Data Structures and Algorithms (DSA) Course 6

Lists (Recapitulation)

Iulian Năstac

Recapitulation

Operations related to a linked list:

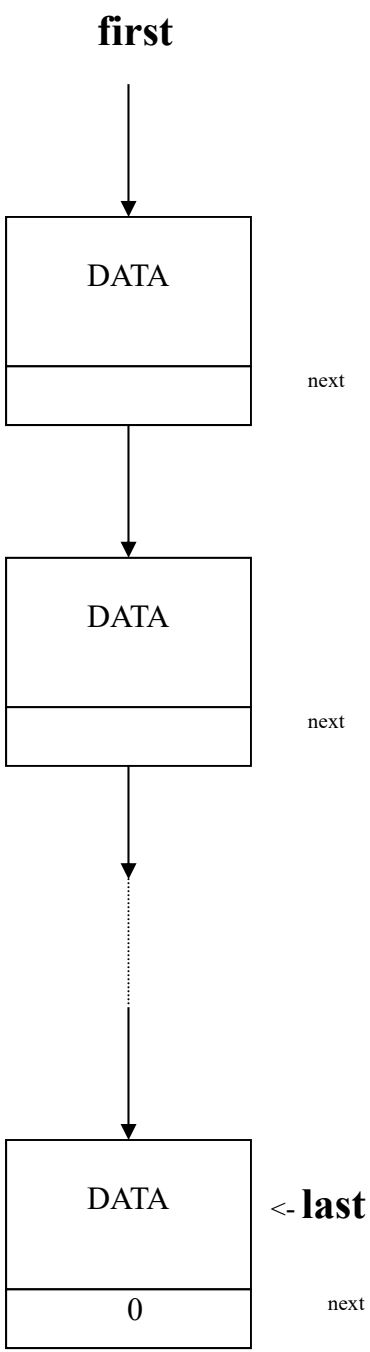
a) creation of a linked list;

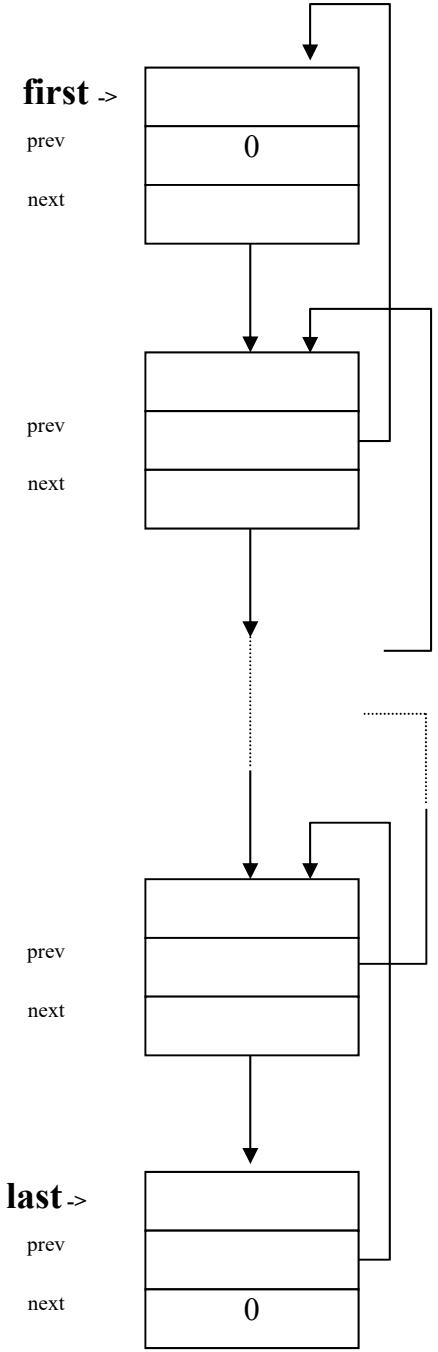
b) access to any node of the list;

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
0
next
prev
next
prev
next
last ->
prev
next
0

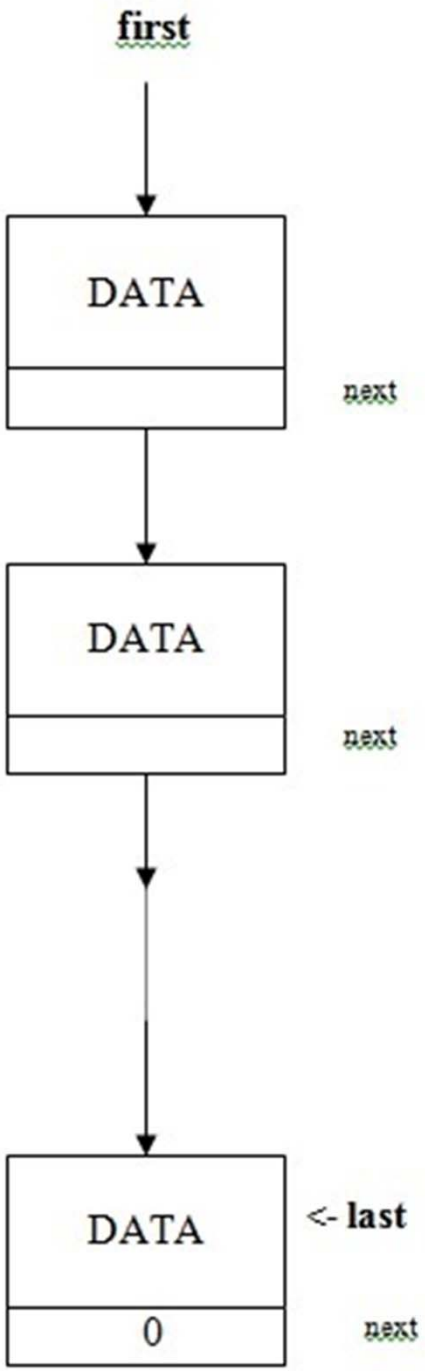
The simple linked list
(recapitulation)

Between the nodes of the simple linked list we have only one order relation. There is a single node that doesn't have a successor and a single node that doesn't have a predecessor.

These nodes are the edges of the list.

We will use two pointers to the front and back edges of the list, which we'll denote with:

- **first** – the pointer to the node that has no predecessor;
- **last** – the pointer to the node that has no successor.



The **next** pointer defines the successor relation for the list's nodes. This is a pointer for connexion. At each node, it has as value the address of the next node from the list. An exception is the linked pointer from the last node (for which **next** takes the value zero).

Notes:

- It is considered the following type:

```
typedef struct nod
        { <statements>;
          struct nod *next;
        } NOD;
```

- A special function (called **incnod**) is used, which loads the current data in a node of NOD type.

- The **incnod** function returns:
  - **1** to mark correct loading of data in the current node
  - **-1** when data are not loaded
  - **0** when an error occurs (eg insufficient memory)
- The prototype of **incnod** function is:

```
int incnod(NOD *p);
```

- Another required function releases the memory area reserved for a specified node:

```
void elibnod(NOD *p);
```

2.2. The access to a node in a simple linked list (Recapitulation)

- We can have access to the nodes of a simple linked list, starting with the **first** node, and then by passing from a node to another one, using the **next** pointer.

- A better method is to have a **key** date, (a component of the nodes) that should have different values, for each node. In this case it can be defined an access to the lists' node based on the value of this **key** (usually with the **char** or **int** type). The function returns the pointer to the identified node or zero, if there is no match to the **key**.

Example:

- It is considered the following user type:

```
typedef struct nod
        { char *word;  /*this is the key */
          int frequency;
          struct nod *urm;
        } NOD;
```

- Using a simple linked list (where nodes are of the NOD type), we have to write a function that can identify, in the respective list, the node for which the pointer **word** has as value the address of a given string.
- This pointer plays the key role and it is requested to be found the node that points to a given word.

```
NOD *search(char *c)
/* searches for a node in the list, which has a
similar word with the argument of the function */
{
 extern NOD *first;
 NOD * p;
 for (p = first; p; p = p->next)
     if (strcmp(p-> word, c) == 0)
         return p; /*It was found a node with a c key */
 return 0; /* There is no node in the list of key c */
}
```

2.3. Inserting a node in a simple linked list

In a simple linked list it can be done insertions of nodes in different positions, such as:

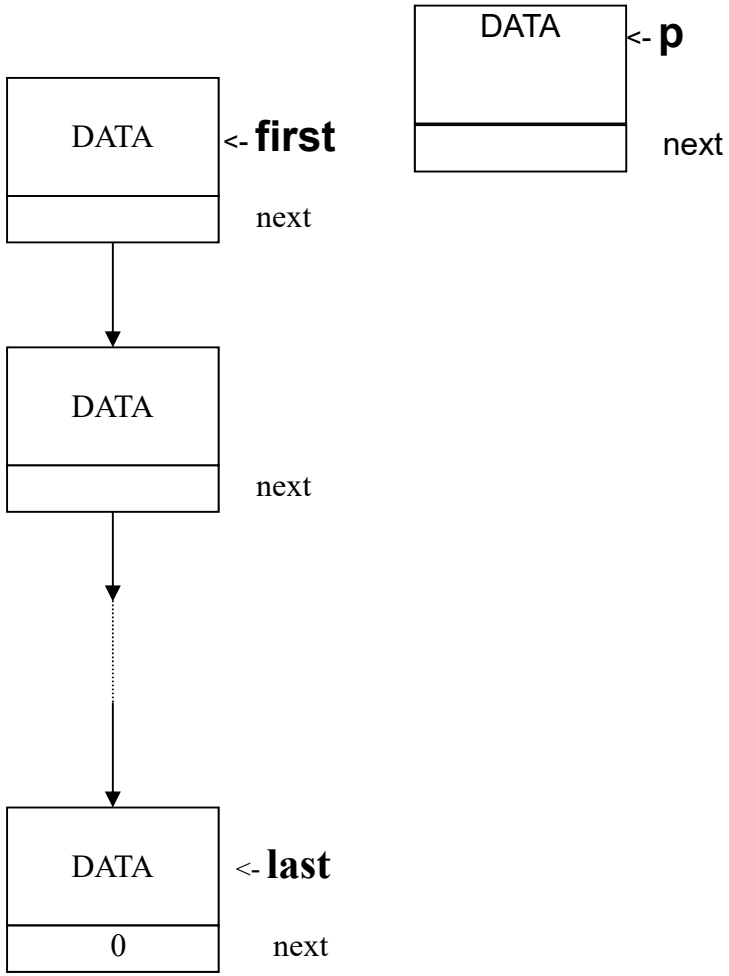
a) insertion before the first node;

b) insertion before a node specified by a key;

c) insertion after a node specified by a key;

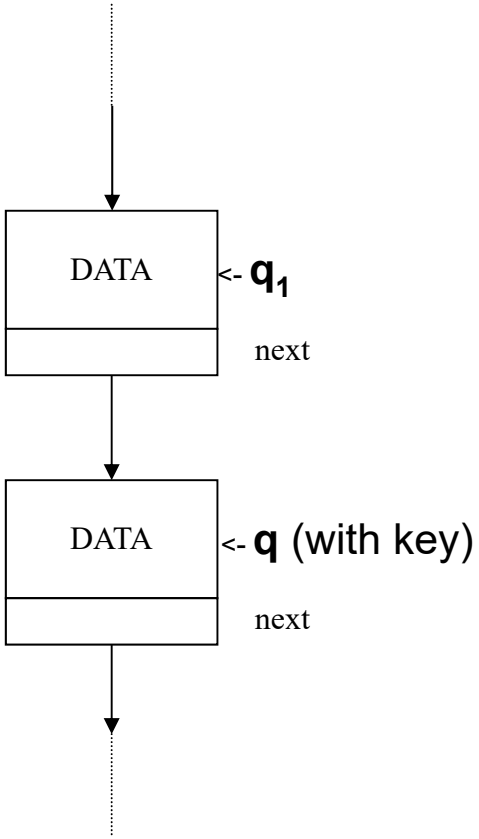
d) insertion after the last node of the list.

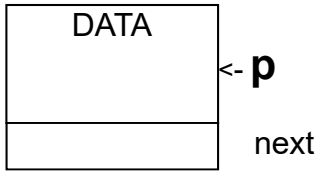
insertion before the first node




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

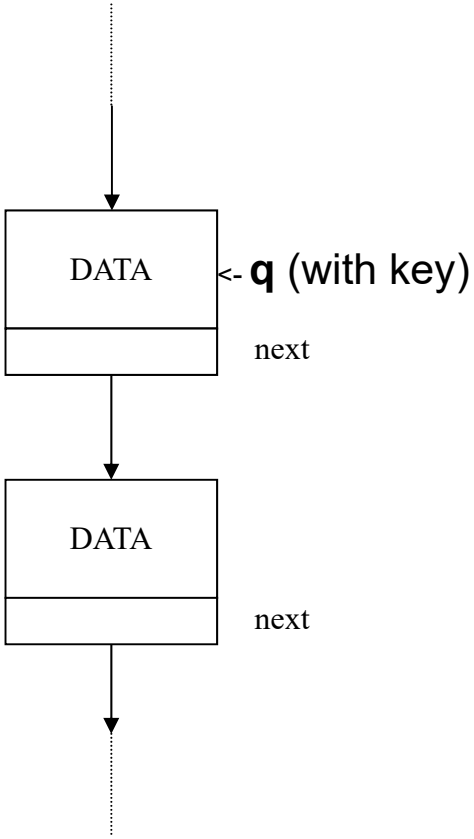
insertion before a node specified by a key

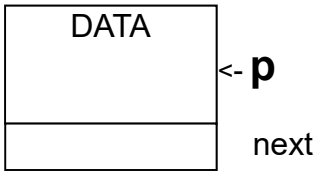





```
…
p->next=q;
q1->next=p;
…
```

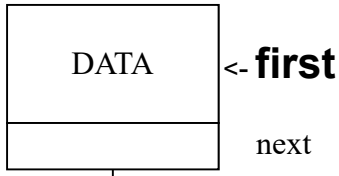
insertion after a node specified by a key

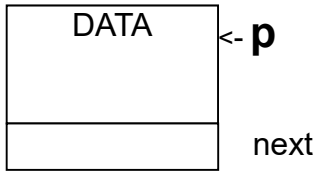





```
…
p->next=q->next;
q->next=p;
…
```

insertion after the last node of the list





…
last->next = p;
last=p;

last->next=0;
….

Example:

- insertion after the last node of the list

```
NOD *add()
/*      - Adds a node to a simple linked list;
- Returns the pointer to this node, or zero if it isn't added.
*/
{
 extern NOD *first, *last;
 NOD * p;
 int n;
/* a memory area is allocated for this node, which is then
filled with data */
 n = sizeof(NOD);
…
```

…

```
if (((p = (NOD *)malloc(n)) != 0) && (incnod(p) == 1))
  {
   if (first == 0) /* list is empty */
   first = last = p;
   else
       {
        last->next = p; /* p becomes the last node */
        last = p;
       }
   p->next = 0; /* similar with    last->next=0; */

   return p;
  }
```

```
…

 if (p == 0) /*it wasn't enough memory*/
  {
   printf ("Insufficient memory \n");
   getch(); /* pause for visualization */
   exit(1);
  }
 elibnod(p);
 return 0;
}
```

Supplement:

```
int incnod(NOD *p)
/* load current data in the node that is pointed by p */
{
 if((p -> word = citcuv()) == 0)  return -1;
 p -> frequency = 1;
 return 1;
}
```

```
char *citcuv()
/* - Reads a word and keeps it in the heap memory;
- Returns the pointer to that word or zero in the case of EOF   */
{
 int c, i;
 char t[255];
 char * p;

/* skip over characters that are not letters */
 while ((c = getchar ()) <'A' || (c> 'Z' && c <'a') || c> 'z')
          if (c == EOF)
                    return 0; /* EOF case */


…
```

```
…

/* read a word and keep it in the vector t */
 i = 0;
 do
 {
 t [i ++] = c;
 } while (((c = getchar()) >= 'A' && c <= 'Z' || c >= 'a') && c <= 'z');

if (c == EOF)
          return 0;
 t[i++] = '\0';

…
```

```
...

/* the word is saved in the heap memory */

 if ((p = (char *) malloc(i)) == 0)
    {
     printf("Insufficient memory \n");
     getch(); /* pause for visualization */
     exit (1);
    }
 strcpy (p, t);
 return p;
}
```

Warning! - Here p is not a pointer to the NOD type, but to a string of characters

Supplement:

```
void elibnod(NOD *p)
/* Release the heap memory areas allocated
by a pointer type node p */
{
  free(p->word);
  free(p);
}
```

2.4. Deleting a node from a simple linked list

There are considered the following cases:

a) deleting the first node of the simple linked list;

b) deleting a specified node with a key;

c) deleting the last node of the simple linked list.

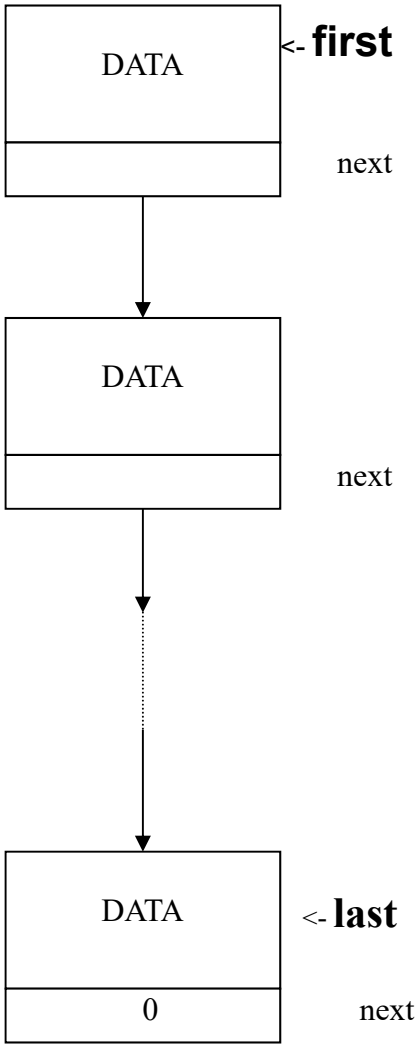
Notes:

- The deleting task uses a function called **elibnod**.

- This function releases the memory zone assigned to the node that is deleted.

- The codes of the **incnod** and **elibnod** functions depend of the application in which are used.

deleting the first node of the simple linked list

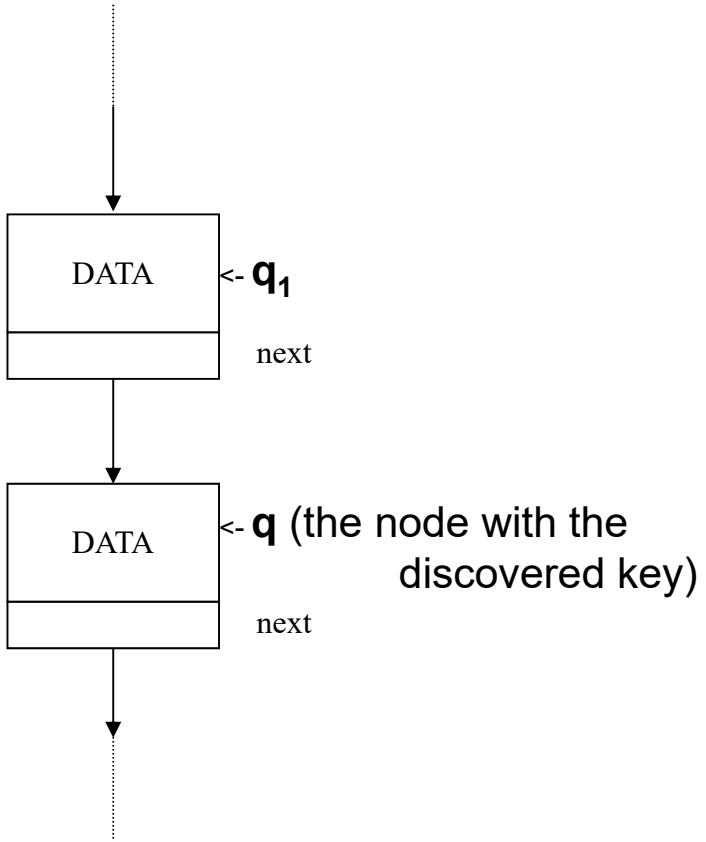



```
...

p = first;
first = p -> next;

elibnod(p);
...
```

deleting a specified node with a key

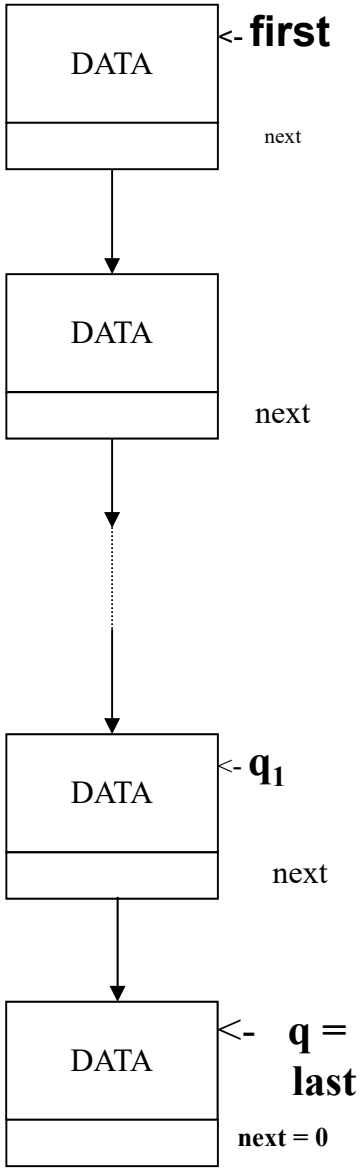



```
...

q1 ->next = q -> next;

elibnod(q);
...
```

deleting the last node of the simple linked list




```
...

q1->next = 0;
last = q1;

elibnod (q);

...
```

Example: deleting the first node of the simple linked list

```
void erase_first_node()
{
extern NOD *first, *last;
NOD *p;
if (first == 0) return;
p = first;
first = first -> next;
elibnod(p);
if (first == 0)  /* The list is empty */
        last = 0;
}
```

Example:

```
void erase() /* delete the last node from the list */
{
 extern NOD *first, *last;
 NOD *q, *q1;
 q1 = 0;
 q = first;
 if (q == 0)
    return; /* empty list */
 while(q != last) /* the list is scrolled */
       {
        q1 = q;
        q = q-> next;
        }
…
```

….

```
 if (q == first)
    first = last = 0;

 else
       {
        q1->next = 0;
        last = q1;
        }
 elibnod (q);
}
```

Observation:

- There is a great difference in complexity, if we compare the function that deletes the first node with a function that removes the last node of the list.

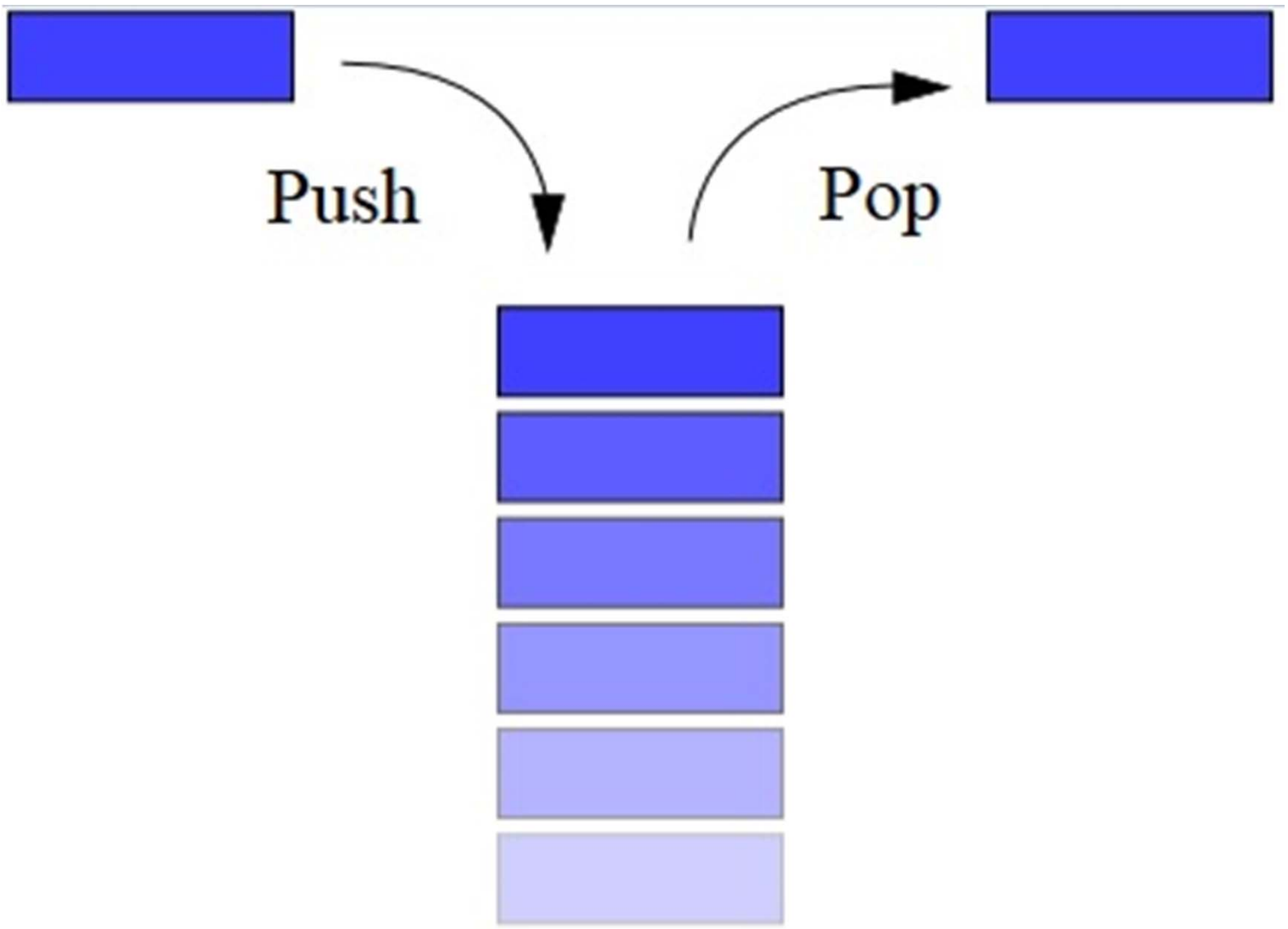
2.5. Deleting a simple linked list

```
void delete_list()
{
        extern NOD *first, *last;
        NOD *p;
        while(first)
                {
                        p = first;
                        first = first -> next;
                        elibnod(p);
                }
        last = 0;
}
```

STACK

- 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**


Push
Pop

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**)

- The first two operations are carried out at ***the top of the stack***. Thus, if an element is removed from the stack, then this element is on the top of the stack and, further, the previous inserted one reaches the top of the stack.

- If an element is inserted in a stack, it will be designate to the top of the stack.

To implement a stack using a simple linked list, we must identify the base and the top of the stack, these two becoming the two extremities of the list. There are two possibilities:

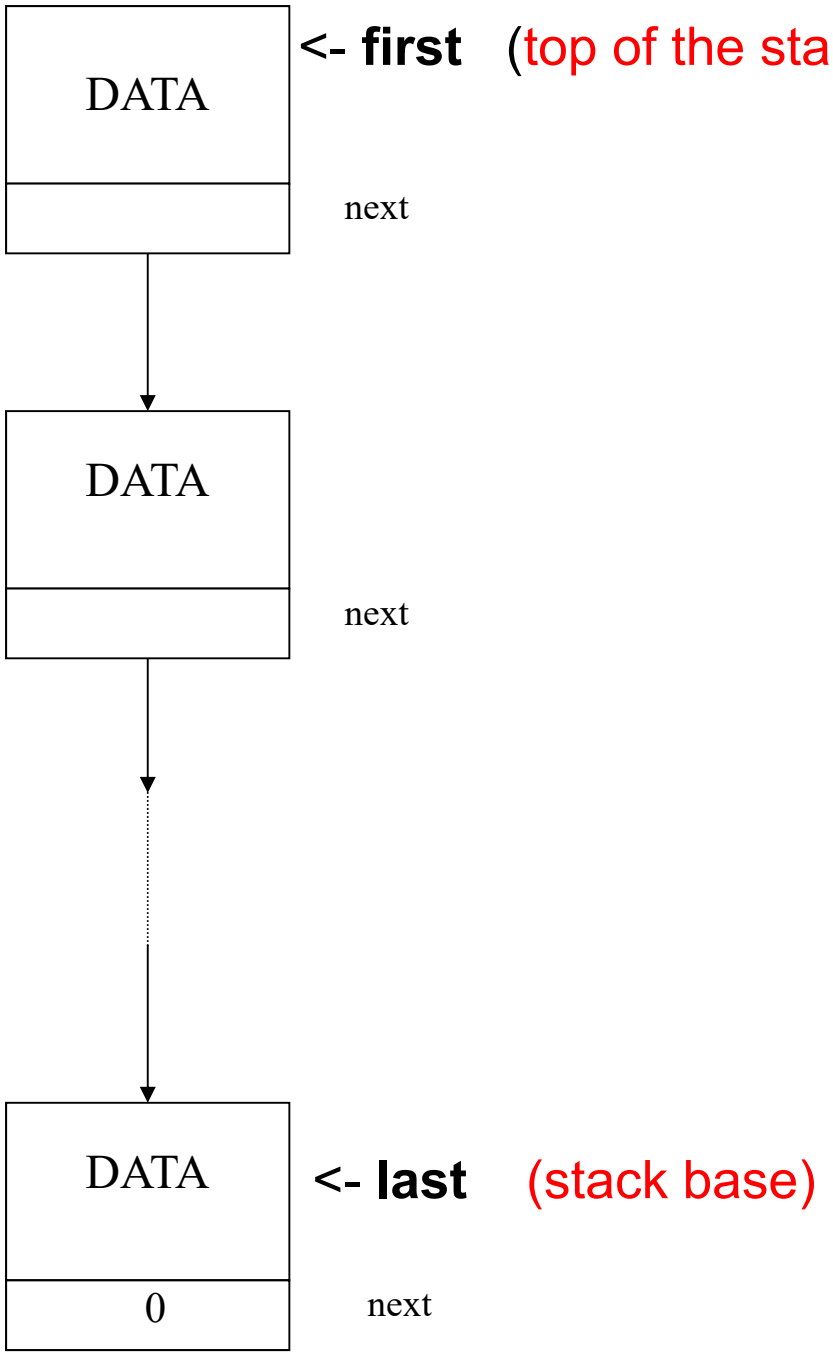
a. **The node indicated by the first variable will be the base of the stack, and the node indicated by the last variable will be the top of the stack;**

b. **The node indicated by the first variable will be the top of the stack, and the node indicated by the last variable will be the base of the stack.**

Notes:

- In case **a**, the PUSH and POP functions are identified by the **add** and **erase_last_node** functions, defined in the LIST laboratory. But, while the **add** function is an efficient one, the **erase_last_node** function isn't efficient.

- In case **b**, the PUSH and POP functions are identified by the **inifirst** and **erase_first_node** functions (to be defined in this lesson). In this case, both functions are efficient. Thus, it is recommended to implement the stack using a simple chained list, according to case **b**.


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

Example of PUSH (case b)

```
TNOD *inifirst() /* - PUSH - insert the current
node before the first node of the list */
{
  extern NOD *first, *last;
  NOD *p;
  int n;

  n = sizeof(NOD);
...
```

```
…

if(((p = (NOD *)malloc(n)) != 0) && (incnod(p) == 1))
   {
    if(first == 0)
     {
      first = last = p;
      p -> next = 0;
     }
    else
     {
      p -> next = first;
      first = p;
     }
    return p;
   }

…
```

```
 ...
  if(p == 0)
   {
    printf("insufficient memory\n");
    getch(); /* pause for visualization */
    exit(1);
   }
  elibnod(p);
  return 0;
} /* end inifirst */
```

Particular case:

The problem with trains (from the third Laboratory)

```
typedef struct tnod
        {
          long cvag; /* Code of the wagon */
          long cmarfa; /* Commodity code */
          int exp; /* sender */
          int dest; /* recipient */
          struct tnod *next;
        } TNOD;
```

```
int incnod(TNOD *p)
/* upload a node with data about the associated wagon  */
{
  char t[255];
  char er[ ] = "EOF was typed in bad position\n";
  long cod;
  int icod;

  /* read code of the wagon */
  for( ; ; )
       {
        printf("\nCode of the wagon: ");
        if(gets(t) == 0)
           return -1;   /* no data */
        if(sscanf(t, "%ld", &cod) == 1 && cod >= 0 && cod <= 999999999)
           break;
        printf("Error for wagon code\n");
       }
  p -> cvag = cod;
…
```

```
...
  /* read code of the goods */
  for( ; ; )
       {
        printf("Code of the goods: ");
        if(gets(t) == 0)
                  {
                    printf(er);
                    return 0;
                  }
        if(sscanf(t, "%ld", &cod) == 1 && cod >= 0 && cod <= 999999999)
                    break;
        printf("Error for code of the goods\n");
       }
  p -> cmarfa = cod;

  /* read sender code */
...
  /* read recipient code */
...
 return 1;
} /* End incnod */
```

```
void elibnod(NOD *p)
 /* erase the node pointed by p */
{
    free(p);
} /* End  elibnod */
```

Example of POP (case b)

```
void erase_first_node() /* POP - delete the first node */
{
  extern TNOD *first, *last;
  TNOD *p;

  if(first == 0)
        return;
  p = first;
  first = first -> next;
  elibnod(p);
  if(first == 0)
        last = 0;
} /* End  erase_first_node */
```

Note:

In some cases a stack can be implemented using an array of pointers (which we call it: **tpnod**).

Considering:
tpnod[0] - bottom of the stack
tpnod[v] - top of the stack

where:
int v; /* is a global variable that can have the maximum value: MAX */

You can define two functions on this stack:
empty() - returns 1 if the stack is empty and 0 otherwise
full() - returns 1 if the stack is full and 0 otherwise

Therefore we can use the type:

typedef enum {false, true} Boolean;

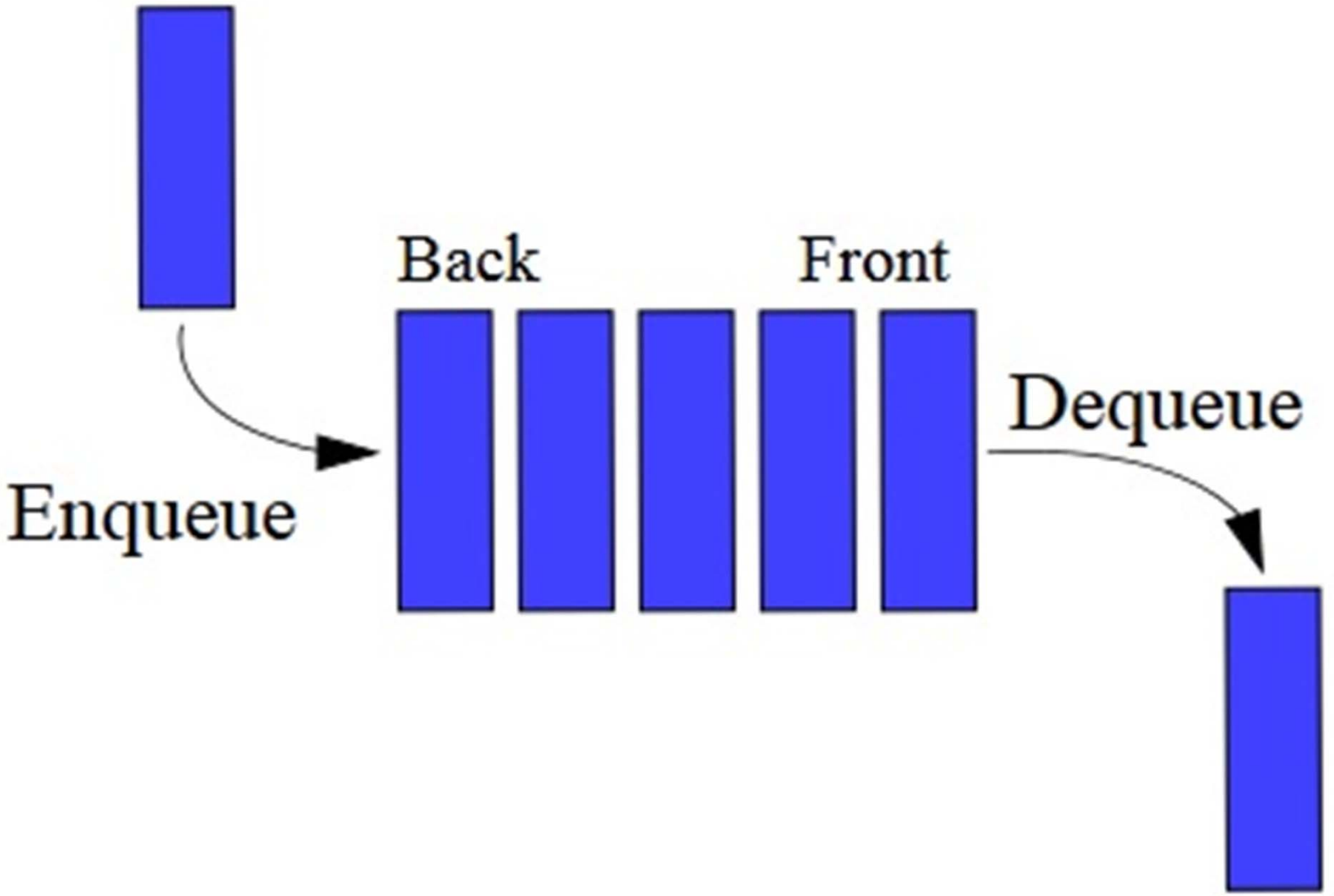
…

```
Boolean empty()
        {
          extern int v;
          return (v == 0);
        }
…

Boolean full()
        {
          extern int v;
          return (v >= MAX);
        }
```

Queue

- 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).

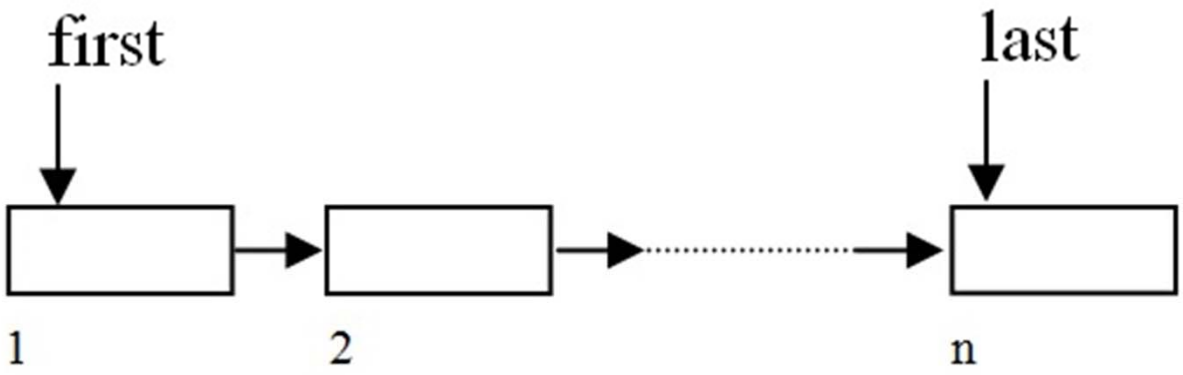

Back
Front
Dequeue
Enqueue

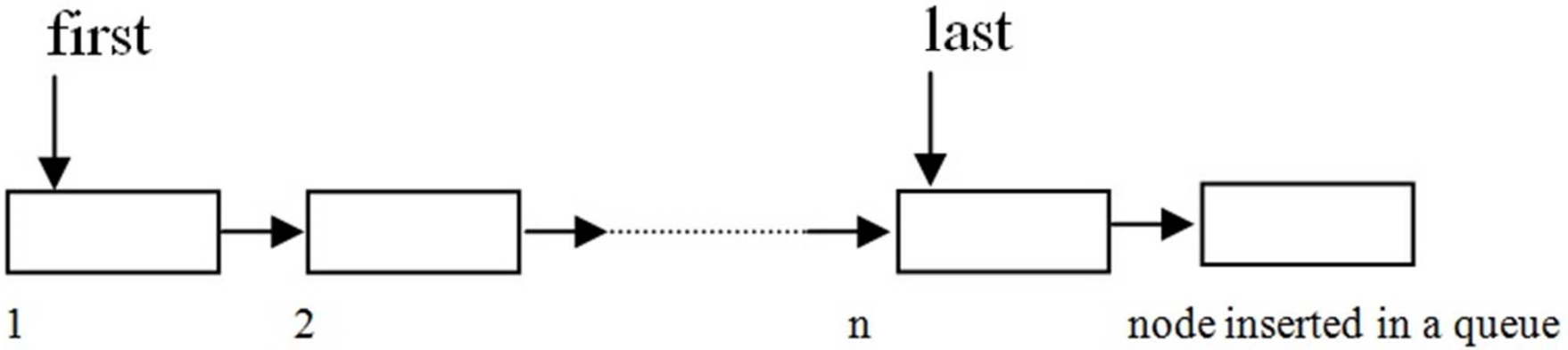
On a **queue** we can define only two operations:

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

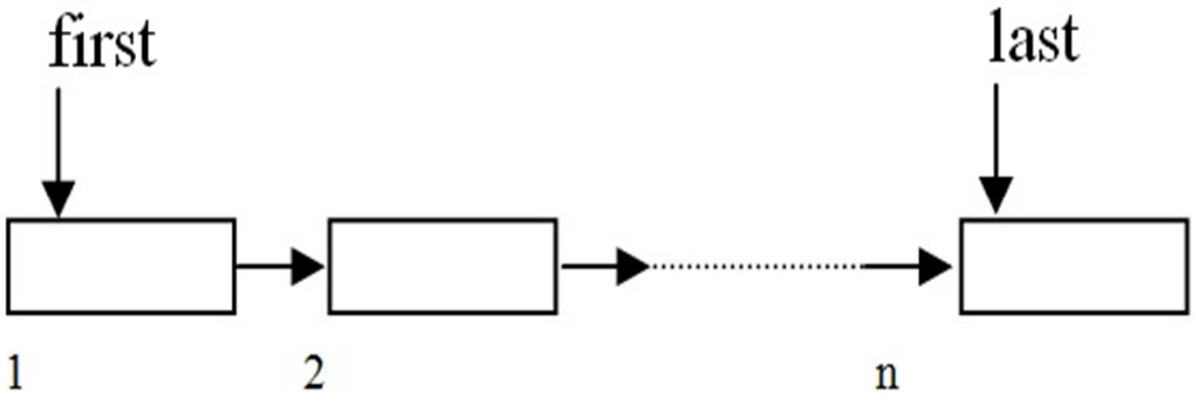
b. extracting an element from a queue (from its front) - DEQUEUE;

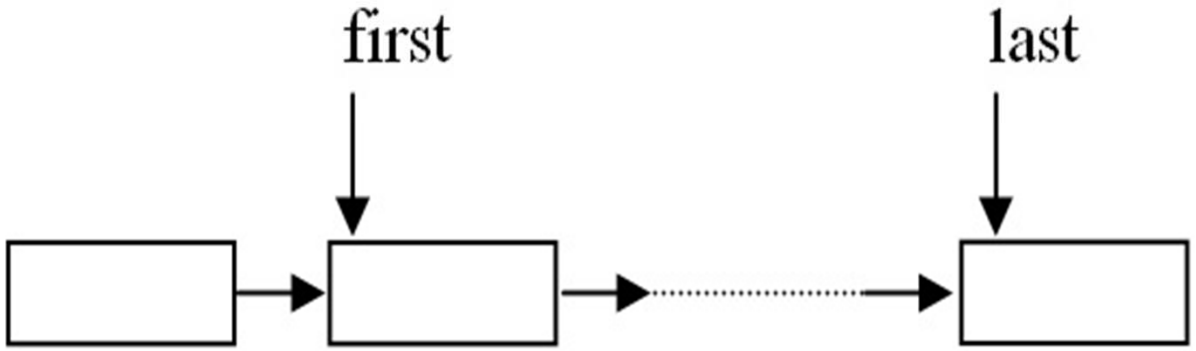
The operation of inserting a node in a queue:





The operation of extracting a node from the queue





the node
extracted from the queue

Circular simple linked list

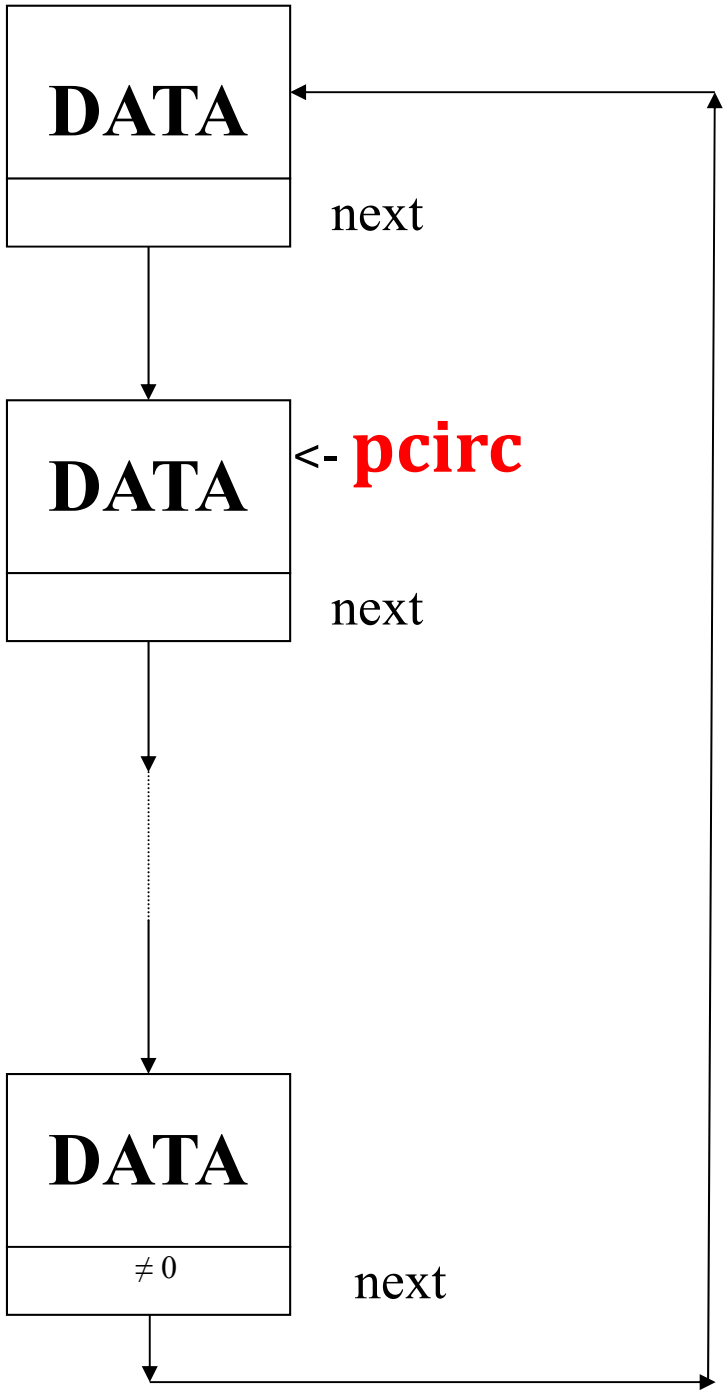
A simple linked list contains:

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**.


DATA
next
DATA
<- pcirc
next
DATA
≠ 0
next