Laborator 11 - Arbori binari de căutare echilibrați. Treapuri.

Responsabil:

- Andrei Pârvu (2013, 2014)

Obiective

În urma parcurgerii acestui laborator studentul va:

- înțelege conceptul unui arbore echilibrat de căutare
- exemplifica acest concept pe structura de treap
- implementa operațiile de adăugare nod, ștergere nod și rotiri
- face operații mai complexe si parcurgri de treapuri

Necesitatea structurii de arbore binar de căutare echilibrat

O structură de date este o metodă de a reține anumite date astfel încât operațiile cu acestea (căutare, inserare, ștergere) să fie făcute cât mai eficient și să respecte cerințele programatorului. De multe ori, o anumită structură de date se află la baza unui algoritm sau sistem, iar o performanță bună a acesteia (complexitate spațială și temporală cât mai mică) influențează performanța întregului sistem.

În laboratoarele precedente am observat că un arbore binar de căutare de înalțime `h` implementează operațiile descrise mai sus într-o complexitate de `O(h)`. Dacă acest arbore binar nu este capabil de a gestiona elementele ce sunt inserate pentru a își menține o structura echilibrată atunci complexitatea pe operațiile de baza va crește. Exemplu: să presupunem ca avem de introdus `n` numere intr-un arbore binar de căutare; întamplarea face ca numerele să fie sortate, de unde rezultă că arborele format va fi liniar (fiecare nod va avea maxim doi vecini); astfel, complexitatea pe operatiile de baza va fi `O(n)` la fel ca în cazul folosirii unui simplu vector.

Noțiuni de bază despre treapuri

Treapurile sunt unii din arborii de căutare echilibrați cel mai des folosiți datorită implementării relativ ușoare (comparativ cu alte structuri similare cum ar fi Red-Black Trees, AVL-uri sau B-Trees), dar și a modului de operare destul de intuitiv. Fiecare nod din treap va retine două câmpuri:

- cheia - informația care se reține în arbore și pe care se fac operațiile de inseare, căutare și ștergere
- prioritatea - un număr pe baza căruia se face echilibrarea arborelui

Această structură trebuie să respecte doi invarianți:

- Invariantul de arbore de cautare (search **tr**ee) - **cheia** unui nod va fi mai mare sau egală decât **cheia** fiului stânga (dacă există) si mai mică sau egală decât **cheia** fiului dreapta (dacă există); cu alte cuvinte o parcurgere in inordine a arborelui va genera șirul sortat de chei.
- Invariantul de h**eap** - **prioritatea** unui nod este mai mare sau egală decât **prioritățile** fiilor.

Astfel, se poate observa că numele structurii de date a venit din acești doi invarianți: tr-eap.

Cum se menține echilibrul structurii? De fiecare dată când un nod este inserat în arbore prioritatea lui este generată random (o metodă similară cu cea de la randomized quick sort, în care la fiecare pas pivotul este generat aleator) - astfel arborele va fi aranjat într-un mod aleator (bineînțeles, respectând cei doi invarianți)cum numărul arborilor echilibrați este mai mare decât cel al arborilor rău echilibrați, șansa este destul de mică ca prioritățile generate aleator să nu mențină arborele echilibrat. Demonstratia complet teoretică

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

asupra faptului că operațiile de baza au complexitatea O(logN) se poate găsi in 2.

Structura unui nod

Mai jos avem codul pentru structura nodului unui treap; se pot observa asemănările cu structura de arbore binar și cu cea de heap.

```
template <typename T> struct Treap {
  T key;
  int priority;
  Treap<T> *left, *right;
};
```

Bineînțeles, tipul de date trebuie să permită o relație de ordine totală astfel încât oricare două elemente să poată fi comparate.

Operații de bază

Mai jos este descris pseudocodul pentru operațiile de bază făcute cu treapuri. Pentru exemplificarea operațiilor am folosit un nod special, numit `nil`, care reprezintă un nod fictiv, ce nu reține date, folosit pentru a arăta că nu există un nod efectiv în treap. De exemplu, dacă un nod `x` are ambii fii egali cu `nil` înseamnă ca `x` este frunză în arbore.

Căutarea

Căutarea se face exact ca la un arbore binar de căutare.

```
bool cautare(nod, cheie) {
  if nod == nil
    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 generand o prioritate aleatoare pentru acesta și procedând asemănător ca pentru un arbore de căutare, adăugând nodul la baza arborelui printr-o procedură recursivă, pornind de la rădăcina acestuia.

Deși inserarea menține invariantul arborelui de căutare, invariantul de heap poate să nu se mai respecte. De aceea, trebuie definite operații de rotire (stânga sau dreapta), care să fie aplicate unui nod în cazul în care prioritatea sa este mai mare decât ce a părintelui său.

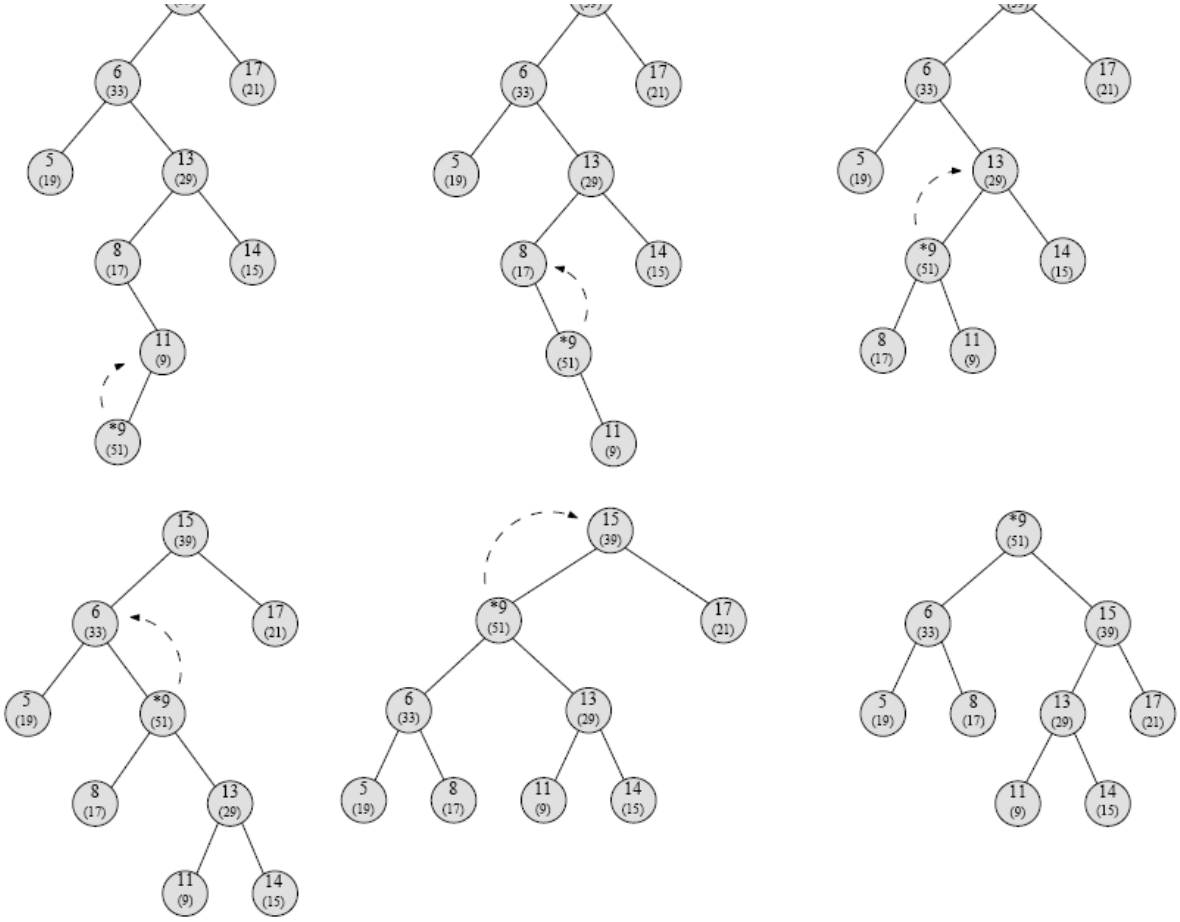
Mai jos avem pseudocodul pentru operația de inserare.

```
void insert(nod, cheie, prioritate) {
  if nod == nil
    nod = creza nou nod pe baza de cheie si prioritate
  else if cheie < nod.cheie
    insert(nod.stanga, cheie, prioritate)
  else
    insert(nod.dreapta, cheie, prioritate)

  if nod.stanga.prioritate > nod.prioritate
    rotireDreapta(nod)
  else if nod.dreapta.prioritate > nod.prioritate
    rotireStanga(nod)
}
```

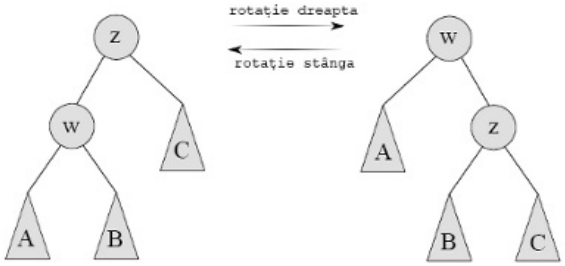
Spre exemplu, dacă am dori să inserăm nodul cu cheia 9 si prioritatea 51, pașii vor arată în felul urmator:





Se observă necesitatea rotirilor pentru a aduce nodul nou inserat în vârful arborelui (are prioritatea cea mai mare).

Cele două tipuri de rotiri sunt prezentate vizual în imaginea de mai jos:



Ștergerea

Operația de ștergere este inversul operației de inserare și se aseamănă foarte mult cu ștergerea unui nod în cadrul unui heap. Nodul pe care îl dorim a fi șters este rotit până când ajunge la baza arborelui, iar atunci este șters. Pentru a menține invariantul de heap, vom face o rotire stânga dacă fiul drept are o prioritate mai mare decât fiul stâng și o rotire drepta în caz contrar.

```
void sterge(nod, cheie) {
  if nod == nil
    return

  if cheie < nod.cheie
    sterge(nod.stanga, cheie)
  else if cheie > nod.cheie
    sterge(nod.dreapta, cheie)
  else if nod.stanga == nil si nod.dreapta == nil
    sterge nod
  else if nod.stanga.prioritate > nod.dreapta.prioritate
    rotireDreapta(nod)
    sterge(nod, cheie);
  else
    rotireStanga(nod)
    sterge(nod, cheie)
```

```
}
```

Exerciții

Pentru exerciții porniți de la acest schelet de laborator.

1. Implementați funcțiile de bază pentru un treap:
   - [0.5p] Căutare
   - [0.5p] Rotiri stânga și dreapta
   - [0.5p] Inserare
   - [0.5p] Ștergere
2. [1p] Realizați o parcurgere a treapului astfel încât să obțineți cheile sortate crescător/descrescător.
3. [2p] Realizați o parcurgere a treapului astfel încât să obțineți o structură arborescentă a priorităților, pentru a observa invariantul de heap.
4. [4p] Scrieți o funcție care să răspundă într-o complexitate de `O(logN)` la următoarea cerință: Care este cea de-a K-a cheie, în ordinea sortării crescătoare, care se află în treap?.

Referințe

- Fast Set Operations Using Treaps
- Randomized Binary Search Trees
- Balanced Search Trees