Stack . Top] = D
   ```
Q) Rewrite procedures for a stack by using record?

Ex) Write procedure to add 3 element from integer number to stack (SET) with size (20)?

Sol)

Type St : array [ 1.. 20] of integer;
Procedure Push3 (Var SET : St; Var Top : integer);
Var I : integer
Begin
For I := 1 To 3 Do
Begin
If   Top := 20
Then writeln (‘the stack SET is Full’)
Else
Begin
Top := Top + 1;
Write (‘ Enter the element’);
Readln ( Set [Top])
End
End
End
Ex) A stack (TABLE) with size (30) elements contain four elements A, B, C, D, write procedure to add 8 elements?

Sol)

Type St : array [1..30] of char;
Procedure Push8 (Var TABLE : St; Var Top : integer);
Var I : integer
Begin
   Top := 4
   For I := 1 To 8 Do
      Begin
         Top := Top +1;
         Write (' Enter the element');
         Readln ( Table [Top])
      End
   End
End

Ex) Write procedure to delete 4 real number from a stack (BOB) with size (15)?

Sol)

Type St : array [1..15] of real;

Procedure Pop4 (BOB : St; Var Top : integer);
Begin
   For I := 1 To 4 Do
      Begin
         Begin
            If Top := 0
            Then writeln ('Error...the stack SET is Empty')
            Else
               Begin
                  Item := BOB [Top]
                  Top := Top - 1;
               End
         End
      End
End
3-2-5) Stack applications

1- Program Processing contain Procedure

A stack are use from compilers in programs process that contain functions & procedure, and arranging call it by storage Return Address, when call procedure or function inside main program, this need storage address of next instruction after call instruction, so that the main program can running the functions or procedure and return to the step or the next instruction, because address of this location is a Return Address, and it’s stored in the stack.

Assume in this program that include call some of procedure:

Begin {this is the main program}

```
100    Call          A
102

CALL B
202

CALL C
302
```

End

Note that:

1- The main program call procedure A, and procedure A Call procedure B, and B call procedure C.

2- To explain it, assume the address of instruction as:

The instruction’s address of procedure A are 100, where the next instruction (Ret . Add) are 102, and instruction’s address of procedure B are 200, and Ret . Add are 202, and instruction’s address of procedure C are 300, and Ret . Add are 302
Compiler used a stack to process this type from programs, with these methods:

1- When access to call procedure A, and before run call, storage Ret. Add 102 in the stack, the operation is Push.

2- When running instructions of procedure A, we found it’s include instruction of call procedure B, and this require before run storage Ret. Add 202 in the stack, the operation is Push too.

3- When running instructions of procedure B, we found it’s include instruction of call procedure C, and this require before run storage Ret. Add 302 in the stack, the operation is Push too.

4- When finish run the procedure C, the main program need to know the Ret. Add that stored in the stack, and they done by run operation Pop, to get it out and run the instruction that’s located in this address and the address after it inside the procedure B.

5- When finish run the procedure B, the main program need to know the Ret. Add that stored in the stack, and they done by run operation Pop, to get it out and run the instruction that’s located in this address and the address after it inside the procedure A.

6- When finish run the procedure A, the main program need to know the Ret. Add that stored in the stack, and they done by run operation Pop, to get it out and run the instruction that’s located in this address and the address after it inside the main program.

7- The main program continue for running the following instructions, after finished the procedures and a stack become empty.

<table>
<thead>
<tr>
<th>Stack</th>
<th>Call</th>
<th>Call</th>
<th>Call</th>
<th>Finish</th>
<th>Finish</th>
<th>Finish</th>
</tr>
</thead>
</table>

& back to M.P
Ex) Draw all states of stack when run program?

Begin { main program }

100 CALL X
102 ___
200 CALL Y
202 ___
___ 400 CALL P
___ 402 ___
___ ___ 600 CALL R
___ ___ 602 ___
___ ___ ___ 700 CALL S
___ ___ ___ 702 ___
___ 500 CALL Q
___ 502 ___
300 CALL Z
302 ___

End
2- Use Stack to process Arithmetic Expressions

There are three expressions:

A) Infix Notation
   The sign of arithmetic operation in the mid of operands:
   X / 20, A - B, 3 + 4.

B) Prefix Notation
   The sign of arithmetic operation is before the operands:
   / X 20, - A B, + 3 4, and this called (Polish Notation).

C) Postfix Notation
   The sign of arithmetic operation is after the operands:
   X 20 /, A B -, 3 4 +, and this called (RPN) Reverse Polish Notation.

Note

To run any arithmetic expressions with Infix notation, then the operations run from left to right:

0 (  
1 ^ (power), Unary (-), Unary (+), Not 
2 * /, AND, DIV, MOD 
3 + -, OR 
4 =, <, >, <=, >= 
5 )

The program that’s include arithmetic expression with infix notation, compiler are convert to postfix notation by using stack according to this algorithm:

Convert Infix Notation to Postfix Notation Algorithm

1- Use two stacks, the first stack (ST1), to storage variables (operands), and in the last step will collect the final notation (Postfix), the second stack are use to storage the sign of arithmetic expression (Operators).
2- Check the arithmetic expression code by code from left to right.
3- In any code we do the following:
If code is
Some operands
we do
push in ST1

( push in ST2

) pop all code from ST1 and push in ST2
Sequentially, to access to the left arc that
Must be cancel with right arc.

Operator pop all arithmetic expression (if found) in
ST2, that’s priority are high or equal to
priority of current arithmetic expression
and storage it in ST1 (stop when not realize
condition), then store new operation in ST2.

4- When finish all code of arithmetic expression will pop all remains codes in
ST2 sequentially then push in ST1, that contained the final notation
(Postfix).

Ex) Convert arithmetic expression from infix to postfix?

\[ a - b * ( c + d ) / ( e - f ) ^ g * h \]

sol)

<table>
<thead>
<tr>
<th>Step No.</th>
<th>Enter Code</th>
<th>ST1</th>
<th>ST2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>a</td>
<td>.....</td>
</tr>
<tr>
<td>2</td>
<td>_</td>
<td>a</td>
<td>_</td>
</tr>
<tr>
<td>3</td>
<td>b</td>
<td>ab</td>
<td>_</td>
</tr>
<tr>
<td>4</td>
<td>*</td>
<td>ab</td>
<td>_ *</td>
</tr>
<tr>
<td>5</td>
<td>(</td>
<td>ab</td>
<td>_ * (</td>
</tr>
<tr>
<td>6</td>
<td>c</td>
<td>abc</td>
<td>_ * (</td>
</tr>
</tbody>
</table>
7    +    abc    _ * ( +
8    d    abcd    _ * ( +
9    )    abcd+    _ *

Note that when access to right arc, we pop all codes until access to left arc from ST2 to ST1, with pop the two arc (left and right) and then cancel them.

10    /    abcd+*    _ /
11    (    abcd+*    _ / ( 
12    e    abcd+*e    _ / ( 
13    _    abcd+*e    _ / ( _
14    f    abcd+*ef    _ / ( _
15    )    abcd+*ef_    _ /
16    ^    abcd+*ef_    _ / ^
17    g    abcd+*ef_g    _ / ^
18    *    abcd+*ef_g^/    _ *
19    h    abcd+*ef_g^/h    _ *

Now all input are finish, so transfer the remain from ST2 to ST2

20    ....    abcd+*ef_g^/h*_    ....
Calculate the Value of Converted Arithmetic Expression to postfix

After compiler converted the arithmetic expression from infix to postfix, the calculation this value in the second step are according to this algorithm by using one stack:

**Algorithm**

1- Using one stack ST.
2- Check (take) arithmetic expression code by code from the left to right as:

**If code is**  \[ \text{we do} \]

**Some operands**  \[ \text{push in ST} \]

**Operator**  \[ \text{run this operation on two operands in the top of stack, pop two operands from stack ST and then run the operation on the operands, and storage (push) the result in the stack ST.} \]

3- When finish all inputs of arithmetic expression, the remain value in the stack are the final result for arithmetic expression.
Ex) 7+ 8 – 6 * 3/2 is in infix and when convert to postfix become 78+63*2/-?

sol) to calculate the value of this final notation we do the following steps:

<table>
<thead>
<tr>
<th>Step No.</th>
<th>Enter Code</th>
<th>ST</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>7 8</td>
</tr>
<tr>
<td>3</td>
<td>+</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>15 6</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>15 6 3</td>
</tr>
<tr>
<td>6</td>
<td>*</td>
<td>15 18</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>15 18 2</td>
</tr>
<tr>
<td>8</td>
<td>/</td>
<td>15 9</td>
</tr>
<tr>
<td>9</td>
<td>-</td>
<td>6</td>
</tr>
</tbody>
</table>

Note that run the operation + for two operands in the stack (7,8) and storage the result of summation (15) instead of them in the stack.

Note that run operation * of two operands (6,3), and store the result (18) instead of them in the stack.

Note that run operation / of two operands (18,2), and store the result (9) instead of them in the stack.

Note that run operation - of two operands (15,9), and store the result (6) instead of them in the stack.
Algorithm of Calculate the Value (Run) Infix Arithmetic Expression

The another applications of stack, are used in Interpreters, to calculate the value of infix, with out to convert to postfix.

Steps of Algorithm

1- Use two stacks, the first stack (ST1), to storage operands, and the second stack (ST2) are use to storage the sign of arithmetic expression (Operators).

2- Check the arithmetic expression code by code from left to right.

3- In any code we do the following:

<table>
<thead>
<tr>
<th>If code is</th>
<th>we do</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operands</td>
<td>push in ST1</td>
</tr>
<tr>
<td>Operator</td>
<td>pop all arithmetic expression (if found) in ST2, that’s priority are high or equal to priority of current arithmetic expression and run each of operands in top of stack (ST1) and storage (push) the result in ST1. And after this operation push the sign of new arithmetic expression in the ST2.</td>
</tr>
</tbody>
</table>

4- When finish all code of arithmetic expression, will start to run all remains arithmetic operations in ST2 sequentially of two operands in top of stack ST1, and replace the result in the same stack ST1, and continue for iteration for this step until made ST2 empty, and the last value in ST1 is the final value (result).
Ex) Find the value of arithmetic expression in infix by using stack?

\[ 3 + 7 \times 2 - 6 \]

sol)

<table>
<thead>
<tr>
<th>Step No.</th>
<th>Enter Code</th>
<th>ST1</th>
<th>ST2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>3</td>
<td>.....</td>
</tr>
<tr>
<td>2</td>
<td>+</td>
<td>3</td>
<td>+</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>3 7</td>
<td>+</td>
</tr>
<tr>
<td>4</td>
<td>*</td>
<td>3 7</td>
<td>+ *</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>3 7 2</td>
<td>+ *</td>
</tr>
<tr>
<td>6</td>
<td>-</td>
<td>17</td>
<td>-</td>
</tr>
</tbody>
</table>

Note there the operation \* of two operands (7,2) and the result are 14 because the priority >= from new operation -, then continue to run operation + on the result (14) and the value (3) to get (17), to finish arithmetic expression that’s priority >= the priority new operation, store the sign of this operation (-) in ST2

<table>
<thead>
<tr>
<th>Step No.</th>
<th>Enter Code</th>
<th>ST1</th>
<th>ST2</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>6</td>
<td>17 6</td>
<td>-</td>
</tr>
</tbody>
</table>

When finish all codes of input arithmetic operation, will start to run remains arithmetic operation in ST2 sequentially of contents of ST1, and the final step are:

<table>
<thead>
<tr>
<th>Step No.</th>
<th>Enter Code</th>
<th>ST1</th>
<th>ST2</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>.....</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

The remain value in ST1 are 11.

Ex) Convert arithmetic expression from infix to postfix?

\[ M := X / 6 + ( a - 2 \times ( b / 3 ) ^ 5 + f ) ^ 2 \]

sol)

<table>
<thead>
<tr>
<th>Step No.</th>
<th>Enter Code</th>
<th>ST1</th>
<th>ST2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>M</td>
<td>M</td>
<td>.....</td>
</tr>
<tr>
<td>2</td>
<td>:=</td>
<td>M</td>
<td>:=</td>
</tr>
</tbody>
</table>
3    X    MX       :=
4    /    MX       :=/
5    6    MX6      :=/
6    +    MX6/     :=+
7    (    MX6/     :=+(  
8    a    MX6/a    :=+(  
9    -    MX6/a    :=+(  
10   2    MX6/a2    :=+(  
11   *    MX6/a2   :=+(  
12   (    MX6/a2   :=+(  
13   b    MX6/a2b   :=+(  
14   /    MX6/a2b   :=+(  
15   3    MX6/a2b3  :=+(  
16   )    MX6/a2b3/ :=+(  
17   ^    MX6/a2b3/ :=+(  
18   5    MX6/a2b3/5 :=+(  
19   +    MX6/a2b3/5^+ :=+(  
20   f    MX6/a2b3/5^+f :=+(  
21   )    MX6/a2b3/5^+f+ :=+
22   ^    MX6/a2b3/5^+f+ :=+^  
23   2    MX6/a2b3/5^+f+ :=+^  
24   ....   MX6/a2b3/5^+f+:^+:=   ....
Another Application

The stack are used as a structure to stored the information, that’s need to return as reverse (reverse order), and the cases that’s need to go back to back tracking, and example of it (Amazing Problem), when passing for special location and there are some paths, is assume one choice to access to the goal, then this need store this location before leaving, and test another path, so need to return back to this location if this path is wrong.

Using stack in this cases, allow to store previous series location, where can to return to it, invert the passing arrange.

Ex) Write algorithm to read string finished with char (.), then print reverse order by using stack.

Sol) Algorithm

Begin
  Clear the Stack
  Repeat
    Read a character
    If character <> ‘.’
      Then push the character onto stack
  Until character = ‘.’
  While stack is not empty Do
    Begin
      Pop the stack
      Print the character
    End
End
Ex) Depend on the algorithm of last example and write procedure to read string, and print reverse order by using stack.

Sol)

Procedure PrintRevers;

Const Dot = ‘.’;

Var character : char; stack : st; top : integer;

Begin
    Clearstack (Top);
    Repeat
        Read (character);
        If character <> Dot
        Then push (stack, top, character)
    Until character = Dot
    While Not Emptystack (top) Do
        Begin
            Pop (stack, top, character);
            write (character)
        End
    End
End;
Program for Stack representation and it’s operation

PROGRAM STACK
USES CRT;
CONST SIZE=10;
TYPE ELEMENT=INTEGER; {OR ANY OTHER TYPE}
   ST=ARRAY [1...SIZE] OF ELEMENT;
VAR STACK1:ST;
   ITEM:ELEMENT;
   TOP1,CHOICE,I,L,M:INTEGER;
FUNCTION FULLSTACK (TOP:INTEGER):BOOLEAN;
BEGIN
   IF TOP=SIZE
       THEN FULLSTACK:=TRUE
       ELSE   FULLSTACK:=FALSE
END;
FUNCTION EMPTYSTACK (TOP:INTEGER):BOOLEAN;
BEGIN
   IF TOP=0
       THEN EMPTYSTACK:=TRUE
       ELSE   EMPTYSTACK:=FALSE
END;
PROCEDURE PUSH (Var STACK:St; Var TOP:INTEGER; ITEM:ELEMENT);
BEGIN
   IF FULLSTACK (TOP)
       THEN WRITELN (‘ERROR...THE STACK IS FULL’)
   ELSE BEGIN
       TOP:=TOP+1;
       STACK[TOP]:=ITEM
   END
END;
PROCEDURE POP (Var STACK:St; Var TOP:INTEGER; ITEM:ELEMENT);
BEGIN
   IF EMPTYSTACK (TOP)
       THEN WRITELN (‘ERROR...THE STACK IS EMPTY’)
   ELSE BEGIN
       ITEM:= STACK[TOP]
       TOP:=TOP-1;
   END
END;
BEGIN {MAIN PROGRAM}
CLRSCR;
TOP1:=0;
REPEAT

   WRITELN('REPRESENTATION OF STACK OPERATIONS');
   WRITELN('_________________________________');
   WRITELN('1- INSERTION OPERATIONS ( PUSH )');
   WRITELN('2- DELETION OPERATIONS ( POP )');
   WRITELN('3- DISPLAY THE CONTENT OF THE STACK ');
   WRITELN('4- EXIT ');
   WRITELN;
   WRITELN('SELECT YOUR CHOICE');
   READLN(CHOICE);
   CASE CHOICE OF
      1:BEGIN
         WRITELN('HWOMAY ELEMENTS YOU LIKE TO ENTER ?');
         READLN(M);
         FOR I:= TO M DO
            BEGIN
               WRITELN('ENTER THE NEW ELEMENTY :');
               READLN(ITEM1);
               PUSH (STACK1, TOP1, ITEM1)
            END
      END;
      2:BEGIN
         WRITELN('HWOMAY ELEMENTS YOU LIKE TO DELETE ?');
         READLN(L);
         FOR I:= TO L DO POP(STACK1,TOP1,ITEM1)
      END;
      3:BEGIN
         WRITELN('THE CONTENT OF THE STACK IS :')
         WRITELN('TOP=',TOP1,'--->'),
         FOR I:= TOP1 DOWNOT 1 DO WRITELN (STACK1[1]:15);
         WRITELN;
      END;
      4:END;
   UNTIL CHOICE=4
END;
Program to read string and print it reverse order by using Stack

PROGRAM Reversst;
USES CRT;
CONST SIZE=10; {OR ANY OTHER VALUE}
TYPE ELEMENT=CHAR;
    ST=ARRAY [1...SIZE] OF ELEMENT;
VAR STACK1:ST;
    ITEM1:ELEMENT;
    TOP1,I:INTEGER;
FUNCTION FULLSTACK (TOP:INTEGER):BOOLEAN;
BEGIN
    IF TOP=SIZE
    THEN FULLSTACK:=TRUE
    ELSE   FULLSTACK:=FALSE
END;
FUNCTION EMPTYSTACK (TOP:INTEGER):BOOLEAN;
BEGIN
    IF TOP=0
    THEN EMPTYSTACK:=TRUE
    ELSE   EMPTYSTACK:=FALSE
END;
PROCEDURE PUSH (Var STACK:St; Var TOP:INTEGER; ITEM:ELEMENT);
BEGIN
    IF FULLSTACK (TOP)
    THEN WRITELN ('ERROR...THE STACK IS FULL')
    ELSE BEGIN
        TOP:=TOP+1;
        STACK[TOP]:=ITEM
    END
END;
PROCEDURE POP (Var STACK:St; Var TOP:INTEGER; ITEM:ELEMENT);
BEGIN
    IF EMPTYSTACK (TOP)
    THEN WRITELN ('ERROR...THE STACK IS EMPTY')
    ELSE BEGIN
        ITEM:= STACK[TOP]
        TOP:=TOP-1;
    END
END;
BEGIN {MAIN PROGRAM}
CLRSCR;
TOP1:=0;
  WRITELN(‘THIS PROGRAM READS IN ANY STRING AND PRINTS’);
  WRITELN(‘IN REVERSE ORDER USING STACK’);
  WRITELN;
  WRITELN(‘INPUT YOUR STRING TERMINATED BY .’);
  ITEM1:='A';
  WHILE ITEM <> ‘.’ DO
    BEGIN
      READLN(ITEM1);
      PUSH (STACK1, TOP1, ITEM1)
      END;
  TOP1:=TOP-1;
  WRITELN(‘YOUR STRING IN REVERSE ORDER’);
FOR I:= TOP1 DOWNTO 1 DO
  BEGIN
    POP(STACK1, TOP1, ITEM1);
    WRITE(ITEM1)
  END
END;
Program to read string and tested if it palindrome or not, i.e. it can be read from both sides by using stack.

PROGRAM PALINDST
USES CRT;
CONST SIZE=30; {OR ANY OTHER VALUE}
TYPE ELEMENT=CHAR;
    ST=ARRAY [1...SIZE] OF ELEMENT;
VAR STACK1,STACK2:ST;
    ITEM1,CH1,CH2:ELEMENT;
    TOP1,TOP2,COUNT,I:INTEGER;
    PALINDROME:BOOLEAN;
FUNCTION FULLSTACK (TOP:INTEGER):BOOLEAN;
BEGIN
    IF TOP=SIZE
    THEN FULLSTACK:=TRUE
    ELSE   FULLSTACK:=FALSE
END;
FUNCTION EMPTYSTACK (TOP:INTEGER):BOOLEAN;
BEGIN
    IF TOP=0
    THEN EMPTYSTACK:=TRUE
    ELSE   EMPTYSTACK:=FALSE
END;
PROCEDURE PUSH (Var STACK:St; Var TOP:INTEGER; ITEM:ELEMENT);
BEGIN
    IF FULLSTACK (TOP)
    THEN WRITELN (‘ERROR...THE STACK IS FULL’)
    ELSE BEGIN
        TOP:=TOP+1;
        STACK[TOP]:=ITEM
    END
END;
PROCEDURE POP (Var STACK:St; Var TOP:INTEGER; ITEM:ELEMENT);
BEGIN
    IF EMPTYSTACK (TOP)
    THEN WRITELN (‘ERROR...THE STACK IS EMPTY’)
    ELSE BEGIN
        ITEM:= STACK[TOP]
        TOP:=TOP-1;
    END
BEGIN {MAIN PROGRAM}
CLRSCR;
TOP1:=0; TOP:=0; COUNT:=0; PALINDROME:=TRUE
WRITELN('THIS PROGRAM CAN READS IN ANY STRING AND TESTED IF ITS');
WRITELN('PALINDROME OR NOT(i.e. IT CAN BE READ FROM BOTH SIDES');
WRITELN;
WRITELN('INPUT YOUR STRING TERMINATED BY THE CHAR (.);
CH1:=A;
WHILE CH <> ' ' DO
BEGIN
READ(CH1);
PUSH(STACK1, TOP1, CH1);
COUNT:=COUNT-1
END;
COUNT:=COUNT-1;
POP(STACK1, TOP1, CH1); {GETRID OF THE.''}
FOR I:=1 TO (COUNT DIV 2) DO
BEGIN
POP(STACK1, TOP1, CH1);
PUSH(STACK2, TOP2, CH1);
END;
IF (COUNT MOD 2)=1
THEN POP(STACK1, TOP1, CH1);
WHILE NOT EMPTYSTACK (TOP1) AND PALINDROME DO
BEGIN
POP(STACK1, TOP1, CH1);
POP(STACK2, TOP2, CH2);
IF CH1 <> CH2
THEN PALINDROME:= FALSE
END;
IF PALINDROME
THEN WRITELN('THE STRING IS PALINDROME');
ELSE WRITELN('THE STRING IS PALINDROME');
END;
Queue

3-3) Queue

It’s a sequential structure, there are the operations of insertion from the rear end, and deletion from front end, as example:

```
   6   5   4   3   2   1
  C   B   A
```

Where the element A in the front of queue, then element B and element C, and when add new element, they will place after element C, but when delete element from the queue, the operation of deletion from front end, mean that delete element A, and the queue become:

```
   6   5   4   3   2   1
  D   C   B
```

We see that the queue are useful for operation including scheduling of jobs, according to arrangement of arrival or request, so we can also defined a queue as (First In First Out)

Mean that, the first arrive get the service first, the operation of adding called (ENQueue) or (Insertion), but the operation of deletion called (DEQueue) or (Deletion).
3-3-1) **Array Representation of Queue**

it can applying the queue by using one dimensional array with requirement size, and with data type that store the (Real, Integer.....etc) by use:

- Rear variable : use as pointer to the last element in queue.
- Front variable : use as pointer to the first element in queue.

The value of pointers when the queue empty (Rear=0/Front=0).

Execution of addition operation element to the queue, after upgrade the value of rear pointer, to refer to the new location, after the last element. Execution of deletion operation element from the queue, after upgrade the value of front pointer, to refer to the next location, after delete element in the front.

<table>
<thead>
<tr>
<th>State of Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty Q</td>
</tr>
<tr>
<td>Add item A</td>
</tr>
<tr>
<td>Add item B</td>
</tr>
<tr>
<td>Add item C</td>
</tr>
<tr>
<td>Delete item</td>
</tr>
<tr>
<td>Add item D</td>
</tr>
<tr>
<td>Add item E</td>
</tr>
<tr>
<td>Delete item</td>
</tr>
<tr>
<td>Delete item</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>A</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>A</td>
<td>B</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>-</td>
<td>B</td>
<td>C</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>-</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>-</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>C</td>
<td>D</td>
<td>E</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>D</td>
<td>E</td>
<td>-</td>
</tr>
</tbody>
</table>

The queue defined as:

```
Const    size = 10; { or any other value}
Type     Queueelement = char; {or any other type}
         Q = array [1....size] of queueelement;
Var      Queue : Q; Rear , Front : integer;
```
**Add to Queue:**

Do the following steps to add one element to the queue:

1. Check the queue is not full, the pointer (Rear<>size), to avoidance of over flow.
2. Modify the value of pointer (Rear = Rear + 1), to refer to the next location.
3. Adding new element in new location (Queue [ Rear ].

**Delete from Queue:**

Do the following steps to delete one element from queue:

1. Check the queue is not empty, the pointer (Front<>0), to avoidance of under flow.
2. Take element from location that refer to Front and store it temporary as independent variable (Item = Queue [ Front ]
3. Modify the value of pointer (Front = Front + 1), to refer to the location of next element for deleted element.

**3-3-2) Queue’s Algorithms**

There are set of algorithms that coverage queue’s actions:

1. **Add Queue Algorithm**
   
   If Queue is full
   Then Overflow ← True
   Else
   
   Overflow ← False
   Rear ← Rear + 1
   Queue [ Rear ] ← New element

2. **Delete Queue Algorithm**
   
   If Queue is Empty
   Then Underflow ← True
   Else
   
   Underflow ← False
   Element ← Queue [Front ]
   Front ← Front + 1
3- **Full Queue Algorithm**  
This algorithm to check if the queue is full or not, depending on the value of pointer (Rear), before add operations.
If Rear = Size
Then Full Queue ← True
Else Full Queue ← False

4- **Empty Queue Algorithm**  
This algorithm to check if the queue is empty or not, depending on the value of pointer (Front), before delete operations.
If Front = 0
Then Empty Queue ← True
Else Empty Queue ← False

5- **Clear Queue Algorithm**  
This algorithm is use to making the queue and clear it from element by made the value of pointers (Front = 0, Rear = 0 )
Front = 0
Rear = 0

3-3-3) **Queue Procedures and Function**  
There are set of procedure or function are write by the same way of stack’s procedure, with assume existing this definition in the start of the program to be all subsequent programs is right

Const Size = 10; {or any other integer value}
Type Queueelement := char; {or any other type}
Q := array [ 1 .. Size ] of Queueelement;
Var Queue : Q;
Rear, Front : integer;
Item : queueelement;
1- **Clear Queue Procedure** برنامج فرعي لإخلاء الطابور

Procedure clear Queue (VAR Front, Rear : integer);

Begin
    Rear := 0
    Front := 0

End

Note that not need to passing to all array locations and made the value of two pointers equal to zero, this procedure are running in the starting work to make the queue empty.

2- **Check Full Queue Procedure** برنامج فرعي للتحقق من امتلاء الطابور

Function FullQueue (Rear : integer) : Boolean

Begin
    If    Rear = Size
    Then  Fullqueue := True
    Else   Fullqueue := False

End

The input of this function is the pointer (Rear), and by its value the output is Boolean of (FullQueue) and it may be true when the queue is full and false when the queue is empty, this procedure are call in procedure InsertionQueue.

3- **Check Empty Queue Procedure** برنامج فرعي للتحقق من خلو الطابور

Function EmptyQueue (Front : integer) : Boolean

Begin
    If    Front = 0
    Then  Emptyqueue := True
    Else   Emptyqueue := False

End

The input of this function is the pointer (Front), and by its value the output is Boolean of (EmptyQueue) and it may be true when the queue is empty and false when the stack is full, this procedure are call in procedure Deletequeue.
4- **Add One Item to Queue Procedure**

Procedure AddQueue(Var Queue: Q; Var Front, Rear: integer;
Var Item: Queueelement); Begin

  If Fullqueue (Rear)
  Then writeln (‘ Error... the queue is full’)
  Else
    Begin
      Rear := Rear + 1;
      Queue [Rear] := Item
    End
  If Front = 0
  Then Front := 1
End

It can call in the main program with many times by using some of instructions like (For ... Do) depend on the number of items required to add.

5- **Delete One Item from Queue Procedure**

Procedure DeleteQueue(Var Queue: Q; Var Front, Rear: integer;
Var Item: Queueelement); Begin

  If Emptyqueue (Front)
  Then writeln (‘ Error... the queue is empty’)
  Else
    Item := Queue [Front];
    If Front = Rear
    Then
      Begin
        Front := 0;
        Rear := 0;
        Front, Rear
      End
    Else
      Front := Front + 1;
End

It can call in the main program with many times, depend on the number of items required to delete.
3-3-4) Record Representation of Queue

The record are used to representation the queue, and the two pointers (Front, Rear) in one data structure, then the record are consist of three parts, first part represent array of queue, second part is the field represent pointer (Front), and the last part represent the pointer (Rear).

The definition in Pascal language as:

```pascal
Const   Size = 10;  {or any other value}
Type    Qelement = char; {or any other type}
Q  =  Record
    Elements : array [1... size] of Qelement;
    Front : integer;
    Rear : integer;
End;
Var   Queue : Q;
    Item : Qelement

Queue :

| A | B | C | D |   |   | 1 | 4 |

Element

( Queue . element )

Front

(Queue . Front)

Rear

(Queue . Rear)
```

To add new item to this queue we flow the following steps:

1- Modify the pointer value (Top)(its field in record Queue) and become (5).
   
   Queue. Rear := Queue. Rear +1

2- Add new item (E) in the new location (5)
   
   Queue . element [Queue. Rear] := E
When delete item from queue we flow the following steps:

1- Take element from front of queue, as pointer are refer to it.
   Item := Queue. Element [Queue. Front]

2- Modify the pointer value (Front) from 1 to 2.
   Queue. Front := Queue. Front +1

3-3-5) Queue applications

The common applications of queue in computer field are:

- Job scheduling, in the system of batch processing, the required jobs are arrange according to the access time, and then run sequentially.
- Operating systems are used queue in scheduling of using the different computer resources, the queue’s jobs that need time to output in the printer, and the another to input and use disk.

Ex) Write procedure to add 5 element to queue (Line) with size (25) integer number?

Sol)

```
Const Size = 25;
Type element = integer;
Q : array [1..size] of integer;
Procedure AddQueue (Var line : Q; Var Front, Rear : integer);
Var I : integer; Item : element;
Begin
```
For \( I := 1 \) To 5 Do
Begin
If Rear = size
Then writeln (‘Error ..the queue is Full’)
Else
Begin
Rear := Rear + 1;
Read (Item);
Line (Rear) := Item
End;
If Front = 0;
Then Front = 1;
End;
End;

Ex) Write algorithm to read string consist of two substring separated by ‘.’, then check there are symmetric or not?

Sol)

Algorithm
Begin
Clear the Queue
Stringmatch \( \leftarrow \) True
Repeat
Read a character and put in the string
If character\( \neq \) ‘.’
Then EnQ the character
Until character = ‘.’
While ( more element in the queue ) Do
Dequeue an element
Read the next character and place in string
If Dequeue character \( \neq \) character
Then Stringmatch \( \leftarrow \) false
End
3-4) **Circular Queue**

We noted that when add element to the queue are need check the value of pointer \( \text{Rear} = \text{size} \), mean’s the queue are full, even it has free locations in the front of queue as shown below:

```
8 7       6                5                  4                 3                 2                1
Y X N M
```

\( \text{Rear} = 8 \)                             \( \text{Front} = 5 \)

So we lose storage space without use, and to prevent this case we use queue as a circular, so we allow to the pointer \( \text{Rear} \) to rotate to the front end of the queue \( \text{warp-around} \), then there are free location in the queue.

Properties of this queue:

1- The front pointer are refer to the location whose front the first element in the queue.
2- The rear pointer are refer to the location of last element in the queue.
3- When the rear pointer arrive to the last location in the queue \( \text{Rear}=\text{size} \) we make it rotate to the beginning \( \text{Rear}=1 \), and so that to the front pointer.
4- The large number of element that storing queue are \( \text{size}-1 \), because the front pointer are refer to the free location front the first element in the queue.

**First case**

Queue has three element A,B,C.

```
8 7 6 5 4 3 2 1
```

\( \text{Rear} = 4 \)                             \( \text{Front} = 1 \)
Second case
After add elements D,E,F.

Rear = 7                        Front = 1

Third case
After delete elements A,B,C,D.

Rear = 7    Front = 5

Fourth case
After add element G.

Rear = 8    Front = 5

We note that, this queue are full because (Rear=size=8), and there are free locations in the front, but in the circular queue we can add element as shown below:
Fifth case

To add M element we note rotate the rear pointer to the front end after arrive to the last location in the queue

8 7 6 5 4 3 2 1

G F E M

Front = 5 Rear = 1

Sixth case

After add elements X,P,N, note there the queue become full, because of when add element it must modify the value of rear pointer to become equal to (5), and its equal to the value of pointer (Front=5), that must be stay one free location, as the size of the queue are (size-1) (8-7=1), and this is the contain the queue, so we can’t add in this case.

8 7 6 5 4 3 2 1

G F E X P N M

Front = 5 Rear = 4

Procedure to add element to the circular queue

Procedure AddCQ( Var Queue :Q; Var Front, Rear: integer; Var Item: Qelement)
Begin
If Rear = size Then Rear :=1 Else Rear := Rear + 1;
If Rear = Front Then writeln (‘Error.. The CQ is full’) Else Queue [Rear] := Item
End;
Procedure to delete element from the circular queue

Procedure DeleteCQ(Var Queue:Q; Var Front, Rear:integer; Var Item:Qelement)
Begin
  If Front = Rear
    Then writeln (‘Error.. The CQ is Empty’)
    Else
      Begin
        If Front = size
          Then Front :=1
          Else Front := Front + 1;
        Item := Queue [Front]
      End;
End;

Illustrating
Let take circular queue has size 8 elements, contained one element is X as shown below, and the value of Front=4 and value of Rear=5

```
8 7 6 5 4 3 2 1
```

Rear = 5        Front = 4

When delete one element:
- Modify the value pointer(Front := Front+1) then get (Front=5).
- Take this item from his location { Item = Q [Front].
- The queue become as shown:

```
8 7 6 5 4 3 2 1
```

Rear = 5        Front = 5

(Rear=Front=5) and the queue are empty from elements, so any operation of extension delete as the procedure of (DeleteCQ), can’t executing.
Important Note
We found the same condition (Rear:=Front) are used in the procedure (AddCQ),
to mean the queue are full, and in the procedure (DeleteCQ) to mean the queue
are empty, and this not mean the gainsay, because of the sequence of steps are
different, when add appear after modify the value of pointer rear, and appear
when delete in the beginning.

3-5) Double Ended Queue (Dequeue)
Is a linear structure, that can add or delete the element from any end, and
represent in the one dimensional array, with four pointers are:
- F1 are refer to the location of the first element in the queue, when used
  from the right side.
- R1 are refer to the location of the last element in the queue, when used
  from the right side.
- F2 are refer to the location of the first element in the queue, when used
  from the left side.
- R2 are refer to the location of the last element in the queue, when used
  from the left side.

\[
\begin{array}{cccccccc}
8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 \\
\hline
\text{T} & \text{N} & \text{M} & \text{X} & \\
\end{array}
\]

It can delete element X by using pointer F1, and add element after element T by
using pointer R1, so can delete element T by using pointer F2, and add element
after element X by using pointer R2.
Program for Queue representation and its operation

PROGRAM QUEUE
USES CRT;
CONST SIZE=10;
TYPE ELEMENT=INTEGER; {OR ANY OTHER TYPE}
   QT=ARRAY [1..SIZE] OF ELEMENT;
VAR QUEUE:QT;
   ITEM:ELEMENT;
   FRONT1,REAR1,CHOICE,I,L,M:INTEGER;
FUNCTION FULLQUEUE (REAR:INTEGER):BOOLEAN;
BEGIN
   IF  REAR=SIZE
      THEN  FULLQUEUE:=TRUE
      ELSE    FULLQUEUE:=FALSE
END;
FUNCTION EMPTYQUEUE (FRONT:INTEGER):BOOLEAN;
BEGIN
   IF FRONT=0
      THEN EMPTYQUEUE:=TRUE
      ELSE   EMPTYQUEUE:=FALSE
END;
PROCEDURE ADDQ (Var Q:Qt; Var FRONT,REAR:INTEGER; Var ITEM:ELEMENT);
BEGIN
   IF FULLQUEUE (REAR)
      THEN WRITELN ('ERROR...THE QUEUE IS FULL')
   ELSE BEGIN
      REAR:=REAR+1;
      Q[REAR]:=ITEM
   END
   IF FRONT =0
      THEN FRONT := 1
END;
PROCEDURE DELETEQ (Var Q:Qt; Var FRONT,REAR:INTEGER; Var ITEM:ELEMENT);
BEGIN
   IF EMPTYQUEUE (FRONT)
      THEN WRITELN ('ERROR...THE QUEUE IS EMPTY')
   ELSE ITEM:= Q[FRONT]
      IF FRONT:=REAR
      THEN BEGIN
FRONT := 0 ; REAR := 0
END
ELSE FRONT := FRONT+1;
END;
BEGIN  {MAIN PROGRAM}
CLRSCR;
FRONT1 :=0 ; REAR1 :=0;
REPEAT
WRITELN('REPRESENTATION OF QUEUE OPERATIONS');
WRITELN('_________________________________   ');
WRITELN('1- INSERTION OPERATIONS             ');
WRITELN('2- DELETION OPERATIONS                          ');
WRITELN('3- DISPLAY THE CONTENT OF THE QUEUE   ');
WRITELN('4- EXIT                                                              ');
WRITELN;WRITELN;
WRITELN('SELECT YOUR CHOICE');
READLN(CHOICE);
CASE CHOICE OF
1:BEGIN
WRITELN('HWOMAY ELEMENTS YOU LIKE TO ENTER ?');
READLN(M);
FOR I:= TO M DO
BEGIN
WRITELN('ENTER THE NEW ELEMENTY :');
READLN(ITEM1);
ADDQ (QUEUE, FRONT1, REAR1, ITEM1)
END
END;
2:BEGIN
WRITELN('HWOMAY ELEMENTS YOU LIKE TO DELETE ?');
READLN(L);
FOR I:= TO L DO DELETEQ(QUEUE, FRONT1, REAR1, ITEM1)
END;
3:BEGIN
WRITELN('THE CONTENT OF THE QUEUE IS :')
FOR I := REAR1 DOWNTO FRONT1 DO WRITE(QUEUE[1]:3);
WRITELN;
WRITELN ('REAR=',REAR1,'    FRONT=',FRONT1)
END;
4:END;
UNTIL CHOICE=4
END;

Program for Circular Queue representation and it’s operation

PROGRAM CQQ
USES CRT;
CONST SIZE=10;
TYPE ELEMENT=INTEGER; {OR ANY OTHER TYPE}
  QT=ARRAY [1..SIZE] OF ELEMENT;
VAR QUEUE:QT;
  ITEM:ELEMENT;
  FRONT1,REAR1,CHOICE,I,L,M:INTEGER;
PROCEDURE ADDCQ (Var CQ: Qt; Var FRONT,REAR:INTEGER; Var ITEM:ELEMENT);
BEGIN
  IF  REAR = SIZE
  THEN REAR := 1;
  ELSE REAR :=REAR + 1
  IF REAR = FRONT
  THEN WRITELN ('ERROR...THE CQ IS FULL')
  ELSE CQ[REAR]:=ITEM
END

PROCEDURE DELETCQ (Var CQ: Qt; Var FRONT,REAR:INTEGER; Var ITEM:ELEMENT);
BEGIN
  IF FRONT:=REAR
  THEN WRITELN ('ERROR...THE CQ IS EMPTY')
  ELSE BEGIN
    IF FRONT = SIZE
    THEN FRONT :=1
    ELSE FRONT := FRONT+1;
    ITEM := CQ [FRONT];
    CQ [FRONT] :=0
  END;
END

BEGIN  {MAIN PROGRAM}
  CLRSCR; FOR I :=1 TO SIZE DO CQUEUE[1] :=0;
  FRONT1 :=1 ; REAR1 :=1;
  REPEAT
    WRITELN('REPRESENTATION OF CQ OPERATIONS');
    WRITELN('1- INSERTION OPERATIONS');
    WRITELN('ERROR...THE CQ IS FULL');
    WRITELN('ERROR...THE CQ IS EMPTY');
    WRITELN('1- INSERTION OPERATIONS');
    WRITELN('ERROR...THE CQ IS FULL');
    WRITELN('ERROR...THE CQ IS EMPTY');
END;
WRITELN('2- DELETION OPERATIONS');
WRITELN('3- DISPLAY THE CONTENT OF THE CQ');
WRITELN('4- EXIT');
WRITELN;WRITELN;
WRITELN('SELECT YOUR CHOICE');
READLN(CHOICE);
CASE CHOICE OF
1:BEGIN
WRITELN('HOW MANY ELEMENTS YOU LIKE TO ENTER?');
READLN(M);
FOR I := TO M DO
BEGIN
WRITELN('ENTER THE NEW ELEMENT:');
READLN(ITEM1);
ADDCQ(CQUEUE, FRONT1, REAR1, ITEM1)
END
END;
2:BEGIN
WRITELN('HOW MANY ELEMENTS YOU LIKE TO DELETE?');
READLN(L);
FOR I := TO L DO
DEDELETECQ(CQUEUE, FRONT1, REAR1, ITEM1)
END;
3:BEGIN
WRITELN('THE CONTENT OF THE CQ IS:');
FOR I := 1 TO SIZE DO WRITE(QUEUE[1]:3);
WRITELN;
WRITELN('FRONT=',FRONT1,'  REAR=',REAR1)
END;
4:END;
UNTIL CHOICE=4
END;
Linked Structures

4-1) Storage Allocation

4-1-1) Sequential Allocation of Storage

The simple method to store linear list by use sequential store in the computer memory, it by store in the sequential locations, we can know any element’s location if we know that location of the first element is the base address, and the locations of the next element will be depended on it.

The location of K element is the next of element (k-1), and so on the remain element.

Advantages:-

1- It’s easier in representation and applications.
2- It’s more economic because it uses less storage location.
3- It’s more efficient in random access.
4- It’s very suitable in dealing with the stack.

Disadvantage

1- Difficult in executing the operations of add and delete.
2- It’s needed to old defining and determine number of require storage elements.

4-2-1) Dynamic Allocation of Storage

The another method to store are used link or pointer with each element, is contain the address of the next element, therefore are not need to store the data in sequential location, it can be store any data element in any location, so each element or node are consist of two part:-

• First part: contain data.
• Second part: field contain the address of the next element (link).

X:

| Data (x) | Link (x) |

The element x consist of two parts are: Data (x) & link (x).
Advantages:-

- Easy to execute the operation of add and delete to any element, it’s not need more from modify the field of pointer, that’s give the address of the next element.

Disadvantage

- It’s need to large storage location to represent the field of pointer and the basic data.

4-2) Pointers

To understand represent and run data structures by using dynamic storage must explain how can defined pointers and the method of using it programming.

When dealing with array we are used index, to access to the address of some elements, means in index are a variable that the program used it to access to the require element, and the compiler are used it as address for special address in the memory with same name (index name).

This pointer (index) is a relative pointer, its guide to the address of element according to base address, store array in the memory.

There are some of case must be construct data structures with different storage capacity through program execution stage, and this is a linked data strictures, where every data element are contain additional field (part), used as pointer refer to the address of next element, means this structure are (growth) and add new elements dynamically through program execution stage on needing new locations (address).

Pointers are a data type, and the values of it are representing as the address of element (node) location in the memory.

```
A     B     C
P ---- Data  Link ---- Data  Link ---- Data  Link
      |                  |                  |
      |                  |                  | nil
```
Each element is consisting of two parts and defines as record:

```
Type     ptr = ^ Node;
Node = record
  Data: integer; {or any other type}
  Link: ptr;
End;
```

This define mean the record name is node, and consist of two parts:

First part: Data and type is integer, and can define as another type.

Second part: Link and type (ptr), is the same type of are shown in the start, because it is a pointer to element (node).

We add to the above definition to the type of pointer’s data that use in the main program as

```
VAR p, q, r: ptr;
```

Ex)

To define linked structure, their elements are consisting of two parts they are (student name – age).

Sol) we need to use type in Pascal language to define structure as:

```
Type     ptr = ^ Student;
Student = record;
  Name: string;
  Age: integer;
  Link: ptr;
End;
Var      First: ptr;
```

Note that the last element’s pointer value (nil) mean there are nothing after it.
**A- Procedure (new)**

If we have defined pointer like (Var p: ptr) then use procedure as form (new (p)) to mean impounding memory address, and pointer refer to it.

The value of pointers in Pascal language allow to use expressions assignment (:=), compare (=), and it can use as parameters in the function or procedure.

Ex) Var p, q: ptr

This definition contains pointers p, q, it means every one content’s address are numerical value are represent memory address, let’s assume they are refer to their addresses:

Where the address P contain value 100, that is the address in memory, content’s are(abc)

the address P contain value 500, that is the address in memory, content’s are (xyz)

When we use the expression as:

P: =q

Meaning the value (p) will take the value (q), and become p: =500, q: =500. The Pointers p, q are refers to the same address (500), as shown:
but when we use \( p^\wedge = q^\wedge \)

Where \( p^\wedge \) mean content of address that the pointer \( p \) are refer (mean contents abc) but \( q^\wedge \) mean content of address that the pointer \( q \) are refer (mean contents xyz).

So the above expression mean copy content of address \( q^\wedge \) in to address \( p^\wedge \), to become each addresses contain (xyz) as shown

\[
\begin{align*}
\text{P} & \quad 100 \quad \rightarrow \quad \text{100 p^\wedge} \\
\text{q} & \quad 500 \quad \rightarrow \quad \text{500 q^\wedge}
\end{align*}
\]

B- Procedure Dispose

This procedure to editing position that impounding by using (New), and write as:

\[\text{Dispose (p)}\]

Mean editing memory address that pointer \( p \) are refer to it, mean the program not need to use it, this important attribute for dynamic storage, is allow to create (impound) the position when we need that and delete or editing when finishing, so when execute operation of delete any element from linked structure we use (Dispose) after it direct.

4-3) Linked List

Its set of elements (nodes), each one contains data and link, which refer to the next element (node) in the list.

Element \( X \) are consist of Data, Link

\[
\begin{align*}
X: & \quad \text{Data ( x ) Link ( x )}
\end{align*}
\]

Ex)

\[
\begin{align*}
\text{start} & \quad 300 \quad \rightarrow \quad 500 \quad \rightarrow \quad 100 \quad \rightarrow \quad 700
\end{align*}
\]

\[
\begin{align*}
\text{A} & \quad 500 \quad \rightarrow \quad \text{B} \quad 100 \quad \rightarrow \quad \text{C} \quad 700 \quad \rightarrow \quad \text{D} \quad \text{nil}
\end{align*}
\]
We note that:

- Start: its main pointer that refer to the begin the first element in the list, and the value in this example are 300.
- Contents of pointer’s field to the first element are (500), and it’s referring to the address of the second element. Link (A) = 500
- Contents of pointer’s field to the second element are (100), and it’s referring to the address of the third element. Link (B) = 100
- Contents of pointer’s field to the third element are (700), and it’s referring to the address of the fourth element. Link (C) = 700
- Contents of pointer’s field to the fourth element are (nil), and it’s refer that no element after the fourth element. Link (D) = nil

Segment of Code of Pascal Program

1- Create a Linked List (of one node)

```pascal
New (P);  \hspace{1cm} P
Start: = P;  \hspace{1cm} \text{Start}
Readln (P ^.data) \hspace{1cm} \text{إدخال بيانات العنصر} \hspace{1cm} \text{ولتكن} (1001)
P ^.link: = nil;  \hspace{1cm} P \hspace{1cm} \text{nil}
```

2- Create a Linked List (of two nodes)

```pascal
New (P);  \hspace{1cm} P
Start: = P;  \hspace{1cm} \text{Start}
Readln (P ^.data) \hspace{1cm} \text{إدخال بيانات العنصر الأول} \hspace{1cm} \text{ولتكن} (1001)
New (P2);  \hspace{1cm} P \hspace{1cm} P2
```

Diagram:
- **Start**: Main pointer that refers to the beginning of the first element in the list.
- **Link (A)**: Pointer's field to the first element (value 300).
- **Link (B)**: Pointer's field to the second element (value 500).
- **Link (C)**: Pointer's field to the third element (value 100).
- **Link (D)**: Pointer's field to the fourth element (value 700).
3- Create a Link List (of N nodes)

New (P);
Start: = P;
Write ('How many elements you like to create ');
Readln (N);
For I: = 1 to N Do
    Begin
        Readln (P ^.data);
        If I <> N Then New (P2)
        Else P2:= nil;
        P ^. link: = P2;
        P: = P2;
    End;

The final forms for the list are:
4- **Procedure to add new element to the list after pointer P**

Procedure InsertAfter (Var p: ptr);
Var p2: ptr
Begin
   New (p2);
   Readln (p2 ^. data);
   P2 ^. link: = p ^. link;
   P2 ^. link: = p2;
End;

5- **Procedure to display (print) all elements of linked list**

Procedure displaylist (First: ptr)
Var p: ptr;
Begin
   P: = First;
   While P <> nil Do
      Begin
         Writeln (p ^. data);
         P: = p ^. link;
      End;
   End;
6- Procedure to delete an element with certain value

Procedure DeleteList (Var First: ptr; value: integer)

Var p, q: ptr;
Begin

P: = First;
While (P ^. data <> value) Do
Begin

q: = p;
P: = p ^.link

q ^. link: = p ^.link;
Dispose (P)
End;

End;

7- Delete the first element

P: = start;
Start: = Start ^. link;
Dispose (P);

End;
8- Delete the last element

\[
P := \text{head};
\]
\[
\text{If } \quad P \text{^ link } = \text{nil} \\
\quad \text{Then Begin} \\
\quad \quad \text{Dispose (p); head: = nil} \\
\quad \text{End;} \\
\text{Else Begin} \\
\quad \quad \text{While } P \text{^ link } <> \text{nil Do} \\
\quad \quad \quad \text{Begin} \\
\quad \quad \quad \quad q := p; \\
\quad \quad \quad \quad P := p \text{^ link} \\
\quad \quad \quad \text{End;} \\
\quad \quad \quad q \text{^ link } := \text{nil} ; \\
\quad \quad \quad \text{Dispose (p)} \\
\quad \text{End;}
\]

Ex) Write procedure to invert the order element of linked list? For example

X refer to the element in the list where:

\[
X = (a1 , a2 , a3 , ............... , an ) \quad X = (an , an-1 , ............... , a2 , a1 )
\]

Procedure Invert (Var X : ptr)

Var p , q , r :ptr ;

Begin
\[
P := X ;
q := \text{nil} ;
\text{while } p <> \text{nil Do} \\
\quad \text{Begin} \\
\quad \quad r := q ; \\
\quad \quad q := p ; \\
\quad \quad p := p \text{^ link } ; \\
\quad \quad q \text{^ link } := r ; \\
\quad \text{End;}
\]

\[
X := q
\]
End ;
9- Add one element to the end of linked list

\[ P := \text{Start} ; \]
\[ \text{While } p \uparrow . \text{link} <> \text{nil} \text{ Do} \]
\[ P := p \uparrow . \text{link} ; \]
\[ \text{New}(q) ; \]
\[ \text{Readln}(q \uparrow . \text{data}) ; \]
\[ q \uparrow . \text{link} := \text{nil} ; \]
\[ p \uparrow . \text{link} := q ; \]

10- Add element after the position that order \( n \) in the linked list

\[ \text{Write}ln(\text{'} \text{input the position (n) : '} \text{'} ) ; \]
\[ \text{Readln}(n) ; \]
\[ P := \text{head} ; \]
\[ \text{For } i := 1 \text{ to } n-1 \text{ Do} \]
\[ P := p \uparrow . \text{link} ; \]
\[ \text{New}(q) ; \]
\[ \text{Readln}(q \uparrow . \text{data}) ; \]
\[ q \uparrow . \text{link} := p \uparrow . \text{link} ; \]
\[ p \uparrow . \text{link} := q ; \]

11- Add element before the position \( n \) in the linked list

\[ \text{عندما يكون عدد عناصر القائمة أكبر أو يساوي } n \]
\[ \text{Write}ln(\text{'} \text{input the position (n) : '} \text{'} ) ; \]
\[ \text{Readln}(n) ; \]
\[ P := \text{head} ; \]
\[ \text{For } i := 1 \text{ to } n-1 \text{ Do} \]
\[ P := p \uparrow . \text{link} ; \]
\[ \text{New}(q) ; \]
\[ \text{Readln}(q \uparrow . \text{data}) ; \]
\[ q \uparrow . \text{link} := p \uparrow . \text{link} ; \]
\[ p \uparrow . \text{link} := q ; \]

\{ from BCD TO BMCD \}
Delete element in the position \((n)\) in the linked list

\[
\text{Deleting element in position (n) in the linked list.}
\]

\[
\begin{align*}
\text{Let the problem be:} & \quad \text{Delete element in position (n) in the linked list.} \\
\text{The given solution idea is:} & \quad \text{Copy the last element in the position that needs to be deleted (P).} \\
\text{The operation is:} & \quad \text{Delete the last element in the position that needs to be deleted.}
\end{align*}
\]

\[
\begin{align*}
q & := p^\text{.link} \\
p^\text{.data} & := q^\text{.data} \\
p^\text{.link} & := q^\text{.link} \\
\text{Dispose (q)}
\end{align*}
\]

**Example**

Adding one element to the linked list that order as ascending, and stay the elements list are order after adding

\[
\text{Readln (value)}; \\
P := \text{start}; \\
\text{While } p^\text{.data} < \text{value} \quad \text{Do} \\
\text{Begin} \\
\quad q := p; \\
\quad p := p^\text{.link} \\
\text{End;}
\]

\[
\text{New (T)}; \\
T^\text{.data} := \text{value}; \\
T^\text{.link} := p; \\
q^\text{.link} := T;
\]
exchange the element value in specific position (i) to the liked list (start) with the element value in another position (j) in the same list, where (i < j)

\[ p := \text{Start} ; \]
\[ \text{For } n := 1 \text{ To } i - 1 \text{ Do} \]
\[ p := p^. \text{ link} ; \]
\[ q := p^. \text{ link} ; \]
\[ \text{For } n := i + 1 \text{ To } j - 1 \text{ Do} \]
\[ q := q^. \text{ link} ; \]
\[ X := p^. \text{ data} ; \]
\[ P^. \text{ data} := q^. \text{ data} ; \]
\[ q^. \text{ data} := X ; \]

Write procedure to run merge linked list that main pointer (y) in the end of linked list that main pointer (x).

Procedure Merge (x, y :ptr ; Var z :ptr ) ;
Var p : ptr ;
Begin
If x = nil ;
Then z := y ;
Else
Begin
z := x ;
If y <> nil
Then
Begin
p := x ;
While p^. link <> nil Do
p := p^. link ;
End ;
End ;
Write procedure to split linked list that main pointer (Start) to two linked lists, one of them are (first) contain all of the elements in the odd positions to the original list, and the second list (Second) contain all elements in the even positions to the original list.

Procedure Split (start :ptr ; Var First, Second :ptr ) ;
Var   L : integer ;
      P , N , M ,: ptr ;
Begin
   First = nil ;   Second = nil ;
   P := start ;   L := 0 ;
   While   p <> nil      Do
   Begin
      L := L + 1 ;
      If ( L mod 2 ) <> 0
         Then
         Begin
            If L = 1
               Then
               Begin  First := P ;         N := First   End
               Else
               Begin  N^. link := p ;            N := p             End
               End
         Else   Begin
            If L = 2
               Then
               Begin  Second := P ;           M := Second End
               Else
               Begin  M^. link := p ;         M := p             End
               End
         End
   P := P^. link ;
   End ;
   N^. link := nil ;   M^. Link := nil
End ;
Program of linked list representation and its operation

PROGRAM LLIST
USES CRT;
TYPE ELEMENT=INTEGER; {OR ANY OTHER TYPE}
    PTR =^NODE ;
    NODE = RECORD
        DATA : ELEMENT ;
        LINK : PTR
    END ;
VAR FIRST, F, T : PTR;
    ITEM1, ITEM2 : ELEMENT;
    L, K, CHOICE : INTEGER;
PROCEDURE CREATELL (VAR HEAD: PTR);
VAR P, P2 : PTR ;
    I, N : INTEGER ;
BEGIN
    NEW (P);
    HEAD := P ;
    WRITE ('HOW MANY ELEMENT YOU LIKE TO ENTER ?');
    READLN (N);
    FOR I := 1 TO N DO
        BEGIN
            WRITE ('ENTER THE NEW ELEMENT :');
            READLN (P^.DATA);
            IF I <> N
            THEN NEW (P2)
            ELSE P2 := NIL;
            P^.LINK := P2;
            P :=P2
        END
    END;
PROCEDURE ADDAFTER (P:PTR ; ITEM:ELEMENT);
VAR P2:PTR;
BEGIN
    NEW (P2);
    P2^.DATA := ITEM;
    P2^.LINK := P^.LINK;
    P^.LINK := P2;
    P :=P2
END;
PROCEDURE ADDBEFORE (P:PTR ; ITEM:ELEMENT);
VAR Q:PTR
BEGIN
    NEW (Q);
    Q^.DATA := P^.DATA;
    P^.DATA := ITEM;
    Q^.LINK := P^.LINK;
    P^.LINK := Q
END;

PROCEDURE DELVALUE (HEAD:PTR ; Var VALUE:ELEMENT);
VAR P, Q : PTR;
BEGIN
    P := HEAD;
    WHILE P^.DATA <> VALUE DO
        BEGIN
            Q := P;
            P := P^.LINK
        END;
    Q^.LINK := P^.LINK;
    DISPOSE (P)
END;

PROCEDURE DELINPOS (P:PTR);
VAR Q : PTR;
BEGIN
    Q := P^.LINK;
    P^.DATA := Q^.DATA;
    P^.LINK := Q^.LINK;
    DISPOSE (P)
END;

BEGIN {MAIN PROGRAM}
CLRSCR;
FIRST := NIL;
REPEAT
WRITELN('REPRESENTATION OF LINKED LIST AND ITS OPERATIONS ');
WRITELN('_____________________________ ');
WRITELN('1- CREATION A LINKED LIST ');
WRITELN('2- INSERTION AFTER A CERTAIN POSITION (ELEMENT) ');
WRITELN('3- INSERTION BEFORE A CERTAIN POSITION (ELEMENT) ');
WRITELN('4- DELETION AN ELEMENT OF CERTAIN VALUE ');
WRITELN('5- DELETION AN ELEMENT(S) AT CERTAIN POSITIO ');
END;

WRITELN ('6- DISPLAY THE CONTENT OF THE LINKED LIST');
WRITELN ('7- EXIT');
WRITELN; WRITELN;
WRITELN ('SELECT YOUR CHOICE');
READLN (CHOICE);
CASE CHOICE OF
1: CREATELL (FIRST);
2: BEGIN
    WRITELN ('GIVE THE ELEMENTS WHERE TO INSERT THE NEW ITEM AFTER IT :');
    READLN (ITEM1);
    F: = FIRST;
    WHILE F^.DATA <> ITEM DO F: = F^.LINK;
    WRITE ('HOW MANY ELEMENT YOU LIKE ADD :');
    READLN (K);
    FOR L: =1 TO K DO
    BEGIN
        WRITE ('ENTER THE NEW ELEMENT :');
        READLN (ITEM2);
        ADDAFTER (F, ITEM2);
    END
END;
3: BEGIN
    WRITELN ('GIVE THE ELEMENT WHERE TO INSERT THE NEW ITEM BEFORE IT :');
    READLN (ITEM1); F: = FIRST;
    WHILE F^.DATA <> ITEM1 DO F: = F^.LINK;
    WRITE ('HOW MANY ELEMENT YOU LIKE TO ENTER :');
    READLN (K);
    FOR L: =1 TO K DO
    BEGIN
        WRITE ('ENTER THE NEW ELEMENT :');
        READLN (ITEM2);
        ADDBEFORE (F, ITEM2);
    END
END;
4: BEGIN
    WRITELN ('GIVE THE VALUE OF THE ELEMENT YOU LIKE TO DELETE :');
    READLN (ITEM1);
    DELVALUE (FIRST, ITEM1);
END;
5: BEGIN
WRITELN (‘GIVE THE POSITION (SEQUENCE) OF THE ELEMENT ‘);  
WRITE (‘YOU LIKE TO DELETE ‘);  
READLN (K);  
T: =FIRST;  
FOR L: =1 TO K-1 DO  
BEGIN  
  F: =T;  
  T: =T^.LINK  
END;  
WRITE (‘HOW MANY ELEMENT YOU LIKE TO DELETE ‘);  
READLN (K);  
FOR L: =1 TO K DO  
DELINPOS (T);  
END;  
6: BEGIN  
  WRITELN (‘THE ELEMENT OF THE LINKED LIST ARE ‘);  
  T: =FIRST;  
  WHILE T<> NIL DO  
  BEGIN  
    WRITE (T^.DATA:3);  
    T: =T^.LINK  
  END;  
  WRITELN  
END;  
7: END  
UNTIL CHOICE=7  
END;
Program of linked list representation and the two operations of print its elements as reverse and invert the element order as reverse

PROGRAM INVLLIST
USES CRT;
TYPE ELEMENT=INTEGER; {OR ANY OTHER TYPE}
  PTR =^NODE;
  NODE = RECORD
    DATA: ELEMENT;
    LINK: PTR
  END;
VAR    FIRST, F, T, X: PTR;
    ITEM1, ITEM2: ELEMENT;
    L, K, CHOICE: INTEGER;
PROCEDURE CREATELL (VAR HEAD: PTR);
VAR    P, P2: PTR;
    I, N: INTEGER;
BEGIN
  NEW (P);
  HEAD: = P;
  WRITE ('HOWMANY ELEMENT YOU LIKE TO ENTER?');
  READLN (N);
  FOR  I: = 1 TO N  DO
  BEGIN
    WRITE ('ENTER THE NEW ELEMENT :');
    READLN (P^.DATA);
    IF I <> N
    THEN NEW (P2)
    ELSE P2:= NIL;
    P^.LINK:= P2;
    P: =P2
  END;
END;
PROCEDURE ADDAFTER (P: PTR; ITEM: ELEMENT);
VAR    P2: PTR;
BEGIN
  NEW (P2);
  P2^.DATA:= ITEM;
  P2^.LINK:= P^.LINK;
  P^.LINK:= P2
END;
END;
PROCEDURE ADDBEFORE (P: PTR; ITEM: ELEMENT);
VAR Q: PTR
BEGIN
    NEW (Q);
    Q^.DATA:= P^.DATA;
    P^.DATA:= ITEM
    Q^.LINK:= P^.LINK;
    P^.LINK:= Q
END;
PROCEDURE DELVALUE (HEAD: PTR; Var VALUE: ELEMENT);
VAR P, Q: PTR;
BEGIN
    P: = HEAD;
    WHILE P^.DATA <> VALUE DO
    BEGIN
        Q: =P;
        P: = P^.LINK
    END;
    Q^.LINK:= P^.LINK;
    DISPOSE (P)
END;
PROCEDURE DELINPOS (P: PTR);
VAR Q: PTR;
BEGIN
    Q := P^.LINK;
    P^.DATA :=Q^.DATA;
    P^.LINK :=Q^.LINK;
    DISPOSE (P)
END;
PROCEDURE PRINTREV (P:PTR);
BEGIN
    IF P<>NIL
    THEN BEGIN
        PRINTREV(P^.LINK);
        WRITE(P^.DATA:3)
        END
END;
PROCEDURE INVERT (VAR X:PTR);
VAR P, Q, R:PTR
BEGIN
  P:=X;
  Q:=NIL;
  WHILE P<>NIL DO
  BEGIN
    R:=Q;
    Q:=P;
    P:=P^.LINK;
    Q^.LINK:=R
  END;
  X:=Q
END;
BEGIN {MAIN PROGRAM}
CLRSCR;
FIRST := NIL;
REPEAT
  WRITELN('REPRESENTATION OF LINKED LIST AND ITS OPERATIONS ');
  WRITELN('__________________________ ');
  WRITELN('1- CREATION A LINKED LIST ');
  WRITELN('2- DISPLAY THE ELEMENTS OF THE LINKED LIST ');
  WRITELN('3- PRINT THE ELEMENT IN REVERSE ORDER ');
  WRITELN('4- REVERSE THE ORDER OF THE LIST ELEMENTS ');
  WRITELN('7- EXIT ');
  WRITELN;
  WRITELN;WRITELN;
  WRITELN('SELECT YOUR CHOICE');
  READLN(CHOICE);
  CASE CHOICE OF
  1:CREATELL (FIRST);
  2:BEGIN
    WRITELN('THE ELEMENTS OF THE LINKED LIST ARE :');
    T := FIRST;
    WHILE T<> NIL DO
    BEGIN
      WRITE(T^.DATA:3);
      T:=T^.LINK
    END;
    WRITELN
  END;
END;
3:BEGIN
    WRITELN('THIS IS THE ELEMENTS OF THE LIST ARE PRINTED IN REVERSE ORDER');
    PRINTREV(FIRST);
    WRITELN;
END;
4:BEGIN
    INVERT (FIRST);
    WRITELN('THE ELEMENTS OF THE LIST ARE REVERSED');
END;
5:END
UNTIL CHOICE=5
END;
4-4 ) Linked Stack

Its can benefiting from proprieties of dynamic storage to present stack, as special case in linked list, where the operations of adding and deleting from one end (opened end).

The principle of adding and deleting are same in the stack but the different in the way of presentation in the memory.

![Diagram of a linked stack]

Procedure add one element to linked stack
Procedure PUSHLS (Var start: ptr);
Var p: ptr;
Begin
  New (P);
  Readln (p^.data);
  If start = nil
  Then p^.link:= nil;
  Else p^.link:= start;
  Start := p
End;
Procedure delete one element from linked stack
Procedure POPLS (Var start: ptr; Var value: element);
Var q: ptr;
Begin
If start = nil
Then writeln (‘ Error.. Linked Stack is empty’)
Else
Begin
q := Start;
Value := q^.data;
Start := q^.link;
Dispose (q)
End
End;
عندما يكون المكدس خال وعملية الحذف غير ممكنة، فإن المكدس خال وعملية الحذف غير ممكنة.  
- استخدام المؤشر ليشير إلى بداية المكدس (أول عنصر في المكدس).  
- حذف عنصر البداية في المكدس 
  . value وخزنًا وقتياً مع المتغير q^.data  
  . q^.link ليشير إلى موقع العنصر التالي المحدد بالحقل start 
  - تحديث قيمة المؤشر ليشير إلى موقع العنصر التالي المحدد بالحقل start 
  - تحرير الموقع الذي كان يشغل العنصر المحدوف والذي يشير اليه المؤشر q. Dispose(q) باستخدام q^.data q^.link

المكدس قبل الحذف
المكدس بعد الحذف
Program for Linked Stack representation and its operation

PROGRAM LSTACK
USES CRT;
TYPE ELEMENT=INTEGER; \{OR ANY OTHER TYPE\}
  PTR=^NODE;
NODE=RECORD
  DATA: ELEMENT;
  LINK: PTR
END;
VAR FIRST, T: PTR;
VALUE: ELEMENT
M, N, I, CHOICE: INTEGER;
PROCEDURE PUSHLS (Var START: PTR);
VAR P: PTR;
BEGIN
  NEW (P);
  WRITELN ('ENTER THE NEW ELEMENT :');
  READLN (P^.DATA);
  IF START=NIL
  THEN P^.LINK:= NIL
  ELSE P^.LINK:= START;
  START:= P
END;
PROCEDURE POPLS (Var START: PTR; VAR ITEM: ELEMENT);
VAR Q: PTR
BEGIN
  IF START= NIL
  THEN WRITELN ('ERROR...THE LIKED STACK IS EMPTY')
  ELSE BEGIN
    Q:= START;
    ITEM:= Q^.DATA;
    START:= Q^.LINK;
    DISPOSE (Q)
  END
END;
BEGIN \{MAIN PROGRAM\}
CLRSCR;
FIRST:= NIL;
REPEAT
WRITELN ('REPRESENTATION OF THE LINKED STACK OPERATIONS');
WRITELN ('___________________________________________ ');
WRITELN ('1- PUSH A NEW ELEMENT(S)      ');
WRITELN ('2- POP AN ELEMENT(S)           ');
WRITELN ('3- DISPLAY THE CONTENT OF THE LINKED STACK   ');
WRITELN ('4- EXIT                                                                          ');
WRITELN; WRITELN;
WRITELN ('SELECT YOUR CHOICE');
READLN (CHOICE);
CASE CHOICE OF
1: BEGIN
WRITELN ('HWOMAY ELEMENTS YOU LIKE TO ENTER?');
READLN (M);
FOR I: = TO M DO
   PUSHLS (FIRST)
END;
2: BEGIN
WRITELN ('HWOMAY ELEMENTS YOU LIKE TO DELETE?');
READLN (L);
FOR I: = TO L DO
   POPLS (FIRST, VALUE)
END;
3: BEGIN
IF FIRST= NIL
THEN WRITELN ('***ERROR...THE LIKED STACK IS EMPTY***')
ELSE BEGIN
   T: = FIRST;
   WRITELN ('THE CONTENT OF THE LINKED STACK IS :')
   WHILE T <>NIL DO
      BEGIN
         WRITELN (T^.DATA:3);
         T: =T^.LINK
      END
   END
4: END;
UNTIL CHOICE=4
END;
4-5) **Linked Queue**

It can represent queue in dynamic storage like way for represent stack, but with two pointers front and rear as shown:

```
  Rear               Front
  ┌──┐   ┌──┐   ┌──┐
  │D│   │C│   │B│ A
  │   └──┘   └──┘
  └──┐   │   │
    │   │   │
    │   └──┘
    │       nil
```

- Pointer Front refers to the first element in the queue.
- Pointer Rear refers to the last element in the queue.
- Each element from queue four elements (A, B, C & D) has field include the value of the pointer to the next element.
- Pointer of the last element have value (nil), there is no element after it.

**Procedure add one element to linked Queue**

```plaintext
Procedure AddQ;
Begin
  New (P);
  Readln (P^.data);
  P^.link := nil;
  If Rear = nil
    Then Front := P;
  Else Rear^.link := P;
  Rear := P
End;
```

The diagram illustrates the process of adding an element to the queue. The steps include creating a new element, reading its data, setting its link to nil, and updating the pointers to the front and rear.
Procedure delete one element from linked Queue

Procedure DeleteLQ;

Begin

    P: = Front;
    If P = nil
    Then writeln(’Error... Linked Queue is empty’)
    Else

        Begin

            q: = P^.link;
            Item: = P^.data;
            Dispose (p);
            Front: = q;
            If Front = nil
            Then Rear: = nil;

        End;

End;

- استخدام مؤشر وقتی P ليشير الى أول عنصر في الطابور حيث يشير المؤشر Front وعندما تكون قيمة Front null فإن هذا يعني أن الطابور خالي من العناصر ولا يمكن تنفيذ عملية الحذف.

- في مقدمة عبارة Else نستخدم مؤشر ثان هو q ليشير الى العنصر الثاني في الطابور الذي نستطيع حذفه.

- في الخطة الرابعة في عبارة Else هي لتحديث قيمة المؤشر Front ليشير الى موقع العنصر الثاني حيث he يشير q.

- الخطوات الثلاث الأخرى هي لمعالجة حالة حذف اخر عنصر في الطابور مما يتطلب جعل قيمة كل من المؤشرين Front, Rear null.
Ex) Write procedure to copy all elements of sequential stack to the linked queue are empty from elements, where top element in stack become first element in queue?

Sol)

Type St= array [1... size] of element;
Procedure Copy1 (Stack: St; Top: integer; Var F, R: ptr);
Var T: integer; Item: element;
    P: ptr
Begin
    T: = top; Front: = nil; Rear: = nil;
    While T<>0  Do
        Begin
            Item: = Stack [T];
            T: = T-1;
            New (p);
            P^.data:= Item;
            P^.link:= nil;
            If  Rear= nil
                Then Front: = P;
            Else     Rear^.link:= P;
                Rear: = P
        End;
End;
Ex) Write procedure to copy all elements of linked queue to the sequential stack are empty from elements, where first element in queue become top element in stack?
Sol)

Type  St= array [1... size] of element;
Procedure Copy2 (Front, Rear: ptr; Var Top: integer; Var Stack: St);
Var   S: St;   T: integer;   F1: ptr;
P: ptr
Begin
    T: = 0; Top: = 0;
    F1:= Front;
    While F1<>nil Do
        Begin
            P:= F1;
            T:= T+ 1;
            S [T]: = P^.data;
            F1:= F1^.link;
        End;
    While T<>0 Do
        Begin
            Top:= Top + 1;
            Stack [Top]: = S [T];
            T:= T-1;
        End;
End;
Program for Linked Queue representation and its operation

PROGRAM LQUEUE
USES CRT;
TYPE ELEMENT=INTEGER; {OR ANY OTHER TYPE}
  PTR= ^NODE;
  NODE= RECORD;
  DATA: ELEMENT;
  LINK: PTR;
END;
VAR FRONT1, REAR1, T: PTR;
VALUE: ELEMENT;
M, N, I, CHOICE: INTEGER;
PROCEDURE ADDLQ (Var FRONT, REAR: PTR);
VAR P: PTR;
BEGIN
  NEW (P);
  WRITELN ('ENTER THE NEW ELEMENT :');
  READLN (P^.DATA);
  P^.LINK: = NIL;
  IF REAR = NIL
  THEN FRONT: = P
  ELSE REAR^.LINK: = P;
  REAR: =P
END;
PROCEDURE DELETELQ (Var FRONT, REAR: PTR; Var ITEM: ELEMENT);
VAR P, Q: PTR
BEGIN
  P: = FRONT
  IF P = NIL
  THEN WRITELN ('ERROR...THE LINKED QUEUE IS EMPTY')
  ELSE BEGIN
    Q: = P^.LINK;
    ITEM: = P^.DATA;
    DISPOSE (P);
    FRONT: = Q;
    IF FRONT = NIL
    THEN REAR: = NIL
    END;
END;
BEGIN    {MAIN PROGRAM}
CLRSCR;
FRONT1:= NIL; REAR1:= NIL;
REPEAT
  WRITELN ('REPRESENTATION OF LINKED QUEUE OPERATIONS');
  WRITELN ('-----------------------------------------------');
  WRITELN ('1- ADD A NEW ELEMENT(S)                     ');
  WRITELN ('2- DELETE AN ELEMENT(S)                      ');
  WRITELN ('3- DISPLAY THE CONTENT OF THE QUEUE          ');
  WRITELN ('4- EXIT                                      ');
  WRITELN; WRITELN;
  WRITELN ('SELECT YOUR CHOICE');
  READLN (CHOICE);
  CASE CHOICE OF
  1: BEGIN
    WRITELN ('HOW MANY ELEMENTS YOU LIKE TO ENTER?');
    READLN (M);
    FOR I: = TO M DO
      ADDLQ (FRONT1, REAR1);
    END;
  2: BEGIN
    WRITELN ('HOW MANY ELEMENTS YOU LIKE TO DELETE?');
    READLN (N);
    FOR I:= TO L DO
      DELETELQ (FRONT1, REAR1, VALUE);
    END;
  3: BEGIN
    IF FRONT1 = NIL
    THEN WRITELN ('ERROR...THE LINKED QUEUE IS EMPTY')
    ELSE BEGIN
      T: = FRONT1;
      WRITELN ('THE CONTENT OF THE LINKED QUEUE IS :');
      WHILE T<> NIL DO BEGIN
        WRITELN (T^.DATA:3);
      END;
    END;
  END;
  4: END;
UNTIL CHOICE=4
END;
4-6) **Circular Linked List**

In normal linked list are used main pointer refer to first element, and the pointers field of last element his value is nil, because no element after it’s.

In circular linked list the pointer’s field in last element are refer to first element in the list as shown below:

![Circular Linked List Diagram](image)

In this case can use only one pointer refer to the last element and let’s be R, and by this pointer can access to the first element as shown:

![Diagram of Circular Linked List with Pointer R](image)

The value of pointer R is 400 to refer to the last element, and the pointer’s field for the last element is (R^link) we found value (100), and this represent address of first element.

**Procedure add one element to circular linked list**

```pascal
Procedure AddCLL;
Begin
    New (P);
    Readln (P^.data);
    If R= nil
    Then P^.link:= P;
    Else
        Begin
            P^.link:= R^.link;
            R^.link:= P;
        End;
    R:= P
End;
```
Procedure delete one element from circular linked list

Procedure DeleteCLL;
Begin
  If R= nil
  Then writeln ('Error... Circular Linked List is empty')
  Else
    Begin
      F: = R^.link;
      Item: = F^.data;
      If R= F
      Then R: = nil
      Else R^.link: = F^.link;
      Dispose (F)
    End;
  End;
End;

الرسم التوضيحي لحالة حذف العنصر A
**Procedure display contents circular linked list**

Procedure DisplayCLL;
Begin
    P: = R;
    If P = nil
    Then writeln (‘Error... Circular Linked List is empty’) 
    Else
    Repeat
        Writeln ((P^.link) ^.data);
        P: = P^.link
    Until P = R 
End;

4-7) **Double Linked List**

In normal linked list there are difficult in delete the element that the pointer P are refer to it, because of can’t come back to the previous element, to change the pointer’s field on it to refer to the next element, its mean the moving in one direction.

But in double linked list there are each elements contain two pointers one refer to address of the next element, and the other refer to the address of the previous element, its mean each element in the list consist of three parts and define as:

```pascal
Type ptr = ^node;
    node = record
        Data: integer; (or any other type)
        Llink: ptr;
        Rlink: ptr;
    End;

Star
        Link A Rlink
---------
        nil

Link B Rlink
---------
Link C Rlink
---------
        nil
```
Add Element After Address P

new (q)
Readln (q^.data)
q^.Llink: = P
q^.Rlink: = P^.Rlink
(P^.Rlink)^.Llink: q
P^.Rlink: = q

Delete Element in Address P

(P^.Llink) ^. Rlink: = P^. Rlink;
(P^.Rlink) ^. Llink: = P^. Llink;
Dispose (P);
Program for Circular Linked List representation and operations of adding and deleting

PROGRAM CIRCULARLINKEDLIST
USES CRT;
TYPE ELEMENT=INTEGER; {OR ANY OTHER TYPE}
   PTR= ^NODE;
   NODE= RECORD;
   DATA: ELEMENT;
   LINK: PTR;
   END;
VAR START: PTR;
   ITEM1: ELEMENT;
   I, N, M CHOICE: INTEGER;
PROCEDURE ADDCLL (Var R: PTR);
   VAR P: PTR;
   BEGIN
      NEW (P);
      WRITELN ('INPUT THE NEW ELEMENT :');
      READLN (P^.DATA);
      IF R = NIL
         THEN P^.LINK: = P
         ELSE BEGIN  P^.LINK: = P;
         END;
      R: =P
   END;
PROCEDURE DELETECLL (Var R: PTR; Var ITEM: ELEMENT);
   VAR F: PTR
   BEGIN
      IF R = NIL
         THEN WRITELN ('ERROR...THE CIRCULAR LINKED LIST IS EMPTY')
      ELSE BEGIN  F: = R^.LINK;
         ITEM: = F^.DATA;
         IF R=F
            THEN R: =NIL
         ELSE R^LINK: =F^.LINK;
         DISPOSE (F);
      END;
   END;
PROCEDURE DISPLAY (VAR R: PTR)
   VAR P, X: PTR;
BEGIN
  P: = R;
  IF P <> NIL THEN BEGIN
    WRITELN ('THE ELEMENTS OF C.L.L ARE');
    REPEAT
      X: = P^.LINK;
      WRITE (X^.DATA: 4);
      P: = P^.LINK
    UNTIL P = R;
    WRITELN
  END;
END;
BEGIN {MAIN PROGRAM}
  CLRSCR;
  START: = NIL;
  REPEAT
    WRITELN ('REPRESENTATION OF CIRCULAR LINKED LIST OPERATIONS');
    WRITELN ('__________________ ___________________________   ');
    WRITELN ('1- ADD AN ELEMENT(S) TO THE C.L.L                      ');
    WRITELN ('2- DELETE AN ELEMENT(S) FROM THE C.L.L                  ');
    WRITELN ('3- DISPLAY THE CONTENT OF THE C.L.L                      ');
    WRITELN ('4- EXIT                                                   ');
    WRITELN; WRITELN;
    WRITELN ('SELECT YOUR CHOICE');
    READLN (CHOICE);
    CASE CHOICE OF
      1: BEGIN
        WRITELN ('HWOMAY ELEMENTS YOU LIKE TO ADD?');
        READLN (N);
        FOR I: = TO N DO ADDCLL (START);
      END;
      2: BEGIN
        WRITELN ('HWOMAY ELEMENTS YOU LIKE TO DELETE?');
        READLN (M);
        FOR I: = TO M DO DELETELQ (START, ITEM);
      END;
      3: DISPLAY (START);
      4: END;
    UNTIL CHOICE=4
END;
Trees

5-1) Graph

Its set of elements (V) represented by points (heads) called (Vertices), these elements connected by relations (E) represent by lines called (edges) mean the graph are: G = (V, E), and its set of elements and relations as shown:

\[ V = \{1, 2, 3, 4, 5, 6\} \quad \text{elements} \]
\[ E = \{(1, 2), (1, 3), (2, 3) \]
\[ (2, 4), (3, 5), (3, 6)\} \quad \text{relations} \]

The graphs are two types:

1- **Undirected graph**

   The relation between its elements (heads) in this graph is unordered, meaning the direction is unimportant in the relation, like edge (1, 2) its same (2, 1).

2- **Directed graph**

   The relation between its elements (heads) in this graph are ordered by special form, the direction is important to determine this relation, for example edge (1, 2) are deferent from (2, 1), and represent this relation by put arrow in the front line to illustrate the direction as shown below the relation between (2, 3) illustrate by rectum, the direction of relation is (2 \(\rightarrow\) 3), and there are another relation are deferent from its are (3, 2) represent by another rectum and mean the relation are (3 \(\rightarrow\) 2).
Path
Its set of lines that connected between any two points in graph, in figure A the path between 1, 5 are (1, 3), (3, 5).

Path Length
Its number of lines that connected between any two points, in figure A
2, 6 path length = (2, 3), (3, 6)
1, 6 there are two path = (1, 2), (2, 3), (3, 6) and length are 3
= (1, 3), (3, 6) and length are 2

Connected Graph
Its graph contains paths between any two points from graph points.

Unconnected Graph
Its graph some of points are unconnected in path.

5-1-1) Graph Representation
Choosing method to represent graph depend on type of required application and functions, and illustrate two method of there:

1- Adjacency Matrix
Represent the graph as esquire matrix with equal degree for number of vertices, and if number of vertices 3 then the matrix are (3 x 3), and when No. of vertices 7 then the matrix must be (7 x 7).
In this graph (Directed Graph) are consist of 4 points (vertices) (S1, S2, S3, S4), and 5 edges, and can represent by esquire matrix had four degree and elements as (S i, j), where (i) represent start point, and (j) represent end point, and when found edge between two points represent with value 1 and when no edge represent by 0.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S₁₁</td>
<td>S₁₂</td>
<td>S₁₃</td>
<td>S₁₄</td>
</tr>
<tr>
<td>2</td>
<td>S₂₁</td>
<td>S₂₂</td>
<td>S₂₃</td>
<td>S₂₄</td>
</tr>
<tr>
<td>3</td>
<td>S₃₁</td>
<td>S₃₂</td>
<td>S₃₃</td>
<td>S₃₄</td>
</tr>
<tr>
<td>4</td>
<td>S₄₁</td>
<td>S₄₂</td>
<td>S₄₃</td>
<td>S₄₄</td>
</tr>
</tbody>
</table>

And when represent the values for each path become:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

This matrix is representing the graph where see:
Edge from S1----> S2.
Edge from S1----> S4.
Edge from S3----> S2.
Edge from S3----> S4.
Edge from S4----> S3.
No edge from S1 to S3, no edge from S2 to any point.
No edge from S3 to S1.
No edge from S4 to S1 & S2.
Note that the matrix is represented by directed graph are:
- Unsymmetrical.
- The sum values in each row represent no. of out lines (out degree) from any point, where in row three(i=3) the sum of values are (2), because of the point (S3) there are two lines are out (S2, S4).
- The sum values in each column represent no. of inter lines (in degree) to each point, where column four (j=4) the sum of values are (2), because of the point (S4) are enter to it two lines they (S1, S3).

But the (Undirected Graph)

This graph are consist of 5 points (vertices) (T1, T2, T3, T4, T5) and 6 edges, are represented by esquire matrix had five degree, and when found edge between two points represent with value 1 with out depend on the direction and when no edge represent by 0.

T:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
The line from T1 to T2 are represented by T (1, 2), and value is 1, this line is the same line from T2 to T1 and represented by T (2, 1) and value is 1, and so for each lines.

In this matrix note that:

- Symmetric in axis (top triangle are symmetric down triangle), so it can reduce storage area by use one of two triangle.

- Sum of values in each row represent no. of out lines (out degree) from each point, in row four (i= 4) the sum values are (1), because of point four (T4) are out one line to point (T5).

2- **Adjacency Lists**

Its used linked list in graph representation, where for each node from graph nodes are represent by linked list contain names of nodes which connect to the nodes of linked list (i) (vertices) adjacent to node (i), where each node consist of two parts (like linked structures) part contain (index of vertex), and the second part is pointer (link) refer to position of next node.

Take undirected graph:

![Graph Diagram](image)

The presentation of the graph by adjacent linked list, each of their have main pointer refers to the first like vertex1, vertex2 ..., etc, as shown:

```
vertex1  2   3   4   nil
vertex2  1   3   4   nil
vertex3  1   2   4   nil
vertex4  1   2   3   nil
```
These pointers are refer to the start of each list, and store as one dimension array and size of this array are equal to no. of list or vertices (n), and define as:

```pascal
Type ptr = ^Node;
Node = record
  Vertex: integer;
  Link: ptr;
End;
Var head: array [1...n] of ptr;
```

Observe that the undirected graph that consist of (n) of vertices and (e) of edges and its representation need (2 x e) from nodes, and matrix had size (n) to store main pointers that refers to start each list, and for above example found we need to (12) nodes because of (e= 6), and matrix have size (4) equal to no. of vertices.

For directed graph

![Directed Graph Diagram]

Note that the sum of nodes in lists is equal to no. of edges in above graph

- vertex1: 4 nil
- vertex2: 1 1
- vertex3: 4 nil
- vertex4: 1 nil
5-2) **Tree Structure**

There are data structures same to directed graph, whichever non linear data structure, like the complexity in roads pursuances in town plan, with depend principle of graphs method are assist to represent this data structures, and deals with it from programming and storing by using computer.

**Tree**

Its type of directed graph (Digraph), but with out formalizing cycle (no cycle), so there are one line are connect between two points (nodes).

![Tree structure](image1)

![Tree structure](image2)

Tree structure  

not Tree structure because formalizing cycle

It can define tree structure as set of nodes are described as:

- There is one node called Root, and its no node before it (it has no father).

- Remaining nodes are divided to separated groups, each of there is having tree structure called sub tree.

Let take this tree structure

![Tree structure](image3)
Now there are some of definition to illustrate particulars that related with tree structure and the way of using dependent on the above form:

**Root**
It’s the node where no node before it, it has no father like node A.

**Branched Node**
Its node has branch like A, C, D, F.

**Terminal Node (Leaf)**
Its node that has no branch like B, E, G, H, I, (it has no children).

**Node Level**
Its number of paths that’s far from the root.

Level of root = 0
Level node E = 2
Level node H = 3

**Node Degree**
Its set of directed exited paths from its (number of children on it)

Node degree A =2  Node degree C =3
Node degree F =1  Node degree H =0

**Tree Degree**
Its highest degree from nodes degrees that ingredient to tree, for above figure the tree degree are 3.

**Tree Height**
Its largest level for any node in tree, meaning it’s the longer path in tree.

**Subtree**
Its can divided the tree to parts, and these parts are subtrees, and for each one had root and each one had leaf.
And this tree can divided to subtree

```
A           B
\   \     \   
D     E     F
```

And can continue dividing, the subtree B can divide to two subtree:

```
B
  D
  E
```

Subtree C can divide to

```
C
  E
  F
```

And so subtree E can divided to four subtrees for each one are consist of one node:

```
E
  G
  H
  I
```
ملاحظات:
- لا يوجد توصيات بين العقد في مستوى واحد.
- لا يوجد توصيات بين أوراق الشجرة.
- لا توجد دوارات في الشجرة.
- كل عقد بمثابة أب بالنسبة للعقد المتفرعة منها مباشرة وكل من تلك العقد بمثابة ابن بالنسبة للعقدة الأب.

- cousin – عم – brother – اب – son


كل عقد لها ابن واحد.

Level 0
Level 1
Level 2
Level 3

From above fig. we note:
- The root of tree is A.
- Trees leaf are D, H, I, J, K.
- Branched nodes are A, B, C, E, F, and G.
- Node B is father for D, E, and F.
- Node C is father for only G.
- Node E is father for H.
- Node F is father for two nodes I, J.
- Node G is father for node K.
- The relation between D, E, F are brother.
- The relation between I, J are brother.
- Node E is had relation cousin with node J.
- Node B is had relation grand father with node I.
- Tree height is 3 because longer path is 3.
- Tree degree is 3 because largest degree for node B is 3.
5-3) **Binary Tree**

Its tree which each node not have more than two nodes (at most two sons), so the degree of any node not more than 2 (may be 0 or 1 or 2), and this tree is represent important data structure and it’s had many application in computer science.

Large number of nodes in level

In binary tree the large number of nodes in level (L) its $2^L$, for the fig. note in $L=2$ the large no. of nodes are $2^2=4$ and they are D, E, F, and G.

But in the next level ($L=3$) the possible large no. of nodes are $2^3=8$, mean that the large no. are not more than 8, and it can be less than that no. where in the tree no. of nodes are two and they are (H, I).
Large number of nodes in binary tree

In binary that height (h) the maximum no. of nodes are \(2^{h+1} - 1\), and may be less than from this no.

Ex) for the fig. A, that height h=3

Max no. of nodes = \(2^{3+1} - 1 = 2^4 - 1 = 16 - 1 = 15\)

![Binary Tree Diagram](A)

But in fig. B, its binary tree that height h= 3, but the current nodes are 7

![Binary Tree Diagram](B)

ملاحظات
- عدد أوراق الشجرة الثنائية = (عدد العقد التي درجتها ٢) + ١
- تكون الأشجار الثنائية مكافئة مع بعضها (equivalent) إذا كان لها نفس التركيب أي نفس الهيئة من حيث عدد ومواقد العقد وشكل التفرعات وتطابق البيانات.
5-4) Another Types of Tree

**Balanced Tree**
Its tree that all leaves position in the same level

Unbalanced Tree  Balanced Tree  Balanced Binary Tree

**Balanced Binary Tree**
Its binary tree that any node in it’s has two branches.

**Full Binary Tree**
Its binary tree that all leaves in one level and any branch node have two branches (like triangle \( \triangle \)).

Full Binary Tree  Not full Binary Tree
**Complete Binary Tree**
Its binary tree that may be full or not full to the level before last level, and the leaves tree position in the left

![Complete Binary Tree](image)

**AVL- Tree**
Its binary tree not empty, and different in the height of subtree for any node not more than 1, the height of left subtree \((T_L)\) for any node not more for the height of right subtree \((T_R)\) to 1

\[ | h_L - h_R | = < 1 \]

**Binary Search Tree**
Its binary tree that the value of left element (son) for any node is less than the value of the element node (father), and the value of right element (son) is more than the value of the element node (father).

![Binary Search Tree](image)
m-way Search Tree
Its balanced tree, and all nodes degree in (m) or low.

B- Tree
Its search tree with degree (m), and may be empty or has height =>1.

5-5) Tree Traversing
The operation of traversing mean passing or visit all nodes in tree only one time and not allow visit again.
Because tree structure are non-linear data structure, and operation of search for special node in this structure or add element or deleting are different from the way of dealing with another structures, and selection one of these methods (way) are independent on tree representation in the memory, and the following are the important way are use for this purpose:

1- Level by Level Traversing:-
A) Top-Down Traversing:

A, B, C, D, E, F, G, H: هي top-down: تكون نتيجة استعراض عقدها بطريقة

```
1
A
  2   3
B   C
  3
D   E
  3
F   G   H
```
B) Bottom – Up Traversing

1- البدء بالورقة في أقصى اليسار بادئي المستوى.
2- التحرك نحو العقدة في اليمين وبنفس المستوى لحين الانتهاء من زيارة جميع عقد ذلك المستوى.
3- الانتقال إلى المستوى الأعلى وزيارة العقد في أيضا من اليسار إلى اليمين وهكذا تستمر العملية لحين الوصول إلى جذر الشجرة.

F, G, H, D, E, B, C, A: هي bottom-up

ملاحظة

لاحظ أن التعامل مع الشجرة وإجازاتها يوضح التشكيل الأساسي والمتكرر فيها هو تكونها من الجذر N

وقد تحتوي ورقة أو أكثر أو بدون أوراق لذا فإن التعامل مع الشجرة يمك ان يبدأ بالجذر ولنفرضه N أو بالورقة في أقصى اليسار L أو بالورقة في أقصى اليمين R.

RLN, RNL, LKN, LNR, NRL, NLR

ولهذا فإن احتمالات الاستعراض هي ستة:

- ولذا لم نأخذ فقط الحالات التي تمثل الاستعراض من اليسار إلى اليمين وهي ثلاثة
- أب البدء بالجذر N ثم التحرك نحو اليسار L ثم اليمين R ونكون الجذر يذكر هنا مسبقاً تسمى NLR
- هذه الطريقة بالترتيب السابق preorder
- أي البدء بالجذر N والانتهاء بالجذر R ينطلق بال주의 ونسمي هذا LRN
- الطريقة بالترتيب اللاحق postorder
- أو البدء بالجذر N ثم اليمين R ثم الجذر N LNR
- أو البدء بالجذر N ثم اليمين R ثم الجذر N LNR
- هذه الطريقة بالترتيب الوسط inorder

2- Preorder Traversing (NLR)

تلتخص خطوات هذه الخوارزمية بما يلي:
- البدء بعقدة الجذر N.
- استعراض الشجرة الفرعية في أقصى اليسار.
- داخل الشجرة الفرعية يتم الاستعراض من أقصى اليسار (يتمثل أكبر الابناء) ثم التحرك للليمين.
- (father's brother) (لا يوجد بالنظام) (لا يوجد الابناء)
- في حالة لا يوجد في اليمين (لا يوجد الأخ)
سوف تكون نتيجة الاستعراض هي

1- A, B, E, C, F, G, D, H, I, J, K
   - جميع الآباء يذكرون قبل الأبناء A قبل B, C, D
   - لو مثلنا هذا بالسير حول الشجرة (الخط المنقط) لوجدنا أن العقدة تذكر عند أول مرور بها بدأ من الجذر.

**Polish Notation**

3- **Postorder Traversing (LRN)**
   - تتللخص خطوات هذه الخوارزمية بما يلي:
     - البدء بالعقدة الورقة في أقصى يسار الشجرة ثم الأوراق التي على يمينها (إن وجدت).
     - الانتقال إلى العقدة الأعلى (father) الاب لتلك العقدة.
     - نسج الشجرة الفرعية التالية في اليمين بنفس الطريقة لحين الوصول إلى الجذر.

**Diagram:**

A, B, E, C, F, G, D, H, I, J, K

D, H, E, F, B, I, J, G, C, A

A, B, E, C, F, G, D, H, I, J, K

A, B, E, C, F, G, D, H, I, J, K
لا مث الابناء بعد يذكرون الاباء جميع بعد

- Reverse Polish Notation (RPN)

4- **Inorder Traversing**

هذه الطريقة تستخدم في مسح الأشجار الثنائية فقط وتتلخص خطوات هذه الخوارزمية بما يلي:

- البدء بالعقدة الورقة في أقصى يسار الشجرة.
- الانتقال إلى عقدة الجذر (اب تلك العقدة).
- زيار العقدة التي في اليمين (إن وجدت) وفي حالة عدم وجودها يكون الانتقال إلى الجد.

(Grand father)

لناخذ الشجرة الثنائية الآتية:

```
A
/ \
B   C
/ \
D   E
   /   \
  G   F
    /   \
   H   I
     /   \
   J   K
```

- ان نتيجة الاستعراض هي G, D, H, B, A, E, C, I, F, J.

- عمليات الطريقة تلاحظ ما يلي:
- أو مثلا هذا الاستعراض بالسير حول الشجرة لوجدنا ان العقدة تذكر عند المرور تحتها.
- Reverse Polish Notation

5-6) **Tree representation**

There are many types to tree representation when store in computer, and determine the best depend on:

- Operations that the problem needed to solve.
- The computer using.
- Program language.
5-6-1) General Tree Representation

1- No. of Pointers are equal to the large No. of Branches

For each node in tree different No. of children and by using linked list to represent like this tree then must determine pointer for each child so:
- The node that has one child needs one pointer.
- The node that has two children needs two pointers.
- The node that has three children needs three pointers...etc

This linked list must define no. of pointers equal to the more no. of children for each node in tree, this mean each node has the same no. of pointers until if the no. of branch (son) less than it, in this case will loss large area in memory.

We note the no. of branch from each node are different and the large branch are three for node B, its has three children D, E, F, then the definition of all elements of linked list with no. of pointers equal to 3 for each element:

Type      ptr = ^node;
          Node = record;
          Data = integer; {or any type}
          Link 1: ptr;
          Link 2: ptr;
          Link 3: ptr;
        End;

VAR       Tree: ptr;
2- Two pointers for each node
   In this case for each element from list elements have two pointers:
   - Pointer refers to the large left son (eldest son in the left).
   - Pointer refers to the next brother.

Then the definition for linked list for the same tree is:
Type    ptr = ^node;
    Node = record;
    Data = integer; {or any type}
    Son       : ptr;
    Brother: ptr;
    End;
VAR      Tree: ptr;
3- **Three pointers for each node**

In this case for each element from list elements have three pointers:
- Pointer refers to the eldest son in the left.
- Pointer refers to the next brother.
- Pointer refers to the node’s father.

Then the definition for linked list for the same tree is:
```pascal
Type ptr = ^node;
Node = record;
  Data = integer; {or any type}
  Son : ptr;
  Brother: ptr;
  Father : ptr;
End;
VAR Tree: ptr;
```
5-6-2) Binary Tree Representation

1- Array Representation

In this representation used one dimension array with size equal to large no. of nodes for binary tree, that height \( h \) by using the relation \( 2^{h+1} -1 \), and store the data value for nodes as:

- تخزين عقدة الجذر في الموقع الأول من المصفوفة وليكن \([T]\). 

- تخزين العقد الأخرى بحيث:-
  - عقدة الابن الأيسر للعقدة في الموقع \([I]\) تكون في الموقع \((2 \times I)\).
  - عقدة الابن الايمن للعقدة في الموقع \([I]\) تكون في الموقع \((2 \times I + 1)\).
  - عقدة الاب لابنة عقدة في الموقع \([I]\) تكون في الموقع \((I \div 2)\).
بما ان ارتفاع الشجرة $h=3$ لذا فإن أكبر عدد ممكن من العقد في مثل هذه الشجرة سيكون

$$\begin{align*}
\text{عدد العقدات} &= 2^{h+1} - 1 \\
&= 2^{3+1} - 1 = 2^4 - 1 = 16 - 1 = 15
\end{align*}$$

اذن سعة المصفوفة لتمثيل هذه الشجرة هو 15

ولكن المصفوفة هي 

\[
T[15]
\]

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>-</th>
<th>-</th>
<th>-</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>-</th>
<th>-</th>
<th>-</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- العقدة $C$ هي الابن الأيمن للعقدة $A$ تقع في الموقع $[3]$ لأن $2 \times 1 + 1 = 3$.


$1 \div 2 = 10$ 

$2 \div 2 = 5$
2- Record Representation
In this representation used linked list to represent tree, and by many way:

A) Representation by two pointers

where one pointer points to the left child and the other points to the right child of the parent and are linked to the list's elements. Each element in the list represents a node in the tree.

Type

```
ptr = ^ node;
Node = Record;
Data: char {or any other type}
Lchild: ptr;
Rchild: ptr
End;
```
B) Representation by three pointers

Type ptr = ^ node;
Node = Record;
Data: integer {or any other type}
Lch : ptr;
Rch : ptr;
f : ptr;
End;

A
 B
 C
D E F
 G H

Lch D Rch f
Lch B Rch f
Lch C Rch f
Lch E Rch f
Lch F Rch f
Lch G Rch f
Lch H Rch f
**Procedure of Binary Tree**

Tree structure is from subtree a structure, then the part is like whole, because subtree is tree, and then we can benefiting from recursion in write the procedure and function for tree representation.

**Procedure Preorder**

```pascal
Begin
  If Root <> nil
  Then Begin
    Writeln (Root^.data);
    Preorder (Root^.Llink);
    Preorder (Root^.Rlink);
  End;
End;
```

**Procedure Postorder**

```pascal
Begin
  If Root <> nil
  Then Begin
    Postorder (Root^.Llink);
    Postorder (Root^.Rlink);
    Writeln (Root^.data);
  End;
End;
```

**Procedure Inorder**

```pascal
Begin
  If Root <> nil
  Then Begin
    Inorder (Root^.Llink);
    Writeln (Root^.data);
    Inorder (Root^.Rlink);
  End;
End;
```
Iteration procedure to traversing binary tree nodes by Inorder Traversing
Procedure Non-Recinorder (Root: ptr)
Var Top: integer; P: ptr;
Begin
Clearstack (Top);
P: = Root;
REPEAT
  WHILE P<> nil DO
  Begin
    PUSH (Stack, Top, P);
    P: = P^.left
  End;
  IF Not Emptystack (Top) Then
  Begin
    POP (Stack, Top, P);
    Writeln (P^.data);
    P: = P^.right
  End
UNTIL (P = nil) AND Emptystack (Top)
End;

Recursion procedure to determine No. of leaves in binary tree
Procedure Leaves (R: ptr)
Var count: integer;
Begin
  IF R<> nil Then
  Begin
    IF (R^.left = nil) AND (R^.right = nil) THEN Count: = Count + 1;
    Leaves (R^.left);
    Leaves (R^.right);
  End;
  Writeln ('The number of leaves = ', count)
End;
Recursion procedure to swap the value of node in left branch with the value of node in right branch

Procedure Swapnodes (R: ptr)
Var T: integer;
Begin
  IF R<> nil
    Then Begin
      T: = R^. left;
      R^. left: = R^. right;
      R^. right: = R^. left;
      Swapnodes (R^.left);
      Swapnodes (R^. right);
    End;
End;

5-7) Representation of Arithmetic Expression using Binary Tree
The important applications for binary tree are using in arithmetic operation (+, -, *, /, etc), they are represented by branch node and about the operand they are represented by leaves, with noted the level of tree are represent the priority of execution the arithmetic operation in arithmetic expression.

Ex) Use binary tree to representation the arithmetic expression:
A = B * C + ( 8 + D * E ) / ( F * 2 )

\[
\begin{array}{c}
A \\
+ \\
* \\
/ \\
\end{array}
\]

\[
\begin{array}{c}
A \\
B \\
C \\
8 \\
\text{ABC} \\
\text{DE} \\
\text{F} \\
\text{2} \\
\text{D} \\
\text{E} \\
\end{array}
\]

١- طريقة الترتيب البيني نحصل على التعبير نفسه
infix notation
٢- طريقة الترتيب السابق نحلول على صيغة النotation
prefix
٣- طريقة الترتيب اللاحق نحلول على صيغة للتعبير الحسابي أي
postfix notation
polish notation
وتسمى أيضا صيغة للتعبير الحسابي اي
reverse polish notation
وتسمية أيضا صيغة
Ex) Use binary tree to represent each of arithmetic expression:
A) \( S \wedge (a + b \wedge n) \)

B) \( X := 2 \ast (a - b / c) \)

C) \( a + b \ast (c + d) \)
D) \( a^5 + 8b^3 - 2c - 5 \)

E) \( a + b - [(c + d) * e] \)
5-8) Transformation of a tree into binary tree

لتحويل الشجرة الاعتيادية الى شجرة ثنائية نتبع خطوات الخوارزمية التالية:

1- جذر الشجرة الاعتيادية يصبح هو جذر الشجرة الثنائية.

2- الأبن الأيسر للشجرة الثنائية يكون نفسه الأبن الأيسر من الشجرة الاعتيادية.

3- ان اخوة الأبن الأيسر في الشجرة الاعتيادية يصبحون الأبن اليمين له في الشجرة الثنائية.

4- نعيد نفس الخطوات واعتيار الأبن الأيسر هو الجذر.

Ex)
Ex) Convert this tree in to binary tree?
5-9) **Binary Search Tree**

Its binary ordered type where the value of left son for any node is less than the value of father this node and the value of right son is more than (large) the value of father this node.

In another way:

{The value of any node (father) is more than the value of left branch (left son) and less than from the value of right branch (right son)}. 

```
50
 /    
20    90
 /  
10    80
   /  
  70   88
```

```
P
J
A
F
D

Z
S
M
R

L
K
G
```

```
K
T
Y

M

F
```

```
G

D
```
Ex) Write binary search tree for the elements:
5, 9, 7, 3, 8, 12, 6, 4, 20

1- Take the first element and consider it the root. Take 5

2- Take the second element 9 and consider it the right branch of the root.

3- Take the next element 7 which is greater than 5 and consider it the right branch of 9.

4- Take the next element 3 which is less than 5 and consider it the left branch of 5.

5- Repeat the procedure with the new element and compare it with the elements of the tree from the beginning to the end. Finally, we get the tree.

```
       5
      /  \
     9   7
    / \  / \n   3  6  8 20
```


Ex) Write the binary search tree for the elements B, A, D, C, G, F, E?
Ex) Write the binary search tree for the elements A, B, C, D, E, F, G?
Iteration procedure to find node in binary search tree
Procedure Findnode (Root: ptr; Value: elementtype; Var Found: Boolean)
Var P: ptr;
Begin
  P: = Root;
  Found: = False;
  WHILE (P<> nil) AND (Not Found) DO
    Begin
      IF P^.data = Value
      THEN Found: = True
      ELSE Begin
        IF P^.data > Value
        THEN P: = P^.left
        ELSE P: = P^.right
      End;
    End;
  End;
End;

Recursion procedure to find node in binary search tree
Procedure BTsearch (Root: ptr; Key: elementtype);
Var P: ptr;
Begin
  P: = Root;
  IF P<> nil
  THEN Begin
    IF P^.data = Key
    THEN writeln ('The key is Found')
    ELSE IF P^.data < key
    THEN BTsearch (P^.right, key)
    ELSE BTsearch (P^.left, key)
  End;
End;
Program of binary search tree and its operations

PROGRAM TREE;
USES CRT;
TYPE    PTR = ^NODE;
    NODE = RECORD
        DATA: INTEGER;
        LLINK, RLINK: PTR
    END;
VAR    T, R, P1, X, H: PTR;
    I, F, D, M: INTEGER;
PROCEDURE CREATE (R, P1: PTR)
VAR    P: PTR; X: INTEGER;
BEGIN
    IF F=0
    THEN BEGIN
        NEW (P);
        READ (P^.DATA);
        P^.RLINK: = NIL; P^.LLINK: = NIL;
        F: = 1       ; P1: = P;
        D: = P^.DATA;
        END;
    IF R <> NIL
    THEN
        IF D >= R^.DATA
        THEN BEGIN
            IF R^.RLINK = NIL
            THEN R^.RLINK: = P1
            ELSE CREATE (R^.RLINK, P1)
        END;
        ELSE BEGIN
            IF R^.LLINK = NIL
            THEN R^.LLINK: = P1
            ELSE CREATE (R^.LLINK, P1)
        END;
    END;
PROCEDURE PREORDER (R: PTR);
BEGIN
IF R <> NIL THEN BEGIN
  WRITELN (R^.DATA);
  PREORDER (R^.LLINK);
  PREORDER (R^.RLINK);
END;

PROCEDURE POSTORDER (R: PTR);
BEGIN
  IF R <> NIL THEN BEGIN
    POSTORDER (R^.LLINK);
    POSTORDER (R^.RLINK);
    WRITELN (R^.DATA);
  END;
END;

PROCEDURE INORDER (R: PTR);
BEGIN
  IF R <> NIL THEN BEGIN
    INORDER (R^.LLINK);
    WRITELN (R^.DATA);
    INORDER (R^.RLINK);
  END;
END;

PROCEDURE INSERT (H: PTR);
BEGIN
  IF (H^.RLINK <> NIL) AND (X^.DATA > H^. DATA) THEN BEGIN
    H: = H^.RLINK; INSERT (H)
  END;
  ELSE IF (H^.RLINK = NIL) AND (X^.DATA > H^. DATA) THEN H^. RLINK: = X
  ELSE IF H^. LLINK <> NIL THEN BEGIN
  END;
BEGIN {MAIN PROGRAM}
CLRSCR;
WRITE ('INPUT THE NO. OF NODES : ');
READLN (M);
NEW (R);
WRITELN ('INPUT THE DATA FIELD OF EACH NODE : ');
READLN (R^. DATA);
R^. RLINK: = NIL;  R^. LLINK: = NIL;
T: = R;  P1: = NIL;
FRO I: = 1 TO M-1 DO
BEGIN
F: = 0;  R: = T;
CREATE (R, P1)
END;
R: = T;
WRITELN ('THE OUTPUT OF THE PREORDER TRAVERSING IS : ');
PREORDER (R);
WRITELN ('THE OUTPUT OF THE INORDER TRAVERSING IS : ');
INORDER (R);
WRITELN ('THE OUTPUT OF THE POSTORDER TRAVERSING IS : ');
POSTORDER (R);
WRITELN ('TO INSERT NEW NODE');
WRITELN ('INPUT THE NEW VALUE: ');
H: = T;  NEW (X);
READLN (X^. DATA);
X^. LLINK: = NIL;  X^. RLINK: = NIL;
INSERT (H);
WRITELN ('AFTER INSERTION – THE INORDER TRAVERSING IS : ');
R: = T;  INORDER (R)
END;
Sorting

6-1) Sorting
Its operation of order (sorting) set of data elements according value of field (fields) and called (key) as ascending or descending.

6-2) Purpose of Sorting
1- Increase the efficiency of search algorithm from some element.
2- Simplified files execution.
3- Solve problem like constraints.

6-3) Steps of Sorting
1- Reading field key.
2- Concluding element’s position in new sorting.
3- Transfer data element to new position.

6-4) Selection Sort
حوارزمية هذا الترتيب هي:

1 - إيجاد أصغر عنصر في القائمة واستبداله من موقعه مع العنصر في الموقع الأول في القائمة.
2 - إيجاد أصغر عنصر في المتبقي من القائمة واستبداله من موقعه مع العنصر في الموقع الثاني في القائمة.
3 - تستمر هذه العملية لحين الوصول إلى نهاية القائمة.
Ex) Sort the list as ascending? (8 3 9 7 2 6 4)

Original list 1 2 3 4 5 6 8 2 2 2 2 2 2 2 3 3 3 3 3 3 3 9 9 9 4 4 4 4 7 7 7 7 6 6 6 2 8 8 8 8 7 7 6 6 6 4 4 9 9 9 9

\[ n = 7 \]

\[ \text{no. of passes} = 6(n - 1) \]

\[ \frac{n}{2} \times (n - 1) \] (average No. of comparisons)

\[ (n - 1) \] (average No. of exchanges)

\[ \text{average No. of exchanges} \]

Const n=10; \{or any other value\}
Type Ar= array [1..n] of element;
Procedure SelectionSort (Var Data: Ar; N: Integer);
Var Item, X, Y: elements;
I, J, k: Integer;
Begin
  For I: = 1 To N-1 Do
  Begin
    K: = I;
    Item: = Data[I];
    For J: = I + 1 To N Do
    Begin
      If Data[J] < Item
      Then Begin
        X: = Data[J]; Data[J]: = Item; Item: = X
      End;
      Y: = Item;
    End;
    Item: = Data[K]; Data[K]: = Y
  End;
End;
PROGRAM SELECTION
USES CRT;
CONST SIZE = 20;
TYPE ELEMENT = INTEGER;          {OR ANY OTHER TYPE}
     AR = ARRAY [1 .. SIZE] OF ELEMENT;
VAR LINE: AR;
     I, M: INTEGER;
PROCEDURE SELECTIONSORT (VAR DATA: AR; N: INTEGER);
VAR ITEM, X, Y: ELEMENT;            
     I, J, K: INTEGER;
BEGIN
FOR I := 1 TO N-1 DO BEGIN
K := I;
ITEM := DATA [I];
FOR J := I+1 TO N DO BEGIN
IF DATA [J] < ITEM THEN BEGIN
X := DATA [J]; DATA [J] := ITEM; ITEM := X
END;
END;
Y := ITEM; ITEM := DATA [K]; DATA [K] := Y
END;
END
BEGIN {MAIN PROGRAM}
CLRSCR;
WRITELN ('REPRESENTATION OF SELECTION SORT ALGORITHM');
WRITELN ('HOW MANY DATA ITEMS YOU LIKE TO ENTER?');
READLN (M);
FOR I := 1 TO M DO BEGIN
WRITE ('ENTER THE ITEM ;');
READLN (LINE [I])
END;
SLCTSORT (LINE, M);
WRITELN ('THE SORTED DATA IS: ');
FOR I := 1 TO M DO WRITE (LINE [I]: 3);
END;
6-5) **Heap Sort**

يعتمد هذا الترتيب فكرة بناء الأشجار الثنائية وتمثيلها في مصفوفة.

**Heap**

هي صيغة بيانات تتوفر فيها خاصيتين الأولى تتبع بالشكل (shape) والذي يجب أن يكون شجرة ثنائية كاملة والخاصة الثانية تتبع بترتيب العناصر والذي يعني أن قيمة كل عقدة يجب أن تكون أكبر أو تساوي قيمة كل من عقدتيها الفرعية اليسرى واليمنى كما في الشكل.

**Reheap**

هي عملية إعادة توزيع عقد الشجرة بحيث تتوفر خاصيتين الشكل والترتيب لعناصرها ابتداء من عقد الجذر واللى ادنى عقدة في ادنى مستوى من خلال استبدال عقد الابناء التي قيمتها أكبر من عقدة الجذر لتصبح عقدة الجذر هي الأكبر.

**Algorithm of Heap Sort**

في ترتيب الاختيار يكون البحث عن أصغر عنصر في مجموعة القيم ونقله الى موقعه الصحيح، ثم يستمر البحث في المتبقية من عناصرقائمة عن أصغر عنصر فيها ليوضع في موقعه الصحيح بعد موقع العنصر السابق وهكذا الى أن يتم ترتيب القائمة. أما في هذا الترتيب يكون الترتيب بالشكل التالي:

1- تستخدم المصفوفة لخزن البيانات الأولى الغير مرتبة ثم تعمد فكرة الكومة في ترتيب العناصر كشجرة ثنائية حيث هو أكبر العناصر ويكون في الموقع الأول للمصفوفة، اما فرعي الجذر (الأيسر والايمن) فيكونا في الموقعين التاليين مباشرة.

2- ناخذ الجذر من الكومة لانه يمثل أكبر قيمة ونضعه في موقعه الصحيح.

3- إعادة ترتيب العناصر المتبقية على شكل كومة جيدة أي شجرة ثنائية أخرى، وهذا يعني أن العنصر الأكبر التالي سيكون هو الجذر فيها.

4- نكرر الخطوات أعلاه لحين الحصول على القائمة المرتبة.
Ex) Take the inputs 6, 2, 10, 1, 9, 3, 8, 4, 7, 5

Sol)

1- Can represent these entries as a binary tree and store the elements in a heap.

2- When reordering the elements of this tree, the heap becomes

3- Take the root node (10) and bring it to the top. Reorder the remaining elements following the new tree structure.
4- تؤخذ قيمة الجذر (9) وتوضع بالتتيب بعد العنصر السابق (10) ويعد ترتيب العناصر المتبقية في

نظام جديد للتصحيح:

5- تؤخذ قيمة الجذر (8) ويعد ترتيب العناصر المتبقية في نظام جديد للتصحيح:

6- تؤخذ قيمة الجذر (7) ويعد ترتيب العناصر المتبقية في نظام جديد للتصحيح:

7- تؤخذ قيمة الجذر (6) ويعد ترتيب العناصر المتبقية في نظام جديد للتصحيح:
٨- تؤخذ قيمة الجذر (٥) ويعاد ترتيب العناصر المتبقية في كومة جديدة لتصبح:

٩- تؤخذ قيمة الجذر (٤) ويعاد ترتيب العناصر المتبقية في كومة جديدة لتصبح:

١٠- تؤخذ قيمة الجذر (٣) ويعاد ترتيب العناصر المتبقية في كومة جديدة لتصبح:

١١- تؤخذ قيمة الجذر (٢) ويعاد ترتيب العناصر المتبقية في كومة جديدة لتبقي فيها العقدة الأخيرة وقيمتها (١).

١٢- ان العناصر التي اخذت تباعا بعد تشكل كل كومة وخزنت في المصفوفة التي أصبحت مرتبتة كالالتالي:

| ١ ٢ ٣ ٤ ٥ ٦ ٧ ٨ ٩ ٠ |
Procedure Heap Sort

Const n = 10; {or any other value}
Type Ar = array [1...n] of integer;
Procedure Heapsort (Var data: Ar; n: integer);
Var index: integer;
Begin
  For Index := n div 2 Downto 1 Do
    Reheap (data, Index, n);
  For Index := n Downto 2 Do
  Begin
    Swap (data [1], data [Index]);
    Reheap (data, 1, Index – 1);
    Display (data)
  End
End;

Procedure Display

Procedure Display (SS: Ar; n: integer);
Var I: integer;
Begin
  For I := 1 To n Do
    Write (SS [I]: 3)
End;

Procedure Swap

Procedure Swap (Var A, B: integer);
Var C: integer;
Begin
  C := A;
  A := B;
  B := C
End;
Procedure Reheap

Procedure Reheap (Var heap: Ar; Root, Bottom: integer);
Var  OK: Boolean;
    Max: integer;
Begin
  Ok: = False;
  While (Root * 2 <= Bottom) And (Not OK) Do 
  Begin 
    IF   Root * 2 = Bottom
        Then  max: = Root * 2
    Else
      If heap [Root * 2]  >  heap [Root * 2 +1]
            Then  max: = Root * 2
      Else  max: = Root * 2 + 1;
      If heap [Root] <   heap [max]
            Then
                Begin
                    Swap (heap [Root], heap [max]);
                    Root: = max
                End;
      Else OK: = True
  End;
End;