Linked Lists

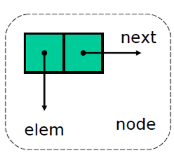
Basic Concepts

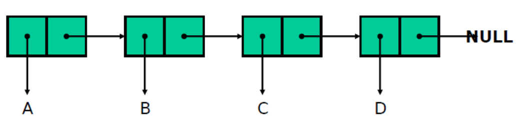
- A list refers to a set of items organized sequentially.
- An array is an example of a list.
- Problems with array:
  - The array size has to be specified at the beginning.
  - Deleting an element or inserting an element may require shifting of elements.

Linked Lists

Each structure of the list is called a *node*, *and* consists of two fields:

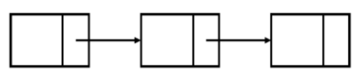
- **Element**
- **link to the next node**





Self Referential Structures

- **A structure referencing itself – how?**



So, we need a pointer inside a structure that points to a structure of the same type.

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

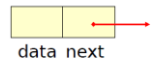
Self-referential structures

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

- The pointer variable next is called a link.
- Each structure is linked to a succeeding structure by next.

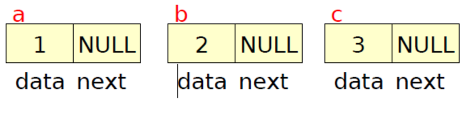
Pictorial representation

- A structure of type struct list



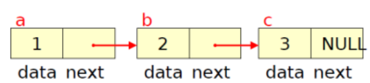
- The pointer variable next contains either
  - an address of the location in memory of the successor list element
  - or the special value NULL defined as 0.
- NULL is used to denote the end of the list.

```
struct list a, b, c;
a.data = 1;
b.data = 2;
c.data = 3;
a.next = b.next = c.next = NULL;
```



Chaining these together

- a.next = &b;
- b.next = &c;

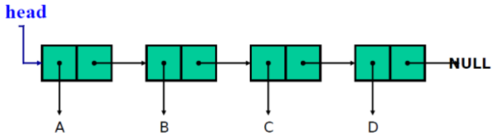


What are the values of :

- a.next->data
- a.next->next->data

Linear Linked Lists

- A singly linked list is a data structure consisting of a sequence of nodes.
- A head pointer addresses the first element of the list.
- Each element points at a successor element.
- The last element has a link value NULL.



Linked Lists

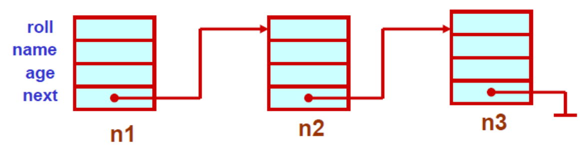
In general, a node may be represented as follows

```
struct node_name
{
   type member1;
   type member2;
   .........
   struct node_name *next;
};
```

Example

```
struct stud
{
   int roll;
   char name[30];
   int age;
   struct stud *next;
};
struct stud n1, n2, n3;
```

- n1.next = &n2 ;
- n2.next = &n3 ;
- n3.next = NULL



Example

```
#include <stdio.h>
struct stud
{ int roll;
   char name[30];
   int age;
   struct stud *next;
}
main() {
struct stud n1, n2, n3;
struct stud *p;
```

```
scanf ("%d %s %d", &n1.roll, n1.name, &n1.age);
scanf ("%d %s %d", &n2.roll, n2.name, &n2.age);
scanf ("%d %s %d", &n3.roll, n3.name, &n3.age);
n1.next = &n2 ;
n2.next = &n3 ;
n3.next = NULL ;
```

```
/*Print the elements */
p = &n1 ; /* point to 1st element */
while (p != NULL)
  {
        printf ("\n %d %s %d",p->roll, p->name, p->age);
        p = p->next;
  }
}
```

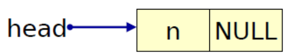
Dynamic memory allocation

```
#include <stdio.h>
struct list {
char d;
struct list * next;
};

typedef struct list ELEMENT;
typedef ELEMENT * LINK;
```

Dynamic memory allocation

```
main() {
LINK head ; // struct list * head
head = (LINK )malloc(sizeof(ELEMENT));
//head=(ELEMENT *) malloc (sizeof (ELEMENT));
//head=(struct list *) malloc (sizeof (struct list));
head->d = 'n';
head->next = NULL;
```

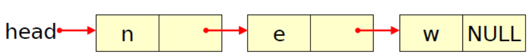


Dynamic memory allocation

```
head->next = (LINK )malloc (sizeof(ELEMENT));
head->next->d = ‘e’;
head->next->next = NULL;
```

A second element is added.


```
head->next->next =(LINK) malloc (sizeof(ELEMENT));
head->next->next->d = ‘w’;
head->next->next->next = NULL;
```




```
while (head != NULL)
  {
       printf("\n %c ",head->d);
       head = head->next;
  }
}
```

List Operations

1. How to initialize a self referential structure (LIST),
2. How to insert a structure into the LIST,
3. How to delete elements from it,
4. How to search for an element in it,
5. How to print it,
6. How to free the space occupied by the LIST.

Inserting at the Head

1. Allocate a new node
2. Insert new element
3. Make new node point to old head
4. Update head to point to new node

Removing at the Head

1. Update head to point to next node in the list
2. Allow garbage collector to reclaim the former first node

Inserting at the Tail

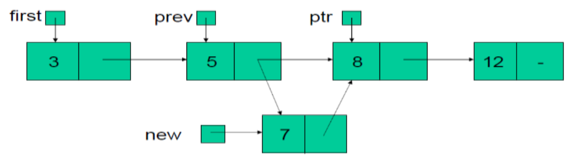
1. Allocate a new node
2. Insert new element
3. Have new node point to null
4. Have old last node point to new node
5. Update tail to point to new node

Insertion

To insert a data item into an ordered linked list involves:

- **creating a new node containing the data,**
- **finding the correct place in the list, and**
- **linking in the new node at this place.**

Example of an Insertion



- Create new **node for the 7**
- Find **correct place – when ptr finds the 8 (7 < 8)**
- Link **in new node with previous (even if last) and ptr nodes**
- **Also check insertion before first node.**

```
#include <stdio.h>
#include <stdlib.h>
struct list {
   int  data;
   struct list * next;
   };
typedef struct list ELEMENT;
typedef ELEMENT * LINK;
```

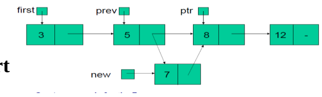
Create_node function

```
LINK create_node(int dat)
{
  LINK new;
  new = (ELEMENT *) malloc (sizeof (ELEMENT));
  new -> data = dat;
  return (new);
}
```

insert function

LINK insert (int data, LINK ptr) //LINK➔ELEMENT*

```
{
LINK new, prev, first;
new = create_node(data);
if (ptr == NULL || data < (ptr -> data))
{ // insert as new first node
      new -> next = ptr;
      return new;
// return pointer to first node
}
```



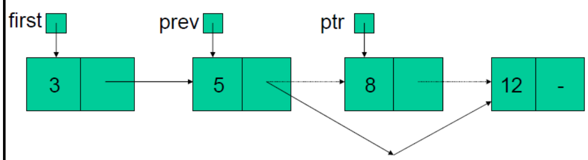

```
else // not first one
  {
    first = ptr; // remember start
    prev = ptr;
    ptr = ptr -> next; // second
    while (ptr != NULL && data > (ptr -> data))
       { // move along
       prev = ptr;
       ptr = ptr -> next;
       }
    prev -> next = new; // link in
    new -> next = ptr; //new node
    return first;
  } // end else
} // end insert
```

Deletion

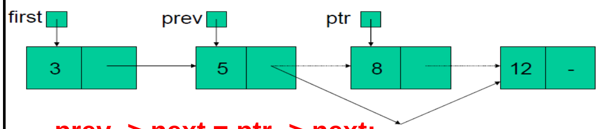
To delete a data item from a linked list involves (assuming it occurs only once):

- finding the data item in the list, and
- linking out this node, and
- freeing up this node as free space.

Example of Deletion (8)

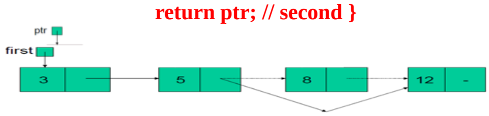


- When ptr finds the item to be deleted, **e.g. 8**, we need the **previous node** to make the **link to the next one after ptr (i.e. ptr -> next).**



prev -> next = ptr -> next;

```
// delete the item
LINK delete_item(int data, LINK ptr) {
   LINK prev, first;
   first = ptr; // remember start
   if (ptr = = NULL)
         return NULL;
   else if (data == ptr -> data) // first node
                 {       ptr = ptr -> next; // second node
                         free(first); // free up node
                         return ptr; // second }
```




```
         else // check rest of list
         {        prev = ptr;
                  ptr = ptr -> next;
                  // find node to delete
                  while (ptr != NULL && data != ptr->data)
                           {        prev = ptr;
                                    ptr = ptr -> next;
                           }
                  if (ptr == NULL || data != ptr->data)
                           return first; // original
                  prev -> next = ptr -> next;
                  free(ptr); // free node
                  return first; // original
         }
} // end delete
```

Initialization

```
typedef struct list {
    int data;
    struct list *next;
    } ELEMENT;

ELEMENT* Initialize (int element)
{
    ELEMENT *head;
    head = (ELEMENT *)calloc(1,sizeof(ELEMENT)); /*Create initial node*/
    head->data = element; head -> next = NULL;
    return head;
}
```

Searching a data element

```
int Search (ELEMENT *head, int element) {
int i;
ELEMENT *temp;
i = 0;
temp = head -> next;
while (temp != NULL)
   {if (temp -> data == element)
   return TRUE;
   temp = temp -> next;
   i++;}
return FALSE;
}
```

Printing the list

```
void Print (ELEMENT *head)
{
   ELEMENT *temp;
   temp = head -> next;
   while (temp != NULL)
     {printf("%d->", temp -> data);
     temp = temp -> next;}
}
```

```
ELEMENT* Insert(ELEMENT *head, int element, int position) {
int i=0;
ELEMENT *temp, *new;
if (position < 0)
   { printf("\nInvalid index %d\n", position);
   return head; }
temp = head;
for(i=0;i<position;i++)
   {temp=temp->next;
   if(temp==NULL)
      {printf("\nInvalid index %d\n", position);
      return head;}
   }
new = (ELEMENT *)calloc(1,sizeof(ELEMENT));
new ->data = element;
new -> next = temp -> next;
temp -> next = new;
return head;}
```

```
ELEMENT* Delete(ELEMENT *head, int position)
  {int i=0; ELEMENT *temp,*hold;
  if (position < 0)
    {printf("\nInvalid index %d\n", position);
    return head;}
  temp = head;
  while ((i < position) && (temp -> next != NULL))
  {temp = temp -> next; i++;}
  if (temp -> next == NULL)
    {printf("\nInvalid index %d\n", position);
    return head;}
  hold = temp -> next;
  temp -> next = temp -> next -> next;
  free(hold);
  return head;
}
```

Print the list backwards

- **How can we print when the links are in forward direction ?**
- **Can we apply recursion?**

```
void PrintArray(ELEMENT *head) {
  if(head -> next == NULL)
    {/*boundary condition to stop recursion*/
    printf(" %d->",head -> data);
    return;}
  PrintArray(head -> next); /* calling function recursively*/
  printf(" %d ->",head -> data);  /* Printing current element*/
  return;
  }
```

Count number of elements in a list recursively

```
int count (ELEMENT *head)
{
  if (head == NULL)
  return 0;
  return 1+count(head->next);
}
```

Count a list iteratively

```
int count (ELEMENT *head)
 {
  int cnt = 0;
  for ( ; head != NULL; head=head->next)
      ++cnt;
  return cnt;
  }
```

Print a List

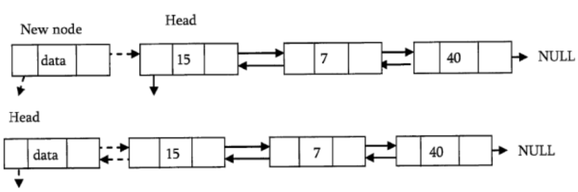
```
void PrintList (ELEMENT *head)
{
   if (head == NULL)
     printf (“NULL”) ;
   else
   {printf (“%d --> “, head->data) ;
     PrintList (head->next);
   }
}
```

Concatenate two Lists

```
void concatenate (ELEMENT *ahead, ELEMENT *bhead)
   { if (ahead->next == NULL)
        ahead->next = bhead ;
   else
        concatenate (ahead->next, bhead);
   }
```

Doubly Linked Lists

```
struct DDNode{
int data;
struct DDNode *next;
struct DDNode * previous;
}
```



Circular Linked Lists

struct CLLNode{
int data;
Struct CLLNode *next;
}