Chapter - 10 : Self Referential Structures and Linked Lists

Self Referential Structures

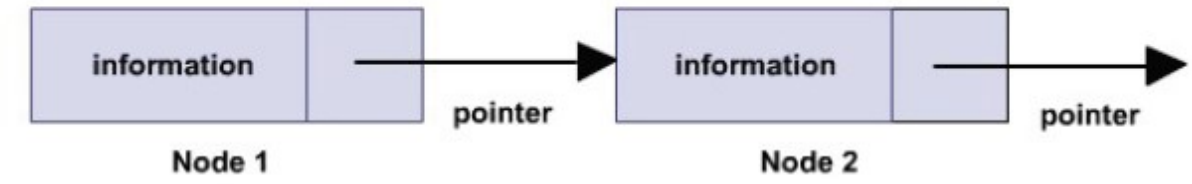
- A structure can have members which point to a structure variable of the same type.
- These types of structures are called self referential structures and are widely used in dynamic data structures like trees, linked list, etc.
- The following is a definition of a self referential structure.

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

Here, **next** is a pointer to a **struct** node variable.

- It should be remembered that a pointer to a structure is similar to a pointer to any other variable.
- A self referential data structure is essentially a structure definition which includes at least one member that is a pointer to the structure of its own kind.

Self Referential Structures and Linked Lists

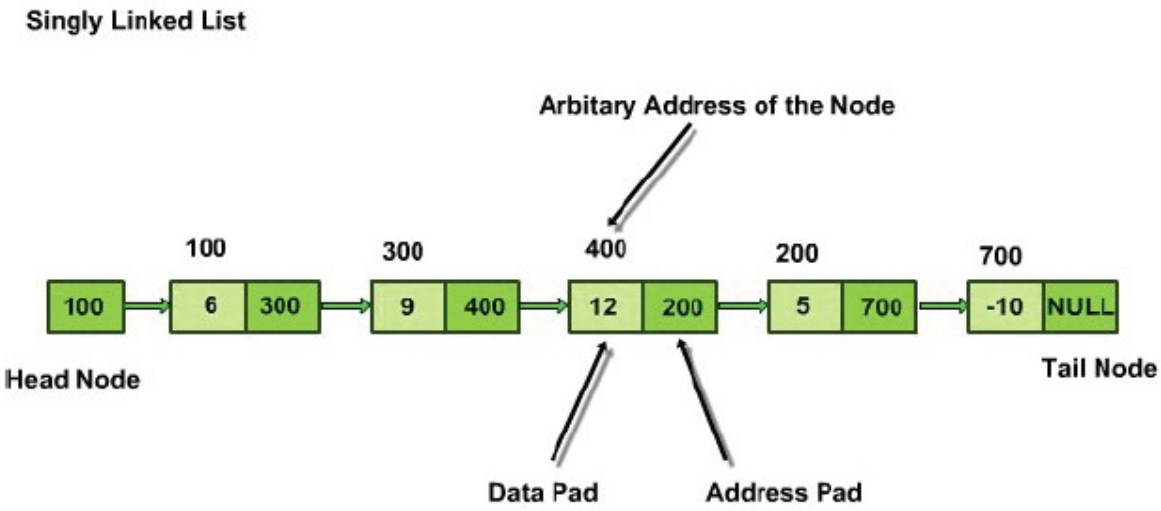


- Such self referential structures are very useful in applications that involve linked data structures, such as lists and trees.
- Unlike a static data structure such as array where the number of elements that can be inserted in the array is limited by the size of the array, a self referential structure can dynamically be expanded or contracted.
- Operations like insertion or deletion of nodes in a self referential structure involve simple and straight forward alteration of pointers.

Linked Lists

- A linked list is a useful method of data storage that can easily be implemented in C.
- There are several kinds of linked lists, including single linked lists, double linked lists, and binary trees.
- Each type is suited for certain types of data storage.
- The one thing that these lists have in common is that the links between data items are defined by information that is contained in the items themselves, in the form of pointers.
- This is distinctly different from arrays, in which the links between data items result from the layout and storage of the array.

Creation of a Singly Connected Linked List



Basics of Linked Lists

- Each data item in a linked list is contained in a structure.
- The structure contains the data elements needed to hold the data being stored. These depend on the needs of the specific program.
- In addition, there is one more data element - a pointer.
- This pointer produces the links in a linked list.

```
struct person
{
        char name[20];
        struct person *next;
};
```

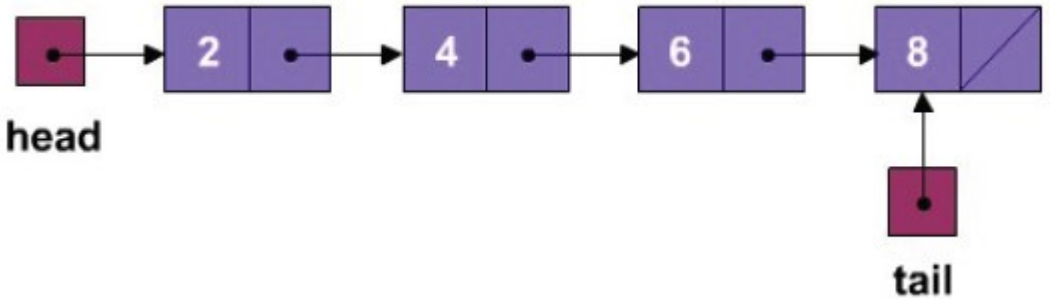
This code defines a structure named person.

- For the data, person contains only a 20 element array of characters.
- The person structure also contains a pointer to type person in other words, a pointer to another structure of the same type.
- This means that each structure of type person can not only contain a chunk of data, but also can point to another person structure.

Head Pointer

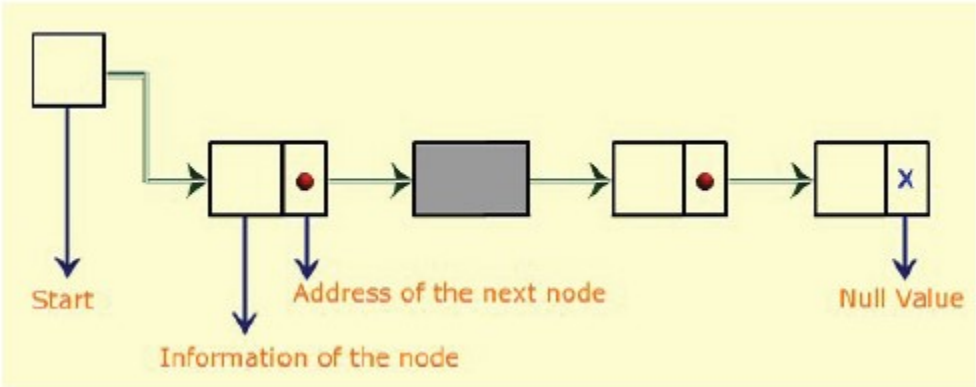
- This is identified by a special pointer (not a structure) called the head pointer.
- The head pointer always points to the first element in the linked list.
- The first element contains a pointer to the second element; the second element contains a pointer to the third, and so on until one encounters an element whose pointer is NULL.
- If the entire list is empty (contains no links), the head pointer is set to NULL.
- The head pointer is a pointer to the first element in a linked list.
- The head pointer is sometimes referred to as the first element pointer or top pointer.

Self Referential Structures and Linked Lists Head and Tail



Traversing Linked List

- Linked list is essentially an ordered list of some data in which one can traverse it (examine it) in only one direction.
- Linked list is made up of nodes that consist of data section that holds the actual data and a link section, which contains information about the next node in the list.
- The link section contains a pointer to the next node, or in array implementation of the list, an index of the next node.
- Usually Linked List has the following functions:
    1. **Add** – adds an element at the end of the list.
    2. **Add** (position) – adds an element at a certain position.
    3. **Remove** (position) – removes a certain element.
    4. **Traverse** – traverses the list and possibly display its content on the screen.
    5. **IsEmpty** – function that checks if the list is empty or not.
- It is important to check the position parameter when Add (position) and Remove (position) functions are called.
- Suppose the list has 5 elements in total – 0 to 4.
- It would be legal to call Add or Remove with position parameter in the range of 0 to 4 and also it would be legal to call Add with the position parameter value of 5, which would essentially add a new Node at the very end of the list (after the 4th element).
- Thus in the above linked list of 4 elements Node 0 has a pointer to Node 1, Node 1 to Node 2, Node 2 to Node 3.
- Also need to keep a pointer to the first node in the list, First Node (which is Node 0 in the above example) to be able to access the list at all.
- Clearly, next is used to create a link to the next node in the list, hence the name linked list.
- Linked list can be represented graphically as follows:



- Steps for Traversing the linked list:
  1. Step 1: Start at the beginning – First Node.
  2. Step 2: Access node′s data section.
  3. Step 3: Print out the data or does some other work per application specification.
  4. Step 4: Access the next node by using the next section of the current node.
- Repeat step 2.
- Do steps 2 and 3 as long as the last node is not reached.
- Last node will point to nothing in its next section.
- Nothing is represented variously depending on the actual implementation.

Program – To implement the Link List and print the items of the node

```
#include<stdio.h>
#include<alloc.h>
#include<stdlib.h>

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

void main( )
{
        struct node *head,*temp;

        head=NULL;

        //First Node
        temp=(strcut node*)calloc(1,sizeof(struct node));

        printf("Enter data ");
        scanf("%d",&temp->data);
        temp->next=head;

        head=temp;
```

```
        //Second Node
        temp=(strcut node*)calloc(1,sizeof(struct node));

        printf("Enter data ");
        scanf("%d",&temp->data);
        temp->next=head;

        head=temp;

        //Third Node
        temp=(strcut node*)calloc(1,sizeof(struct node));

        printf("Enter data ");
        scanf("%d",&temp->data);
        temp->next=head;

        head=temp;


        //print the list items

        printf("List items are - "          );
        temp=head;

        while(temp!=NULL)
        {
          printf("%d", temp->data);
          temp=temp->next;
        }

}
```