Semestrálny projekt z MI-PAL, LS 2016/17

Implementácia dátových štruktúr

Treap a Implicit Treap

Peter Mitura (miturpet@fit.cvut.cz)

13. mája 2017

Treap

Treap (názov zložený zo slov *tree* a *heap*) je stromová dátová štruktúra, ktorá je kombináciou binárneho vyhľadávacieho stromu (ďalej *BVS*) a haldy. Základy tejto štruktúry vychádzajú z *kartézskych stromov* [1] – binárnych stromov postavených nad poľom, ktoré spĺňajú haldovú podmienku a sú usporiadané tak, že ich in-order priechod vypíše originálne pole.

Treapy sú rozšírením tejto myšlienky. Tiež sú to binárne stromy, v každom svojom vrchole ale ukladajú dve hodnoty: *kľúč* a *prioritu*, pri čom simultánne sa udržujú zoradené ako BVS podľa kľúča a ako minimová halda podľa priority. Nakoľko sa najčastejšie používajú ako implementácia abstraktného dátového typu *množina* , kľúče vrcholov je logické využiť na uloženie prvkov v množine. Priority potom slúžia jedine k vyvažovaniu stromu, pri čom ďalej ukážeme, že ak ich budeme voliť náhodne zo spojitého rovnomerného rozloženia, strom získa z hľadiska efektivity operácií nad kľúčmi zaujímavé vlastnosti.

Treap v tejto podobe bol po prvý krát popísaný Aragonom a Seidlom v roku 1989 [2], s určitými rozdielmi v názvosloví [1]. Základné operácie, podporované touto štruktúrou zodpovedajú, ako už bolo zmienené, abstraktnému dátovému typu množina:

- $Insert(x)$ – vloží prvok x do stromu, ak sa tam ešte nenachádza
- $Find(x)$ – zistí, či sa prvok x nachádza v strome
- $Delete(x)$ – zmaže prvok x zo stromu, ak sa v ňom nachádza

[1] Aragon a Seidel pri pojme *Treap* nešpecifikovali spôsob voľby priority, náhodnú hodnotu jej dávali až v štruktúre ktorú pomenovali *Randomized Search Tree* – tento pojem sa ale neskôr začal využívať pre zložitejšiu dátovú štruktúru a označenie Treap sa zaužíval pre verziu s náhodnou voľbou priority.

Podobne ako u iných BVS, aj tu od univerza vkladaných prvkov požadujeme, aby tvorilo úplne usporiadanú množinu a reláciu usporiadania je nutné definovať pri inicializácií štruktúry (štandardne v implementácií využívame funkciu `std::less`, ak je dostupná). Základnou myšlienkou potom je, že všetky operácie implementujeme analogicky ako v bežnom BVS, s jediným rozdielom že v nich chceme zachovať haldové usporiadanie podľa priority bez zvýšenia asymptotickej časovej zložitosti.

Ak by sa nám to podarilo (konkrétnu implementáciu popíšeme nižšie), zložitosť operácií nad stromom T by dosiahla $\mathcal{O}(h(T))$, kde $h(T)$ je hĺbka daného stromu. Tá by v nevyvažovanom BVS mohla dosiahnúť až $\mathcal{O}(n)$, kde n je počet prvkov v strome, čo pre nás rozhodne nie je priaznivé. Kľúčové pre celú štruktúru je teda nasledujúce tvrdenie:

Veta 1. Ak je priorita každého vrchola nezávislá, rovnomerne rozložená spojitá náhodná veličina, stredná hodnota hĺbky *ľubovoľného* vrchola v Treape je $\mathcal{O}(\log n)$, kde n je počet vrcholov v Treape.

Pri dôkaze tejto vety sa inšpirujeme postupom z kurzu na Univerzite v Illinois [3], ktorý je do istej miery prehľadnejší než dôkaz v originálnom článku.

Počet vrcholov v strome budeme nadalej označovať ako n a ďalej si označme vrchol s k-tým najmenším kľúčom ako x_k. Zároveň si zavedieme sadu premenných A_k^i, ktorá bude indikovať či je vrchol x_i predkom vrchola x_k (predka zavádzame tak, že vrchol nie je predkom sám sebe):

$$A_k^i = \begin{cases} 1 & \text{ak je } x_i \text{ predkom } x_k \\ 0 & \text{inak} \end{cases}$$

Pomocou indikátora A dokážeme jednoducho vyjadriť hĺbku vrchola $h(x_k)$:

$$h(x_k) = \sum_{i=1}^{n} A_k^i \tag{1}$$

Takže stredná hodnotu hĺbky vrchola x_k bude rovná sume pravdepodobností nad $i \in 1, \ldots, n$ že x_i je predkom x_k.

$$\mathbb{E}(h(x_k)) = \sum_{i=1}^{n} P(A_k^i = 1) \tag{2}$$

Veta 1 o tejto hodnote tvrdí, že je rovná $\mathcal{O}(\log n)$. Pre overenie tejto rovnosti si zavedieme ešte jedno značenie, $X(i,k)$ bude označovať množinu vrcholov $\{x_i, x_{i+1}, \ldots, x_k\}$ alebo $\{x_k, x_{k-1}, \ldots, x_i\}$ podľa toho či $i < k$ alebo $i > k$.

Lemma 1. Pre každé $i \neq k$ platí, že x_i je predkom x_k vtedy a len vtedy, keď má x_i najnižšiu prioritu spomedzi vrcholov v $X(i,k)$.

Dôkaz. V prípadoch kedy je x_i alebo x_k koreňom stromu je rovno vidieť, že lemma platí (x_i resp. x_k má vtedy najnižšiu prioritu z celého stromu), obmedzíme sa teda na prípad kedy je koreňom x_j pre nejaké $i \neq j \neq k$. Potom môžeme rozlíšiť dve možnosti: x_i a x_k sú buď v rovnakom alebo rozdielnom podstrome x_j.

Ak sú v rôznom podstrome, tak x_j určite patrí do $X(i,k)$ (jedná sa o BVS, takže jedna hodnota z dvojice i, k musí byť menšia než j, a jedna väčšia než j). Priorita x_j je

ale najnižšia z celého stromu, takže aj z $X(i,k)$, čo by podľa lemmy malo znamenať že x_i nie je predkom x_k (a nakoľko sú v rôznych podstromoch, je to aj pravda).

Pre x_i a x_k v jednom podstrome môžeme uvažovať len daný podstrom a v ňom postupovať analogicky ako v prípadoch vyššie (rekurzívne). □

Pravdepodobnosť že vrchol x_i je predkom x_k je teda zhodná s pravdepodobnosťou že x_i má minimálnu prioritu z $X(i,k)$ a nakoľko je táto šanca pre všetky vrcholy z daného rozsahu rovnaká, môžeme ju jednoducho vyjadriť ako 1/počet vrcholov v rozsahu:

$$P(A_k^i = 1) = \begin{cases} \dfrac{1}{k-i+1} & \text{ak } i < k \\ \dfrac{1}{i-k+1} & \text{ak } i > k \\ 0 & \text{ak } i = k \end{cases}$$

Zostáva už len vyjadriť hodnotu strednej hodnoty $\mathbb{E}(h(x_k))$. Z (2) máme jej vzťah k $P(A_k^i = 1)$, ich sumu môžeme rozdeliť na 2 časti podľa vzťahu medzi i a k:

$$\sum_{i=1}^{n} P(A_k^i = 1) = \sum_{i=1}^{k-1} \frac{1}{k-i+1} + \sum_{i=k+1}^{n} \frac{1}{i-k+1} \tag{3}$$

(prvá suma je prípad s $i < k$, druhá $i > k$)

Tieto sumy ďalej môžeme previesť na harmonické rady, pre ktoré máme známe odhady:

$$\sum_{i=1}^{k-1} \frac{1}{k-i+1} + \sum_{i=k+1}^{n} \frac{1}{i-k+1} = \sum_{j=2}^{k} \frac{1}{j} + \sum_{j=2}^{n-k} \frac{1}{j} = H_k - 1 + H_{n-k} - 1 \tag{4}$$

$$H_k - 1 + H_{n-k} - 1 < \ln k + \ln(n-k) - 2 < 2\ln n - 2 \tag{5}$$

čo asymptoticky dáva

$$\mathbb{E}(h(x_k)) < 2\ln n - 2 = \mathcal{O}(\log n) \tag{6}$$

čím je veta 1 dokázaná.

Implementácia operácií

Ako sme už uviedli vyššie, vďaka predpokladu logaritmickej hĺbky stromu v priemernom prípade nám stačí implementovať operácie *Insert*, *Delete* a *Find* v zložitosti zodpovedajúcej hĺbke stromu.

Pre zjednodušenie ich implementácie (ktorá by inak musela byť riešená komplikovanými rotáciami) sa bežne zavádza dvojica primitívnych operácií *Split* a *Merge*:

- $Split(x,t)$ – rozdelí Treap s koreňom vo vrchole t na dva Treapy: ľavý Treap l, v ktorom sú kľúče všetkých vrcholov menšie rovné hodnote x a pravý Treap r, v ktorom sú kľúče všetkých vrcholov väčšie než x. Originálny Treap v tomto procese zanikne.

- $Merge(l, r)$ – spojí dva Treapy l a r do jedného, pod podmienkou že maximálny kľúč z l je menší ako minimálny kľúč z r. Oba originálne Treapy zaniknú a sú nahradené novým spojeným stromom.

Pre potreby práce so stromom označíme ľavého syna vrchola v ako $v.left$ a pravého ako $v.right$. K Treapu bežne pristupujeme skrz jeho koreňový vrchol, ak teda niekedy dostaneme ako argument funkcie Treap t, budeme sa ako na t zároveň odkazovať na jeho koreň (čo je zhodné so sémantikou implementácie).

Realizácia operácie $Split$

Operácia $Split(x, t)$ môže naraziť na celkom 3 prípady:

- Ak je x menšie ako kľúč v t, tak ako r vrátime vrchol t s jeho pôvodným pravým podstromom, a ako ľavý podstrom mu priradíme pravý strom z výsledku operácie $Split(x, t.left)$. Takto zvolené r bude určite spĺňať haldovú podmienku (pôvodný strom bol Treap, takže všetky prvky z ľavého podstromu t musia mať väčšiu prioritu ako r) a zároveň sú všetky kľúče v ňom väčšie ako x (pre vrchol t a jeho pravý podstrom to platí nakoľko sa jedná o BVS, pre pravý strom zo splitu ľavého podstromu t to platí z definície operácie $Split$).

  Ako Treap l ďalej vrátime ľavý strom získaný operáciou $Split(x, t.left)$ (ktorú sme už zavolali pre zísakanie Treapu r). Z definície $Split$-u sa musí jednať o Treap so všetkými kľúčmi menšími než x, takže spĺňa všetky podmienky.

- Ak je x väčšie alebo rovné kľúču v t, potom vykonáme reverznú verziu minulého prípadu – ako Treap l vrátime vrchol t s jeho originálnym ľavým podstromom a jeho pravý podstrom nahradíme ľavým Treapom z operácie $Split(x, t.right)$. Ako pravý Treap vrátime pravý strom získaný tou istou funkciou.

- Ak je t prázdny Treap (napr. v situácií kedy sa zanoríme do neexistujúceho syna), vrátime ako l aj r prázdny Treap.

Je zjavné, že operácia $Split$ sa v každej instancií môže rekurzívne zanoriť maximálne jeden krát do vrcholu o jednu úroveň nižšie. Jej časová zložitosť v najhoršom prípade je teda lineárna v hĺbke stromu, čo v priemere podľa vety 1 dáva asymptoticky $O(\log n)$.

Realizácia operácie $Merge$

Podmienka kladená na kľúče pri spájaní dvoch Treapov robí udržanie BST poradia stromu pomerne jednoduché, jediné o čo sa musíme starať je zachovanie haldovej podmienky v prioritách. Pri zavolaní funkcie $Merge(l, r)$ opäť môžu nastať tri prípady:

- Priorita l je menšia než priorita r. V tom prípade chceme zlúčený Treap zakoreniť vo vrchole l, a ľavého syna l môžeme ponechať tak ako je (spĺňa všetky podmienky a žiadne vrcholy z r do neho pridávať nemôžeme). Pravého syna koreňa nového stromu postavíme rekurzívne, zlúčením pôvodného pravého syna l a stromu r, pomocou operácie $Merge(l.right, r)$.

Môžeme si povšimnúť že podmienka minimovej haldy bude zachovaná, lebo v stromoch r aj $l.right$ sa nachádzajú len vrcholy s väčšou prioritou než má l. Poradie BVS bude taktiež platné, pretože vrcholy v oboch stromoch ktoré zlúčime do pravého syna l majú väčšie kľúče než l.

- Priorita l je väčšia než priorita r. Tento prípad je analogický k prvému, koreňom zlúčeného stromu bude r, jeho pravý syn zostane nezmenený, a za ľavého syna dosadíme výsledok $Merge(l, r.left)$.
- Treap l alebo r je prázdny, prípadne sú prázdne oba (nastane napr. po zavolaní *Merge* na neexistujúceho syna). Vtedy vrátime ten Treap, ktorý nie je prázdny, prípadne prázdny Treap ak sú prázdne oba.

Podobne ako u $Split$u môžeme konštatovať, že *Merge* má lineárnu časovú zložitosť vzhľadom k hĺbke stromu, čo je v priemernom prípade $\mathcal{O}(\log n)$.

Realizácia operácií *Insert*, *Delete* **a** *Find*

Všetky tri operácie implentujeme podobným spôsobom ako v nevyvažovanom BVS, musíme si ale pomôcť vyššie uvedenými primitívnymi funkciami aby sme zachovali haldovú podmienku v strome.

U operácie *Insert* najprv nainicializujeme nový vrchol x s vkladaným kľúčom a náhodnou prioritou. Následne zostupujeme rekurzívne stromom vždy do ľavého alebo právého listu podľa toho, či je vkladaný kľúč menší alebo väčší než kľúč vrchola, na ktorom sa nachádzame (ďalej ho označíme ako v).

Zostup zastavíme v momente, keď je priorita nového vrchola menšia, než priorita v. Vtedy je z hľadiska haldovej podmienky vhodné umiestniť nový prvok na aktuálnu pozíciu, rozdelíme teda podstrom s koreňom v pomocou operácie $Split(x.key, v)$, čím získame dva stromy l a r. Všetky vrcholy týchto stromov majú vyššiu prioritu než x a menší resp. väčší kľuč ako x. Môžeme teda vložiť x na miesto kde sa v strome nachádzal vrchol v a ako jeho ľavého resp. pravého syna použiť stromy l a r.

Ak náhodou prídeme zostupom do neexistujúceho vrchola, vložíme x na jeho miesto.

Z hľadiska časovej zložitosti sa opäť opakovane zanorujeme do hlbších vrcholov do momentu, kým nevložíme prvok a nezavoláme $Split$. Zložitosť $Splitu$ aj zostupu je v priemernom prípade $\mathcal{O}(\log n)$.

Operácia *Delete* taktiež rekurzívne zostupuje stromom, a podobne ako v normálnom BVS hľadá pozíciu mazaného vrchola x. Keď je tento vrchol nájdený, po jeho zmazaní ho nemôžeme len nahradiť jedným z jeho potomkov, nakoľko by bola porušená haldová podmienka. Môžeme ale využiť operáciu $Merge(x.left, x.right)$ (všetky podmienky pre *Merge* sú splnené) a výsledný strom umiestniť na pôvodnú pozíciu x (ktorý následne je zmazaný).

Zostup aj *Merge* majú opäť v priemernom prípade časovú zložitosť $\mathcal{O}(\log n)$.

Nakoniec, *Find* je nad kľúčmi možné implementovať úplne zhodne ako v nevyvažovanom BVS.

Implementačné detaily

Teoreticky sme ukázali, ako realizovať Treap s funkcionalitou vyvažovaného BVS pri časovej zložitosti všetkých operácií $\mathcal{O}(\log n)$ v priemernom prípade ($\mathcal{O}(n)$ v najhoršom).

Reálna implementácia sa mierne rozchádza s teóriou v priraďovaní náhodných priorít, u ktorých sa rátalo s rovnomerným rozložením v nejakom spojitom rozsahu (napríklad $[0, 1]$). V praxi sú ale spojité náhodné veličiny problematické, a preto sú miesto nich v implementácií použité celé čísla reprezentované pomocou 32 bitov, ktoré je jednoduché pseudonáhodne generovať.

To samozrejme komplikuje dokazovanie vety 1, nakoľko môže s nenulovou pravdepodobnosťou nastať situácia, kedy budú vygenerované dve rovnaké priority. Reálne ale tento prípad pri rozumnom počte prvkov v strome nastáva zriedkavo a nemá veľký dopad na rýchlosť operácií. Pri obrovských objemoch ukládaných dát je žiadúce zvýšiť rozsah premennej určujúcej prioritu na 64 bitov.

Je tiež vhodné dodať, že pamäťová zložitosť Treapu je bez ohľadu na použitý dátový typ pre prioritu $\mathcal{O}(n)$ – každý prvok je reprezentovaný jedným vrcholom, a ten uchováva konštatné množstvo informácií: ľavého a pravého syna, kľúč a prioritu.

Implicit Treap

Implicit Treap (ďalej len IT) je dátová štruktúra využívajúca myšlienku náhodných priorít v Treape k realizácií ADT, ktorý reprezentuje dynamické pole s možnosťou rýchleho vkladania a mazania na ľubovoľnej pozícií (v podstate to isté, čo vykonávajú bez vyvažovania už spomínané kartézske stromy). Z hľadiska metód a ich zložitostí je podobná štruktúre Rope [4], ktorá používa odlišný spôsob vyvažovania stromu [2].

Naša implementácia sa opiera o špecifikáciu určenú pre kompetitívne programovanie [5], so zameraním na základnú funkcionalitu ktorú poskytuje Rope. Ďalšie možné rozšírenia zahŕňajú intervalové dotazy a modifikácie (s veľmi podobnou logikou ako v intervalových stromoch), množinové operácie (prienik, zjednotenie), obracanie poradia prvkov na ľubovoľnom rozsahu a mnoho ďalších.

Princíp a operácie

IT je v princípe Treap, ktorý reprezentuje pole o n prvkoch tak, že originálne pole je možné dostať pomocou in-order priechodu daného stromu. Tento koncept v kombinácií s vyvažovaním podľa logiky Treapu umožňuje vykonávať rýchly (priemerne logaritmický) náhodný prístup k ľubovoľnému prvku.

Vďaka tomu, že takto zoradený strom zodpovedá BVS s indexmi v poli ako kľúčmi vrcholov, ponúka sa jednoduchá myšlienka realizovať tak aj našu štruktúru. Keby sme ale indexy ukladali do vrcholov explicitne, znemožnilo by to rýchle náhodné vkladanie alebo mazanie, nakoľko bolo pri každej operácií potrebné prečíslovať až $\mathcal{O}(n)$ indexov.

Hlavným princípom IT je tým pádom ukladať indexy len *implicitne*, čo v tomto prípade znamená, že ich budeme počítať z veľkostí podstromov. Udržiavať počet

[2] Presné merania v rámci tejto práce neuvádzame, ale na klasických úlohách ako napríklad *UVA 10909 - Lucky Numbers* dosahoval IT zhruba o tretinu lepšie časy než Rope implementovaný v SGI rozšírení STL.

vrcholov v podstrome každého prvku nie je problematické, nakoľko pri každej modifikácií stačí zmenu počtu prvkov propagovať do všetkých predkov ovplyvneného vrchola (ktorých je v Treape priemerne $\mathcal{O}(\log n)$).

Pamäťová zložitosť IT bude rovnako ako u Treapu $\mathcal{O}(n)$, každý prvok v poli bude zastúpený jedným vrcholom a ten v sebe uchováva konštatný počet premenných: ľavého a pravého syna, hodnotu, prioritu a po novom veľkosť podstromu.

Poďme sa detailne pozrieť na podporované operácie:

- $Insert(x, p)$ – Pridá do poľa prvok s hodnotou x na pozícií p (všetkým prvkom na pozíciach $p, p+1, \ldots, n-1$ v pôvodnom poli sa zvýši index o 1).
- $Delete(p)$ – Zmaže prvok na pozícií p z poľa (všetkým prvkom na pozíciach $p+1, \ldots, n-1$ v pôvodnom poli sa zníži index o 1).
- $At(p)$ – Vráti hodnotu prvku na pozícií p.

Ako pri Treape, aj v tomto prípade si pomôžeme primitívnymi operáciami $Split$ a *Merge*, ktorých funkčnosť bude mierne modifikovaná:

- $Split(p, t)$ – Rozdelí pole špecifikované IT t s indexmi $0, \ldots, n-1$ na dva IT l a r, reprezentujúce polia $t[0, \ldots, p]$ a $t[p+1, \ldots, n-1]$. Pôvodný IT t zanikne.
- $Merge(l, r)$ – Spojí dva IT l a r do jedného tak, že pole reprezentované výsledným IT bude zodpovedať konkatenácií polí z l a r.

Realizácia $Split$ **a** *Merge*

$Split(p, t)$ je možné implementovať veľmi podobne ako u normálneho Treapu – hlavnou zmenou je, že miesto porovnávania kľúčov si $Split$ bude voliť či sa zanorí doľava alebo doprava podľa súčasnej pozície.

Výpočet pozície, na ktorej sa v strome nachádzame je možné vyjadriť ako

$$pos(v) = size(v.left) + \sum_{x \in ar(v)} size(x.left)$$

kde pre vrchol v je $size(v)$ veľkosť podstromu s koreňom vo v a $ar(v)$ je množina takých predkov v, že v leží v ich pravom podstrome.

Počet vrcholov je v ľavom podstrome priamo uložený, a v ľavých podstromoch predkov ho môžeme napočítavať počas rekurzívneho zostupu a posielať nižšie cez argument. Logika delenia je potom podľa podmienky $pos > p$ úplne rovnaká ako pri $Split$e v Treape. Po vykonaní rozdelenia je ešte potrebné na vrchole ktorému sme menili synov (koreň ľavého alebo pravého výsledku delenia) prepočítať veľkosť jeho podstromu.

Merge je k verzií z Treapu ešte podobnejší, jediná zmena sa týka prepočítavania veľkosti podstromu, ktorú je nutné vykonať na vzniknutom koreni nového stromu (po zavolaní rekurzívnych volaní *Merge* ju stačí spočítať ako súčet veľkosti ľavého a pravého podstromu plus 1).

Analýza časovej zložitosti oboch metód je úplne zhodná s tou v Treape, všetky pridané kroky sú konštantné.

Realizácia *Insert*, *Delete* **a** *At*

Narozdiel od neimplicitného Treapu vkladanie a mazanie v IT samo o sebe neprechádza strom, a dá sa napísať čisto pomocou primitívnych operácií. V nasledujúcich popisoch predpokladáme, že sú funkcie zavolané na IT t obsahujúci prvky s indexami $0, 1, \ldots, n-1$.

Pri volaní $Insert(x, p)$ môžeme najprv rozpojiť pole na dve časti v rozsahoch $0, \ldots, p-1$ a $p, \ldots, n-1$ pomocou $Split$u. Do takto jednej z týchto častí stromov je ľahké pripojiť vrchol obsahujúci hodnotu x, napríklad volaním $lx = Merge(l, x)$, kde l je ľavá rozdelená časť pôvodného poľa. Následne už stačí len znovu spojiť oba stromy pomocou $Merge(lx, r)$ a dostaneme nové pole s rozsahom $0, \ldots, n$ s prvkom x na pozícií p.

Časová zložitosť pozostáva z konštantného počtu volaní $Split$ a $Merge$, ktoré v priemernom prípade majú zložitosť $\mathcal{O}(\log n)$.

Delete sa dá implementovať obdobne, opäť môžeme rozpojiť originálne pole pomocou $Split(p-1, t)$ na časti l a r. Následne ale chceme zmazať prvok na originálnej pozícií p, ktorý sa teraz nachádza na pozícií 0 v r. Môžeme teda znova deliť r pomocou $Split(0, r)$ a oddeliť tak samostatný vrchol ktorý chceme mazať. Časť l sa potom dá naspäť zlúčiť so zvyškom r a vznikne pole s rozsahom $0, \ldots, n-2$, v ktorom chýba p-tý prvok.

Analýza časovej zložitosti je analogická ako u $Insert$u.

Nakoniec, At je opäť možné realizovať rekurzívnym zostupom podobným ako u BVS, akurát sa po strome pohybujeme na základe pozície, ktorú môžeme počítať rovnako ako v metóde $Split$.

Poznámky

Priložená implementácia je napísaná v jazyku C++, v podobe header-only knižnice pod menom `treap.hpp`. Všetky implementované metódy sú testované v programe `tests.cpp`, spustenie testov je možné príkazom `make test` (vyžaduje program GNU Make).

Kódy implementácie a tohto dokumentu je tiež možné stiahnúť z repozitára na adrese `https://github.com/PMitura/implitreaps`.

Literatúra

[1] J. Vuillemin, „A unifying look at data structures“, Communications of the ACM, vol. 23, no. 4. Association for Computing Machinery (ACM), s. 229–239, 1. apríla 1980.

[2] C. R. Aragon a R. G. Seidel, „Randomized search trees“, 30th Annual Symposium on Foundations of Computer Science. IEEE, 1989.

[3] J. Erickson, „Randomized Binary Search Trees“, 2013. [Online]. Dostupné z: `http://jeffe.cs.illinois.edu/teaching/algorithms/notes/10-treaps.pdf`. [prístup 9. mája 2017].

[4] H.-J. Boehm, R. Atkinson, a M. Plass, “Ropes: An alternative to strings”, Software: Practice and Experience, roč. 25, č. 12. Wiley-Blackwell, s. 1315–1330, december 1995.

[5] e-maxx.ru, "Treap (treap, deramida)", 2008. [Online]. Dostupné z: `http://e-maxx.ru/algo/treap`. [prístup 13. mája 2017].