Programming & Data Structure

CS 11002

Partha Bhowmick

http://cse.iitkgp.ac.in/~pb

CSE Department
IIT Kharagpur

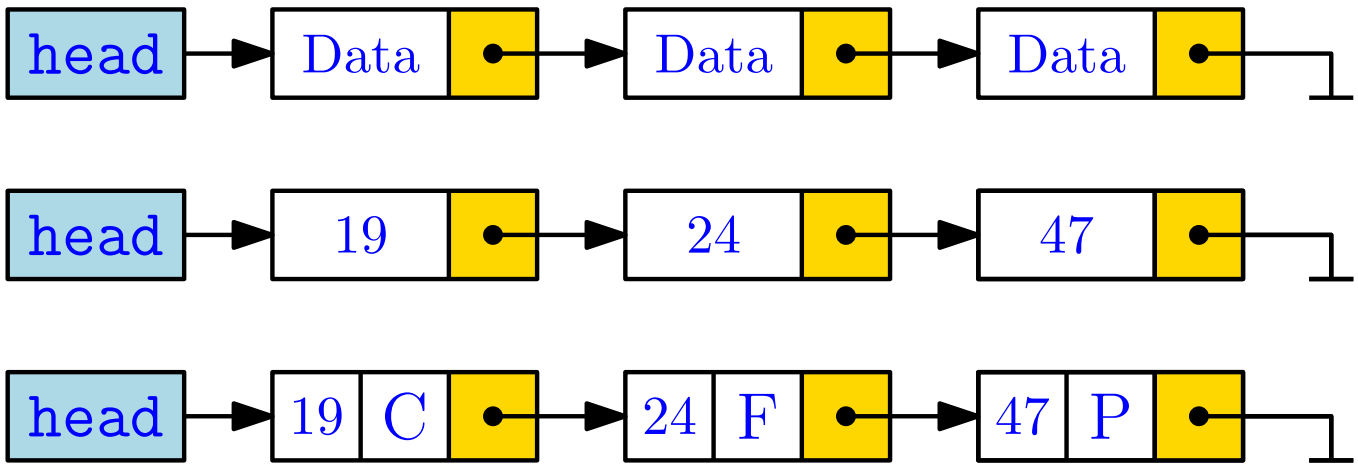
Spring 2012-2013

Linked List

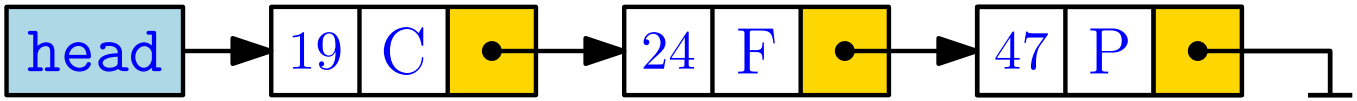
Introduction (1)

Definition

A **linked list** is a *data structure* consisting of a *sequence of nodes (records)*, which are connected by pointers in succession.



Introduction



Properties

- Successive nodes connected by *pointers*.
- Last node points to `NULL`.
- Nodes can *dynamically change* (grow/ shrink/ change in content) during execution.
- Unlike array, it can be made just as long as required, and hence *space-efficient*.
- Admits efficient *insertion / deletion* of nodes.

Introduction (3)

```
#include <stdio.h>
struct node {
  int data;
  struct node *next; //self-reference!
};
int main() { // selfRef2.c
  struct node n;
  n.next = &n; //head
  printf("&n: %p\tn.next: %p\n", &n, n.next);
  return 0; }
```

Introduction

Output

```
$ cc -Wall selfRef2.c
$ a.out
&n:  0xbff52f70 n.next:  0xbff52f70
```

Introduction

Basic Operations on a List

- Creating a list
- Traversing the list
- Inserting an item in the list
- Deleting an item from the list
- Concatenating two lists into one

Introduction

List is an Abstract Data Type

Q: What is an abstract data type (ADT)?

A: It is a data type defined by the user, and typically more complex than simple data types like `int`, `float`, etc.

Q: Why abstract?

A: Implementation details are hidden. To do some operation on the list, e.g., insertion, we just call a function. Details of implementation or the insert function are not required.

Creation (1)

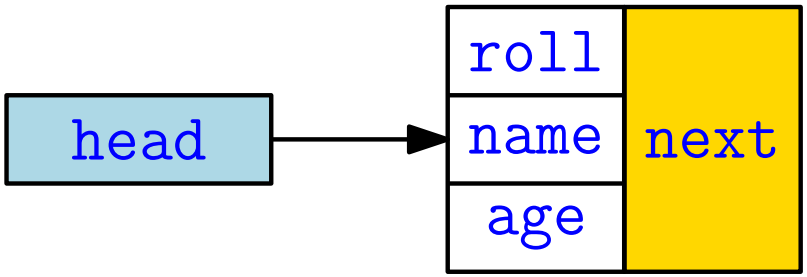
Consider the following node structure.

```
struct stud {
  int   roll;
  char  name[25];
  int   age;
  struct stud *next;};

/* A user-defined data type called node */
typedef struct stud node;
node *head;
```

Creation (2)

To start with, we have to create the *first node* and make `head` point to it.

```
head = (node *) malloc(sizeof(node));
```



Creation (3)

```
node *create_list(){
  int k, n; node *p, *head;
  printf("\n How many elements to enter?");
   scanf("%d", &n);
  for(k=0; k<n; k++){
    if(k == 0){
      head = (node *)malloc(sizeof(node));
      p = head;}
    else{
      p->next = (node *) malloc(sizeof(node));
      p = p->next;}
    scanf("%d %s %d", &p->roll, p->name, &p->age);
  } //end for
  p->next = NULL;
  return(head);}
```

Creation

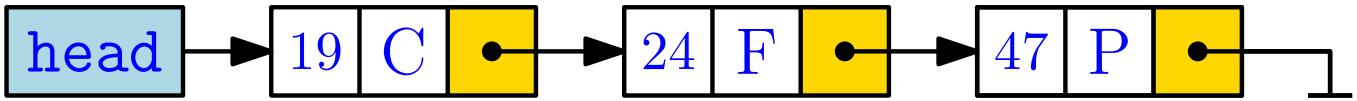
To be called from `main()` as follows.

```
int main(){
  node *head;
  ...
  head = create_list();
  ...
}
```

Creation

How to display or print

1. Follow the pointers, starting from `head`.
2. Display the contents of the nodes as they are traversed.
3. Stop when the next pointer points to `NULL`.



Creation

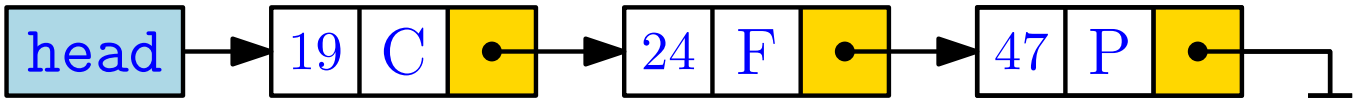

```
void display (node *head){
  int count = 1;
  node *p;

  p = head;
  while(p != NULL){
    printf("\nNode %d: %d %s %d", count,
           p->roll, p->name, p->age);
    count++;
    p = p->next;}
  printf("\n");
}
```

Creation

To be called from `main()` as follows.

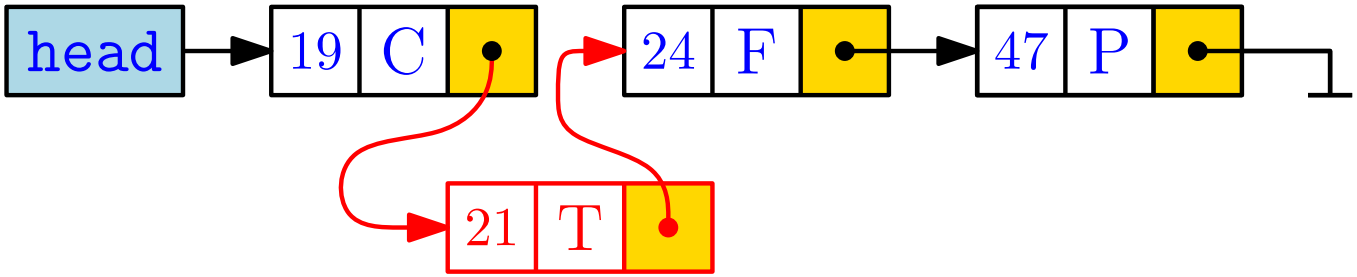
```
int main(){
  node *head;
  ...
  head = create_list();
  display(head);
  ...
}
```

Insertion

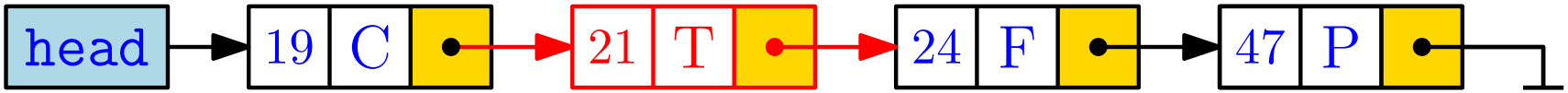


node to be inserted

Insertion (2)



Insertion



Insertion

How to insert

The problem is to insert a new node *before a specified node*, so that *key* (e.g., `roll`) of the new node is smaller than that of the specified node.

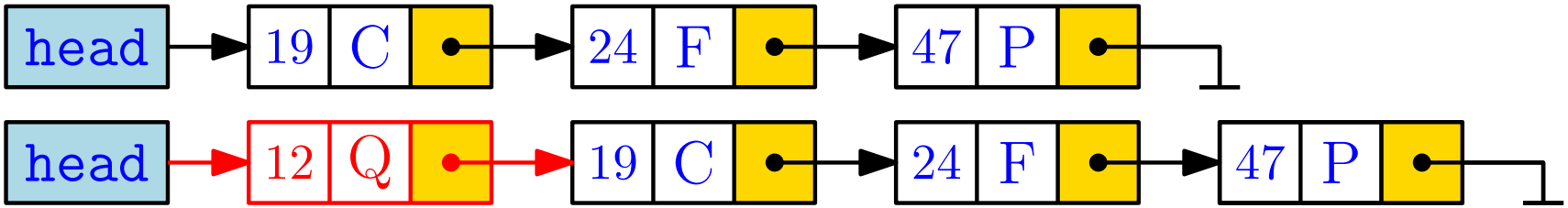
3 cases

1. New node is added at the beginning
2. New node is added at the end
3. New node is added somewhere in the middle

Insertion

New node is added at the beginning

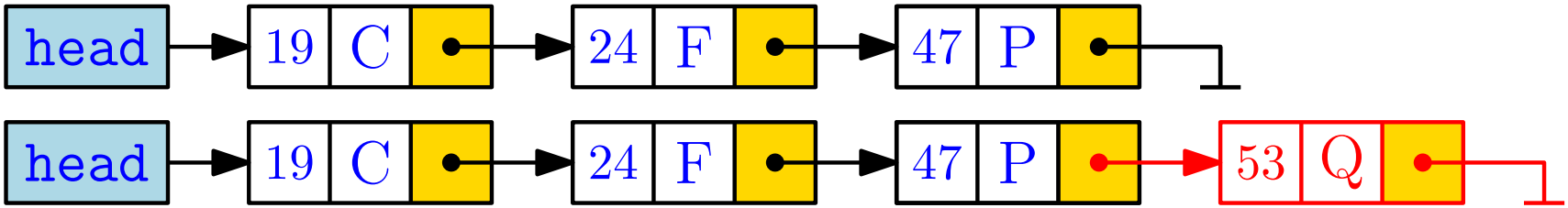
Only one `next` pointer needs to be modified. `head` is made to point to the new node. New node points to the previously first element.



Insertion

New node is added at the end

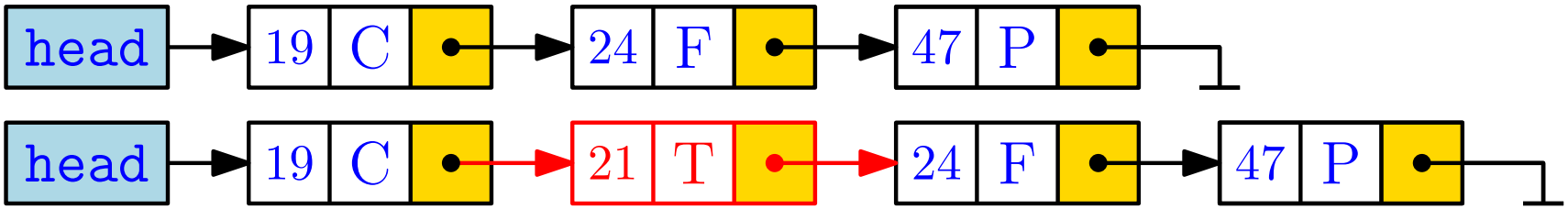
Two `next` pointers need to be modified. Last node now points to the new node. New node points to `NULL`.



Insertion

New node is added somewhere in the middle

Two `next` pointers need to be modified. Previous node now points to the new node. New node points to the next node.



Insertion


```
void insert (node **head){
  int  k = 0, rno;
  node *p, *q, *new;

  new = (node *) malloc(sizeof(node));

  printf("\nData to be inserted: ");
  scanf("%d %s %d", &new->roll, new->name, &new->age);
  printf("\nInsert before roll (-ve for end):");
  scanf("%d", &rno);

  p = *head;
```

Insertion


```
if (p->roll == rno){ /* At the beginning */
    new->next = p;
    *head = new; }
else{
  while((p != NULL) && (p->roll != rno)){
    q = p;
    p = p->next;}
  if(p == NULL){ /* At the end */
    q->next = new;
    new->next = NULL;}
  else if(p->roll  == rno){ /* In the middle */
    q->next = new;
    new->next = p;}
}
```

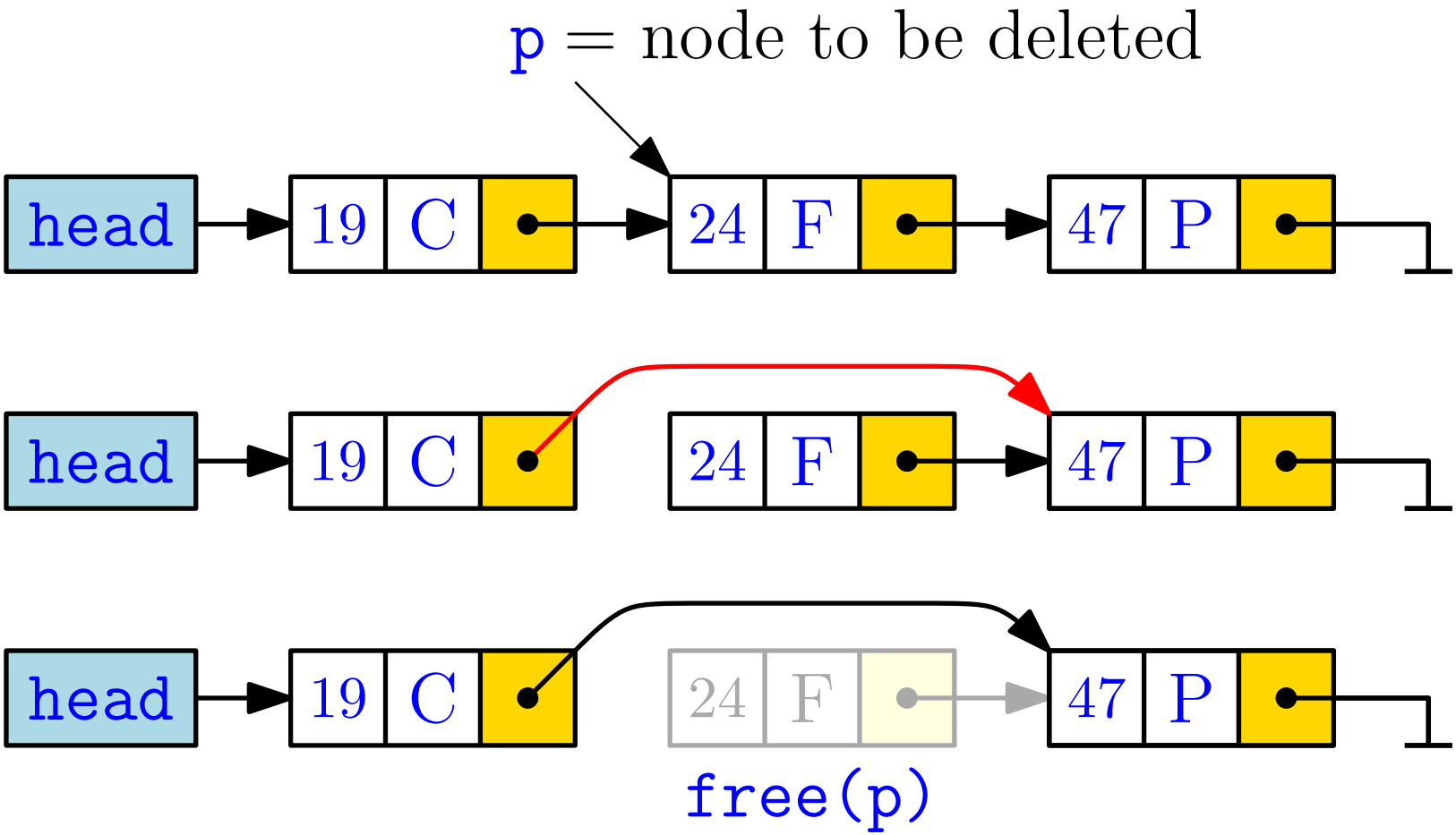
Insertion


```
}
```

To be called from `main()` as follows.

```
node *head;
...
insert(&head);
```

Delete


p = node to be deleted
head
19
C
24
F
47
P
head
19
C
24
F
47
P
head
19
C
24
F
47
P
free(p)

Delete

How to delete

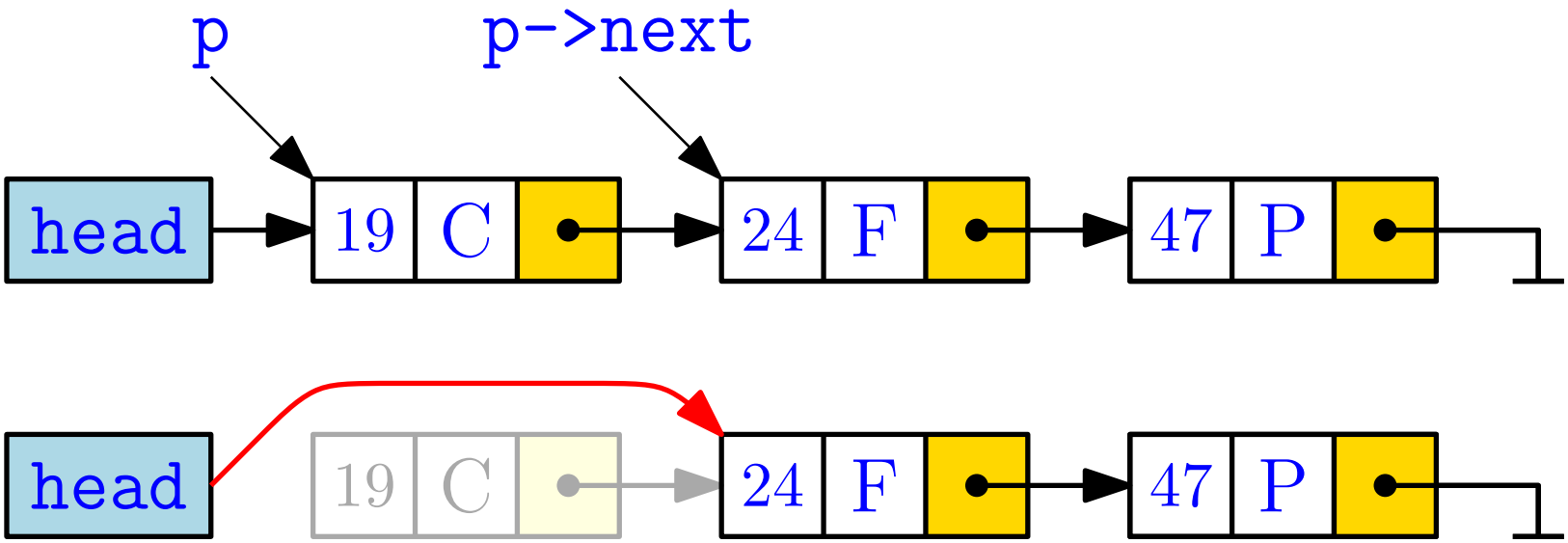
The problem is to delete a node whose *key* (e.g., `roll`) matches the specified value.

3 cases

1. Deleting the first node
2. Deleting the last node
3. Deleting an intermediate node

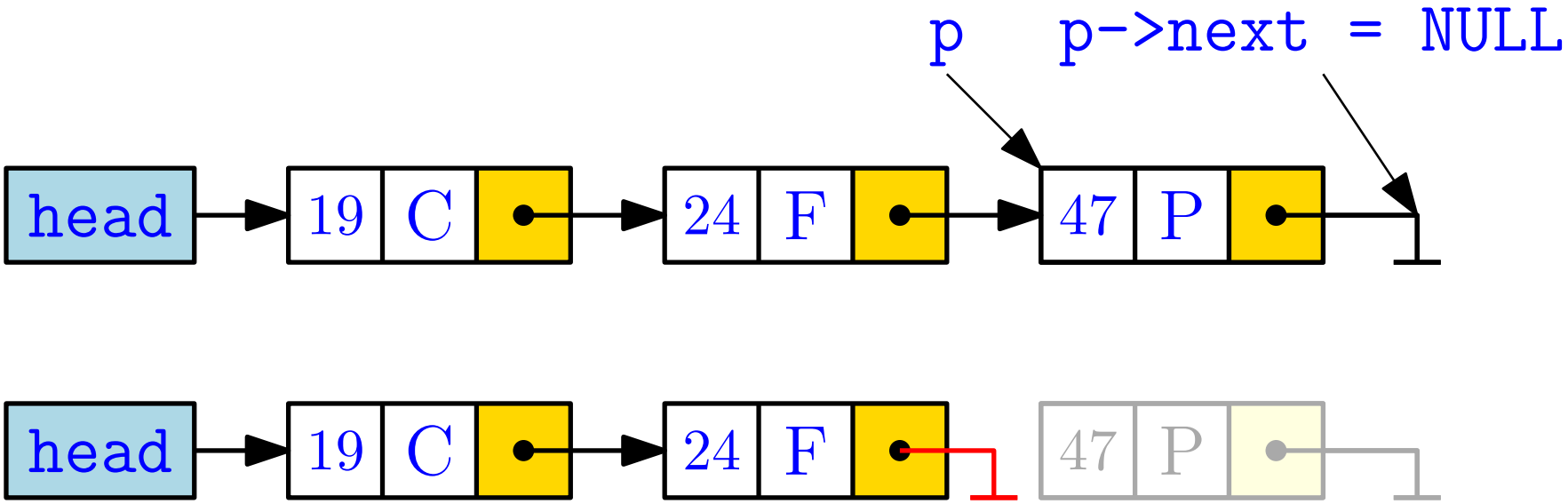
Delete

Deleting the first node



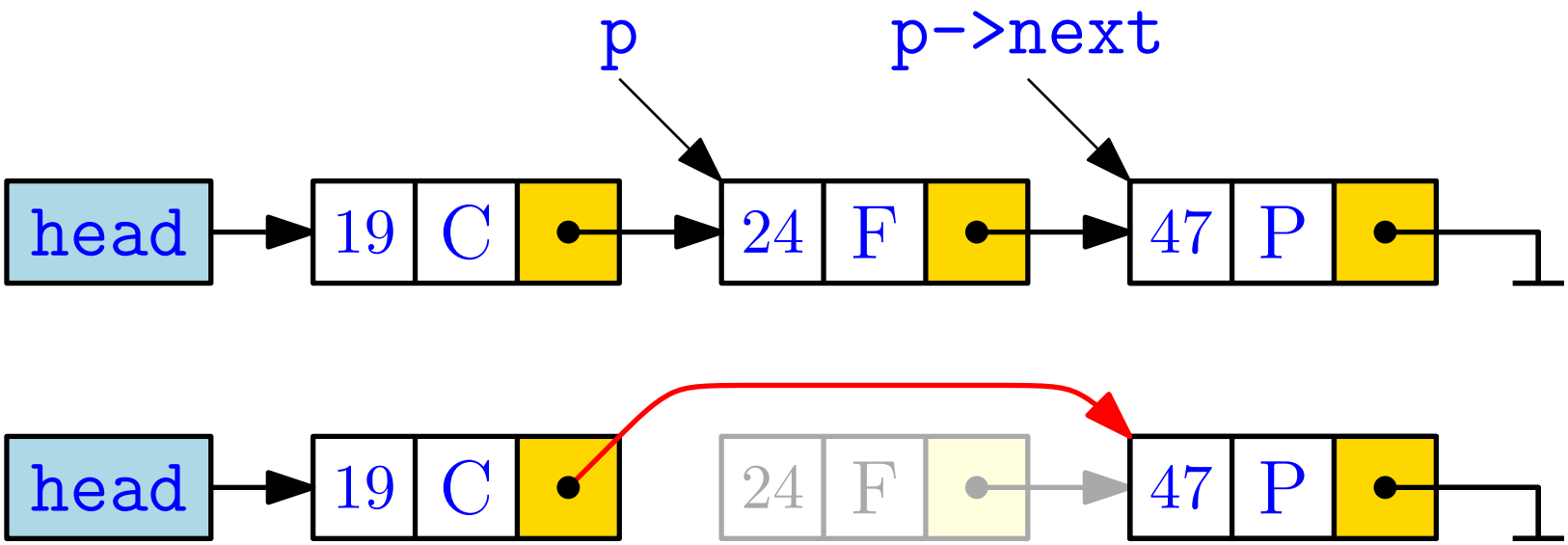
Delete

Deleting the last node



Delete

Deleting an intermediate node



Delete


```
void delete (node *head){
  int rno;
  node *p, *q;

  printf("\nDelete for roll :");
  scanf("%d", &rno);

  p = head;
  if(p->roll == rno){/* Delete the first element */
    head = p->next;
    free(p);}
  else{
    while((p != NULL) && (p->roll != rno)){
      q = p;
```

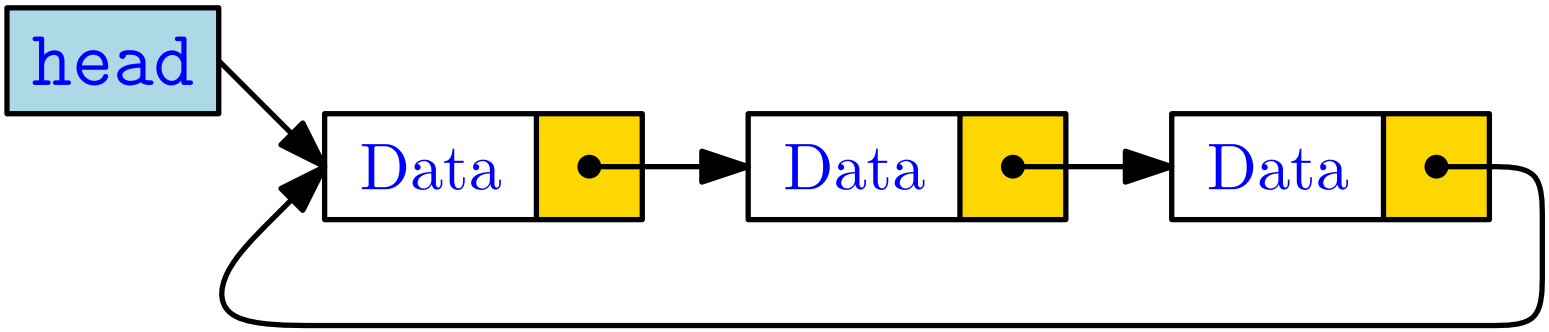
Delete


```
      p = p->next;}
    if (p == NULL) /* Element not found */
      printf("\nNo match - deletion failed");
    else if (p->roll == rno){ /* Delete */
      q->next=p->next;
      free(p);}
  }
}
```

To be called from `main()` as follows.

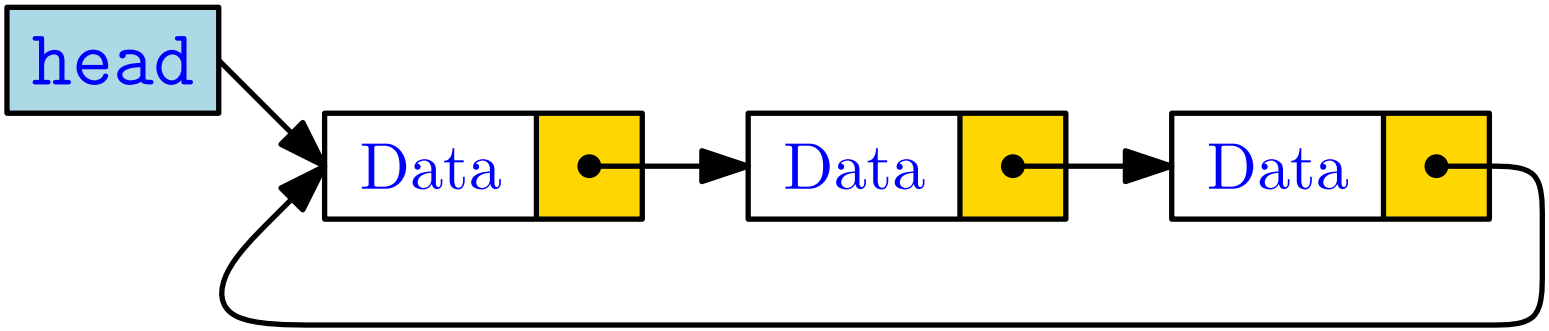
```
node *head;
...
delete(head);
```

Circularly Linked List (1)



Unlike *open or linear linked list*, the last node of *circularly linked list* points to its first node.

Circularly Linked List (2)



Advantages

We can visit any node from any node, e.g., from last node to first node in a single step. But in linear linked list it is not possible to go to previous nodes. And in doubly linked list, we will have to traverse all through to visit last to first node.

Circularly Linked List (3)

Disadvantages

- If proper care is not taken, then the problem of *infinite loop* can occur.
- It is not easy to *reverse* the linked list.
- Visit to the *previous node* cannot be done in a single step; we have to complete the entire circle by traversing through all the other nodes.
  Doubly linked list is of course better in this context.

Doubly Linked List (1)

There are two pointers: `head` pointing to the first node, and `tail` pointing to the last node of the list.

For each `node`, there are two pointers: `prev` pointing to the previous node and `next` pointing to the next node. The `prev` of first node and the `next` of last node are `NULL`, which shows the end of list on both sides.

Doubly Linked List (2)

Advantages

- We can traverse in both both *forward* and *backward directions*.
- Easy to *reverse* the linked list.
- We can visit any node from any node using *bidirectional pointers*, which is not possible in linear linked list.

Doubly Linked List (3)

Disadvantages

- Requires *more space* per node because of the extra pointer to previous node.
- Insertion and deletion take *more time* than linear linked list because more pointer operations are required than linear linked list.

Exercise (1)

1. Concatenate two given lists into one list.
   ```
   node *concatenate (node *head1, node *head2);
   ```
2. Insert an element in a linked list in sorted order. The function will be called for every element to be inserted.
   ```
   void insert-sorted (node *head, node *element);
   ```

Exercise (2)

3. Always insert elements at one end, and delete elements from the other end (first-in first-out or *FIFO*: QUEUE).

```
void insertQ (node *head, node
*element);
node *deleteQ (node *head); /* Return
the deleted node */
```