MODULE II

Linear Data Structures (Linked list)

Data

Data means values or set of values. Data can be defined as representation of facts, concepts or instruction in a formalized manner which should be suitable for communication, interpretation or processing by human or machine.

Information

It is the organized or classified data . So that it has some meaningful values to the receiver. Information is the processed data on which decision and actions are performed.

Eg:-

Data	Information
16	Age of a person
	Distance between two places
28/9/1990	Age of a person
	Month of birth is September

Data Structure

In computer science data structure is a particular way of storing and organizing data in a computer, so that it can be use efficiently.

Data structure is an arrangement of data in computer memory or even disk storage (secondary memory).A data structure can be considered as a combination of data and a set of algorithms that stores and organize data in computer memory efficiently.

Thus data structure=Data +set of algorithms that stores and organize the data

Data structure perform the following

1. Storage representation of user data:-

   User data should be stored in a such a way that computer can understand it.

2. Retrieval of stored data

   The data stored in a computer should be retrieve in such a way that user can understand it.

3. Transformation of user data

   Various operations which require to be perform on user data, so that it can transform from one form to another

A data structure D is a triplet

D=(d,f,a)

Where d-→Domain

Domain is the range of values that a data made up. This domain is also term as data object.

f-→ functions

This is the set of operations which may be applied to the element of data object

a→Axioms

this is the set of rules with which different operations belongs to f actually can be implemented

Classification of Data Structure

Data structure can be classified into linear and nonlinear data structure

In case of linear data structure all the elements form a sequence and maintain a linear fashion. In case of nonlinear data structure elements are distributed over a plane, ie follow a nonlinear fashion.

Arrays

It is a collection of data of same data types stored in consecutive memory location that is referred by a common name

Linked List

It is a collection of data of same data type ,but the data items need not be stored in consecutive memory locations

Stack

A stack is Last in First Out(LIFO) data structure, here insertion and deletion takes place at one end called stack top. Here insertion is known as PUSH operation and deletion is known as POP operation

Queues

A queue is a First in First Out (FIFO) in which insertion takes place at real end and deletion takes place at front end. Insertion is known as **enqueue operation** and deletion is known as **dequeue operation.**

Trees

Trees are used to represent data that has some hierarchical relationship among data elements

Graphs

A graph is used to represent data that has relationship between pair of elements, not necessarily hierarchical

All trees are graphs, but all graphs are not trees

Tables

In table , data is arranged in no.of rows and columns

Sets

A set is a abstract data type that can store certain values without any particular order and without any repeatation.

Data types

→Data type is a term which refers to kind of data that may appear in computation.

→A data type is a classification for identifying one of the various type of data such as real numbers, integers, characters, Booleans etc

→Data types takes the possible values, the operation on that type, the way the value of that types are stored.

Built-in Data types

Data types which are built in with the programming language is called built-in data type/fundamental data type.

Eg:- the fundamental data type in C are int, float, char, double

Data type	**Number of bytes**	**Range of values**
char	1 byte	-2^7 to 2^7-1
int	2 byte	-2^{15} to $2^{15}-1$

unsigned int	2 byte	0 to 2^{16}-1
signed int	2 byte	-2^{15} to 2^{15} -1
short int	1 byte	-2^{7} to 2^{7}-1
long int	4 bytes	-2^{31} to 2^{31} -1
float	4 byte	3.4 e $^{-38}$ to 3.4e^{+38}
double	8 byte	1.7 e $^{-308}$ to 1.7e^{+308}
long double	10 bytes	3.4 e $^{-4932}$ to 1.1e^{+4932}

Fundamental data type in Fortran Language:-
integer, logical, character, complex, double precision, single precision

Fundamental data type in Pascal Language:-
real, character, boolean, array, record, file

Fundamental data type in Java Language:-
int, float, long, byte,etc

Scalar Data type

→A scalar data type is a single unit of data
→A scalar data type is a single valued data type that can be used for individual variable, constant etc.
→A scalar data type is atomic:-it is not made up of other variable components
→Hold a single value
→ Have no internal components
→The range of values depends on hardware architecture
Eg:- character, number, Boolean etc

Primitive Data type

In computer science primitive data type referred to either the following concepts

1. Basic type:- it is the data type provided by programming language as a built in block
   Most languages allow more complicated composite type to be recursively constructed starting from basic type.
2. Built-in type:- is a data type for which the programming language provides built-in support

→ In most programming languages, all basic data types are built-in.
→ In addition, many languages also provide a set of composite data types

→ Depending on the language and its implementation, primitive data types may or may not have a one-to-one correspondence with objects in the computer's memory

→Actual range of primitive data type depends upon the programming language

Typical primitive types are

1. Character
2. Integer
3. Floating point number
4. Fixed point number(fixed)
5. Reference(pointer,handle)

More complex built-in type include

→tuple in matlab or python

→linked list in LISP

→complex number in Fortran ,LISP,python

→rational number in LISP

Enumerated Data type

In computer programming, an enumerated type (also called enumeration or enum) is a data type consisting of a set of named values called elements, members, enumeral, or enumerators of the type. The enumerator names are usually identifiers that behave as constants in the language.

A variable that has been declared as having an enumerated type can be assigned any of the enumerators as a value. Some enumerator types may be built into the language. The Boolean type, for example is often a pre-defined enumeration of the values *FALSE* and *TRUE*. Many languages allow the user to define new enumerated types.

In C enumerators are created by keyword **enum**

Eg:-

enum fruit{ mango,apple,.....};--→declaration of fruit data type

enum fruit x; --→declaration of variable x of data type fruit

x=mango;

x=apple; -→ assigning values to variable x

C also allows programmers to choose the value of enumeration constants explicitly.

enum fruit{ mango=4,apple,.....};

Enumerated data type in Visual Basic can be defined as follows

```
Enum fruit
        Apple
        Mango
        .
        .
        .
End Enum
```

Subranges

Subrange facility of a programming language offers to assign range of values to variables of different data types

Eg:- subrange facility in PASCAL type

```
Tsmall:=0..9;
Var
    X:Tsmall;
begin
    x:=7;
    x:=25;
end
```

here the data type Tsmall takes the value from 0 to 9.The statement x:=7 works correctly. But x:=25 will not work, because 25 is out of these subrange. Here subrange is represented by **..**

Records

In computer science a record (structure, or tuple) is a data type consisting of 2 or more variables stored in consecutive memory locations. so that each components (called field or member of the record) can be accessed by applying different offset to a single physical address. A record type is a data type that describe such values and variables.

Most modern computer languages allow programmer to define new record types. The definition includes specification of the data type of each field, its position in the record and identifier(name or label) by which it can be accessed.

Records can exist in any storage medium, including main memory and mass storage devices such as magnetic tapes or hard disks. Records

are a fundamental component of most data structures, especially linked data structures. Many computer files are organized as arrays of logical records

A programming language that support record type usually provide the following

1. Declaration of a new record type, including the position, type, and (possibly) name of each field;
2. Declaration of variables and values as having a given record type
3. Construction of a record value from given field values
4. Selection of a field of a record with an explicit name
5. Assignment of a record value to a record variable
6. Comparison of two records for equality

Linked list

It is a data structure where the amount of memory required can be varied. A linked list is an ordered collection of finite, homogeneous data elements called nodes. Where the linear order is maintained by means of links or pointer. Ie, in a linked list adjacency is between the elements are maintained by means of pointers. A link or pointer actually stores the address of subsequent element.

An element in a linked list specially termed as nodes. A node consist of two fields. 1) data field:- to store actual information 2) link field:- to points to the next node.

The structure of node

The representation of a node can be coded as

```
struct node
{
        int data;
        struct node *link;
};
```

Consider the following example

START is a pointer which stores the address of the first node. START requires only 2 bytes because it stores only an address. Consider the first node, it contain its information 1 in data field and address of the next node (2000) in link field and so on. The last node contains only information field and link field contain NULL value. The NULL value indicates the end of the linked list.

Memory Representation

Advantages of Linked List

→Linked list is a dynamic data structure, which can grow and shrink during the execution of program

→Efficient memory utilization, ie memory is allocated whenever it is required and it is de allocated when it is not needed.

→Insertion and deletion are easier

→Many complex application can be easily carried out with linked list.

Disadvantages

→It is required more memory to store information

→Access to particular element is time consuming

→Extra care must needed for all the operation in linked list.

Operations

1)creation 2)insertion 3)searching 4)deletion 5)traversing 6) concatenation

1) Creation operation:- it is used to create a linked list
2) Insertion operation is used to insert a new node at specified location in a linked list. A new node may be inserted a) at beginning of linked list b)at the end of the linked list c) at any specified position of linked list.
3) Deletion is used to delete a particular node from any specified location, it may be a) from beginning b)from end c) from at any specified position.
4) Traversing is the process of going through all the nodes from one end to another end of the linked list exactly once.
5) Searching is the process of searching a particular key in the linked list.
6) Concatenation is the process of appending second linked list to the end of first linked list.

Classification of Linked list

1. Singly linked list
2. Doubly linked list
3. Circular linked list

Singly linked list

In singly linked list each node contains only one link field which points to the next node in linked list. Here START is a pointer which points to the first node of linked list. So , starting from first node one can reach to the last node whose link field does not contain any address rather than a NULL value. In a linked list one can move from left to right only. So it is also termed as one way list.

Insertion Operation

1)Insert node at beginning of Linked list

insert_begin_sll(START,key)

i/p: a singly linked list whose starting address is pointed by START, Key is the data to be inserted

O/p: a new node is inserted at the beginning of linked list

Data Structure: singly linked list

Algorithm

1. Start
2. Create a node and store its starting address to TEMP
3. If TEMP=NULL then
    1. Print ―node is not created, insertion is not possible‖
4. Else
    1. TEMP->data=Key
    2. TEMP->link=NULL

3. TEMP->link=START
4. START=TEMP

5. End If
6. stop

Explanation

Here we will first create an empty node by means of dynamic memory allocation. if node is created ,the memory allocation function will return the starting address of the node , otherwise returns a NULL pointer. Then we will fill data field of the newly created node as key and link field as the starting address of the old linked list.

2)Insert node at the end of Linked list

insert_end_sll(START,key)

i/p: a singly linked list whose starting address is pointed by START, Key is the data to be inserted

O/p: a new node is inserted at the end of linked list

Data Structure: singly linked list

Algorithm

1. Start
2. Create a node and store its starting address to TEMP
3. If TEMP=NULL then
    1. Print ―node is not created, insertion is not possible‖
4. Else
    1. TEMP->data=Key
    2. TEMP->link=NULL
    3. If START=NULL then
        1. START=TEMP
    4. Else
        1. TEMP1=START
        2. While TEMP1->link !=NULL do
            1. TEMP1=TEMP1->link
        3. End While
        4. TEMP1->link=TEMP
    5. End If
5. End If
6. stop

Explanation

The algorithm is used to insert a new node at the end of linked list. For this we have to create an empty node, then fill data field and link field of the newly created node and store its starting address to a pointer TEMP. Insertion at the end of linked list consist of two parts

1. for inserting a node into an empty linked list
2. inserting new node at the end of already existing linked list

for the first case, we simply assign the address of START as address of newly created node. For the second case we have to traverse linked list to last node using a pointer TEMP1. Then change the link field of the last node as address of the newly created node.

3)Insert node at the specified position of Linked list

insert_position_sll(START,key,position)

i/p: a singly linked list whose starting address is pointed by START, Key is the data to be inserted at position

O/p: a new node is inserted at the specified position

Data Structure: singly linked list

Algorithm

1. Start
2. If position <= 0 then
    1. Print ―position is invalied‖
3. else
    1. Create a node and store its starting address to TEMP
        1. If TEMP=NULL then
            1. Print ―node is not created, insertion is not possible‖
        2. Else
            1. TEMP->data=Key
            2. TEMP->link=NULL
            3. If position=1 then
                1. TEMP->link=START
                2. START=TEMP
        3. Else
            1. TEMP1=START
            2. Count=1
            3. While count<position-1 and TEMP1!=NULL do
                1. TEMP1=TEMP1->link
                2. Count=count+1
            4. End While

5. If TEMP1=NULL then
    1. Print —linked list is out of range‖
6. Else
    1. TEMP->link=TEMP1->link
    2. TEMP1->link=TEMP
7. End If

4. End if

4. End if
5. stop

Deletion Operation

1)Delete a node from the beginning of Linked list

delete_begin_sll(START)

i/p: a singly linked list whose starting address is pointed by START

O/p: a node is deleted from the beginning of linked list

Data Structure: singly linked list

Algorithm

1. Start
2. If START=NULL then
    1. Print — deletion is not possible, linked list is empty‖
3. Else
    1. TEMP=START
    2. START=START->link
    3. Delete node pointed by TEMP
4. End if
5. Stop

2)Delete a node from the end of Linked list_

delete_end_sll(START)

i/p: a singly linked list whose starting address is pointed by START

O/p: a node is deleted from the end of linked list

Data Structure: singly linked list

Algorithm

1. Start
2. If START=NULL then
    1. Print — deletion is not possible, linked list is empty‖
3. Else

1. TEMP=START
2. PREVE=START
3. While TEMP->link!=NULL do
    1. PREVE=TEMP
    2. TEMP=TEMP->link
4. End while
5. If TEMP=PREVE
    1. START=NULL
6. Else
    1. PREVE->link=NULL
7. End if
8. Delete node pointed by TEMP

4. End if
5. Stop

3) Delete a node from any position of Linked list_

delete_position_sll(START,position)

i/p: a singly linked list whose starting address is pointed by START, position where node is to be delete

O/p: a node is deleted from the specified position of linked list

Data Structure: singly linked list

Algorithm

1. Start
2. If position<=0 then
    1. Print ― position must be positive value‖
3. Else
    1. If START=NULL then
        1. Print ― linked list is empty, deletion is not possible‖
    2. else
        1. if position=1 then
            1. TEMP=START
            2. START=START->link
        2. Else
            1. TEMP=START
            2. PREVE=START
            3. Count=1
            4. While count< position and TEMP !=NULL do
                1. PREVE=TEMP
                2. TEMP=TEMP->link

5. End while
6. If TEMP=NULL then
    1. Print ―Linked list is out of range‖
7. Else
    1. PREVE->link=TEM->link
8. End if
9. Delete node pointed by TEMP

3. End if

4. End if
5. Stop

Traversing of Linked List

traverse_sll(START)

i/p: a singly linked list whose starting address is pointed by START

O/p: visit all nodes exactly once

Data Structure: singly linked list

Algorithm

1. Start
2. TEMP=START
3. If TEMP=NULL then
    1. Print ―Linked list is empty‖
4. Else
    1. While TEMP!=NULL then
        1. print TEMP->data
        2. TEMP=TEMP->link
    2. End while
5. End If
6. stop

Searching a key in Linked List

search_sll(START,key)

i/p: a singly linked list whose starting address is pointed by START, key to be search

O/p: print present or not

Data Structure: singly linked list

Algorithm

1. Start

2. TEMP=START
3. Flag=0
4. If TEMP=NULL then
    1. Print ―Linked list is empty‖
5. Else
    1. While TEMP!=NULL and flag=0 then
        1. If TEMP->data=key then
            1. Flag=1
        2. End if
        3. TEMP=TEMP->link
    2. End while
    3. If flag=0 then
        1. Print ―key is not found‖
    4. Else
        1. Print ―key is found‖
    5. End If
6. End if
7. stop

concatenation of two linked list

concatinate_sll(START1,START2)

i/p: two singly linked lists, whose starting node are pointed by START1, START2 respectively

O/p: second linked list is concatenated at the end of first linked list

Data Structure: singly linked list

Algorithm

1. Start
2. If START1=NULL then
    1. START1=START2
3. Else
    1. TEMP=START1
    2. While TEMP->link !=NULL do
        1. TEMP=TEMP->link
    3. End while
4. TEMP->link=START2
5. stop

Insertion of nodes in ascending order

insert_sll(START,key)

i/p: a singly linked list whose starting address is pointed by START, Key is the data to be inserted in ascending order

O/p: a new node is inserted in ascending order

Data Structure: singly linked list

Algorithm

1. Start
2. Create a node and store its starting address to TEMP
3. If TEMP=NULL then
    1. Print ―node is not created, insertion is not possible‖
4. Else
    1. TEMP->data=Key
    2. TEMP->link=NULL
    3. TEMP1=START
    4. PREV=START
    5. While TEMP1 !=NULL and key >TEMP1->data do
        1. PREV=TEMP1
        2. TEMP1=TEMP1->link
    6. End while
    7. If TEMP1=PREV then
        1. TEMP->link=START
        2. START=TEMP
    8. else
        1. TEMP->link=TEMP1
        2. PREV->link=TEMP
    9. End If
5. End if
6. stop

Deletion of a node from a ordered list

delete_sll(START,key)

i/p: a singly linked list whose starting address is pointed by START

O/p: a node is deleted from ordered singly linked list

Data Structure: singly linked list

Algorithm

1. Start
2. If START=NULL then
    1. Print ―Linked list is empty‖

3. else
    1. TEMP=START
    2. PREV=START
    3. While TEMP !=NULL and TEMP1->data !=key do
        1. PREV=TEMP
        2. TEMP=TEMP->link
    4. End while
    5. If TEMP=NULL then
        1. Print ―key is not found‖
    6. else
        1. if PREV=TEMP then
            1. START=START->link
        2. Else
            1. PREV->link=TEMP->link
        3. End If
        4. Delete node pointed by TEMP
    7. End if
4. End if
5. stop

Doubly Linked List

a doubly linked list is one in which are nodes are linked together by multiple links which helps to access both successor (next) and predecessor (previous) nodes from any arbitrary position.

In a singly linked list one can move from first node to any node in one direction only(from left to right). So a singly linked list is also called one way list. In other hand a doubly linked list is a two way list because one can move in either direction from left to right or right to left. This is accomplished by maintaining two link field called NEXT and PREV

Node Structure:-

Here PREV points to the left side of the node(previous node) and NEXT points to the right side of the node (successor). Each node consisting of a data field where the actual information is stored.

Node structure can be coded as

```
struct node
{
        struct node *prev;
        int data;
        struct node *next;
};
```

Operations

1)creation 2)insertion 3)searching 4)deletion 5)traversing 6) concatenation

Insertion Operation

1)Insert node at beginning of doubly Linked list

insert_begin_dll(START,key)

i/p: a doubly linked list whose starting address is pointed by START, Key is the data to be inserted

O/p: a new node is inserted at the beginning of doubly linked list

Data Structure: doubly linked list

Algorithm

1. Start
2. Create a node and store its starting address to TEMP
3. If TEMP=NULL then
    1. Print ―node is not created, insertion is not possible‖
4. Else
    1. TEMP->PREV=NULL
    2. TEMP->data=Key
    3. TEMP->NEXT=NULL
    4. If START=NULL then
        1. START=TEMP
    5. Else
        1. TEMP->NEXT=START
        2. START->PREV=TEMP
        3. START=TEMP
    6. End If
5. End if
6. stop

2)Insert node at end of doubly Linked list

insert_end_dll(START,key)

i/p: a doubly linked list whose starting address is pointed by START, Key is the data to be inserted

O/p: a new node is inserted at the end of doubly linked list

Data Structure: doubly linked list

Algorithm

1. Start
2. Create a node and store its starting address to TEMP
3. If TEMP=NULL then
    1. Print ―node is not created, insertion is not possible‖
4. Else
    1. TEMP->PREV=NULL
    2. TEMP->data=Key
    3. TEMP->NEXT=NULL
    4. If START=NULL then
        1. START=TEMP
    5. Else
        1. TEMP1=START
        2. While TEMP1->NEXT !=NULL
            1. TEMP1=TEMP1->NEXT
        3. End while
        4. TEMP1->NEXT=TEMP
        5. TEMP->PREV=TEMP1
    6. End If
5. End if
6. stop

3)Insert node at specified position of doubly Linked list

insert_position_dll(START,key,pos)

i/p: a doubly linked list whose starting address is pointed by START, Key is the data to be inserted at pos

O/p: a new node is inserted at specified position of doubly linked list

Data Structure: doubly linked list

Algorithm

1. Start
2. If pos<=0 then
    1. Print ―position must be positive‖
3. Else
    1. Create a node and store its starting address to TEMP
    2. If TEMP=NULL then
        1. Print ―node is not created, insertion is not possible‖
    3. Else
        1. TEMP->PREV=NULL
        2. TEMP->data=Key
        3. TEMP->NEXT=NULL
        4. If pos=1 then
            1. TEMP->NEXT=START
            2. If START !=NULL
                1. START->PREV=TEMP
            3. End if
            4. START=TEMP
        5. Else
        6. TEMP1=START
        7. Count=1
        8. While count<pos-1 and TEMP1!=NULL do
            1. TEMP1=TEMP1->NEXT
            2. Count=count+1
        9. End while
        10. If TEMP1=NULL then
            1. Print ―DLL is out of range‖
        11. Else
            1. If TEMP1->NEXT=NULL then
                1. TEMP1->NEXT=TEMP
                2. TEMP->PREV=TEMP1
            2. Else
                1. TEMP->NEXT=TEMP1->NEXT
                2. TEMP1->NEXT->PREV=TEMP
                3. TEMP1->NEXT=TEMP
                4. TEMP->PREV=TEMP1
            3. End if
        12. End if
    4. End If
4. End if
5. stop

Deletion Operation

1)Delete a node from the beginning of doubly Linked list_

delete_begin_dll(START)

i/p: a doubly linked list whose starting address is pointed by START

O/p: a node is deleted from the beginning of doubly linked list

Data Structure: doubly linked list

Algorithm

1. Start
2. If START=NULL then
    1. Print ―DLL is empty‖
3. Else
    1. TEMP=START
    2. If START->NEXT=NULL then
        1. START=NULL
    3. Else
        1. START=START->NEXT
        2. START->PREV=NULL
    4. End if
    5. Delete node pointed by TEMP
4. End if
5. stop

2)Delete a node from the end of doubly Linked list_

delete_end_dll(START)

i/p: a doubly linked list whose starting address is pointed by START

O/p: a node is deleted from the end of doubly linked list

Data Structure: doubly linked list

Algorithm

1. Start
2. If START=NULL then
    1. Print ―DLL is empty‖
3. Else
    1. TEMP=START
    2. If START->NEXT=NULL then
        1. START=NULL
    3. Else
        1. While TEMP->NEXT !=NULL then
            1. TEMP=TEMP->NEXT
        2. End while
        3. TEMP->PREV->NEXT=NULL

4. Delete node pointed by TEMP

4. End if

4. End if

5. stop

3)Delete a node from the specified position of doubly Linked list_

delete_position_dll(START,pos)

i/p: a doubly linked list whose starting address is pointed by START, pos from node to be deleted

O/p: a node is deleted from the specified position of doubly linked list

Data Structure: doubly linked list

Algorithm

1. Start
2. If START=NULL then
    1. Print ―DLL is empty‖
3. Else
    1. If pos<=0 then
        1. Print ―position must be positive‖
    2. else
        1. TEMP=START
        2. If pos =1 then
            1. If START->NEXT=NULL then
                1. START=NULL
            2. Else
                1. START=START->NEXT
                2. START->PREV=NULL
            3. End if
        3. else
            1. TEMP=START
            2. Count=1
            3. While count <pos and TEMP !=NULL then
                1. TEMP=TEMP->NEXT
                2. Count=count+1
            4. End while
            5. If TEMP=NULL then
                1. Print ―DLL is out of range‖
            6. Else
                1. If TEMP->NEXT=NULL then
                    1. TEMP->PREV->NEXT=NULL

2. Else
    1. TEMP->PREV->NEXT=TEMP->NEXT
    2. TEMP->NEXT->PREV=TEMP->PREV
3. End if

7. End if

4. End if

5. Delete node pointed by TEMP

3. End if

4. End if

5. stop

traverse_dll(START)

i/p: a doubly linked list whose starting address is pointed by START

O/p: visit all nodes exactly once

Data Structure: doubly linked list

Algorithm

1. Start
2. TEMP=START
3. If TEMP=NULL then
    1. Print ―DLL is empty‖
4. Else
    1. While TEMP!=NULL then
        1. print TEMP->data
        2. TEMP=TEMP->NEXT
    2. End while
5. End If
6. stop

Insert a node to an ordered Doubly Linked List

insert_dll(START,key)

i/p: a doubly linked list whose starting address is pointed by START, Key is the data to be inserted

O/p: key inserted in ascending order

Data Structure: doubly linked list

Algorithm

1. Start
2. Create a node and store its starting address to TEMP
3. If TEMP=NULL then
    1. Print ―node is not created, insertion is not possible‖

4. Else
    1. TEMP->PREV=NULL
    2. TEMP->data=Key
    3. TEMP->NEXT=NULL
    4. TEMP1=START
    5. While TEMP1->NEXT !=NULL and key >TEMP1->data do
        1. TEMP1=TEMP1->NEXT
    6. End while
    7. If TEMP1=START then
        1. If START=NULL then
            1. START=TEMP
        2. Else
            1. TEMP->NEXT=START
            2. START->PREV=TEMP
            3. START=TEMP
        3. End if
    8. else
        1. if TEMP1->NEXT=NULL
            1. TEMP1->NEXT=TEMP
            2. TEMP->PREV=TEMP1
        2. Else
            1. TEMP->PREV=TEMP1->PREV
            2. TEMP1->PREV->NEXT=TEMP
            3. TEMP->NEXT=TEMP1
            4. TEMP1->PREV=TEMP
        3. End if

    9. End If
5. End if
6. stop

Delete a node from an ordered Doubly Linked List

delete_dll(START,key)

i/p: a doubly linked list whose starting address is pointed by START, Key is the data to be deleted

O/p: key is deleted from ordered doubly linked list

Data Structure: doubly linked list

Algorithm

1. Start
2. If START=NULL then

1. Print —doubly linked list is empty‖

3. Else
    1. TEMP1=START
    2. While TEMP1 !=NULL and key !=TEMP1->data do
        1. TEMP1=TEMP1->NEXT
    3. End while
    4. If TEMP1=NULL then
        1. Print — key is not found‖
    5. else
        1. If TEMP1=START then
            1. If START->NEXT=NULL then
                1. START=NULL
            2. Else
                1. START=START->NEXT
                2. START->PREV=NULL
            3. End if
        2. else
            1. if TEMP1->NEXT=NULL
                1. TEMP1->PREV->NEXT=NULL
            2. Else
                1. TEMP1->PREV->NEXT=TEMP1->NEXT
                2. TEMP->NEXT->PREV=TEMP1->PREV
            3. End if
        3. End if
    6. End if
    7. Delete node pointed by TEMP1
4. End if
5. stop

Circular Linked List

A circular linked list is one which has no beginning and end. A singly linked list can be changed to a circular linked list by simply storing the address of the first node in the link field of the last node.

A circular doubly linked list can be form by storing the address of the first node in the NEXT field of the last node and address of the last node in the PREV field of first node.

Operations

Traversing of Circular Linked List

traverse_cll(START)

i/p: a circular singly linked list whose starting address is pointed by START

O/p: visit all nodes exactly once

Data Structure: circular singly linked list

Algorithm

1. Start
2. TEMP=START
3. If TEMP=NULL then
    1. Print ―Circular Linked list is empty‖
4. Else
    1. While TEMP->link!=START then
        1. print TEMP->data
        2. TEMP=TEMP->link
    2. End while
    3. Print TEMP->data
5. End If
6. stop

Insert node to Circular Linked list

Insert_cll(START,key)

i/p: a circular singly linked list whose starting address is pointed by START, Key is the data to be inserted

O/p: a new node is inserted to circular linked list

Data Structure: circular singly linked list

Algorithm

1. Start
2. Create a node and store its starting address to TEMP
3. If TEMP=NULL then

1. Print —node is not created, insertion is not possible‖

4. Else
    1. TEMP->data=key
    2. TEMP->link=TEMP
    3. If START=NULL then
        1. START=TEMP
    4. Else
        1. TEMP1=START
        2. While TEMP1->link !=START then
            1. TEMP1=TEMP1->link
        3. End while
        4. TEMP1->link=TEMP
        5. TEMP->link=START
    5. End if
5. end if
6. stop

Delete node from Circular Linked list

delete_cll(START,key)

i/p: a circular singly linked list whose starting address is pointed by START, Key is the data to be delete

O/p: key is deleted from circular linked list

Data Structure: circular singly linked list

Algorithm

1. Start
2. If START=NULL then
    1. Print —node is not created, insertion is not possible‖
3. Else
    1. TEMP=START
    2. While TEMP->data != key and TEMP->link != START then
        1. PREV=TEMP
        2. TEMP=TEMP->link
    3. End while
    4. If TEMP->data=key then
        1. If TEMP=START then

1. If START->link=START then
    1. START=NULL
2. Else
    1. TEMP1=START
    2. While TEMP1->link != START then
        1. TEMP1=TEMP1->link
    3. End while
    4. TEMP1->link=START->link
    5. START=START->link
3. End if

2. else
    1. PREV->link=TEMP->link
3. End if
4. Delete node pointed by TEMP

5. else
    1. print ―key is not found‖

6. end if

4. end if
5. stop

Application of Linked List

1. polynomial representation
2. sparse matrix representation

Polynomial Representation

An important application of linked list is to represent polynomial and their manipulations. Main advantages of linked list for polynomial representation is that it can accommodate number of polynomial of growing size. So that the combined size does not exceed a total memory available.

Consider general form of polynomial f(x)=

Where is the term in the polynomial , a_i is the non-zero coefficient and e_i is the exponent. Here we will arrange the terms in descending order of exponent. Ie, $e_n > e_{n-1} > e_{n-2}.....>e_1$. Therefore the node structure of term in a polynomial contains the following fields

1. coefficient
2. Exponent
3. Link field

Where coeff represent the coefficient of term and exp represent the exponent and link represent the address of the next term

Insert a term into Linked List representation of Polynomial

Insert_term_poly(START,c,e)

i/p: START represent the starting address of the polynomial(address of the first term),c represent the coefficient of new term, e represent the exponent of the new term

O/p: a new term is added to polynomial

Data Structure: singly linked list

Algorithm

1 start
2. if START=NULL then
    1. create a node and store its starting address to TEMP
    2. if TEMP=NULL then
        1. print ―node is not created ,term cannot be inserted‖
    3. else
        1. TEMP->coeff=c
        2. TEMP->exp=e
        3. TEMP->link=NULL
        4. START=TEMP
    4. End if
3. Else
    1. TEMP1=START
    2. While TEMP1->exp > e and TEMP1 !=NULL do
        1. PREV=TEMP1
        2. TEMP1=TEMP1->link
    3. End while
    4. If TEMP1=NULL then
        1. create a node and store its starting address to TEMP
        2. if TEMP=NULL then
            1. print ―node is not created ,term cannot be inserted‖
        3. else

1. TEMP->coeff=c
2. TEMP->exp=e
3. TEMP->link=NULL
4. PREV->link=TEMP

4. End if

5. Else

1. If TEMP1->exp=e then

1. TEMP1->coeff=TEMP1->coeff+c

2. Else

1. If TEMP1=START then

1. create a node and store its starting address to TEMP
2. if TEMP=NULL then

1. print ―node is not created ,term cannot be inserted‖

3. else

1. TEMP->coeff=c
2. TEMP->exp=e
3. TEMP->link=START
4. START=TEMP

4. End if

2. Else

1. create a node and store its starting address to TEMP
2. if TEMP=NULL then

1. print ―node is not created ,term cannot be inserted‖

3. else

1. TEMP->coeff=c
2. TEMP->exp=e
3. TEMP->link=TEMP1
4. PREV->link=TEMP

4. End if

3. end if

3. end if

6. end if

4.end if

5.stop

Addition of two Polynomial

Polynomial_addition(START_P,START_Q,START_R)

i/p: START_P,START_Q represent the starting address of polynomial P and Q respectively. START_R represents the starting address of the resultant polynomial.

O/p: resultant polynomial R

Data Structure: singly linked list

Algorithm

1. start
2. START_R=NULL
3. P_PTR=START_P
4. Q_PTR=START_Q
5. While P_PTR !=NULL and Q_PTR!=NULL do
    1. If P_PTR->exp=Q_PTR->exp then
        1. C=P_PTR->coeff+Q_PTR->coeff
        2. e=P_PTR->exp
        3. insert_term_poly(START_R,c,e)
        4. P_PTR=P_PTR->link
        5. Q_PTR=Q_PTR->link
    2. end if
    3. if P_PTR->exp > Q_PTR->exp then
        1. C=P_PTR->coeff
        2. e=P_PTR->exp
        3. insert_term_poly(START_R,c,e)
        4. P_PTR=P_PTR->link
    4. end if
    5. if P_PTR->exp <Q_PTR->exp then
        1. C=Q_PTR->coeff
        2. e=Q_PTR->exp
        3. insert_term_poly(START_R,c,e)
        4. Q_PTR=Q_PTR->link
    6. end if
6. end while
7. while P_PTR != NULL do
        1. C=P_PTR->coeff
        2. e=P_PTR->exp
        3. insert_term_poly(START_R,c,e)
        4. P_PTR=P_PTR->link
8. end while

9. while Q_PTR != NULL do
    1. C=Q_PTR->coeff
    2. e=Q_PTR->exp
    3. insert_term_poly(START_R,c,e)
    4. Q_PTR=Q_PTR->link
10. end while
11. stop

Multiplication of two Polynomial

Polynomial_Product(START_P,START_Q,START_R)

i/p: START_P,START_Q represent the starting address of polynomial P and Q respectively. START_R represents the starting address of the resultant polynomial.

O/p: resultant polynomial R

Data Structure: singly linked list

Algorithm

1. start
2. START_R=NULL
3. P_PTR=START_P
4. Q_PTR=START_Q
5. If START_P=NULL OR START_Q=NULL then
    1. Print ―multiplication is not possible‖
6. Else
    1. While P_PTR!=NULL do
        1. Q_PTR=START_Q
        2. While Q_PTR!=NULL do
            1. C=P_PTR->coeff * Q_PTR->coeff
            2. e=P_PTR->exp+Q_PTR->exp
            3. insert_term_poly(START_R,c,e)
            4. Q_PTR=Q_PTR->link
        3. End while
        4. P_PTR=P_PTR->link
    2. End while
7. End if
8. stop