**HEAP BINAR**

**Definitie:**

Heap binar = arbore binar complet

Fiecare nod intern are exact 2 descendenti, cu exceptia eventual a ultimului nod intern de pe penultimul nivel. Frunzele se afla doar pe ultimele 2 niveluri, frunzele de pe ultimul nivel sunt ordonate de la stanga la dreapa (tot timpul se insereaza pe ultimul nivel cel mai in stanga, pana cand nivelul este plin, abia apoi se trece la urmatorul).

Max heap: info din fiecare nod este mai mare decat info din oricare descendent

Min heap: info din fiecare nod este mai mica decat info din oricare descendent

**Pozitiile in array:**

* Radacina se afla pe pozitia 0
* Parintele se afla pe pozitia (i-1)/2
* Fiul stang se afla pe pozitia 2\*i+1
* Fiul drept se afla pe pozitia 2\*i+2

**Inaltimea: log2 din n**

**Complexitatea operatiilor: O(log 2 din n)**

Daca se da un vector si intreaba daca e heap max -> verificare: se umple fiecare nivel pana cand se termina vectorul

**COZI DE PRIORITATE**

Operatii:

1. **Determinarea elem cu prioritate maxima**: radacina; O(1)
2. **Extragerea maximului**: Se plaseaza ultimul element din heap pe prima pozitie, se scade dimensiunea heap-ului cu o unitate iar apoi se reface prop de heap max; O(log 2 din n)

* se compara nodul nou din radacina cu nodurile de pe stanga si de pe dreapta, se ia max(st, dr), swap(parinte, max) pana cand ambii copii sunt mai mici decat parintele

1. **Insertia unui element nou**: Se mareste dim heap-ului, se plaseaza noul element pe ultima pozitie si apoi se stabileste pozitia corecta a acestiua; O(log 2 din n)

* se compara nodul nou adaugat cu parintele, daca parintele e mai mic -> swap, pana cand parintele e mai mare

**ARBORI BINARI DE CAUTARE**

Cheile din subarborele stang < parintele

Cheile din subarborele drept > parintele

Complexitatea operatiilor = inaltimea

Log 2 din n <= inaltimea <= n  
deci complexitatea poate fi logaritmica sau linara (worst case: liniara)

! Fiecare cheie nou inserata este o frunza

Numarul de arbori binari de cautare formati din n chei: (2n)!/(n!\*(n+1)!)

Operatii: cautarea binara, minim, maxim, succesor (prima pe dreapta, restul pe stanga), predecessor, insertie, stergere

Numarul minim de chei dintr-un arbore binar cu inaltimea h: h+1

Numarul maxim de chei dintr-un arbore binar cu inaltimea h: 2^(h+1)-1

Stergerea nodului x:

Daca x nu are fii -> null

Daca x are un singur fiu -> se inlocuieste cu acel fiu

Daca are ambii fii nenului -> x se inlocuieste cu succesorul, iar succesorul cu fiul sau drept, daca exista

**ARBORI AVL**

Fiecare nod are un factor de balansare = h(subarbore dreapta) - h(subarbore stanga)

Nod balansat: factorul de balansare = {-1, 0, 1}

Inaltimea < 2(log 2 din n)

Complexitatea operatiilor: O(log 2 din n)

**INSERTIA:**

Se recalculeaza factorul de balansare pt fiecare nod astfel:

Fie x nodul curent:

* Daca insertia este la stanga lui x -> scade factorul de balansare cu 1
* Daca insertia este la dreapta lui x -> creste factorul de balansare cu 1

Dupa recalcularea factorului de balansare:

* Daca fb = 0 => return
* Daca fb = -1 sau fb = 1 => se continua reechilibrarea
* Daca fb = -2 sau fb = 2 => debalansare => procedee de rebalansare:

1. x.fb = -2, x.st.fb = -1 => rotatie la dreapta
2. x.fb = 2, x.dr.fb = 1 => rotatie la stnaga
3. x.fb = -2, x.st.fb = 2 => rotatie stanga x.st , rotatie dreapta x
4. x.fb = 2, x.dr.fb = -1 => rotatie dreapta x.dr, rotatie stanga x

**STERGEREA:**

Fie x nodul curent:

* Daca stergerea a avut loc pe stanga lui x -> creste factorul de balansare cu 1
* Daca stergerea a avut loc pe dreapta lui x -> scade factorul de balansare cu 1

Dupa recalcularea factorului de balansare:

* Daca fb = -1 sau fb = 1 => return
* Daca fb = 0 => continua
* Daca x.fb = -2 si x.st.fb = -1 sau 0 => rotatie la dreapta
* Daca x.fb = 2, x.dr.fb = 0 sau 1 =>rotatie la stanga
* Daca x.fb = -2 si x.st.fb = 1 => rotatie stanga x.st , rotatie dreapta x
* Daca x.fb = 2 si x.dr.fb = -1 => rotatie dreapta x.dr, rotatie stanga x

**ARBORI ROSU NEGRU**

Proprietati:

* Fiecare nod este rosu sau negru
* Radacina este neagra
* Fiecare frunza e neagra si NIL
* Daca un nod e rosu => ambii fii sunt negri
* Parintele unui nod rosu e negru
* Black height = inaltimea de la radacina la nil
* Inaltimea maxima: 2log 2 din (n+1)
* Complexitatea operatiilor: O(log 2 din n)

**INSERTIA:**

Intotdeauna nodul care se insereaza este rosu.

z = nodul inserat

p = parintele lui z

b = bunicul lui z

u = unchiul lui z

Caz 1: u e rosu => recolorarea lui p, u, b (p, u negri, b rosu); se reia procedeul considerand z=b deoarece se poate contrazice prop 4

Caz 2: u este negru si:

1. z se afla pe dreapta lui p, iar p pe stanga lui b => rotatie la stanga dupa p
2. z se afla pe stanga lui p, iar p pe dreapta lui b => rotatie la dreapta

Caz 3: u este negru si:

1. z se afla pe stanga lui p, iar p pe stanga lui b => b rosu, p negru , rotatie la dreapta in jurul lui b
2. z se afla pe dreapta lui p, iar p pe dreapta lui b => b rosu, p negru, rotatie la stanga in jurul lui b

Imbogatirea arborilor: complexitate = O(log2 din n).

Proces: det informatiei suplimentare; verificare ca nu creste complexitateta

**B ARBORI**

Proprietati:

Fiecare nod x are:

* x.n = numarul de chei din x
* vectorul de chei
* x.leaf = true/ false
* legatura catre copii

**Fiecare nod intern x are x.n+1 copii.** Daca x are 3 chei => 4 copii

**Toate frunzele au aceeasi adancime**

**Nr de chei ale unui nod x: limitat de t = gradul minim astfel:**

* **fiecare nod x (! Radacina) -> cel putin t-1 chei => cel putin t copii**
* **fiecare nod x -> cel mult 2t-1(nod plin) chei => 2t copii**

**INSERTIA:**

De la inceput, prin fiecare nod prin care se trece se verifica daca acela nu este plin (2t-1 chei) -> daca este se face divizare

Daca radacina e plina: se creeaza un nod vid, se leaga vecea radacina ca descendent al noii radacini, se realizeaza operatia de divizare

DIVIZAREA: Mijlocul nodului urca, partea stanga devine copilul stang, partea dreapta devine copilul drept.

!Inaltimea unui B arbore creste doar prin divizarea radacinii

**STERGEREA:**

* Rotatii (fie x nod curent, muta ultima cheie a lui x.st in locul lui x, si pe x ca prima cheie a lui x.dr, iar copilul lui x.st[n] il muta pe stanga lui x, deoarece e mai mare decat noul parinte, fiind pe dreapta lui initial).
* Fuziune: nu poate fi realizata decat intre 2 frati vecini, fii ai aceluiasi nod x, amandoi avand exact t-1 chei. Parintele x trebuie sa aiba cel putin t chei. Se considera nodul x si y si z doi copii ai lui x. Fuziunea lui x cu y presupune: reuniunea celor doua noduri, iar parintele lor (x[i]) coboara din nodul x in noul nod rezultat intre elementele lui x si ale lui y.

Pasi stergere:

* Se porneste de la radacina pana la intalnirea nodului care contine informatia ce va fi stearsa
* Fie x nodul curent. Pe parcurs ne asiguram ca orice nod in care se coboara are cel putin t-1 chei (altfel prin stergere nu se vor mai respecta proprietati).

*Caz 1:*

X e frunza si k (elem ce va fi sters) este cheie a lui x. => se sterge k

*Caz 2:*

X nu e frunza si k e cheie a lui x. Atunci:

* Daca copilul de pe stanga lui k are nr de chei >=t atunci se cauta k’ predecesorul lui k, se sterge recursive k’ si se inlocuieste k cu k’ in x
* Daca copilul de pe dreapta lui k are nr de chei >= t atunci se cauta k’ succesorul lui k, se sterge recursive k’ si se inlocuieste k cu k’
* Daca ambii copii au exact t-1 chei atunci: fuziune -> nodul nou va contine cheia k => stergere recursiva

*Caz 3:*

X nu e frunza si k nu e cheie a lui x

Se determina i astfel incat x[i-1]<k<x[i] => cheia trebuie cautata in subarborele x.c[i].

Daca acest descendent are doar t-1 chei atunci:

* Daca x.c[i] are un frate vecin cu cel putin t chei => rotatie astfel incat o cheie din acel frate sa urce la parintele x si o cheie din x sa coboare in x.c[i]
* Daca ambii frati vecini au exact t-1 chei => fuziune, apoi se sterge k din nodul rezultat prin fuziune.

**HASH TABLES**

**Adresare directa:** T[i] = i

**Tabele de repartizare:**

Se foloseste o functie de repartizare. Pozitia elementului in vector se determina pe baza functiei de repartizare (ex: h(k) = k mod m, k – elementul, m – size array). Coliziunile se rezolva cu ajutorul listelor.

Metode de repartizare:

* **Metoda diviziunii:** h(k) = k mod m
* **Metoda multiplicarii:** h(k) = [m(kA – [kA])] unde [] – parte intreaga, A = 0,61…
* **Repartizarea universala:** Se utilizeaza mai multe functii de repartizare
* **Adresare directa:** fiecare pozitie are un singur element

1. **Repartizare cu testare liniara: h(k,i) = (h1(k)+i) mod m**
2. **Repartizare cu testare patratica: h(k,i) = (h1(k)+c1i+c2\*i^2) mod m**
3. **Dubla repartizare: h(k,i) = (h1(k)+i\*h2(k)) mod m**

**COMPLEXITATI**

|  |  |  |
| --- | --- | --- |
| **Structura de date** | **Operatie** | **Complexitate** |
| Stiva | Push | 1 |
|  | Pop | 1 |
| Coada | Push | 1 |
|  | Pop | 1 |
| Liste simplu inlantuite | Search | N |
|  | Insert | 1 |
|  | Delete | N |
| Liste dublu inlantuite | Search | N |
|  | Insert | 1 |
|  | Delete | 1 (cautare inainte) |
| Heap / priority queue | Max heap (reface proprietate) | Log 2 din n |
|  | Construire heap | N |
|  | Determinare maxim | 1 |
|  | Extragere maxim | Log 2 din n |
|  | Creste prioritate | Log 2 din n |
|  | Insertie/Stergere | Log 2 din n |
| Arbori binari de cautare | Toate op | Average: log n , Worst: n |
| AVL | Toate op | Log 2 din n |
| ARN | Toate op + imbogatire | Log 2 din n |
| B Arbori | Toate op | T\* log t din n |