Laboratory: Simple linked list
I. THEORETICAL ASPECTS

1. Introduction

As first definition, a list is a dynamic group, meaning that it has a variable number of elements.

At the beginning, a list is an empty group. During the program execution, we can add new items to the list and also various elements can be removed from the list.

* The elements of a list consist of a sequence of data (of the same type). The common type of the list's elements is a user type. It results that the elements of a list are structures of the same type.

* Often the lists are organized as an array. This is not efficient because the lists are dynamic and the arrays from the C language aren't.

* There are situation when it's difficult to evaluate the maxim number of a list's elements, and cases when their number varies from execution to execution. In consequence it appears the problem about organizing such a type list lot.

The ordering of the elements of a list is done using pointers that enter into the composition of list's elements. Due to these pointers, the list's elements become recursive structures. The lists organized in this way are called linked lists.

By definition, a dynamic group of recursive structures of the same type, for which are defined one or more relations of order using some pointers from the composition of the respective structures, is called linked list.

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 order relations.

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

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 linked pointer. For 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, see **figure 1.1.a**)

3. Double-linked list

In such a list each node contains two pointers: one to the next node and one to the previous one (**figure 1.1.b**). Therefore, almost any node of the list has a previous node defined by the pointer **prev** and a next node defined by the pointer **next**.

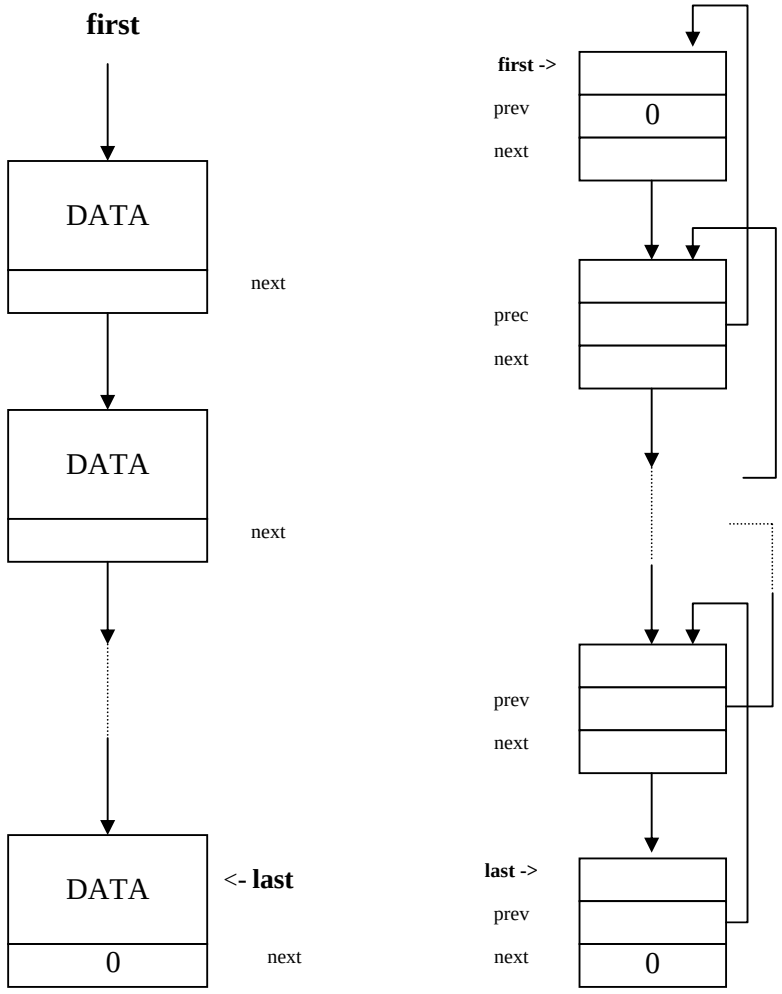


Figure 1.1. The simple linked list (a) The double linked list (b)

4. Creating a simple linked list

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

last -> next = p, if the list is not empty;
first = next = p, if the list is empty.
e) the current node is assigned with the pointer which denote the last element of the list
f) **last -> next** = 0
g) the process jumps to point b) in order to add a new node to the list.

A special function, called **incnod**, is used.

5. 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**.

6. 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.

7. 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.

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.

II. ASSIGNMENT WORKFLOW

NOTE: The functions or programs from each point of this section will be written in separate files with the extension **.cpp**.

1. Write a function that reads one word and keeps it in the heap memory (a word means a succession of uppercase and lowercase letters). The function returns the starting address of the memory zone in which the word is maintained, or zero in the case of EOF (Ctrl + Z).

A possible solution:

```
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");
     exit (1);
    }
 strcpy (p, t);
 return p;
}
```

Note: **strcpy** function was studied during previous semester

2. It is consider the following user type:

```
typedef struct tnod
{
  char * word;
  int frequency;
  struct tnod *next;
}TNOD;
```

which will be used in all the next assignments.

It is required to write a sort of **incnod** function, which loads the current data in a TNOD type node. This function calls the **citcuv** function and assigns the returned address of it to the word pointer from the current node. Also, it is assigned the value 1 to the frequency variable. The **incnod** function returns the value -1 if **citcuv** returns 0, otherwise returns 1.
A possible solution:

```
int incnod(TNOD *p)
/* tries to load the current data in the node p and shows if it was done successfully */
{
  if ((p->word = citcuv()) == 0) return -1;
  p -> frequency = 1;
  return 1;
}
```

3. Write a function that releases the zones from the heap memory allocated by the node that was defined in the previous exercise.

A possible solution:

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

4. Write a function called **add** that allows the adding of a TNOD type node at the end of a simple linked list. This means that after the insertion of this node, it becomes the last node that has no successor. This function calls the **incnod** and **elibnod** functions defined in the previous exercises.

A possible solution (**add after the last one**):

```
TNOD *add()
/*        - Adds a node to a simple linked list;
- Returns the pointer to this node or zero if it isn't added. */
{
 extern TNOD *first, *last;
 TNOD * p;
 int n;
/* a memory area is reserved for this node, which is then filled with data */
 n = sizeof(TNOD);
 if (((p = (TNOD *)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) /* the node was not inserted in the list */
  {
   printf ("Insufficient memory \n");
   exit(1);
  }
 elibnod(p);
 return 0;
}
```

5. Considering a simple linked list (nodes are of the TNOD type, defined at point 2), write a function that searches, 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.

A possible solution:

```
TNOD *search(char *c)
/* searches for a node in the list, which has a similar word with the argument of the function */
{
 extern TNOD *first;
 TNOD * 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 */
}
```

Note: **strcmp** function was studied during previous semester

6. Write a function that deletes the last node from a simple linked list (whose nodes have the TNOD type, defined at point 2). The function will call the **elibnod** function.

A possible solution (**erase_last_node** function):

```
void erase() /* delete the last node from the list */
{
 extern TNOD *first, *last;
 TNOD *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);
}
```

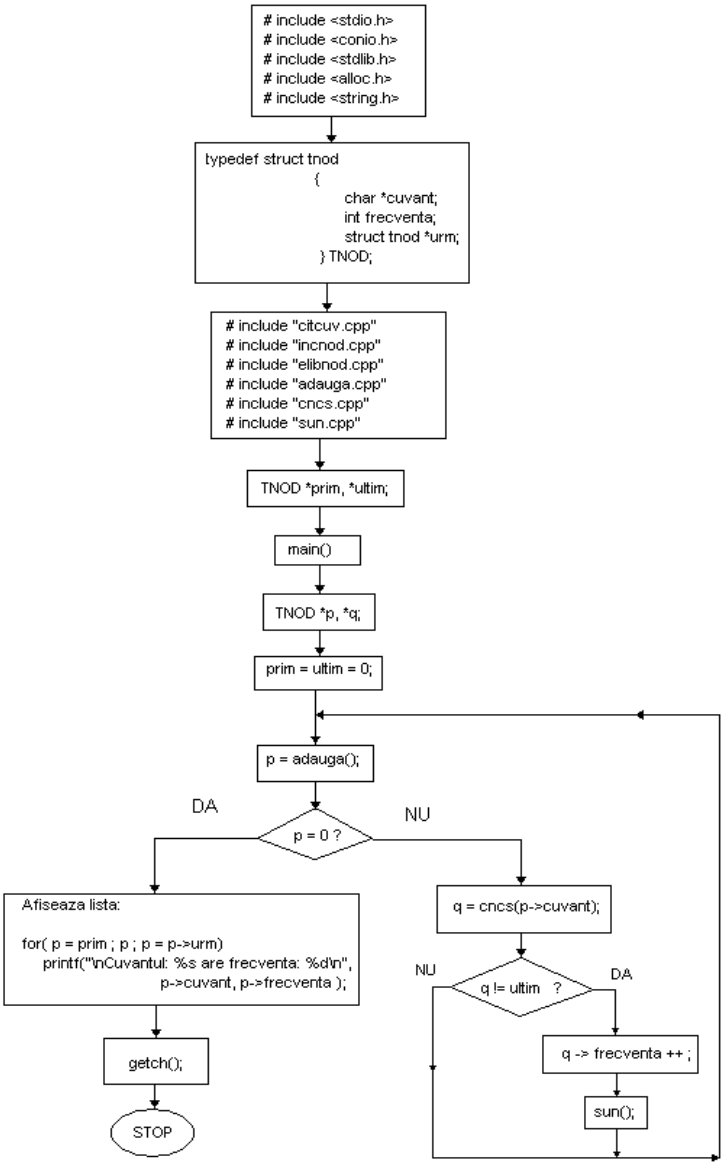
7. Write a program that reads the words from a text and displays the number of occurrences of each word from this text. The word is defined as a succession of upper and lower cases. The text ends when CTRL+Z are typed.

The program uses a simple linked list whose nodes have the TNOD type already defined and it is executed as follows:
a) For every new word a node for the respective word is built. This contains the pointer to the word, which is kept in the heap memory. The frequency of the current word is set to 1.
b) The new node from the point a) is added at the end of the simple linked list.
c) In the list is searched another node similar with the last added word. If it exists such a node (and this is not the last node of the list) then the frequency from the respective node is increased and it is deleted the last node of the list (the one added at point b), because the word already exists in the list.
d) Then, we return to step a) and the cycle continues until there are no more words to read (CTRL+Z are typed).
e) Now the words and their frequencies are listed.
The program will include all the functions defined at the previous points.

8. Write and add a function in the program which will arrange the list in an alphabetical order.

III.ANSWERS

7. The work flow of the program:



Listing of the file that contains the main function:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <malloc.h>
#include <string.h>

typedef struct tnod
{
 char *word;
 int frequency;
 struct tnod *next;
}TNOD;

TNOD *first, *last;

#include "citcuv.cpp"
#include "incnod.cpp"
#include "elibnod.cpp"
#include "add.cpp"
#include "search.cpp"
#include "erase.cpp"

int main() /* read a text and displays the frequency of words in text */
{
  TNOD *p, *q;
  first = last = 0; /* The start list is empty */
  while((p = add())!= 0)
        if ((q = search(p-> word))!= last)
           {
            q->frequency ++;
            erase();
           }

/* Lists the words and their frequencies */
  for (p = first; p; p = p->next)
    printf ("\n The word %s has frequency = %d \n", p->word, p->frequency);
  getch ();
}
```

8. The listing of the file that includes the **ordlist** function (main function is modified accordingly):

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <malloc.h>
#include <string.h>

typedef struct tnod
{
 char * word;
 int frequency;
 struct tnod *next;
}TNOD;

TNOD *first, *last;

#include "citcuv.cpp"
#include "incnod.cpp"
#include "elibnod.cpp"
#include "add.cpp"
#include "search.cpp"
#include "erase.cpp"

void ordlist()
{
  /* extern  TNOD *first, *last   */
  TNOD *p, *p1, *q;
  int ind;
  if (first == 0)
     return; /* Empty list */

  ind = 1;
  while(ind)
  /* interchange the neighboring nodes that are not in alphabetical order */
     {
      ind = 0;
      p = first; /* p points to the current node */
      p1 = 0; /* p1 points to the previous node */
      q = p->next; /* q points to the next node */

      while (q != 0) /* next node exists */
           if (stricmp (p -> word, q -> word)> 0)
           /* The words from the current node and the next one are not in alphabetical order, and must
be interchanged  */
              {
               if (p == first) /* the current node has no a previous one */
                  first = q; /* q becomes the first node of the list */
               else
```

```
                p1->next = q; /* q is after p1 */
              p->next = q->next;
              q->next = p; /* p is after q */
              p1 = q;
              q = p->next;
              if(q == 0)
                  last = p;
              ind = 1; /* there is a permutation, so the ordering is not finished yet */
             } /* End if */
           else /* advancing to the next pair */
               {
               p1 = p;
               p = q;
               q = q->next;
               } /* End else */
    } /* End while(ind) */
} /* End ordlist */

int main() /* read a text and displays the words in alphabetical order */
{
  TNOD *p, *q;
  first = last = 0; /* The list is empty */
  while ((p = add())!= 0)
       if ((q = search(p->word))!= last)
          {
           q->frequency ++;
           erase();
          }

  /* Sort */
  ordlist();

  /* Print the nodes in alphabetical order */
  for (p=first; p; p = p->next)
    printf ("\n Thes word %s has the frequency = %d \n", p->word, p->frequency);
  getch ();
}
```