Linked Lists

Static and Dynamic Variables

- **Static Variables**:

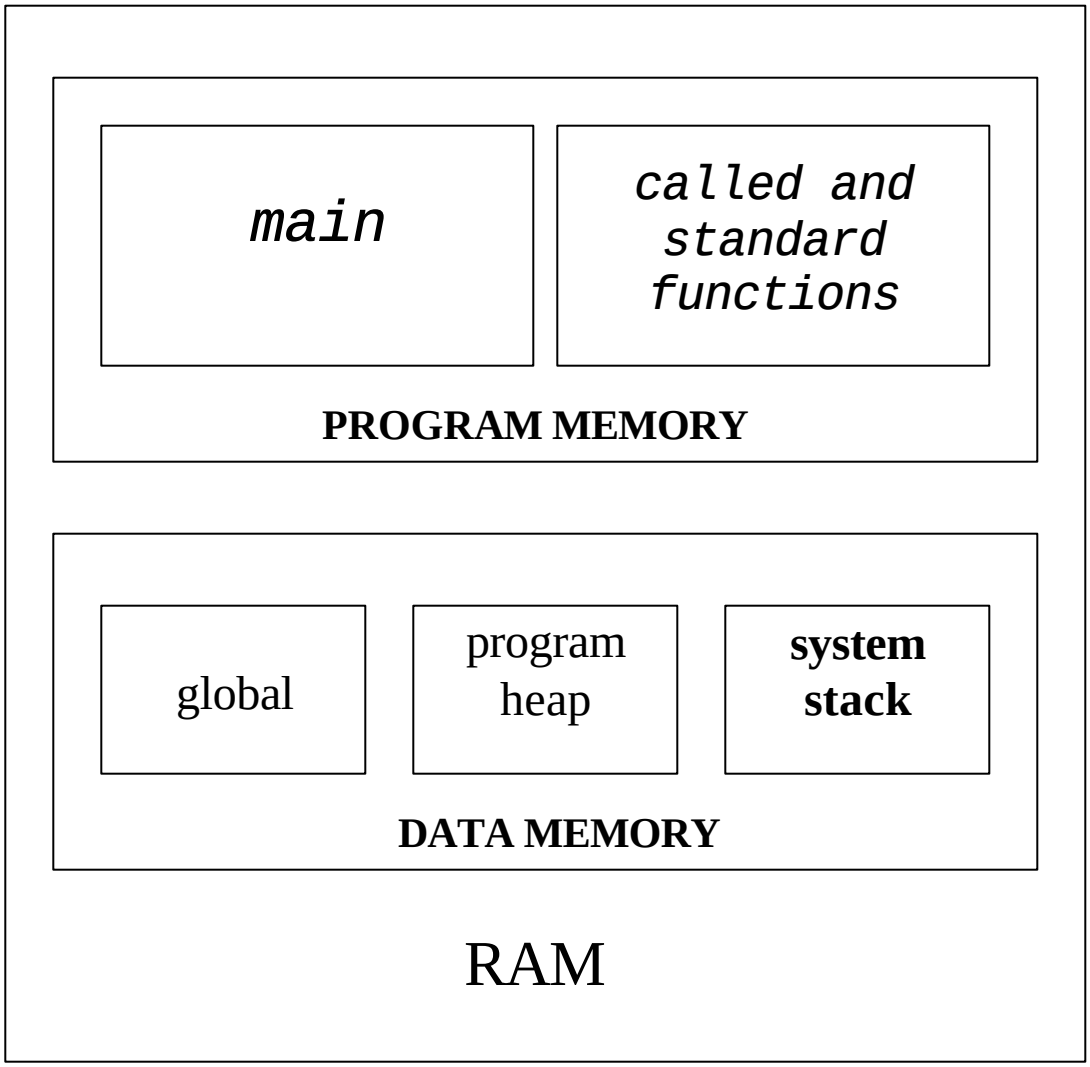
– They are created during compilation. (Fixed memory is reserved for them.)
– They cannot be allocated / de-allocated during the execution of the program.
– Names are associated with them.

```
int x;
char y[10];
int z[100];
```

- **Dynamic Variables**:

– They are created (allocated) and de-allocated during the execution of the program.
– no names are associated with them. The only way to access them is to use pointers.
– They don't exist during compilation. Once they are created they contain data and must have a type like any other variable. Thus we can talk about creating a new dynamic variable of type *x* and setting a pointer to point to it, or returning a dynamic variable of type *x* to the system (de-allocation).

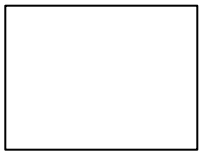
A Conceptual View of Memory



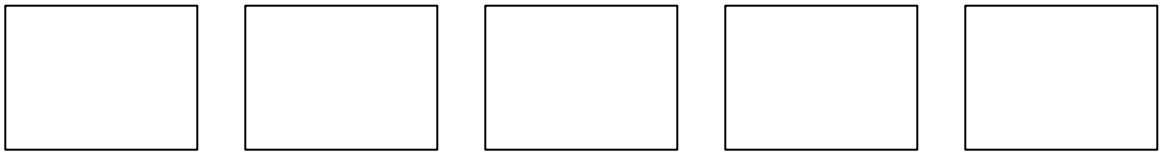
Dynamic Data

For example:

- We must maintain a list of data
- At some moments, the list is small, so we want to use only a little memory



- At other moments, the list is larger, so we need to use more memory



- Declaring variables in the standard way won't work here because we *don't know how many* variables to declare
- We need a way to *allocate* and *de-allocate* data *dynamically* (i.e., *on the fly)*

Dynamic Memory Allocation

Creating and maintaining dynamic data structures requires dynamic memory allocation – the ability for a program to obtain more memory space at execution time to hold new values, and to release space no longer needed.

In C, functions `malloc` and `free`, and operator `sizeof` are essential to dynamic memory allocation.

- Unary operator `sizeof` is used to determine the size in bytes of any data type.

  e.g.

  ```
  sizeof(double)
  sizeof(int)
  ```

- Function `malloc` takes as an argument the number of bytes to be allocated and return a pointer of type `void *` to the allocated memory. (A `void *` pointer may be assigned to a variable of any pointer type.) It is normally used with the `sizeof` operator.

Let us say we want to store an integer pointed by nump, a character pointed by letp and a user defined structure of type planet_t pointed by planetp. This will be done as follows:

```
int         *nump;
 char       *letp;
planet_t *planetp;      user defined structure type
```

```
nump = (int *)malloc(sizeof (int));

letp = (char *)malloc(sizeof (char));

planetp = (planet_t *)malloc(sizeof (planet_t));
```

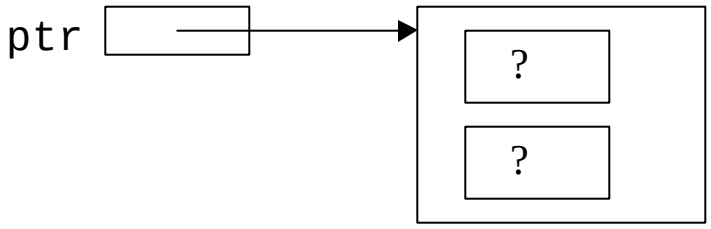
nump will now point to one byte of integer location in memory.

Letp will now point to one location in memory to hold a character

Planetp will now point to a group of locations in memory sufficient to hold the information needed for a structure of type planet_t.

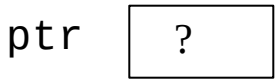
An example:

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

struct node *ptr;

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



- 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 impossible.

– 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.

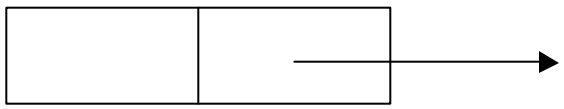
- But getting to a particular node may take large number of operations, as we do not know the address of any individual node .

- 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;
}
```

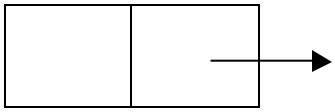


data **next**

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

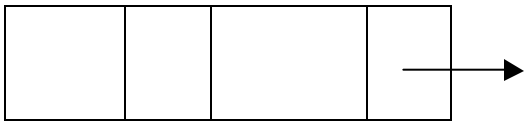
A node with one data field:



number link

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

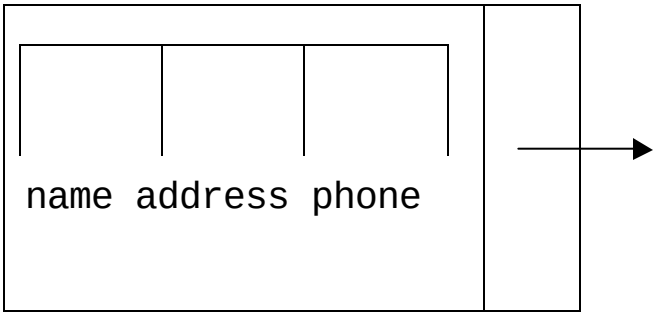
A node with 3 data fields:



name id grdPts next_student

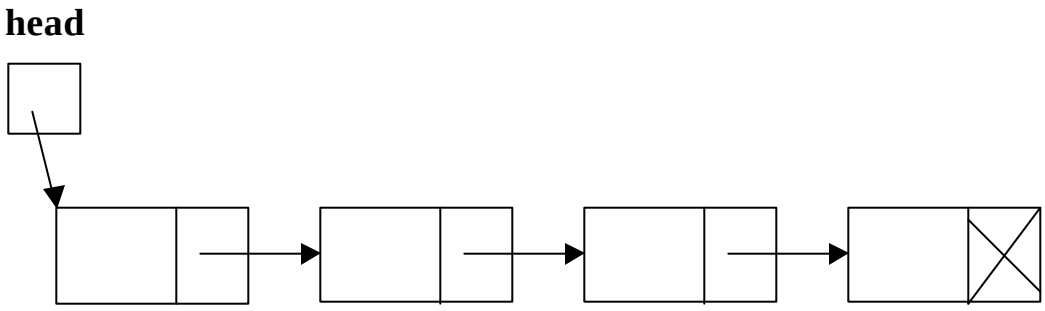
```
 struct student{
     char name[20];
     int id;
     double grdPts;
     struct student
          *next_student;
 };
```

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



- 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 to Empty List

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.

```
typedef   struct node_s{
    int     data;
    struct node_s *next;
} node_t;

node_t   *pNew, *pHead;
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

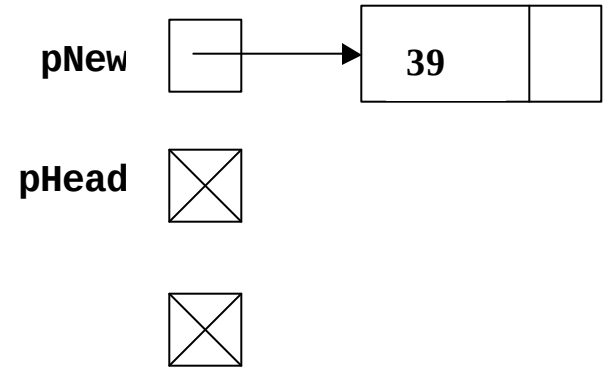
```
pNew->data = 39;
pNew->next = NULL;
```

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

Pointer pHead points to NULL.

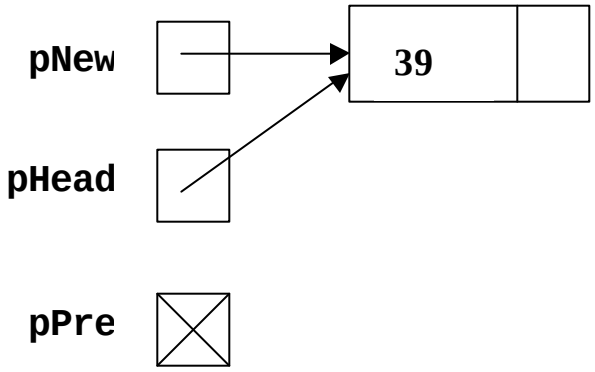
First element 39 is stored in node pNew.

BEFORE




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

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

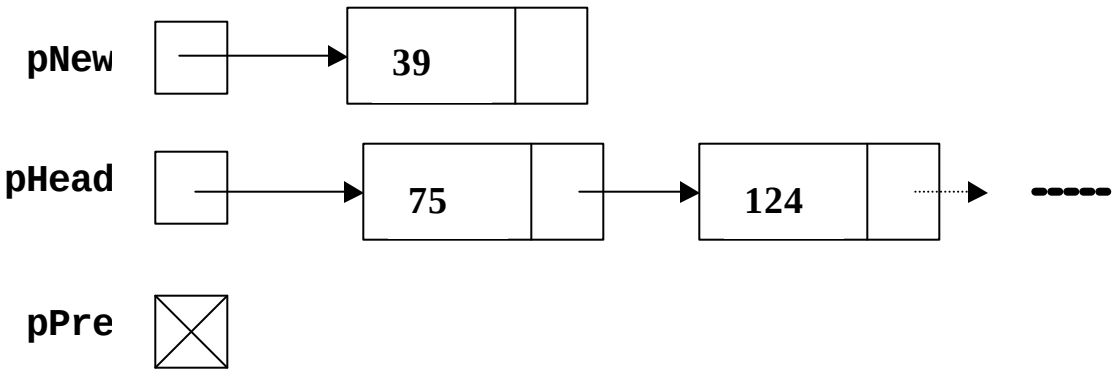
AFTERWARDS



Add Node at Beginning

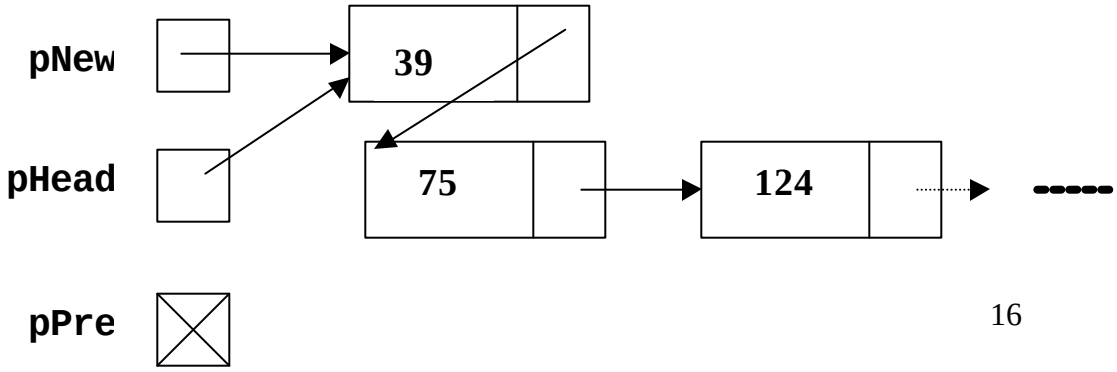
Suppose we want to add node containing data 39 to a list 75, 124,….. The head node points to node containing first element 75.

BEFORE



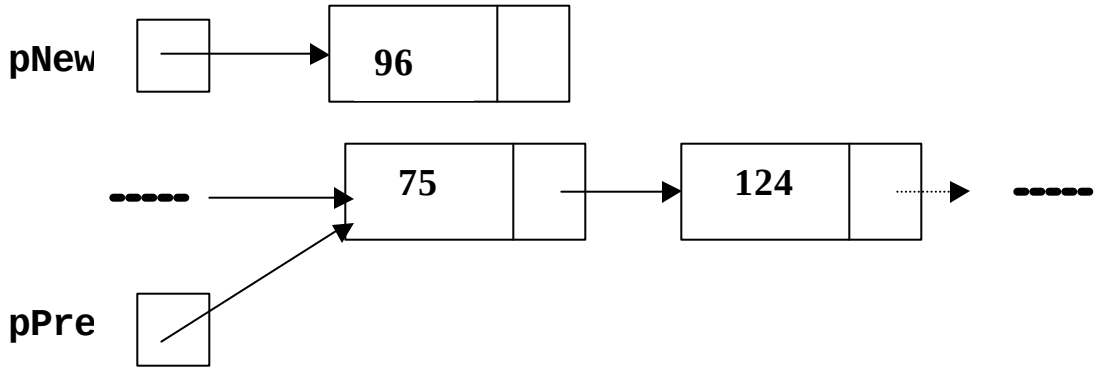

```
pNew->next = pHead;
pHead = pNew;
```

AFTER



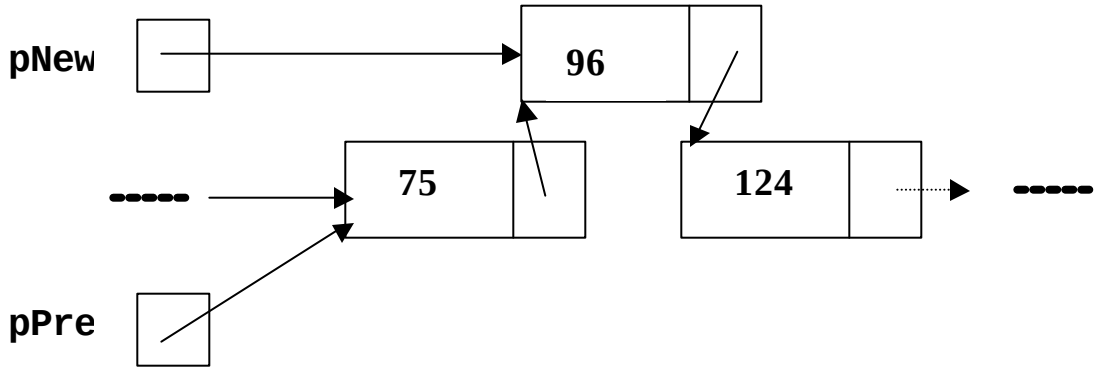
Insert a node in the middle

Now a new node containing value 96 is to be inserted in between nodes containing 75 and 124 . Let pPre be the previous node containing 75.



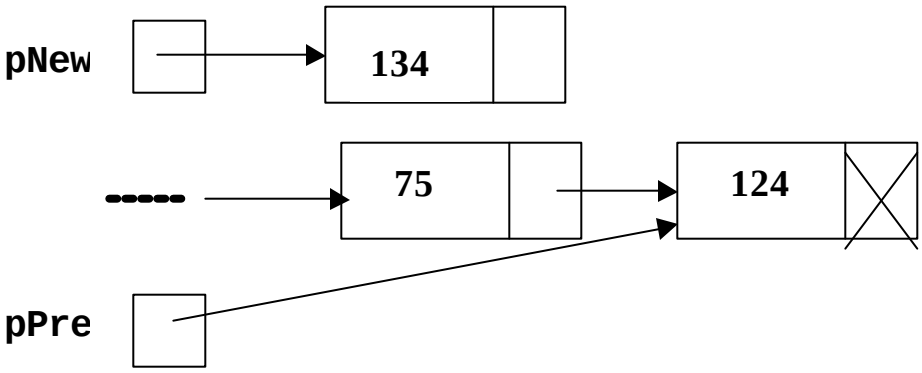

```
pNew->next = pPre->next;
pPre->next = pNew;
```

AFTER



Add Node at End

BEFORE

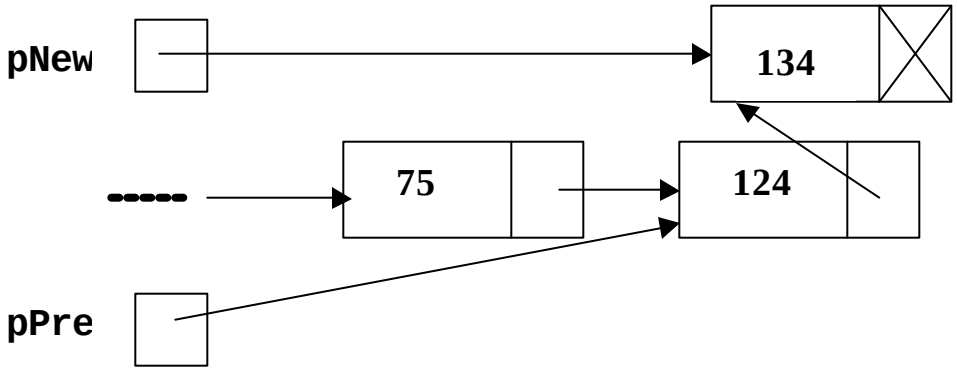



```
pNew->next = NULL;
pPre->next = pNew;
```

OR:

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

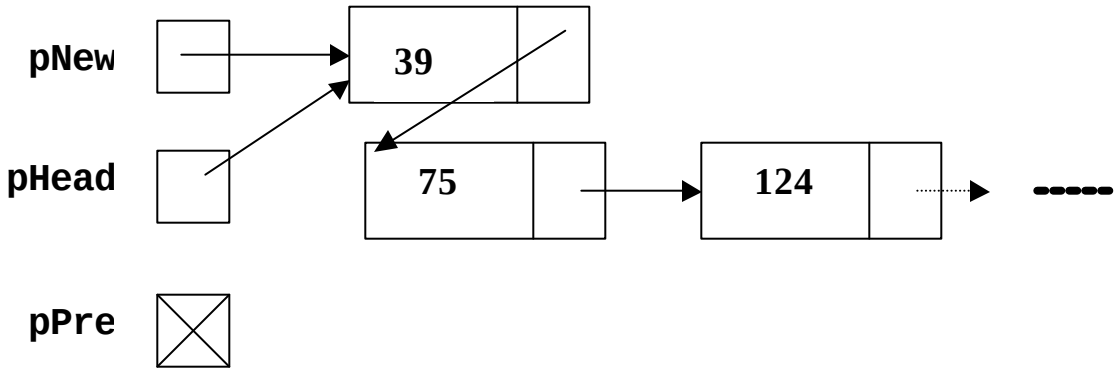
AFTER



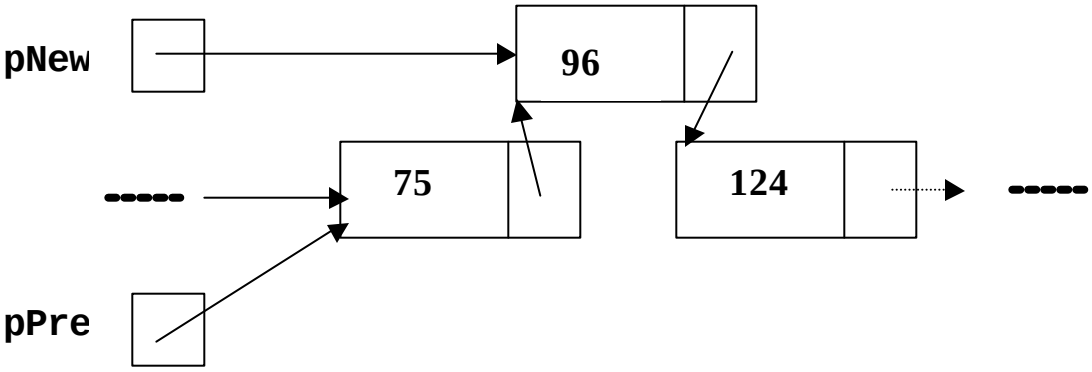
Inserting a Node in a Linked List

Given the head pointer (pHead), the predecessor (pPre) and the data to be inserted (item), we must allocate memory for the new node (pNew) and adjust the link pointers.

```
/*Insert a node in a linked list*/

 struct node *pNew;

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

 pNew->data = item;

 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;
}
```

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;
}
```

Look up an item in the list pointed by head

/* Given the item and the pointer to the head of the list,

the function returns NULL if the item is not found;

or

returns a pointer to the node which matches the item */

```
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));
}
```

Creating a List

- **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);
```