Chapter 9: Linked Lists

How can a program access or modify a variable indirectly? Is it possible for two variables to refer to the same memory location? What is the difference between a variable and its address in memory? How are arrays passed to functions in C? Why do some programs crash when accessing memory incorrectly? How does dynamic memory allocation work? What problems may occur if memory is not managed properly?

This chapter answers these questions.

8.1 Introduction

In the previous chapters, we studied static and dynamic arrays. Although arrays are simple and efficient for indexed access, they suffer from important limitations such as fixed size (static arrays) and costly insertions or deletions.

To overcome these limitations, we introduce Linked Lists, which are dynamic data structures that allow efficient insertion and deletion operations without shifting elements in memory.

8.2 Definition of a Linked List

A linked list is a linear data structure that does not have a fixed size at the time of its creation. Its elements, which are of the same type, are scattered in memory (The nodes are not stored contiguously in memory) and connected to one another by pointers. Its size can be modified according to the available memory. The list is accessible only through its head, that is, its first element.

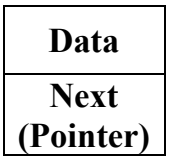
Each element of a linked list is called node, where:

- Each node contains data
- Each node contains a pointer (or link) to the next node in the list

A linked list is implemented through the pointer carried by each node, which indicates the location of the next node. The last node in the list does not point to anything (NIL: Not Identified Link). An element of the list is accessed by traversing the nodes using their pointers.

General Structure of a Node

Adress of the node



In a linked list, the head is a pointer that stores the address of the first node of the list. It is the only direct access point to the entire list. If the head is NLL, the list is considered empty. All operations on a linked list—such as traversal, insertion, or deletion—start from the head, since the other nodes are accessed sequentially by following the pointers from one node to the next.

8.3 Declaration of a Linked List

Declaring a linked list is, in practice, equivalent to declaring a pointer to the first node, which is a record (or struct).

Syntax

In algorithmics

1st Method

```
Type Node_id = Record ;
    Data_id : Type ;
    Next_id : ^Node_id ;
End;
Type List_id = ^ Node_id;
```

2nd Method

```
Type List_id = ^ Node_id ;
    Node_id = Record
    Data_id : Type ;
    Next_id : List_id;
End;
```

Example:

```
Type Node = Record ;
    Person_id : String[30];
    FName, FirstN: String[30];
    Age: integer;
    Next : ^Node;
End;
Type List = ^ Node;
```

In C language

```
typedef struct Node_id {
   type data;
   struct Node_id * next_id;
} Node_id;

Typedef Node_id * List_id;
```

8.4 Types a Linked List

a. Singly Linked List: In such lists:

- Each node contains data and a pointer to the next node only.
- The last node points to NIL.
- Access is sequential from the head.
- Simple and memory-efficient, but cannot traverse backward.

b. Doubly Linked List : In such lists:

- Each node contains data and two pointers:
    - One to the previous node
    - One to the next node
- Allows traversal forward and backward.
- Requires extra memory for the previous pointer.

c. Circular Linked List : In such lists:

- The last node points back to the first node, forming a circle.
- Can be singly or doubly circular.
- Useful for applications like round-robin scheduling.
- No NIL pointers, so traversal can continue indefinitely if not stopped.

8.5 Manipulation of a Linked List (Case of a singly one)

8.5.1 Create a list (Singly):

```
Algorithm Create_list;
    Type Node = Record ;
        Person_id : String[30];
        FName, FirstN: String[30];
        Age: integer;
        Next : ^Node;
    End;
    Type List = ^ Node;
Variables Head, Current : List;
                Choice : Integer ;
Begin
Allocate (Head) ; //Or Allocate (Current) ; Head ← Current;
Current ← Head ;
Repeat
   Write ("Enter the identifier of the person : ");
   Read (Current ^. Person_id) ;
   Write ("Enter the family name of the person : ");
   Read (Current ^.FName) ;
   Write ("Enter the first name of the person : ");
   Read (Current ^.FirstN) ;
   Write ("Enter the age of the person : ");
   Read (Current ^.Age) ;
   Write ("Another entry ? 1. Yes  2.No ") ;
   Repeat
       Read (Choice) ;
   Until (Choice=1) or (Choice=2);
   If (Choice = 1 ) then
      Allocate(Current ^.Next) ;
      Current ← Current ^.Next;
   Else
      Current ^.Next ← NIL;
      Current ← NIL;
    End-If;
Until (Choice = 2);
End.
```

8.5.2 Insertion of a node in a Singly Linked List (SLL)

To insert an element in a Singly Linked List, there are three cases:

- We insert the node at the head (beginning) of the list.
- We insert the node in the middle of the list.
- We insert the node at the tail (end) of the list.

8.5.2.1 Insertion at the head of the list

…

```
Variables Head, New: P;
Begin
//We assume that the list is already created using the “Create_list” algorithm

Allocate (New);
Write ("Enter the identifier of the person : ");
Read (New^. Person_id) ;
Write ("Enter the family name of the person : ");
Read (New^.FName) ;
Write ("Enter the first name of the person : ");
Read (New^.FirstN) ;
Write ("Enter the age of the person : ");
Read (New^.Age) ;
New^.Next ← Head;
Head ← New;
New ← NIL;
End.
```

8.5.2.2 Insertion at the middle of the list

…

```
Variables Head, New, Current: P;
        ID: String[30]; Found: Boolean;

Begin
//We assume that the list is already created using the “Create_list” algorithm
// The node will be insert after a given identifier of a person

Write ("Enter the ID of the person after whom you want to insert another person: ");
Write ("Enter the identifier of the person after whom you want to insert: ");
Read(ID);
Found ← False;
Current ← Head;
While (Current ≠ NIL and Found= False) do
    If (Current^.Person_ID = ID) then Found ← True;
    Else
        Current ← Current^.Next;
    End-If;
End-While;
```

```
If (Current=NIL) then write("ID not found");
Else
        Allocate (New);
        Write ("Enter the identifier of the person : ");
        Read (New^. Person_id) ;
        Write ("Enter the family name of the person : ");
        Read (New^.FName) ;
        Write ("Enter the first name of the person : ");
        Read (New^.FirstN) ;
        Write ("Enter the age of the person : ");
        Read (New^.Age) ;
        New^.Next ← Current^.Next;
        Current^.Next ← New;
        New ← NIL;
End-If;
Current  ← NIL;
End.
```

8.5.2.3 Insertion at the end (tail) of the list

…

```
Variables Head, New: P;
Begin
//We assume that the list is already created using the “Create_list” algorithm

Allocate (New);
Write ("Enter the identifier of the person : ");
Read (New^. Person_id) ;
Write ("Enter the family name of the person : ");
Read (New^.FName) ;
Write ("Enter the first name of the person : ");;, Current
Read (New^.FirstN) ;
Write ("Enter the age of the person : ");
Read (New^.Age) ;
New^.Next ← NIL;
Current ← Head;
While (Current^.Next ≠ NIL) do
        Current ← Current^.Next;
End-While;
Current^.Next ← New;
Current ← NIL;
New ← NIL ;
End.
```

8.5.3 Deletion of a node in a Singly Linked List (SLL)

To delete an element in a Singly Linked List, there are three cases:

- We delete the node at the head (beginning) of the list.
- We delete the node in the middle of the list.
- We delete the node at the tail (end) of the list.

8.5.3.1 Deletion at the head of the list

…

Variables Head, Temp: P;
Begin
//We assume that the list is already created using the "Create_list" algorithm

Temp ← Head;
Head ← Head^.Next;
Release (Temp);
Temp ← NIL;
End.

8.5.3.2 Deletion at the middle of the list

…

Variables Head, Temp, Current: P;
ID: String[30]; Found: Boolean;

Begin
//We assume that the list is already created using the "Create_list" algorithm
//We assume that the searched id is at the middle
// The node with a given identifier of a person will be deleted we

Write ("Enter the ID of the person after whom you want to insert another person: ");
Read(ID);
Found ← False;
Current ← Head; // Current ← Head^.next;
Temp ← NIL;// Temp ← Head;
While (Current ≠ NIL and Found= False) **do**
If (Current^.Person_ID = ID) **then** Found ← True;
Else
Temp ← Current;
Current ← Current^.Next;
End-If;

```
End-While;
If (Current=NIL) then write("ID not found");
Else
        Temp^.Next ← Current^.Next;
        Release(Current);
End-If;
Current  ← NIL;
Temp ← NIL;
End.
```

8.5.3.3 Deletion at the end (tail) of the list

…

```
Variables Head, Current, Temp: P;
Begin
//We assume that the list is already created using the "Create_list" algorithm

Current ← Head;
While (Current^.Next ≠ NIL) do
        Temp ← Current;
        Current ← Current^.Next;
End-While;
Temp^.Next ← NIL;
Release (Current);
Current ← NIL;
Temp ← NIL ;
End.
```

8.5.3.4 Deletion of the whole list

…

```
Variables Head, Current : P;
Begin
//We assume that the list is already created using the "Create_list" algorithm


While (Head ≠ NIL) do
        Current ← Head;
        Head ← Head^.Next;
        Release (Current);
End-While;
Current ← NIL;
End.
```