Laborator 9 - ABC și Heap

Responsabili

- Cristian Crețeanu [mailto:mailto:cristiancreteanu06@gmail.com]
- Cosmin Petrișor [mailto:mailto:cosmin.ioan.petrisor@gmail.com]

Obiective

În urma parcurgerii articolului, studentul va fi capabil să:

- înţeleagă structura şi proprietățile unui arbore binar de căutare;
- construiască, în limbajul C++, un arbore binar de căutare;
- realizeze o parcurgere a structurii de date prin mai multe moduri;
- realizeze diferite operaţii folosind arborii binari de căutare;
- definească proprietăţile structurii de heap;
- implementeze operaţii de inserare, ştergere şi căutare care să păstreze proprietatea de heap;
- folosească heap-ul pentru a implementa o metodă de sortare eficientă.

Noțiuni teoretice - ABC

Un arbore binar de căutare este un arbore binar care are în plus următoarele proprietăți:

- cheile stocate în noduri (informația utilă) aparțin unei mulțimi peste care există o relație de ordine **totală**
- cheia dintr-un nod oarecare este **mai mare** decât cheile tuturor nodurilor din subarborele stâng şi este **mai mică** decât cheile tuturor nodurilor ce compun subarborele drept

Arborii binari de căutare permit menţinerea datelor în ordine şi o căutare rapidă a unei chei, ceea ce îi recomandă pentru implementarea de mulţimi şi dicţionare ordonate.

O importantă caracteristică a arborilor de căutare, este aceea că parcurgerea `inordine` produce o secvenţă ordonată crescător a cheilor din nodurile arborelui.

Valoarea maximă

Valoarea maximă dintr-un arbore binar de căutare se află în **nodul din extremitatea dreaptă** şi se determină prin coborârea pe subarborele drept, iar **valoarea minimă** se află în **nodul din extremitatea stângă**, determinarea fiind simetrică.

Căutarea

Căutarea unei chei într-un arbore binar de căutare este asemănătoare căutării binare: cheia căutată este comparată cu cheia din nodul curent (iniţial nodul rădăcină). În funcţie de rezultatul comparaţiei apar trei cazuri:

- acestea coincid –> elementul a fost găsit
- elementul căutat este mai mic decât cheia din nodul curent –> căutarea continuă în subarborele stâng

- elementul căutat este mai mare decât cheia din nodul curent → căutarea continuă în subarborele drept

Pseudocod:

```
bool cautare(nod, cheie) {
  if nod == NULL
    return false;
  if nod.cheie == cheie
    return true;

  if cheie < nod.cheie
    return cautare(nod.stanga, cheie);
  else
    return cautare(nod.dreapta, cheie);
}
```

Inserarea

Inserarea unui nod se face, în funcţie de rezultatul comparaţiei cheilor, în subarborele stâng sau drept. Dacă arborele este vid, se creează un nod care devine nodul rădăcină al arborelui. În caz contrar, cheia se inserează ca fiu stâng sau fiu drept al unui nod din arbore.

Ștergerea

Ștergerea unui nod este o operaţie puţin mai complicată, întrucât presupune o rearanjare a nodurilor. Pentru eliminarea unui nod dintr-un arbore binar de căutare sunt posibile următoare cazuri:

- nodul de şters nu există → operaţia se consideră încheiată
- nodul de şters nu are succesori → este o frunză
- nodul de şters are un singur successor
- nodul de şters are doi succesori

În cazul ştergerii unui nod frunză sau a unui nod având un singur successor, legătura de la părintele nodului de şters este înlocuită prin legătura nodului de şters la succesorul său (**NULL** în cazul frunzelor).

Eliminarea unui nod cu doi succesori se face prin înlocuirea sa cu nodul care are cea mai apropiată valoare de nodul şters. Acesta poate fi nodul din extremitatea dreaptă a subarborelui stâng (**predecesorul**) sau nodul din extremitatea stânga a subarborelui drept (**succesorul**). Acest nod are cel mult un successor.

Complexitatea operaţiilor (căutare, inserare, ștergere) într-un arbore binar de căutare este - *pe cazul mediu* - **O(logn)**.

Notiuni teoretice - Heap

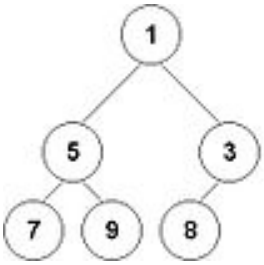
Mai sus considerat arborii binari ca fiind o înlănţuire de structuri, legate între ele prin pointeri la descendenţii stâng, respectiv drept. Această reprezentare are avantajul flexibilităţii şi a posibilităţii de a creşte sau micşora dimensiunea arborelui oricât de mult, cu un efort minim. Cu toate acestea, metoda precedentă nu poate fi folosită atunci când este nevoie de o reprezentare compactă a arborelui în memorie (de exemplu pentru stocarea într-un fişier), pentru că acei pointeri nu sunt valizi decât în cadrul programului curent.

Din acest motiv, există câteva moduri de a stoca arborii într-o structura liniară de date (vectori), dintre care:

- Înlocuirea pointer-ilor din structurile asociate nodurilor cu întregi ce reprezintă indici într-un vector de astfel de structuri. Primul element din vector va fi rădăcina arborelui, şi va exista un contor curent (la nivelul întregului vector) care indică următoarea poziţie liberă. Atunci când un nod trebuie

adăugat în arbore, i se va asocia valoarea curentă a contorului, iar acesta va fi incrementat. În nodul părinte se va reţine indicele **în vector** al noului nod, în locul adresei lui în memorie (practic acesta este un mic mecanism de alocare de memorie, pe care îl gestionăm noi).
- Eliminarea totală a informaţiei legate de predecesori, şi folosirea unei formule de calcul a părintelui si a descendenţilor unui nod pe baza indicelui acestuia în vector.

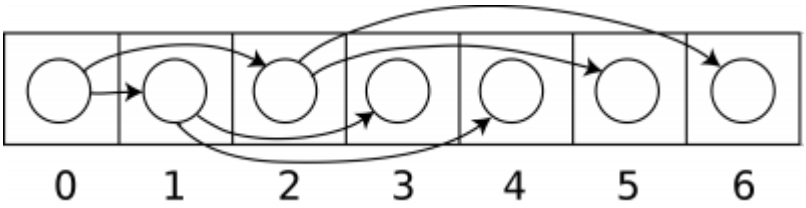
Pentru un arbore binar, cea de-a doua modalitate se implementează conform figurii de mai jos:



Node	1	5	3	7	9	8
Index	0	1	2	3	4	5

Se consideră că arborele este aşezat în vector în ordine (începând de la 0) de la primul nivel până la ultimul, iar nodurile fiecărui nivel se aşează de la stânga la dreapta.

Reprezentarea liniara (sub formă de vector) pentru un arbore binar complet devine:



Se constată că poziţia nodului rădăcină în vector este 0, iar pentru fiecare nod în parte, părintele şi descendenţii se pot calcula după formulele:

- Parinte(i) = (i - 1) / 2, unde i este indicele nodului curent
- IndexStanga(i) = 2 * i + 1, unde i este indicele nodului curent
- IndexDreapta(i) = 2 * i + 2, unde i este indicele nodului curent

Proprietăţi ale structurii de heap binar. Operaţii elementare.

În cele ce urmează vom considera un heap ca fiind de fapt un min-heap. Noţiunile sunt perfect similare şi pentru max-heap-uri.

Un min-heap binar este un arbore binar în care fiecare nod are proprietatea că valoarea sa este mai mare sau egală cu cea a părintelui său.

Într-o enunțare echivalentă:

Un min-heap binar este un arbore binar în care fiecare nod are proprietatea că valoarea sa este mai mică sau egală decât cea a tuturor descendenților săi.

```
h[parinte(x)] <= h[x]
```

h[x] reprezintă valoarea nodului **x**, din vectorul h asociat arborelui.

În mod similar, un max-heap are semnul inegalităţii inversat. Astfel, putem defini şi recursiv proprietatea de heap pentru orice (sub)arbore:

- nodul rădăcină trebuie să respecte proprietatea de heap (inegalitatea);
- cei doi subarbori descendenţi sa fie heap-uri.

Pentru a implementa operaţiile de inserare, ştergere, etc. pentru un heap, vom avea nevoie mai întâi de două operaţii elementare:

- `pushDown`, care presupune că heap-ul a fost modificat într-un singur nod şi noua valoare este mai mare decât cel puţin unul dintre descendenţi, şi astfel ea trebuie "cernută" către nivelurile de jos, până când heap-ul devine din nou valid.
- `pushUp`, care presupune că valoarea modificată (sau adăugată la sfârşitul vectorului, în acest caz) este mai mică decât părintele, şi astfel se propagă acea valoare spre rădăcina arborelui, până cand heap-ul devine valid.

Operaţii uzuale asupra heap-ului

Având implementate cele două operaţii de bază, putem defini operaţiile uzuale de manipulare a heap-urilor:

Peek

Operația întoarce valoarea minimă din min-heap. Valoarea se va afla la indexul 0 al vectorului de implementare a heap-ului.

Push (insert)

Adaugă o nouă valoare la heap, crescându-i astfel dimensiunea cu 1.

Algoritmul pentru această funcție este următorul:

1. introducem elementul de inserat pe prima poziție liberă din vectorul de implementare a heap-ului (în principiu `dimVect`);
2. "împingem" elementul adăugat în vector până la poziția în care se respectă proprietatea de heap; veți folosi funcția `pushUp`.

```
push(X)
{
    heap[dimVec] = X;
    dimVec++;
    pushUp(dimVec - 1);
}
```

Pop (extractMin)

Funcția aceasta scoate valoarea minimă din heap (și reactualizează heap-ul). Poate întoarce valoarea scoasă din heap.

Pentru a face operația de pop veți urma pașii:

1. elementul minim din heap (de pe prima poziție) va fi interschimbat cu elementul de pe ultima poziție a vectorului;
2. dimensiunea vectorului va fi redusă cu 1 (pentru a ignora ultimul element, acum cel pe care doream să-l înlăturăm)
3. vom "împinge" nodul care se afla acum în rădăcina heap-ului către poziția în care trebuie sa fie pentru a fi respectată proprietatea de heap; acest lucru se va face cu funcția `pushDown`.

```
extractMin()
{
    interschimba(heap[0], heap[dimVec - 1]);
    dimVect--;
    pushDown(0);
}
```

Algoritmul Heap Sort

Întrucât operaţiile de extragere a minimului şi de adăugare/reconstituire sunt efectuate foarte eficient (complexităţi de O(1), respectiv O(log n)), heap-ul poate fi folosit într-o multitudine de aplicaţii care necesită rapiditatea unor astfel de operaţii. O aplicaţie importantă o reprezintă sortarea, care poate fi implementată foarte eficient folosind heap-uri. Complexitatea acesteia este O(n*log n), aceeaşi cu cea de la quick sort şi merge sort.

Se poate implementa inserand, pe rând, în heap, toate elementele din vectorul nesortat. Apoi într-un alt şir se extrag minimele. Noul şir va conţine vechiul vector sortat.

```
HeapSort()
{
    ConstruiesteMaxHeap();
    for (i = dimHeap - 1; i >= 1; i--)
    {
        // Punem maximul la sfarsitul vectorului
        interschimba(heap[0], heap[i]);
        // 'Desprindem' maximul de heap (valoarea ramanand astfel in pozitia finala)
        dimHeap--;
        // Reconstituim heap-ul ramas
        pushDown(0);
    }
}
```

ABC vs Heap

Deși la prima vedere nu există mari diferențe între cele două structuri de date, ele sunt complet diferite. Se poate observa că ele diferă atât la nivelul implementării (bst:pointeri către fii vs heap:vector), cat și al complexităților operațiilor specifice. Totuși, deși ambele se pot folosi în rare cazuri pentru același scop (fără a fi la fel de eficiente), ele au întrebuințări diferite.

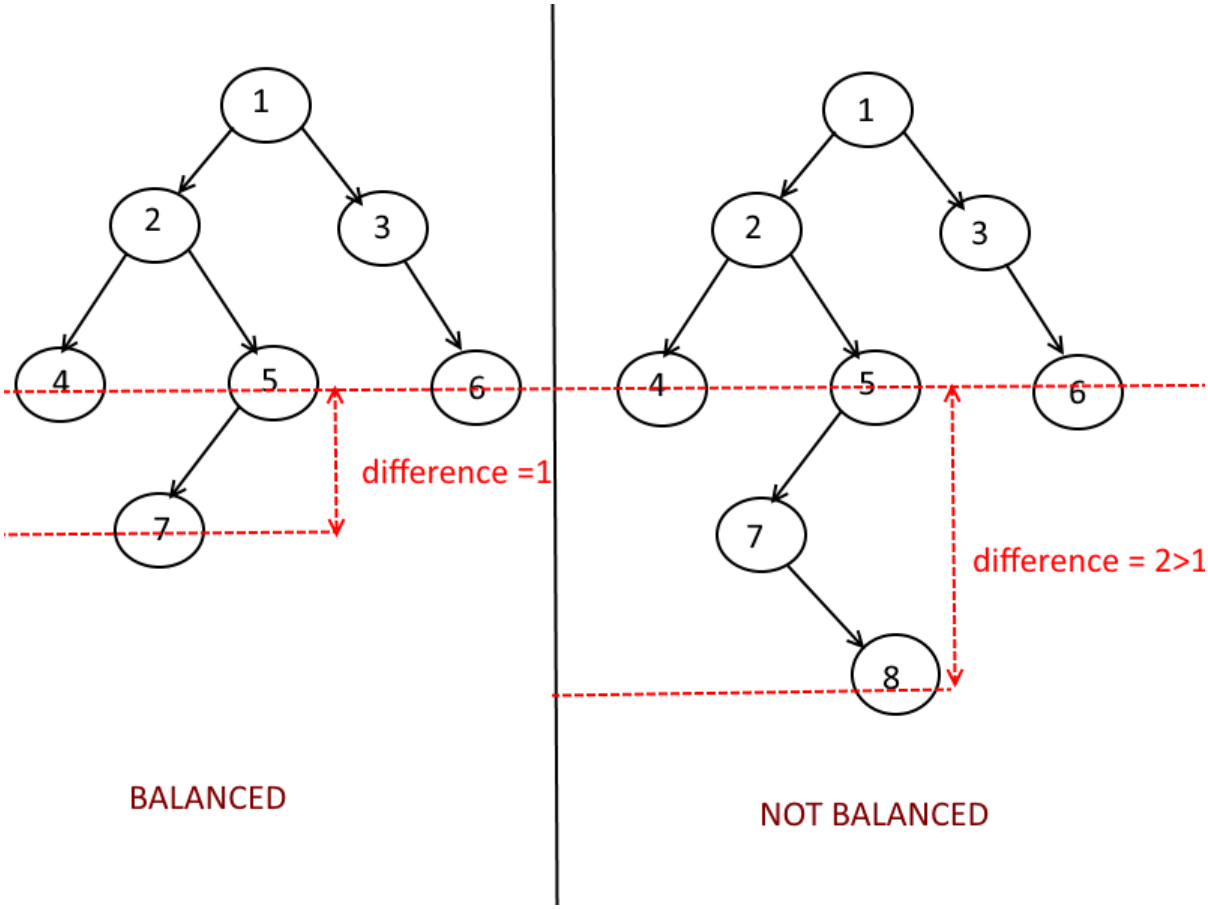
ABC

- Se folosește pentru a implementa arbori echilibrați, precum AVL, Red-Black
- Prezintă toate avantajele unui vector sortat, venind în plus cu inserare în timp constant.
- Nu este mereu echilibrat.

HEAP

- Heap-ul stă la baza implementării cozii de priorități și a algoritmului heapsort
- Se poate folosi pentru găsirea eficientă a celui de-al k-lea cel mai mic/mare (minheap/maxheap) element.
- Este mereu un arbore echilibrat (complet)

Un arbore este **echilibrat** dacă fiecare subarbore este echilibrat și înălțimea orcăror doi subarbori diferă cu cel mult 1. Această regulă stă la baza implementării unui arbore AVL. Arborii roșu-negru impun alte reguli.



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.

ABC

Observații privind scheletul de cod pentru arbore binar de cautare:

- Scheletul citește N numere dintr-un fișier dat ca parametru în linia de comandă.
- Aceste N numere sunt introduse într-un arbore binar de căutare, funcționalitate pe care voi trebuie sa o implementaţi.
- Clasa `BinarySearchTree` conține un membru de tip pointer către T pentru a reţine informaţia utilă şi de asemenea pentru a putea determina mai simplu cazul contrar. Astfel putem verifica dacă un BinarySearchTree conține minim un element (vezi funcția `isEmpty`).
- Funcția `removeKey` întoarce adresa noului nod rădăcină, dacă s-a șters vechea rădăcină.

1) [**4p**] (BinarySearchTree.h) Implementați următoarele funcționalități de bază ale unui arbore binar de căutare:

- [**1p**] constructor(**TODO 1.1**), destructor(**TODO 1.2**)
- [**1p**] adăugare elemente în arbore.(**TODO 1.3**)
- [**1p**] căutare elemente în arbore. (**TODO 1.4**)

- [**1p**] parcurgere inordine arbore. (**TODO 1.5**)

2) [**0.5p**] funcții pentru returnare valoare minimă/maximă din arbore. Implementaţi eficient, ţinând cont de faptul ca arborele binar este unul de căutare. (**TODO 2**)

3) [**1.5p**] functie pentru aflarea numarului de nivele din ABC. funcție pentru afișarea cheilor (informației utile) din nodurile situate pe un anumit nivel primit ca parametru. Nivel = distanța de la rădăcină la un nod, nivelul rădăcinii fiind 0. (**TODO 3**)

4) [**2p**] (BinarySearchTree.h) Implementați funcția de ștergere a unui element (trebuie să tratați și cazul în care se va șterge elementul din rădăcină). (**TODO 4**)

5) [**2p**] (BinaryTree.h) Implementați o funcție care verifică dacă un arbore este un ABC. (**TODO 5**)

Heap

6) [**4p**] **heap.h** Definiţi o structură de vector pe care să poată fi folosite operaţiile de baza ale unui heap, şi funcţii de construcţie şi eliberare a structurii:

- Constructor pentru inițializarea unui heap. `capacity` reprezintă numărul maxim de elemente din vector. Codul va trebui să aloce memorie separată şi apoi să lucreze cu acea memorie.
- Funcție pentru eliberarea memoriei alocate pentru values.

Implementaţi operaţiile elementare de lucru cu heap-uri, prezentate în secţiunile anterioare:

- Implementati functiile de calcul ai parintelui si ai descendentilor(functiilor vor întoarce -1 în cazul în care părintele, respectiv descendenţii nu există).
- Implementati pushUp si pushDown.

Implementaţi operaţiile uzuale de lucru cu heap-uri.

Bibliografie

1. Vizualizare Binary Search Tree [http://www.cs.usfca.edu/~galles/visualization/BST.html]
2. Binary Search Tree [https://en.wikipedia.org/wiki/Binary_search_tree]
3. Binary Heap [http://en.wikipedia.org/wiki/Binary_heap]
4. Heap Sort [http://en.wikipedia.org/wiki/Heap_sort]

sd-ca/2018/laboratoare/lab-09.txt · Last modified: 2019/02/01 13:26 by teodora.serbanescu