In a game of Treasure Hunt, you start by looking for the first clue. When you find it, instead of having the treasure, it has the location of the next clue and so on. You keep following the clues until you get to the treasure.

A linked list is similar. It is a series of connected "nodes" that contains the "address" of the next node. Each node can store a data point which may be a number, a string or any other type of data.

Linked List Representation

linked list concept of chaining data points

You have to start somewhere, so we give the address of the first node a special name called HEAD.

Also, the last node in the linkedlist can be identified because its next portion points to NULL.

How another node is referenced?

Some pointer magic is involved. Let's think about what each node contains:

A data item

An address of another node

We wrap both the data item and the next node reference in a struct as:

struct node

{

int data;

struct node \*next;

};

Understanding the structure of a linked list node is the key to having a grasp on it.

Each struct node has a data item and a pointer to another struct node. Let us create a simple Linked List with three items to understand how this works.

/\* Initialize nodes \*/

struct node \*head;

struct node \*one = NULL;

struct node \*two = NULL;

struct node \*three = NULL;

/\* Allocate memory \*/

one = malloc(sizeof(struct node));

two = malloc(sizeof(struct node));

three = malloc(sizeof(struct node));

/\* Assign data values \*/

one->data = 1;

two->data = 2;

three->data=3;

/\* Connect nodes \*/

one->next = two;

two->next = three;

three->next = NULL;

/\* Save address of first node in head \*/

head = one;

If you didn't understand any of the lines above, all you need is a refresher on pointers and structs.

In just a few steps, we have created a simple linkedlist with three nodes.

linked list with data

The power of linkedlist comes from the ability to break the chain and rejoin it. E.g. if you wanted to put an element 4 between 1 and 2, the steps would be:

Create a new struct node and allocate memory to it.

Add its data value as 4

Point its next pointer to the struct node containing 2 as data value

Change next pointer of "1" to the node we just created.

Doing something similar in an array would have required shifting the positions of all the subsequent elements.

Utility of Linked List

Lists are one of the most popular and efficient data structures, with implementation in every programming language like C, C++, Python, Java and C#.

Apart from that, linked lists are a great way to learn how pointers work. By practicing how to manipulate linked lists, you can prepare yourself to learn more advanced data structures like graphs and trees.

Types of Linked List

There are three common types of Linked List.

1. [Singly Linked List](https://www.programiz.com/dsa/linked-list-types#singly%20linked%20list)
2. [Doubly Linked List](https://www.programiz.com/dsa/linked-list-types#Doubly%20Linked%20List)
3. [Circular Linked List](https://www.programiz.com/dsa/linked-list-types#Circular%20Linked%20List)

Singly Linked List

It is the most common. Each node has data and a pointer to the next node.



Node is represented as:

struct node {

int data;

struct node \*next;

}

A three member singly linked list can be created as:

/\* Initialize nodes \*/

struct node \*head;

struct node \*one = NULL;

struct node \*two = NULL;

struct node \*three = NULL;

/\* Allocate memory \*/

one = malloc(sizeof(struct node));

two = malloc(sizeof(struct node));

three = malloc(sizeof(struct node));

/\* Assign data values \*/

one->data = 1;

two->data = 2;

three->data = 3;

/\* Connect nodes \*/

one->next = two;

two->next = three;

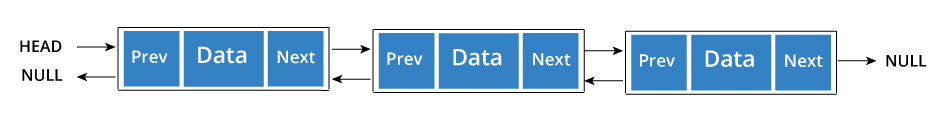
three->next = NULL;

/\* Save address of first node in head \*/

head = one;

Doubly Linked List

We add a pointer to the previous node in a doubly linked list. Thus, we can go in either direction: forward or backward.



A node is represented as

struct node {

int data;

struct node \*next;

struct node \*prev;

}

A three member doubly linked list can be created as

/\* Initialize nodes \*/

struct node \*head;

struct node \*one = NULL;

struct node \*two = NULL;

struct node \*three = NULL;

/\* Allocate memory \*/

one = malloc(sizeof(struct node));

two = malloc(sizeof(struct node));

three = malloc(sizeof(struct node));

/\* Assign data values \*/

one->data = 1;

two->data = 2;

three->data = 3;

/\* Connect nodes \*/

one->next = two;

one->prev = NULL;

two->next = three;

two->prev = one;

three->next = NULL;

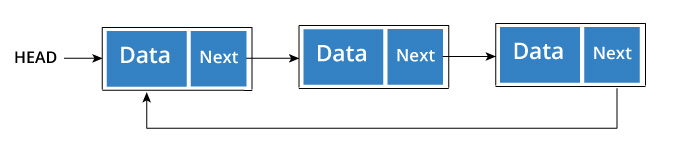
three->prev = two;

/\* Save address of first node in head \*/

head = one;

Circular Linked List

A circular linked list is a variation of linked list in which the last element is linked to the first element. This forms a circular loop.



A circular linked list can be either singly linked or doubly linked.

• for singly linked list, next pointer of last item points to the first item

• In doubly linked list, prev pointer of first item points to last item as well.

A three member circular singly linked list can be created as:

/\* Initialize nodes \*/

struct node \*head;

struct node \*one = NULL;

struct node \*two = NULL;

struct node \*three = NULL;

/\* Allocate memory \*/

one = malloc(sizeof(struct node));

two = malloc(sizeof(struct node));

three = malloc(sizeof(struct node));

/\* Assign data values \*/

one->data = 1;

two->data = 2;

three->data = 3;

/\* Connect nodes \*/

one->next = two;

two->next = three;

three->next = one;

/\* Save address of first node in head \*/

head = one;

**Self Referential Data Structure in C - create a singly linked list**

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. A chain of such structures can thus be expressed as follows.

struct name {

member 1;

member 2;

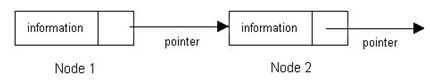
. . .

struct name \*pointer;

};

The above illustrated structure prototype describes one node that comprises of two logical segments. One of them stores data/information and the other one is a pointer indicating where the next component can be found. .Several such inter-connected nodes create a chain of structures.

The following figure depicts the composition of such a node. The figure is a simplified illustration of nodes that collectively form a chain of structures or linked list.



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.

Linear (Singly) Linked List

A linear linked list is a chain of structures where each node points to the next node to create a list. To keep track of the starting node's address a dedicated pointer (referred as start pointer) is used. The end of the list is indicated by a NULL pointer. In order to create a linked list of integers, we define each of its element (referred as node) using the following declaration.

struct node\_type {

int data;

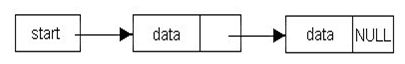
struct node\_type \*next;

};

struct node\_type \*start = NULL;

Note: The second member points to a node of same type.

A linear linked list illustration:



Example

Let us now develop a C program to manipulate linked lists. For this purpose we introduce a few basic functions, which can be used to create a list, displaying its contents, inserting into a list and deleting an existing element. We also introduce two functions reverse and recReverse for reversing the elements of the list.

When a list is created a pointer called start is used to indicate the beginning of the list. A function createNode, creates a node and returns a pointer to it. The function insert is used to insert a new node in an existing list provided the data is not already present in the list. If it is not present, we place the data in a manner so that the new element is appended at the end of the list.

#include <stdio.h>

struct node\_type {

int data;

struct node\_type \*next;

};

typedef struct node\_type list;

void showList(); //displays list contents

list \*reverse(); //reverses the list

list \*insert();

list \*createNode();

list \*delete();

list \*find();

main()

{

list \*temp, \*start = NULL; //start will point to first node of the list

char c = 'y';

int n;

while(c != 'n' && c != 'N')

{

printf("\nEnter the data: ");

scanf("%d",&n); getchar(); fflush(stdin);

temp = createNode();

temp->data = n;

temp->next = NULL;

if(!find(start,temp->data))

start = insert(start,temp);

printf("\nDo you want to add new data in the list? (y/n): ");

scanf("%c",&c); fflush(stdin);

}

printf("\nTHE LIST IS: "); showList(start); printf("\n\n");

c = 'y';

while(c != 'n' && c != 'N')

{

printf("\nEnter the data to be deleted: ");

scanf("%d",&n); getchar(); fflush(stdin);

if(find(start, n)) start = delete(start, n);

printf("\nDo you want to delete another data from the list? (y/n):");

scanf("%c", &c) ; fflush(stdin);

}

printf("\nTHE LIST AFTER DATA DELETION IS: "); showList(start); printf("\n\n");

start = reverse(start);

printf("\nTHE REVERSED LIST IS: "); showList(start); printf("\n\n");

}

/\* Function to create a Node. Allocates memory for a new node. \*/

list \*createNode()

{

list \*new;

new = (list \*)malloc(sizeof(list));

return(new);

}

/\* Recursive function to create and insert a new node at the end of the list \*/

list \*insert(list \*st, list \*ndt)

{

if(!st) return(ndt);

st->next = insert(st->next, ndt);

return(st);

}

/\*

Function to search a data item in the list and return the node address

that matches data. In case no match found, returns NULL

\*/

list \*find(list \*st, int dt)

{

while(st)

if(st->data == dt)

return (st);

else

st = st->next;

return(st);

}

void showList(list \*temp)

{

while(temp)

{

printf("%d ", temp->data);

temp = temp->next;

}

printf("\n");

}

/\* Function to reverse the list \*/

list \*reverse(list \*l)

{

list \*nxt, \*temp;

if(!l) return(l);

else

{

nxt = l->next;

l->next = NULL;

while(nxt)

{

temp = nxt->next;

nxt->next = l;

l = nxt;

nxt = temp;

}

return(l);

}

}

/\* Recursive function for deleting a node from the list \*/

list \*delete(list \*st, int n)

{

list \*tmp;

if(!st) return(st);

if(st->data == n)

{

tmp = st;

st = st->next;

free(tmp);

return(st);

}

st->next = delete(st->next,n);

return(st);

}

Exercises

1. Identify which one of the following declaration correctly defines a self referential data structure.

a) struct class\_1 {

char name[31];

. . .

struct class\_2 \*c;

};

b) struct class\_1 {

char name[31];

. . .

struct class\_1 \*c;

};

c) struct class\_1 {

char name[31];

. . .

struct class\_1 c;

};

2. Identify the errors (if any) in the following C programs.

a) The function addList inserts a new node (nw) at the beginning of the list pointed to by the start pointer.

addList(list \*start, list \*nu)

{

if(start)

start = nu;

else

nw->nxt = start;

}

b) The function displayList (start pointer is passed as an argument) recursively displays all the nodes in the list.

displayList(list \*st)

{

if(!st) return;

printf("%d ", st->data);

displayList(st->nxt);

}

Self Referential Data Structure in C - create an ordered singly linked list

The process of creation of an ordered singly linked list involves inserting a new node maintaining an order either ascending or descending. It revolves around three basic operations, viz.

i. Insertion at the beginning of the list.

ii. Insertion somewhere in the middle, i.e. anywhere other than at the beginning or at the end.

iii. Appending at the end.

To understand how to carry out the above three types of operations on a list so as to create an ordered list, let us look at an example.

Consider that we have to write a program which will take in words from the user and build a lexically ordered list. For ease of understanding, we will assume that the user provides the following 4 words – bat, mat, cat, ant.

The list will initially begin as an empty list. We will use a start pointer to point to the first node in the list. Each node will have a pointer element that will point to the next node in lexical order. The pointer in the last node of the list will point to NULL to indicate end of the list.

When the program begins, the list is empty. So our start pointer will point to NULL.

http://www.how2lab.com/be/uploads/imglib/c/struct3.jpg

The user now feeds in the first word, viz. bat. Since bat is the first node to be inserted, this node address is assigned to start.

http://www.how2lab.com/be/uploads/imglib/c/struct4.jpg

To maintain the increasing order (lexically) the data for the next node mat is to be inserted after bat. Since there are no other nodes at present, essentially the node mat is appended at the end of the list.

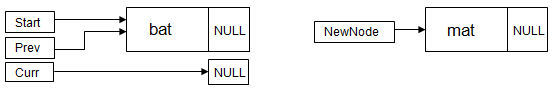
To implement such an operation we use two pointers prev and curr. Both these pointers initially point to the first node of the list (same as start). Then curr is traversed to point to the next node in the sequence. The pointer prev will always follow behind curr and will remain one position behind. When initially curr is pointing to the first node, viz. bat, we compare the node element bat with mat and find that mat is lexically bigger than bat, so we move curr to the next node, letting prev remain one node behind. Now, prev points to bat and curr points to NULL. We know that prev->next will give us a handle to the pointer element of the first node. So, to attach mat after bat, we assign the address of mat to prev->next.

Thus the operation will entail the following statements. The image following the statements illustrate the effect of the statements.

curr = start

prev = curr

curr = curr->next (curr points to NULL)

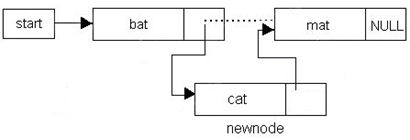


prev->next = newNode (Since newNode points to mat, mat gets attached after bat)

newNode->next = curr (since curr points to NULL, newNode->next will remain NULL)

For the next insertion, curr and prev are again reset to start. To insert a new node cat the previously mentioned steps are repeated. The comparison of new node information (cat) and current node information (bat) indicates that cat should appear after bat. Hence, curr is made to point to next node that stores mat and the process of comparing new node and current node information continues. The prev pointer follows curr, and points to the first node of the, list (bat).

The next comparison finds cat is to be placed before mat (pointed to by curr), Hence the new node information cat is to be inserted in between bat (pointed to by prev) and mat (pointed to by curr). To achieve the requisite insertion, prev->next is made to point to the new node created to accommodate cat and newNode->next is made to point to the node pointed to by curr i.e. newNode->next = curr.



The next node to be inserted contains data ant, which is to be placed before the first element's data in the list i.e. before bat. Here the comparison with the first node indicates that the new node is to be inserted before bat that means the startneeds to be redefined. This context is trapped by comparing the address of prev and curr. If both of them are stuck at start, it indicates insertion is to take place at the beginning.

Here is the full C program to implement the above. Review the program carefully and also run it to develop full understanding of the above concept.

#include <stdio.h>

struct node\_type {

char data[21];

struct node\_type \*next;

};

typedef struct node\_type list;

void showList();

list \*sortInsert();

list \*createNode();

list \*find();

main()

{

list \*newnode, \*start = NULL; //start will point to first node of the list

char c = 'y';

char word[21];

while(c != 'n' && c != 'N')

{

printf("\nEnter the word: ");

scanf("%s",word); fflush(stdin);

newnode = createNode();

strcpy(newnode->data, word);

newnode->next = NULL;

if(!find(start,newnode->data))

start = sortInsert(start,newnode);

printf("\nTHE LIST SO FAR: "); showList(start); printf("\n\n");

printf("\nDo you want to add new data in the list? (y/n): ");

scanf("%c",&c); getchar(); fflush(stdin);

}

printf("\nTHE SORTED LIST IS: "); showList(start); printf("\n\n");

}

/\* Function to create a Node. Allocates memory for a new node. \*/

list \*createNode()

{

list \*new;

new = (list \*)malloc(sizeof(list));

return(new);

}

list \*sortInsert(list \*start, list \*newnode)

{

list \*prev,\*curr;

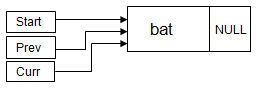
if(start==NULL)

{

//List is empty. Insert first node

return(newnode);

}



//The code below will be executed when list is not empty

curr = start;

prev = curr;

if(strcmp(newnode->data,curr->data)<0)

{

//new node < first node. Insert at the beginning.

start = newnode;

newnode->next = curr;

return(start);

}

//The code below will be executed when new node is to

//be inserted anywhere in the middle or at the end

while(curr!=NULL)

{

curr = curr->next;

if(curr==NULL)

{

//We have reached the end. Attach node to the end.

prev->next = newnode;

newnode->next = curr;

return(start);

}

else

{

if(strcmp(newnode->data,curr->data)<0)

{

prev->next = newnode;

newnode->next = curr;

return(start);

}

prev = prev->next;

}

}

return(start);

}

/\*

Function to search a data item in the list and return the node address

that matches data. In case no match found, returns NULL

\*/

list \*find(list \*st, int dt)

{

while(st)

if(strcmp(st->data,dt) == 0)

return (st);

else

st = st->next;

return(st);

}

void showList(list \*temp)

{

while(temp)

{

printf("%s ", temp->data);

temp = temp->next;

}

printf("\n");

}