Data Structures and Algorithms (DSA) Course 5

Lists

Iulian Năstac

Cap. Lists

(recapitulation)

1. Introduction

- Linked lists are the best and simplest example of a dynamic data structure that uses pointers for theirs implementation.
- Understanding pointers is crucial to think how linked lists work.
- Essentially, linked lists function as an array that can grow and shrink as needed, from any point in the array.

Recapitulation

A group of dynamic recursive structures of the same type, which includes one or more relationships for linking elements (using pointers from these structures), is called a **linked list**.

Recapitulation

- The elements of a list are called **nods.** If between the nods of a lists is only one order relation, then the list is called **simple linked**.

- Similarly, the list is **double linked** if in between the nods are defined two relations for order.

- A list is **n-chained** if, between every two successive nodes, there are defined ***n*** relations for order.

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.

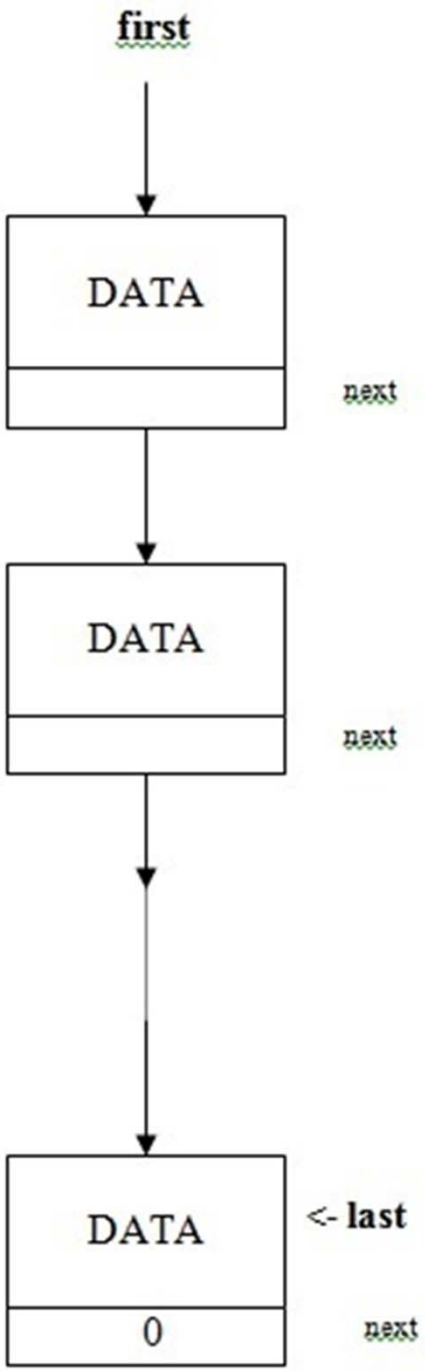
2. The simple linked list
(recapitulation)

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).

2.1. Creating a simple linked list
(recapitulation)

When a simple linked list is built, we are doing the following tasks:

a) initialize both pointers **first** and **last** with zero value, because at the beginning the list is empty
b) allocate the memory zone in the heap memory for the current node **p**
c) load the current node **p** with the correspondent data (if it exists), and then go to the next step **d).** Otherwise eliminates p and returns from the function.

d) assign the addresses of the list, for the current node:

last -> next = p, if the list is not empty;

first = last = p, if the list is empty.

e) the current node is assigned with the pointer that denotes the last element of the list (**last=p**)

f) **last -> next** = 0

g) the process jumps to point **b)** in order to add a new node to the list.

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

- 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**.

- Usually a **key** variable is inserted into the user type:

```
typedef struct nod
        { <statements>;
          type key;
          struct  nod *urm;
        } NOD;
```

- The key can be int, long, double, char, etc.

Example:

- It is consider the following user type:

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

- Considering a simple linked list (where nodes are of the TNOD 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.
- In other words, 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 can be done insertions of nodes in different positions:

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.

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 reserved 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;
   return p;
  }

...
```

```
…

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

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.

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;
}
```

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.

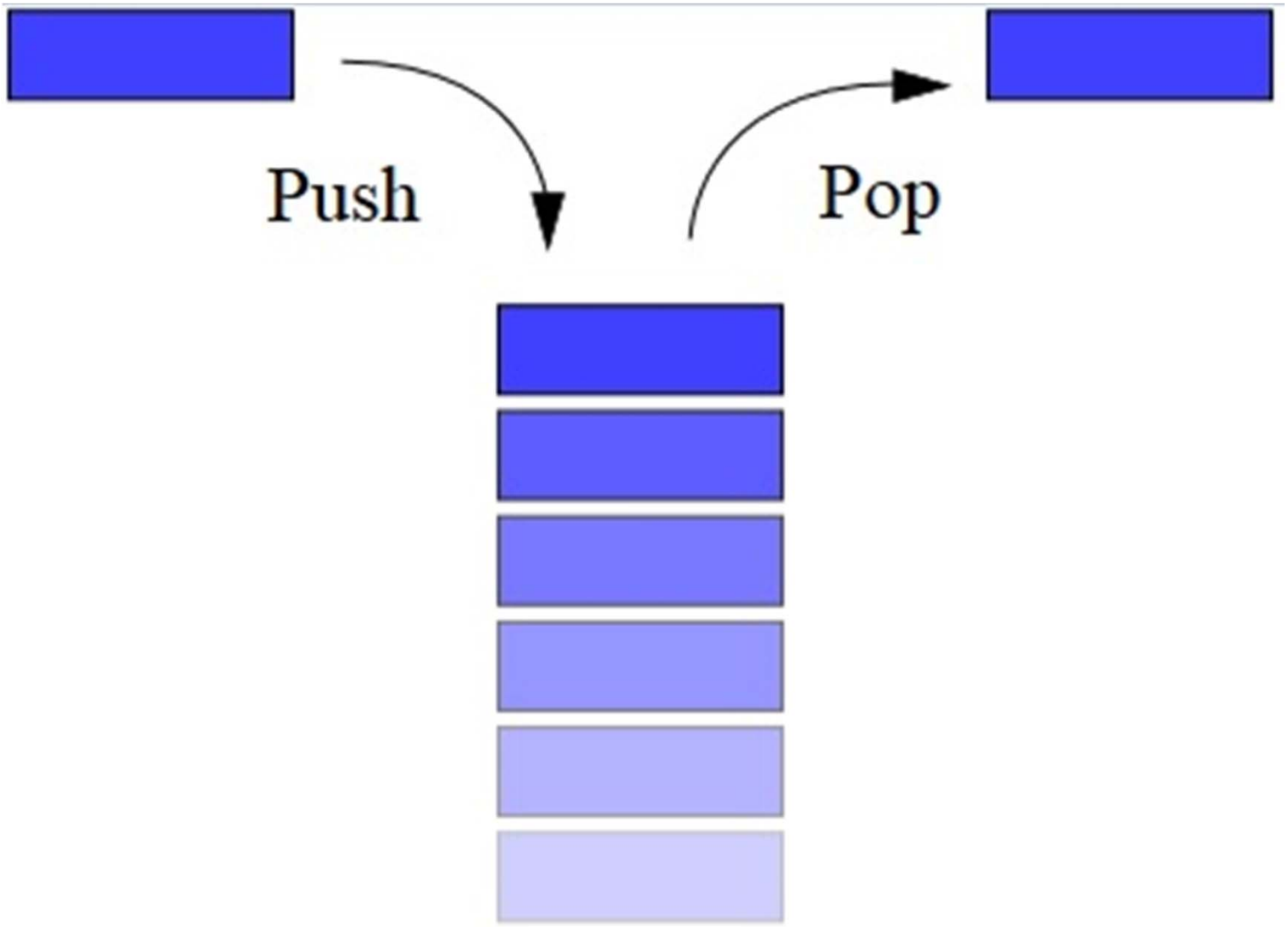
2.5. Deleting a simple linked list

```
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**.