Finger Search Tree with Constant Insertion Time

Krzysztof Piecuch

16 lutego 2011

Spis treści

1 Wstęp **1**
- 1.1 Do czego służy finger? . . . 2
- 1.2 Operacje jakie możemy wykonać na strukturze palczastej . . . 2
- 1.3 Co to jest "d"? . . . 3
- 1.4 Intuicja . . . 3
- 1.5 Przykładowe struktury umożliwiające `Finger Search` . . . 5
- 1.6 Zastosowania . . . 6

2 Struktura **6**
- 2.1 Podstawy . . . 6
- 2.2 Wstawianie . . . 7
- 2.3 Twierdzenie . . . 8
- 2.4 Wyszukiwanie przodka w czasie stałym . . . 9
- 2.5 Złe wskaźniki . . . 10
- 2.6 Dzielenie wierzchołka o dowolnym stopniu . . . 10
- 2.7 Liniowy rozmiar struktury . . . 12

3 Usuwanie **12**

4 Finger search **13**

5 Liniowy rozmiar struktury **14**

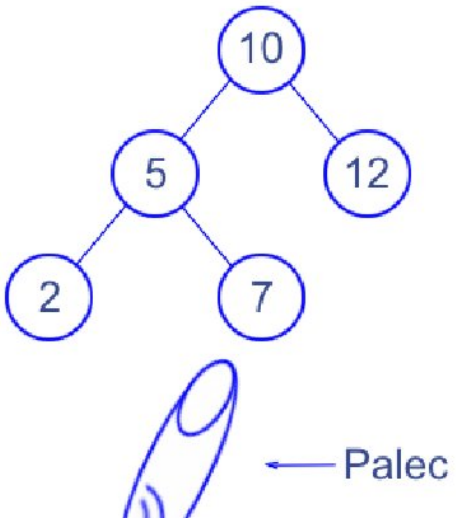
1 Wstęp

Struktura przedstawiona w niniejszej pracy jest na swój sposób niezwykła. Z jednej strony jest ona bardzo skomplikowana. Trudno ją sobie w całości wyobrazić. Choć przy niewielkim wysiłku potrafimy zrozumieć jak działa każda z operacji wykonywana na strukturze, o tyle połączenie tego w całość jest niebagatelnym zadaniem dla naszej wyobraźni. To co będzie dla nas listą podczas wykonywania `Insert` będzie dla nas zmodyfikowanym drzewem Dietza-Ramana zaimplementowanym przy pomocy drzew Levcopoulosa-Overmarsa przy operacji `Delete`. Podczas każdej z operacji będziemy koncentrować na jej szczegółach i abstrahować od tego jak nasza struktura wygląda naprawdę. Z drugiej jednak strony podczas tworzenia tej struktury napotkamy na szereg problemów. Brodal

(autor tej struktury) rozwiązuje je w sposób prosty, piękny i mniej lub bardziej elegancki. Warto zainteresować się tą pracą właśnie dla tych trików i sztuczek użytych w tej strukturze. Można powiedzieć, że Finger Search Tree with Constant Insertion Time jest istną "Piękną i bestią"wśród struktur danych.

1.1 Do czego służy finger?

Finger jest dodatkowym wskaźnikiem w naszej strukturze. Za jego pomocą będą odbywać się wszystkie operacje. Ten wskaźnik będzie zazwyczaj oznaczał miejsce w pobliżu którego będziemy dokonywać operacji. Dzięki temu operacje na strukturze z palcem będą asymptotycznie lepsze od operacji na zwykłych strukturach.



1.2 Operacje jakie możemy wykonać na strukturze palczastej

Mamy następujące operacje, jakie możemy wykonać:

`FINGER Insert(FINGER F, VALUE V)` - Tworzy nowy wierzchołek z wartością `V` bezpośrednio na prawo od wierzchołka na którego wskazuje `F`. Funkcja zwraca wskaźnik na nowo wstawiony element i działa (zazwyczaj) w czasie stałym.

`VOID Delete(FINGER F)` - Usuwamy wierzchołek wskazywany przez finger `F`. Funkcja działa (zazwyczaj) w czasie stałym.

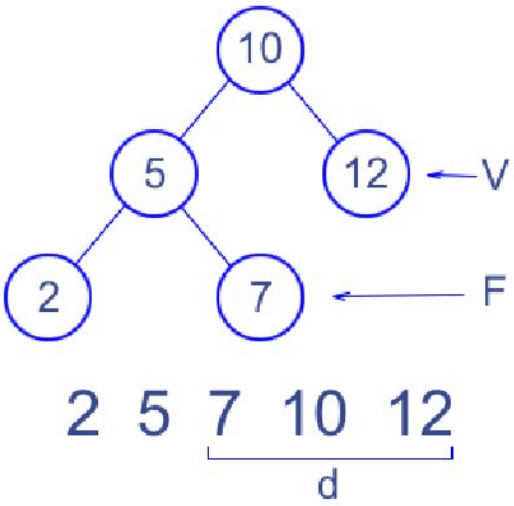
`FINGER Search(VALUE V)` - Funkcja zwraca finger na największy element nie większy niż `V`. Funkcja ta powinna działać w czasie $O(\log d)$ co jest istotą struktur palczastych.

Kryje się tu pewne niebezpieczeństwo. Struktura nie zwraca uwagi jaka wartość jest wstawiana po jakim elemencie (struktura pozwoli na dodanie nowego wierzchołka o dowolnej wartości po dowolnym wierzchołku w strukturze). Z drugiej strony struktury te zakładają, że wierzchołki są posortowane rosnąco (Aby można było szybko wykonywać operację `Search`). Zatem to programista powinien zapewnić, że wstawia nową wartość w odpowiednie miejsce. Jeśli nie

wiemy jaką wartość F powinniśmy podać procedurze to możemy użyć funkcji `Search(V)`.

1.3 Co to jest "d"?

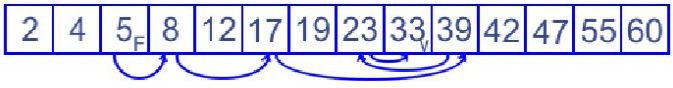
W złożoności operacji `Search` pojawia się zmienna `d`. Czemu jest ona równa? Podczas wyszukiwania będziemy korzystać z dwóch rzeczy: palca `F` i wartości `V`. Wyobraźmy sobie, że wszystkie wartości ze struktury w której wyszukujemy, umieszczamy na posortowanej liście. Wartość `d` jest równa ilości elementów pomiędzy `F` a `V`.



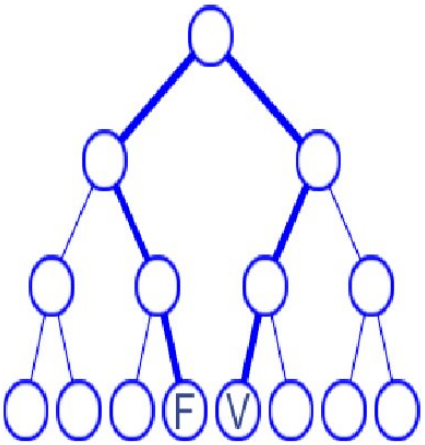
1.4 Intuicja

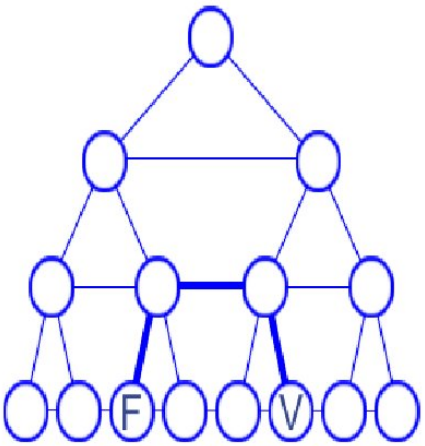
Spróbujmy lepiej zrozumieć jak wygląda wyszukiwanie zadanego elementu w zależności od logarytmu z `d`.

Wyobraźmy sobie posortowaną tablicę. Mamy wskaźniki oznaczające początek i koniec tablicy, zadany palec `F` oraz szukaną wartość `V`. Jak ją wyszukać szybko? Pierwszy pomysł jaki zazwyczaj przychodzi nam do głowy gdy słyszymy hasło "posortowana tablica"to "wyszukiwanie binarne". No i niby słusznie - bo szybciej wyszukać danego elementu `V` się nie da. Jednak w tej sytuacji jest to dla nas... zbyt wolno. Wyszukiwanie binarne zadziała w czasie $O(\log n)$ nawet jeśli element `V` znajduje się blisko naszego palca. Lepszym pomysłem będzie tzw. wyszukiwanie wykładnicze. Bez straty ogólności przyjmijmy, że wyszukiwany element jest większy niż element wskazywany przez palec. Patrzymy na elementy oddalone od palca o $2^0, 2^1, 2^2, 2^3, \ldots$ aż napotkamy na element większy od elementu szukanego. Teraz możemy skorzystać z wyszukiwania binarnego w celu znalezienia naszego elementu. Algorytm działa w czasie $O(\log d)$.



Rozważmy teraz pełne drzewo binarne. Wartości będziemy przechowywać w liściach. Wszystkie liście będą leżały na tym samym poziomie i dla uproszczenia przyjmijmy, że mamy 2^k różnych wartości w drzewie. Jak teraz wygląda wyszukiwanie wartości? Pierwszym pomysłem jaki przychodzi nam do głowy to przechodzenie drzewa do góry od wierzchołka `F` tak długo, aż napotkamy na $LCA(F, V)$, a następnie zejście w dół do wierzchołka `V`. Niestety ta metoda jest za wolna. Na rysunku poniżej widzimy, że metoda ta może wymagać czasu $O(\log n)$ nawet gdy wierzchołek `V` jest następnikiem `F`. Aby usprawnić działanie algorytmu dodajemy dodatkowe wskaźniki do struktury. Łączymy w listę wszystkie wierzchołki od lewej do prawej leżące na tym samym poziomie (rysunek). Wyszukiwanie wygląda teraz następująco. Podobnie jak ostatnio idziemy od wierzchołka `F` w górę. Tym razem jednak oprócz sprawdzenia czy aktualnie rozważany wierzchołek to $LCA(F, V)$ patrzymy także na sąsiadów tego wierzchołka na liście. Jeśli wierzchołek `V` znajduje się w poddrzewie, któregoś z sąsiadów to przechodzimy do niego i rozpoczynamy wyszukiwanie w dół. Algorytm ten działa w czasie $O(\log d)$. Wynika to z prostej obserwacji: Jeśli w którymś kroku będąc w wierzchołku na poziomie i poszliśmy do góry, zamiast do sąsiada to `F` od `V` jest oddalony o conajmniej 2^i.





1.5 Przykładowe struktury umożliwiające `Finger Search`

Poniżej przedstawiam najważniejsze struktury danych wspierające `Finger Search` wraz ze złożonością obliczeniową poszczególnych operacji.

Nazwa struktury	`Insert`	`Delete`	`Search`	Uwagi
linked (2,4)-trees	$O(1)$	$O(1)$	$O(\log d)$	Zamortyzowany
skip list	$O(1)$	$O(1)$	$O(\log d)$	Oczekiwany
treaps	$O(1)$	$O(1)$	$O(\log d)$	Oczekiwany
Dietz-Raman	$O(1)$	$O(1)$	$O(\log d)$	RAM Comparison
Brodal	$O(1)$	$O(\log^* n)$	$O(\log d)$	**Pointer Machine**
Brodal-Lagogiannis-Makris-Tsakalidis-Tsichlas	$O(1)$	$O(1)$	$O(\log d)$	Pointer Machine
Andersson-Thorup	$O(1)$	$O(1)$	$O\left(\sqrt{\frac{\log d}{\log\log d}}\right)$	RAM Word

W 1980 roku Brown i Tarjan zauważyli, że w strukturze zwanej linked (2,4)-trees można wykonać `Finger Search` w najgorszym czasie $O(\log d)$. W 1982 roku Huddleston i Mehlhorn pokazali jak zaimplementować wstawianie i usuwanie w tej strukturze w zaamortyzowanym czasie stałym. W tabelce pojawiają się dwie struktury zrandomizowane. Skip-listy (Pugh 1989) oraz drzewce (Seidel, Aragon 1996). W 1994 roku Dietz-Raman pokazali, że można zaimplementować strukturę wspierającą `Finger Search` w której wstawianie i usuwanie jest w najgorszym czasie stały. Dokonali tego jednak w modelu obliczeń RAM. Dokonali tego tablicując małe drzewa. Założyli jednak że elementy można jedynie porównywać. W 1998 roku Brodal przedstawił strukturę danych działającą w modelu pointer-base, która umożliwiała wstawianie w najgorszym czasie stałym. Jednak usuwanie ze struktury odbywało się w czasie $O(\log^* n)$. Poprawił tym sposobem wynik Harela i Luekera z 1980 roku, którzy to zaprezentowali strukturę wspierającą `Finger Search` z czasem wstawiania i usuwania $O(\log^* n)$. W 2002 roku Brodal, Lagogiannis, Makris, Tsakalidis i Tsichlas przedstawili strukturę danych działającą w modelu pointer-base w której usuwanie i dodawanie wartości odbywa się w najgorszym czasie stałym. W 2000 roku Anderson i Thorup przekroczyli w modelu RAM granicę logarytmu otrzymując czas zapytania $O\left(\sqrt{\frac{\log d}{\log\log d}}\right)$. Rezultat ten został osiągnięty poprzez rozpatrzenie wartości jako słów maszynowych i zastosowanie technik pokonujących dolne ograniczenia dla struktur bazujących na porównywaniu.

1.6 Zastosowania

Wyobraźmy sobie, że mamy dane dwie struktury danych (X oraz Y) zawierające jakieś elementy. Przyjmijmy, że X ma n elementów, a Y ma m elementów, przy czym $n \leq m$. Chcemy utworzyć jedną strukturę złożoną z elementów obu struktur. Rozważmy następujący algorytm:

MERGE(X, Y)
1 **for** $x \in X$
2 Y.INSERT(x)

Jeśli Y jest zwykłym zbalansowanym drzewem wyszukiwań binarnych to powyższy algorytm zadziała w czasie $O(n \log m)$. Gdyby jednak Y było strukturą

palczastą to złożoność algorytmu wyniosłaby $O(n \log \frac{m}{n})$. Dlaczego? Algorytm wykonuje n wstawień w czasie stałym. Przed każdym wstawieniem poszukujemy w czasie $O(\log d)$ miejsca w które należy wstawić dany element. Ponadto ponieważ wstawiamy elementy coraz większe zachodzi: $\sum_{i=1}^{n} d_i \leq m$. Zatem $\sum_{i=1}^{n} \log d_i \leq n \log \frac{m}{n}$.

Poniżej przedstawiam inne zastosowania struktur wspierających `Finger Search`. Są to problemy z geometrii obliczeniowej oraz algorytmów tekstowych. Nie byłem pewny polskiej terminologii niektórych zagadnień dlatego postanowiłem pozostawić je w języku angielskim.

- Three-dimensional layers of maxima
- Visibility and shortest path problem inside simple polygon
- Visibility graph of a simple polygon
- Triangulating a simple polygon
- Maximal quasiperiodicities in strings
- Maximal pairs with bounded gap

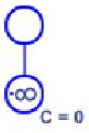
2 Struktura

2.1 Podstawy

Zdefiniujmy sobie na początku nieskończenie wiele stałych: $\Delta_1, \Delta_2, \Delta_3, \ldots$, przy czym wymagamy od nich aby spełniały warunek: $\Delta_d \geq 2^{2^d} - 1$. Stałe te użyjemy przy algorytmie wstawiania wierzchołka do drzewa. Nasza struktura będzie ponadto miała następujące własności:

- Nasze drzewo będzie miało wartości tylko w liściach.
- W każdym wierzchołku wewnętrznym zapiszemy przedział wartości jakie zawiera.
- Wszystkie liście znajdują się na tym samym poziomie.
- W każdym liściu trzymamy dodatkowo wartość c.
- Poziom na którym znajdują się liście oznaczamy jako poziom 0.

Pusta struktura wygląda następująco:



2.2 Wstawianie

INSERT(f, v)

1 l = liść na który wskazuje f
2 p = ojciec liścia l
3 l' = nowy wierzchołek z wartością v
4 wstawiamy l' jako prawego brata l i syna p
5 $c_{l'} = ++c_l$
6 Niech d spełnia $c_l \mod 2^d = 2^{d-1}$
7 v = przodek l oraz l' na poziomie d
8 Jeśli wierzchołek v ma stopień conajmniej $2\Delta_d$ to
9 dzielimy go na dwa wierzchołki v' oraz v'' każdy o stopniu conajmniej Δ_d.
10 Jeśli rozdzieliliśmy korzeń to tworzymy nowy wierzchołek o stopniu 2.

Mam nadzieję, że algorytm jest w miarę jasny. Niepokojącą może być wartość d. W następnym rozdziale wyjaśnie, że istnieje tylko jedna wartość d spełniająca warunek określony w linii 6. Ponadto wyjaśnimy jak rozumieć intuicyjnie ten warunek. Algorytm ten jak wykażemy działa w czasie stałym. Wszystkie operacje opisane potrafimy wykonać w czasie stałym. Wyjątek mogą stanowić instrukcje dotyczące znalezienia przodku na poziomie d oraz podziału wierzchołka dowolnego stopnia. Jak to zrobić opiszemy w kolejnych rozdziałach.

2.3 Twierdzenie

Zajmiemy się teraz twierdzeniem, który mówi że nasz algorytm wstawiania elementu do drzewa gwarantuje nam, że stopień każdego z wierzchołków nie jest zbyt duży. Dowód tego twierdzenia jest dość trudny ale i piękny. Jeśli ogarnia Cię senność pomiń ten rozdział - w tym dokumencie jest dużo o wiele ciekawszych rzeczy do przeczytania.

Twierdzenie 1 *Algorytm gwarantuje, że każdy wierzchołek na poziomie d ma stopień conajmniej Δ_d a conajwyżej $2^{2\times 2^d}\Delta_d$. Oprócz korzenia, który może mieć stopień conajmniej* 2.

Twierdzenie to będziemy próbowali udowodnić aż do końca tego rozdziału. Zacznijmy od prostego spostrzeżenia.

d*jestpozycjnajbardziejprawejjedynkiwzapisiebinarnymliczbyc*$_l$ wtedy i tylko wtedy gdy $c_l \mod 2^d = 2^{d-1}$.

Spostrzeżenie to wyjaśnia czym jest d w naszym algorytmie. Pozycję najbardziej prawego bitu w zapisie binarnym oznaczamy jako pozycję pierwszą. Dowód tego spostrzeżenia otrzymamy gdy zorientujemy się, że równość $c_l = b\times 2^d + 2^{d+1}$ jest równoważna z obydwoma warunkami z spostrzeżenia.

Wprowadźmy teraz kilka oznaczeń. Dla każdego liścia l oraz poziomu p wprowadzamy potencjał: $\Phi_l^p = 2^{2^p - 1 - ((c_l + 2^{p-1}) \mod 2^p)}$. Oczywiście zachodzi: $1 \leq \Phi_l^p \leq 2^{2^p-1}$. Ponadto d takie jak w spostrzeżeniu wtedy i tylko wtedy gdy $\Phi_l^d = 2^{2^d-1}$. Dla każdego wierzchołka v leżącego na poziomie p definiujemy potencjał $\Phi_v^p = \sum_{l\in T_v} \Phi_l^p$, gdzie T_v oznacza zbiór wszystkich liści znajdujących się w poddrzewie o korzeniu v. Potencjały wyglądają trochę przerażająco. Pomogą nam one jednak w udowodnieniu naszego spostrzeżenia. Przy pierwszym

czytaniu pracy Brodala zacząłem się zastanawiać skąd autor wiedział, że potencjał ma wyglądać właśnie tak. Z kolejnymi krokami rozjaśni nam się to nieco bardziej.

Zastanówmy się jak wstawienie nowego wierzchołka wpływa na potencjał Φ_v^p. Wartość ta ma szansę się zmienić jedynie wtedy gdy wstawiamy wierzchołek do poddrzewa o korzeniu w v. Rozpatrzymy zatem jedynie takie sytuacje. Rozważmy dwa przypadki. W pierwszym z nich przyjmijmy, że wierzchołek nie leży na poziomie d (ze spostrzeżenia). Jeśli $p \neq d$ to $c_l \mod 2^p \neq 2^{p-1}$. Zatem po powiększeniu się wartości c_l wartość Φ_l^p zmniejszy się dwukrotnie (bo zmniejszy się o jeden wartość w wykładniku). Niech Φ oznacza starą wartość Φ_l^p. Mamy $\Phi = \frac{\Phi}{2} + \frac{\Phi}{2} = \Phi_l^p + \Phi_l^p \Phi_l^p + \Phi_{l'}^p$. Stąd wartość Φ_v^p nie zmienia się!

Drugi przypadek. Rozważamy wierzchołek, który w naszym algorytmie jest kandydatem do rozdzielenia. Czyli $p = d$. Stare $\Phi_l^p = 2^{2^p-1-((2^{p-1}-1+2^{p-1}) \mod 2^p)} = 2^{2^p-1-((2^p-1) \mod 2^p)} = 2^{2^p-1-2^p-1} = 2^0 = 1$. Z kolei nowe $\Phi_l^p = \Phi_{l'}^p = 2^{2^p-1-((2^{p-1}+2^{p-1}) \mod 2^p)} = 2^{2^p-1}$. Stąd wartość Φ_v^p zwiększa się o $2 \times 2^{2^p-1} - 1 = 2^{2^p} - 1$ zanim postanowiliśmy rozdzielić wierzchołek v.

Kolejnym naszym krokiem będzie próba udowodnienia następującej nierówności: $\Phi_v^p \leq 2^{2\times 2^p} \prod_{i=1}^p \Delta_i$. Dowód przebiegnie indukcyjnie. Dla drzewa pustego nierówność oczywiście zachodzi. Wstawiamy do drzewa nowy wierzchołek. Jak powiedzieliśmy przed chwilą jedynie Φ_v^d zwiększa się o $2^{2^d} - 1$. Jeśli wierzchołek v się rozdzielił na v' oraz v'' to ich stopnie są conajmniej $\Delta_d \geq 2^{2^d} - 1$ zatem ich potencjały także. $\Phi_v^d + 2^{2^d} - 1 = \Phi_{v'}^d + \Phi_{v''}^d$. $\Phi_{v'}^d = \Phi_v^d + 2^{2^d} - 1 - \Phi_{v''}^d \leq \Phi_v^d + 2^{2^d} - 1 - (2^{2^d} - 1) = \Phi_v^d \leq 2^{2\times 2^d} \prod_{i=1}^d \Delta_i$. Ostatnia nierówność wynika z założenia indukcyjnego. Analogicznie dowodzimy dla $\Phi_{v''}^d$. Co jeśli wierzchołek nie został rozdzielony? Przeprowadzimy dowód indukcyjny po wartości d. Gdy $d = 1$ to stopień v jest conajwyżej $2\Delta_1 - 1$. Każdy z liści w tym poddrzewie ma potencjał $\Phi_l^1 \leq 2$. Stąd $\Phi_v^1 \leq (2\Delta_1 - 1) \times 2 \leq 2^{2\times 2^1} \Delta_1$. Krok indukcyjny - $d > 1$. Nasz wierzchołek ma nie więcej niż $2\Delta_d$ synów. Każdy z nich ma z założenia indukcyjnego nie więcej niż $2^{2\times 2^{d-1}} \prod_{i=1}^{d-1} \Delta_i$ liści. Każdy z tych liści ma potencjał nie większy niż 2^{2^d-1}. Otrzymujemy $\Phi_v^d \leq 2\Delta_d \left(2^{2\times 2^{d-1}} \Pi_{i=1}^{d-1}\right) 2^{2^d-1} \leq 2^{2\times 2^d} \prod_{i=1}^d \Delta_i$.

Ponieważ wierzchołek na poziomie d dzieli się tylko wtedy gdy ma stopień conajmniej $2\Delta_d$ to wierzchołki na poziomie d (prócz korzenia) mają stopień conajmniej Δ_d. Zatem ilość liści w poddrzewie o korzewniu w v spełnia nierówność: $|T_v^d| \geq \prod_{i=1}^d \Delta_i$.

Ostatecznie otrzymujemy dowód naszego twierdzenia. Wierzchołek na poziomie d ma potencjał Φ_v^d przy czym każdy z jego synów ma co najmniej $\prod_{i=1}^{d-1} \Delta_i$ liści, zatem także conajmniej taki potencjał. Stąd wierzchołek ten ma conajwyżej $\Phi_v^d / \prod_{i=1}^{d-1} \Delta_i \leq 2^{2\times 2^d} \prod_{i=1}^d \Delta_i / \prod_{i=1}^{d-1} \Delta_i = 2^{2\times 2^p} \Delta_d$ synów. Co kończy dowód.

2.4 Wyszukiwanie przodka w czasie stałym

Czeka nas trudne zadanie. Chcemy w czasie stałym mieć dostęp do przodka na poziomie d. Nie możemy trzymać wskaźników do wszystkich przodków liścia l, bo po pierwsze nasza struktura nie będzie miała rozmiaru liniowego, po drugie rozdzielenie wierzchołka w naszym algorytmie wymagałoby bardzo dużego nakładu pracy. Wykorzystamy fakt, że dla zadanego wierzchołka nasze zapytania o kolej-

nych przodków mają charakterystyczną strukturę $(1, 2, 1, 3, 1, 2, 1, 4, 1, 2, 1, \ldots)$.

Z każdym liściem będziemy trzymali stos zawierający trójki (i, j, u_j), gdzie $i \leq j$ są pozycjami (w binarnej reprezentacji c_l) a u_j jest wstaźnikiem na przodka liścia l na poziomie j. $[i, j]$ reprezentuje spójny fragment samych jedynek. Na przykład jeśli mamy wartość $c_l = 001110011010_2$. to na stosie będziemy trzymać wartości (począwszy od szczytu stosu): $(2, 2, .), (4, 5, .), (8, 10, .)$. Ponieważ taki stos reprezentuje jednoznacznie wartość c_l to nie będziemy już dłużej trzymali tej wartości w liściach - wystarczy nam stos.

Potrzebujemy teraz zastanowić się nad trzema rzeczami. W algorytmie mamy operację zwiększenia wartości c_l, skopiowania stosu do nowo utworzonego wierzchołka - musimy pokazać jak wykonywać te operacje. No i w końcu musimy pokazać jak znaleźć przodka w czasie stałym.

Zajmijmy się najpierw kopiowaniem stosu. Nie możemy po prostu skopiować każdego elementu z osobna. Zajęłoby nam to $O(\log \log n)$ czasu. Jak zatem rozwiązać nasz problem? Użyjemy techniki zwanej współdzieleniem danych. Tak zmodyfikowany stos pojawia się w językach funkcyjnych. „Skopiowanie" stosu będzie polegało teraz tylko na utworzeniu nowego wskaźnika do obecnego już stosu. Oba stosy nie będą wchodziły sobie w drogę, musimy jedynie uważać przy usuwaniu elementów ze stosu. Usuwamy element jedynie wtedy gdy nie wskazuje na niego już żaden wskaźnik.

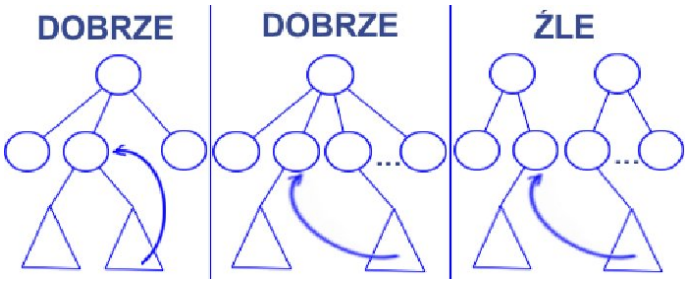
Zaimplementujemy teraz funkcję, która zwiąksza o jeden wartość c_l i przy okazji zwraca nam przodka znajdującego się na poziomie d. Oczywiście funkcja będzie działała w czasie stałym.

INCREMENT$(Leafl, StackS)$

```
1  if S == ∅
2       Wrzuć do S (1, 1, p_l)
3  else (i, j, u_j) = szczyt stosu
4       if i > 1
5            Wrzuć do S (1, 1, p_l)
6       else Usuń(i, j, u_j)
7            Wrzucamy (j + 1, j + 1, p_{u_j})
8  (i, j, u_j) = szczyt stosu
9  v = u_j
10 Jeśli trzeba mergujemy dwa wierzchołki ze szczytu
11 return v
```

2.5 Złe wskaźniki

Pojawia się pewien problem. Otóż podczas dzielenia wierzchołków może się zdarzyć, że wskaźniki w liściach przestaną być aktualne. Przyczym niestać nas aby podczas dzielenia zaktualizować wszystkie wierzchołki. Rozwiązaniem tego problemu jest... zignorowanie go. Zauważmy, że zwykłe podzielenie wierzchołków niczego nam jeszcze nie psuje. Po ściągnięciu wierzchołku ze stosu funkcja inkrementacji od razu przechodzi do jego ojca. Nie robi nam zatem różnicy czy mamy wskaźnik do faktycznego przodka liścia l czy do jego brata (patrz rysunek). Pecha będziemy mieli kiedy zarówno wierzchołek na który mieliśmy wskaźnik, jak i jego ojciec zostanę podzielone. Intuicyjnie czujemy jednak, że taka sytuacja nie będzie miała miejsca zbyt często.

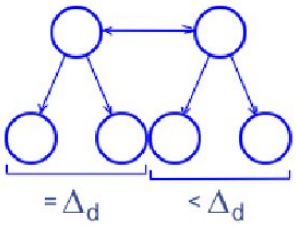


Algorytmy pozostają bez zmian. Zdarzać się zatem będzie, że rozdzielimy wierzchołek, który nie będzie przodkiem liścia, który teraz wstawiliśmy. Zmianie ulegnie twierdzenie, które tak pracowicie dowiedliśmy 3 sekcje wcześniej. Nowej wersji tego twierdzenia nie będziemy tu dowodzić.

2.6 Dzielenie wierzchołka o dowolnym stopniu

Mamy kolejny nietrywialny problem. Otóż chcemy podzielić wierzchołek o dowolnym stopniu na dwa wierzchołki. Algorytm wstawiania wierzchołka do drzewa zakładał, że w każdym wierzchołku trzymamy wskaźnik do ojca. Zatem gdy dzielimy wierzchołek musimy zadbać o aktualizację wskaźników jego dzieci. Zgodnie z algorytmem tych aktualizacji ma być Δ_d czyli ponad wykładniczo wiele względem d! Zadanie to wygląda na niewykonalne. Jednak możemy skorzystać tu z faktu, że nie dzielimy wierzchołków byle jak. Dzielimy tylko te wierzchołki które mają conajmniej $2\Delta_d$ synów na dwa wierzchołki mające conajmniej Δ_d synów.

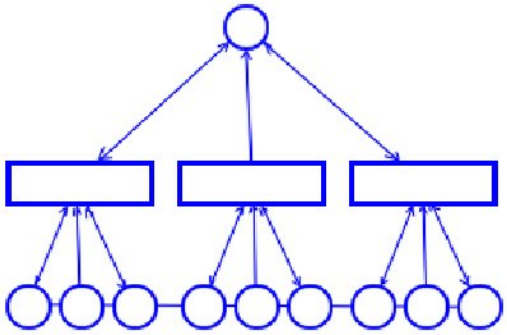
Utwórzmy warstwę pośrednią. Będzie to warstwa zawierająca się między wierzchołkiem v a jego synami. Synów podzielimy na bloki. Każdy wierzchołek z warstwy pośredniej będzie reprezentował jeden taki blok. Każdy blok będzie zawierał conajmniej Δ_d wierzchołków, ale nie więcej niż $2\Delta_d - 1$.



W wierzchołku v trzymamy wskaźnik na najbardziej lewy i najbardziej prawy blok. Wierzchołki w warstwie pośredniej mają wskaźniki na najbardziej lewe i najbardziej prawe dziecko oraz wskaźnik do v. Natomiast dzieci wierzchołka v będą trzymane na liście (od lewego do prawego) oraz będą miały wskaźnik na blok w którym się znajdują. Zauważmy, że znalezienie ojca dziecka nadal odbywa się w czasie stałym - dziecko zawiera wskaźnik na blok, a blok zawiera wskaźnik na ojca.

Struktura jest już prawie gotowa. Dodanie nowego syna dla v jest dość proste. Co jednak jeśli wstawiając nowy wierzchołek sprawimy, że któryś z bloków stanie się za duży? Zmodyfikujmy zatem nieco naszą warstwę. Jeśli blok będzie zawierał

więcej niż Δ_d wierzchołków to będziemy go reprezentowali za pomocą dwóch wierzchołków połączonych wskaźnikami. Pierwszy element pary będzie zawierał dokładnie Δ_d wierzchołków, natomiast drugi będzie miał ich ostro mniej niż Δ_d. Jeśli teraz w bloku będziemy mieli wierzchołek, który zawiera $2\Delta_d$ wierzchołków to wystarczy zamienić parę reprezentującą ten blok na dwa osobne wierzchołki (wystarczy wymazać wskaźniki do pary). Wstawienie do tak zmodyfikowanej struktury będzie wyglądało następująco:



INSERT($Leafl, StackS$)

```
1  if w jest parą
2      if wstawiamy do pierwszego el. pary
3          Wstaw do pierwszego el. pary
4          Skrajny prawy wierzchołek bloku przepisz do drugiego el. pary
5      else wstaw do drugiego el. pary
6      if drugi el. pary ma Δ_d wierzchołków
7          Rozdziel parę
8  else Utwórz do niego parę
```

Możemy teraz łatwo sprawdzić czy wierzchołek v powinien zostać rozdzielony według algorytmu z początku rozdziału. Powinien on zostać rozdzielony wtedy i tylko wtedy gdy w warstwie pośredniej znajdują się conajmniej dwa bloki. Rozdzielenie wierzchołka v wygląda teraz w ten sposób, że wierzchołkowi v' pzyporządkowujemy skrajnie prawy blok warstwy pośredniej v. Przestawiając w tym bloku wskaźnik na ojca na wierzchołek v' dokonujemy czegoś co na początku wydawało się niemożliwe - przyporządkowujemy nowemu wierzchołkowi ponad wykładniczo wiele synów w czasie stałym i wciąż dla każdego z tych synów w czasie stałym potrafimy znaleźć jego ojca.

2.7 Liniowy rozmiar struktury

Czy rozmiar struktury jest liniowy? Być może dla niektórych nie jest to takie oczywiste. W każdym liściu możemy trzymać nawet $\log\log n$ wskaźników. Całość struktury może ograniczyć przez $n \log\log n$. Czy da się lepiej? Okazuje się, że tak! Struktura zajmuje $O(n)$ pamięci, a argument na to jest naprawdę prosty. Nasza struktura powstaje przez wstawienie n elementów do początkowo pustego drzewa. Każde wstawienie zajmuje czas stały. W tym czasie możemy zająć conajwyżej stałą ilość nowej pamięci. Stąd po n wstawieniach struktura będzie miała rozmiar $O(n)$.

3 Usuwanie

Oznaczmy $2^{(1)} = 2$ oraz $2^{(d+1)} = 2^{2^{(d)}}$. Jeśli teraz ustawimy $\Delta_d = 2^{(d)}$ to nasze drzewo będzie miało wysokość rzędu $O(\log^* n)$. Będziemy mogli w związku z tym przejść w górę przez całe drzewo.

Podstawowa idea algorytmu wygląda następująco. Usuwamy wierzchołek l. Jeśli jego ojciec ma odpowiedni stopień to kończymy. Jeśli natomiast ma teraz za mały stopień to próbujemy go scalić z jego lewym bądź prawym bratem. Jeśli teraz jego brat ma zbyt duży stopień to dzielimy go na dwa wierzchołki (w tym miejscu może nas kusiś aby zamiast tych operacji przestawić jakiś wierzchołek od naszego brata tak aby nasz wierzchołek miał wciąż odpowiedni stopień; nie możemy tego zrobić ze względu na wskaźniki w liściach!). Jeśli tego niezrobiliśmy to ojciec tego wierzchołka stracił jednego syna - wchodzimy na poziom wyżej i wykonujemy to samo.

Pierwszy problem z jakim się musimy zmierzyć to fakt, że być może usuwamy jakiś wierzchołek na którego wskazuje jakiś wskaźnik ze stosu w liściu. Rozwiązaniem tego problemu jest oznaczenie tego wierzchołka jako martwe dziecko (dead children). Usuwamy wszystkie jego wskaźniki, prócz wskaźnika na jego ojca. Może się zdarzyć, że jego ojciec jest również martwy. Martwe wierzchołki rezydują w pamięci, jednak są oznaczone jako martwe. Możemy je usunąć, gdy nie będzie w strukturze wskaźnika na niego.

Drugi problem to potencjał. Najpierw łączymy wierzchołek v z jego bratem v' co spowoduje zwiększe się potencjału $\Phi^d_{v'}$. Gdy rozdzielamy wierzchołek v' na v' oraz v'' chcielibyśmy „cofnąć" wzrost tego potencjału. Niestety nie wystarczy przesunąć $\Theta(\Delta_d)$ wierzchołków do v''. Może bowiem się zdarzyć, że liście które dodaliśmy do wierzchołka v' mają większy potencjał niż wierzchołki, które wyrzuciliśmy z niego przerzucając je do v''. Można wyliczyć, że jeśli chcemy zagwarantować aby nasz potencjał nie wzrósł potrzebujemy przenieść $\Gamma_d\Delta_d$ synów, gdzie $\Gamma_d = 2^{3(d-1)2^{d-1}}2^{2^d+2^{d-1}-2}$. Wartość Γ_d jest równa maksymalnemu potencjałowi wierzchołka na wysokości $d-1$ (spostrzegawczy czytelnik zauważy że wartość którą tu podaliśmy różni się od tej którą wyliczyliśmy w rozdziale 2; wynika to stąd że po modyfikacjach struktury którą przeprowadziliśmy zmodyfikowały się również te wartości; Brodal w swojej pracy sumiennie po każdej takiej modyfikacji liczy ponownie te wartości; my sobie to odpuściliśmy) podzielony przez $\prod_{i=1}^{d-1}\Delta_i$ i pomnożemy przez maksymalny potencjał liścia Φ^d_l.

Jak teraz w czasie stałym przenieść $\Gamma_d\Delta_d$ wierzchołków? Czytelnik być może już się domyśla, gdyż użyjemy metody, którą już wykorzystywaliśmy. Wprowadzimy dodatkową warstwę pośrednią pomiędzy ojcem v a poprzednią warstwą pośrednią. Blok konstruujemy w analogiczny sposób jak poprzednio, z tym że teraz nasz blok będzie miał rozmiar conajmniej Γ_d i mniej niż $2\Gamma_d$.

4 Finger search

Pierwszym krokiem będzie zmodyfikowanie struktury Dietza-Ramana (patrz rozdział dotyczący przykładowych struktur) tak aby działał na pointer machine. Struktura Dietza-Ramana działa na (2,3)-drzewach gdzie w liściu trzymamy kubełek zawierający $\Theta(log^2 n)$ elementów. Możemy tutaj użyć linked (2,3)-tree przedstawionego przez Browna i Tarjana. Jedynym elementem w którym struktura Dietza-Ramana korzysta z maszyny RAM są kubełki. Jeśli taki kubełek

umożliwia wstawianie i usuwanie w czasie stałym i kubełki możemy rozdzielać i łączyć w czasie $O(\log n)$ to możemy wstawiać i usuwać ze struktury liście w czasie stałym i wykonać `Finger Search` w czasie $O(\log d)$ plus czas potrzebny do wykonania `Finger Search` w kubełku. Dalej Dietz i Raman pokazują jak wykonać wyszukiwnie w kubełku w czasie $O(\log d)$. My pokażemy, że można je wykonać w czasie $O(\log \log n)$ wykorzystując pointer machine.

Nasze kubełki reprezentujemy w postaci drzewa posiadającego 3 poziomy. Każdy wierzchołek na poziomie 1 (czyli środkowym) posiada stopień pomiędzy $\log n$ a $2 \log n - 1$. Jeśli wierzchołek taki posiada conajmniej $\log n + 1$ synów to reprezentujemy go jako parę wierzchołków. Tak! To wygląda jak warstwa pośrednia wprowadzona w poprzednich rozdziałach. Dodawanie i usuwanie liścia odbywa się podobnie jak w warstwie pośredniej. Rozdzielenie kubełka odbywa się w czasie $O(\log n)$ poprzez liniowe przeniesienie $O(\log n)$ wierzchołków z poziomu 1 do nowego kubełka. Dla każdego wierzchołka w tej strukturze jego dzieci trzymamy w strukturze Levcopoulosa i Overmarsa. Jest to struktura oparta na BST umożliwiająca wyszukiwanie w czasie $O(\log n)$, a wstawianie i usuwanie w czasie $O(1)$. Możemy teraz w naszej strukturze dodawać i usuwać liście w czasie stałym, a wyszukiwać w tej strukturze w czasie $O(\log \log n)$ (bo rozmiar naszej struktury jest $O(\log^2 n)$). Zatem zgodnie z twierdzeniem panów Dietza i Ramana (patrz akapit wyżej) otrzymaliśmy strukturę danych która umożliwia wstawianie i usuwanie w czasie stałym, a `Finger Search` w czasie $O(\log \log n + \log d)$.

Teraz trzymajmy się mocno krzeseł. W naszej strukturze danych każdy wierzchołek z warstwy pośredniej (mówimy tu o warstwie z rozdziału drugiego, a nie tej z rozdziału o usuwaniu wierzchołków) zastępujemy strukturą opisaną wyżej.

Wyszukiwanie w naszej strukturze wygląda teraz następująco. Wejście do góry po drzewie jest dla nas tanie. Powiedzmy, że doszliśmy do wierzchołka v na poziomie p takiego, że poddrzewo o korzeniu w v zawiera szukany przez nas element x. Szukamy teraz tego elementu schodząc w dół: Patrzymy na warstwę pośrednią. Jeśli szukany element jest w tym samym albo sąsiednim bloku co palec f to schodzimy do niego i wyszukujemy dziecko v które zawiera x w czasie $O(\log d + \log \log 2^{(p)})$ jak to opisaliśmy dwa akapity temu. Jeśli szukanego elementu nie ma w tych blokach, to znaczy, że $d \geq 2^{(p)}$. Przeglądamy w takiej sytuacji wszystkie bloki w poszukiwaniu tego do którego mamy zejść. Możemy to zrobić gdyż bloków jest conajwyżej 2^{3p2^p}. Zatem wyszukiwanie odbędzie się wtedy w czasie $O(2^{3p2^p} + \log 2^{(p)}) = O(\log d)$. Podsumowując potrzebujemy w tym kroku czasu $O(\log d + \log \log 2^{(p)})$. Gdy schodzimy poziom niżej kontynuujemy poszukiwanie z tym, że teraz będziemy za każdym razem przeszukiwać wszystkie bloki w warstwie pośredniej. Zatem na każdym poziomie i wykonamy pracę w czasie $O(2^{3i2^i} + \log 2^{(i)})$.

Całkowity czas działania wyniesie zatem:

$$\log d + \log \log 2^{(p)} + \sum_{i=1}^{p-1} \left(2^{3i2^i} + \log 2^{(i)} \right. = O(\log d + \log 2^{(p-1)}) = O(\log d)$$

Bo $d \geq 2^{(d-1)}$. Ufff... Oto i cała struktura.

5 Liniowy rozmiar struktury

Niestety nasza struktura może nie mieć liniowego rozmiaru. Wyobraźmy sobie, że wstawiamy do naszej struktury n elementów. Następnie wstawiamy i usuwamy ten sam element powiedzmy k razy - gdzie k najlepiej jest podwójnie wykładnicze względem n. Po takim obrocie sprawy mamy wciąż n elementów w strukturze, jednak jeden z liści może mieć wartość c na poziomie $O(k)$ - zatem potrzebuje $O(\log k)$ pamięci na stos. Innym problemem są martwe dzieci. Każdy liść może mieć ich $O(\log^* n)$ stąd rozmiar struktury może osiągnąć $O(n \log^* n)$. Obie te sytuacje są czysto teoretyczne. Trudno sobie wyobrazić abyśmy wykonywali wykładniczą ilość operacji na strukturze. Także druga sytuacja jest trudna do uzyskania, a nawet rozmiar $O(n \log^* n)$ w praktycznych zastosowaniach jest tak samo dobry jak liniowy. Zajmiemy się jednak tutaj tym problemem, gdyż technika, którą tu użyjemy jest bardzo ciekawa.

Użyjemy techniki zwanej global rebuilding technique. Ogólny zarys tej techniki wygląda następująco. Oprócz naszej struktury T będziemy trzymać drzewo T', które na początku jest puste. Przy każdej operacji `Insert` oraz `Delete` wstawiamy/usuwamy wierzchołek do obu drzew (z T' usuwamy wierzchołek tylko jeśli on w T' istnieje). Ponadto w strukturze T mamy wskaźnik początkowo ustawiony na najmniejszy element w drzewie. Przy każdej operacji `Insert` oraz `Delete` przestawiamy ten wskaźnik na następny element i wstawiamy go do drzewa T' jeśli jeszcze go tam nie ma.

Gdy dojdziemy do ostatniego elementu drzewa T możemy to drzewo usunąć. Nowym drzewem T staje się T' i zabawa zaczyna się od początku. Musimy jednak jeszcze usunąć drzewo T. Wykonujemy to w podobny sposób jak poprzednio. Przy każdej operacji wstawiania i usuwania staramy się pozbyć pewnej stałej ilości elementów starego drzewa. Dzięki temu gwarantujemy liniowy rozmiar naszej struktury.

Opisałem tylko idee tej techiniki. Aby móc ją zastosować w czasie stałym potrzebujemy dodatkowych wskaźników. Z kolei przy analizie rozmiaru drzewa T' przydaje się aby podczas usuwania wstawiać do drzewa T' $O(\log^* n)$ nowych wierzchołków. Po więcej szczegółów odsyłam do pracy Brodala.