

Laboratorul 5 - TDA listă. Aplicații.

Responsabil: Mihail-Alexandru Balan

Data publicării: 25 martie 2007

Data ultimei modificări: 25 martie 2007

Obiective

În urma parcurgerii acestui laborator studentul va fi capabil:

- să înțeleagă structura tipului de date *listă*
- să descrie și să folosească diversele implementări ale TAD *listă*
- să folosească în aplicații structura de date *listă*

Prezentare teoretică

O listă este o instanță a unui tip de date abstract ce formalizează conceptul de colecție ordonată de entități. În mod minimal, o listă are următoarele proprietăți și operații:

- Operații:
 - **Add** - adaugă un element (entitate) la listă. Adăugarea se poate face la sfârșitul listei, sau într-o poziție arbitrară.
 - **Remove** - șterge un element din listă. Identificarea elementului ce urmează a fi eliminat se poate face pe baza poziției sale (iterator), sau a conținutului.
 - **Get** - consultă un element din listă. Identificarea elementului se face printr-un iterator.
 - **Update** - actualizează informația conținută de un element din listă.
- Proprietăți:
 - **lungimea** - numărul de elemente din listă. De cele mai multe ori, se implementează într-o funcție de tipul `GetSize()`
 - **tipul** - felul elementelor din listă. Întâlnită mai ales în implementările din limbaje care suportă tipuri generice (C++, Java, .NET)

Implementări

Pentru implementarea listelor, există două modalități de bază: folosind liste înlănțuite (simplu sau dublu înlănțuite) sau folosind arrayuri dinamice.

În cazul listelor înlănțuite, fiecare nod din listă va conține pe lângă informația utilă și legături către nodurile vecine (liste dublu înlănțuite), sau către nodul următor (liste dublu înlănțuite). Alocând dinamic nodurile pe măsură ce este nevoie de ele, practic se pot obține liste de lungime limitată doar de cantitatea de memorie accesibilă programului.

În cazul arrayurilor dinamice, elementele sunt stocate în vectori de tipul specificat. În momentul în care, prin adăugarea unui element s-ar depăși lungimea vectorului, acesta este realocat și extins cu un factor specificat (fixat în implementare sau setat de către utilizator). Această implementare are avantajul vitezei de acces sporite (elementele sunt în locații succesive de memorie), dar este limitată de cantitatea de **memorie contiguă** accesibilă programului.

Biblioteca Cobble

În cadrul acestui laborator se va folosi listele implementate în biblioteca de tipuri de date Cobble. Această bibliotecă are 2 implementări de liste, perfect similare din punct de vedere al interfeței: liste dublu înlănțuite și arraylisturi. În ceea ce urmează este prezentat câte un exemplu de folosire a celor

două tipuri de liste, împreună cu makefile-ul aferent.

LinkedList

```
#include <stdio.h>
#include <stdlib.h>
#include <Cobble/linkedlist.h>

int main(int argc, char *argv[])
{
    int *nr;
    LinkedList list;

    /*declaring a new list*/
    list = LL_New();
    LinkedListIterator it;

    /*adding elements to the list*/
    do
    {
        nr=(int*)malloc(sizeof(int));
        scanf("%d",nr);
        LL_Add(list, LL_Begin(list), nr);
    }while (*nr>0);

    printf("List size: %d\n", LL_Size(list));

    /*listing the queue*/

    for (it = LL_Begin(list); it!= LL_End(list); it = LL_Next(list, it))
    {
        nr = LL_Get(list, it);
        printf("%d\n",*nr);
        fflush(stdout);
    }

    /*Clearing the queue from memory*/
    LL_Free(list);

    return 0;
}
```

ArrayList

```
#include <stdio.h>
#include <stdlib.h>
#include <Cobble/arraylist.h>

int main(int argc, char *argv[])
{
    int *nr;
    ArrayList list;

    /*declaring a new list*/
    list = AL_New(10);
    ArrayListIterator it;

    /*adding elements to the list*/
```

```

do
{
    nr=(int*)malloc(sizeof(int));
    scanf("%d",nr);
    AL_Add(list, AL_Begin(list), nr);
}while (*nr>0);

printf("List size: %d\n", AL_Size(list));

/*listing the queue*/

for (it = AL_Begin(list); it!= AL_End(list); it = AL_Next(list, it))
{
    nr=AL_Get(list, it);
    printf("%d\n",*nr);
    fflush(stdout);
}

/*Clearing the queue from memory*/
AL_Free(list);

return 0;
}

```

Makefile:

```

#in cazul unei instalari de MinGW pe Windows se va specifica locatia in care
#sunt instalate bibliotecile GCCului (e.g. C:\MinGW\gcc\lib\))
LIB_BASE = /usr/lib

```

```

arraylist:
    gcc -o alSample ArrayListSample.c -lcobble -L $(LIB_BASE)/Cobble

linkedlist:
    gcc -o llSample LinkedListSample.c -lcobble -L $(LIB_BASE)/Cobble

run-al:
    ./alSample
run-ll:
    ./llSample

```

Notă:

- Pentru instalarea bibliotecii Cobble sunt necesari 2 pași:
 - copierea fișierelor antet (*.h) în directorul include implicit al compilatorului (pe o mașină Linux acesta este /usr/include)
 - copierea bibliotecii statice (libcobble.a) în directorul de librării implicit al compilatorului (pe o mașină Linux acesta este /usr/lib)
- makefileul și sursele prezentate anterior presupun faptul că librăria Cobble a fost instalată în directorul /Cobble (din /usr/include respectiv /usr/lib)

Aplicații

1. Evaluarea unui polinom într-un punct. Dându-se un polinom prin gradul său maxim și prin coeficienții săi, se cere evaluarea sa într-un punct specificat de utilizator. Se va folosi tipul de

date ArrayList.

2. Operații cu polinoame. Fiind date două polinoame în forma specificată la problema 1. se cere să se afișeze suma și produsul celor două polinoame.