DATA STRUCTURES AND ALGORITHMS

Linear data structures: List

Summary of the previous lecture

- Definition of data, data structure and algorithm
- Abstract data types:
  - Integer (int),
  - Real (float, double)
  - Boolen (bool)
  - Character (char)
  - Array
  - Class (class)
- Arrays
  - Features (elements, indexes, numbering)
  - Adding, deleting, searching

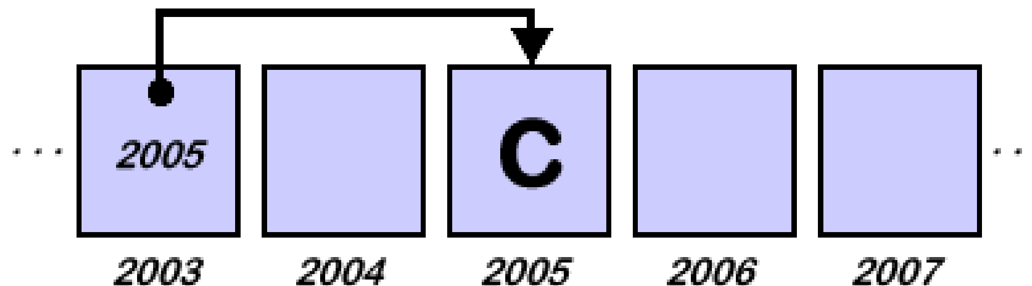
Before starting a new material

Pointers and Indirection

We need some way to map Data Structures to the computer's linear memory. One solution to this, is to use **pointers**.

Pointers are variables that store memory addresses.

Pointer holds a memory address. One memory cell can point to another memory cell. The process of accessing the data through pointers is known as **indirection**. Using pointers, we can also create multiple levels of indirection.



List

The most common linear data structure used is the **list**. The most important concept related to lists is that of position.

Ordered list is very similar to the **alphabetical list** of employee names for the *ABC* company.

Ordered list keeps items in a specific order such as alphabetical or numerical order. Whenever an item is added to the list, it is placed in the correct sorted position so that the entire list is always sorted.

List to be a finite, ordered sequence of data items known as elements. "**Ordered**" in this definition means that each element has a position in the list.

List features

All elements of the list have the **same data type**, although there is no conceptual objection to lists whose elements have differing data types if the application requires it.

For example, the list ADT (abstract data type) can be used for lists of integers, lists of characters, lists of payroll records, even lists of lists.

A list is said to be **empty** when it contains no elements.

The number of elements currently stored is called the **length** of the list.

The beginning of the list is called the **head**, the end of the list is called the **tail**.

There might or might not be some relationship between the value of an element and its position in the list.

List operations

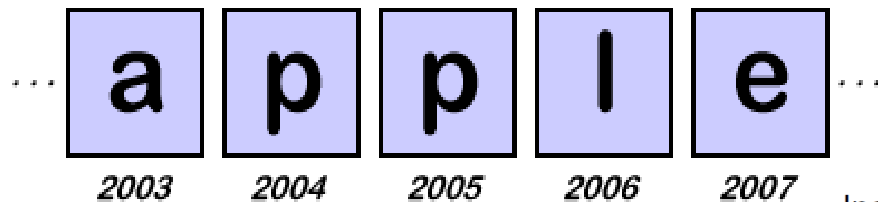
Lists feature following operations:

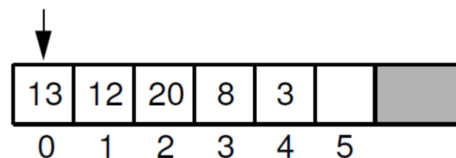
- to **grow and shrink in size** as we insert and remove elements.
- to **insert and remove elements** from anywhere in the list.
- to **gain access to any element's value**, either to read it or to change it.
- to **create and clear** (or reinitialize) lists.
- to **access the next or previous element** from the "current" one.

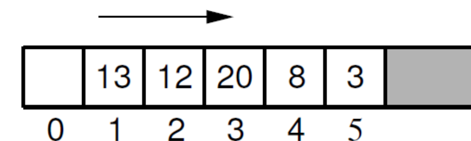
Array Implementation

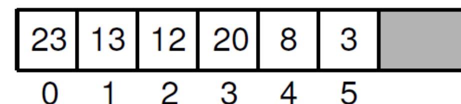
One approach to creating a list is simply to reserve a block of adjacent memory cells to large enough to hold the entire list.

Such a block of memory is called as **array**. Of course, since we want to add items to our list, we need to reserve more memory cells.









Linked lists

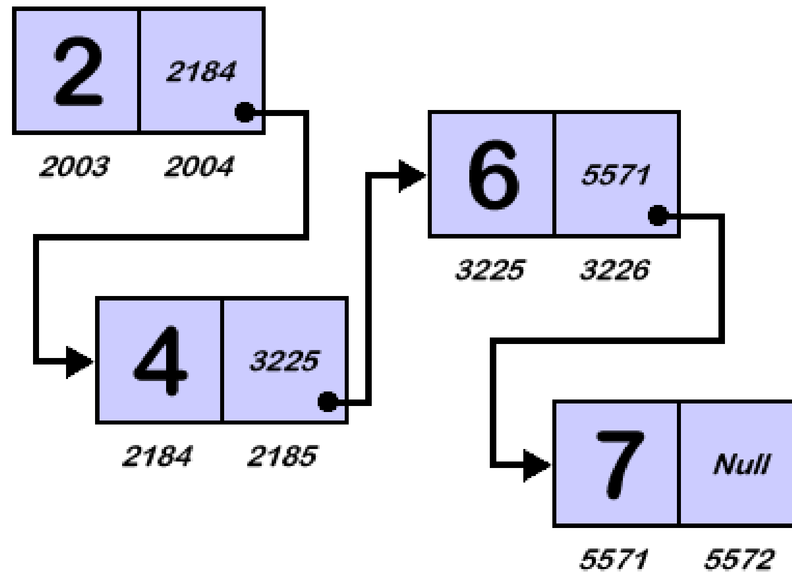
The linked list uses **dynamic memory allocation**, that is, it allocates memory for new list elements as needed.

A linked list is made up of a series of objects, called the **nodes of the list**. List node is a distinct object (as opposed to simply a cell in an array).

Every node contains the **data item** and **the pointer** to the next item in the list.

Linked list is as a chain of nodes linked together by the pointers. As long as we know where the chain begins, we can follow the links to reach any item in the list.

Linked list



Notice that the last memory cell in our chain contains the symbol called "**Null**". This symbol is a special value that tells us that we have reached the end of our list. You can think that this symbol as a pointer that points to nothing.

Types of linked lists

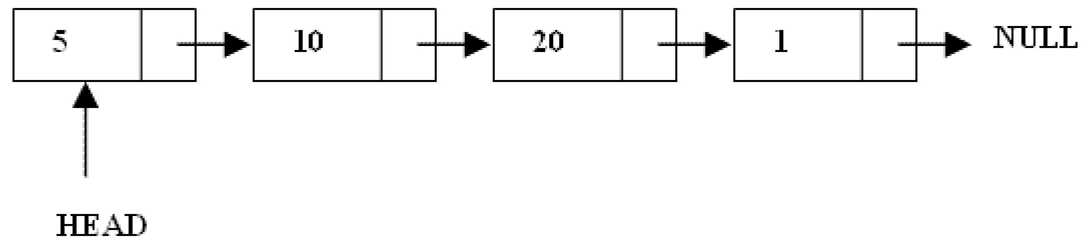
Depending on the number of links in the node and their types, linked list can be:

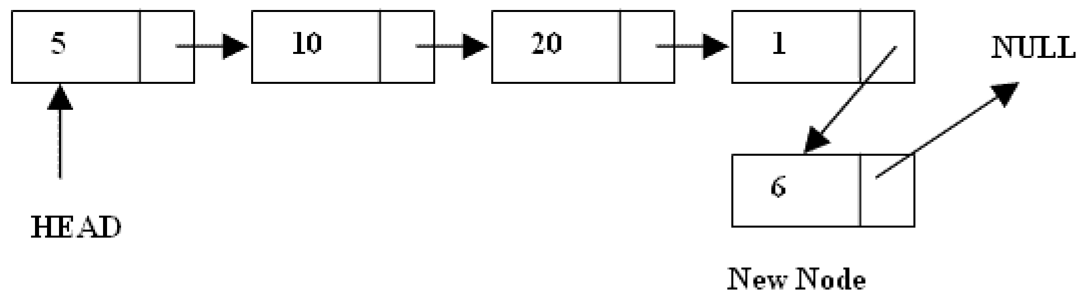
- Singly-linked lists
- Doubly-linked list,
- Singly-circularly-linked list
- Doubly-circularly-linked list

Singly-linked list

Singly-linked list is a list of elements in which the elements of the list can be placed anywhere in memory.

All of list elements are linked together with each other using an explicit link field, that is, by storing the address of the next element in the link field of the previous element.



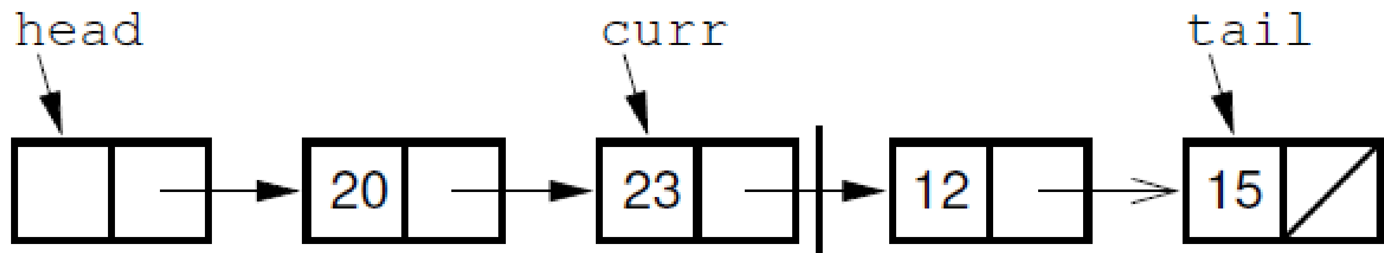


Header node

Special header node as the first node of the list is used in the linked lists.

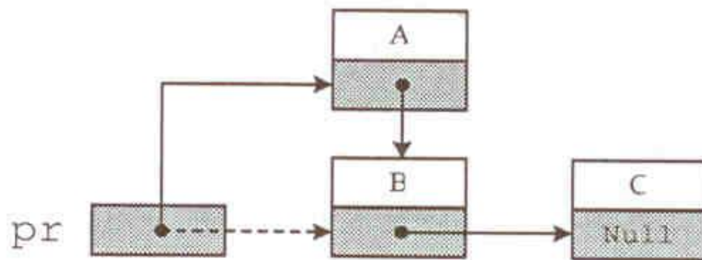
This header node is a link node like any other, but its value is ignored and it is not considered to be an actual element of the list.

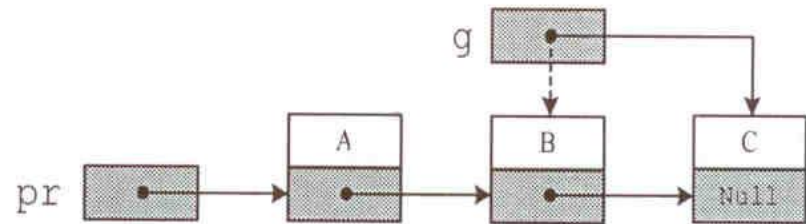
The header node saves coding effort because we no longer need to consider special cases for empty lists or when the current position is at one end of the list.



Implementation of singly linked list

Adding a new element into the singly linked list.



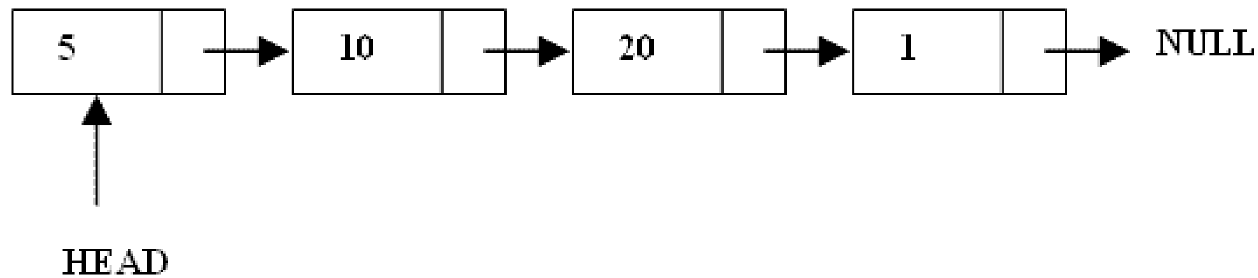


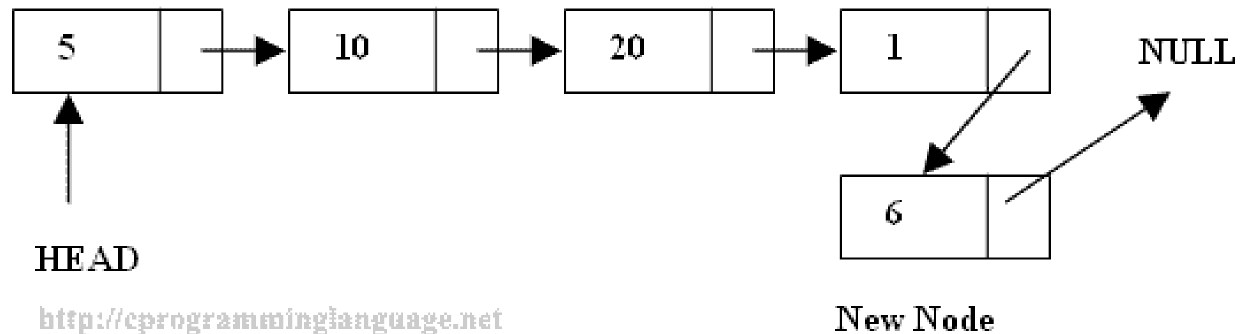
Converse order

Reguliar order

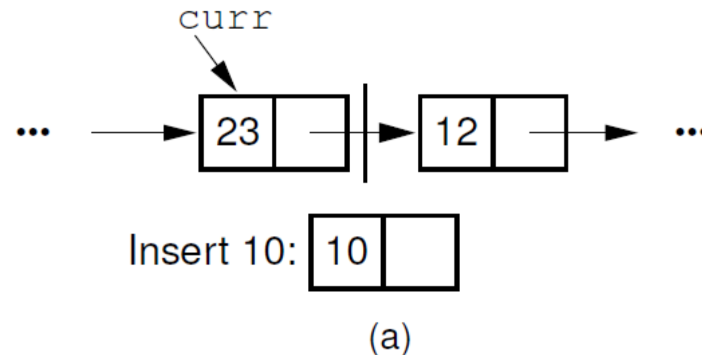
here **pr** is pointer to the first element of the list.

Adding a new element to the end

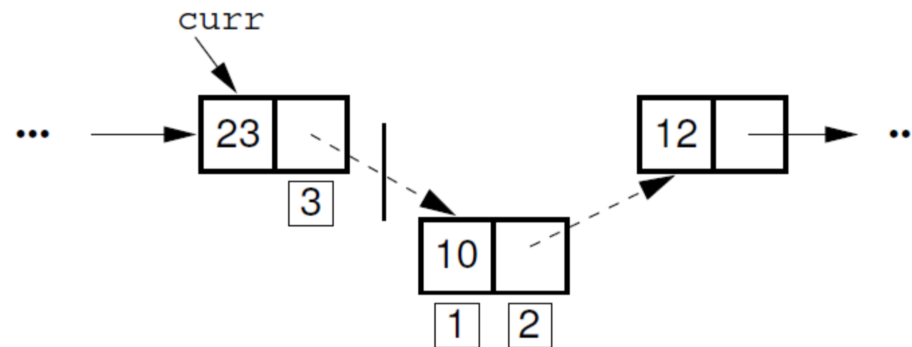




Insertion of a new element

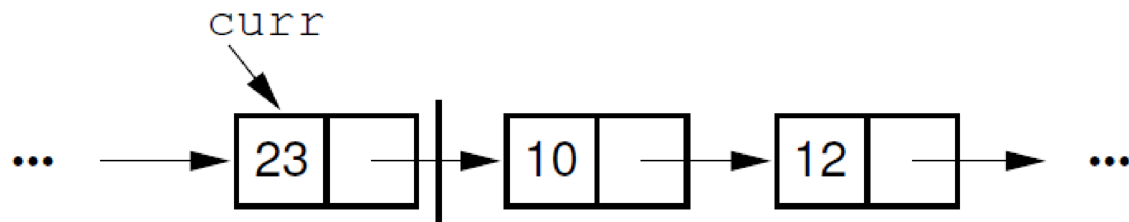


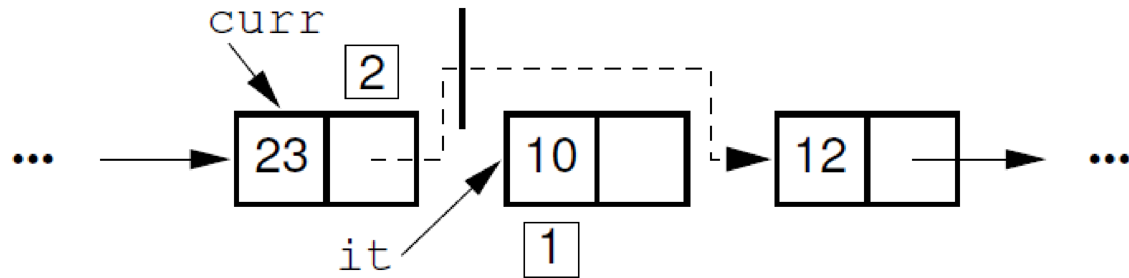
(a)



- Pointer *curr* shows the element (23) address.
- Copy pointer field value of (23) to pointer field of the new element.
- Change the pointer value of (23) to address of (10).

Deletion of the element





Search of the element

Algorithm

- Loop while element is found or tail of the list is reached.
- Compare data field value of the element with the value that you are looking for.

```
while (node-> next  != NULL )
 { if ( node->data == skaicius)
        {  printf (“Node was found \n”);
              exit (0);
        }
   node = node-> next;
 }
```

Element of the list

```
struct node {
        int data;
        struct node *next;
};
```

```
class node {
        private: int data;
                 node *next;
        public:
                 node()
                 { }
    };
```

Example

Write program to implement linked list and the following functions for the nodes: add, remove, print, destroy.

```
#include <stdio.h>
#include <stdlib.h>
 struct node{
            int data;
            struct node
*next;
};
```

Example

```
struct node* add (struct node *head, int data) {
        struct node *tmp;
        if (head == NULL){
                        head = (struct node *)malloc(sizeof (struct node));
                        if (head == NULL){
                        printf("Error! memory is not available\n");
                                        exit(0);
                        }
                        head-> data = data;
                        head-> next = NULL;
        }else {
                        tmp = head;
                        while (tmp-> next != NULL)
                                        tmp = tmp-> next;
                  tmp-> next = (struct node *)malloc(sizeof(struct node));
                        if(tmp -> next == NULL)
                        {         printf("Error! memory is not available\n");
                                  exit(0);
                        }
                        tmp = tmp-> next;
                        tmp-> data = data;
                        tmp-> next = NULL;
        } return head;       }
```

Example

```
void printlist (struct node *head)
{
            struct node *current;
            current = head;
            if (current != NULL)
            {
            do
            {  printf ("%d\t",current->data);
               current = current->next;
            }
                while (current != NULL);

            printf ("\n");
            }
            else
            printf ("The list is empty \n");
}
```

```
void destroy (struct node *head)
{
    struct node *current, *tmp;

    current = head->next;
    head->next = NULL;

     while(current != NULL) {
         tmp = current->next;
         free(current);
         current = tmp;
      }
}
```

```
void remove (struct  node *e)
{  struct node *d = e->next;
    if ( d != NULL )
        e->next = d->next;
     free(d);
}
```

Example

```
void main()
{
        struct node *head = NULL;
        head = add(head,1);       /* 1 */
        printlist(head);

        head = add(head,20);      /* 20 */
        printlist(head);

        head = add(head,10);      /* 1 20 10 */
        printlist(head);

        head = add(head,5);        /* 1 20 10 5*/
        printlist(head);

        destroy(head);
        getchar();
}
```

Doubly linked lists

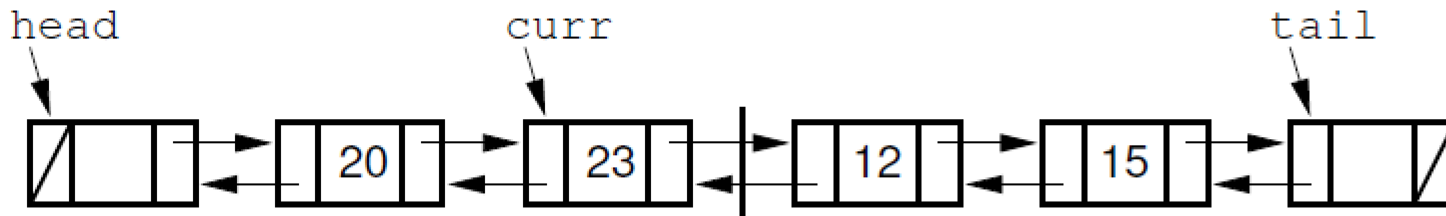
A **doubly linked list** allows convenient access from a list node to the next node and also to the preceding node on the list.

The doubly linked list node accomplishes this in the obvious way by storing **two pointers**: one to the node following it (as in the singly linked list), and a second pointer to the node preceding it.

The most common reason to use a doubly linked list is because it is easier to implement than a singly linked list.

While the code for the doubly linked implementation is a little longer than for the singly linked version, it tends to be a bit more "obvious" in its intention, and so easier to implement and debug.

Doubly linked lists



Like singly linked list implementation, the doubly linked list implementation makes use of a header node**. A tailer node is added to the end of the list.**

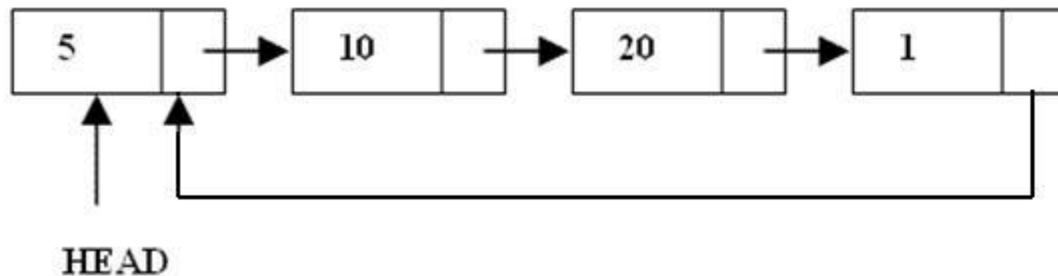
The tailer is similar to the header, in that it is a node that **contains no value, and it always exists**.

When the doubly linked list is initialized, the header and tailer nodes are created. Data member **head** points to the header node, and **tail** points to the tailer node. The purpose of these nodes is to simplify the **insert**, **append**, and **remove** methods by eliminating all need for special-case code when the list is empty.

Singly-circularly-linked list

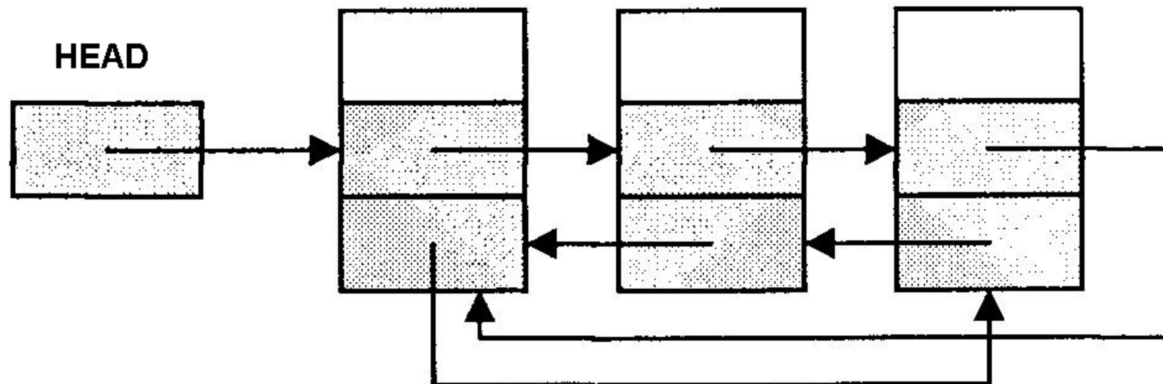
With singly-circularly-linked list a pointer to the last node gives access to the first node, by following one link.

Thus, in applications that require access to both ends of the list, a circular structure allows one to handle the structure by a single pointer, instead of two.



Doubly-circularly-linked list

Doubly-circularly-linked list is based on doubly linked list. One of the pointer of the first node give access to the last node while one pointer of the last node give access to the first node.



Homework tasks

No.1 Find min/max elements of the array and index number of min/max elements.

No.2 Find element of the array with the smallest deviation and remove it.

No.3 Write program that swaps values of two variables without using additional variable.

No.4 Find the date (yyyy.mm.dd) from user defined date by adding/deleting one day.