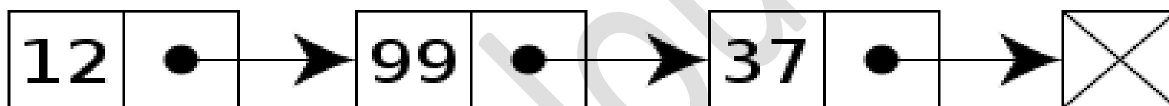
UNIT – 4 : LINKED LISTS

4.1. Singly Linked lists and Chains:

Let us discuss about the drawbacks of stacks and queues. During implementation, *overflow* occurs. No simple solution exists for more stacks and queues. In a sequential representation, the items of stack or queue are *implicitly* ordered by the sequential order of storage.

If the items of stack or queue are *explicitly* ordered, that is, each item contained within itself the address of the next item. Then a new data structure known as *linear linked list.* Each item in the list is called a *node* and contains two fields, an *information field* and a *next address field*. The information field holds the actual element on the list. The next address field contains the address of the next node in the list. Such an address, which is used to access a particular node, is known as a *pointer.*The *null pointer* is used to signal the end of a list. The list with no nodes – *empty list*or *null list.* The notations used in algorithms are:If *p* is a pointer to a node, *node(p)* refers to the node pointed to by *p. Info(p)* refersto the information of that node. *next(p)* refers to next address portion. If *next(p)* is notnull, *info(next(p))* refers to the information portion of the node that follows *node(p)* inthe list.

A linked list (or more clearly, "singly linked list") is a data structure that consists of a sequence of nodes each of which contains a reference (i.e., a *link*) to the next node in the sequence.



A linked list whose nodes contain two fields: an integer value and a link to the next node

Linked lists are among the simplest and most common data structures. They can be used to implement several other common abstract data structures, including stacks, queues, associative arrays, and symbolic expressions, though it is not uncommon to implement the other data structures directly without using a list as the basis of implementation.

The principal benefit of a linked list over a conventional array is that the list elements can easily be added or removed without reallocation or reorganization of the entire structure because the data items need not be stored contiguously in memory or on disk. Linked lists allow insertion and removal of nodes at any point in the list, and can do so with a constant number of operations if the link previous to the link being added or removed is maintained during list traversal.

On the other hand, simple linked lists by themselves do not allow random access to the data other than the first node's data, or any form of efficient indexing.

Eduoncloud.com

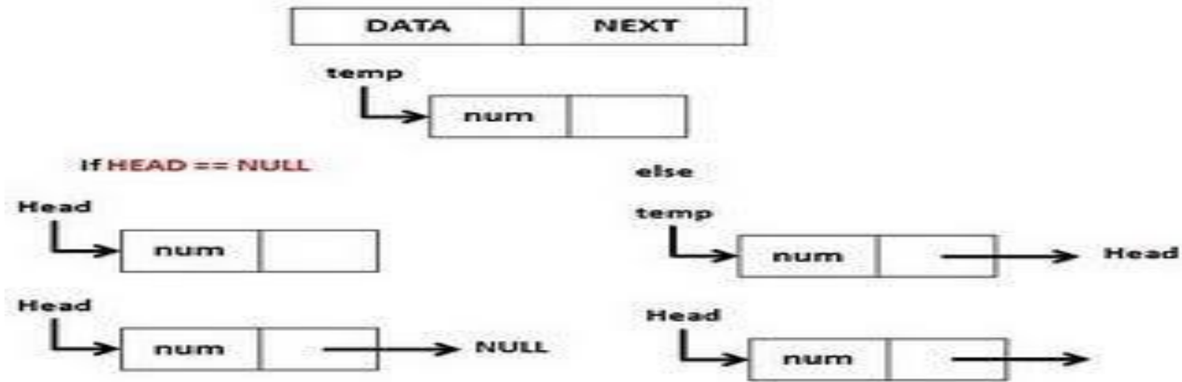


Fig :Inserting and removing nodes from a list

A list is a dynamic data structure. The number of nodes on a list may vary dramatically and dynamically as elements are inserted and removed. For example, let us consider a list with elements 5, 3 and 8 and we need to add an integer 6 to the frontof that list. Then,

p=getnode();

info(p)=6;

next(p)=list;

list=p;

Similarly, for removing an element from the list, the process is almost exactly opposite of the process to add a node to the front of the list. Remove the first node of a nonempty list and store the value of *info field* into a variable *x*. then,

p=list;

list=next(p);

x=info(p);

4.2. Representing Chains in C:

A chain is a linked list in which each node represents one element.

- There is a link or pointer from one element to the next.
- The last node has a NULL (or 0) pointer

An array and a sequential mapping is used to represent simple data structures in the previous chapters

•This representation has the property that successive nodes of the data object are stored a fixed distance apart

(1) If the element aijis stored at location Lij, then aij+1is at the location Lij+1

Eduoncloud.com

(2) If the i-thelement in a queue is at location Li, then the (i+1)-th element is at location Li+1% n for the circular representation

(3) If the topmost node of a stack is at location LT , then the node beneath it is at location LT-1, and so on

•When a sequential mapping is used for ordered lists, operations such as insertion and deletion of arbitrary elements become expensive.

In a linked representation–To access list elements in the correct order, with each element we store the address or location of the next element in the list–A linked list is comprised of nodes; each node has zero or more data fields and one or more link or pointer fields.

4.3. Linked Stacks and Queues:

Pushing a Linked Stack

```
Error code Stack :: push(const Stack entry &item)

/* Post: Stack entry item is added to the top of the Stack; returns success or

returns a code of over_ow if dynamic memory is exhausted. */

{

Node *new top = new Node(item, top node);

if (new top == NULL) return over_ow;

top node = new top;

return success;

}
```

Popping a Linked Stack

```
Error code Stack :: pop( )

/* Post: The top of the Stack is removed. If the Stack is empty the method returns

under_ow; otherwise it returns success. */

{

Node *old top = top node;

if (top node == NULL) return under_ow;

top node = old top->next;

delete old top;
```

Eduoncloud.com

```
return success;

}
```

A queue is a particular kind of collection in which the entities in the collection are kept in order and the principal (or only) operations on the collection are the addition of entities to the rear terminal position and removal of entities from the front terminal position. This makes the queue a First-In-First-Out (FIFO) data structure. In a FIFO data structure, the first element added to the queue will be the first one to be removed. This is equivalent to the requirement that once an element is added, all elements that were added before have to be removed before the new element can be invoked. A queue is an example of a linear data structure.

Queues provide services in computer science, transport, and operations research where various entities such as data, objects, persons, or events are stored and held to be processed later. In these contexts, the queue performs the function of a buffer.

```
#include<malloc.h>
#include<stdio.h>
structnode{
intvalue;
structnode*next;
};

voidInit(structnode*n){
n->next=NULL;
}
voidEnqueue(structnode*root,intvalue){
structnode*j=(structnode*)malloc(sizeof(structnode));
j->value=value;
j->next=NULL;
structnode*temp                                                    ;
temp=root;
while(temp->next!=NULL)
{
temp=temp->next;
}
temp->next=j;
printf(―Value Enqueued is : %d\n‖,value);

}
voidDequeue(structnode*root)
{
if(root->next==NULL)
{
printf(―NoElementtoDequeue\n‖);
}
```

Eduoncloud.com

```
else
{
structnode*temp;
temp=root->next;
root->next=temp->next;
printf(―ValueDequeuedis%d\n‖,temp->value);
free(temp);
}
}

voidmain()
{
structnodesample_queue;
Init(&sample_queue);
Enqueue(&sample_queue,10);
Enqueue(&sample_queue,50);
Enqueue(&sample_queue,570);
Enqueue(&sample_queue,5710);
Dequeue(&sample_queue);
Dequeue(&sample_queue);
Dequeue(&sample_queue);
}
```

4.4. Polynomials:

In mathematics, a polynomial (from Greek *poly*, "many" and medieval Latin *binomium*, "binomial"[1] [2] [3]) is an expression of finite length constructed from variables (also known as indeterminates) and constants, using only the operations of addition, subtraction, multiplication, and non-negative integer exponents. For example, $x^2 - 4x + 7$ is a polynomial, but $x^2 - 4/x + 7x^{3/2}$ is not, because its second term involves division by the variable x (4/x) and because its third term contains an exponent that is not a whole number (3/2). The term "polynomial" can also be used as an adjective, for quantities that can be expressed as a polynomial of some parameter, as in "polynomial time" which is used in computational complexity theory.

Polynomials appear in a wide variety of areas of mathematics and science. For example, they are used to form polynomial equations, which encode a wide range of problems, from elementary word problems to complicated problems in the sciences.

A polynomial is a mathematical expression involving a sum of powers in one or more variables multiplied by coefficients. A polynomial in one variable (i.e., a univariate polynomial) with constant coefficients is given by

$$a_n x^n + \ldots + a_2 x^2 + a_1 x + a_0 . \quad (1)$$

Eduoncloud.com

The individual summands with the coefficients (usually) included are called monomials (Becker and Weispfenning 1993, p. 191), whereas the products of the form $x_1^{a_1} \cdots x_n^{a_n}$ in the multivariate case, i.e., with the coefficients omitted, are called terms (Becker and Weispfenning 1993, p. 188). However, the term "monomial" is sometimes also used to mean polynomial summands *without* their coefficients, and in some older works, the definitions of monomial and term are reversed. Care is therefore needed in attempting to distinguish these conflicting usages.

The highest power in a univariate polynomial is called its order, or sometimes its degree.

Any polynomial $P(x)$ with $P(0) \neq 0$ can be expressed as

$$P(x) = P(0) \prod_{\rho} \left(1 - \frac{x}{\rho}\right), \tag{2}$$

where the product runs over the roots ρ of $P(\rho) = 0$ and it is understood that multiple roots are counted with multiplicity.

A polynomial in two variables (i.e., a bivariate polynomial) with constant coefficients is given by

$$a_{nm} x^n y^m + \ldots + a_{22} x^2 y^2 + a_{21} x^2 y + a_{12} x y^2 + a_{11} x y + a_{10} x + a_{01} y + a_{00}. \tag{3}$$

The sum of two polynomials is obtained by adding together the coefficients sharing the same powers of variables (i.e., the same terms) so, for example,

$$(a_2 x^2 + a_1 x + a_0) + (b_1 x + b_0) = a_2 x^2 + (a_1 + b_1) x + (a_0 + b_0) \tag{4}$$

and has order less than (in the case of cancellation of leading terms) or equal to the maximum order of the original two polynomials. Similarly, the product of two polynomials is obtained by multiplying term by term and combining the results, for example

$$(a_2 x^2 + a_1 x + a_0)(b_1 x + b_0) = a_2 x^2 (b_1 x + b_0) + a_1 x (b_1 x + b_0) + a_0 (b_1 x + b_0) \tag{5}$$

$$= a_2 b_1 x^3 + (a_2 b_0 + a_1 b_1) x^2 + (a_1 b_0 + a_0 b_1) x + a_0 \tag{6}$$

and has order equal to the sum of the orders of the two original polynomials.

A polynomial quotient

$$R(z) = \frac{P(z)}{Q(z)} \tag{7}$$

of two polynomials $P(z)$ and $Q(z)$ is known as a rational function. The process of performing such a division is called long division, with synthetic division being a simplified method of recording the division. For any polynomial $P(x)$, $P(x) - x$ divides $P(P(x)) - x$, meaning that the polynomial quotient is a rational polynomial or, in the case of an integer polynomial, another integer polynomial (N. Sato, pers. comm., Nov. 23, 2004).

Exchanging the coefficients of a univariate polynomial end-to-end produces a polynomial

$$a_0 x^n + a_1 x^{n-1} + \ldots + a_{n-1} x + a_n = 0 \quad (8)$$

Whose roots are reciprocals $1/x_i$ of the original roots x_i.

Horner's rule provides a computationally efficient method of forming a polynomial from a list of its coefficients, and can be implemented in *Mathematica* as follows.

```
Polynomial[l_List, x_] := Fold[x #1 + #2&, 0, l]
```

The following table gives special names given to polynomials of low orders.

polynomial order	polynomial name
2	quadratic polynomial
3	cubic polynomial
4	quartic
5	quintic
6	sextic

Polynomials of fourth degree may be computed using three multiplications and five additions if a few quantities are calculated first (Press *et al.* 1989):

$$a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 = \left[(A x + B)^2 + A x + C\right]\left[(A x + B)^2 + D\right] + E, \quad (9)$$

where $A \equiv (a_4)^{1/4}$ (10)

$$B \equiv \frac{a_3 - A^3}{4 A^3} \quad (11)$$

$$D \equiv 3 B^2 + 8 B^3 + \frac{a_1 A - 2 a_2 B}{A^2} \quad (12)$$

$$C \equiv \frac{a_2}{A^2} - 2 B - 6 B^2 - D \quad (13)$$

$$E \equiv a_0 - B^4 - B^2 (C + D) - C D. \quad (14)$$

Similarly, a polynomial of fifth degree may be computed with four multiplications and five additions, and a polynomial of sixth degree may be computed with four multiplications and seven additions.The use of linked lists is well suited to polynomial operations. We can easily imagine writing a collection of procedures for input, output addition, subtraction and multiplication of polynomials using linked lists as

the means of representation. A hypothetical user wishing to read in polynomials $A(x)$, $B(x)$ and $C(x)$ and then compute $D(x) = A(x) * B(x) + C(x)$ would write in his main program:

```
call READ(A)
call READ(B)
call READ(C)
T  PMUL(A, B)
D  PADD(T, C)
call PRINT(D)
```

Now our user may wish to continue computing more polynomials. At this point it would be useful to reclaim the nodes which are being used to represent T(x). This polynomial was created only as a partial result towards the answer D(x). By returning the nodes of T(x), they may be used to hold other polynomials.

```
procedure ERASE(T)
//return all the nodes of T to the available space list avoiding repeated
calls to procedure RET//
if T = 0 then return
p  T
while LINK (p)  0 do       //find the end of T//
p  LINK (p)
end
LlNK (p) AV        // p points to the last node of T//
AV  T      //available list now includes T//
end ERASE
```

Study this algorithm carefully. It cleverly avoids using the RET procedure to return the nodes of *T* one node at a time, but makes use of the fact that the nodes of *T* are already linked. The time required to erase *T*(*x*) is still proportional to the number of nodes in *T*. This erasing of entire polynomials can be carried out even more efficiently by modifying the list structure so that the LINK field of the last node points back to the first node as in figure 4.8. A list in which the last node points back to the first will be termed a *circular list*. A *chain* is a singly linked list in which the last node has a zero link field.

Circular lists may be erased in a fixed amount of time independent of the number of nodes in the list. The algorithm below does this.

```
procedure CERASE(T)
//return the circular list T to the available pool//
if T = 0 then return;
X  LINK (T)
LINK(T)  AV
AV  X
end CERASE
```

4.5. Additional List operations:

It is often necessary and desirable to build a variety of routines for manipulating singly linked lists. Some that we have already seen are: 1) INIT which originally links together the *AV* list; 2) GETNODE and 3) RET which get and return nodes to *AV*. Another useful operation is one which inverts a chain. This routine is especially interesting because it can be done "in place" if we make use of 3 pointers.

```
procedure INVERT(X)
//a chain pointed at by X is inverted so that if X = (a_1, ...,a_m)
then after execution X = (a_m, ...,a_1)//
p  X;q  0
while p  0 do
r  q;q  p      //r follows q; q follows p//
p  LINK(p)       //p moves to next node//
LINK(q)  r       //link q to previous node//
end
X  q
end INVERT
```

The reader should try this algorithm out on at least 3 examples: the empty list, and lists of length 1 and 2 to convince himself that he understands the mechanism. For a list of m 1 nodes, the **while** loop is executed m times and so the computing time is linear or $O(m)$.

Another useful subroutine is one which concatenates two chains X and Y.

```
procedure CONCATENATE(X, Y, Z)
//X = (a_1, ...,a_m), Y = (b_1, ...,b_n), m,n  0, produces a new chain
Z = (a_1, ...,a_m,b_1 , ...,b_n)//
Z  X
if X = 0 then [Z  Y; return]
if Y = 0 then return
p  X
while LINK(p)  0 do      //find last node of X//
p  LINK(p)
end
LINK(p)  Y      //link last node of X to Y//
end CONCATENATE
```

This algorithm is also linear in the length of the first list. From an aesthetic point of view it is nicer to write this procedure using the case statement in SPARKS. This would look like:

```
procedure CONCATENATE(X, Y, Z)
case
: X = 0 :Z  Y
: Y = 0 : Z  X
: else : p  X; Z  X
while LINK(p)  0 do
p  LINK (p)
end
LINK(p)  Y
end
end CONCATENATE
```

Suppose we want to insert a new node at the front of this list. We have to change the LINK field of the node containing x_3. This requires that we move down the entire length of A until we find the last node. It is more convenient if the name of a circular list points to the last node rather than the first.

Now we can write procedures which insert a node at the front or at the rear of a circular list and take a fixed amount of time.

procedure *INSERT FRONT*(*A, X*)
//insert the node pointed at by X to the front of the circular list
A, where A points to the last node//
if A = 0 **then** [*A X*
LINK (*X*) *A*]
else [*LINK*(*X*) *LINK* (*A*)
LINK(*A*) *X*]
end *INSERT--FRONT*

To insert *X* at the rear, one only needs to add the additional statement *A X* to the **else** clause of *INSERT_-_FRONT.*

As a last example of a simple procedure for circular lists, we write a function which determines the length of such a list.

procedure *LENGTH*(*A*)
//find the length of the circular list *A*//
i 0
if A 0 **then** [*ptr A*
repeat
i i + 1; *ptr* LINK(*ptr*)
until *ptr* = *A*]
return (*i*)
end *LENGTH*

4.6. Sparse Matrices:

A sparse matrix is a matrix populated primarily with zeros (Stoer & Bulirsch 2002, p. 619). The term itself was coined by Harry M. Markowitz.

Conceptually, sparsity corresponds to systems which are loosely coupled. Consider a line of balls connected by springs from one to the next; this is a sparse system. By contrast, if the same line of balls had springs connecting each ball to all other balls, the system would be represented by a dense matrix. The concept of sparsity is useful in combinatorics and application areas such as network theory, which have a low density of significant data or connections.

A sparse matrix is a matrix that allows special techniques to take advantage of the large number of zero elements. This definition helps to define "how many" zeros a matrix needs in order to be "sparse." The answer is that it depends on what the structure of the matrix is, and what you want to do with it. For example, a randomly generated sparse $n \times n$ matrix with $c\,n$ entries scattered randomly throughout the matrix is not sparse in the sense of Wilkinson (for direct methods) since it takes $O(n^3)$.

Creating a sparse matrix

If a matrix A is stored in ordinary (dense) format, then the command S = sparse(A) creates a copy of the matrix stored in sparse format. For example:

```
>> A = [0 0 1;1 0 2;0 -3 0]

A =

     0     0     1

     1     0     2

     0    -3     0

>> S = sparse(A)

S =

   (2,1)        1

   (3,2)       -3

   (1,3)        1

   (2,3)        2

>> whos

  Name      Size         Bytes  Class


  A         3x3             72  double array

  S         3x3             64  sparse array

Grand total is 13 elements using 136 bytes
```

Unfortunately, this form of the sparse command is not particularly useful, since if A is large, it can be very time-consuming to first create it in dense format. The command S = sparse(m,n) creates an $m \times n$ zero matrix in sparse format. Entries can then be added one-by-one:

```
>> A = sparse(3,2)

A =

   All zero sparse: 3-by-2

>> A(1,2)=1;

>> A(3,1)=4;
```

```
>> A(3,2)=-1;

>> A

A =

   (3,1)        4

   (1,2)        1

   (3,2)       -1
```

(Of course, for this to be truly useful, the nonzeros would be added in a loop.)

Another version of the sparse command is S = sparse(I,J,S,m,n,maxnz). This creates an $m \times n$ sparse matrix with entry ($I(k)$,$J(k)$) equal to $S(k), k = 1, \ldots, length(S)$. The optional argument maxnz causes Matlab to pre-allocate storage for maxnz nonzero entries, which can increase efficiency in the case when more nonzeros will be added later to S.

The most common type of sparse matrix is a banded matrix, that is, a matrix with a few nonzero diagonals. Such a matrix can be created with the spdiags command. Consider the following matrix:

```
>> A

A =

    64  -16    0  -16    0    0    0    0    0
   -16   64  -16    0  -16    0    0    0    0
     0  -16   64    0    0  -16    0    0    0
   -16    0    0   64  -16    0  -16    0    0
     0  -16    0  -16   64  -16    0  -16    0
     0    0  -16    0  -16   64    0    0  -16
     0    0    0  -16    0    0   64  -16    0
     0    0    0    0  -16    0  -16   64  -16
     0    0    0    0    0  -16    0  -16   64
```

This is a 9×9 matrix with 5 nonzero diagonals. In Matlab's indexing scheme, the nonzero diagonals of A are numbers -3, -1, 0, 1, and 3 (the main diagonal is number 0, the first subdiagonal is number -1, the first superdiagonal is number 1, and so forth). To create the same matrix in sparse format, it is first necessary to create a 9×5 matrix containing the nonzero diagonals of A. Of course, the diagonals, regarded as column vectors, have different lengths; only the main diagonal has length 9. In order to gather the various

diagonals in a single matrix, the shorter diagonals must be padded with zeros. The rule is that the extra zeros go at the bottom for subdiagonals and at the top for superdiagonals. Thus we create the following matrix:

```
>> B = [
  -16  -16   64    0    0
  -16  -16   64  -16    0
  -16    0   64  -16    0
  -16  -16   64    0  -16
  -16  -16   64  -16  -16
  -16    0   64  -16  -16
    0  -16   64    0  -16
    0  -16   64  -16  -16
    0    0   64  -16  -16
];
```

(notice the technique for entering the rows of a large matrix on several lines). The spdiags command also needs the indices of the diagonals:

```
>> d = [-3,-1,0,1,3];
```

The matrix is then created as follows:

```
S = spdiags(B,d,9,9);
```

The last two arguments give the size of S.

Perhaps the most common sparse matrix is the identity. Recall that an identity matrix can be created, in dense format, using the command eye. To create the $n \times n$ identity matrix in sparse format, use I = speye(n). Another useful command is spy, which creates a graphic displaying the sparsity pattern of a matrix. For example, the above penta-diagonal matrix A can be displayed by the following command; see Figure 6:

```
>> spy(A)
```

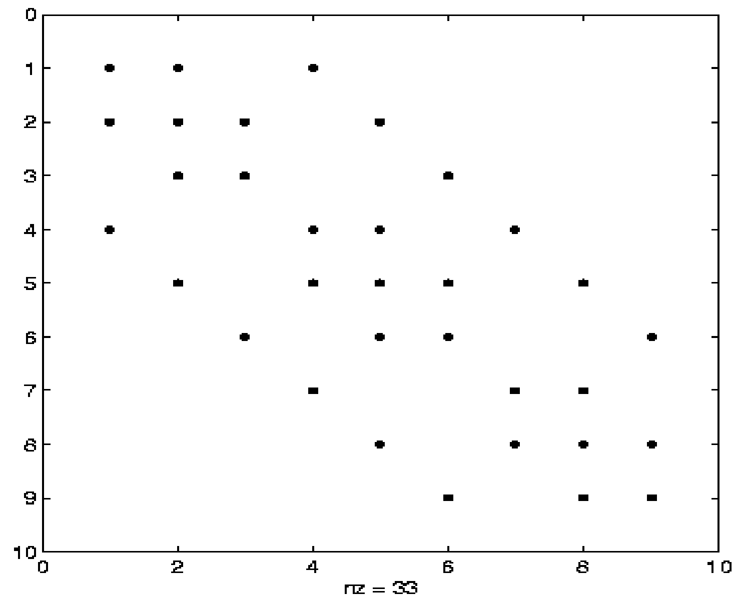


Figure 6: The sparsity pattern of a matrix

4.7. Doubly Linked Lists :

Although a circularly linked list has advantages over linear lists, it still has some drawbacks. One cannot traverse such a list backward. Double-linked lists require more space per node , and their elementary operations are more expensive; but they are often easier to manipulate because they allow sequential access to the list in both directions. In particular, one can insert or delete a node in a constant number of operations given only that node's address. (Compared with singly-linked lists, which require the *previous* node's address in order to correctly insert or delete.) Some algorithms require access in both directions. On the other hand, they do not allow tail-sharing, and cannot be used as persistent data structures.

Operations on Doubly Linked Lists

One operation that can be performed on doubly linked list but not on ordinary linked list is to delete a given node. The following c routine deletes the node pointed by pfrom a doubly linked list and stores its contents in x. It is called by delete(p).

```
delete( p )

{

NODEPTR p, q, r;

int *px;

if ( p = = NULL )

{

printf(―Void Deletion \n‖);
```

```
return;

}

*px = p -> info;

q = p -> left;

r = p -> right;

q -> right = r;

r -> left = q;

freenode( p );

return;

}
```

A node can be inserted on the right or on the left of a given node. Let us consider insertion at right side of a given node. The routine insert right inserts a node with information field x to right of node(p) in a doubly linked list.

```
insertright( p, x) {

NODEPTR p, q, r;

int x;

if ( p = = NULL ) {

printf(―Void Insertion \n‖);

return;

}

q = getnode();

q -> info = x;

r = p -> right;

r -> left = q;

q -> right = r;

q -> left = p;

p -> left = q;
```

```
return;

}
```

4.8. RECOMMENDED QUESTIONS

1.Define Linked Lists

2.List & explain the basic operations carried out in a linked list

3.List out any two applications of linked list and any two advantages of doubly linked list over singly linked list.

4. Write a C program to simulate an ordinary queue using a singly linked list.

5. Give an algorithm to insert a node at a specified position for a given singly linked list.

6. Write a C program to perform the following operations on a doubly linked list:

i) To create a list by adding each node at the front.

ii) To display all the elements in the reverse order.