Pointers

Structural Programming

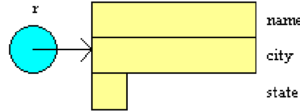
Pointers to Structures

- Example 1:

```
struct rec
{
   int i;
   float f;
   char c;
};

int main()
{
   struct rec *p;

   p=(struct rec *) malloc (sizeof(struct rec));

   (*p).i=10;
   (*p).f=3.14;
   (*p).c='a';
   printf("%d %f %c\n",(*p).i,(*p).f,(*p).c); free(p);
}
```

Pointers to Structures

- Example 2:



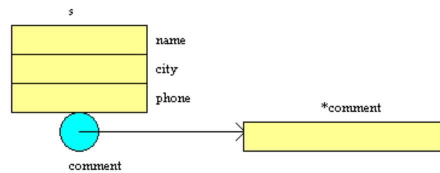

```
typedef struct
{
   char name[21];
   char city[21];
   char state[3];
} Rec;
typedef Rec *RecPointer;

RecPointer r;
r = (RecPointer)malloc(sizeof(Rec));

strcpy((*r).name, "Leigh"); //or strcpy(r->name, "Leigh");
strcpy((*r).city, "Raleigh");
strcpy((*r).state, "NC");
printf("%s\n", (*r).city);
free(r);
```

The **r->** notation is exactly equivalent to **(*r).** but takes two fewer characters.

Structures Containing Pointers

- Example:

```
typedef struct
{
   char name[21];
   char city[21];
   char phone[21];
   char *comment;
} Addr;

Addr s;
char comm[100];

gets(s.name, 20);
gets(s.city, 20);
gets(s.phone, 20);
gets(comm, 100);
s.comment=(char*)malloc(sizeof(char[strlen(comm)+1]));
strcpy(s.comment, comm);
```

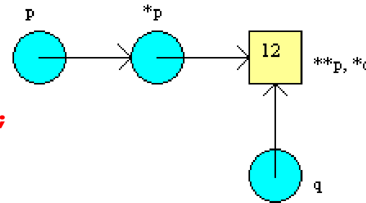


This technique is useful when only some records actually contained a comment in the comment field.

If there is no comment for the record, then the comment field would consist only of a pointer (4 bytes).

Pointers to Pointers

- This technique is sometimes called a **handle**
- useful in certain situations where the operating system wants to be able to move blocks of memory on the heap around at its discretion.
- Example:

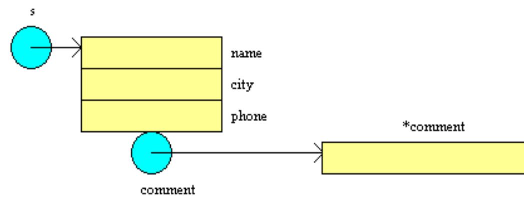



```
int **p;
int *q;

p = (int **)malloc(sizeof(int *));
*p = (int *)malloc(sizeof(int));
**p = 12;
q = *p;
printf("%d\n", *q);
free(q);
free(p);
```

- Pointers to pointers are frequently used in C to handle pointer parameters in functions.

Pointers to Structures Containing Pointers(1)

- Example:



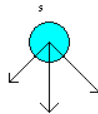

```
typedef struct
{
     char name[21];
     char city[21];
     char phone[21];
     char *comment;
} Addr;

Addr *s;
char comm[100];

s = (Addr *)malloc(sizeof(Addr));
gets(s->name, 20);
gets(s->city, 20);
gets( s->phone, 20);
gets(comm, 100);
s->comment = (char *)malloc(sizeof(char[strlen(comm)+1]));
strcpy(s->comment, comm);
```

Pointers to Structures Containing Pointers(2)

- In this example, it is very easy to create lost blocks if you aren't careful. For example:

```
s = (Addr *)malloc(sizeof(Addr));
gets(comm, 100);
s->comment = (char*)malloc(sizeof(char[strlen(comm)+1]));
strcpy(s->comment, comm);
free(s);
```

- This code creates a lost block because the structure containing the pointer pointing to the string is disposed of before the string block is disposed of, as shown below:

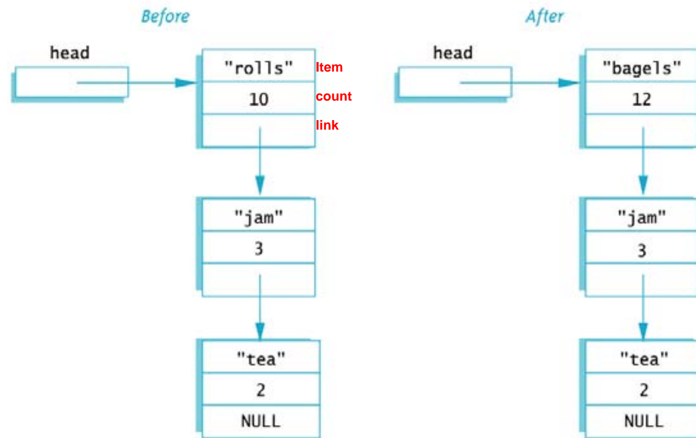


Orphaned block of memory

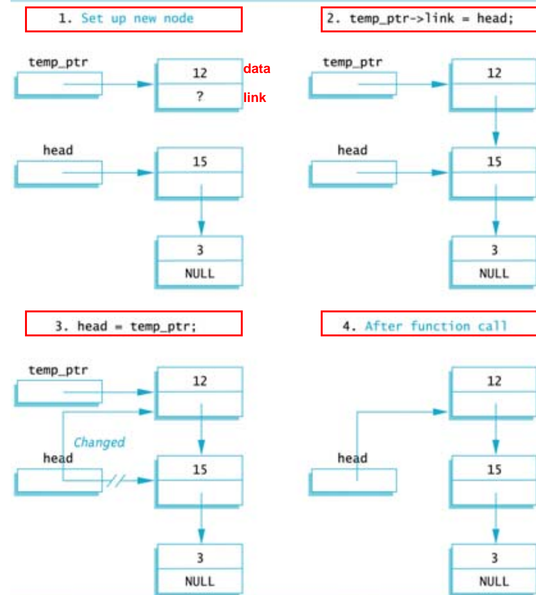
Linked Lists

Structural Programming

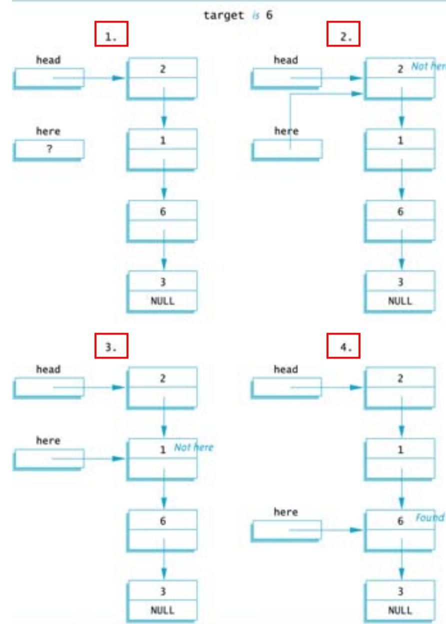
Λίστες



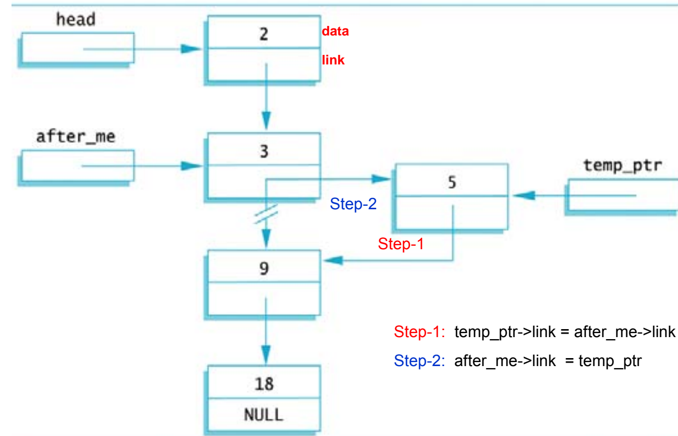
Λίστες



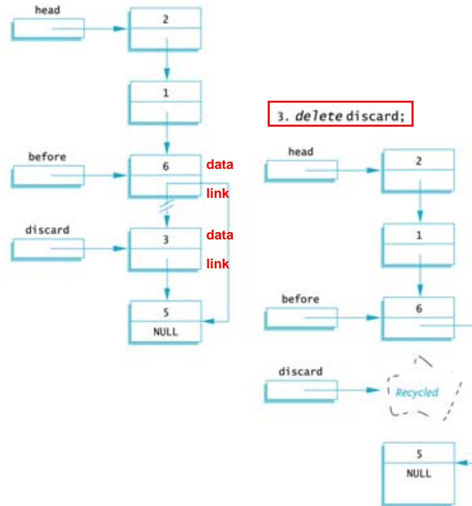
Λίστες



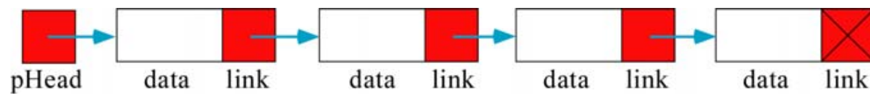
Λίστες



Λίστες



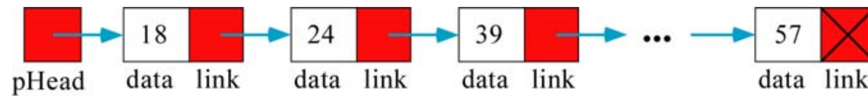
Λίστες



A LINKED LIST WITH A HEAD POINTER pHead

AN EMPTY LINKED LIST

Λίστες



pPre

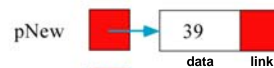
NULL: Add to empty list or add at beginning of list

pPre

Not NULL: Add in middle of list or add at end of list

Λίστες – Add node (σε άδεια λίστα)

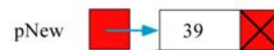
BEFORE ADD



pHead

pPre

```
pNew->link  = pHead ;
pHead       = pNew ;
```

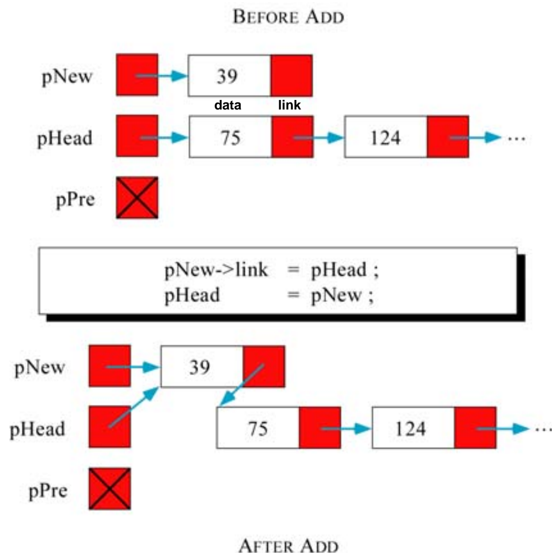


pHead

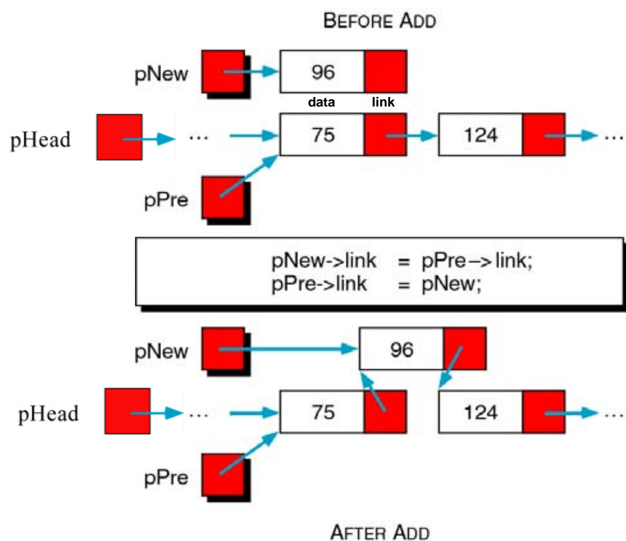
pPre

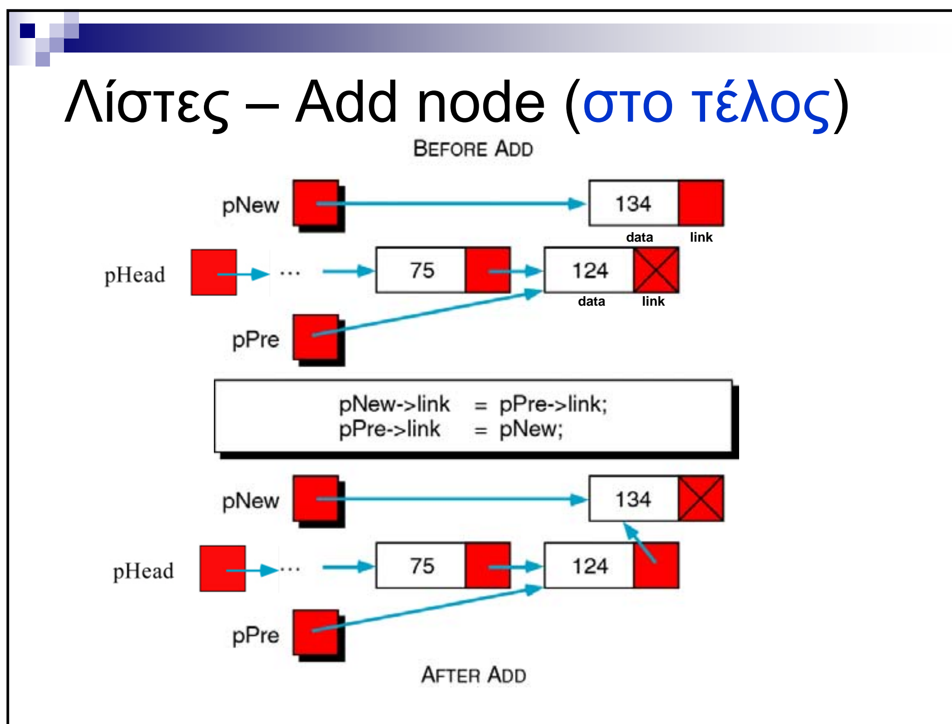
AFTER ADD

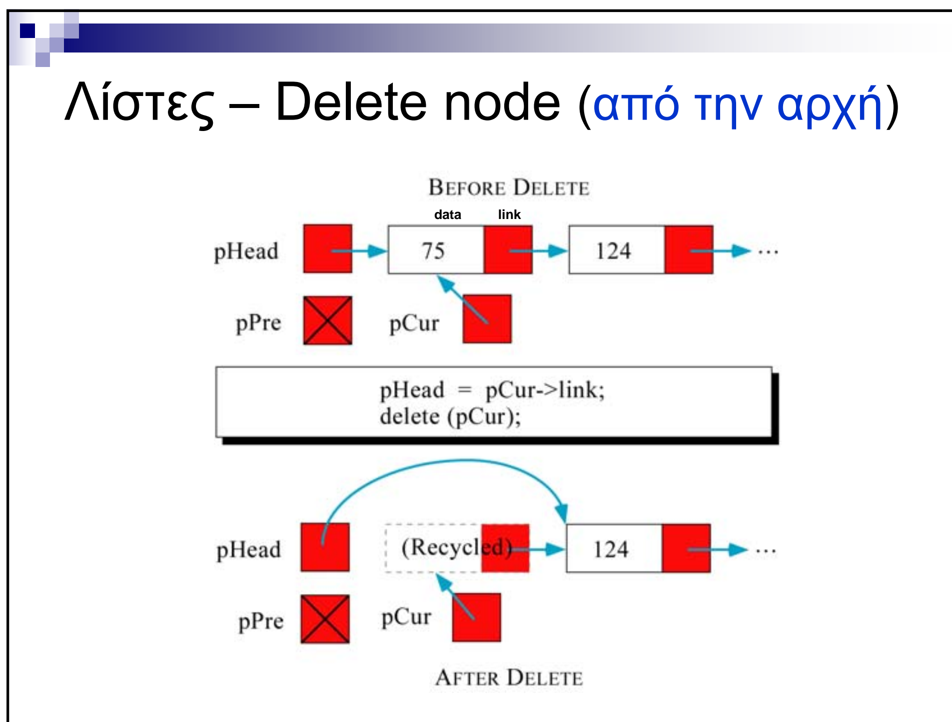
Λίστες – Add node (στην αρχή)



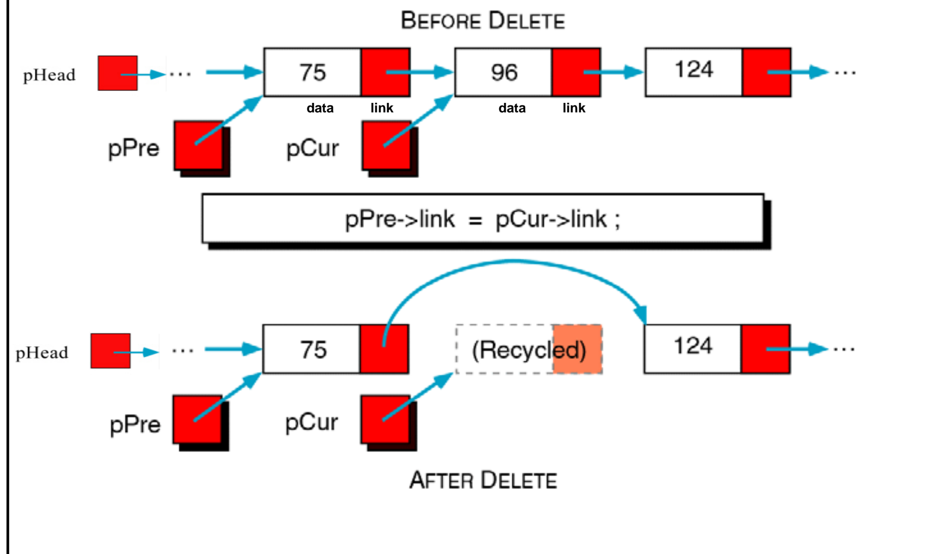
Λίστες – Add node (στη μέση)




Λίστες – Add node (στο τέλος)
BEFORE ADD
pNew
134
data
link
pHead
...
75
124
data
link
pPre
pNew->link = pPre->link;
pPre->link = pNew;
pNew
134
pHead
...
75
124
pPre
AFTER ADD


Λίστες – Delete node (από την αρχή)
BEFORE DELETE
data
link
pHead
75
124
...
pPre
pCur
pHead = pCur->link;
delete (pCur);
pHead
(Recycled)
124
...
pPre
pCur
AFTER DELETE

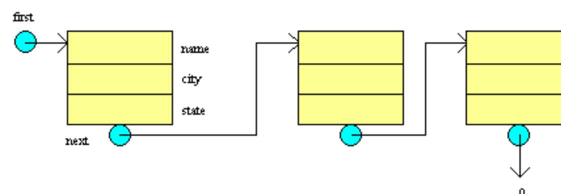
Λίστες – Delete node (από τη μέση)



Lists

- It is possible to create structures that are able to point to identical structures:

```
typedef struct
{
    char name[21];
    char city[21];
    char state[21];
    Addr *next;
} Addr;
Addr *first;
```



Building a linked list in C

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

struct list_el {
   int val;
   struct list_el * next;
};

typedef struct list_el item;

void main() {
   item * curr, * head;
   int i;

   head = NULL;

   for(i=1;i<=10;i++) {
      curr = (item *)malloc(sizeof(item));
      curr->val = i;
      curr->next  = head;
      head = curr;
   }

   curr = head;

   while(curr) {
      printf("%d\n", curr->val);
      curr = curr->next ;
   }
}
```

Double Linked Lists

Double Linked Lists

- Like a single linked list, a double linked list is a list of structures allocated in memory, which are linked together by pointers.

Double Linked Lists vs Single Linked Lists

- The difference between the two types is:
  - that in a **single linked list**, you either have the structure pointer inside each NODE, or structure, point to the next node in the list, or the previous one, etc.
  - In a **double linked list**, each node has two struct pointers within them. One pointer pointers to the next node in the list, and one points to the previous node in the list.

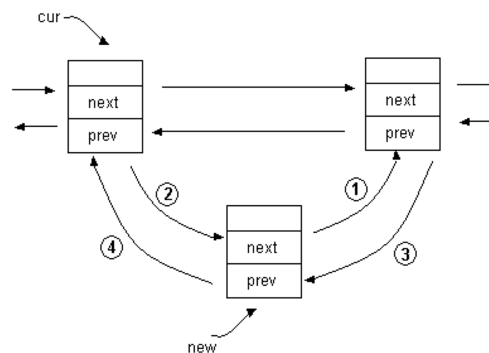
Double Linked Lists vs Single Linked Lists (cont'd)

- A double linked list is good if you want to run through a list from the first node to the last, or from the last node to the first.
- Also, when it comes to deleting a node from a double linked list, there's a bit more work to it. You must remove the node by the list by having the previous node point to the next node and the next node point to the previous node.
- Then you MUST free the memory allocated to this node you are deleting. It is always important to conserve memory.

Double Linked Lists

- Inserting a node

1) new->next = cur->next
2) cur->next = new
3) new->next->prev=new
4) new->prev=cur



Double Linked Lists

- Deleting a node

1) cur->prev->next=cur->next
2) cur->next->prev=cur->prev
3) cur->next=null
4) cur->prev=null
5) Delete cur