Structuri de Date
Laboratorul 2: Liste simplu-înlănțuite

Tudor Berariu

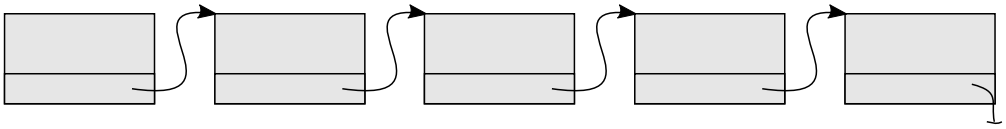
22 februarie 2019

1. Introducere

Scopul acestui laborator îl reprezintă implementarea unei 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 SortedList {
2    T value;
3    struct SortedList* next;
4  } SortedList;
```

Cerința 1 Scrieți în `SortedList.h` o implementare non-funcțională pentru fiecare funcție dintre cele descrise mai jos. Atenție, acolo unde este necesar, trebuie trimisă adresa pointerului care reprezintă lista. Atunci când nu este necesar, se va trimite o copie a acestui pointer. Pentru această cerință trebuie doar să scrieți corect tipul fiecărei funcții și tipurile argumentelor.

- Funcția `init` nu întoarce nicio valoare și inițializează un pointer la `NULL` (convenția pentru lista vidă).

```
1  void init(SortedList** list) { ; }
```

- Funcția `isEmpty` verifică dacă o listă este vidă.
- 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 nu întoarce nimic.
- Funcția `deleteOnce` primește o listă și un element și șterge prima apariție a acelui element din listă (dacă el există). Funcția nu întoarce nimic.
- Funcția `length` primește o listă și întoarce lungimea ei (tipul este `long`);
- Funcția `getNth` primește o listă și un număr `N` și întoarce elementul de pe poziția `N` din listă (primul element se află pe poziția 1).
- `freeSortedList` care primește o listă și eliberează toată memoria alocată pentru elementele ei.

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 Implementați funcțiile de mai sus. Verificați-vă pe parcurs cu testerul. Ordinea indicată de implementare a funcțiilor este: `init`, `isEmpty`, `insert`, `length`, `contains`, `deleteOnce`, `getNth`.

```
$ make test
valgrind --leak-check=full ./testSortedList
==3254== Memcheck, a memory error detector
==3254== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==3254== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==3254== Command: ./testSortedList
==3254==
. Testul Init&IsEmpty a fost trecut cu succes!  Puncte: 0.05
. . . . Testul Insert a fost trecut cu succes!  Puncte: 0.20
. . . . Testul Length a fost trecut cu succes!  Puncte: 0.05
. . . Testul Contains a fost trecut cu succes!  Puncte: 0.05
. . Testul DeleteOnce a fost trecut cu succes!  Puncte: 0.20
. . . . Testul GetNth a fost trecut cu succes!  Puncte: 0.05
......... Testul Mega a fost trecut cu succes!  Puncte: 0.10
....... *Programul se va verifica cu valgrind*  Puncte: 0.10.

Scor total: 0.80 / 0.80

==3254==
==3254== HEAP SUMMARY:
==3254==     in use at exit: 0 bytes in 0 blocks
==3254==   total heap usage: 73 allocs, 73 frees, 1,208 bytes allocated
==3254==
==3254== All heap blocks were freed -- no leaks are possible
==3254==
==3254== For counts of detected and suppressed errors, rerun with: -v
==3254== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Cerința 3 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` 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
```

Cerința 4 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.