An introduction to Linked List

Ankit Dalal[1*] andAnkur Atri[2]

Department of Information Technology, Computer Science and Information Technology, Dronacharya College of Engineering,Gurgaon-122001, India

**E-mail: ankit.16898@ggnindia.dronacharya.info,ankur.16900@ggnindia.dronacharya.info*

Abstract:

This paper is a general overview of non-linear data structure (Linked lists). This research paper covers a brief history of the early development of Linked lists and the purpose behind this to learnTypes of linked lists, advantages and disadvantages and operations on linked lists which includes creating a linked list, traversing, insertion of a node and deletion of a node with program codes is discussed. Linked list is used to do many advanced operations in computer science. Linked lists are a great way to store a theoretically infinite amount of data with a small and versatile amount of code.

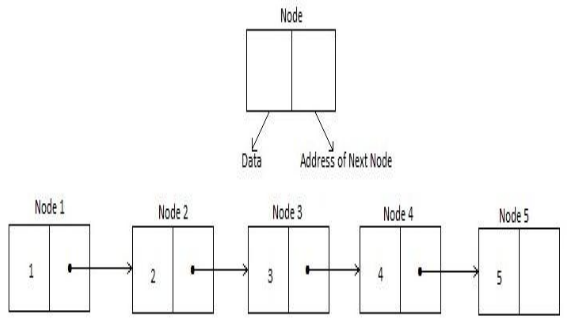
Keywords:

Node, reference, linked list, rpt, lpt and info etc.

History

Linked lists were developed in 1955-1956 by Allen Newell, Cliff Shaw and Harbert A. Simon at RAND Corporation as the primary data structure for their Information Processing Language with which early artificial intelligence programs, including the Logic Theory Machine, General Problem Solver, and a Computer Chess program were developed[1].

Introduction



Linked lists are linear data structures. The data is stored in consecutive memory location. Every element in the structure has a unique predecessor and unique successor. In this, elements are stored in a sequential form. Linked list is among the simplest and most commonly data structure used to store similar type of data in memory. It is a linear collection of data elements called nodes, where the linear order is given by means of pointers. Every node has two parts: first part contains the information/data and the second part contains the link/address of the next node in the list. Linked lists provide advantage over conventional arrays. The elements of linked list is not stored in continuous memory location[2,3]. Memory is allocated for every node when it is actually required and will be freed when not needed.

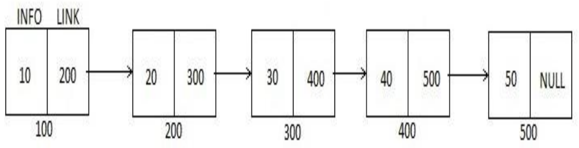
Types of linked lists

Linked lists are classified into following categories depending upon the number of pointers on the basis of requirement and usage.

1. Linear linked list
2. Circular linked list
3. Doubly linked list
4. Circular doubly linked list
5. Header linked list

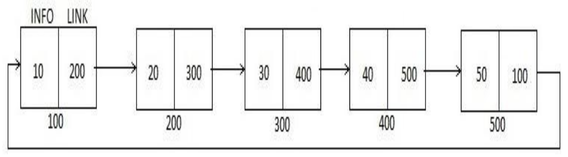
Definitions

Linear Linked List

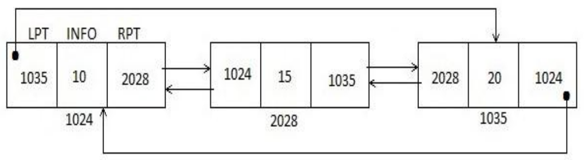


In linear linked list each node is divided into two parts: First part contains the information of the element. And the second part contains the address of the next node in the list. This means, each node has single pointer to the next node[4]. The pointer of last node is NULL representing end of the linked list.

Circular Linked List

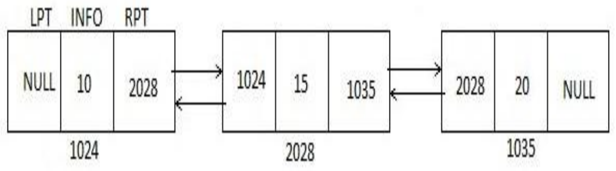


This is similar to linear linked list. But here the pointer of last node is not NULL but contains address of first node in the list.



Doubly Linked List

Doubly inked lists are two-way lists. In this



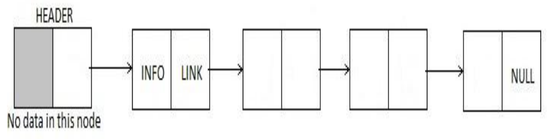
case, two link fields are there (say*lpt* which is predecessor pointer and *rpt*which is successor pointer)instead of one as in singly linked list. It helps accessing both the successor and predecessor. The rpt of last node is NULL[5,6].

Circular Doubly Linked List

Circular doubly linked list is also a two-way list which has both successor and predecessor pointer in circular manner. Here rpt of last node is not NULL but contains address of first node and lpt of first node contains the address of last node in the list. It is easier to perform insertion and deletion operation in circular doubly linked list.

Header Linked List

Header linked list always contains a single node, called *header node,* at the beginning of the linked list. This header node contains vital information about the linked list such as the number of nodes in the list, whether



the list is sorted or not[7].

Advantages

1. Linked lists are dynamic data structures: Memory is allocated for every node when it is actually required and will be freed when not needed.
2. The size is not fixed.
3. Data is stored in non-continues memory blocks.
4. Insertion and deletion of nodes are easier and efficient: Linked lists provide flexibility in inserting and deleting nodes at any specified position and a node can be deleted from any position on the linked list.

Disadvantages

More memory: In the linked list, there is an special field called **link** field which holds address of the next node, so linked lists require more space[8].

Working with Linked List

Singly linear linked list

1. Creating Nodes

```
#include<stdio.h>
#include<conio.h>
#include<alloc.h>
struct node
{
int info;
struct node *link;
};
struct node *first;
void create()
{
struct node *ptr, *cpt;
char ch;
ptr=(struct node*)malloc(sizeof(struct node));
printf("Input first node information");
scanf("%d",&ptr->info);
first=ptr;
do
{
cpt=(struct node*)malloc(sizeof(struct node));
printf("Enter next node information");
scanf("%d",&cpt->info);
ptr->link=cpt;
ptr=cpt;
printf("Press <Y/N> for more nodes");
ch=getch();
}
while(ch=='Y');
ptr->link=NULL;
}
void main()
{
clrscr();
create();
getch();
}
```

2. Traversing of a linked list

```
void traverse()
{
struct node *ptr;
printf("Traversing of linked list\n");
ptr=first;
while(ptr!=NULL)
{
printf("d",ptr->info);
prt=ptr->link;
}
}
```

3. Insertion in a Singly Linked List

   i. At the beginning

```
void insert_beginning()
{
struct node *ptr;
ptr=(struct node*)malloc(sizeof(struct node));
printf("Input New Node information");
scanf("%d",&prt->info);
ptr->link=first;
first=ptr;
}
```

   ii. At the end

```
void insert_end()
{
struct node *ptr, *cpr;
ptr=(struct node*)malloc(sizeof(struct node));
printf("Input New Node information");
scanf("%d",&ptr->info);
cpt=first;
while(cpt->link!=NULL)
cpr=cpt->link;
cpt->link=ptr;
ptr->link=NULL;
}
```

4. Deletion in a Singly Linked List

   i. From the beginning

```
void delete_beginning()
{
struct node *ptr;
if(first==NULL)
{
printf("Underflow\n");
}
ptr=first;
first=ptr->link;
free(ptr);
}
```

   ii. From the end

```
void delete_end()
{
struct node *ptr, *cpt;
if(first==NULL)
{
printf("Underflow");
}
ptr=first;
while(ptr->link!=NULL)
{
cpt=ptr;
ptr=ptr->link;
}
cpt->link=NULL;
free(ptr);
}
```

Conclusion

Linked lists are a great way to store a theoretically infinite amount of data with a small and versatile amount of code. They are great in that you can change and write them to serve your particular needs. For example: our lists were one directional, in that the individual node has no idea who is behind him in the list. This can be easily altered by the addition of a reference to the node behind as well as in front. This will give you greater control over the list. We could also write sorting algorithms, delete functions, and any number of methods that we find beneficial.

REFERENCES

1. Juan, Angel (2006). "Ch20 –Data Structures; ID06 - PROGRAMMING with JAVA (slide part of the book "Big Java", by CayS. Horstmann)" (PDF). p. 3
2. "Definition of a linked list". National Institute of Standards and Technology. 2004-08-16. Retrieved 2004-12-14.
3. Antonakos, James L.; Mansfield, Kenneth C., Jr. (1999). *Practical Data Structures Using C/C++*. Prentice-Hall. pp. 165–190. ISBN 0-13-280843-9.
4. Collins, William J. (2005) [2002]. *Data Structures and the Java Collections Framework*. New York: McGraw Hill. pp. 239–303. ISBN 0-07-282379-8.
5. Cormen, Thomas H.; Charles E. Leiserson; Ronald L. Rivest; Clifford Stein (2003). *Introduction to Algorithms*. MIT Press. pp. 205–213 & 501–505. ISBN 0-262-03293-7.
6. Cormen, Thomas H.; Charles E. Leiserson; Ronald L. Rivest; Clifford Stein (2001). "10.2: Linked lists". *Introduction to Algorithms* (2md ed.). MIT Press. pp. 204–209. ISBN 0-262-03293-7.
7. Green, Bert F. Jr. (1961). "Computer Languages for Symbol Manipulation". *IRE Transactions on Human Factors in Electronics* (2): 3–8.doi:10.1109/THFE2.1961.4503292.
8. McCarthy, John (1960). "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I". *Communications of the ACM* **3** (4): 184.doi:10.1145/367177.367199.