Linked Lists

CS10001: Programming & Data Structures

Pallab Dasgupta

Dept. of Computer Sc. & Engg.,
Indian Institute of Technology
Kharagpur

Arrays: pluses and minuses

+ Fast element access.

-- Impossible to resize.

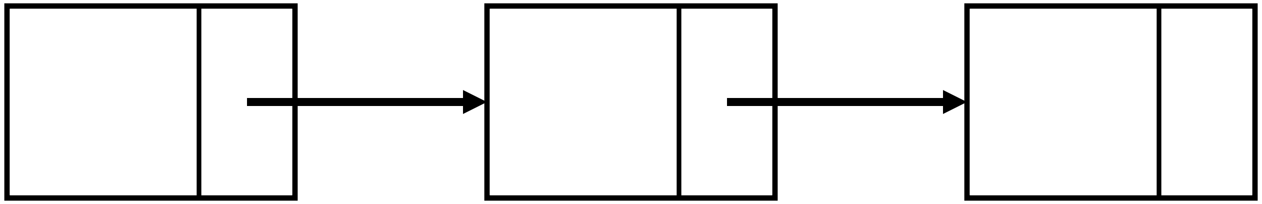
- Many applications require resizing!
- Required size not always immediately available.

Dynamic memory allocation: review

```
typedef struct {
        int hiTemp;
        int loTemp;
        double precip;
} WeatherData;
int main () {
      int numdays;
      WeatherData * days;
      scanf (“%d”, &numdays) ;
      days=(WeatherData *)malloc (sizeof(WeatherData)*numdays);
      if (days == NULL) printf (“Insufficient memory”);
      ...
      free (days) ;
}
```

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;
}  ;
```

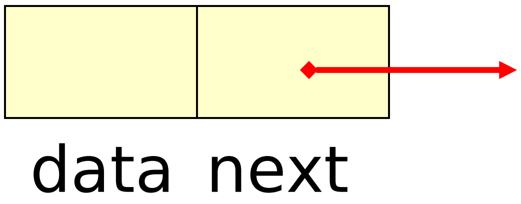
Self-referential structures

```
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

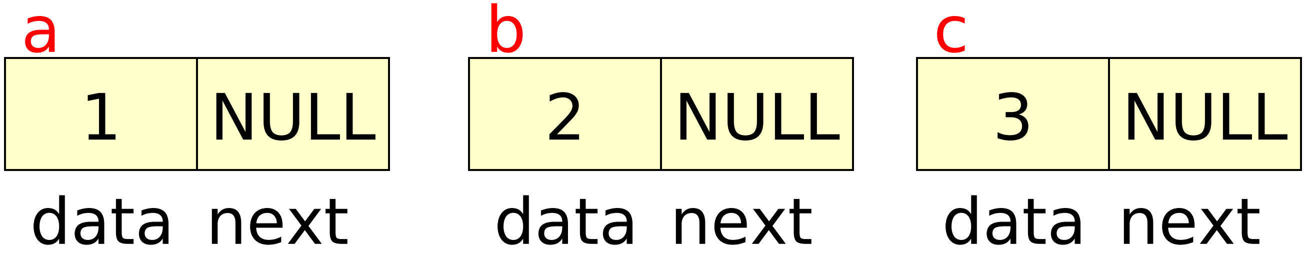


The pointer variable next contains either

- 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;
```



a

1 NULL

data next

b

2 NULL

data next

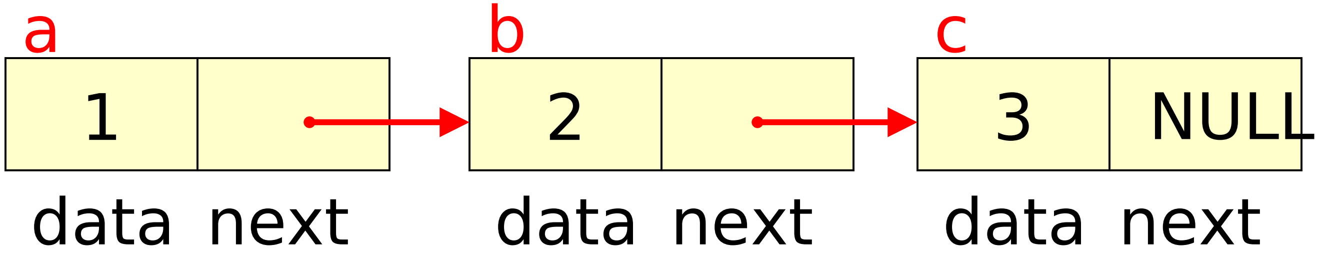
c

3 NULL

data next

Chaining these together

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

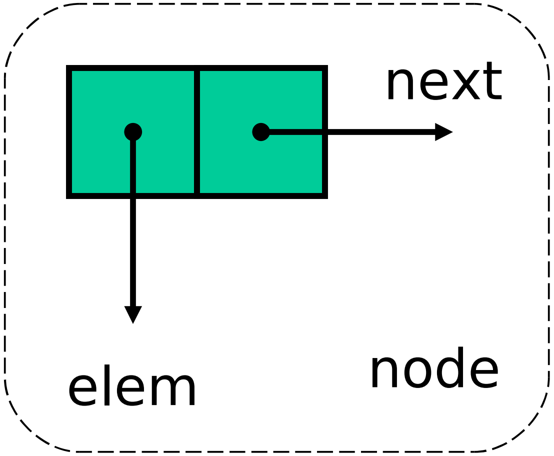


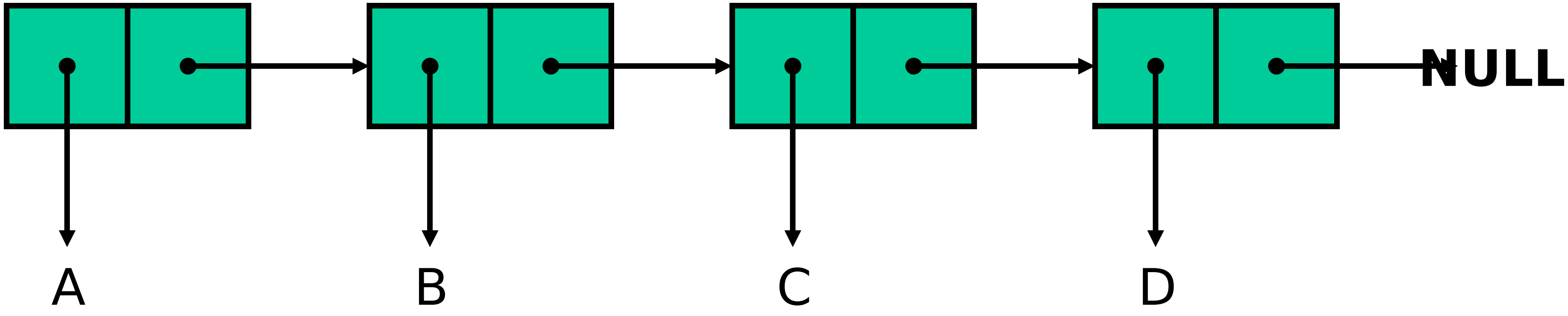
What are the values of :

- a.next->data 2
- a.next->next->data 3

Linked Lists

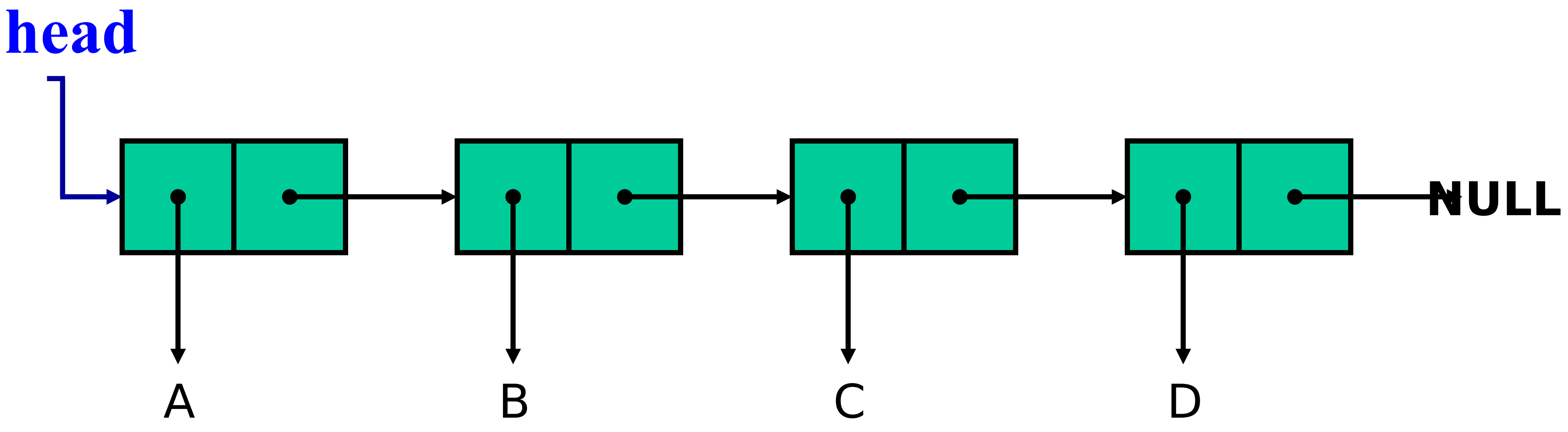
- A singly linked list is a concrete data structure consisting of a sequence of nodes
- Each node stores
  - element
  - link to the next node





Linear Linked Lists

- 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.



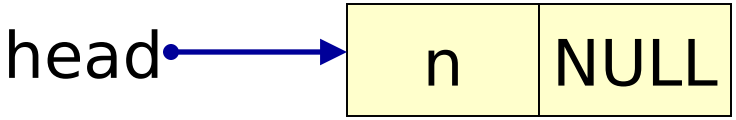
Header file : list.h

```
#include <stdio.h>
#include <stdlib.h>
typedef char DATA;
struct list {
    DATA d;
    struct list * next;
};
typedef struct list ELEMENT;
typedef ELEMENT * LINK;
```

Storage allocation

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

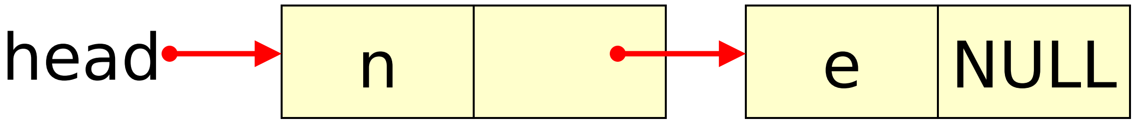
creates a single element list.



Storage allocation

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

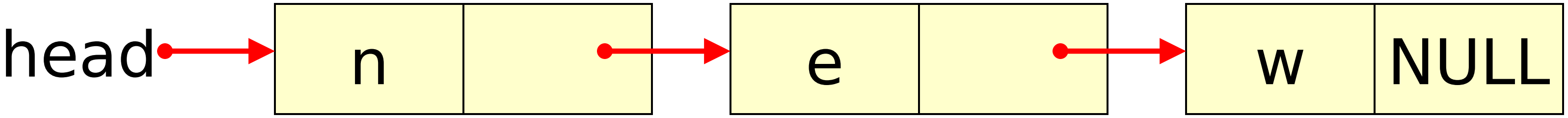
A second element is added.



Storage allocation

```
head->next->next = malloc (sizeof(ELEMENT));
head->next->next->d = 'w';
head->next->next-> = NULL;
```

We have a 3 element list pointed to by head.
The list ends when next has the sentinel value NULL.



List operations

List operations

- (i) How to initialize such a self referential structure (LIST),
- (ii) how to insert such a structure into the LIST,
- (iii) how to delete elements from it,
- (iv) how to search for an element in it,
- (v) how to print it,
- (vi) how to free the space occupied by the LIST?

Produce a list from a string (recursive version)

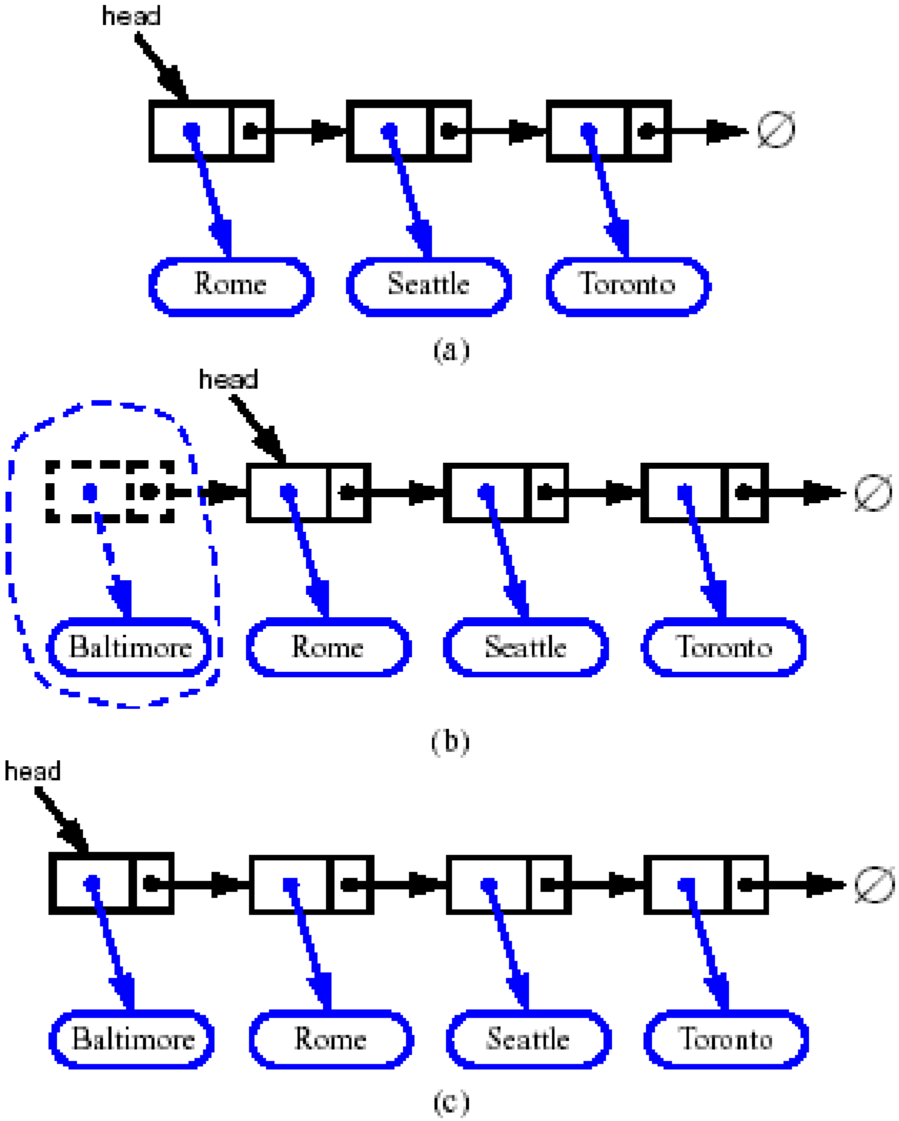
```
#include “list.h”
LINK StrToList (char s[]) {
    LINK head ;
    if (s[0] == ‘\0’)
    return NULL ;
    else   {
    head = malloc (sizeof(ELEMENT));
    head->d = s[0];
    head->next = StrToList (s+1);
    return head;
    }
}
```

list from a string
(iterative version)

```
#include “list.h”
LINK SToL (char s[]) {
    LINK head = NULL, tail;
    int          i;
    if (s[0] != ‘\0’)   {
      head = malloc (sizeof(ELEMENT));
      head->d = s[0];
      tail = head;
      for (i=1; s[i] != ‘\0’; i++)  {
             tail->next = malloc(sizeof(ELEMENT));
             tail = tail->next;
             tail->d = s[i];
      }
      tail->next = NULL;
    }
    return head;
}
```

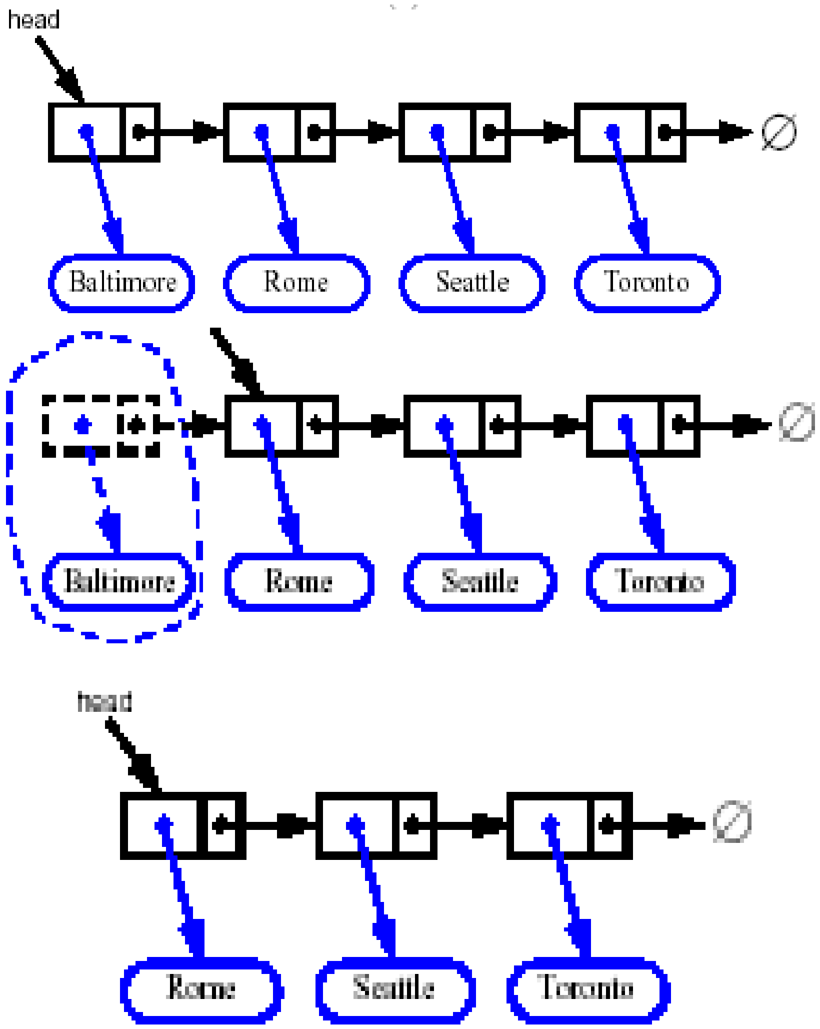
Inserting at the Head

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



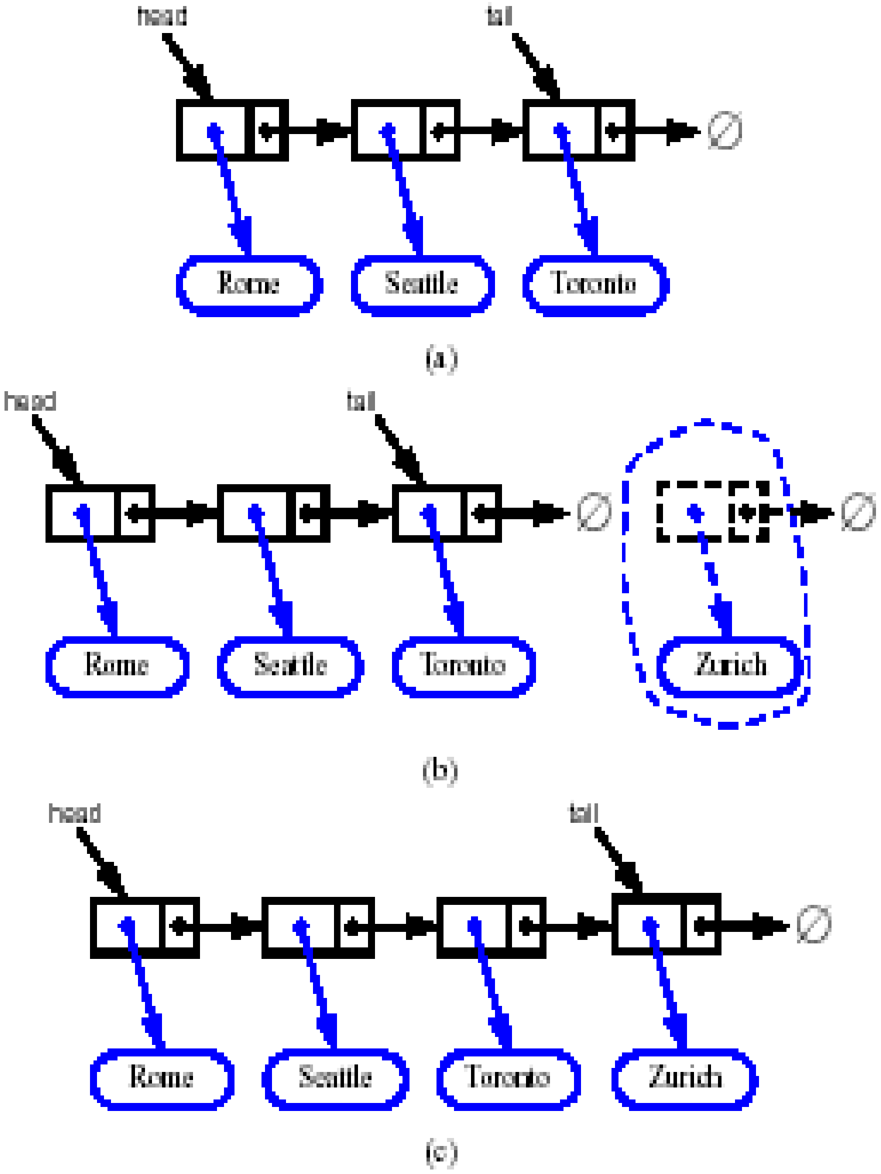
Removing at the Head

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



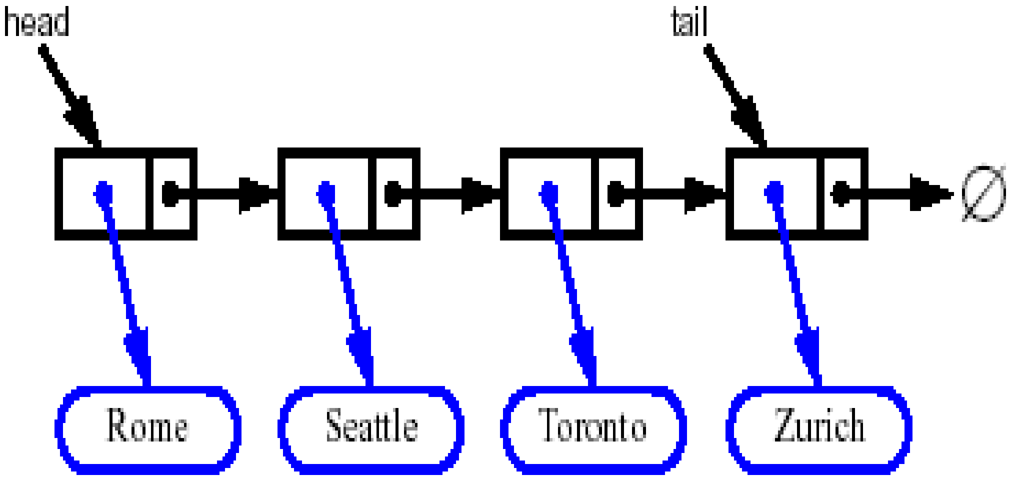
Inserting at the Tail

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



Removing at the Tail

- Removing at the tail of a singly linked list cannot be efficient!
- There is no constant-time way to update the tail to point to the previous node

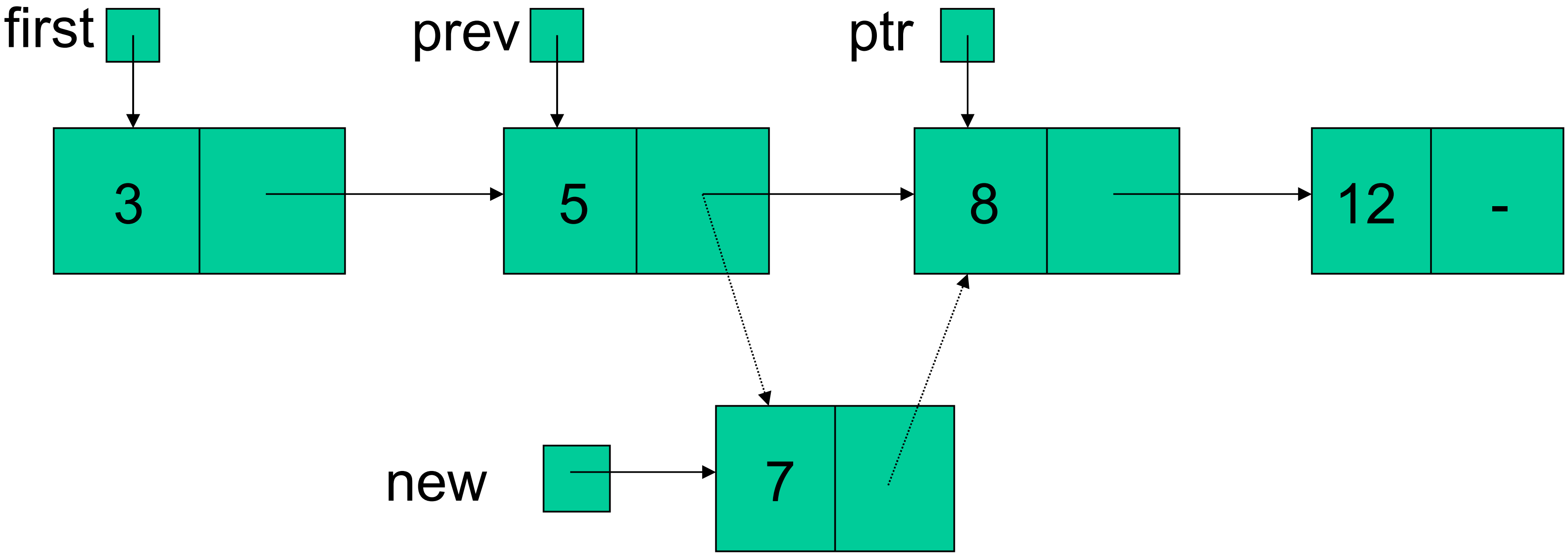


Insertion

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** previ**ous (even if last) and** ptr **nodes***
- ***Also check insertion** before first **node!***

Header file : list.h

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

Create_node function

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

insert function

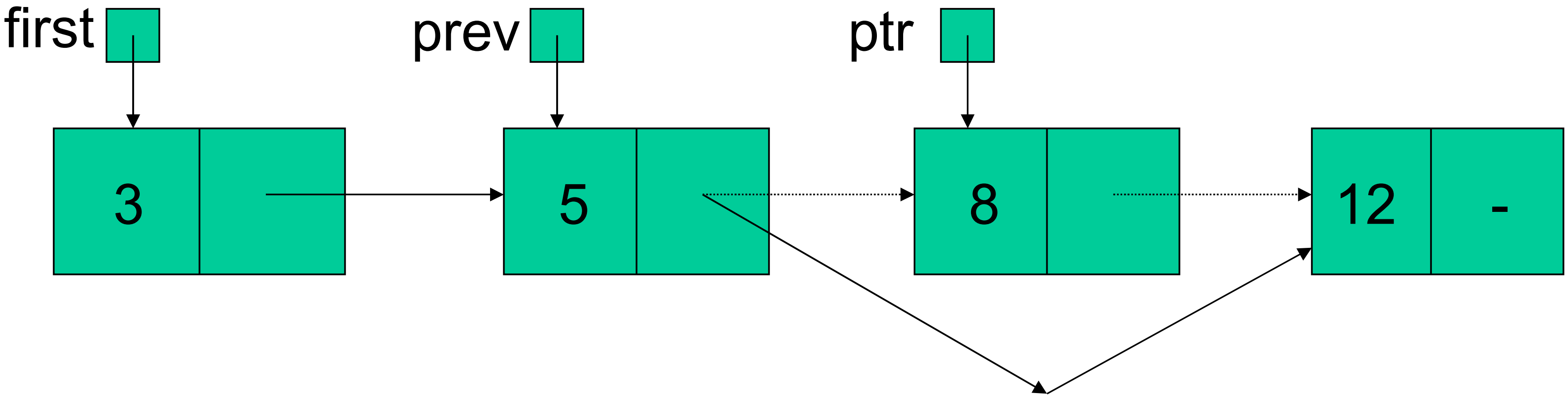
```
LINK insert (int data,  LINK ptr)
{
    LINK new, prev, first;
    new = create_node(data);
    if (ptr == NULL || data < ptr -> value)
    {                // insert as new first node
          new -> next = ptr;
          return new;
                     // return pointer to first node
    }
 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



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

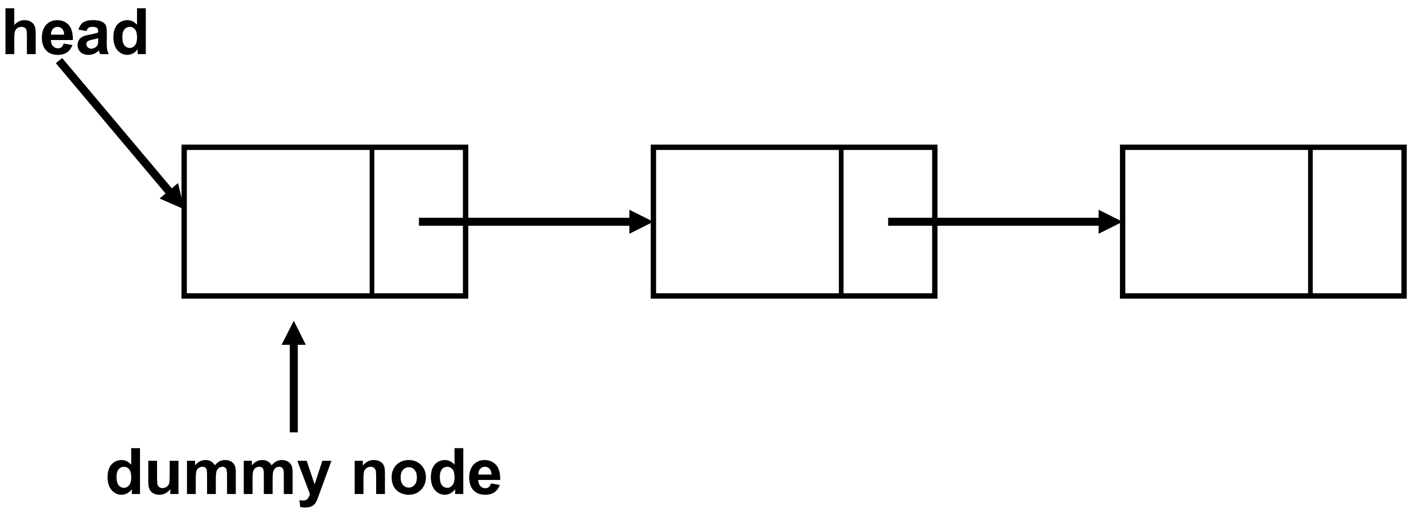
- **Also check whether** first **node is to be deleted.**

```
//  delete the item from ascending list
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;
        }
```

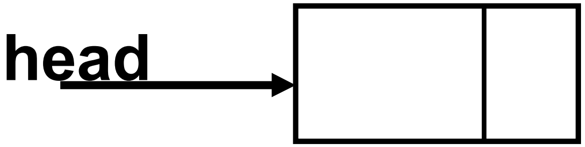
```
                if (ptr == NULL || data != ptr-
>data)
                        // NOT found in ascending
list
                        // nothing to delete
                {
                        return first;  //  original
                }
                else    //  found, delete ptr node
                {
                        prev -> next = ptr -> next;
                        free(ptr);     //  free node
                        return first;  //  original
                }
        }
    }   //  end delete
```

Representation with Dummy Node



- **Insertion at the beginning is the same as insertion after the dummy node**

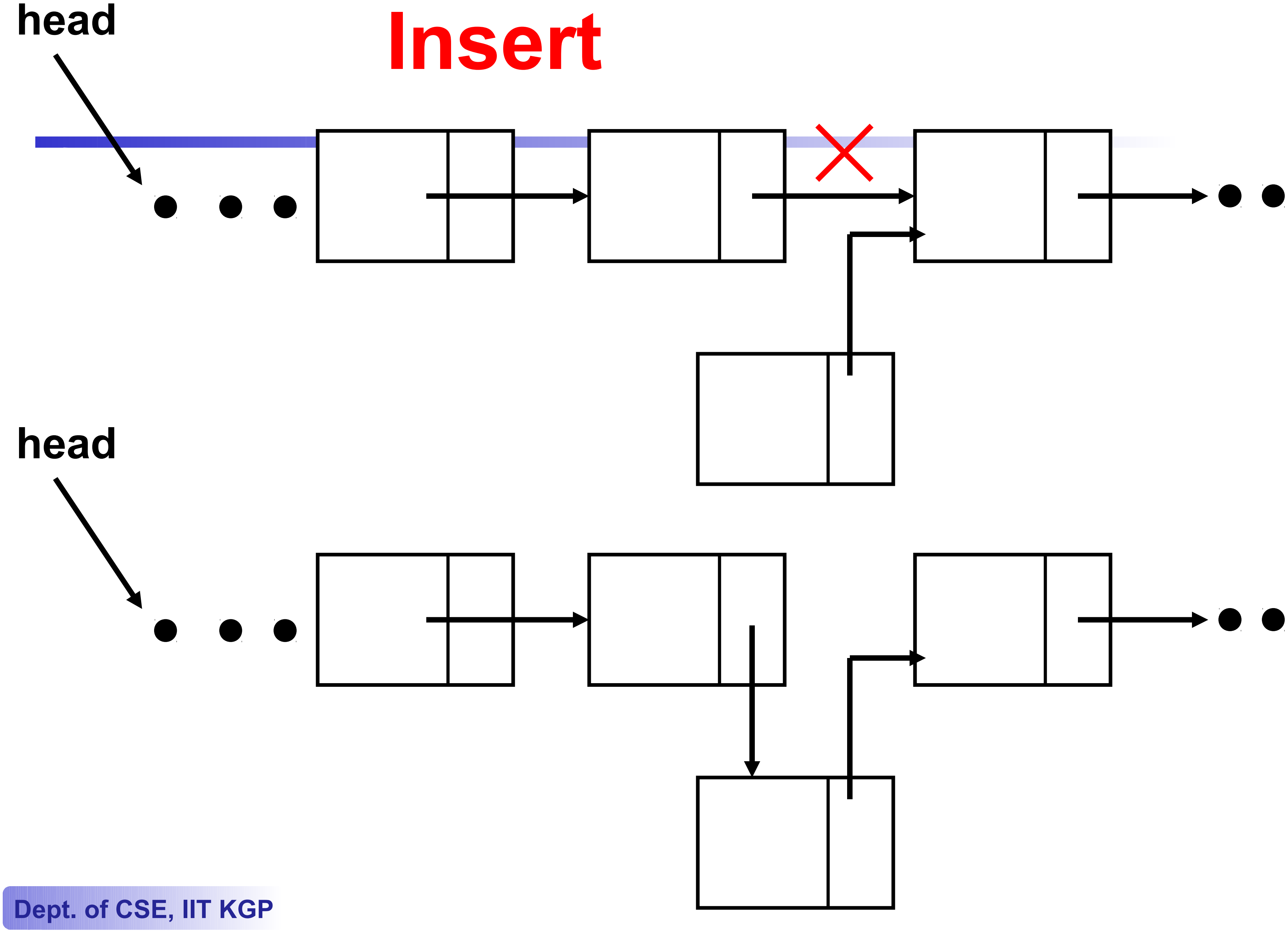
Initialization



Write a function that initializes LIST

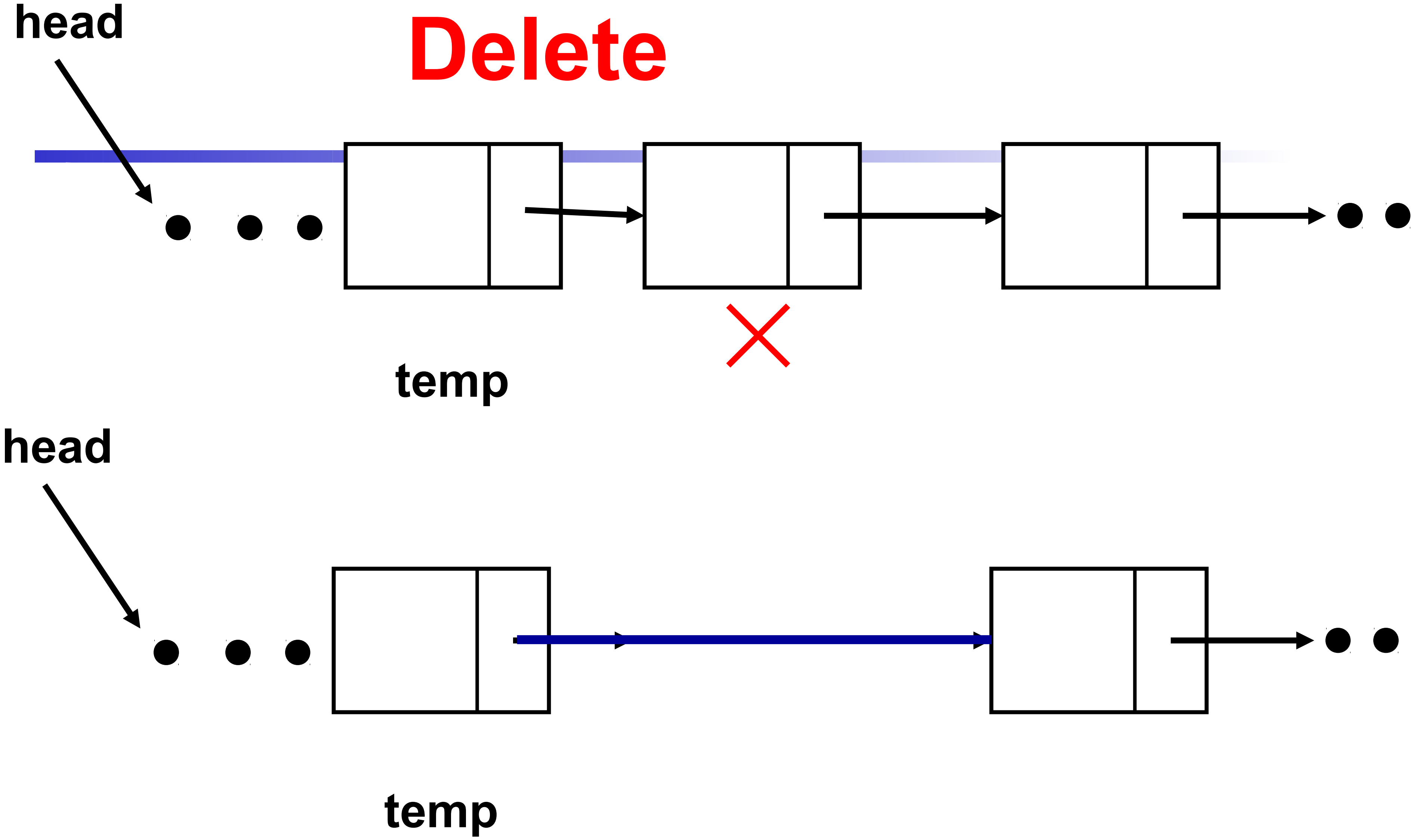
```
typedef struct list {
        int data;
        struct list *next;
} ELEMENT;

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

Insert



```
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;
}
```

Delete



```
ELEMENT* Delete(data *head, int position) {
    int i=0;data *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;
}
```

De

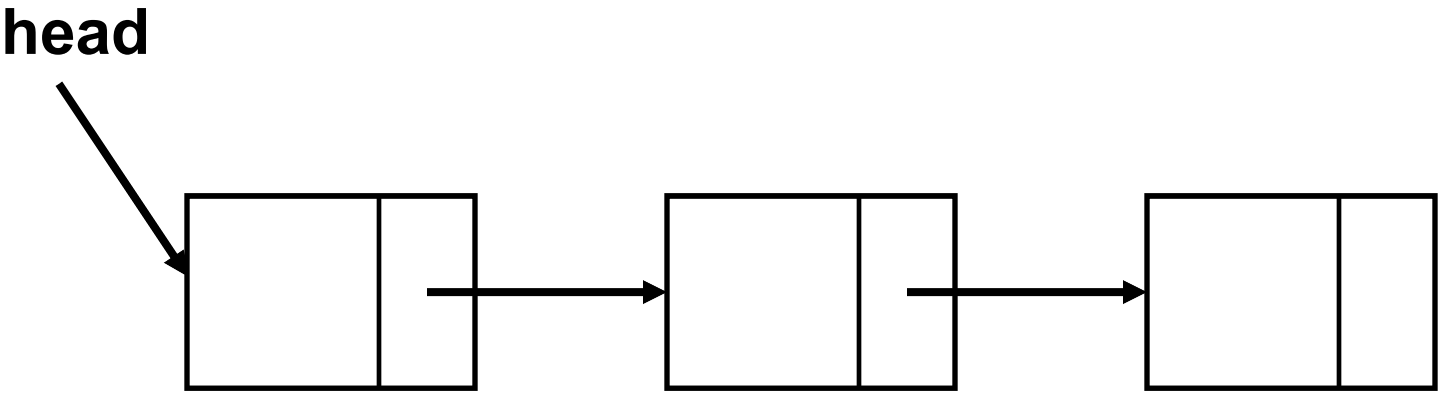
Searching a data element

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

Print the list backwards



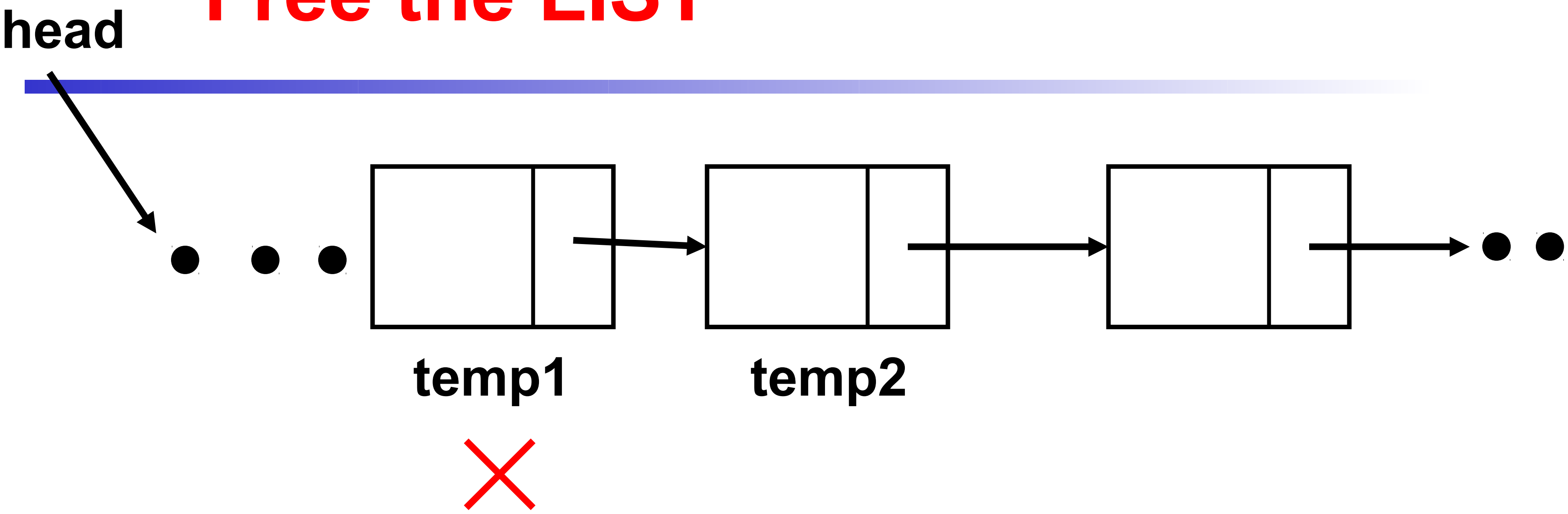
How can you when the links are in forward direction ?

Can you apply recursion?

Print the list backwards

```
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 elemen
   return;
}
```

Free the LIST

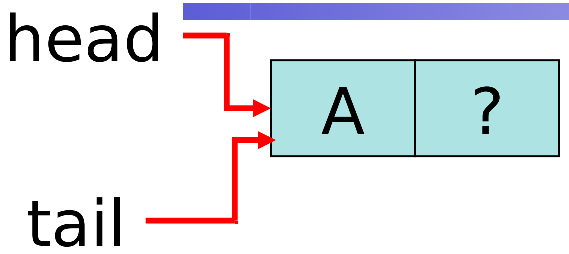


We can free temp1 only after we have retrieved the address of the next element (temp2) from temp1.

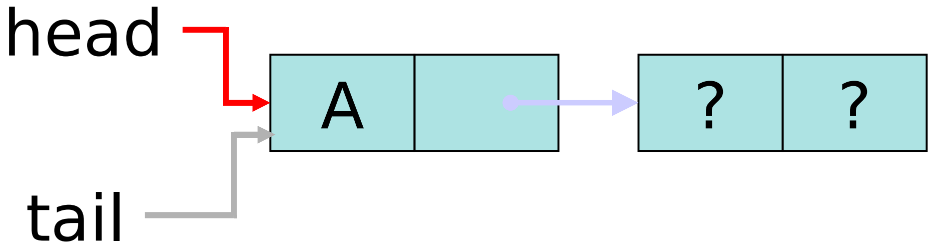
Free the list

```
void Free(ELEMENT *head)  {
    ELEMENT *temp1, *temp2;
    temp1 = head;
    while(temp1 != NULL)  /*boundary condition check*/
    {
          temp2 = temp1 -> next;
          free(temp1);
          temp1 = temp2;
    }
}
```

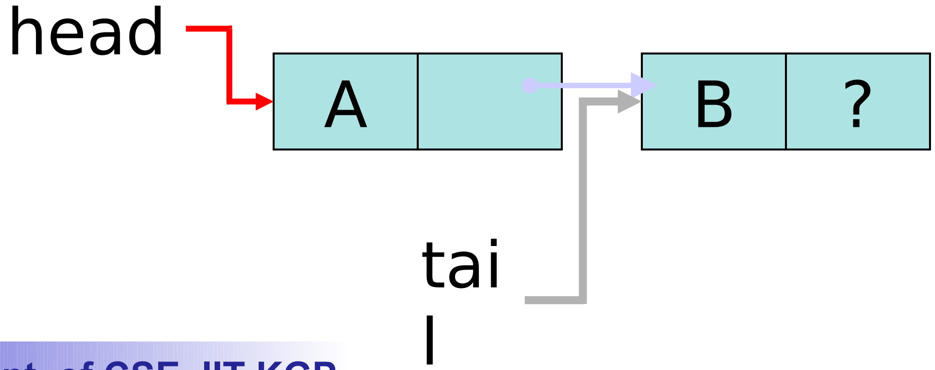
1. A one-element list



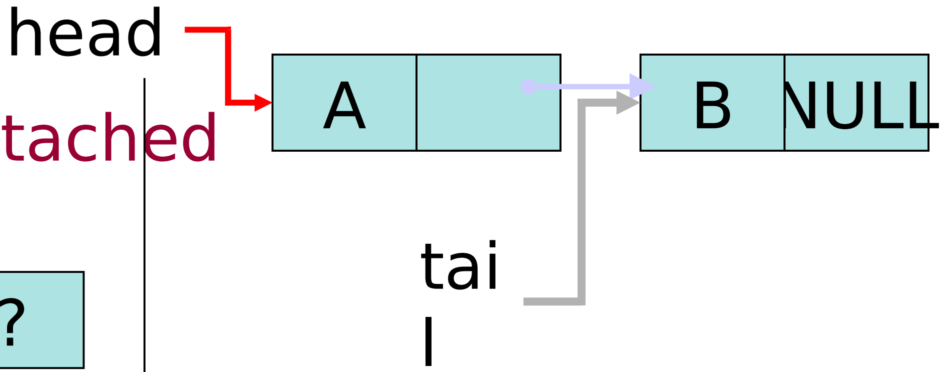
2. A second element is attached



3. Updating the tail



4. after assigning NULL



/* Count a list recursively */

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

/* Count a list iteratively */

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

/* Print a List */

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

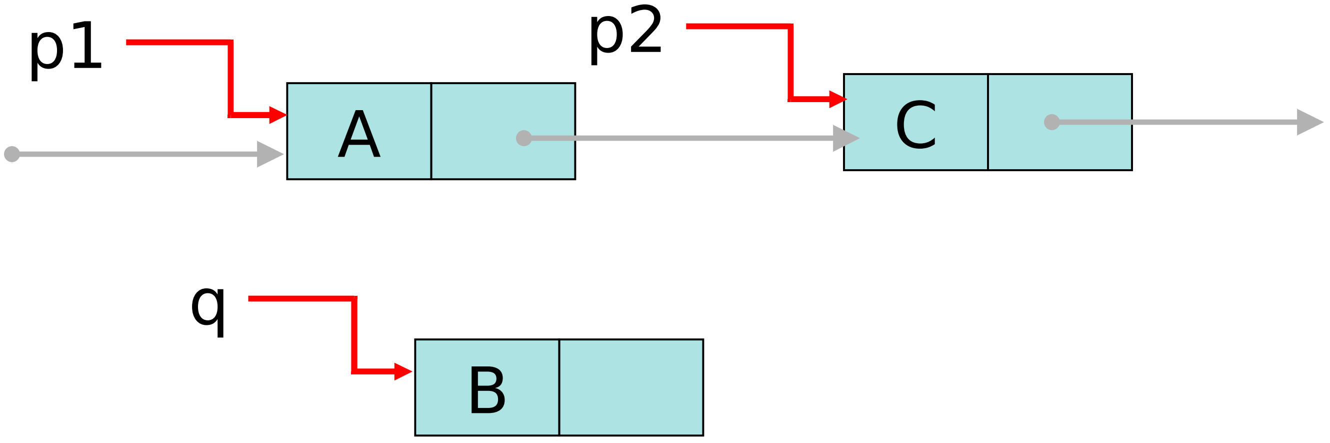
/* Concatenate two Lists */

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

Insertion

- Insertion in a list takes a fixed amount of time once the position in the list is found.

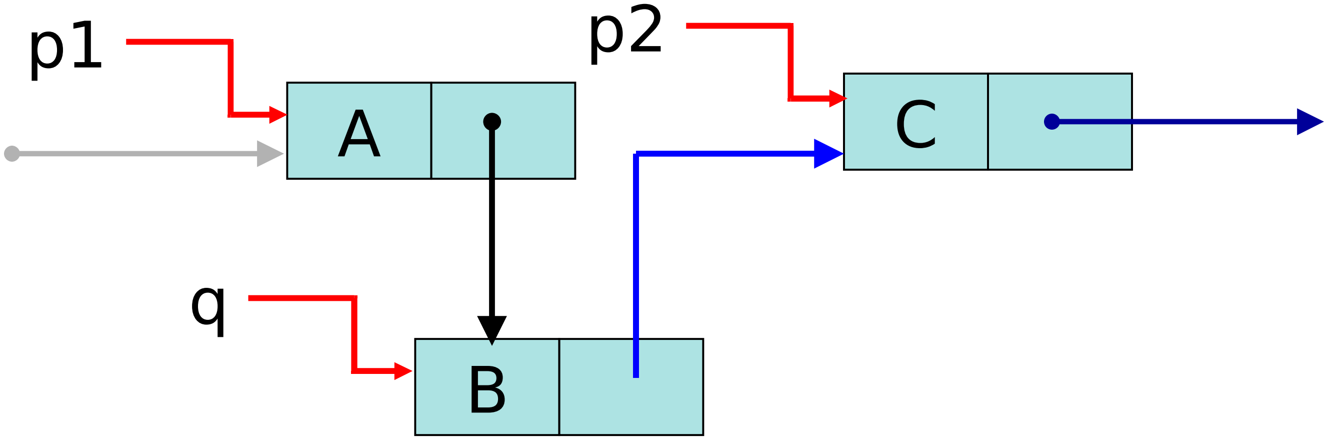
Before Insertion



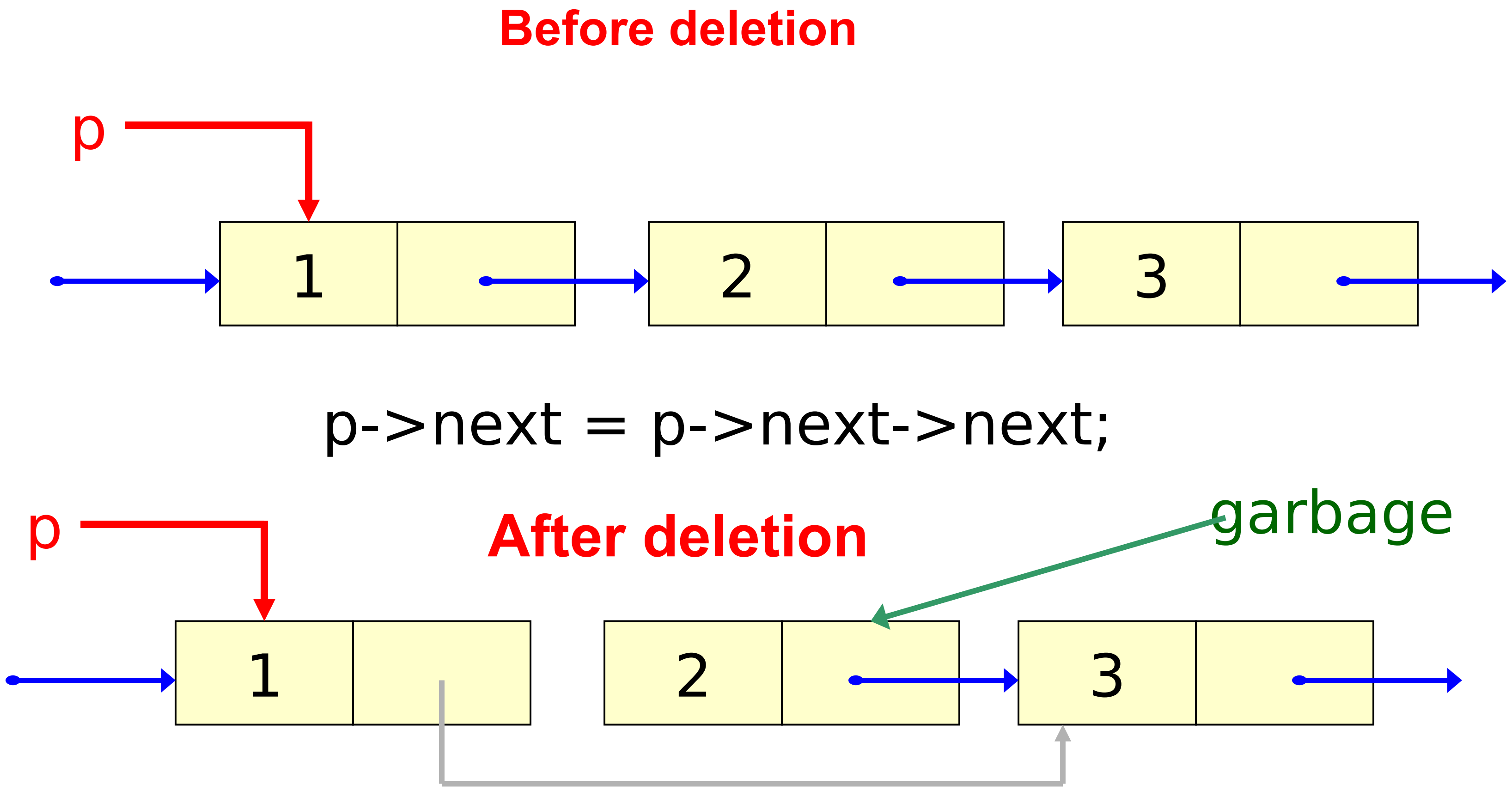
Insertion

```
/* Inserting an element in a linked list. */
void insert (LINK p1, LINK p2, LINK q) {
            p1->next = q;
            q->next = p2;
}
```

After Insertion



Deletion

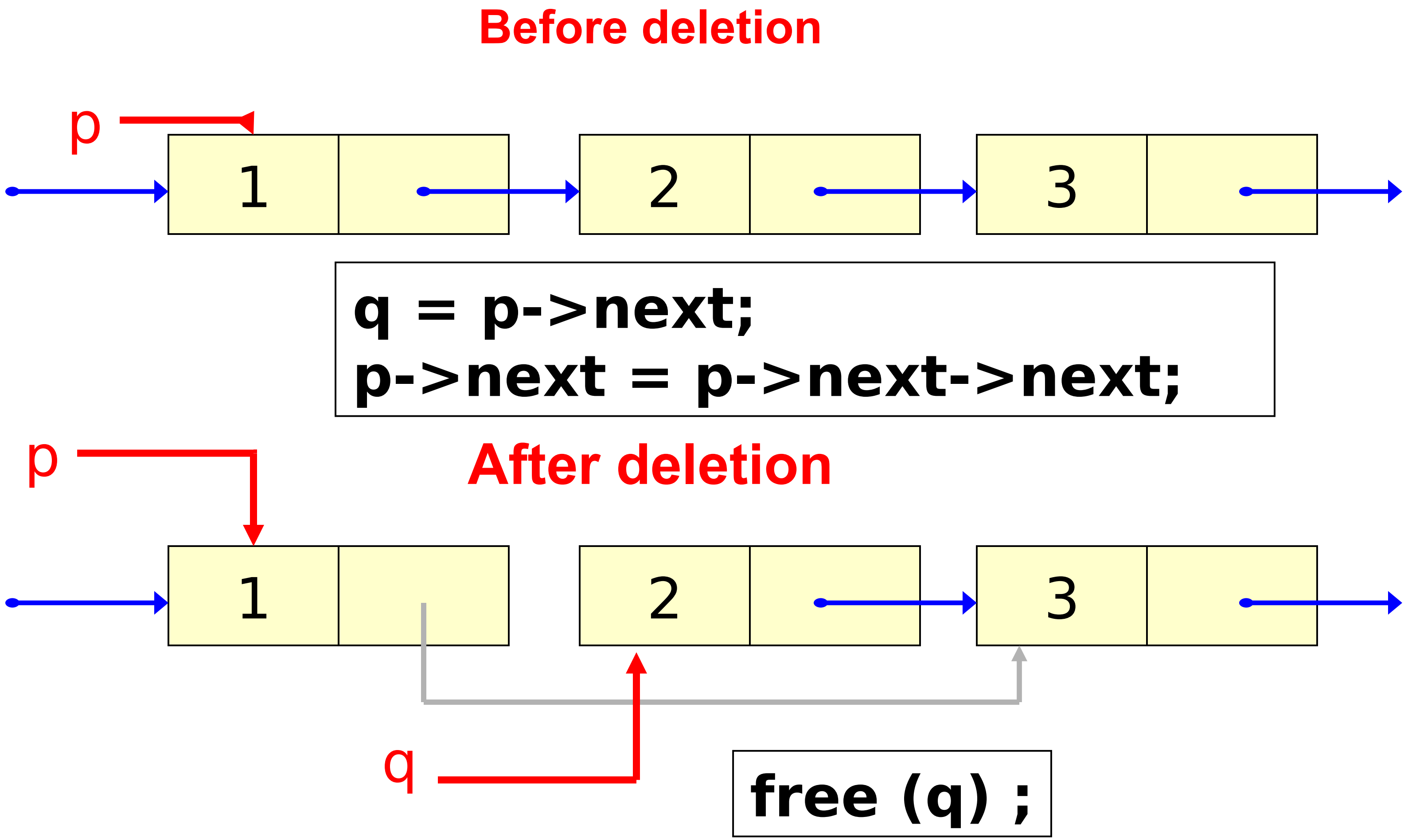


Before deletion

```
p->next = p->next->next;
```

After deletion

Deletion



Before deletion

```
q = p->next;
p->next = p->next->next;
```

After deletion

```
free (q) ;
```

Delete a list and free memory

```
/* Recursive deletion of a list */
void delete_list (LINK head)  {
        if (head != NULL)   {
                delete_list (head->next) ;
                free (head) ; /* Release storage */
}
```