Laborator 3 - Templates. ArrayList și LinkedList

Responsabili

- Gabriel Bercaru [mailto:mailto:gabriel.bercaru10@gmail.com]
- Alexandru-Mihai Stroie [mailto:mailto:andistroie@gmail.com]
- Cosmin Petrișor [mailto:mailto:cosmin.ioan.petrisor@gmail.com]

Obiective

În urma parcurgerii acestui laborator studentul va:

- înțelege conceptul de template
- putea implementa propriul model de ArrayList
- putea implementa propriul model de LinkedList
- putea compara cele două structuri de date

Templates

În C++, conceptul de "Templates" permite crearea de funcții/clase care folosesc variabile ale căror tipuri nu sunt inițial cunoscute. Astfel, implementarea va putea fi adaptată mai multor tipuri de date fără a recurge la duplicarea codului.

Sintaxa pentru acestea este:

```
template <class identifier> declarație;
template <typename identifier> declarație;
```

declaratie poate fi fie o funcție, fie o clasă. Nu există nicio diferență între keyword-ul *class* și *typename* - important este că ceea ce urmează după ele este un placeholder pentru un tip de date.

Function Template

În primul rând template-urile pot fi aplicate funcțiilor.

Sintaxa pentru declararea unei funcții care folosește template:

templateMax.cpp

```
template <typename T>
T getMax(T a, T b) {
    return a > b ? a : b;
}
```

Funcția anterioară întoarce maximul dintre cele două valori date ca parametru. Singura diferență față de o funcție `getMax()` care ar returna, de exemplu, maximul dintre două numere întregi este că tipul celor doi parametri este T, nu int.

Prin adăugarea construcției `template <typename T>` înaintea antetului, se declară un parametru de template denumit T. Parametrii de template sunt un tip special de parametri prin care se pot transmite tipuri de date, spre deosebire de parametrii de funcție prin care se transmit valori. Astfel, în toată implementarea funcției, T va putea ține locul unui tip obișnuit de date (int, float etc.).

Dacă se dorește calculul maximului dintre două valori de tip int, codul care apelează funcția ar arăta astfel:

```
int intMax = getMax<int>(0, 1); // intMax este 1
```

Similar, codul pentru a calcula maximul dintre două valori de tip double:

```
float doubleMax = getMax<double>(0.5, 1.0); // doubleMax este 1.0
```

Pentru a asigura funcționarea corectă a unei funcții template, compilatorul va genera, in funcție de nevoie, câte o variantă diferită pentru funcția respectivă.

Astfel, în urma următoarelor apeluri:

```
int intMax = getMax<int>(0, 1);
float floatMax = getMax<float>(0.5, 1.0);
```

compilatorul va genera două variante ale funcției `getMax()`: una în care parametrul de template T a fost înlocuit cu `int`, iar alta în care a fost înlocuit cu `float`. Procesul de "rescriere" a funcției este transparent programatorului.

Dacă valorile de tip T sunt date ca parametri pentru funcție, atunci compilatorul poate infera tipul de date pentru care să apeleze funcția, astfel că nu mai este nevoie de menționarea explicită a tipului în apelul funcției. Următoarele intrucțiuni vor fi valide:

```
int intMax = getMax(0, 1);
float floatMax = getMax(0.5, 1.0);
```

Class Template

Conceptul de template poate fi aplicat și claselor, nu doar funcțiilor. Astfel, se realizează genericizarea unei clase: instanțele unei clase pot conține membri de diferite tipuri, nu doar de un tip predefinit.

Sintaxa pentru declararea unei clase template este similară ca în cazul funcțiilor template.

Mai jos este prezentat un exemplu de clasă care poate stoca un array de elemente al căror tip nu este cunoscut în momentul scrierii clasei:

GenericContainer.h

```
template <typename T>
class GenericContainer {
public:
    int size;
    int maxCapacity;

    GenericContainer(int maxCapacity) {
        this->maxCapacity = maxCapacity;
        size = 0;

        dataArray = new T[maxCapacity];
    }

    ~GenericContainer() {
        delete[] dataArray;
    }

    T *getArray() {
        return dataArray;
    }

private:
    T *dataArray;
}
```

Instanțierea clasei pentru stocarea a maxim 10 elemente de tip `double`, respectiv maxim 5 elemente de tip `int` se face astfel:

main.cpp

```
#include "GenericContainer.h"

int main() {
    GenericContainer<double> doubleContainer(10);
    GenericContainer<int> intContainer(5);
    return 0;
}
```

Se observă și existența metodelor generice în clasă, nu doar a variabilelor. Practic, oriunde folosim tipul de date T în clasă, este înlocuit cu tipul pe care îl specificăm.

La fel ca în cazul funcțiilor, compilatorul analizează modul în care este folosită clasa și generează un șablon corespunzător pentru fiecare mod în care este folosită. Toate aceste operații se întâmplă la **compile** time, nu la run time. Instantierile de mai sus determină compilatorul să genereze cod pentru ambele clase (înlocuind o dată T cu int, altă dată cu double).

Mapări cheie-valoare

Un alt exemplu de templates aplicat claselor - o clasă numită KeyStorage care are:

- o cheie (de tip int);
- un membru de date generic (al cărui tip de date nu îl știm la momentul scrierii clasei).

Vrem să putem folosi codul clasei indiferent de tipul de date al membrului.

Iată cum putem face acest lucru:

KeyStorage.h

```
template<typename T>
class KeyStorage {
public:
    int key;
    T member;
};
```

În funcția main, să presupunem că vrem să folosim clasa cu membrul de tip long.

main.cpp

```
#include "KeyStorage.h"

int main() {
    KeyStorage<long> keyElement;
    return 0;
}
```

Guideline-uri implementare

Pentru că totul se întâmplă la compile time, înseamnă că în momentul în care compilatorul întâlnește secvența de cod ce folosește template-uri trebuie să știe *toate* modurile în care aceasta este folosita.

Asta înseamnă că:

- Trebuie să scrieți întreaga implementare în header! *sau*
- Scrieți descrierea clasei generice în header, în fișierul de implementare fiecare metodă declarată este de fapt o funcție cu template și la sfârșitul implementării adăugat *template class numeclasa<numetip>*;

Ultimul rând de fapt forțează folosirea template-ului cu un anumit tip de date și deci compilatorul generează cod corespunzător (trebuie să scrieți asta pentru toate tipurile).

Clasa KeyStorage

Iată mai jos o structură mai dezvoltată pentru clasa KeyStorage, în care cheia este setată în constructor.

KeyStorage.h

```
template<typename T>
class KeyStorage {
public:
    KeyStorage(int k);
    ~KeyStorage();

    T getMember();
    T setMember(T element);

private:
    T member;
    int key;
};
```

Implementarea completă a ei poate fi realizată:

- în header (în cazul template-urilor, acest mod este cel mai indicat).
- în fișierul de implementare .cc / .cpp (al cărui schelet parțial îl găsiți mai jos).

KeyStorage.cpp

```
#include "KeyStorage.h"

template<typename T>
KeyStorage<T>::KeyStorage(int k) {
    // TODO
}

template<typename T>
KeyStorage<T>::~KeyStorage() {
}

// TODO: restul metodelor.

// La sfarsit, se mentioneaza tipurile de date
// pentru care urmeaza sa fie instantiata clasa.
template class KeyStorage<int>;
template class KeyStorage<long>;
```

ArrayList și LinkedList

ArrayList și LinkedList reprezintă două structuri de date, de obicei abstracte, ce formalizează conceptul de colecție ordonată de entități. În mod minimal, acestea sunt caracterizate prin:

- Operații:
  - `Add` - adaugă un element în cadrul containerului. Adăugarea se poate face la început, la sfârșit sau pe o poziție arbitrară
  - `Remove` - șterge un element din container. Identificarea se poate face pe baza poziției din container (iterator) sau pe baza valorii elementului ce se dorește a fi șters
  - `Get` - consultă un element din listă. Identificarea se face pe baza poziției elementului din container
  - `Update` - actualizează informația unui element din container. Identificarea se face pe baza poziției elementului din container
- Proprietăți:
  - `Lungimea` - numărul de elemente din listă
  - `Tipul` - felul elementelor din listă. Această proprietate este întâlnită mai ales la implementările în limbaje care suportă tipuri generice (C++, Java, etc.)

ArrayList

În cazul array-urilor dinamice, elementele sunt stocate într-un vector 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). De asemenea, în cazul în care, în urma ștergerilor, arraylist-ul are un numar de poziții ocupate mai mic decât capacitatea sa, dat de un factor specificat, se poate opta pentru redimensionarea array-ului care conține elementele, la o dimensiune mai mică. Obținem astfel mai multă memorie liberă, însă, trebuie sa plătim prețul overhead-ului dat de realocarea array-ului și în acest caz.

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.

Descriere metode

- **int size()**: întoarce numărul curent de elemente stocate în ArrayList. Complexitate: O(1).
- **void addLast(E element)**: adaugă la sfârșitul listei elementul `element`. Se va

redimensiona în prealabil dacă se constată că este necesar. Complexitate: O(1).

- **void addFirst(E element)**: adaugă la începutul listei elementul `element`. Se va

redimensiona în prealabil dacă se constată ca este necesar. Adăugarea la început presupune deplasarea tuturor elementelor deja existente cu o poziție la dreapta → se realizează un număr de operații proporțional cu dimensiunea listei: O(size).

- **void removeLast()**: șterge elementul de la sfârșitul listei. Se va redimensiona

ulterior dacă se constată că este necesar. Complexitate: O(1).

- **void removeFirst()**: șterge elementul de la incepului listei. Ștergerea de la început

presupune deplasarea tuturor elementelor următoare cu o poziție la stânga → se realizează un număr de operații proporțional cu dimensiunea listei. Complexitate: O(size).

- **bool isEmpty()**: returnează `true` în cazul în care lista nu conține niciun element,

`false` în caz contrar. Complexitate: O(1).

- **void addOnPos(E element, int pos)**: adaugă elementul `element` pe poziția `pos` în lista.

Se va redimensiona în prealabil dacă se constată că este necesar. Presupune deplasarea elementelor de la `pos + 1` până la `size` cu o poziție la dreapta. Complexitate: O(size)-în cel mai defavorabil caz (`addOnPos(e, 0)`, se vor deplasa la dreapta toate elementele listei). Observație: trebuie verificat dacă `pos` indică o poziție cuprinsă între 0 și `size-1`.

- **void removeFromPos(int pos)**: șterge elementul de pe poziția `pos` din lista.

Se va redimensiona ulterior dacă se constată ca este necesar. Presupune deplasarea elementelor de la `pos + 1` până la `size` cu o poziție la stânga. Complexitate: O(size) (similar ca la `addOnPos()`). Observație: trebuie verificat dacă `pos` indică o poziție cuprinsă între 0 și `size-1`.

LinkedList

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.

Această implementare are avantajul unei mai bune folosiri a memoriei, putând fi ocupată toată memoria liberă disponibilă, indiferent de dispunerea ei. Dezavantajul constă în timpul de acces la elementele containerului.

Descrierea metodelor (pentru listă simplu înlănţuită)

- **int size()**: întoarce numărul curent de elemente stocate în LinkedList. Complexitate: O(1)
- **void addLast(E element)**: adaugă la sfârșitul listei elementul `element`. Adăugarea

presupune modificarea câmpului `next` al nodului `tail` astfel încât să indice spre noul nod ce va fi adăugat. La final, noul nod adăugat va deveni `tail`. Complexitate: O(1)

- **void addFirst(E element)**: adaugă la începutul listei elementul `element`. Adăugarea

presupune modificarea câmpului `next` al noului nod astfel încât să indice spre `head`-ul curent al listei. La final, noul nod adăugat va deveni `head`. Complexitate: O(1)

- **E removeLast()**: șterge și întoarce ultimul element al listei. Operația presupune

parcurgerea listei nod cu nod folosind un iterator, cât timp iteratorul mai are un nod care să îl succeadă. Se va păstra și un pointer la nodul care precede iteratorul. Când se ajunge cu iteratorul pe ultimul nod, se va tăia legătură de la penultimul nod la ultimul, iar `tail`-ul listei va deveni penultimul nod. Complexitate: O(size), unde size este dimensiunea listei. Complexitate mai bună se obține în cazul unei liste dublu înlănțuite, unde se poate cunoaște de la început nodul care precede `tail`-ul listei.

- **E removeFirst()**: șterge și întoarce primul element al listei. Operația presupune

modificarea `head`-ului listei astfel încât acesta să indice spre nodul imediat următor lui. Dacă nu există un nod următor, `head` va deveni NULL iar lista va fi goală. Complexitate: O(1)

- **bool isEmpty()**: returnează `true` în cazul în care lista nu conține niciun element,

`false` în caz contrar.

- **void addOnPos(E element, int pos)**: se inițializează un contor cu valoarea 0 și

se parcurge lista cu un iterator de la nodul `head`, incrementand cu 1 la fiecare pas valoarea contorului. Când `contor == pos-1` se modifică pointer-ul `next` al iteratorului astfel încât să indice spre noul nod. De asemenea, pointer-ul `next` al noului nod va fi modificat astfel încât să indice spre nodul de pe poziția `pos`. Complexitate: O(size).

- **void removeFromPos(int pos)**: la fel ca în cazul adăugării, se itereaza prin lista până

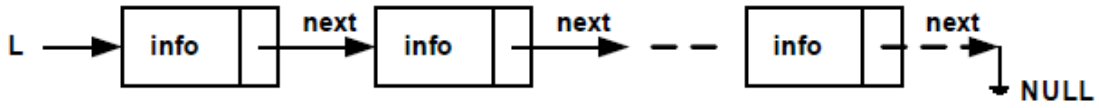
când se găsește nodul care trebuie șters. Apoi, pointer-ul `next` al nodului precedent va indica spre nodul care succede iteratorul. Complexitate: O(size)

Pentru metodele de `remove` vor trebui, în funcție de caz, efectuate verificări speciale pentru cazul unei liste vide, o lista cu un singur element etc.

Tipuri de LinkedList

Listă liniară simplu înlănţuită

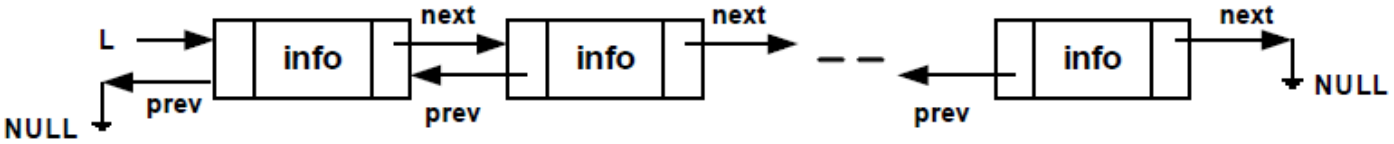
Are o singură legatură la fiecare nod. Această legatură indică întotdeauna următorul nod din listă, sau o valoare nulă (dacă suntem la finalul listei), sau o listă liberă (pentru identificarea ei).



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.

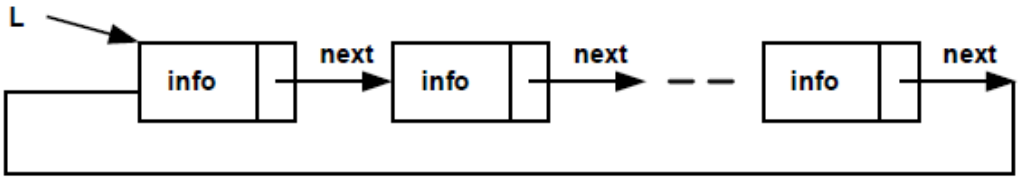


Listă circulară simplu-înlănţuită

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.



Schelet

Schelet

Exerciții

Fiecare laborator va avea unul sau doua exerciții publice si un pool de subiecte ascunse, din care asistentul poate alege cum se formeaza celelalte puncte ale laboratorului.

1) [**5p**] Implementați în header-ul definit funcțiile pentru o listă liniară dublu înlănțuită.

2) [**5p**] Implementați în header-ul definit funcțiile pentru un vector alocat dinamic cu redimensionare.

3) [**2.5p**] Sortați o listă simplu înlănțuită doar prin modificarea de pointeri (fără a copia informația dintr-un nod în altul).

4) [**2.5p**] Implementați o funcție `LinkedListNode<T> *reverse(LinkedListNode<T> *head)` care primește ca parametru un pointer la începutul unei liste simplu înlănțuite și inversează ordinea nodurilor din lista (fără alocare de memorie auxiliară pentru o nouă listă).

Exemplu: pentru lista `1 → 2 → 3`, se întoarce lista `3 → 2 → 1`.

5) [**2p**] Implementați o fereastră glisantă de lungime fixă. Dându-se un container de elemente (de exemplu numere întregi), fereastra de dimensiune fixă (de exemplu 3) va conține inițial primele 3 elemente din container. La un apel al metodei `advance()`, din fereastră se va scoate primul element și se va adaugă următorul element din container.

Exemplu: pentru `container == [1 2 3 4 5]` și `dimensiuneFereastra == 3`, inițial fereastra va conține `[1 2 3]`.

După un apel `advance()`, fereastra conține `[2 3 4]`. Când fereastra ajunge la finalul container-ului, se vor adauga elementele de la începutul acestuia. Conform exemplului de mai sus, după 3 apeluri `advance()` (pornind din starea inițială), fereastra va conține `[4 5 1]`.

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. Care este sintaxa pentru declararea unei funcții template?
2. Ce este o clasă template?
3. Implementați o funție de ștergere a duplicatelor dintr-o listă simplu înlănțuită sortată.
4. Implementați o funcție de detectare (+ înlăturare) a unui ciclu într-o listă simplu înlănțuită.
5. Găsirea celui de-al k-lea nod de la sfârșit spre început într-o listă simplu înlănțuită.

Bibliografie

1. C++ Reference [http://www.cplusplus.com]
2. Linked list visualization [https://visualgo.net/en/list]
3. Vector C++ STL Implementation [http://www.cplusplus.com/reference/vector/vector/]

4. Doubly linked list C++ STL Implementation [http://www.cplusplus.com/reference/list/list/]
5. CLRS - *Introduction to Algorithms, 3rd edition*, capitol 10.2 - Linked lists

sd-ca/2018/laboratoare/lab-03.txt · Last modified: 2019/02/01 13:23 by teodora.serbanescu