LINKED LIST MANIPULATIONS

Deleting a Node from a Linked List

Deleting a node requires that we logically remove the node from the list by changing various link pointers and then physically deleting the node from the heap.

We can delete

- the first node
- any node in the middle
- the end node

To logically delete a node:

1. first locate the node itself , name the current node as `pCur` and its predecessor node as `pPre`.

2. change the predecessor's link field to point to successor of the current node.

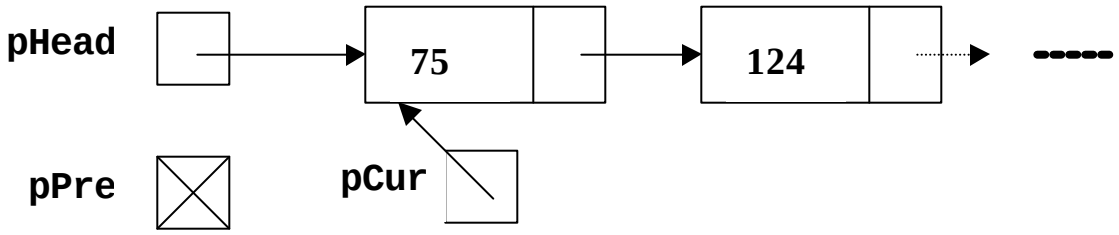
3. recycle the node (send it back to memory) using *free*.

Note: We may be deleting the only node in a list. So take care of it separately.

This will result in an empty list in which case the head pointer is set to NULL.

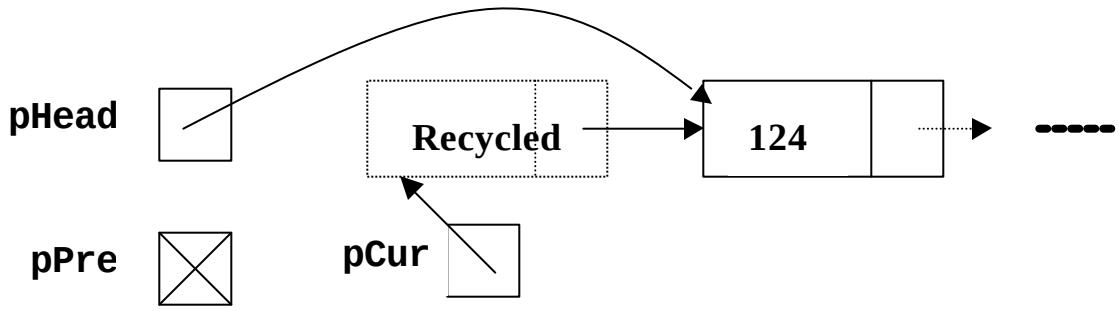
Delete First Node

BEFORE



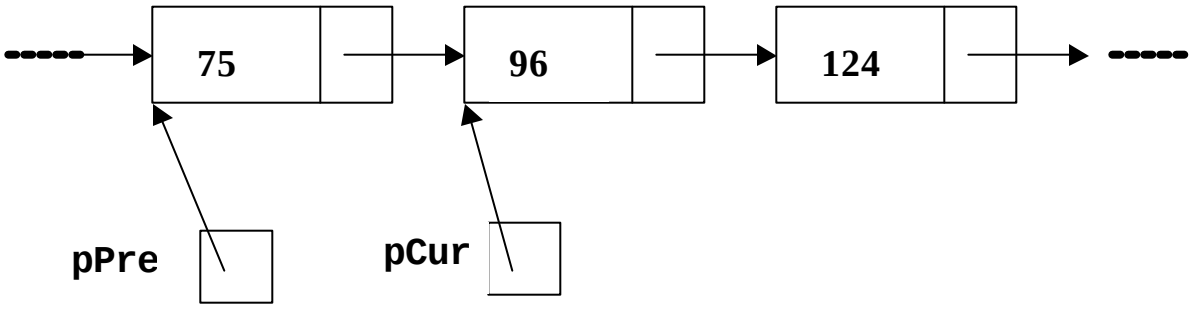

```
pCur = pHead;
pHead = pCur -> next;
free (pCur);
```

AFTER



General Delete Case

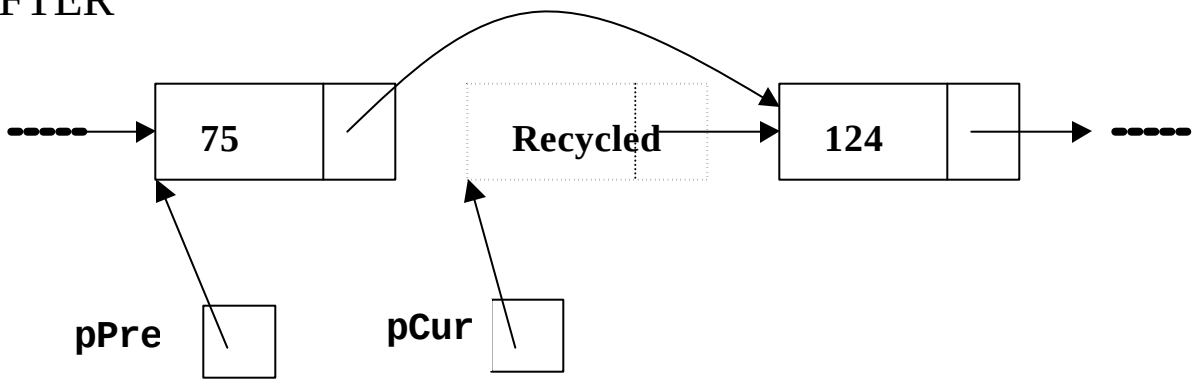
BEFORE



Here , you have to keep track of not only the node to be deleted but also its predecessor

```
pPre->next = pCur->next;
free(pCur);
```

AFTER



Algorithm for Deleting a node

Given pHead,
PCur, the node to be deleted,
pPre the delete node's predecessor,
Delete the node and **recycle** it back to memory.

```
   if (pPre == NULL)
          // Deleting first node
     pCur = pHead;
      pHead = pCur->next;

 else
           // Deleting other nodes
      pPre->next = pCur ->next;

   free(pCur);
```

But, we have to move from the head node to the current node , and at the same time the pointer to its predecessor must be moved one step at a time.

This implies searching the list.

Search Linked List

Both insert and delete operations need searching the linked list.

- To add a node, we must identify the logical predecessor of the new node.
- To delete a node, we must identify the location of the node to be deleted and its logical predecessor.

Basic Search Concept

Given a target value, the search attempts to locate the requested node in the linked list.

If a node in the list matches the target value, the search returns 1; otherwise it returns 0.

```
// Start with a dummy pointer to headnode
pCur = pHead;


// Search until target is found or we reach
// the end of list


while (pCur != NULL ){
if( pCur->data == target)
```

```
return 1;
pCur = pCur->next;
}

// target not found

return 0;
```

Traversing Linked Lists

List traversal requires that all of the data in the list be processed.

```
// Traverse a linked list

struct node *pWalker;

pWalker = pHead;
printf("List contains:\n");

while (pWalker != NULL)
{
   printf("%d ", pWalker->data);
   pWalker = pWalker ->next;
}
```

Manipulation of Linked Lists

To manipulate linked lists, we can pass simply the "pointer-to-the list " to a function. The purpose of the function may be to create a fresh list, append items, print list of items or delete a particular item.

Let us take an example. Suppose we have a list of nodes containing integer values. We can define a structure

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

Now we can define any node in terms of node_t.

In main program we may have two statements, one to define a sequence of nodes starting from the address pList1

```
Node_t *pList1;
print_list(pList1);
```

/* i.e. **pList1** points to a sequence of nodes all of type **node_t** */

The function to print list can be

```
void  print_list( node_t *headp)
{
  if(headp==NULL)  {
     printf("\n");
   }
else  {
     printf("%d", headp->digit);
     print_list(headp->next);
   }
}
```

So here it is printing one data item and then calling the function recursively to print the rest of the nodes.

Passing pointer to node pointer

Now let us take a different way to access the list. Instead of passing just the pointer to head of the list, we pass the reference to the complete list itself. The main program now sends a pointer to the address of the node (pointer to pointer).

In the following program, we pass the pointer-to-a-list to a function named Addstart, and ask it to add elements to the start of the list.

Here *list refers to a data structure which consists of sequence of nodes(with a specific structure).

Then we pass the address of the list to a PrintList function, to print the contents of the linked list.

Creating and Printing a Linked List

/* To add elements at the start of a list and to print the contents */

```
#include <stdio.h>
#include <stdlib.h>

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

/* The following function modifies the given list by adding a new node with the data given.
Note , we are sending the whole list so it is coming as *list. Double dereferencing because it is pointer to address of list */

```
int Addstart(struct node* *list,int d )
{
```

/* request Memory to allocate a node pNew with structure of node */

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

```
pNew-> data =  d ;
pNew->next = NULL) ;

if(*list== NULL)
   {
       *list  =  pNew;
    }
else
   {
       pNew->next  =  *list;
      *list  =  pNew ;
   }
 return 1;
}
```

/* the following function prints out the contents of given list */

```
int PrintList( struct node *list)
{
 struct node current =   list ;
 /* dummy name "current "  assigned to list . We shall be moving down the list by following the " next " link of this "current ", without disturbing any item in the list including its name. */

while (current != NULL)
    {
        printf("%d", current -> data);
        current    =   current -> next;
     }
 return  1;
}
```

Having defined the two functions, we are now in a position to write the main program. Here we prompt the user to supply a number, which is to be added to the beginning of the list. The user is to type -1 whenever he wants to quit.

/* The main function */

```
main( ) {
int number = 0;
struct node *pList
/* create our own main variable */

pList=NULL;          /* Initialize the linked list */
while(number!= -1)
{
      printf("enter data for next node \n ");
      scanf("%d", &number);

      if (number !=-1)
        {
           AddStart ( &pList, number );
         /* pass address of pList to the add function */
        }
 }
printf("items in linked list \n");
PrintList (  pList  );
return 1;
}
```

Creating a list by adding elements at the end of list

The objective here is to create a linked list of numbers , with the numbers being added at the end of the list. The user is asked to type all the numbers in a row followed by –1 in one line.

```
#include <stdio.h>
#include <stdlib.h>

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

int AddElement ( struct node ** list,
int data)
{
 struct node * tempNode=(struct node *)
(malloc(sizeof(struct node)));

tempNode-> data =data;
tempNode->next= NULL) ;
if (   list   ==  NULL)
    {
       list  =  tempNode;
    }
else
```

```
   {
```
/*simply add to the end of the list. Note to find the end we must go through the whole list . Start list at the front , and dereference to get the actual list */

```
     struct node *end;
     end  =   *list;
     while  ( end->next    != NULL )
              {
                  end =   end->next;
              }
     end->next = tempNode;
    }
return 1;
}
int PrintList( struct node *list)
{
 struct node *currentFront  =  list;
```

/* this assigns a variable to point to the front on the linked list. If we do not do this and traverse list directly then we will lose the front of the list */

```
while (currentFront !  =  NULL)
   {
     printf("%d", currentFront-> data);
    currentFront= currentFront -> next;
   }
return 1;
}
```

/* The main function */

```
main() {
struct node *pList
pList=NULL;

int number = 0;
char line[500];
char response='y';
while(number! =  -1)
 {
     printf("please enter a number to
add to the list \n");
     scanf("%s", line);
     number=atoi(line);
     /* atoi converts string to a number   */
     if(number  !=  -1)
       {
          AddElement(&pList,  number );
        }
  }
printf(" numbers you entered were \n");
PrntList (  pList  ) ;
return 1;
}
```