FIBONACCI HEAPS REVISTED

-Referat – Structuri de date avansate – 2024

***Toma David***

**Ce este heap-ul Fibonacci ?**

Heap-ul Fibonacci este o structură de date bine cunoscută, apreciată pentru capacitatea sa de a gestiona ștergerile în timp amortizat logaritmic și de a executa celelalte operațiuni specifice unui heap în timp amortizat O(1).

**“Cuprins”:**

În acest articol, s­­­-a propus să se exprime și să se extinda posibilitățile de proiectare ale acestei structuri. Se prezinta o versiune optimizată care aduce următoarele îmbunătățiri semnificative față de designul original:

1. Fiecare heap este organizat sub forma unui singur arbore ordonat, eliminând necesitatea utilizării unui set de arbori.
2. Operațiile de reducere a cheilor (decrease-key) sunt simplificate, necesitând o singură tăiere și o serie de modificări ale rangurilor, înlocuind procesul complex al tăierilor în cascadă.
3. Toate comparațiile efectuate de algoritm sunt stocate explicit în structura de date, evitând orice pierdere inutilă de informație.

De asemenea, se demonstreaza, printr-un exemplu, că fără tăieri sau modificări în cascadă, atât versiunea originală, cât și cea propusă nu pot atinge nivelul de eficiență dorit.

În plus, se exploreaza noi direcții interesante de design, sugerând metode alternative pentru ajustarea rangurilor, inclusiv o strategie randomizată inspirată de lucrările anterioare ale lui Karger. Analiza detaliată a acestor alternative rămâne deschisă, oferind oportunități pentru cercetări viitoare.

Acest articol cuprinde 7 sectiuni:

* Sectiunea 1: Introducere
* Secțiunea 2: Descrierea structurii propuse.
* Secțiunea 3: Analiza teoretică.
* Secțiunea 4: Implementare.
* Secțiunea 5: Importanța ajustărilor cascadate.
* Secțiunea 6: Posibilități alternative.
* Secțiunea 7: Concluzii.

**Sectiunea 1: Introducere**

Heap-ul este o structură de date care gestionează un set de elemente, fiecare având o cheie dintr-un univers total ordonat. Operațiile suportate de acesta includ:

* crearea unui heap gol (**make-heap**)
* găsirea minimului (**find-min**)
* inserarea (**insert**)
* ștergerea minimului (**delete-min**)
* combinarea heap-urilor (**meld**)
* reducerea cheii unui element (**decrease-key**)
* ștergerea unui element specific (**delete**)

Structura inițială este distrusă în timpul acestor operații, iar gestionarea eficientă a particionării poate necsita structuri suplimentare

Inventat de Fredman și Tarjan, **Fibonacci Heap** optimizează operațiile *delete-min* și *delete* pe un heap cu n elemente la O(log n) în timp amortizat, în timp ce restul operațiilor au complexitatea O(1). Aplicațiile includ algoritmi rapizi pentru drumuri minime (ex: Dijkstra) și arbori de acoperire minima. Deși au fost propuse numeroase variante, structura originală rămâne una dintre cele mai simple de implementat.

**Sectiunea 2: Descrierea structurii propuse**

In versiunea simplificată a Fibonacci Heap, structura este reprezentată de un arbore rădăcină ale cărui noduri conțin elementele heap-ului. Arborele respectă proprietatea de ordine a heap-ului: fiecare copil are o cheie mai mare sau egală cu cea a părintelui său. Astfel, rădăcina reprezintă elementul cu cheia minimă, iar operația find-min returnează rădăcina arborelui.

Principii de bază ale operațiilor

1. Legarea nodurilor (**linking**):
   * Pentru a combina două heap-uri, se compara rădăcinile lor și se face rădăcina cu cheia mai mică a parintelui cu cheia mai mare. Aceasta este baza pentru operații precum meld sau insert.
2. Ștergerea minimului (**delete-min**):
   * Se elimină rădăcina arborelui, iar copiii acesteia devin rădăcini separate.
   * Se efectuează legături succesive între rădăcini până când rămâne un singur arbore.
3. Reducerea cheii (**decrease-key**):
   * Cheia unui nod este redusă, iar dacă acesta nu este rădăcină, este „tăiat” din arbore, formând un nou arbore. Ulterior, este legat de rădăcina originală.
4. Ștergerea unui nod arbitrar (**delete**):
   * Se reduce cheia nodului la o valoare mai mică decât toate celelalte chei, urmată de o operație delete-min.

**Eficiența prin utilizarea rangurilor**

Pentru a minimiza numărul de legături, fiecărui nod i se atribuie un rang (un număr întreg). O legătură „corectă” (**fair link**) între două rădăcini poate fi realizată doar dacă rangurile acestora sunt egale. Această legătură crește rangul rădăcinii cu cheia mai mică și o face părinte al celeilalte rădăcini. În contrast, o legătură „simplă” (**naïve link**) ignoră rangurile.

Operațiile precum delete-min folosesc legături corecte pentru a menține eficiența, în timp ce operațiile obișnuite, precum insert, utilizează legături simple.

**Marcarea nodurilor**

Fiecare nod are o stare: marcat sau nemarcat, indicată de o variabilă booleană. Un nod marcat poate deveni nemarcat sau poate fi tăiat în timpul operațiilor de reducere a cheii. Această marcaj ajută la gestionarea mai eficientă a ajustărilor de rang.

Cod pseudo pentru ajustarea rangurilor:

*h.state ← unmarked*

*y ← x*

*repeat:*

*y ← y.parent*

*if y.rank > 0:*

*y.rank ← y.rank − 1*

*y.state ← not y.state*

*until y.state = marked*

Acest proces unifică ajustările de rang și stările nodurilor în timpul operației **decrease-key.**

Diferențe față de structura original:

1. Structura propusă utilizează un singur arbore ordonat, nu o mulțime de arbori.
2. În locul tăierilor în cascadă (cascading cuts), se implementează ajustări de ranguri, reducând redundanțele.
3. Rezultatele tuturor comparațiilor de chei sunt menținute explicit în structura de date, eliminând pierderile de informații din implementarea originală.

Această versiune simplificată reduce complexitatea implementării, păstrând însă performanța ridicată a operațiilor de bază.

**Sectiunea 3: Analiza teoretica**

În analiza noastră, se folosesc conceptele de **copii activi** și **copii pasivi**. Un nod care devine copil printr-o legătură corectă (**fair link**) este activ, în timp ce unul care devine copil printr-o legătură simplă (**naïve link**) este pasiv. Un copil activ care este marcat și apoi nemarcat devine pasiv atunci când este nemarcat. Acesta rămâne pasiv până devine din nou rădăcină și copil printr-o legătură corectă. Aceste concepte sunt utilizate doar în aceasta analiza, nu și în algoritm.

**Lemă 3.1:** Orice nod x are cel puțin x.rank copii activi.  
**Demonstrație:** Singura modalitate prin care rangul unui nod poate crește este printr-o legătură corectă, care adaugă un copil activ. Când un nod pierde un copil activ, rangul său scade cu 1, cu excepția cazului în care este deja 0.

Definim **dimensiunea unui nod** x, notata x.size, ca fiind numărul său de descendenți, inclusiv el însuși. Demonstrarea arată că rangul unui nod este cel mult **logaritmic** în raport cu dimensiunea sa.

**Lemă 3.2:** Dimensiunea minimă a unui nod de rang cel puțin k este nk≥=ϕk, unde ϕ este raportul de aur.  
**Demonstrație:** Folosind inducția, demonstrăm că dimensiunea unui nod de rang k este legată de valorile Fibonacci.

**Corolar 3.3:** Un nod de dimensiune n are rang cel mult logϕn

Pentru a analiza eficiența amortizată a operațiilor, folosim cadrul standard bazat pe potențial. Timpul amortizat al unei operații este timpul său real plus creșterea netă a potențialului cauzată de operație. Dacă potențialul inițial este zero și potențialul final este non-negativ, atunci timpul total real al operațiilor este mai mic sau egal cu suma timpurilor amortizate.

**Eficiența operațiilor:**

* **Timpul amortizat** al operațiilor make-heap, find-min, sau meld este 1.
* **Timpul amortizat** pentru insert este 2.
* **Timpul amortizat** pentru decrease-key este cel mult 5.
* **Timpul amortizat** pentru delete-min pe un heap cu nn elemente este cel mult 3logϕn

**Lemă 3.4**

Timpul amortizat al operațiilor make-heap, find-min și meld este 1, al operației insert este 2, al operației decrease-key este cel mult 5, iar al operației delete-min pe un heap cu n elemente este cel mult 3logϕn.

**Demonstrație:**

* **Legătura simplă (naïve link)** nu schimbă potențialul: potențialul rădăcinii care devine copil scade cu 1, dar potențialul rădăcinii care rămâne rădăcină crește cu 1 (gradul acesteia crește, iar rangul nu se schimbă).
* **Legătura corectă (fair link)** reduce potențialul cu 1: în acest caz, potențialul rădăcinii care rămâne rădăcină nu se schimbă, deoarece atât gradul, cât și rangul acesteia cresc cu 1.

Operațiile make-heap, find-min și meld nu schimbă potențialul, deci timpul lor amortizat este 1.  
Operația insert creează o nouă rădăcină cu potențial 1, deci timpul amortizat este 2, indiferent dacă se face o legătură simplă sau corectă.

Pentru decrease-key:

* Dacă rădăcina nu este marcată, timpul amortizat este 1.
* Dacă nodul nu este rădăcină, fiecare iterație a buclei de scădere a rangului nemarchează un nod și poate scădea rangul acestuia cu 1, reducând potențialul cu cel puțin 1. Ultima iterație marchează un nod și poate scădea rangul cu 1, crescând potențialul cu cel mult 3.
* Tăierea nodului x din părintele său nu schimbă potențialul, iar legătura dintre x și rădăcina originală crește potențialul cu 1, dar reduce potențialul părintelui cu 1.

Astfel, timpul amortizat pentru decrease-key este cel mult 5.

**Teoremă 3.5:** Timpul total estimat pentru o secvență de operații pe heap-uri, pornind de la heap-uri goale, este O(1) pe operație, cu excepția operațiilor delete-min și delete, care au timp O(logn) unde n este numărul de elemente din heap.

Această analiză este susținută de Lemă 3.4 și de observațiile legate de potențialul nodurilor și legăturile realizate în timpul operațiilor.

**Sectiunea 4: Implementare**

Implementarea sugerata a **Fibonacci Heap** simplificat urmează designul original, folosind liste dublu legate pentru copii, ceea ce permite realizarea tăierilor în timp O(1). Fiecare nod are indicii către părintele său și primul său copil. În operația delete-min, folosim un array global de noduri, indexate după rangul lor, pentru a stoca rădăcinile și a le combina corect.

CREARE NOD

*make-item(info, v):*

*x ← new node*

*x.info ← info*

*x.key ← v*

*x.rank ← 0*

*x.state ← unmarked*

*x.child ← null*

*return x*

CREARE HEAP GOL

*make-heap():*

*return null*

*-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------*

INSERARE NOD

*insert(x, h):*

*return meld(x, h)*

*-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------*

COMBINARE HEAP-URI

*meld(g, h):*

*if g = null: return h*

*if h = null: return g*

*return link(g, h)*

*-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------*

STERGERE MINIM

*delete-min(h):*

*x ← h.child*

*max-rank ← 0*

*while x ≠ null:*

*y ← x*

*x ← x.after*

*while A[y.rank] ≠ null:*

*y ← link(y, A[y.rank])*

*A[y.rank] ← null*

*y.rank ← y.rank + 1*

*A[y.rank] ← y*

*if y.rank > max-rank:*

*max-rank ← y.rank*

*for i ← 0 to max-rank:*

*if A[i] ≠ null:*

*if x = null: x ← A[i]*

*else: x ← link(x, A[i])*

*A[i] ← null*

*return x*

*-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------*

SCADERE CHEIE

*decrease-key(x, v, h):*

*x.key ← v*

*if x = h: return h*

*h.state ← unmarked*

*decrease-ranks(x)*

*cut(x)*

*return link(x, h)*

*-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------*

*FUNCTII AUXILIARE*

LEGATURA INTRE NODURI

*link(x, y):*

*if x.key > y.key:*

*add-child(x, y)*

*return y*

*else:*

*add-child(y, x)*

*return x*

*-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------*

ADAUGARE COPIL

*add-child(x, y):*

*x.parent ← y*

*z ← y.child*

*x.before ← null*

*x.after ← z*

*if z ≠ null: z.before ← x*

*y.child ← x*

*-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------*

TAIERE NOD

*cut(x):*

*y ← x.parent*

*if y.child = x: y.child ← x.after*

*if x.before ≠ null: x.before.after ← x.after*

*if x.after ≠ null: x.after.before ← x.before*

*-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------*

SCADERE RANGURI

*decrease-ranks(y):*

*repeat:*

*y ← y.parent*

*if y.rank > 0: y.rank ← y.rank − 1*

*y.state ← not y.state*

*until y.state = marked*

*-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------*

**Modificări posibile și îmbunătățiri:**

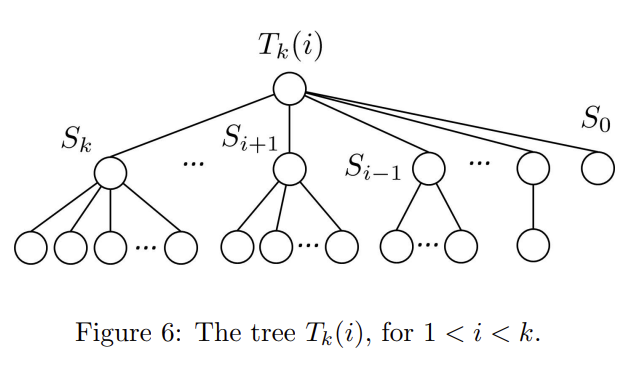
1. **Reducerea numărului de pointeri**: În loc de două pointere (before și after), putem utiliza un singur pointer pentru a economisi spațiu.
2. **Actualizarea heap-urilor existente**: În loc să returnăm un nou heap, putem modifica heap-ul existent pentru a economisi timp și spațiu.
3. **Legături leneșe (lazy links)**: Legăturile între arbori sunt realizate doar atunci când sunt necesare, pentru a răspunde la întrebările de tip find-min. Astfel, find-min devine o operație amortizată, mai eficientă în anumite cazuri.

**Heuristici pentru performanță:**

1. **Heuristica ordinii heap-ului**: Tăierea unui nod x se face doar dacă este copil și perturba ordinea heap-ului, economisind astfel timp.
2. **Heuristica rangului crescător**: Operația decrease-key oprește bucla de scădere a rangului atunci când întâlnește un nod cu rangul mai mare sau egal cu părintele său. Aceasta reduce timpul în cel mai rău caz la O(log(n)).
3. **Heuristica copilului pasiv**: Nodurile sunt marcate ca pasive, reducând numărul de iterații ale buclei de scădere a rangului și simplificând implementarea.

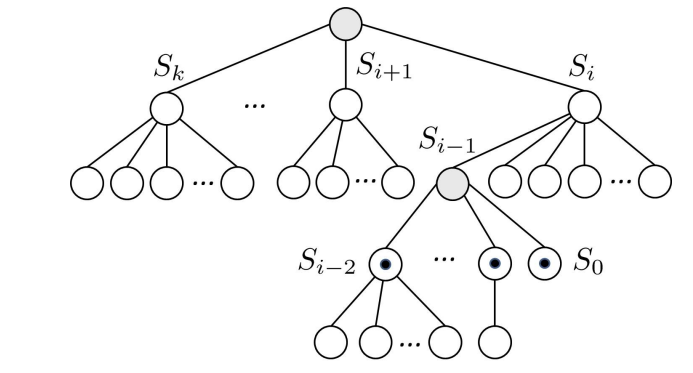
**Sectiunea 5: Importanta ajustarii casacadate**

S-au înlocuit tăierile în cascadă din Fibonacci heaps cu scăderi de rang în cascadă. Fredman întreabă dacă tăierea poate fi complet evitată. O variantă simplificată păstrează rangurile, dar elimină marcarea nodurilor. Pentru a scădea cheia unui nod non-rădăcină, tăiem nodul din părintele său, scădem rangul părintelui (dacă este pozitiv) și legăm nodul la rădăcina originală. Această structură este un **heap fără cascadă**.

 Construirea unui heap fără cascadă cu n noduri necesită O(n3/2) operații, iar o secvență de operații poate dura Ω(m4/3). Definim arborii Sk și Tk(i), care pot fi construiți pentru a demonstra că eliminarea cascadei crește semnificativ timpul de execuție, confirmând astfel că tăierea în cascadă este necesară pentru eficiența Fibonacci heaps.

Putem transforma **Tk(0)** în **Tk+1(k + 1)** inserând un nou element cu o cheie mai mare decât rădăcina. Pentru a transforma **Tk(i)** cu i>0i > 0i>0 în **Tk(i - 1)**, realizăm două inserări, un **delete-min** și i−1 operații decrease-key. Astfel:

1. Inserăm două elemente cu chei mai mari decât rădăcina.
2. Facem un **delete-min**, iar după eliminarea rădăcinii originale, obținem trei rădăcini de rang 0.
3. Legăm una dintre noile rădăcini cu rădăcina originală a lui S0 printr-o legătură corectă, apoi le legăm succesiv cu S1,S2,...,Si−1.
4. Alegem cheile astfel încât rădăcina Si−1să aibă cheia cea mai mică și unul dintre noile noduri să aibă cheia a doua ca mărime.
5. Creăm un arbore **Si**, dar cu un frate care are copii din rădăcinile S0,S1,...,Si−2.
6. Combinăm arborii rămași folosind legături simple, iar la final realizăm i−1 decrease-key pe rădăcinile S0,S1,...,Si−2, transformându-le în copii ai rădăcinii finale. Astfel, obținem **Tk(i - 1)**.



Transformarea **Tk(i)** în **Tk(i-1)** după două inserări și un **delete-min** presupune legături între arbori și i−1 operații decrease-key. Prin inducție, putem transforma **Tk(k)** în **Tk+1(k+1)** în O(k2) operații, iar **Tk(k)** poate fi construit de la zero în O(k3) operații.

**Teorema 5.1** :Afirmă că pentru orice n, un heap fără cascadă cu n noduri poate avea o operație **insert** urmată de **delete-min** care ia Ω(n1/2) timp. Un astfel de heap poate fi construit în O(n3/2) operații. De asemenea, o secvență de m operații poate dura Ω(m4/3) timp.

Exemplul arată că, dacă tăierea în cascadă este eliminată, structura de date rezultată poate avea un timp fracțional-polinomial pe operație, răspunzând astfel negativ întrebării lui Fredman.

**Sectiunea 6: Posibilități alternative**

Având în vedere că scăderile de rang în cascadă sunt necesare, o întrebare naturală este dacă există o metodă mai simplă de a decide când să le aplicăm. Propunem patru metode posibile, dintre care trei sunt determinate și una este aleatorie. Nu s-a reușit să se analizeze complet niciuna dintre aceste metode, iar acest lucru rămâne o problemă deschisă interesantă.

Două dintre cele trei metode determinate simplifică două dintre heuristicile din Secțiunea 4. **Marcarea activă** simplifică heuristica copilului pasiv făcând nodurile marcate și active identice, iar rădăcinile sunt implicit nemarcate. Un nod care devine copil printr-o legătură corectă devine marcat. Bucla de scădere a rangului se oprește atunci când întâlnește un nod nemarcat.

Celelalte două metode determinate elimină complet marcarea: **metoda naivă de rang crescător** oprește bucla atunci când întâlnește un nod cu rangul mai mare sau egal cu al părintelui, iar **metoda de rang 0** oprește bucla atunci când întâlnește un nod cu rangul 0.

O altă metodă de eliminare a marcării este utilizarea **cascadei aleatorii**, care la fiecare iterație a buclei de scădere a rangului se oprește cu o probabilitate de 1/2. Aceasta este o variantă a metodei de **tăiere aleatorie** propusă de Karger, care nu folosește marcarea, ci se oprește aleatoriu la fiecare iterație a buclei de tăiere.

**Cascada aleatorie** are timpul așteptat O(1) pentru decrease-key, dar nu oferă eficiența Fibonacci heaps. Analiza recentă a lui Li și Peebles arată că, deși tăierea aleatorie are o limită superioară similară cu cascada aleatorie, limita inferioară nu se aplică în același mod. Astfel, nu se știe încă dacă cascada aleatorie atinge eficiența Fibonacci heaps.

**Sectiunea 7: Concluzii**

A fost prezentata o versiune a unui singur arbore pentru Fibonacci heaps, în care fiecare operație decrease-key necesită doar o singură tăiere. Nu este singura structură de heap cu aceste proprietăți; de exemplu, rank-pairing heaps au fost concepute pentru a necesita doar o tăiere per decrease-key, mai există și o versiune de tip un arbore pentru aceste heap-uri. Totuși, se considera că aceasta structura este cea mai simplă dintre toate structurile de date cunoscute care au aceleași limite de timp amortizat ca Fibonacci heaps. Un alt avantaj al acestei structuri este că rezultatele tuturor comparațiilor de chei sunt reprezentate direct în structură (prin legături).

Deși pairing heaps este o structură mai simplă și mai rapidă în practică, aceasta nu susține decrease-key în O(1) timp amortizat, iar obținerea unei limite stricte pentru acest timp în cazul pairing heaps este o problemă deschisă interesantă.

S-a arătat că, fără schimbările de rang în cascadă, aceasta structura nu atinge eficiența Fibonacci heaps, rezolvând astfel o problemă deschisă de Fredman. Au fost propuse patru metode simplificate de cascade și au fost lasae analizele acestora ca probleme deschise interesante.