Structuri de Date
Laboratorul 2: Liste simplu-înlănțuite

Tudor Berariu – 2016
Mihai Nan – 2021

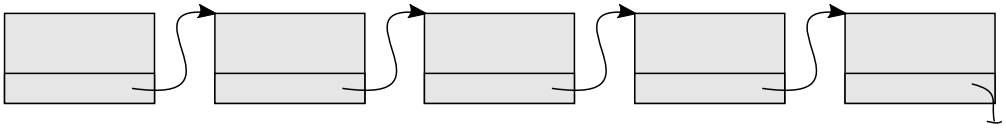
8 martie 2021

1. Introducere

Scopul acestui laborator îl reprezintă implementarea unei liste simplu înlănțuite care păstrează elementele în ordine. Obiectivul este acela de a vă familiariza cu lucrul cu listele alocate dinamic și de a face o comparație cu structura ordonată din laboratorul anterior. Laboratorul are 4 cerințe care urmăresc:

- definirea unei interfețe de lucru cu liste;
- implementarea funcțiilor pentru lucrul cu liste;
- compararea a trei abordări pentru implementarea funcțiilor ce parcurg o listă;
- rezolvarea unei probleme simple folosind liste.

2. Liste ordonate simplu înlănțuite



Deoarece într-o listă simplu înlănțuită se alocă individual memorie pentru fiecare element, nu mai există garanția (precum în cazul vectorilor) că elementele listei ocupă un spațiu contiguu în memorie. Alături de fiecare element al listei se reține și un pointer către adresa de memorie la care se găsește următorul element.

```
1  typedef struct node {
2      T value;
3      struct node* next;
4  } TNode, *TSortedList;
```

Cerința 1 (1 punct) Scrieți în fișierul `SortedList.h` o implementare minimală (ca în exemplul oferit pentru funcția `create`) pentru fiecare funcție dintre cele descrise mai jos. Pentru această cerință trebuie doar să scrieți corect tipul fiecărei funcții și tipurile argumentelor.

- Funcția `create` primește ca argument o valoare de tip `T` și întoarce o listă care conține un singur nod ce conține valoarea primită ca parametru.

  Exemplu:

```
1  TSortedList create(T value) {
2      // TODO (se va completa la Cerința 2)
3      return NULL;
4  }
```

- Funcția `isEmpty` verifică dacă o listă este vidă. Această funcție primește ca parametru o listă și returnează 1 dacă lista respectivă este vidă și 0 altfel.
- Funcția `contains` primește o listă și un element și întoarce 1 dacă elementul se găsește în listă și 0 altfel;
- Funcția `insert` primește o listă și un element și introduce elementul în listă, păstrând (la final) lista ordonată crescător. Funcția va întoarce lista rezultată.
- Funcția `deleteOnce` primește o listă și un element și șterge prima apariție a acelui element din listă (dacă el există). Funcția va întoarce lista rezultată.
- Funcția `length` primește o listă și întoarce lungimea ei (tipul returnat este `long`);
- Funcția `getNth` primește o listă și un număr natural `N` și întoarce elementul de pe poziția `N` din listă (primul element se află pe poziția 1).
- Funcția `freeTSortedList` primește o listă și eliberează toată memoria alocată pentru elementele ei. Funcția returnează lista vidă (prin convenție, aceasta o să fie `NULL`).

Pentru a verifica faptul că ați scris corect antetul, compilarea trebuie să reușească.

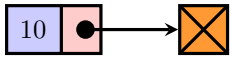
```
$ make clean
rm -f eratostene length testSortedList
$ make
gcc -std=c9x -g eratostene.c -o eratostene
gcc -std=c9x -g length.c -o length
gcc -std=c9x -g testSortedList.c -o testSortedList
```

Cerința 2 (7 puncte) Implementați funcțiile de mai sus. Verificați-vă pe parcurs cu testerul. Ordinea indicată de implementare a funcțiilor este: `create`, `isEmpty`, `insert`, `length`, `contains`, `deleteOnce`, `getNth`, `freeTSortedList`.

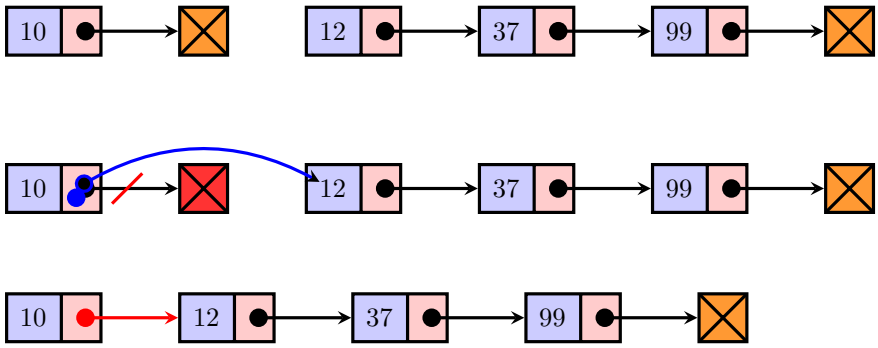
Indicații de rezolvare:

- Atunci când aveți nevoie să introduceți un nod nou în listă, puteți folosi funcția `create`.
- Atunci când implementați funcția `insert` trebuie să tratați toate cazurile:

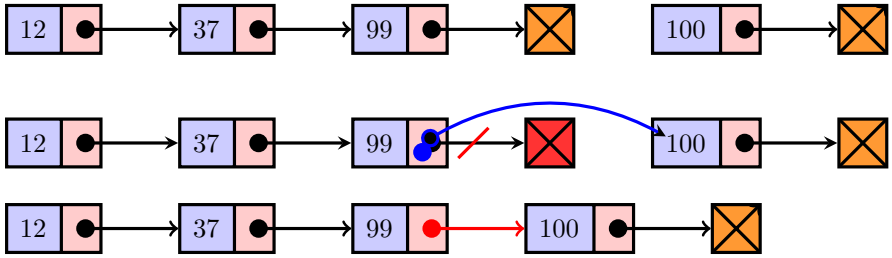
1. Inserarea unei valori într-o listă vidă;



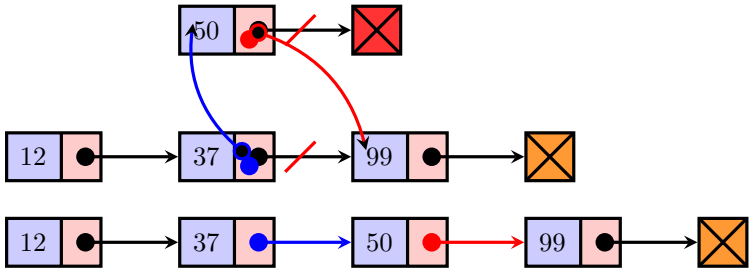
2. Adăugarea unei valori la începutul listei;



3. Adăugarea unei valori la finalul listei;



4. Adăugarea în interiorul listei.



- Atunci când implementați funcția `deleteOnce` trebuie să tratați toate cazurile:

1. Valoarea care urmează să fie ștearsă este reținută în primul nod al listei;
2. Valoarea care urmează să fie ștearsă este reținută în ultimul nod al listei;
3. Valoarea care urmează să fie ștearsă este reținută într-un nod interior al listei;

Exemplu de testare:

```
$ make test
gcc -std=c9x -g -O0 testSortedList.c -o testSortedList
valgrind --leak-check=full ./testSortedList
==17225== Memcheck, a memory error detector
==17225== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==17225== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==17225== Command: ./testSortedList
==17225==
create       - Toate testele au trecut! Puncte: 0.25
isEmpty      - Toate testele au trecut! Puncte: 0.25
insert       - Toate testele au trecut! Puncte: 2.00
length       - Toate testele au trecut! Puncte: 0.50
contains     - Toate testele au trecut! Puncte: 0.50
deleteOnce   - Toate testele au trecut! Puncte: 2.00
getNth       - Toate testele au trecut! Puncte: 0.50
*Programul se va verifica cu valgrind!* Puncte: 1.00
Toate testele au trecut! Felicitari!
Punctajul obtinut este: 6.00
Teste rulate: 7
==17225==
==17225== HEAP SUMMARY:
==17225==     in use at exit: 0 bytes in 0 blocks
==17225==   total heap usage: 421 allocs, 421 frees, 8,648 bytes allocated
==17225==
==17225== All heap blocks were freed -- no leaks are possible
==17225==
==17225== For counts of detected and suppressed errors, rerun with: -v
==17225== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Cerința 3 (2 puncte) În fișierul `length.c` este deja implementată o funcție care construiește o listă cu un număr dat de elemente.

Implementați în trei variante o funcție care calculează lungimea listei:

1. iterativ (utilizând o buclă `while`/`for` pentru a parcurge lista);
2. recursiv pe stivă (se adună 1 pentru elementul curent după ce a fost calculat numărul de elemente ce îi urmează);
3. recursiv pe coadă (se folosește un acumulator; rezultatul funcției curente este dat de rezultatul apelului recursiv).

Observați care dintre acestea este mai eficientă pentru diferite dimensiuni ale liste: mii, sute de mii, milioane de elemente.

```
$ make length
gcc -std=c9x -g -O0 length.c -o length
$ ./length
Size: 200000 |            Iterative: 0.001332
Size: 200000 |       Tail Recursive: 0.002954
Size: 200000 |      Stack Recursive: 0.002010
```

Noțiuni teoretice și exemple:

În timpul rulării, un program utilizează o zonă de memorie, denumită **stivă**, cu informațiile de care are nevoie, în diferite momente. La fiecare apel de funcție, stiva crește. La fiecare revenire dintr-un apel, cu o anumită valoare, stiva se reduce. Când recursivitatea este foarte adâncă - există multe apeluri de funcție realizate, din care nu s-a revenit încă - stiva devine foarte mare.

Acest lucru influențează:

- ***memoria*** - în unele situații, trebuie reținute foarte multe informații.
- ***timpul*** - în locul redimensionării stivei, ar putea fi utilizat pentru alte calcule, programul rezultat fiind, astfel, mai rapid.

Recursivitatea pe stivă

O funcție este recursivă pe stivă dacă apelul recursiv este parte a unei expresii mai complexe, fiind necesară reținerea de informații, pe stivă, pe avansul în recursivitate.

```
1  int factorial(int nr) {
2      if(nr == 0)
3          return 1;
4      else
5          nr * factorial(nr - 1);
6  }
```

Recursivitatea pe coadă

O funcție este recursivă pe coadă dacă valoarea întoarsă de apelul recursiv constituie valoarea de retur a apelului curent, nefiind necesară reținerea de informație pe stivă.

În general, pentru a trece din recursivitatea pe stivă în cea pe coada, se utilizează o variabilă de acumulare.

```
1  int tail_recursion(int nr, int acumulator) {
2     if(nr == 0)
3        return acumulator;
4     else
5        return tail_recursion(nr - 1, nr * acumulator);
6  }
7
8  int factorial(int nr) {
9     return tail_recursion(nr, 1);
10 }
```

Bonus Se dorește implementarea metodei *ciurului lui Eratostene* pentru aflarea numerelor prime mai mici decât un număr dat. În fișierul `eratostene.c` implementați funcția

```
SortedList* getPrimes(int N)
```

în care trebuie să folosți doar bucle ale căror condiții să fie exclusiv rezultatul evaluării funcției `isEmpty`. Folosoți-vă de funcția

```
SortedList* getNaturals(int A, int B)
```

deja implementată, care construiește o listă cu toate numerele naturale de la A la B.

Explicații:

Ciurul lui Eratostene este un algoritm simplu și vechi de descoperire a tuturor numerelor prime până la un întreg specificat.

Pașii pe care îi execută acest algoritm sunt următorii:

1. Se scrie o listă a numerelor de la 2 la cel mai mare număr ce urmează a fi testat pentru primalitate. Numim această listă lista A.

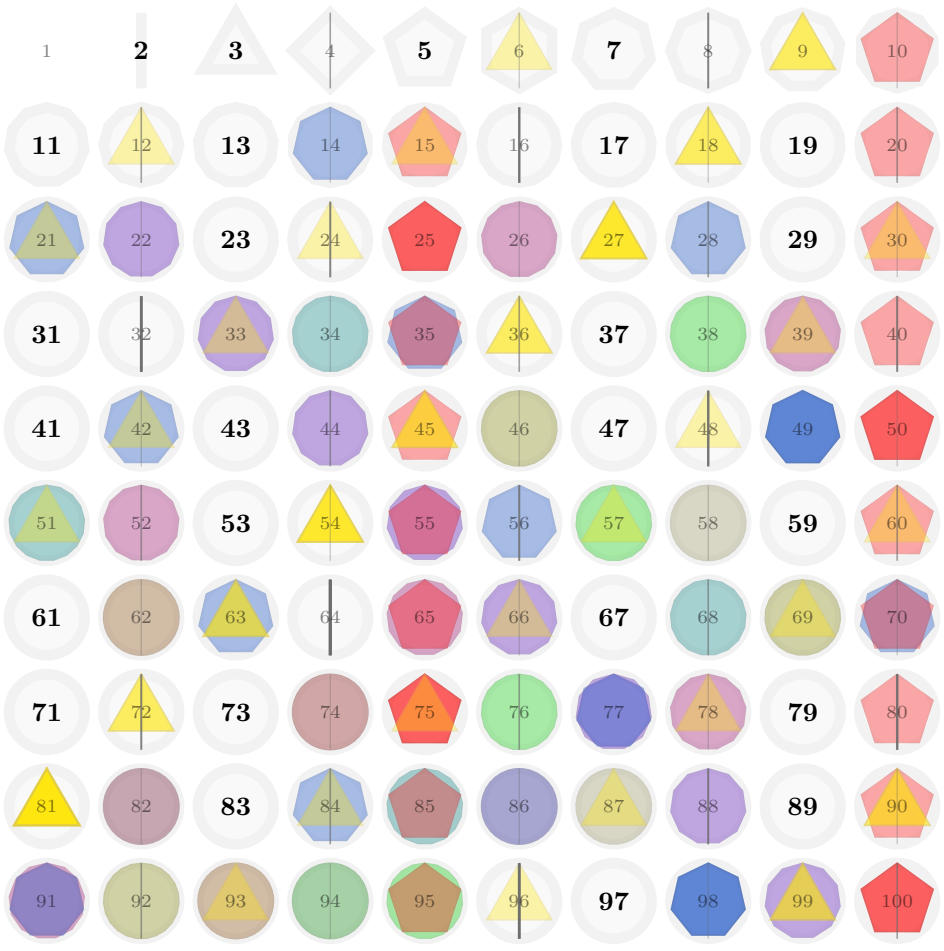
2. Se analizează numărul 2, primul număr prim găsit. Acest număr este păstrat în listă, însă eliminăm toate numerele din listă care sunt multiplii pentru 2.

3. Avansăm în listă, iar al doilea element întâlnit în lista A după aplicarea pasului 2 este numărul prim 3.

4. Se analizează numărul 3, al doilea număr prim găsit. Acest număr este păstrat în listă, însă eliminăm toate numerele din listă care sunt multiplii pentru 3.

5. Se repetă pașii 3 și 4 până când se epuizează toate numerele din lista A.

6. În final, lista A va conține doar numerele prime.



În figura de mai sus este prezentat un exemplu de aplicare pentru a determina toate numerele prime mai mici decât 100.

3. Informații utile

Pentru cerința 2, putem testa și individual funcțiile implementate. Pentru acest lucru, va trebui să rulăm executabilul `testSortedList` utilizând argumente în linia de comandă:

- 1 – pentru a testa `create` (0.25 puncte);
- 2 – pentru a testa `isEmpty` (0.25 puncte);
- 3 – pentru a testa `insert` (2 puncte);
- 4 – pentru a testa `length` (0.5 puncte);
- 5 – pentru a testa `contains` (0.5 puncte);
- 6 – pentru a testa `deleteOnce` (2 puncte);
- 7 – pentru a testa `getNth` (0.5 puncte);

Observație: Punctajul pentru `freeTSortedList` se va acorda manual dacă nu există erori sau pierderi de meorie!

Exemple de testări:

- Testarea funcției `isEmpty`

```
$ ./testSortedList 2
isEmpty  - Toate testele au trecut! Puncte: 0.25
Toate testele selectate au trecut!
Punctajul obtinut este: 0.25
Teste rulate: 1
```

- Testarea funcțiilor `create` și `length`

```
$ ./testSortedList 1 4
create  - Toate testele au trecut! Puncte: 0.25
length  - Toate testele au trecut! Puncte: 0.50
Toate testele selectate au trecut!
Punctajul obtinut este: 0.75
Teste rulate: 2
```

- Testarea tuturor funcțiilor de la **Cerința 2**

```
$ ./testSortedList
create       - Toate testele au trecut! Puncte: 0.25
isEmpty      - Toate testele au trecut! Puncte: 0.25
insert       - Toate testele au trecut! Puncte: 2.00
length       - Toate testele au trecut! Puncte: 0.50
contains     - Toate testele au trecut! Puncte: 0.50
deleteOnce   - Toate testele au trecut! Puncte: 2.00
getNth       - Toate testele au trecut! Puncte: 0.50
*Programul se va verifica cu valgrind!* Puncte: 1.00
Toate testele au trecut! Felicitari!
Punctajul obtinut este: 6.00
Teste rulate: 7
```