Finger search tree

Notatka do wykładu

Rafał Sławik

11 grudnia 2012

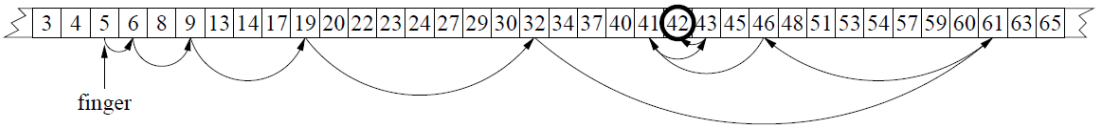
Spis treści

1 Wstęp

Bardzo często spotykanym w informatyce problemem jest zarządzanie uporządkowanym zbiorem elementów, w taki sposób, aby umożliwić *szybkie* sprawdzanie czy dany element e należy do tego zbioru. Problem możemy rozważać w dwóch wersjach:

- statycznej, gdzie zbiór elementów jest ustalony,

Rysunek 1: Wyszukiwanie wykładnicze na tablicy (rysunek z [4])



- dynamicznej, gdzie pozwalamy na operacja dodawania i usuwania elementów ze zbioru.

W przypadku statycznym możemy umieścić elementy w kolejności w tablicy i zastosować wyszukiwanie binarne do sprawdzania czy element e należy do zbioru. Z kolei dla wersji dynamicznej możemy skorzystać z drzew BST. W obu tych przypadkach zużyjemy liniowo dużo pamięci na przechowanie zbioru, a czas wyszukiwania będzie logarytmiczny względem liczby elementów.

Czasem taka złożoność wyszukiwania jest niewystarczająca i potrzebujemy wydajniejszej struktury danych. Taką strukturą są na przykład **finger search trees**, które rozważamy tutaj.

1.1 Finger - wskaźnik, czas $O(\log d)$

Dodajemy do struktury wskaźniki (ang. fingers), które umożliwią nam *szybkie* uaktualnienia i wyszukiwanie w ich pobliżu. O ich liczbie i rozmieszczeniu decyduje osoba korzystająca ze struktury. Na niej także spoczywa obowiązek zapewnienia, że wstawianie nowej wartości odbywa się we właściwym miejscu.

Zastosowanie fingerów pozwala na zredukowanie czasu wyszukiwania z $O(\log n)$, gdzie n oznacza liczbę wszystkich elementów w zbiorze, do $O(\log d)$, gdzie d jest odległością pomiędzy fingerem f (od którego rozpoczynamy szukanie) i poszukiwanym elementem e.

Jakie korzyści daje użycie FST można przeczytać w części 4.

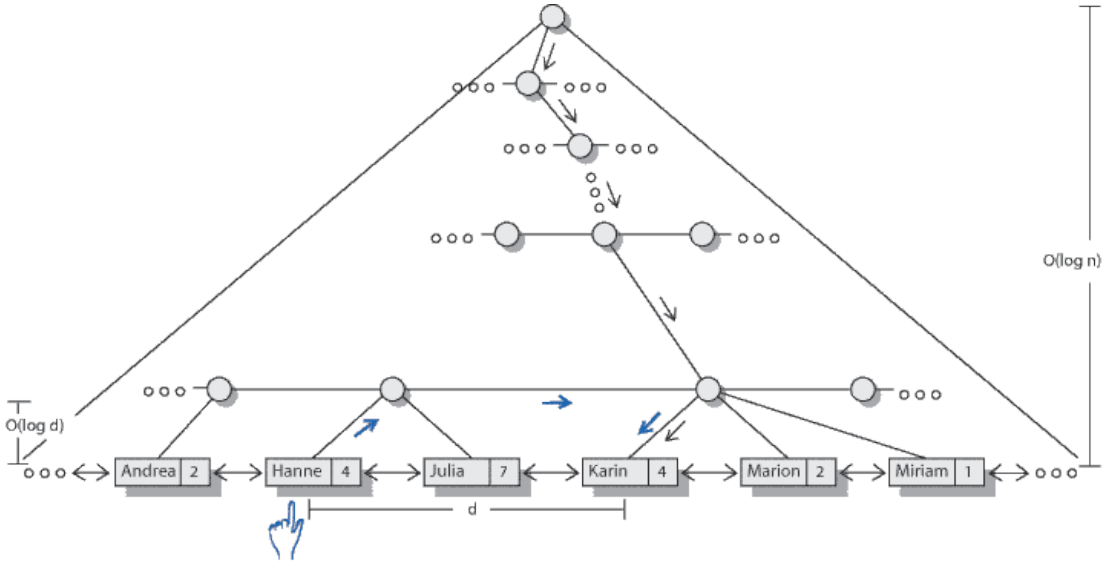
1.2 Idea finger search

Cała idea finger search sprowadza się do tego, że chcemy uniknąć przeszukiwania całej struktury. Dzięki temu, że dysponujemy fingerem, który potencjalnie znajduje się blisko elementu, którego szukamy, możemy wyszukiwanie wykonać *lokalnie*. Patrząc na rysunki struktur, możemy dostrzec jak *naturalny* dla człowieka jest to proces.

1.2.1 Na tablicy

Spójrzmy jak można zastosować ten pomysł do zbioru przechowywanego w tablicy (rysunek 1). Wykorzystujemy wyszukiwanie wykładnicze. Badamy kolejne przedziały o długościach $2^i - 1$ dla rosnących wartości i. Gdy znajdziemy już przedział, do którego należy szukany element, wykonujemy na nim poszukiwanie binarne. Osiągamy w ten sposób czas $O(\log d)$, ponieważ z każdym kolejnym skokiem pomijamy wykładniczo wiele elementów (argument taki sam jak w 3.4.1).

Rysunek 2: Finger search na drzewie (rysunek ze strony http://leda-tutorial.org)



1.2.2 Na drzewie

W przypadku drzew proces wyszukiwania wygląda następująco (rysunek 2). Niebieskie strzałki pokazują jaką drogę przebywami wykonując finger search, czarne drogę przebytą w czasie zwykłego wyszukiwania. W części 3.4.1 pokażemy, że droga złożona z niebieskich strzałek ma długość $O(\log d)$.

1.3 Operacje na FST

Na omawianej strukturze wykonywać będziemy następujemy operacje:

- `Search`(f, x)
- `Insert`(f, x)
- `Delete`(f, x)

Oczekujemy, aby każda z nich zajmowała w pesymistycznym przypadku $O(\log d)$. Operacje `Insert` i `Delete` możemy podzielić na dwie części:

1. wyszukiwanie miejsca, w którym należy wstawić/usunąć element - $O(\log d)$,
2. dokonanie zmiany - $O(1)$.

2 Historia

W tablicy 1 przedstawiona jest skrócona historia FST. (*Uwaga: czasy podane przy operacjach uaktualnienia są podane przy założeniu, że miejsce zmiany jest już ustalone*).

Struktura	`Insert`	`Delete`	`Search`	Uwagi
'97 Guibas	$O(1)$	$O(1)$	$O(\log d)$	B-drzewo, stała liczba wskaźników
'88 Levocopolous, Overmars	$O(1)$	$O(1)$	$O(\log n)$	Dowolna liczba wskaźników
'79 Harel, '96 Fleisher, '03 Brodal	$O(1)$	$O(\log^* n)$	$O(\log d)$	Dowolna liczba wskaźników
'03 Brodal i in.	$O(1)$	$O(1)$	$O(\log d)$	„Optimal finger search trees in the pointer machine"
'94 Dietz, Raman	$O(1)$	$O(1)$	$O(\log d)$	
'00 Anderson, Thorup	$O(1)$	$O(1)$	$O\left(\sqrt{\frac{\log d}{\log\log d}}\right)$	

Tablica 1: Historia

Poza wymienionymi pracami należy wspomnieć także o *(2,4)-linked tree* (patrz: część 3.4), które umożliwia uaktualnienia w czasie stałym (zamortyzowanym) i wyszukiwanie w czasie $O(\log d)$. Struktura ta jest używana w praktyce.

3 Struktura

3.1 Model

Modelem, w którym rozważamy strukturę, jest **model maszyny RAM**. Zakładamy tutaj, że mamy do dyspozycji słowa maszynowe długości $\omega = c\log_2 M$, gdzie wartość M oznacza maksymalny rozmiar struktury. Cała struktura musi zmieścić się w pamięci, więc słowa muszą umożliwiać jej adresowanie. Zakładamy również, że wszystkie operacje na słowach wykonywane są w czasie stałym.

3.2 Idea

Struktura składa się z dwóch części:

- struktury na górnym poziomie - drzewa,
- kubełków, których reprezentanci są przechowywani w strukturze na górnym poziomie.

Mamy pewną dowolność przy wyborze sposobu przechowywania elementów w kubełku, jak również przy wyborze rodzaju drzewa. Drzewa umożliwiają uaktualnienia w czasie logarytmicznym, dlatego tutaj będziemy je realizować *przyrostowo*, to znaczy przy każdej operacji na strukturze wykonamy pewną stałą ilość kroków procesu naprawiania struktury na górnym poziomie. Pokażemy, że konieczność takich uaktualnień jest na tyle rzadka, że jesteśmy w stanie taki proces pomyślnie wykonać.

Opisujemy tu strukturę zaproponowaną przez Dietza i Ramana w [1]. Jest to poprawiona wersja struktury z pracy [2]. Wyeliminowana została konieczność *globalnego przebudowania*, gdy liczba elementów w strukturze sięgała granicznym wartościom. Wymagało ono uruchomienia działacego przryostowo procesu, który przebudowywał strukturę dla większej/mniejszej liczby elementów. Po przekroczeniu granicznej wartości struktura była zastępowana przygotowaną kopią.

3.3 Kubełki

Kubełki będą miały rozmiar $O(\log^2 n)$.
Oznaczmy przez $\hat{n}$ maksymalną liczbę elementów w strukturze do tej pory. Do opisu kubełków definiujemy dodatkowo następuję wartości:

$$\phi(b) = \frac{\text{size}(b)}{\log^2 \hat{n}} \tag{1}$$

$$\rho(b, \hat{n}) = \frac{1}{\log^2 \hat{n}} \max\left(0, 0.7 \log^2 \hat{n} - \text{size}(b), \text{size}(b) - 1.8 \log^2 \hat{n}\right) \tag{2}$$

Wartość 1 nazywać będziemy zapełnieniem kubełka, a wartość 2 krytycznością. Gdy krytyczność danego kubełka będzie większa od 0 powiemy, że ten kubełek jest krytyczny.

Będziemy dbać, aby $0.5 \leq \phi(b) \leq 2$. Oznacza to, że kubełek może maksymalnie podwoić swoją wielkość albo dwukrotnie ją zmniejszyć. Gdy krytyczność kubełka będzie wysoka, będziemy wykonywać na nim operację `Split`, `Merge`, `Transfer`. Z kolei na podstawie lematu mamy gwarancję, że zdarzenia te będą na tyle rzadkie, że proces przyrostowego wstawiania/usuwania reprezentanta w górnej strukturze zdąży zakończyć swoją pracę nim pojawi się konieczność następnej aktualizacji.

3.3.1 Lemat o zapełnieniu kubełków

Poniższy lemat gwarantuje, że krytyczność kubełki nigdy nie będzie za duża. Lemat pochodzi z [3].

Lemat 1. *Niech $x_1, x_2, \ldots, x_n$ będą liczbami rzeczywistymi (wszystkie początkowo równe 0).*
Powtarzamy następujące czynności:

1. *wybierz x_i, takie że $x_i = \max_j x_j$*
2. *ustaw x_i na 0*
3. *wybierz n nieujemnych liczb $a_1, a_2, \ldots, a_n$, takich że $\sum_{j=1}^{n} a_j = 1$*
4. *ustaw x_j na $x_j + a_j$ (dla każdego j).*

Wtedy każdy x_k jest zawsze mniejszy niż $H_{n-1} + 1$ (H_k oznacza k-tą liczbę harmoniczną).

Dowód. **Obserwacja**: W każdym momencie suma wszystkich x_j jest nie większa niż n: $S = \sum_{j=1}^{n} x_j \leq n$. Jest to prawdą na samym początku, gdy wszystki $x_j = 0$. W czasie jednej iteracji procesu opisanego w lemacie wartość S wzrasta do $\frac{n-1}{n} S + 1$. Składnik $\frac{n-1}{n} S$ wynika z tego, że chcemy, aby po wykonaniu iteracji suma pozostała jak największa, dlatego rozkładamy *masę* po równo na wszystkich x_j (wtedy usunięcie maksimum usuwa najmniej). Z kolei 1 to suma dodanych w tej iteracji a_j. Z założenia indukcyjnego $S \leq n$, zatem $\frac{n-1}{n} S + 1 \leq \frac{n-1}{n} n + 1 = n$.

Definiujemy ciąg wektorów $x^{(1)}, x^{(2)}, \ldots, x^{(m)}$, opisujących stany wartości x_j po kolejnych iteracjach. Bez straty ogólności zakładamy, że $m \geq n$. Takie założenie jest dopuszczalne, ponieważ możemy dołożyć dowolnie dużo wektorów zerowych na początku ciągu. Zerowe wektory oznaczają iteracje, w których tylko jedno a_j było równe 1. Wybierzmy również ciąg permutacji $\pi_1, \pi_2, \ldots, \pi_n$. Będą to permutacje liczb $1, 2, \ldots, n$ mające następującą własność:

$$x^{(m-k+1)}_{\pi_k(i)} \geq x^{(m-k+1)}_{\pi_k(i+1)}$$

Permutacja π_k mówi jaka jest kolejność kubełków w porządku malejącym w $(m-k+1)$-tej (czyli k-tej od końca) iteracji. Zdefiniujmy wartość:

$$S_k = \sum_{j=1}^{k} x^{(m-k+1)}_{\pi_k(j)}$$

oznaczającą sumę k największych elementów w k-tej od końca iteracji. S_1 to najcięższy element w ostatniej iteracji. Podobnie jak w obserwacji, zauważamy, że $S_k \leq \frac{k}{k+1} S_{k+1} + 1$. Wystarczy w $k+1$-szej od końca iteracji rozłożyć *masę* pomiędzy $k+1$ najcięższych elementów. Stąd wynika, że:

$$S_1 \leq \frac{1}{2} S_2 + 1 \leq \frac{1}{2}\left(\frac{2}{3} S_3 + 1\right) + 1 = \frac{1}{3} S_3 + \frac{1}{2} + 1 \leq \cdots \leq \frac{1}{k} S_k + H_{k-1} \leq$$
$$\leq \cdots \leq \frac{1}{n} S_n + H_{n-1} \leq 1 + H_{n-1}$$

Ostatnia nierówność wynika z obserwacji. Natomiast z dowolności wyboru m wynika teza lematu. □

Interpretacja i wykorzystanie lematu Zastosowanie lematu w naszym przypadku wymaga odniesienia opisanego w nim procesu do akcji na kubełkach. Wartości $x_1, x_2, \ldots, x_n$ oznaczają przeskalowane *krytyczności* kubełków. Wybór maksimum i jego wyzerowanie to podzielenie (połaczenie) najbardziej krytycznego kubełka - ten który był krytyczny przestaje nim być, a także te których ta akcja dotyczyła (powstały dwa nowe kubełki) nie staną się krytyczne. Ustawienie któregoś a_j na 1 oznacza, zwiększenie krytyczności w j-tym kubełku.

My jednak akcję dzielienia najbardziej krytycznego kubełka (usuwania maksimum) wykonujemy co $k = c \log \hat{n}$ kroków. Więc możemy sobie wyobrazić, że zamiast dodawać $\sum a_i = 1$, gdzie rozmiar każdego kubełka moglibyśmy ograniczyć przez $H_{n-1} + 1 = O(\log \hat{n})$, możemy dodawać k razy więcej i w konsekwencji dostać k razy większe ograniczenie końcowe, które wynosi $O(\log^2 \hat{n})$.

3.3.2 Zarządanie kubełkami

Kubełkami będziemy zarządzać w następujący sposób. Co $O(\log \hat{n})$ operacji na strukturze wybierzemy najbardziej krytyczny kubełek b i wykonamy na nim jedną z operacji:

- `Split`, gdy $\phi(b) > 1,8$. Dzielimy kubełek na dwa podobnej wielkości i dodejmy nowego reprezentanta,
- `Transfer`, gdy $\phi(b) < 0,7$ oraz sąsiad b' ma $\phi(b') > 1$. Wyrównujemy wielkości kubełków b i b',
- `Fuse`, gdy $\phi(b) < 0,7$ i transfer nie jest możliwy. Usuwamy kubełek b, a elementy z niego przenosimy do sąsiada b'.

Na krytyczność kubełków mają wpływ uaktualnie na strukturze. Jeśli nie zmieniają wartości $\hat{n}$, to do zapewnienia właściwej wielkości kubełków wystarczy lemat. Natomiast w przypadku wzrostu $\hat{n}$ potrzebne jest dodatkowe oszacowanie.

Popatrzmy o ile wzrasta krytyczność jednego kubełka przy wzroście $\hat{n}$ o 1. Niech δ_b oznacza przyrost w kubełku b.

$$\delta_b \leq \frac{d\rho(b, \hat{n})}{d\hat{n}} \leq \frac{c}{\hat{n}} \tag{3}$$

gdzie c jest pewną stałą. Cały przyrost wyraża się wzorem:

$$\sum_{b \text{ - kubełek}} \delta_b \leq \sum_{b \text{ - kubełek}} \frac{c}{\hat{n}} = \frac{\hat{n}}{\log^2 \hat{n}} \cdot \frac{c}{\hat{n}} = O\left(\frac{1}{\log^2 \hat{n}}\right) \tag{4}$$

Pomiędzy przebalansowaniami wartość $\hat{n}$ może wzrosnąć co najwyżej $O(\log \hat{n})$ razy. Zatem całkowity przyrost wyniesie co najwyżej $O(\log^{-1} \hat{n})$. Stosując lemat, widzimy, że górne ograniczenie na krytyczność dowolnego kubełka to $kH_{\hat{n}-1} + O(1)$, a to pozwala na dobranie odpowiedniej (stałej) liczby operacji procesu przyrostowego uaktualniania wykonywanych przy każdej operacji na strukturze.

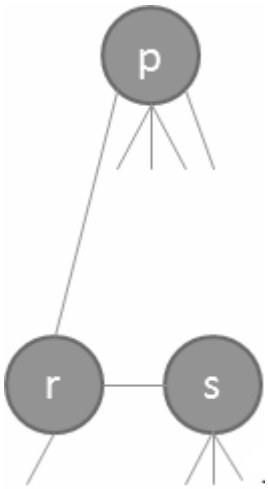
3.3.3 Struktura wewnętrzna

Każdy kubełek reprezentujemy jako drzewo o wysokości 3 o współczynniku rozgałęzienia $\beta = \theta(\log^{2/3} \hat{n})$. Przy uaktualnieniach będziemy dbać, aby ten współczynnik znajdował się pomiędzy $\beta/2$ a 2β. W tym celu zezwalamy na to, aby wierzchołki mogły posiadać dodatkowy wierzchołek (tak jak na rysunku 3). Reguły balansowania są dokładniej opisane w oryginalnej pracy [1] - tutaj podamy wyłącznie krótki ich opis.

Przy dodawaniu elementu, gdy r ma więcej niż 2β dzieci, tworzony jest dodatkowy wierzchołek s podłączony wyłącznie do r (nie do p). Z kolei przy usuwaniu, gdy mamy mniej niż $\beta/2$ dzieci, próbujemy zabrać dzieci sąsiadowi, a jeśli jest to niemożliwe to sami podczepiamy się jako dodatkowy wierzchołek. Balansowanie ma na celu zachowanienie odpowiedniego współczynika rozgałęzienia β oraz nierówności $\text{nchild}(s) \leq \text{nchild}(r)$, która mówi, że r ma więcej dzieci niż s.

Konieczne są dodatkowe operacje, aby nierówności pozostawały spełnione. Operacje na kubełku mogą zaburzyć te nierówności o $O(1)$, o ile nie zmieniają wartości $\hat{n}$. Z kolei zmiana $\hat{n}$ może spowodować zaburzenia we wszystkich kubełkach. Aby je naprawić, uruchomiamy proces, który będzie *inkrementacyjnie* przesuwać wierzchołki z s do r lub na odwrót. Naprawa pojedynczego węzła to koszt stały. Nim proces powróci do danego

Rysunek 3: Dodatkowy wierzchołek s w drzewie reprezentującym kubełek



wierzchołka to wartość $\hat{n}$ może zmienić się maksymalnie o czynnik $1 + o(1)$ (wynika to z tego, że $\log^{2/3}(2\hat{n}) - \log^{2/3}\hat{n} < c$, ponieważ $\lim_{\hat{n}\to\infty} \log^{2/3}(2\hat{n}) - \log^{2/3}\hat{n} = 0$).

Ponadto każdy wewnętrzny wierzchołek drzewa przechowuje zbiór S maksymalnych elementów swoich poddrzew za pomocą:

- tablicy $A[1..k]$ przechowująca wskaźniki na elementy
- tablicy $Perm[1..k]$ reprezentująca permutację elementów z A

$$\text{Perm}[i] = \begin{cases} 0 & A[i] = null \\ |\{\text{key}(A[j]) : j \leq i \wedge \text{key}(A[j]) \leq \text{key}(A[i])\}| & \text{wpp.} \end{cases}$$

- podwójnie wiązanej listy L, na której będą w istocie przechowywane elementy (ponadto każdy element wie jaki jest jego indeks w tablicy A)

Na zbiorze S będziemy wykonywać operacje wstawiania i usuwania. Chcemy, aby każda z nich działała w czasie stałym. Poniżej znajduje się opis ich realizacji.

- `Insert` - dodawanie elemenentu do zbioru maksimów wygląda w następujący sposób:
  1. Ustalamy j, takie że $A[j]$ wskazuje na klucz, za którym będzie wstawiony nowy element,
  2. Dołączamy element do listy,
  3. Ustawiamy $A[|S| + 1]$, żeby wskazywało na ten element,
  4. Uaktualniamy $Perm$

- `Delete` - usuwanie elementu ze zbioru maksimów:
  1. Ustalamy j, takie że $A[j]$ wskazuje na klucz do usunięcia,
  2. Zamieniamy $A[j]$ z $A[|S|]$,
  3. Ustawiamy $A[|S|]$ na $null$,
  4. Usuwamy element z listy,
  5. Poprawiamy element wskazywany przez $A[j]$, żeby wskazywał na j (bo aktualnie wzkazuje na $|S|$),
  6. Uaktualniamy $Perm$

Wstawianie i usuwanie bardzo łatwo zrealizować w czasie $O(|S|)$. My chcemy jednak przyspieszyć te operacje. Zauważamy, że do zapisania w pamięci tablicy $Perm$ potrzebujemy $|S| \cdot \lceil \log |S| \rceil$ bitów (ponieważ w $Perm$ znajdują się liczby z $\{0, \ldots, |S|\}$, a do zapisu każdej z nich wystarczy nam $\log |S|$ bitów).

Pokażemy, że ta potrafimy $Perm$ trzymać w pojedynczym słowie maszynowym. Maksymalny rozmiar zbioru maksimów S to $2\log^{2/3} \hat{n}$. Oczywiście $\hat{n}$ nie przekroczy maksymalnego rozmiaru struktury M. Zatem $|S| \leq 2\log^{2/3} \hat{n} \leq 2\log^{2/3} \hat{M}$. Udowodnimy najpierw nierówność $|S| \leq \frac{\log M}{\log\log M}$.

$$|S| \leq 2\log^{2/3} M \leq \frac{\log M}{\log\log M} \tag{5}$$

$$2\log\log M \leq \log^{1/3} M = \sqrt[3]{\log M}$$

Dla odpowiednio dużych wartości M ta nierówność zachodzi. Intuicyjnie funkcja $\sqrt[3]{\cdot}$ rośnie szybciej niż log. Pokażemy teraz, korzystając z nierówności 5, że $|S| \cdot \lceil \log |S| \rceil \leq \log M$.

$$|S| \cdot \lceil \log |S| \rceil \leq \frac{\log M}{\log\log M} \cdot \log \frac{\log M}{\log\log M} = \frac{\log M}{\log\log M} \cdot \left(\log\log M - \log^{(3)} M \right) \leq \log M \tag{6}$$

Dzięki temu możemy wykorzystać kilka funkcji pomocniczych i stablicować ich wyniki, a w czasie operacji na strukturze jedynie pobierać z tablicy ich wynik.

- `GetRank`$(Perm, i)$ - zwraca pozycję ma element i-ty w zbiorze S w porządku rosnącym,
- `GetIndex`$(Perm, i)$ - zwraca indeks w tablicy A elementy, który znajduje się na i-tej pozycji w zbiorze S w porządku rosnącym,
- `UpdatePerm`$(Perm, rank, operation)$ - uaktualnia tablicę $Perm$, $operation$ mówi czy dodajemy, czy usuwamy element, a $rank$ wskazuje ten element

Należy jeszcze sprawdzić czy te tablice nie zajmą za dużo miejsca. Przykładowo dla funkcji `UpdatePerm` możliwych różnych wartości argumentów jest $2|S| \cdot 2^{|S|\lceil \log |S| \rceil}$. Pokażemy teraz, że jest to $O(M)$. Korzystamy z udowodnionej nierówności 5, która ogranicza

rozmiar S. Najpierw przekształcamy $2|S| \cdot 2^{|S|\lceil \log |S| \rceil}$ do postaci $2|S|^{|S|+1}$. Dowodzimy nierówności w notacji O, dlatego możemy pominąć stałą 2. Ostatecznie pokazywać będziemy, że dla $c \leq 1$ zachodzi:

$$|S|^{|S|+1} = M^c$$
$$(|S|+1)\log|S| = c \cdot \log M$$

Druga równość powstaje w wyniki zlogarytmowania pierwszej. Po wstawieniu oszacowania 5 mamy:

$$\left(\frac{\log M}{\log\log M}+1\right)\cdot(\log\log M - \log^{(3)} M) = \log M + \left[\log\log M - \log^{(3)} M \cdot \left(\frac{\log M}{\log\log M}+1\right)\right] \tag{7}$$

Wystarczy teraz pokazać, że składnik w nawiasie kwadratowym jest ujemny. Wprowadźmy najpierw oznaczenie $x = \log\log M$, wtedy to co chcemy udowonić przedstawia się następująco:

$$x \leq \log x \left(\frac{2^x}{x}+1\right)$$
$$x^2 \leq \log x(2^x + x)$$

Tutaj również intuicyjnie widać, że nierówność zachodzi, ponieważ funkcja wykładnicza rośnie dużo szybciej niż kwadratowa.

Zatem możliwe jest przygotowanie tablic wyników dla funkcji pomocniczych.

3.3.4 Wyszukiwanie

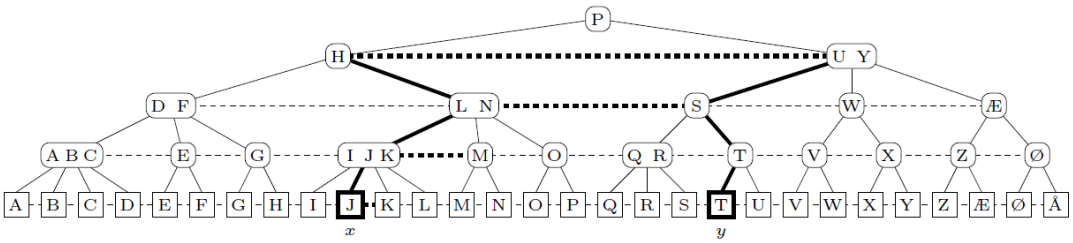
Wyszukiwanie wewnątrz kubełka rozpoczynamy od znalezienia LCA finger f i szukanego klucza s. Jako, że drzewo ma wysokość równą 3, możemy to zrobić w czasie stałym. Dalej szukamy odpowiedniego węzła w dół. Bez utraty ogólności załóżmy, że LCA f i s jest najwyższym wierzchołkiem (korzeniem). Oznaczmy przez d_1, d_2, d_3, w którym poddrzewie znajduje się s, odpowiednio pierwszego, drugiego i trzeciego wierzchołka na ścieżce od korzenia do s. Przy tych oznaczeniach odległość d między f a s wyraża się $d = d_3 + d_2 \log^{2/3} \hat{n} + d_1 \log^{4/3} \hat{n}$. Asymptotycznie największym składnikiem (względem $\hat{n}$) jest $d_1 \log^{4/3} \hat{n}$. Znalezienie odpowiedniego poddrzewa w wierzchołku zajmuje czas $\log d_i$, wykonujemy w tym celu wyszukiwanie wykładnicze (patrz: 1.2.1) na tablicy A, wykorzystując funkcję `GetIndex`. W sumie koszt wyszukania to $O(\log d_1) + O(\log d_2) + O(\log d_3) = O(\log d_1 d_2 d_3) \leq O(\log(d_1 \log^{4/3} \hat{n}))$, czyli $O(\log d)$.

3.4 Struktura na górnym poziomie

W zakresie wyboru struktury na górnym poziomie mamy dowolność, o ile zagwarantujemy, że uaktualnienia w niej będą wykonywane w czasie $O(\log n)$.

W tej notatce wybieramy *(2,4)-drzewo*. Charakteryzuje się ono tym, że wszystkie liście znajdują się na tej samej głębokości. Każdy wierzchołek wewnętrzny jest stopnia

Rysunek 4: Przykładowe (2,4)-drzewo z połączeniami między węzłami na tym samym poziomie



2, 3 lub 4. Klucze przechowywane są w liściach, a węzły wewnętrzne zawierają informację ułatwiająca wyszukiwanie. Przykładowe dzewo znajduje się na rysunku 4. Warto tu zauważyć, że (2,4)-drzewa to *prawie* to samo, co B-drzewa rzędu 4. Dokładny opis operacji na (2,4)-drzewach można znaleźć w wielu źródłach (np. w [5]), więc zostanie tu pominięty.

Dodatkowo łączymy węzły na tym samym poziomie między sobą. Wykorzystujemy do tego listę dwukierunkową. Na rysunku 4 te połączenia są zaznaczone liniami przerywanymi. Ich zastosowanie pozwala na wyszukiwanie w czasie $O(\log d)$, a ich aktualizacja nie zwiększa czasu uaktualnienień w drzewie.

3.4.1 Wyszukiwanie

Do opisu wyszukiwania wykorzystamy rysunek 4. Załóżmy, że mamy finger w miejscu x oraz szukamy elementu y.

Wyszkiwanie rozpoczynamy od ustalenia czy poszukiwany element y jest większy od x. Bez straty ogólności zakładamy, że tak, więc będziemy poszukiwać go w prawo. Następnie idziemy od x w stronę korzenia drzewa, aż ustalimy, że y znajduje się w poddrzewie aktualnie rozpatrywanego węzła v lub jego prawym sąsiedzie v'. Wtedy rozpoczynamy poszukiwanie w dół zarówno od wierzchołka v jak i v'.

Pokażemy teraz, że czas wyszukiwania to $O(\log d)$. Zauważamy, że robiąc krok w górę, pomijamy wykładniczo dużo elementów. Wynika to z tego, że każdy węzeł wewnętrzny ma co najmniej dwóch synów, a liście są na tym samym poziomie. Tak więc element o wysokości h ma 2^{h-1} kluczy pod sobą. Idąc w górę, dodajemy do siebie kolejne potęgi dwójki. Wiemy, że robiąc k kroków do górym pominiemy $\sum_{i=1}^{k} 2^{i-1} = 2^k - 1$, zatem skoro odległość pomiędzy x a y wynosiła d, to $k = O(\log d)$. Całe wyszukiwanie trwa więc $O(k) = O(\log d)$.

4 Zastosowania

FST mają swoje stasowowanie między innymi w:

- operacjach na zbiorach

- geometrii obliczeniowej:
  - Three-dimensional layers of maxima
  - Triangulating a simple polygon
- algorytmach tekstowych:
  - Finding repeats with gaps
  - Quasiperiodicities in strings

Zwykle używając finger search tree możemy zaoszczędzić czynnik $O(\log n)$ w porównaniu z drzewami BST.

4.1 Łączenie zbiorów

Niech X, Y - zbiory, $|X| = n$, $|Y| = m$. Bez utraty ogólności $n \leq m$. Rozważmy prosty algorytm:

for $x \in X$ **do**
  $Y \leftarrow Y \cup \{x\}$
end for

W przypadku przechowywania zbiorów X i Y za pomocą BST złożoność tego algorytmu wynosi $O(n \log m)$, n wstawień kosztujących po $\log m$.

Korzystając z FST, możemy przesuwać finger monotonicznie. Wtedy złożoność wyniesie $\sum_{i=1}^{n} O(\log d_i)$. Kolejne wartości d_i oznaczają o ile przesuwa się finger przy wstawianiu i-tego elementu, oczywiście $\sum_{i=1}^{n} = m$.

$$\sum_{i=1}^{n} O(\log d_i) = O\left(\sum_{i=1}^{n} \log d_i\right) \leq O\left(n \log \frac{m}{n}\right)$$

Oszacowanie to wynika wprost z nierówności Jensena zastosowanej dla logarytmu, który jest funkcją wklęsłą.

Literatura

[1] P. Dietz, R. Raman, *A constant update time finger search tree*, 1989.

[2] C. Levcopolous, M. Overmars, *A balanced search tree with O(1) worst-case update time*, 1988.

[3] P. Dietz, D. Sleator, *Two algorithms for maintaining order in a list*, 1987.

[4] G. Brodal, *Finger search trees*, 2005

[5] Wikipedia, *2-3-4 tree*, `http://en.wikipedia.org/wiki/2%E2%80%933%E2%80%934_tree`