LINKED LIST STRUCTURE

In sequential list representation, same kind of information are stored in a continuous memory addresses.

For example in an array data structure, sorted names are stored in the following memory representation.

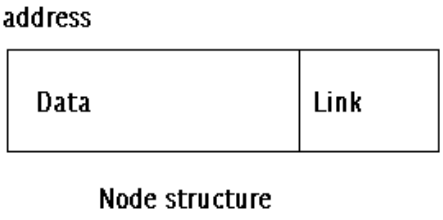
Names : Array[1.. max] of string

1	ABBAS
2	CANDAN
3	DENIZ
4	EMRAH
5	FIKRET
6	HALIL
7	KEMAL
8	MEHMET
9	MUSTAFA
10	NILGUN
11	OYA
12	REYHAN
13	TUNA
14	UMUT
15	
16	
..	
..	
max	

If we **insert** a new name in to this sorted array representation, we have to first determine the location and then do the necessary swapping for the other elements and then we have to insert the new name into the correct position. We have the same problem if we **delete** any given name also.

An elegant solution to this problem of data movement in sequential representation is achieved by using **linked** representation. Unlike a sequential representation where successive items of a list are located a fixed distance apart, in a linked representation these items may be placed anywhere in memory. To access elements in sequential list(arrays) representation we use subscript variables. Where else in linked representation we need a pointer value which point the address of next

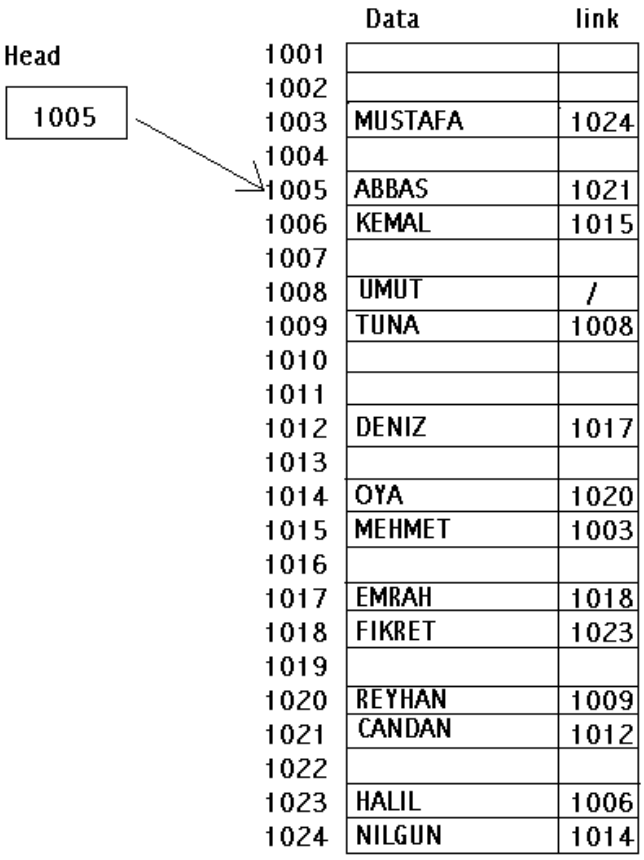
element. That's why a typical node structure of a linked list data structure is as follows.



Node structure

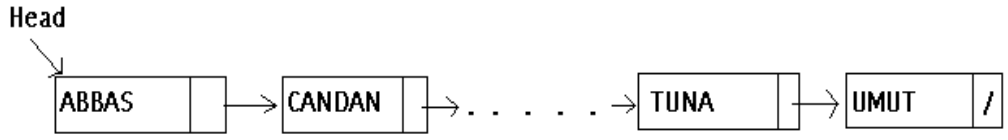
Thus, associated with each data item in a linked list representation is a pointer to the next item. This pointer is often referred to as a link. In general, a node is a collection of data, $Data_1,\ldots,Data_n$ and links $Link_1,..,Link_m$. Each item in a node is called a field. A field contains either a data item or a link.

Using the same example we can represent data structure using linked list.



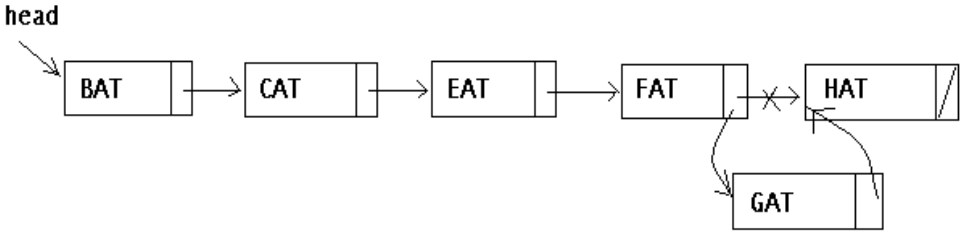
Sorted names in linked list structure.

Starting from the head node we can obtain sorted name using linked fields. It is customary to draw linked list as an ordered sequence of nodes with links being represented by arrows as in the following figure.

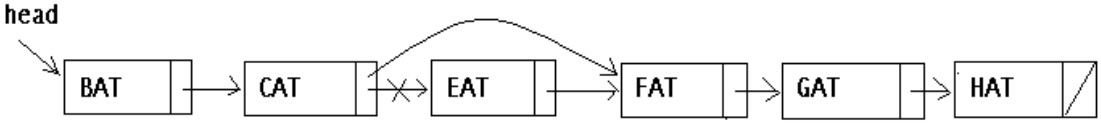


Usual way to Draw a Linked list

It is easier to make insertions and deletions using a linked list rather than a sequential list like array.



Insertion of GAT into linked list



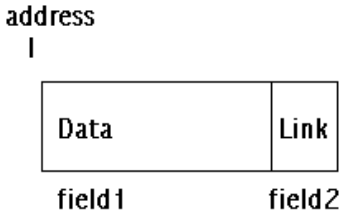
Deletion of EAT from the linked list

For the time being let us assume that all free nodes are kept in a 'black box' called the storage pool. x=g**etnode()** and freenode**(x)** are the sub procedures to get a new node(x=getnode()) at the address of **x** from the storage pool and returns node(freenode(x)) at the address x to the storage pool.

Following algorithm creates two nodes with a linked list structure.

```
 create2node()
{
 i=getnode()
 t=getnode()
 data(t) = 'AHMET'
 data(i) = 'MEHMET'
 link(t) = i
 link(i) = t
}
```

In this algorithm node structure at the address of i is following.

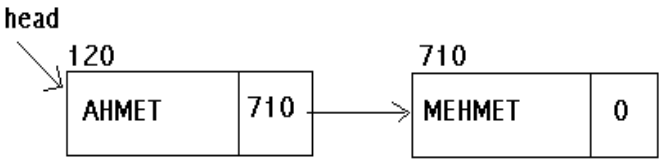


After calling **getnode(),** notation of the algorithm describes the following.

data(i) represents : Data field at the address of i.

link(i) represents : Link field at the address of i.

Let us assume in first call of **getnode()** we had address of 120 which is the value of i, and in second call of **getnode()** we get 710 for the value of i.



head defines head node of simple linked list structure.

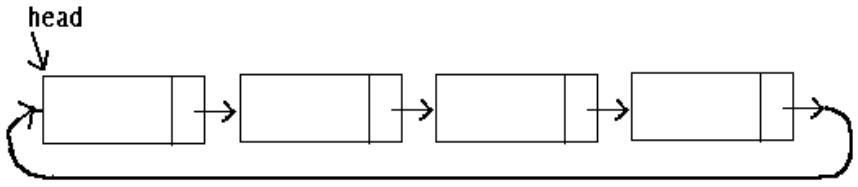
So the following algorithm is more general form of creation of link list structure using input of numbers.

```
read x
i=getnode()
data(i) = x
save = i
head = i
count  =1
while count  <  4
{ read x
  i=getnode()
  link(save) = i
  data(i) = x
  save = i
  count  ++
}
link(save) ← null
/*This part List the link list structure */
save = head
while save != null
{
        print data(save)
        save = link(save)
}
```

Types Of link list Structures :

Circular linked list structure.

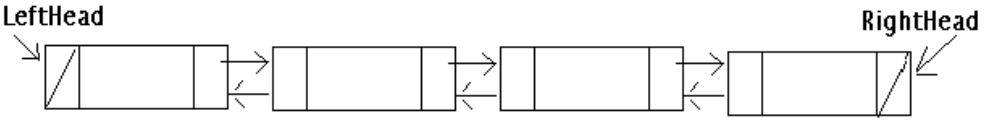
In this structure link field of the last node points the head address of the list and we obtain a circular linked list structure.



Circular linked list structure

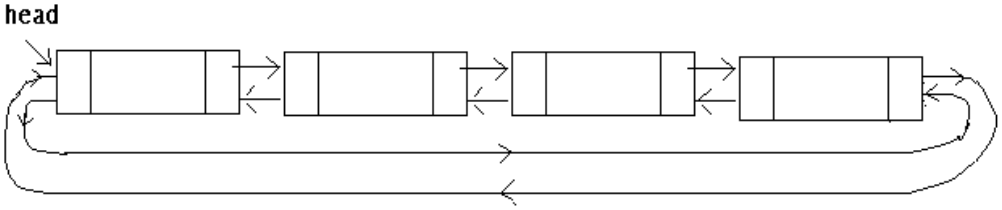
Doubly Linked list Structure.

In this kind of linked list structure each node has two link(pointer) fields, Right pointer and left pointer. We mostly use this kind of structure for ordered(sorted) list structure. So it becomes so easy to obtain ascending and descending order of information. In this structure there are two head nodes. Which are the beginning points of Left head and Right head.



Doubly linked list structure

Doubly circular linked list structure : Each node has two links and last nodes are connected to the first node . There is only one head node. We can obtain ordered descending or ascending linked list structure using this kind of data structures.



Doubly circular linked list structure

Operations On linked list Structures :

- Following algorithm deletes the node of a given address x from the simple linked list structure.

```
delete(x)
if  head = Null then
  print 'Emty-list'
  return
else if x = head then
  head = link(head)
  free(x)
Else
   next = head
        do{
            prev =next
            next = link(next)
            If next = Null Then {
                print 'Node not Found'
                 return
              }
            }while  next  != x
  link(prev) = link(next)
  free(x)
end delete
```

- **Get an input of num and Create a SORTED simple linked list structure.**

```
insort(num)
   new=getnode()
   info(new) = num
   If head = Null then{
      link(new) = Null
      head = new
      return
   }
   If info(head) >= info(new) then {
      link(new) = head
      head = new
      return
   }
   save = head
   while link(save) ≠ Null & info(link(save)) <= info(new)
            save = link(save)

   link(new) = link(save)
   link(save) = new
end insort
```

Let **insafter(p,x)** denote the operation of inserting an item x into a list after a node pointed to by **p**.

```
insafter(p,x)
{
 q = getnode();
 info(q) = x;
 link(q) = link (p);
 link (p) = q;
}
```

Let **delafter(p,x)** denote the operation of deleting the node following node(p) and assigning its contents to the variable **x**.

```
delafter(p)
{
 q = link(p);
 x = info(q);
 link(p) = link(q);
 freenode(q);
}
```

Add a node at the beginning of the linked list

First, we need to create a new node. We will need to create a new node each time we want to insert a new node into the list so we can develop a function that creates a new node and return it.

```
create(head, value)
{ new_node = getnode();
   if(new_node == NULL)
     {  printf("Error creating a new node.\n");
        exit(0);
     }
new_node(data) = value;
  if(head== NULL){
    head= new_node
    link( head) = NULL;
  }else
  {
    link( new_node) = head;
  }
}
```

Programming Details

Linked List Using Dynamic variables of C struct data type can be used to represent dynamic linked list storage representation of C. A linked list consists of a set of nodes, each of which has two fields: an information field and a pointer to the next(link) node in the list. In addition, an external pointer points to the first node in the list which is head node.

```
struct  node {
        int  info;
        struct  node *link;
};
typedef struct node *NODEPTR;
```

Only One Node

Using malloc function we can allocate a memory for a node structure at the address p.

```
p = (NODEPTR)  malloc(sizeof(struct node));
```

Using above declarations getnode() function can be written as follows.

struct node *

```
NODEPTR getnode()
{
 NODEPTR q;
 q = (NODEPTR) malloc(sizeof(struct node));
 return(q);
 }
```

Execution of the statement

```
freenode(p);
```

should return the node whose address is at p to available storage. We present the routine *freenode*;

```
void freenode(NODEPTR p)
{
free(p);
}
```

Since the routines *getnode* and *freenode* are so simple under this implementation, they are often replaced by the in-line statement**.**

```
p=(NODEPTR) malloc(sizeof (struct node));
```

and

```
free(p);
```

The procedures *insafter(p,x)* and *delafter(p,px)* are presented below using dynamic implementation of a linked list.

This function inserts *x* into the linked list after the address *p*.

```
void insafter(NODEPTR p, int x)
{  NODEPTR q;
if (p==NULL) {
   printf("Void insertion \n");
   exit(1);
}
    q=getnode();
    q->info=x;
    q->next=p->next;
    p->next=q;
}
```

This function deletes a node after the address *p*

```
void delafter(NODEPTR p, int x)
{  NODEPTR q;
if ((p==NULL) || (p->next == NULL))
 { printf("Void deletion \n");
    exit(1);
}
    q = p->next;
    x = q->info;//can be used to print the value of the deleted node
    p->next = q->next;
    freenode(q);
}
```

Examples of List Operations in C

Operation place insert an element *x* into *sorted* linear linked list structure. Where *head* points to a sorted list and *x* is an element to be inserted into its proper position within the list.

```
void place(NODEPTR *head , int x)
{
 NODEPTR p, q;
 q= NULL;
 for(p=*head; p!=NULL && x>p->info; p=p->next)
   q = p;
 if (q == NULL) /* insert x at the head of the list */
                { p=getnode()
                  p->info=x;
                  p->next = head;
                  head=p;
                }
         Else{
                  p=getnode();
                  p->info=x;
                  p->next=q->next;
                  q->next=p;
              }
}
```

As a second example, we write a function *insend(head , x)* to insert the element *x* at the end of a linked list *head*.

```
void insend(NODEPTR *head, int x)
{ NODEPTR p , q;
   p=getnode();
   p->info = x;
   p->next = NULL;
   if (*head ==NULL)
      *head = p;
   else {
      for (q=*head; q->next !=NULL; q=q->next) ;

       q->next = p;
     }
}
```

Linked list creation program.(inserting *x* at the *head* of the linked list structure). Creates 5 nodes of linked list.

```
#include<stdio.h>
#include<stdlib.h>
#define NULL 0
struct node{
        int info;
        struct node *link;
        };
typedef struct node *NODEPTR;
NODEPTR getnode();
int main()
{
NODEPTR p, head , save ;
 int i , number;
/* inserts first node */
 printf("%s\n","Enter Info number");
 scanf("%d",&number);
 p =getnode();
 p->info=number;
 head=p;
/* inserts the other 4 nodes */
save=head;
 for (i=0;i<4;++i)
   {printf("%s\n","Enter Info number");
    scanf("%d",&number);
    p=getnode();
    p->info=number;
    save->link=p;
    save=p;
   }
 p->link=NULL;
 save=head;
 do {
  printf("%d \n",save->info);
  save=save->link;
```

```
  } while (save!=NULL);
return 0;
}


NODEPTR getnode()
{ NODEPTR q;
 q = (NODEPTR) malloc(sizeof(struct node));
 return(q);
 }
```

We now present a function *search(list , x)* that returns a pointer to the first occurrences of *x* within the linked list *list* and the NULL pointer if *x* does not occur in the linked list.

```
NODEPTR  search(NODEPTR list , int x)
{ NODEPTR p;
 for(p=list ; p!=NULL; p=p-> link)
   if (p->info ==x)  return (p);
return (NULL);
}
```

Removex() routine deletes all nodes whose info field contains the value **x**.

```
void removex(NODEPTR *head , int x)
{ NODEPTR p , q , save;
 int y;
q=NULL;
p=*head;
while (p!=NULL)
  if (p->info == x)
   {save=p;
     p= p->next;
      if (q==NULL)   {
      freenode(*head);
     *head = p;
    }
   else {freenode(save);
    q->next=p;}
 }else
{ q = p;
  p = p->next;
}
}
```

Self-Study

Linked list insertion program without using typedef declaration

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
#define NULL 0
struct node{
  int info;
  struct node *next;
};
struct node  *getnode(void);
void insert(struct node **plist,int);

void main(void)
{
   struct node *list ;
    list=NULL;

     insert(&list,99);
  insert(&list,88);
  do{
        printf("%d\n",list->info);
        list=list->next;
  }while (list !=NULL);
}
void insert(struct node **plist,int x)
{
  struct node *p;
  p=getnode();
  p->info=x;
  p->next=*plist;
  *plist=p;
}

struct node *getnode(void)
{
 struct node *q;
 q=(struct node *) malloc(sizeof(struct node));
 return q;
}
```

REVERSE LINK PROGRAM

Traverse the linked list in both direction

```
#include<stdlib.h>
#define NULL 0
struct node{
        int info;
        struct node *link;
        };
typedef struct node *NODEPTR;

NODEPTR getnode(void);

/* This program replace link fields in reverse form }
    and traverse the list in both direction  */

int main()
{NODEPTR p, head , save ;
 int i , number;
/* assume that linked list is created and head points the beginning
   address of the linked list */
save=head;
temp =save->link;
printf("%d \n", save->info);
save->link=NULL;
pred = save;
do
{ save = temp->link;
  temp->link = pred;
  printf("%d \n", temp->info);
  pred = temp;
  temp = save;
}while  (temp != NULL);
 head = pred;
 save = pred;
 while (save != NULL     )
  { printf("%d \n", save->info);
    save=save->link;
 }
```