UNIT IV:SYLLABUS

- Concept of an Abstract Data Type (ADT),
- Lists as dynamic structures,
- operations on lists,
- Implementation of linked list using arrays and its operations.
- Introduction to linked list implementation using self-referential-structures/pointers.

ABSTRACT DATA TYPE

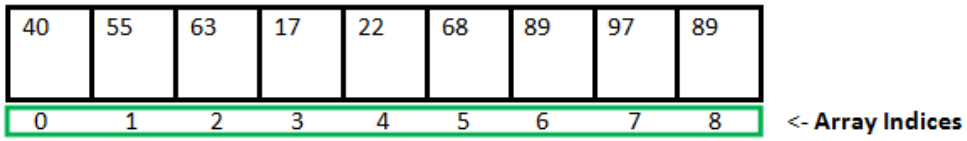
- The abstract datatype is special kind of datatype, whose behavior is defined by a set of values and set of operations.
- The keyword “Abstract” is used as we can use these datatypes, we can perform different operations. But how those operations are working that is totally hidden from the user.
- The ADT is made of with primitive datatypes, but operation logics are hidden.
- Some examples of ADT are Stack, Queue, List etc.

- List −
  - size(), this function is used to get number of elements present into the list
  - insert(x), this function is used to insert one element into the list
  - remove(x), this function is used to remove given element from the list
  - get(i), this function is used to get element at position i
  - replace(x, y), this function is used to replace x with y value

LIST AS DYNAMIC STRUCTURE

Static Data structure

In Static data structure the size of the structure is fixed. The content of the data structure can be modified but without changing the memory space allocated.



Array Length = 9
First Index = 0
Last Index = 8

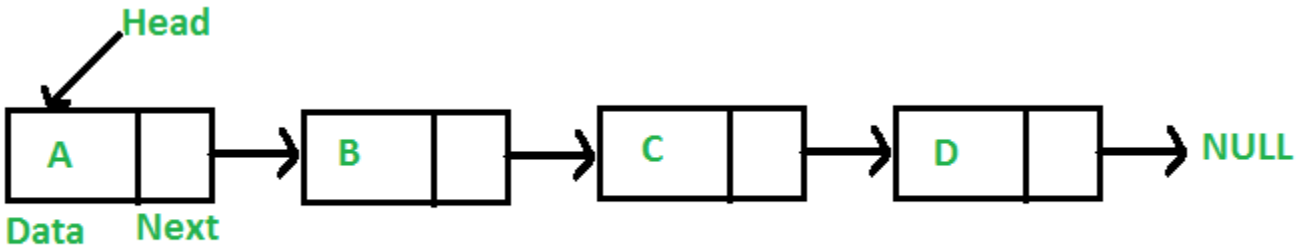
Dynamic Data Structure

In Dynamic data structure the size of the structure in not fixed and can be modified during the operations performed on it. Dynamic data structures are designed to facilitate change of data structures in the run time.

The Linked List Data Structure

A linked list is an ordered collection of nodes, each of which contains some data, connected using pointers.

Each node points to the next node in the list. The first node in the list is called the head. The last node in the list is called the tail.



Linked Lists

- A linked list consists of a finite sequence of elements or nodes that contain information plus (except possibly the last one) a link to another node.

- If node *x* points to node *y*, then *x* is called the *predecessor* of *y* and *y* the *successor* of *x*. There is a link to the first element called the *head* of the list.

LINKED LIST VERSUS ARRAYS

- An array is linear collection of data elements and a linked list is a linear collection of nodes. But unlike an array, a linked list does not store its nodes in consecutive memory locations.
- Another point of difference between an array and a linked list is that a linked list does not allow random access of data. Nodes in a linked list can be accessed only in a sequential manner.
- But like an array, insertions and deletions can be done at any point in the list in a constant time.
- Another advantage of linked list over an array is that, we can add any number of elements in the list. This is not possible in case of an array. For example, if we declare an array as int marks[10], then the array can store maximum ten data elements but not even a single more than that. There is no such restriction in case of a linked list.
- Thus linked lists provide an efficient way of storing related data and perform basic operations such as insertion, deletion and updating of information at the cost of extra space required for storing address of the next node.

Linked List vs. Array

A linked list can only be accessed sequentially.

To find the 5th element, for instance, you must start from the head and follow the links through four other nodes.

Advantages of linked list:

- Dynamic size
- Easy to add additional nodes as needed
- Easy to add or remove nodes from the middle of the list (just add or redirect links)

Advantage of array: Can easily and quickly access arbitrary elements

Types of linked lists

- Singly linked list
  - Begins with a pointer to the first node
  - Terminates with a null pointer
  - Only traversed in one direction
- Circular, singly linked
  - Pointer in the last node points back to the first node
- Doubly linked list
  - Two “start pointers” – first element and last element

Cont…

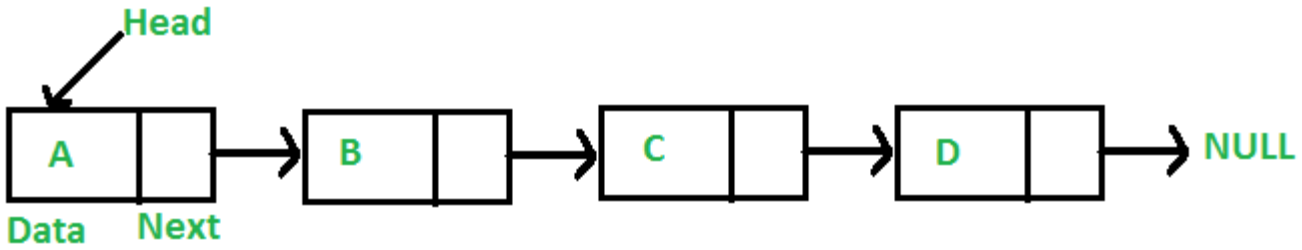
- If there is a link from the last element to the first, the list is called *circular*.
- If in a linked list each node (except possibly the first one) points also to its *predecessor*, then the list is called a *doubly linked list*.
  - Each node has a forward pointer and a backward pointer
  - Allows traversals both forwards and backwards
- If the first and the last nodes are also connected by a pair of links, then we have a *circular doubly linked list*.
  - Forward pointer of the last node points to the first node and backward pointer of the first node points to the last node

Self-Referencing structures

A structure cannot contain instances of itself but it can contain pointers to instances of itself, see:

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

This struct is called a *self-referencing structure*, which contains a member of pointer to an instance of itself. To implement dynamic data structs like a Singly Linked List I will use self-referencing structures.



Linked List (continued)

- Items of list are usually same type
  - Generally obtained from **malloc()**
- Each item points to next item
- Last item points to null
- Need “**head**” to point to first item!

Dynamic Memory Allocation

- Dynamic memory allocation
  - Obtain and release memory during execution

- malloc
  - Takes number of bytes to allocate
    - Use sizeof to determine the size of an object
  - Returns pointer of type void *
    - A void * pointer may be assigned to any pointer
    - If no memory available, returns NULL

Cont…

- Example

```
newPtr = malloc( sizeof( struct node ) );
```

- free
  - Deallocates memory allocated by malloc
  - Takes a pointer as an argument
  - free (newPtr);

Creating a link list of 3 nodes

```
struct node
{ int data;
struct node* next;
};
/* Build the list {1, 2, 3} */
#include<stdio.h>
#include<stdlib.h>
main()
{
struct node* head = NULL;
struct node* second = NULL;
struct node* third = NULL;
head =(struct node *) malloc(sizeof(struct node));          // allocate 3 nodes in the heap
second =(struct node *)  malloc(sizeof(struct node));
third = (struct node *) malloc(sizeof(struct node));
head->data = 1; // setup first node
head->next = second; // note: pointer assignment rule
second->data = 2; // setup second node
second->next = third;
third->data = 3; // setup third link
third->next = NULL;
// At this point, the linked list referenced by "head" // matches the list in the drawing.
return head;
}
```

CREATING LINKED LIST WITH N NODES

```
#include <stdio.h>
#include<stdlib.h>
struct node
{
   int data;
    struct node* link;
}*head;
main()
{
   int d, ch;
  struct node *temp, *temp1;
  head= (struct node*)malloc( sizeof (struct node));
  printf( " enter data");
  scanf("%d", &d);
  head->data= d;
  head-> link= NULL;
  temp= head;
  printf(" want to continue");
  scanf("%d",&ch);
```

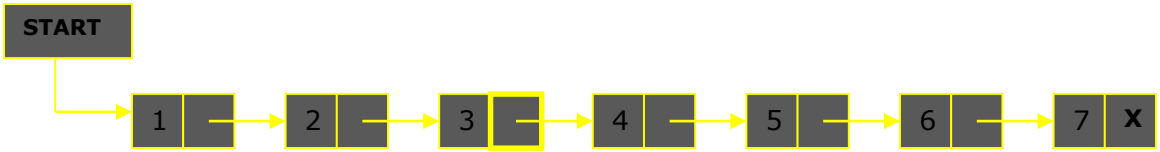
```
while(ch==1)
  {
    temp1= (struct node*)malloc( sizeof (struct node));
  printf( " enter data");
  scanf("%d", &d);
  temp1->data= d;
  temp1-> link= NULL;
  temp->link=temp1;
  temp= temp1;
  printf(" want to continue");
  scanf("%d",&ch);
  }
     temp= head;
  while(temp!=NULL)
  {
      printf("%d", temp->data);
      temp= temp->link;
  }
}
```

Basic Operations supported by a list.

- **Insertion** − Adds an element at the beginning of the list, end of list, at specified position in list
- **Deletion** − Deletes an element at the beginning of the list, end of list, at specified position in list
- **Display/ Traverse** − Traverse and Displays the complete list.
- **Search** − Searches an element using the given key.
- **Delete** − Deletes an element using the given key.

Traversing a list

- A singly linked list is the simplest type of linked list in which every node contains some data and a pointer to the next node of the same data type. By saying that the node contains a pointer to the next node we mean that the node stores the address of the next node in sequence.




```
Algorithm for traversing a linked list

Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Steps 3 and 4 while PTR != NULL
Step 3:              Apply Process to PTR->DATA
Step 4:              SET PTR = PTR->NEXT
           [END OF LOOP]
Step 5: EXIT
```

```
Algorithm to print the information stored in each node of the linked list

Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Steps 3 and 4 while PTR != NULL
Step 3:              Write PTR->DATA
Step 4:              SET PTR = PTR->NEXT
           [END OF LOOP]
Step 5: EXIT
```

```c
// C program for the all operations in
// the Singly Linked List
#include <stdio.h>
#include<stdlib.h>

struct node {
    int data;
    struct node* link;
};
struct node* start = NULL;
  // Function to traverse the linked list
void traverse()
{
    struct node* temp;

    // List is empty
    if (start == NULL)
        printf("\nList is empty\n");

    // Else print the Linked List
    else {
        temp = start;
        while (temp != NULL) {
            printf("Data = %d\n",temp->data) ;
            temp = temp->link;
        }
    }
}
```

Function to add node at the front of the linked list

```
void addAtFront()
{
    int data1;
    struct node* temp;
    temp = (struct node*) malloc(sizeof(struct node));
    printf("\nEnter number to be inserted : ");
    scanf("%d", &data1);
    temp->data  = data1;

    // Pointer of temp will be
    // assigned to start
    temp->link = start;
    start = temp;
}
```

Function to add node at the end ofthe linked list

```
void addAtEnd()
{
    int data1;
    struct node *temp, *head;
    temp = (struct node*)malloc(sizeof(struct node));

    // Enter the number
    printf("\n Enter number to be inserted : ");
    scanf("%d", &data1);

    // Changes links
    temp->link = NULL;
    temp->data = data1;
    head = start;
    while (head->link != NULL) {
        head = head->link;
    }
    head->link = temp;
}
```

Function to add node at any specified position in the linked]

```
void addAtPosition()
{
    struct node *temp, *newnode;
    int pos, data1, i = 1;
    newnode = malloc(sizeof(struct node));

    // Enter the position and data
    printf("\nEnter position and data :");
    scanf("%d %d", &pos, &data11);

    // Change Links
    temp = start;
    newnode->data = data1;
    newnode->link = 0;
    while (i < pos - 1) {
        temp = temp->link;
        i++;
    }
    newnode->link = temp->link;
    temp->link = newnode;
}
```

Function to delete from the front of the linked list

```
 void deleteFirst()
{
    struct node* temp;
    if (start == NULL)
        printf("\nList is empty\n");
    else {
        temp = start;
        start = start->link;
        free(temp);
    }
}
```

Function to delete from the end of the linked list

```
void deleteEnd()
{
    struct node *temp, *prevnode;
    if (start == NULL)
        printf("\nList is Empty\n");
    else {
        temp = start;
        while (temp->link != 0) {
            prevnode = temp;
            temp = temp->link;
        }
        free(temp);
        prevnode->link = 0;
    }
}
```

Function to delete from any specified position from the linked list

```
void deletePosition()
{
    struct node *temp, *position;
    int i = 1, pos;

    // If LL is empty
    if (start == NULL)
        printf("\nList is empty\n");
      // Otherwise
    else {
        printf("\nEnter index : ");

        // Position to be deleted
        scanf("%d", &pos);
               temp = start;

        // Traverse till position
        while (i < pos - 1) {
            temp = temp->link;
            i++;
        }
          // Change Links
        position = temp->link;
        temp->link = position->link;
          // Free memory
        free(position);
    }
}
```

Concatenate 2 single linked list

```
struct node *concat( struct node *start1,struct node *start2)
{
        struct node *ptr;
        if(start1==NULL)
        {
                start1=start2;
                return start1;
        }
        if(start2==NULL)
                return start1;
        ptr=start1;
        while(ptr->link!=NULL)
                ptr=ptr->link;
        ptr->link=start2;
        return start1;
}
```

Function to reverse the linked list

```
//void reverseLL( struct node* start)
{
   struct node *prev, *current, *nnext;

Current= start;
Nnext= start-> next
Previous= NULL;

Current->next=NULL;
While(nnext!= NULL)
{
Prev= current;
Current= nnext;
Nnext=current->next;
Current->next=prev;
}
 *start= current;
}
```

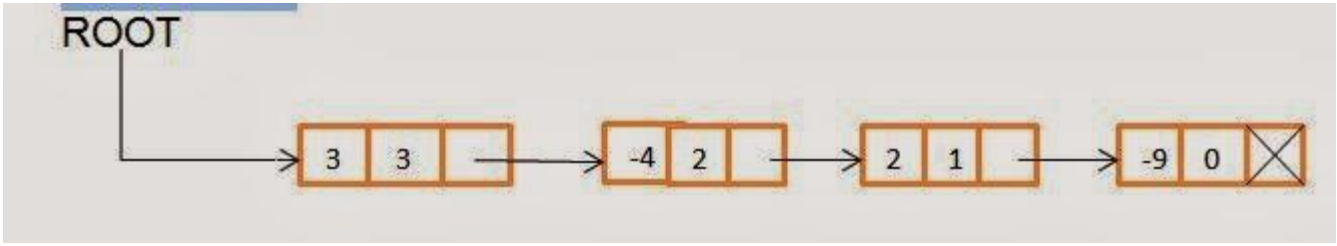
DYNAMICALLY ALLOCATED ARRAY

```
#include <stdio.h>
#include<stdlib.h>
int main()
{    int i,   *p_array;
p_array = (int *)malloc( 5*sizeof(int));      // allocate 5 integers
printf(" enter  array elements ");
for(i=0; i < 5; i++)
{
scanf("%d",&p_array[i]);
printf("%d", p_array[i]);
}
}
```

Polynomial Representation using Linked List

A polynomial can be thought of as an ordered list of non zero terms. Each non zero term is a two-tuple which holds two pieces of information:

- The exponent part
- The coefficient part




```
struct Node {
    int coeff;
    int pow;
    struct Node* next;
};
```

Linear List Application: Polynomials

- Polynomials:

$$P_n(x) = c_0 + c_1 x + c_2 x^2 + \ldots + c_n x^n$$

  - where ci$\in$R and n $\in$Nat
  - uniquely determined by a linear list:

$$(c_0, c_1, c_2, \ldots, c_n)$$

  - For this representation, all the list operations apply.

Linear List Application: Polynomials

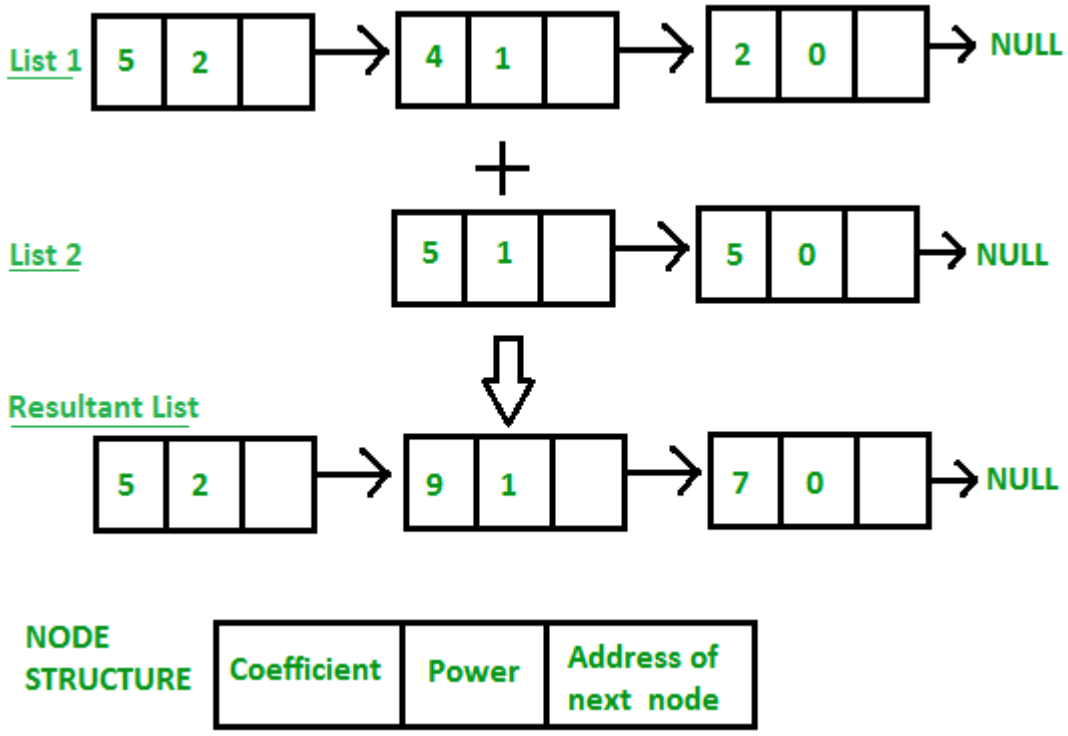
- Space waste:
  - Consider this:

$$S(x) = 1 + 3x^{10000} + 4x^{20000}$$

- A refined representation:
  - ci $((c_0, e_0), (c_1, e_1), (c_2, e_2), \ldots, (c_m, e_m))$
  - Ex:

$$((1, 0), (3, 10000), (4, 20000))$$

Polynomial addition using Linked List



```c
struct Node {
    int coeff;
    int pow;
    struct Node* next;
};
#incude<stdio.h>
#include<stdlib.h>
int main()
{
    struct Node *poly1 = NULL, *poly2 = NULL, *poly = NULL;
     // Create first list of 5x^2 + 4x^1 + 2x^0
    create_node(5, 2, &poly1);
    create_node(4, 1, &poly1);
    create_node(2, 0, &poly1);
     // Create second list of -5x^1 - 5x^0
    create_node(-5, 1, &poly2);
    create_node(-5, 0, &poly2);
     printf("1st Number: ");
    show(poly1);
     printf("\n2nd Number: ");
    show(poly2);
     poly = (struct Node*)malloc(sizeof(struct Node));
     // Function add two polynomial numbers
    polyadd(poly1, poly2, poly);
     // Display resultant List
    printf("\nAdded polynomial: ");
    show(poly);
     return 0;
}
```

```
// Function to create new node
void create_node(int x, int y, struct Node** temp)
{
    struct Node *r, *z;
    z = *temp;
    if (z == NULL) {
        r = (struct Node*)malloc(sizeof(struct Node));
        r->coeff = x;
        r->pow = y;
        *temp = r;
        r->next = (struct Node*)malloc(sizeof(struct Node));
        r = r->next;
        r->next = NULL;
    }
    else {
        r->coeff = x;
        r->pow = y;
        r->next = (struct Node*)malloc(sizeof(struct Node));
        r = r->next;
        r->next = NULL;
    }
```

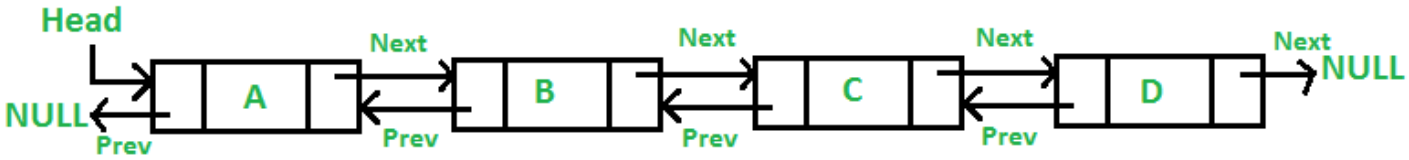
```
// Function Adding two polynomial numbers
void polyadd(struct Node* poly1, struct Node* poly2,
             struct Node* poly)
{
    while (poly1->next && poly2->next) {
        // If power of 1st polynomial is greater then 2nd,
        // then store 1st as it is and move its pointer
        if (poly1->pow > poly2->pow) {
            poly->pow = poly1->pow;
            poly->coeff = poly1->coeff;
            poly1 = poly1->next;
        }

        // If power of 2nd polynomial is greater then 1st,
        // then store 2nd as it is and move its pointer
        else if (poly1->pow < poly2->pow) {
            poly->pow = poly2->pow;
            poly->coeff = poly2->coeff;
            poly2 = poly2->next;
        }
```

```
// If power of both polynomial numbers is same then
        // add their coefficients
        else {
            poly->pow = poly1->pow;
            poly->coeff = poly1->coeff + poly2->coeff;
            poly1 = poly1->next;
            poly2 = poly2->next;
        }
         // Dynamically create new node
        poly->next
            = (struct Node*)malloc(sizeof(struct Node));
        poly = poly->next;
        poly->next = NULL;
    }
    while (poly1->next || poly2->next) {
        if (poly1->next) {
            poly->pow = poly1->pow;
            poly->coeff = poly1->coeff;
            poly1 = poly1->next;
        }
        if (poly2->next) {
            poly->pow = poly2->pow;
            poly->coeff = poly2->coeff;
            poly2 = poly2->next;
        }
        poly->next
            = (struct Node*)malloc(sizeof(struct Node));
        poly = poly->next;
        poly->next = NULL;
    }
}
```

```
// Display Linked list
void show(struct Node* node)
{
    while (node->next != NULL) {
        printf("%dx^%d", node->coeff, node->pow);
        node = node->next;
        if (node->coeff >= 0) {
            if (node->next != NULL)
                printf("+");
        }
    }
}
```

DOUBLY LINKED LIST



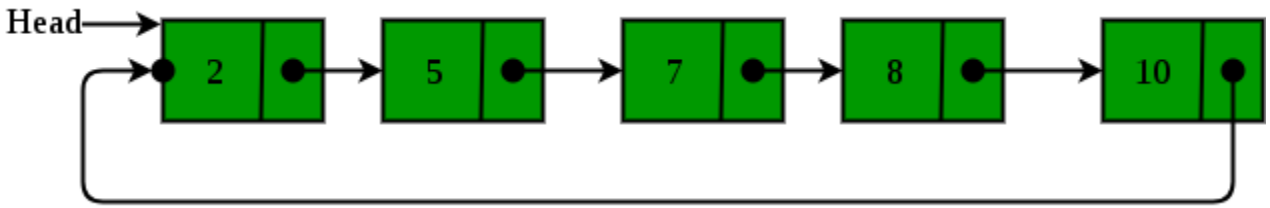
A **D**oubly **L**inked **L**ist (DLL) contains an extra pointer, typically called *previous pointer*, together with next pointer and data which are there in singly linked list.

```
struct dllNode
{
   int data;
   dllNode *next; // Pointer to next node in DLL
   dllNode *prev; // Pointer to previous node in DLL
};
```

Advantages over singly linked list

1) A DLL can be traversed in both forward and backward direction.
2) The delete operation in DLL is more efficient if pointer to the node to be deleted is given.
3) We can quickly insert a new node before a given node.
In singly linked list, to delete a node, pointer to the previous node is needed. To get this previous node, sometimes the list is traversed. In DLL, we can get the previous node using previous pointer.

Circular linked list



Advantages of Circular Linked Lists

1) Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.

2) Useful for implementation of queue. Unlike this implementation, we don't need to maintain two pointers for front and rear if we use circular linked list. We can maintain a pointer to the last inserted node and front can always be obtained as next of last.

3) Circular lists are useful in applications to repeatedly go around the list.

For example, when multiple applications are running on a PC, it is common for the operating system to put the running applications on a list and then to cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application. It is convenient for the operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list.

SUMMARY

- LINKED LIST
- Linked lists are useful to study for two reasons.
- Most obviously, linked lists are a data structure which you may want to use in real programs.
- Seeing the strengths and weaknesses of linked lists will give you an appreciation of the some of the time and space
- And code issues which are useful to thinking about any data structures in general.

References

- Data Structures, Schuam Series by Seymour Lipschutz, G.A.V pai.
- C & Data Structure by

  P.G. Deshpande and O.G. Kakde.
- An Introduction to data Structures with applications by Tremblay and Sorenson.