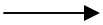


Linked Lists – I

Lists

Lists are ordered collections of objects. This does not mean that lists are sorted, but rather that as items are added, one can predict their relative positions. For example, one can define an operator that adds items at the end of the list, or at the front. Given the list 20,30,50, the operation *append 69* will result in 20,30,50,69, while *addfront 12* to the new list would result in 12,20,30,50,69. As you observe this particular list is already sorted. Now to insert element 42 in its proper position in the list, we may like to define a new operation *insert 42*, which would result in the list 12,20,30,42,50,69. If we are searching for a specific item, and we find it in the list, we return 1. The operation *search 30* would return 1, as the element is present in the list. We can also remove items from the list through a delete operation such as *delete 20*, which would create an empty position (a hole) in the list, and all elements after 20 would have to be shifted one position forward to result in the final list 12,30,42,50,69. The effect of *insert 25* would require shifting all elements after 12 one position back to accommodate the new element 25.

You know how easy it is to implement the lists using arrays, where we could do searching or sorting operations. If the list were sorted, the searching operation would take O(log n) operations. What about insert and delete operations? To insert a new element in the beginning of the array, you have to shift all the elements down by one position to accommodate it. Similarly, to delete an item, you have to shift up all the elements to fill in the hole created by the item. For a list with n elements, these operations could be more expensive computations of the order O(n).

Another shortcoming with arrays is that they have to have a fixed size, which must be declared in advance before the program starts executing. In many applications, the number of elements in the list may vary greatly, depending on the input, so memory requirements may change during execution of the program.

Recursive data structures

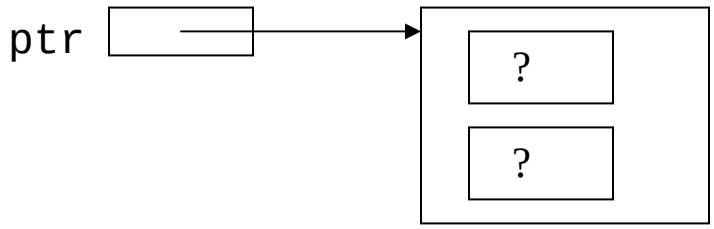
What is desired is a way to allocate memory for new items ONLY WHEN necessary, and to free that memory space when it is no longer needed. Furthermore, this newly allocated memory must be logically combined into a single entity (representing the list).
What is a list? We can think of a list as an entity containing a similar entity. A list is a data item followed by another list. The list 12,20,30,42,50,69 can be thought of as being composed of the data item 12, followed by another list 20,30,42,50,69. Thus we can define a list structure in terms of itself. Data types that are defined in this way are called recursive data types.
Consider the structure

```
struct list{
     int data;
     struct list *next;
};
```
This structure defines a list as containing an item of type int, followed by another list. The second list is connected through the link *next*. More appropriately, this list is referred to as a **linked list**. Each element is linked to the next one through a link field, which is a pointer to the address of the next element.

But the definition alone is not sufficient to store the values of the elements of the list. You have to request memory to allocate space to hold each item. As the structure is dynamic (it can grow or shrink), more memory can be requested dynamically. Here is an example which allocates memory for a particular node to hold the value of an integer, and also the pointer to the next node.

Memory allocation for a node:

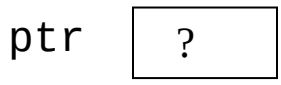
```
struct node{
     int data;
     struct node *next;
};

struct node *ptr;

ptr = (struct node *) malloc(sizeof(struct node));
```



Once the data has been processed, and the program does not require the data, the memory occupied by the node can be returned to main memory .

- Function ***free*** de-allocates memory- i.e. the memory is returned to the system so that the memory can be reallocated in the future.
  e.g.

```
free(ptr);
```



Linked Lists

- It is an important data structure.
- An abstraction of a list: i.e. a sequence of nodes in which each node is linked to the node following it.
- Lists of data can be stored in arrays, but linked lists provide several advantages:

Arrays

– In an array each node (element) follows the previous one physically (i.e. contiguous spaces in the memory)

– Arrays are fixed size: either too big (unused space) or not big enough (overflow problem)

– Maximum size of the array must be predicted which is sometimes not possible.

– Inserting and deleting elements into an array is difficult. Have to do lot of data movement, if in array of size 100, an element is to be inserted after the 10^{th} element, then all remaining 90 have to be shifted down by one position.

Linked Lists

– Linked lists are appropriate when the number of data elements to be represented in the data structure are not known in advance.

– Linked lists are dynamic, so the length of a list can increase or decrease as necessary.

– A linked list is a collection of nodes, each node containing a data element.

– Each node does not necessarily follow the previous one physically in the memory. Nodes are scattered at random in memory.

– Insertion and Deletion can be made in Linked lists , by just changing links of a few nodes, without disturbing the rest of the list. This is the greatest advantage.

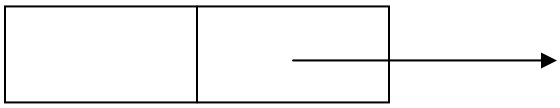
– But getting to a particular node may take large number of operations, as we do not know the address of any individual node .

– Every node from start needs to be traversed to reach the particular node.

A simple Node Structure

A node in a linked list is a structure that has at least two fields. One of the fields is a data field; the other is a pointer that contains the address of the next node in the sequence.

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

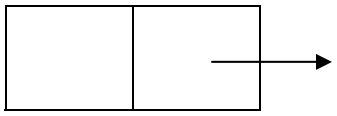


data **next**

The pointer variable *next* is called a ***link***. Each structure is linked to a succeeding structure by way of the field *next*. The pointer variable *next* contains an address of either the location in memory of the successor `struct list` element or the special value **NULL**.

More examples of Nodes

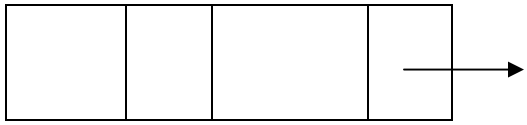
A node with one data field:



number link

```
struct node{
    int number;
    struct node * link;
};
```

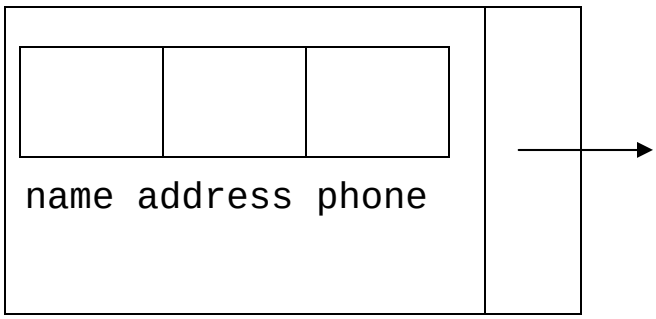
A node with 3 data fields:



name id grdPts next_student

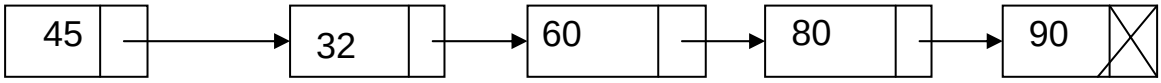
```
 struct student{
     char name[20];
     int id;
     double grdPts;
     struct student
          *next_student;
 };
```

A structure in a node:




```
struct person{
    char name[20];
    char address[30];
    char phone[10];
};

struct person_node{
    struct person data;
    struct person_node
           *next;
};
```

A simple Linked List



pHead

- The head pointer addresses the first node of the list, and each node points at its successor node.
- The last node has a link value NULL.

Empty List

No data elements, no nodes. So Head points to NULL.

Empty Linked list is a single pointer having the value of NULL.

```
pHead = NULL;
```

Basic Linked List Operations

1. Add a node
2. Delete a node
3. Looking up a node
4. List Traversal (e.g. Counting nodes)

Add a Node

There are four steps to add a node to a linked list:

1. Allocate memory for the new node.
2. Determine the insertion point (you need to know only the new node's predecessor (`pPre`)
3. Point the new node to its successor.
4. Point the predecessor to the new node.

Adding Two nodes to an Empty List

We are interested in forming a linked list which contains two integers 39 and 60. Let us first define a structure to hold two pieces of information in each node- an integer value, and the address corresponding to the next structure of same type.

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

struct node *pNew, *pHead;
```

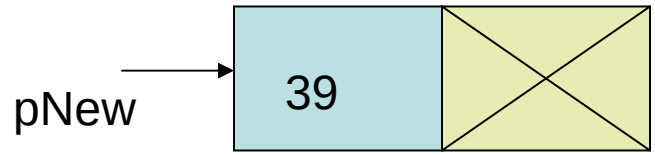
Use malloc to fetch one node from the memory, and let *pNew* point to this node.

```
pNew = (struct node *) malloc(sizeof(struct node));
pHead = NULL;
```

Now to store 39 in the data part of the node, we can use
`(*pNew).data = 39;`

A more convenient way is to use the notation

```
pNew->data = 39;
pNew->next = NULL;
```



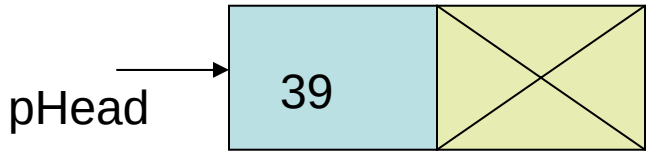
At this moment there are no elements in the list. (why?)

Pointer pHead points to NULL.

First element 39 is stored in node pNew. So make it the head node.

```
pNew->next = pHead;
/* set link to NULL*/
```

```
pHead = pNew;
     /* point list to first node*/
```

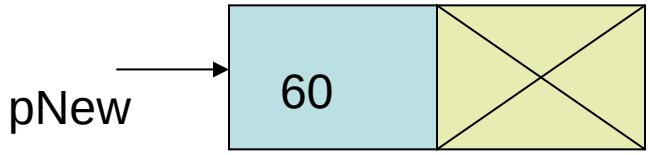


Now we have got one node in the list, which is being pointed to by *pHead*. To put the next value 60 on the list, we must use malloc to fetch another node from the memory.

```
pNew = (struct node *) malloc(sizeof(struct node));
```

Let us now assign the data value to this node:

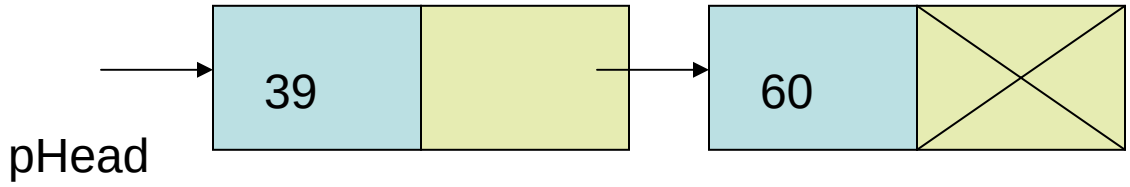
```
pNew->data = 60;
pNew->next = NULL;
```



Now we need to link this new node to the *pHead* node, which can be done by following statement:

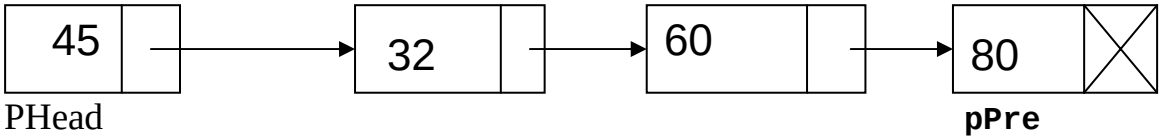
```
pHead->next = pNew;
```

This gives us the list of two nodes:



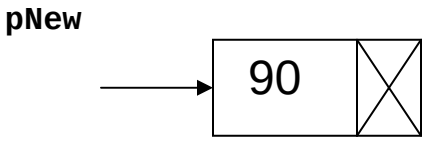
Add a node at the end of the list

Given the list *pHead*



let us say we are interested in adding a node containing 90 at the end of this list. As a first step, use malloc to get a new node *pNew* from the memory and load 90 in the data field and NULL in the next field.

```
pNew->data = 90;
pNew->next = NULL;
```



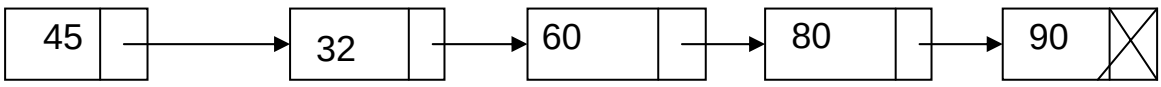
Now, the last node containing 80 would become the predecessor of the node containing 90 after the node is added. Let us call this node as *pPre*. But how do we get to the last node? Use a temporary variable *pPre* and initialize it to *pHead.* Now hop through the nodes till you get a node whose *next* link is NULL. Then *pPre* will be pointing to the last node.

```
pPre = pHead;
while ( pPre->next != NULL )
     pPre = pPre->next;
```

Now link *pPre* with *pNew* .

```
pPre->next = pNew;
```

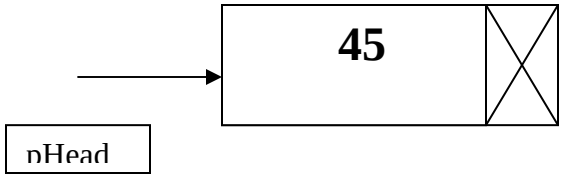
This would result in the node ***pNew*** to be attached to the list as the last node.



PHead

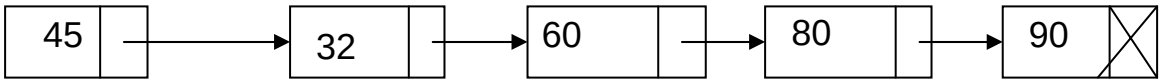
Add a Node at the beginning of a list

Suppose we want to add node containing data 45 in front of a list 32,60,80,90. The head node *pHead* points to node containing first element 32. Use malloc to get a new node *pNew* and store 45 in the data field and NULL in the next field.



To link the old pHead to the new node *pNew*, store address pHead in next field of *pNew*. Since it is the new head pointer, set *pHead* to *pNew*.

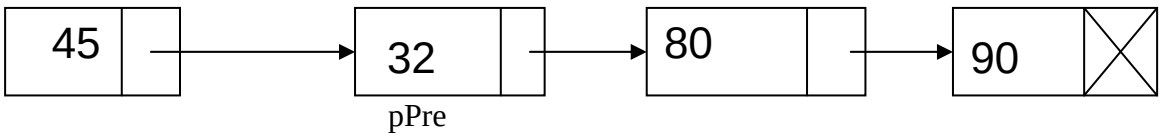
```
pNew->next = pHead;
pHead = pNew;
```



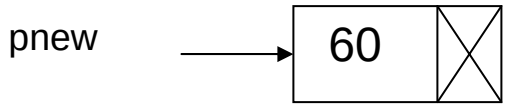
pHead

Insert a node in the middle (after a specific node)

Let us now insert an element 60 in a sorted list 45,32,80,90. After insertion, 32 is going to be the predecessor of 60, so let us call that node as pPre.



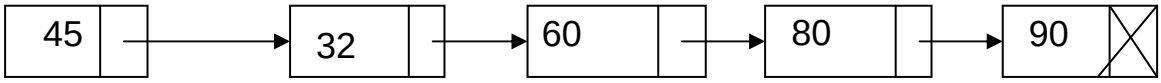
Use malloc to get the new node pNew and store 60 in its data field.



Currently, 32 is pointing to 80. After 60 is inserted, it should point to 80. So link of 32 must be transferred to link of 60. Also, 32 and 60 must be linked. So the next field of pPre must contain the pointer to 60.

```
pNew->next = pPre->next;
pPre->next = pNew;
```

This results in



Inserting a node in a Linked List (General case)

So far we have been discussing separate codes for inserting a node in the middle, at the end or at front of a list. We can combine all the codes and write a general code for inserting a node anywhere in the list. Given the head pointer (`pHead`), the predecessor (`pPre`) and the data to be inserted (`item`), we first allocate memory for the new node (`pNew`) and adjust the link pointers.

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

struct node *pNew, *pHead;

pNew = (struct node *)
       malloc(sizeof(struct node));
pHead = NULL;



pNew->data = item;
pNew->next= NULL;

 if (pPre == NULL){
  /*Adding before first node or to empty list*/

        pNew->next = pHead;
        pHead = pNew;
   }
   else {
    /* Adding in middle or at end*/


       pNew->next = pPre->next;
       pPre->next = pNew;
   }
```

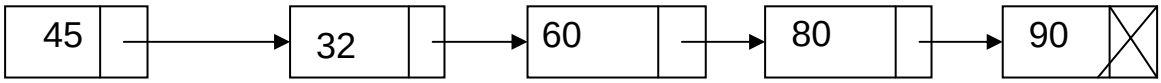
Search for an item in the list

/* Given the item and the pointer to the head of the list, the function returns a pointer to the node which matches the item, or returns NULL if the item is not found
*/

```
struct node * lookup(int item, struct node *pHead)
{
     if (head == NULL)
          return NULL;
     else if (item == pHead->data)
          return pHead;
     else
 return(lookup(item, pHead->next));
}
```

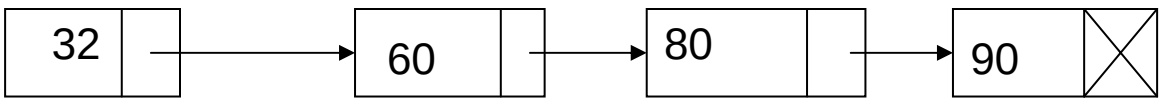
Get Head and Tail of a List

Suppose we want to divide a list in two parts
- Head, containing the first node, and
- Tail, containing all the remaining nodes

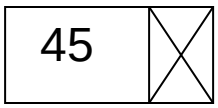


pHead gives the first node, and its next field contains the pointer to the remaining nodes. Thus it is a very easy task, and can be carried out through the following code segment:

head = pHead;
tail = pHead -> next;



tail



head

Counting the nodes in a List

- ***Recursive version:***

```
int count (struct node * pHead)
{
     if (pHead==NULL)
          return 0;
     else
     return(1 + count(pHead->next));
}
```

- ***Iterative version:***

```
int count(struct node *pHead)
{
     struct node * p;
     int c = 0;

     p = pHead;
     while (p != NULL){
          c = c + 1;
          p = p->next;
     }
     return c;
}
```

Creating a Linked list from an array

- **Recursive version:**

```
// Copies the contents of an array into a  dynamically growing
list.

struct node *alist(int a[],int j, int n)
{
        struct node *pHead;

        if (j >= n)  //base case
             return NULL;
        else {
             pHead = (struct node *)
            malloc(sizeof(struct node));
             pHead->data = a[j];
             pHead->next = alist(a, j+1, n);
             return pHead;
        }
}
```

Calling the function:

```
int array[] = {1, 2, 3, 4};
struct node *my_list;

my_list = alist(array, 0, 4);
```

- **Iterative version:**

```
struct node *alist(int a[], int n)
{
     struct node *pHead, *current;
     int j;

     if (n == 0)

//the array is empty
          return NULL;

     else {
```

```
      //create the head node for
    //the first element

          pHead = (struct node *)
              malloc(sizeof(struct node));
          current = pHead;
          current->data = a[0];


      //create nodes for the other elements
          j = 1;

          while (j < n){

               current->next = (struct node *)
                    malloc(sizeof(struct node));

               current = current->next;
               current->data = a[j];

               j = j +1;
          }
          current->next = NULL;

//finish the list

          return head;
     }
}
```

Calling the function:

```
my_list = alist(numbers, 4);
```