Laborator 05 - Liste generice

Responsabili:

- Claudia Cârdei
- Alex Fărcășanu

Obiective

În urma parcurgerii acestui laborator studentul va fi capabil:

- să înţeleagă structura tipului de date listă.
- să descrie şi să folosească diversele implementări ale TAD listă.
- să folosească în aplicaţii structura de date listă (să descrie soluţia unor aplicaţii ce folosesc liste pentru modelarea datelor utilizate).
- sa implementeze structurile studiate folosind liste.

Liste

O listă este o instanţă a unui tip de date abstract ce formalizează conceptul de colecţie ordonată de entităţi. În mod minimal, o listă este caracterizată prin:

- Operaţii
  - `Add` - adaugă un element (entitate) la listă. Adăugarea se poate face la începutul listei, sfârşitul listei, sau într-o poziţie arbitrară.
  - `Remove` - şterge un element din listă. Identificarea elementului ce urmează a fi eliminat se poate face pe baza poziţiei sale (iterator), sau a conţinutului. De asemenea ştergerea se poate face de la începutul sau de la sfîrşitul listei.
  - `Get` - consultă un element din listă. Identificarea elementului se face prin poziţie.
  - `Update` - actualizează informaţia conţinută de un element din listă.
- Proprietăţi:
  - `lungimea` - numărul de elemente din listă. De cele mai multe ori, se implementează într-o funcţie de tipul GetSize()
  - `tipul` - felul elementelor din listă. Întâlnită mai ales în implementările din limbaje care suportă tipuri generice (C++, Java, .NET)

Cum se implementează?

Pentru implementarea listelor, există două modalităţi de bază: folosind liste înlănţuite (simplu sau dublu înlănţuite) sau folosind array-uri dinamice.

În cazul listelor înlănţuite, fiecare nod din listă va conţine pe lângă informaţia utilă şi legături către nodurile vecine (liste dublu înlănţuite), sau către nodul următor (liste simplu înlănţuite). Alocând dinamic nodurile pe măsură ce este nevoie de ele, practic se pot obţine liste de lungime limitată doar de cantitatea de memorie accesibilă programului.

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

Tipuri de liste înlănţuite

Listă liniară simplu înlănţuită

Search

- Reguli generale și de notare
- Catalog
- Concursuri
- Calendar

Laboratoare

- Laborator 1 - Introducere in C++
- Laborator 2 - Noțiuni de C++
- Laborator 3 - Stive
- Laborator 4 - Cozi
- Laborator 5 - Liste generice
- Laborator 6 - HashTable
- Laborator 7 - Grafuri
- Laborator 8 - Arbori Binari
- Laborator 9 - Arbori Binari de Căutare
- Laborator 10 - Heap-uri
- Laborator 11 - Treap-uri
- Laborator 12 - Mulțimi Disjuncte

Teme

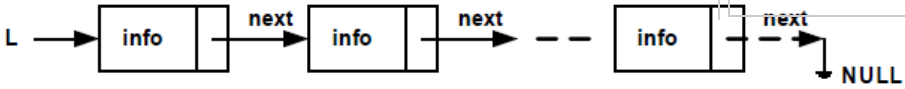
- Tema 1
- Tema 2
- Tema 3
- Tema 4

Resurse

- Debugging
- Data Structure Visualization

Table of Contents

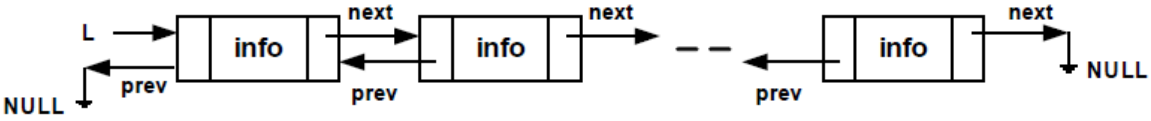
Are o singură legatură la fiecare nod. Această legatură indică întotdeauna următorul nod din listă, sau o valoare nulă (dacă suntem la finalul listei), sau o listă liberă (pentru identificarea ei).



Listă liniară dublu-înlănţuită

Fiecare nod din listă liniara dublu înlănţuită are două legături:

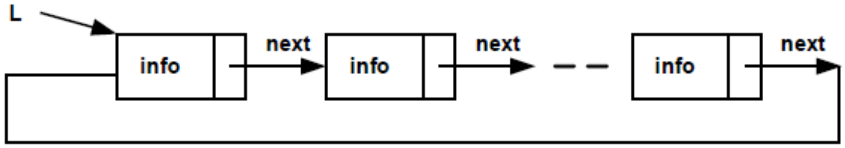
- una leagă nodul actual de nodul de dinaintea lui, sau leagă nodul actual cu o listă libera, sau cu o listă care are o valoare nulă dacă aceasta este la începutul primului nod.
- cealaltă legatură leagă nodul actual de o listă care are o valoare nulă sau cu o listă liberă dacă această reprezintă nodul final.



Listă circulară simplu-înlănţuită

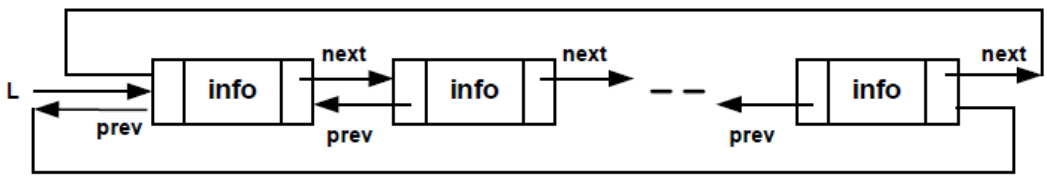
Primul şi ultimul nod sunt legate împreună. Pentru a parcurge o listă circular înlănţuită se începe de la oricare nod şi se urmăreşte lista prin aceasta direcţie aleasă până când se ajunge la nodul de unde s-a pornit parcurgerea (lucru valabil şi pentru listele circulare dublu-înlănţuite).

Fiecare nod are o singură legatură, similar cu listele liniare simplu-înlănţuite, însă, diferenţa constă în legătura aflată după ultimul nod ce îl leagă pe acesta de primul nod. La fel ca şi în listele liniare simplu-înlănţuite, nodurile noi pot fi inserate eficient numai dacă acestea se află după un nod care are referinţe la acesta. Din acest motiv, este necesar să se menţină numai o referinţă către ultimul element dintr-o listă circulară simplu-înlănţuita, căci aceasta permite o inserţie rapidă la nodul de început al listei, şi de asemenea, permite accesul la primul nod prin legatura dintre acesta şi ultimul nod.



Listă circulară dublu-înlănţuită

Fiecare nod are două legături, asemanator ca şi la listele liniare simplu-înlănţuite, însă diferenţa este că listele circulare dublu-înlănţuite legatura dinaintea primului nod îl leagă pe acesta de ultimul nod, şi legatura de dinaintea ultimului nod catre îl leagă pe acesta de primul nod. La fel ca şi la listele liniare dublu-înlănţuite, operaţiile de inserţie şi ştergere pot fi făcute în orice punct din listă, cu acces la oricare nod apropiat.



Exemplu

list.h

```
/* list.h */
#ifndef __LIST__H
#define __LIST__H
```

```
template <typename T>struct Node {
 T value;
 Node<T> *next;
 Node<T> *prev;
};

template <typename T>class LinkedList {
private:
 Node<T> *pFirst, *pLast;

public:
 // Constructor
 LinkedList();
 // Destructor
 ~LinkedList();

 /* Adauga un nod cu valoarea == value la inceputul listei. */
 void addFirst(T value);

 /* Adauga un nod cu valoarea == value la sfarsitul listei. */
 void addLast(T value);

 /* Elimina elementul de la inceputul listei si intoarce valoare
 T removeFirst();

 /* Elimina elementul de la sfarsitul listei listei si intoarce
 T removeLast();

 /* Elimina prima aparitie a elementului care are valoarea == va
 T removeFirstOccurrence(T value);

 /* Elimina ultima aparitie a elementului care are valoarea == v
 T removeLastOccurrence(T value);

 /* Afiseaza elementele listei pe o singura linie, separate prin
 void printList();

 /* Intoarce true daca lista este vida, false altfel. */
 bool isEmpty();
};

#endif
```

Exerciții

1) [**3p**] Implementați în header-ul definit anterior funcțiile pentru o listă liniară dublu înlănțuită.

- [**1p**] constructor, destructor(eliberați memoria folosită) și isEmpty.
- [**1p**] addFirst și addLast.
- [**1p**] removeFirst si removeLast.

2) [**2p**] Implementați functia removeFirstOccurrence și demonstrați funcționarea acesteia printr-un cod simplist.

3) [**2p**] Implementați o stivă folosind liste.

stack.h

```
#ifndef __STACK__H
#define __STACK__H

template<typename T>
class Stack {
public:
    // Constructor
    Stack();

    // Destructor
    ~Stack();

    void push(T x);
```

```
    T pop();
    T peek();
    int isEmpty();

private:
    // Vectorul de stocare.
    LinkedList<T> stackList;
    // De ce nu mai este nevoie sa retinem topLevel?
};
#endif
```

Interviu

Această secțiune nu este punctată și încearcă să vă facă o oarecare idee a tipurilor de întrebări pe care le puteți întâlni la un job interview (internship, part-time, full-time, etc.) din materia prezentată în cadrul laboratorului.

1. Scrieti un program care sa afizeseze elementul din mijlocul unei liste fara a folosi un contor si fara a sti dimensiunea listei.

2. Scrieti un program care se detecteze o bucla intr-o lista simplu inlantuita.

3. Scrieți o funcție care inversează o listă simplu înlănțuită fără a folosi memorie auxiliară.

Resurse

[1] C++ Reference

[2] Linked List Queue Visualization

[3] Linked List Stack Visualziation

[4] List C++