Bibliotecă de tipuri abstracte

Proiect de semestru - Cerc C

Documentatie

Nagy Lilla
22.01.2017.
AC - Calculatorare, Anul 2, 30421

1. Introducere

Structurile de date abstracte sunt structuri agregate care permit operarea uşoară cu datele şi execuţie eficientă. În acest proiect, cerinţa a fost să implementăm mai multe tipuri de date abstracte şi să le furnizăm sub forma unei biblioteci. Am avut opţiunea să le facem ori generice, ori nu - eu am ales să fie generice. Pentru fiecare tip abstract am furnizat şi un tester, care citeşte din fişierele aflate într-un folder şi le testează.

2. Obiectivele proiectului

Tipurile abstracte implementate în acest proiect au toate funcţionalităţile de bază de care ar avea nevoie, ca mai târziu să fie folosite la rezolvarea altor probleme.

Tipurile de bază şi funcţionalităţile lor sunt:

- Vector: create, print, add, put, get, delete, search, sort, merge şi ştergere completă
- LinkedList: create, print, add, put, get, delete, search, sort, merge şi ştergere completă
- HashTable: create, print, add, delete, search, rehash şi ştergere completă
- MinHeap: create, print, add, get (minimul), delete (minimul), delete (orice), merge şi ştergere completă
- BinarySearchTree: create, print, preorder, inorder, postorder, add, search, delete, merge, height şi ştergere completă
- BalancedBST: create, print, preorder, inorder, postorder, add, search, delete, merge, height şi ştergere completă

3. Proiectare şi implementare

1. Vector

Pentru implementarea vectorului am folosit un array de structuri, în care dacă nu adaug la final, trebuie să mut toate elementele după indexul specificat, în dreapta, iar dacă şterg, în stânga. Căutarea este cea mai eficientă, O(1), pentru că se poate returna structura respectivă doar folosind indexul.

Toate testerele sunt implementate la fel, deci voi vorbi despre asta doar aici. Testerul citeşte dintr-un directory, toate fişierele (sau doar cele specificate, în funcţie de utilizator) şi le citeşte pe rând. Caută instrucţiunea specificată, şi după aceea, numele elementului specificat (la mine, un char), care determină indexul structurii de dată in array-ul de structuri de date iniţializat la începutul testerului. În acest mod, vom şti tot timpul, pe ce să facem operaţiile. Sunt şi mesaje de eroare generate, iar totul se scrie într-un fişier .out.

2. LinkedList

Pentru implementarea listei înlănţuite am legat nodurile (singly linked list), iar am furnizat o structură care are un pointer către primul şi ultimul element. Adăugarea şi ştergerea e simplă, O(1), dar ca să returnăm un nod, trebuie să parcurgem toată lista.

3. HashTable

Am implementat hashtable-ul folosind un array (nu vectorul făcut de mine), şi LinkedList.h, care m-a ajutat ca fiecărui element din array să-i adaug un pointer către un linked list. În cazul în care două sau mai multe elemente au acelaşi index în table, le leg folosind o listă de acesta. Valorile nu sunt în nici o ordine specifică, iar returnarea unui element nu e simplă, deoarece trebuie să parcurgem prima dată arrayul, iar după asta şi lista, deci ideal ar fi ca tabelul să nu fie prea încărcat, şi în acest fel să nu fie prea multe elemente legate de un element din array-ul nostru.

4. MinHeap

MinHeap este un vector, dar ceea ce diferă e că trebuie să fie aranjat ca structura unui arbore - deci fii elementului n să fie la indexul 2n+1 şi 2n+2, iar părintele nodului n să fie la (n-1)/2. Pentru asta erau nevoie de operaţii adiţionale, dar returnarea minimului are complexitatea O(1), în ciuda faptului că structura ei trebuie corectată la fiecare adăugare/ştergere.

5. BinarySearchTree

În binary search tree, nodurile mai mici ca nodul anterior se adaug la stânga ei, iar cele mai mari, la dreapta. În acest fel, căutarea devine foarte simplă: O(log n), la fel ca adăugarea şi ştergerea. Majoritatea funcţiilor implementate sunt recurzive, simplificând codul, dar îngreunând situaţia la prea multe elemente.

6. BalancedBST

BalancedBST e un arbore binar echilibrat. Singura diferenţă între acesta şi structura anterioară e că trebuie adăugată algoritmul de AVL tree, diferenţa de nivele trebuie păstrată la

maxim 1. În cazul în care condiţia nu e îndeplinită, trebuie să rotim nodurile, deci după fiecare adăugare sau ştergere e nevoie de operaţii adiţionale.

4. Manual de utilizare

Pentru folosirea testerului, codul trebuie rulat, iar outputurile se generează automat, dacă fileurile .in sunt puse în directorul potrivit: Input. Se poate preciza în linia de comandă fişierul input (001.in), range-ul de fişiere input (001.in 003.in) sau runall, care înseamnă că toate testerele sunt rulate din fişierul specificat.

5. Concluzii

Toate cerinţele au fost îndeplinite, în afară de testerul pentru LinkedList, care nu vrea să ruleze într-un mod miraculos, în ciuda faptului că testerul e acelaşi pentru fiecare structură de date.

Pentru mine, MinHeap era cel mai greu de implementat, pentru îmi era greu să leg elementele array-ului de listele înlănţuite, iar cel mai mult mi-au plăcut arborele, pentru că codul e foarte simplu, doar că trebuie să stai puţin să înţelegi recurziile.

Dar cel mai greu era să scriu această documentaţie, pentru că mai am un minut până la dealine, şi nu am timp să-l recitesc :)

Dar promit că dacă va fi nevoie, o să rescriu documentaţia în condiţiile în care n-o să am examen în ziua următoare :D