Data Structures and Algorithms (DSA) Course 7

Special Lists

Iulian Năstac

Recapitulation

Operations related to a linked list:

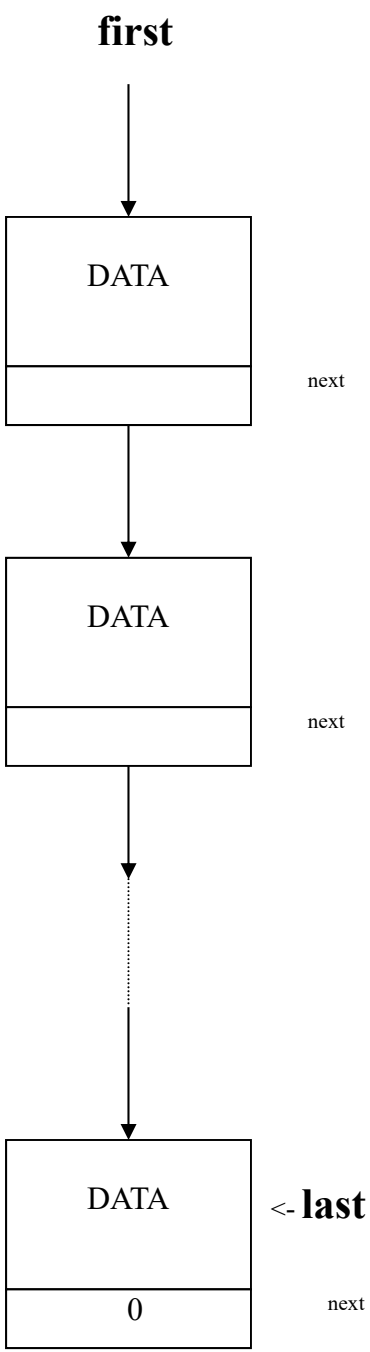
a) creation of a linked list;

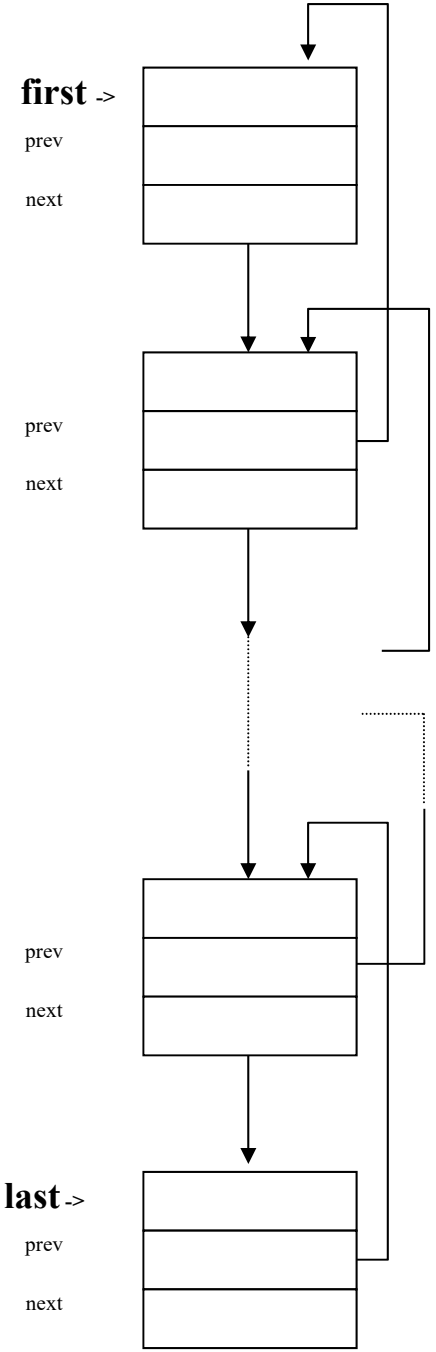
b) access to any node of the list;

c) inserting a node in a linked list;

d) deleting a node from a linked list;

e) deleting a linked list.


first
DATA
next
DATA
next
DATA
<- last
0
next


first ->
prev
next
prev
next
prev
next
last ->
prev
next

STACK
(Recapitulation)

- A **stack** could be a simple chained list, which is managed according to the LIFO principle (Last In First Out).

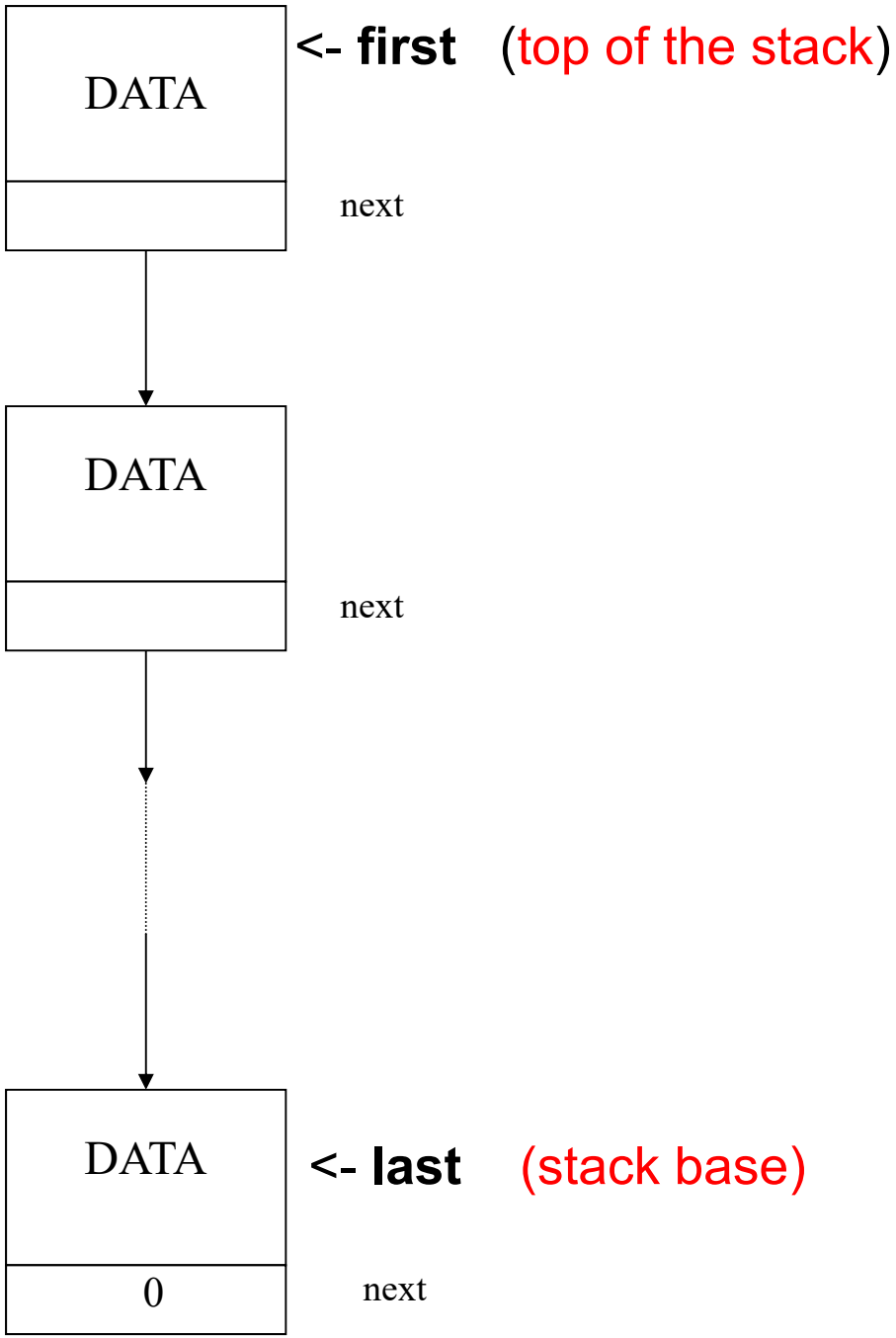
- According to this principle, the last node in the stack is the first one taken out. **The stack,** as a basic list, has two parts (to be indicated by two pointers):
  - **the base of the stack**
  - **the top of the stack**

On a stack we can define only three operations:

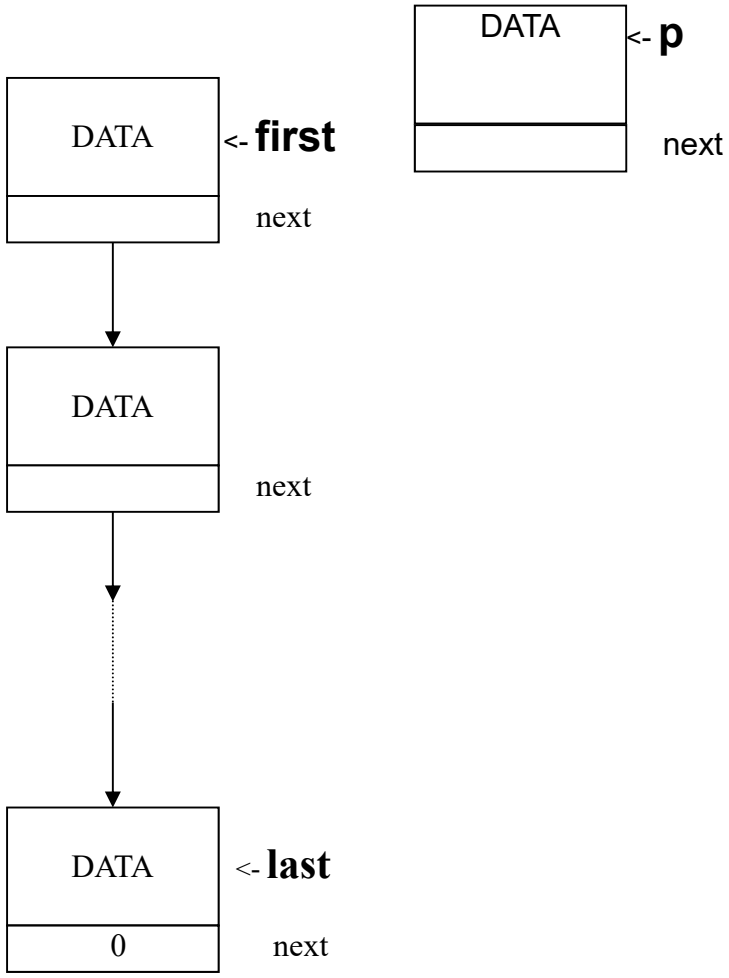
1. inserting an element in a stack (on its top) - **PUSH**

2. removing an element from a stack (the last element that was previously added) - **POP**

3. deleting a stack (**CLEAR**)

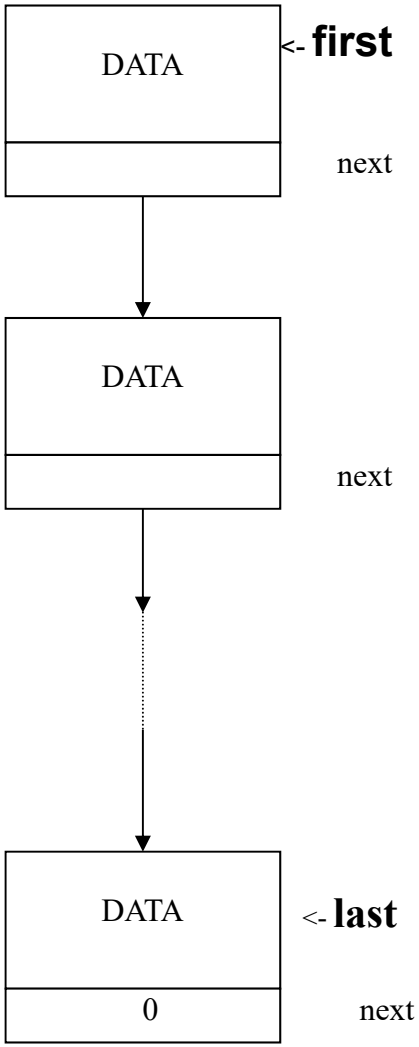

DATA
<- first (top of the stack)
next
DATA
next
DATA
<- last (stack base)
0
next

PUSH (insertion before the first node)




```
…
p->next = first;
first=p;
….
```

POP (deleting the first node of the simple linked list)



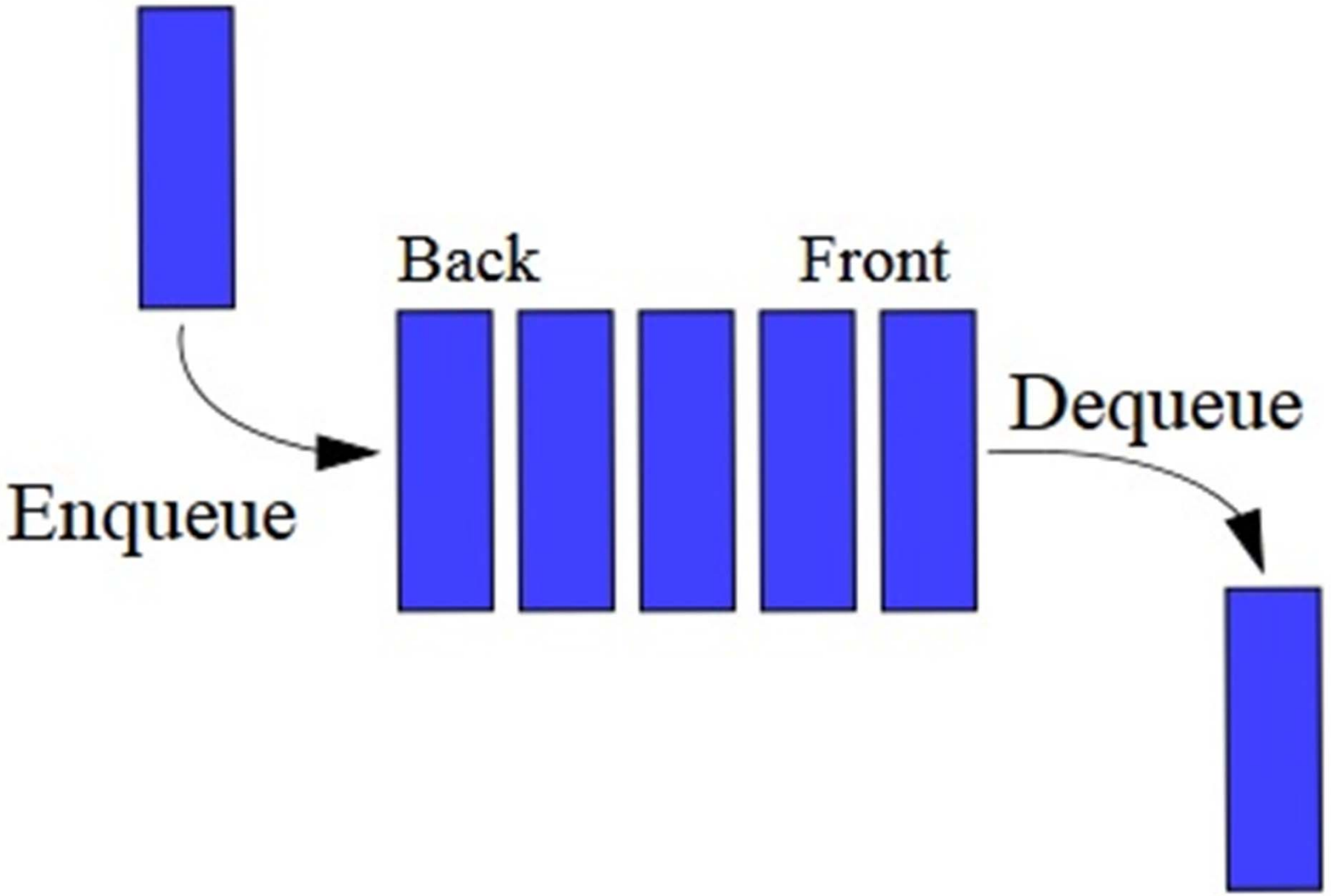

```
...

p = first;
first = p -> next;

elibnod(p);
...
```

Queue (Recapitulation)

- A queue is a collection of nodes (that are kept in order) and the principal (or only) operations on this collection are the addition of entities to the rear terminal position, known as **enqueue**, and removal of entities from the front terminal position, known as **dequeue**. This makes the queue a **First-In-First-Out (FIFO)** data structure.
- A simple linked list can be managed according to the FIFO principle. The edges of the simple linked list are the front (for dequeue) and the back (for enqueue).

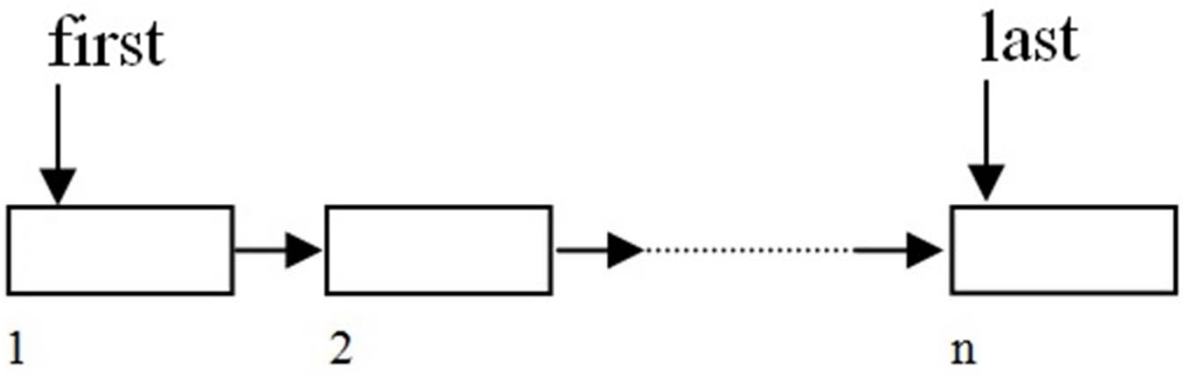

Back
Front
Dequeue
Enqueue

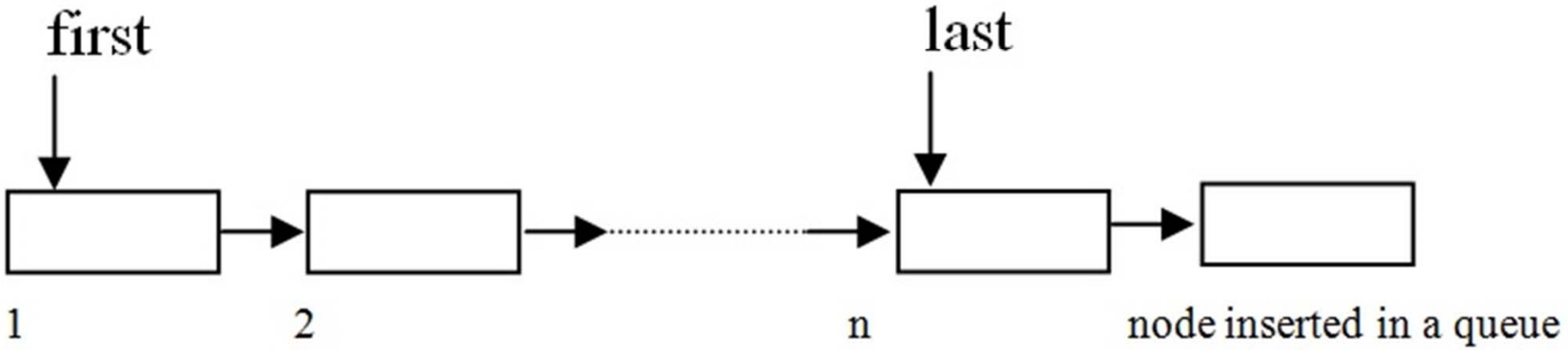
On a **queue** we can define only two operations:

a. inserting an element in a queue (at its back) - ENQUEUE;

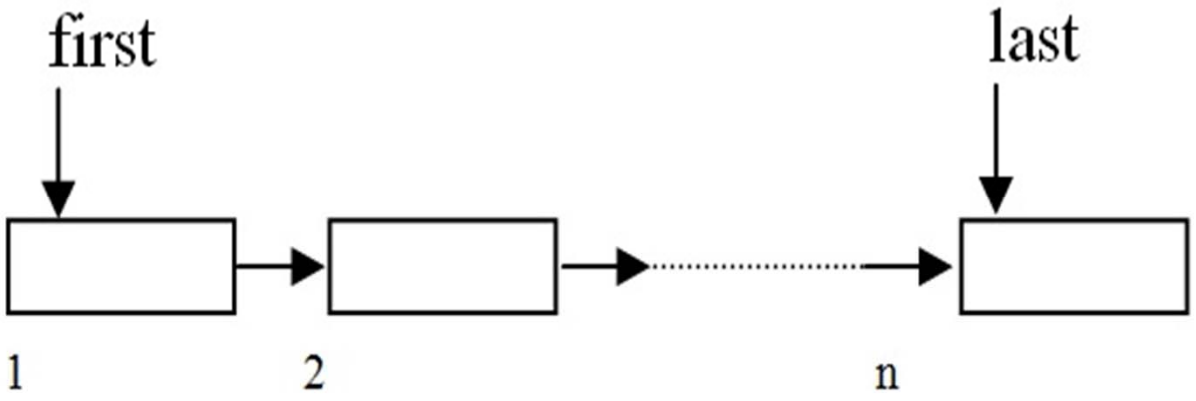
b. extracting an element from a queue (from its front) - DEQUEUE;

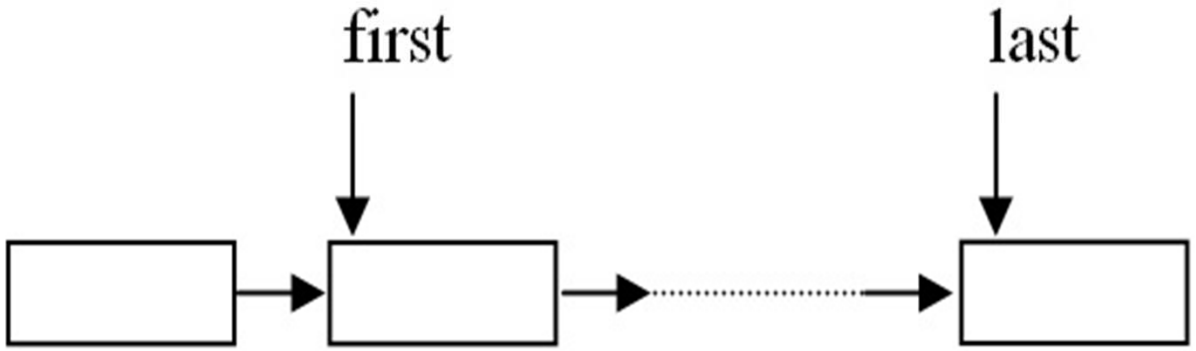
The operation of inserting a node in a queue:





The operation of extracting a node from the queue





the node
extracted from the queue

Circular simple linked list

A simple linked list contains:

first – the pointer to the node that has no predecessor;

last – the pointer to the node that has no successor.

We know that: **last -> next = 0**;

But if **last -> next = first** , then we can obtain a **circular linked list**.

Remarks:

- In a circular list all the nodes are equivalent.

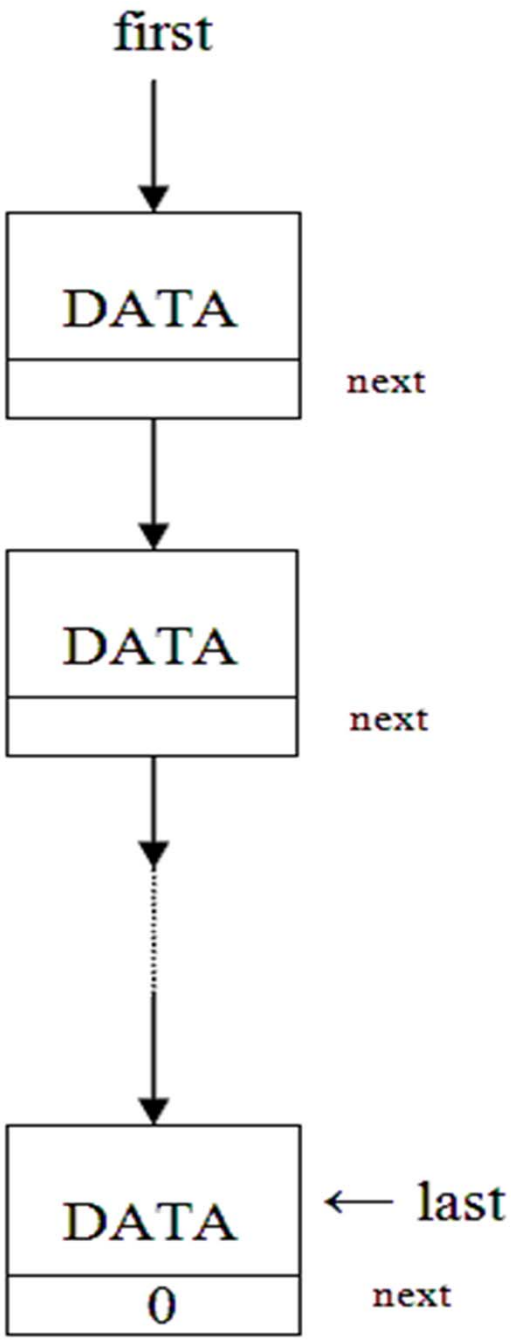
- Each node has a successor and predecessor.

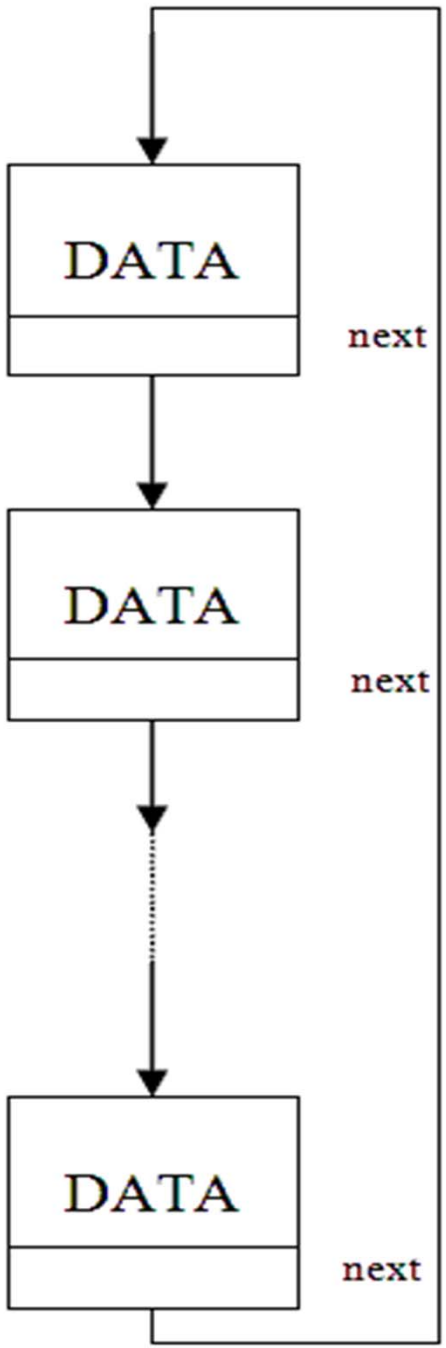
- In such a list we do not have ends; therefore the edges variables (first and last) are not necessary.

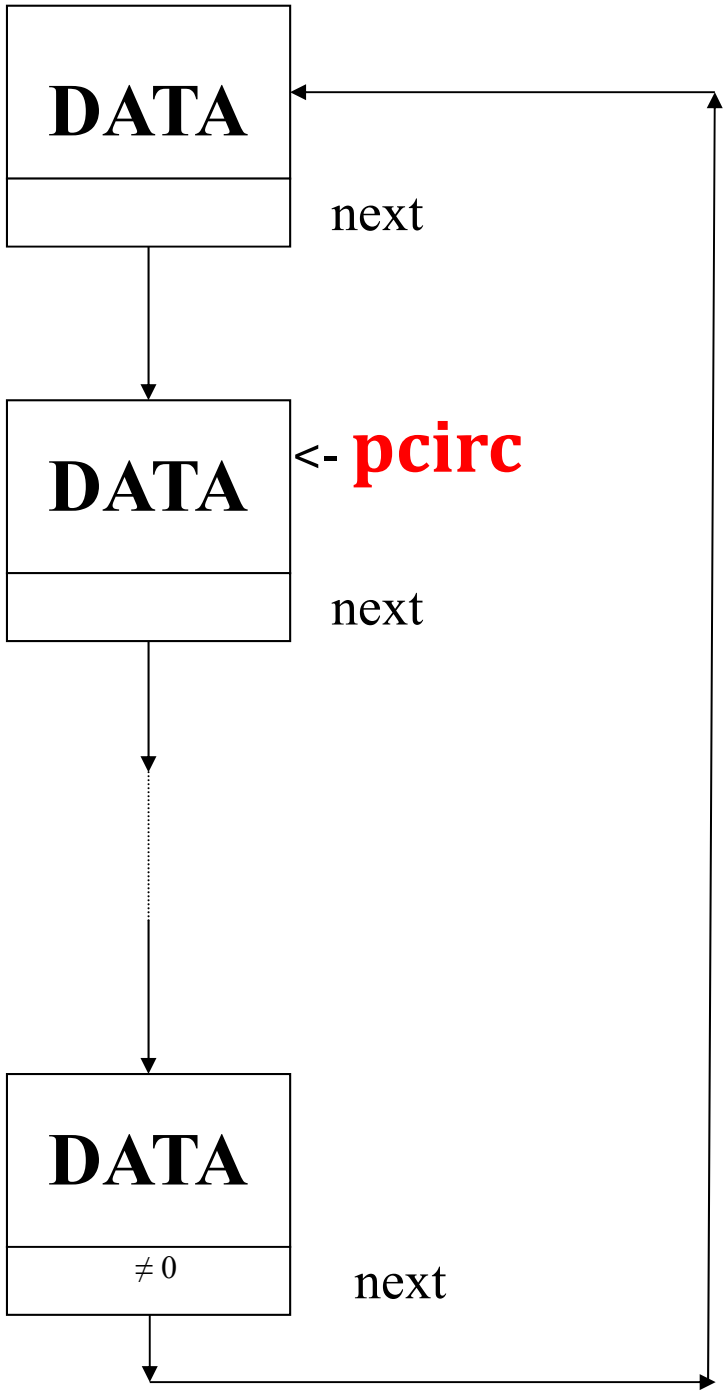
For the management of the nodes, it is used a global variable (denoted **pcirc**) that points to an arbitrary node of the list:

NOD *pcirc;

where NOD is the common type of the list nodes.


first
DATA
next
DATA
next
DATA
last
0
next


DATA
next
DATA
next
DATA
next


DATA
next
DATA
<- pcirc
next
DATA
≠ 0
next

The circular lists have various applications:

- operations with integers having a large number of digits;
- polynomial operations with one or more variables;
- dynamic memory allocation.

Notes 1:

- The functions for dynamic allocation (**malloc** and **free**) use a memory called **heap**, which is somehow organized as a circular list.

- When we use the function **malloc**, the circular list nodes is searched until it finds a continuous memory with the size at least equal to that required by the function call. If the obtained free zone is large, it is divided and the unused part is chained with the other blocks of the circular list.

Notes 2:

- The **free** function frees a memory zone that is then inserted in such circular list.

- Use **free** function to release a memory as soon as you no longer need the data in it.

Operations related to a circular linked list:

1) creation of a circular linked list;
2) access to a node of the list;
3) inserting a node in a circular linked list;
4) deleting a node from a circular linked list;
5) deleting a circular linked list.

Creation of a circular linked list

We can use the functions for dynamic allocation, such as:
incnod and **elibnod**.

Required steps are as follows:

1) At first, the circular list is empty (**pcirc** = 0).

2) A memory area is allocated (with **malloc**) in the heap memory for the current node.

3) Are there data to upload them in the current node **p**?

- NO → returns from function (after using **elibnod(p)**);
- YES → loading node with current data (by using **incnod(p)**) and jump to step 4;

4) If the list:

- is empty:

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

- is not empty :

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

Current node is inserted after the one pointed by **pcirc**.

5) **pcirc** will indicate to the new node, which was inserted

```
pcirc = p;
```

6) Jump to step 2.

Access to a node of a circular linked list

We define an **access** function that searches a node, by using a numerical key, and returns one of the following values:

- a pointer to the searched node;
- **0** if there is no such a node.

We consider the following structure type:

```
typedef struct nod
                    { <statements>;
                      type key; /* like integer */
                      struct  nod *next;
                    } NOD;
```

A possible solution:

```
NOD * access (int c)
/ * Search a node with a key that is similar with function parameter c.
- Returns the pointer to that node, or 0 if there is no such node * /
{       extern NOD *pcirc;
        NOD * p;
        p = pcirc;
        if (p == 0) return 0;  /* Empty list */
        do
                {
                  if (p-> key == c) return p;
                  p = p->next;
                } while (p! = pcirc);
        return 0;
}
```

+ Access to a node by counting!

…

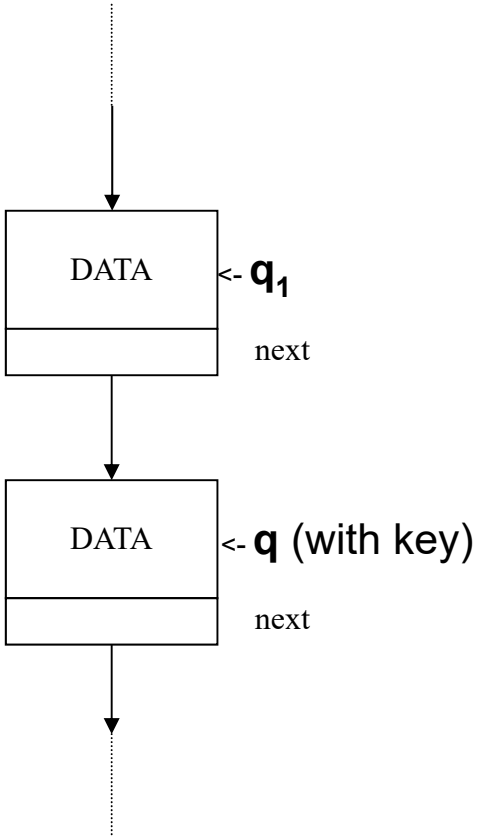
Inserting a node in a circular linked list

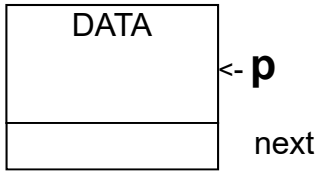
Cases (for which we will present the corresponding flow charts):

1. Inserting before a node specified by a key

2. Inserting after a node specified by a key

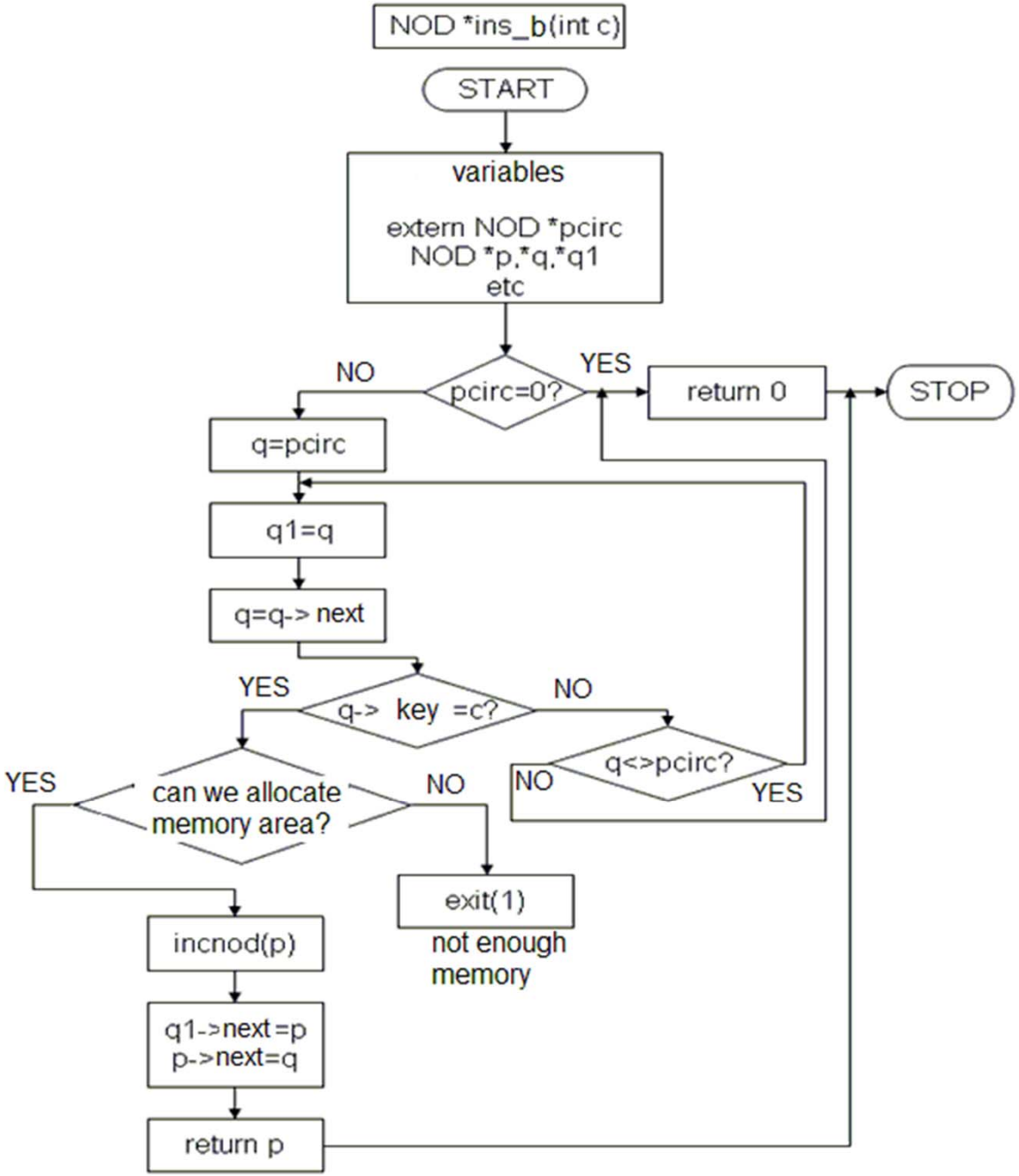
insertion before a node specified by a key



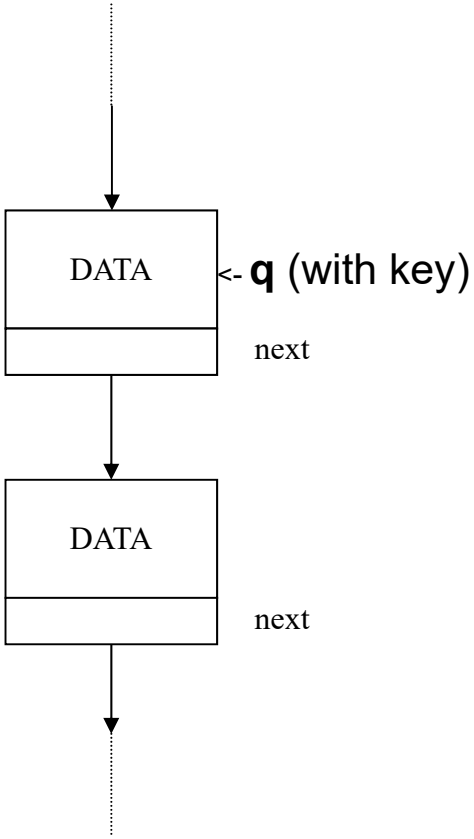


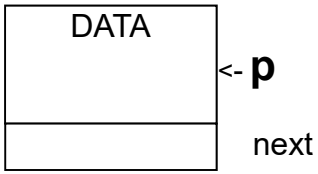

```
…
p->next=q;
q1->next=p;
…
```

Inserting before a node specified by a key



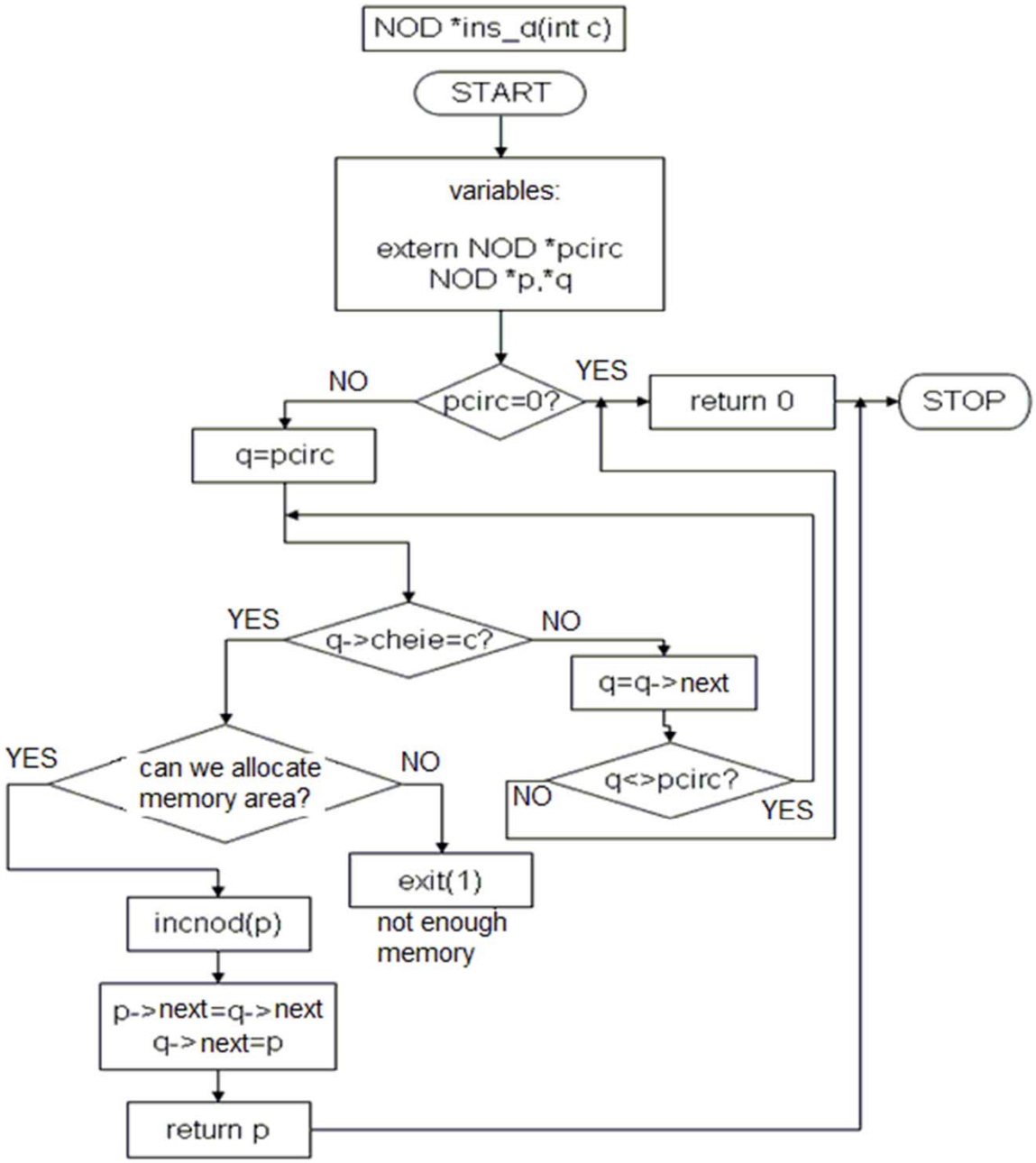
insertion after a node specified by a key






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

Inserting after a node specified by a key



+ Inserting a node by counting!

...

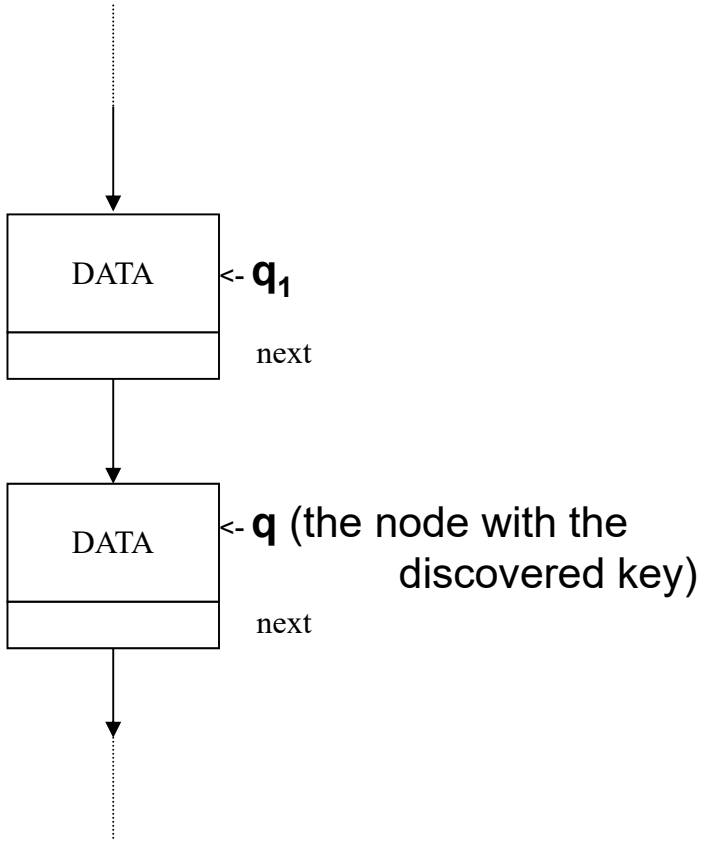
Deleting a node from a circular linked list

For a function that deletes, from a circular list, a node specified by a key, we can agree about the following procedures:

- if **pcirc** indicates to the node that has to be deleted, then, after deleting, **pcirc** will point to the previous node.

- if the list is empty, then **pcirc** = 0.

deleting a specified node with a key




```
...

q1 ->next = q -> next;

elibnod(q);
...
```

Notes:

- The erase function is similar to

  NOD *ins_b(int c)

- Deleting a node can be done by specifying its position to a **landmark** (which we will denote by **rep**).

```
typedef struct person
{
  char *soldier_name;
  struct person *next;
}SOLDIER;
```

Example:

- The **eliminating** function will delete the n-th node from the circular linked list (indicated by the **rep** pointer). The numbering will start from rep. The function will return the new **rep** pointer, which indicates the precedent node to the eliminated one.
- The **rep** pointer (landmark) is similar to **pcirc**.

```
SOLDIER *eliminating (SOLDIER *rep, int n)
{
      SOLDIER *q,*p;
      int k=1;

      if(rep==0)
         return(0); /* the case when the list is already empty */

…
```

```
…
    if(rep != rep->next) /*the list has at least two nodes */
    {
        p=rep;
        while(k<=n)
                {
                q=p;
                p=p->next;
                k++;
                }
        q->next = p->next;
        elibnod(p);
        rep=q;
    }
…
```

```
…
    else  /* the case when the list had had a single node */
          {
            elibnod(rep);
            rep=0;
          }

    return(rep);
}
```

Deleting a circular linked list

```
void erase_circular_list()
{
   extern NOD *pcirc;
   NOD *p,*p1;
   if ((p=pcirc)==0) return; /* the list is already empty*/
   do
      {
         p1 = p;
         p = p->next;
         elibnod(p1);
      } while (p != pcirc);
   pcirc = 0;
}
```

Josephus problem

(variant for a program in C)

A besieged city is defended by a number of knights (soldiers). The soldiers had decided to choose one of them to go for help. Soldiers are arranged in a circle (as in a circular list). The knights have to choose a number n, then starting from one of them start numbering and will successively eliminate each the **n**-th soldier, till the list will finally remain with only one soldier, which will be sent into the mission.

The steps of the algorithm are:

1) The numbering starts with node immediately following the **pcirc** pointer and delete the ***n***-th node from the list .

2) Repeat step 1, continuing with the node immediately following the deleted one, until the list is reduced to a single node.

The program solves the following requirements:

- it creates a circular list indicated by the **pcirc** pointer;
- reads a number indicated in the problem statement;
- eliminates nodes according to steps 1 and 2;
- shows the word from the remaining node in the list.