Analiza drzew samoorganizujących

(Analysis of self ordering binary search trees)

Julia Majkowska

Praca licencjacka

Promotor: dr hab. Marcin Bieńkowski

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

9 maja 2019

Streszczenie

W tej pracy porównuje wydajność drzewa Splay, drzewa Tango oraz drzewa czerwono - czarnego

---

...

Spis treści

Rozdział 1.

Wprowadzenie

1.1. Wstęp

Binarne drzewo przeszukiwań to jedna z najbardziej podstawowych struktur danych w informatyce. Stosuje się je w wielu systemach informatycznych i algorytmach. Ideą struktury jest trzymanie elementów w wierzchołkach, trzymających wartość elementu oraz wskaźniki na lewe i prawe poddrzewo. Elementy umieszcza się w strukturze w taki sposób, aby wszystkie wartości elementów znajdujące się w lewym poddrzewie wierzchołka w były elementami mniejszymi od w w zdefiniowanym porządku. Analogicznie wszystkie elementy w prawym poddrzewie są elementy większe w tym porządku. Na tak uporządkowanych danych można wyszukiwać danej wartości v poprzez zagłębianie się w drzewo wybierając odpowiednio lewe lub prawe poddrzewo w zależności od wyniku porówna v z wartością korzenia. W najbardziej podstawowej formie tej struktury operacje wstawiania, wyszukiwania i usuwania wykonuje się w czasie O(H), gdzie H to wysokość drzewa. W pesymistycznym wypadku jest to czas liniowy od liczby elementów w strukturze. Złożoność operacji słownikowych na drzewie BST zależy od głębokości drzewa, dlatego powstało wiele struktur danych ograniczających maksymalną głębokość drzewa. Spośród deterministycznych można wymienić drzewa Splay, drzewa AA oraz czerwono-czarne, obsługujące operacje słownikowe w pesymistycznym czasie O(logn).

1.2. Problem optymalnego drzewa binarnego

W wielu zastosowaniach dane nie podlegają rozkładowi jednostajnemu tylko przejawiają jakąś lokalność. Przykładowo, jeśli dla drzewa 100- elementowego będą się pojawiać głównie zapytania o jeden element, można by znacząco zmniejszyć czas działania przesuwając dany element do korzenia. W takim wypadku struktura naszego drzewa zależy od zapytań. Problem znajdowania drzewa binarnego, które przy użyciu najmniejszej liczby operacji obsłuży ciąg zapytań to problem optymalnego

drzewa binarnego. Możemy rozróżnić dwa warianty tego problemu statyczny oraz dynamiczny.

1.2.1. Statyczne optymalne drzewo binarne

Wg. definicji Knutha definiujemy uporządkowany ciąg elementów $e_1...e_n$, ciąg prawdopodobieństw $A_1...A_n$, będące prawdopodobieństwem wykonania wyszukiwaniae_i i $B_0 \dots B_n$, będące prawdopodobieństwem wyszukania elementu z przedziału $[e_i, e_{i+1}]$. Szukamy drzewa minimalizujący oczekiwany czas wyszukiwania.

Wraz z definicją problemu Knuth opublikował algorytm opierający się n a programowaniu dynamicznym, znajdujący optymalne drzewo binarne w czasie O(n^2). W 1975 r. Kurt Mehlhorn opublikował algorytm aproksymacyjny, działający w czasie O(n). Dla wariantu, w którym $B_i = 0$, istnieje algorytm Garsia-Wachsa, który znajduje optymalne drzewo w czasie O(nlogn).

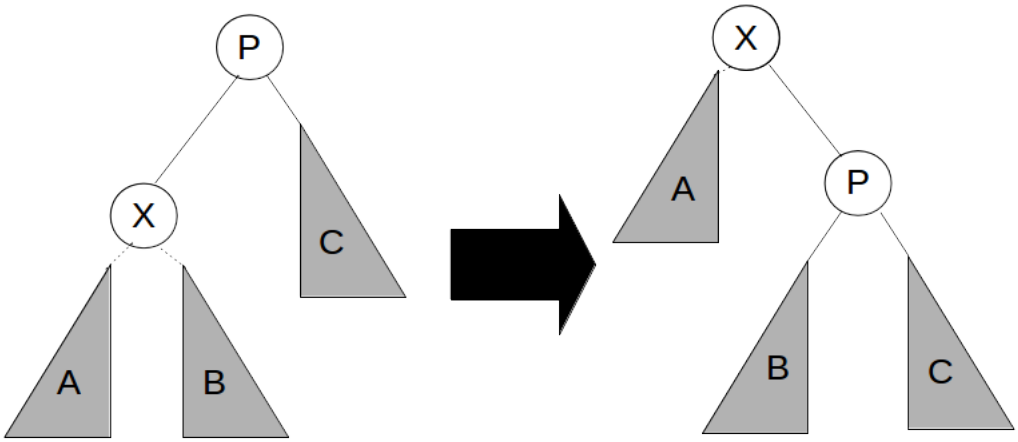
1.2.2. Dynamiczne optymalne drzewo binarne

W dynamicznym wariancie problemu mamy na wejściu ciąg zapytań x_i o wyszukanie klucza $k \in [1, n]$. Dla każdego zapytania o wartość w zaczynamy ze wskaźnikiem na korzeń drzewa i wykonując tylko niżej wymienione operacje chcemy przesunąć wskaźnik na wierzchołek zawierający klucz o wartości w. Dozwolone operacje to:

1. Przesunąć wskaźnik na lewego syna obecnie wskazanego wierzchołka
2. Przesunąć wskaźnik na prawego syna obecnie wskazanego wierzchołka
3. Przesunąć wskaźnik na rodzica obecnie wskazanego wierzchołka
4. Wykonać pojedynczą rotację obecnie wskazywanego wierzchołka. Poniżej ilustracja pojedynczej rotacji punku p w prawo

W tym problemie wyszukujemy drzewo, które minimalizuje liczbę operacji koniecznych do obsłużenia ciągu zapytań x_i.

Dla każdego ciągu zapytań istnieje minimalny ciąg operacji odpowiadający na te zapytania. Znalezienie takiego rozwiązania wymaga znajomości całego ciągu zapytań z góry, co w wielu sytuacjach nie jest możliwe. Niech OPT(x_i) będzie minimalną liczbą operacji konieczna do zrealizowania ciągu zapytań x_i, będziemy mówić, że drzewo jest dynamicznie optymalne, jeśli dla każdego ciągu wejść y_i algorytm wykonuje O(OPT(y_i) operacji. Innymi słowy ma stały współczynnik konkurencyjności. Udowodnienie dynamicznej optymalności dowolnego drzewa jest problemem otwartym.



Rysunek 1.1: Pojedyncza rotacja w prawo w punkcie p

Rozdział 2.

Drzewa Splay

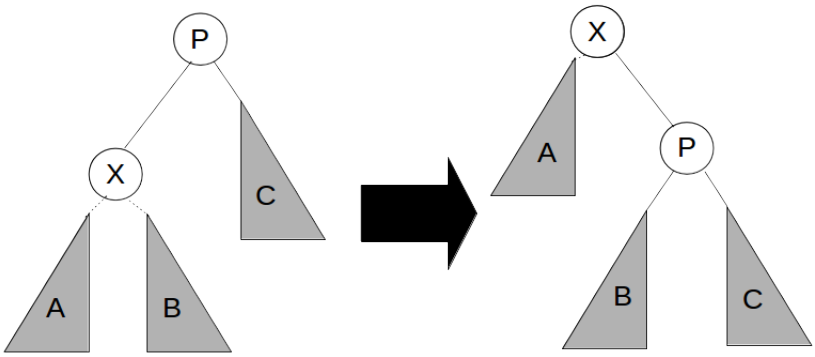
2.1. Opis struktury

Drzewo Splay jest drzewem samoorganizującym się. Podstawią działania tej struktury jest operacja splay, polegająca na przesunięciu odpowiedniego wierzchołka do korzenia, korzystając z rotacji.

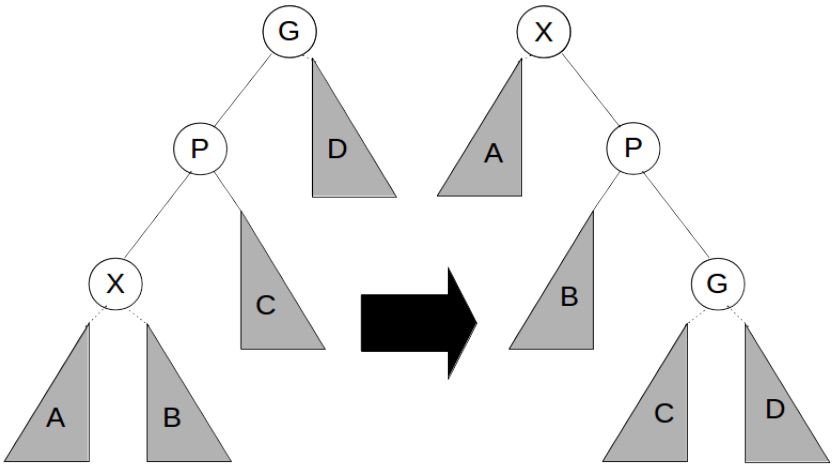
2.1.1. Splay

Operacja splay dla danego wierzchołka odbywa się w krokach, do momentu, kiedy wierzchołek nie stanie się korzeniem. Możemy wyróżnić 3 przypadki kroków operacji splay:

1. ZIG - jeśli x jest synem korzenia wykonujemy pojedynczą rotację w stronę korzenia



Rysunek 2.1: Krok ZIG operacji Splay



Rysunek 2.2: Krok ZIGZIG operacji Splay

2. ZIGZIG - jeśli zarówno x i jego ojciec są lewymi synami wykonujemy u ojca rotację w prawo, a następnie rotujemy w prawo x. Analogicznie postępujemy kiedy x i jego ojciec są prawymi synami.

3. ZIGZAG - jeśli x jest prawym synem, a jego ojciec jest lewym synem, rotujemy x w lewo a następnie w prawo. Analogicznie postępujemy kiedy x jest lewym synem, a jego ojciec jest lewym synem.

2.1.2. Wyszukiwanie

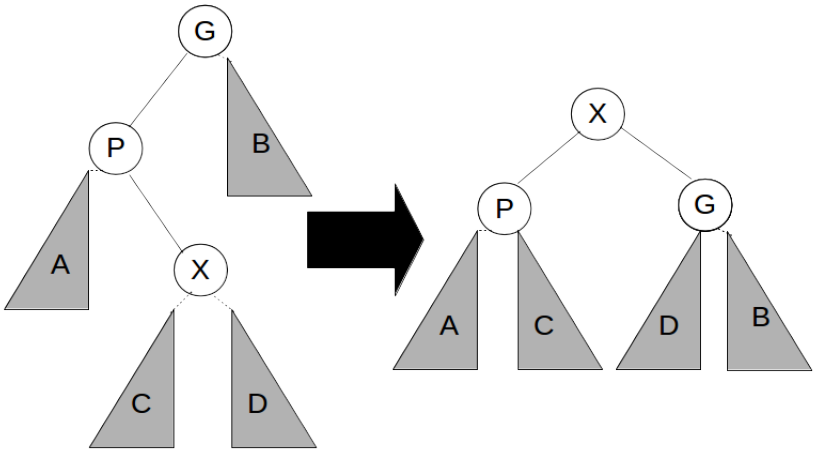
Drzewo splay spełnia zależność BST, więc można wyszukiwać w nim elementu jak w normalnym drzewnie BST. Po wykonaniu znalezieniu odpowiedniego wierzchołka wykonujemy na nim operację splay przenosząc go do korzenia.

2.1.3. Rozdzielanie drzewa względem elementu

Aby rozdzielić drzewo względem danego elementu e, wyszukujemy go w drzewie i wykonujemy operację splay na elemencie, na którym zakończyliśmy poszukiwanie. Wynikiem będą lewe i prawe poddrzewo korzenia.

2.1.4. Łączenie dwóch drzew

Będziemy łaczyć drzewa o własności, że wszystkie elementy jednego drzewa są mniejsze od wszystkich elementów drugiego drzewa. Aby to zrobić wykonujemy



Rysunek 2.3: Krok ZIGZAG operacji Splay

operację splay na wierzchołku zawierającym minimalny element drugiego drzewa. Następnie podłączamy pierwsze drzewo jako lewe poddrzewo.

2.1.5. Wstawianie

Aby wstawić element e do drzewa najpierw wyszukujemy go w drzewie, jeśli element już znajduje się w drzewie to wykonujemy operację splay na odpowiadającym mu wierzchołku. Jeśli elementu nie ma w drzewie, wtedy rozdzielamy drzewo względem e na drzewa d_1 i d_2. Następnie podłączamy d_1 jako lewe poddrzewo elementu e a d_2 jako prawe poddrzewo.

2.1.6. Usuwanie

Aby usunąć element, wyszukujemy go w drzewie i wykonujemy na odpowiadającym wierzchołku operację splay. Następnie łączymy lewę i prawe poddrzewo w jedno drzewo.

2.2. Analiza złożoności

2.2.1. Złożoność pamięciowa

W każdym węźle trzymamy dwa wskaźniki na synów oraz wartość klucza. Nie ma dodatkowego narzutu pamięci. Cała struktura zajmuje pamięć $O(n)$.

2.2.2. Analiza zamortyzowana złożoności czasowej

Twierdzenie 2..1. *Dla ciągu zapytań* $X = \{x_1, x_2, ..., x_m\}$, *gdzie* $\forall_{1 \leq i \leq m} x_i \in \{1, 2, ..., n\}$, *drzewo splay działa w czasie* $O((m+n)logn)$.

Dowód. Zauważmy, że główną składową kosztu operacji słownikowych jest operacja splay. Pozostałe części operacji można wykonać w czasie stałym. Przeprowadzimy analizę zamortyzowaną kosztu wykonywania operacji splay. Wprowadźmy następujące oznaczenia :

- $s(w)$ - liczba wierzchołków w poddrzewie wierzchołka w
- $r(w) = \lfloor log(s(w)) \rfloor$
- Funkcja potencjału $\Phi = \sum_w r(w)$

Przeanalizujmy najpierw zmianę potencjału Φ przy poszczególnych krokach operacji Splay. Będziemy korzystali z notacji na 2.1, 2.2, 2.3

1. **ZIG** : Zmienia się tylko potencjał x i p

$$\begin{aligned} Delta\Phi = & \\ & = 1 + \Delta r(x) + \Delta r(p) \\ & = 1 + r'(x) - r(x) + r'(p) - r(p) \\ & = 1 + r'(p) - r(x) \text{ ponieważ r'(x) = r(p)} \end{aligned}$$

2. **ZIGZIG** : Zmienia się potencjał x, p i g

$$\begin{aligned} \Delta\Phi = & \\ & = 2 + r'(x) - r(x) + r'(p) - r(p) + r'(g) - r(g) & \\ & = 2 - r(x) + r'(p) - r(p) + r(g) & \text{ponieważ r'(x) = r(g)} \\ & \leq 2 + r'(g) + r'(x) - 2r(x) & \text{ponieważ r(x) ¡ r(p) i r'(x) ¿ r(p)} \\ & \leq 3(r'(x) - r(x)) & \text{z wypukłości funkcji logarytmicznej} \end{aligned}$$

3. **ZIGZAG** : Zmienia się potencjał x, p i g

$$\begin{aligned} \Delta\Phi = & \\ & = 2 + r'(x) - r(x) + r'(p) - r(p) + r'(g) - r(g) & \\ & = 2 - r(x) + r'(p) - r(p) + r(g) & \text{ponieważ r'(x) = r(g)} \\ & \leq 2 + r'(g) + r'(p) - 2r(x) & \text{ponieważ r(x) ¡ r(p)} \\ & \leq 2(r'(x) - 2r(x)) & \text{ponieważ 2r'(x) - r'(p) - r'(g) } \geq 2 \end{aligned}$$

We wszystkich przypadkach mamy zatem koszt operacji nie większy niż $3(r'(x) - r(x)) + 1$, gdzie czynnik 1 występuje tylko przy operacji ZIG, a zatem tylko raz przy każdej operacji Splay. Dla całej operacji Splay(x_i) musimy zsumować zmianę potencjału wszytkich kroków. Otrzymujemy w ten sposób sumę teleskopową skracającą się do $3(r(korze) - r(x_i)) + 1 \leq logn$. Aby otrzymać oszacowanie całego czasu działania na całej sekwencji X, musimy jeszcze uwzględnić zmianę potencjału pomiędzy stanem początkowym a końcowym : $\sum_x r'(x) - r(x) \leq \sum_x logn = nlogn$. Zatem sumaryczny czas działania to $O((m+n)logn)$.

□

2.3. Konkurencyjność

Hipoteza 2..2. *Drzewo splay jest dynamicznie optymalne tzn. $SPLAY(X) = O(OPT(X)$ gdzie $SPLAY(X)$ to koszt obsłużenia ciągu zapytań X przez drzewo splay, a $OPT(X)$ przez optymalne dynamiczne drzewo działające offline.*

Rozdział 3.

Drzewo Tango

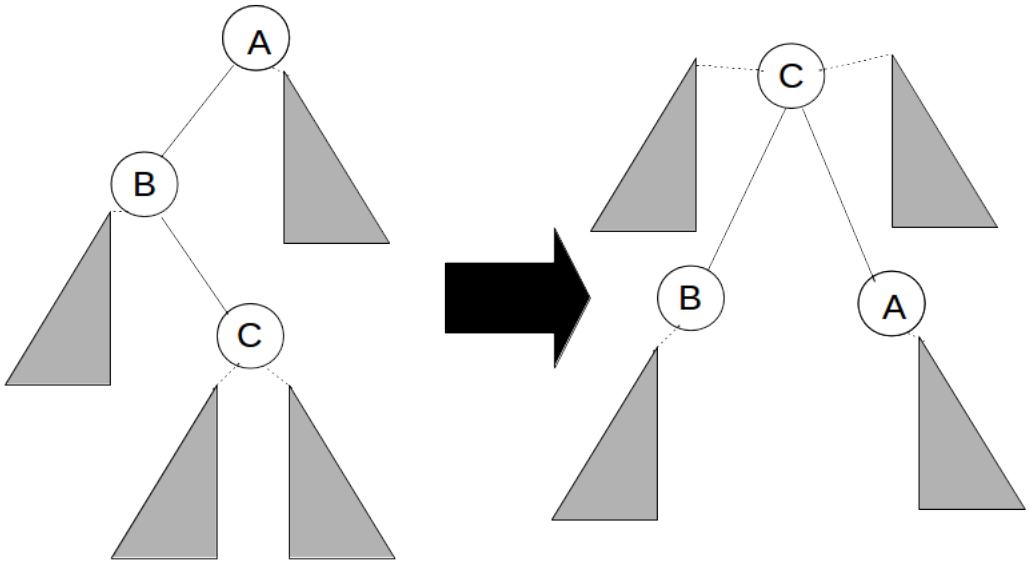
3.1. Opis struktury

Drzewo Tango jest strukturą samoorganizującą, symulującą działanie zbilansowanego drzewa BST, w której wyróżniamy ścieżkę od korzenia do ostatnio odwiedzonego elementu. Wprowadźmy następujące pojęcia.

- Drzewo referencyjne - zbilansowane drzewo BST zawierające wszystkie te same elementy co nasza struktura. Jeśli liczba elementów nie jest potęgą dwójki, w ostatniej warstwie może brakować liści z prawej strony drzewa.
- Preferowana ścieżka wierzchołka– ścieżka prowadząca od wierzchołka (korzenia poddrzewa) do wyróżnionego elementu, w naszym wypadku będzie to najpóźniej żądany element w poddrzewie
- Preferowany syn – syn wierzchołka leżący na preferowanej ścieżce wierzchołka. Jeżeli ostatnim odwiedzonym wierzchołkiem poddrzewa jest korzeń, wtedy korzeń nie ma preferowanego syna
- Krawędź preferowana – krawędź pomiędzy wierzchołkiem a preferowanym synem
- Drzewo pomocnicze – drzewo czerowono-czarne zawierające wierzchołki ścieżki preferowanej
- Nie preferowany syn – syn wierzchołka nie leżący na preferowanej ścieżce
- Nie preferowana krawędź – krawędź prowadząca do nie preferowanego syna
- Głębokość wierzchołka $\mathcal{D}(w)$– odległość wierzchołka w od korzenia drzewa
- Maksymalna głębokość wierzchołka $\mathcal{G}(w)$– maksimum głębokości w poddrzewie wierzchołka w w drzewie pomocniczym

3.1.1. Konstrukcja struktury

Oznaczmy zbiór elementów w naszym drzewie jako $\mathcal{E}$. Analogicznie niech $\mathcal{E}(v)$ będzie zbiorem elementów Rozpatrzmy drzewo BST $\mathcal{B}$ zawierające elementy $\mathcal{E}$. Niech $\mathcal{S}$ będzie zbiorem wierzchołków preferowanej ścieżki w $\mathcal{B}$. Wierzchołki $\mathcal{S}$ umieszczamy na drzewie czerwono czarnym. Nie preferowanych synów podpinamy pod odpowiadających ojców z preferowanej ścieżki dodatkowymi krawędziami. Dodane krawędzie nie są częścią drzewa czerwono czarnego. Powtarzamy procedurę dla poddrzew, w którym korzeniami są nie preferowani synowie wierzchołków z $\mathcal{S}$.



3.1.2. Operacje na drzewie pomocniczym

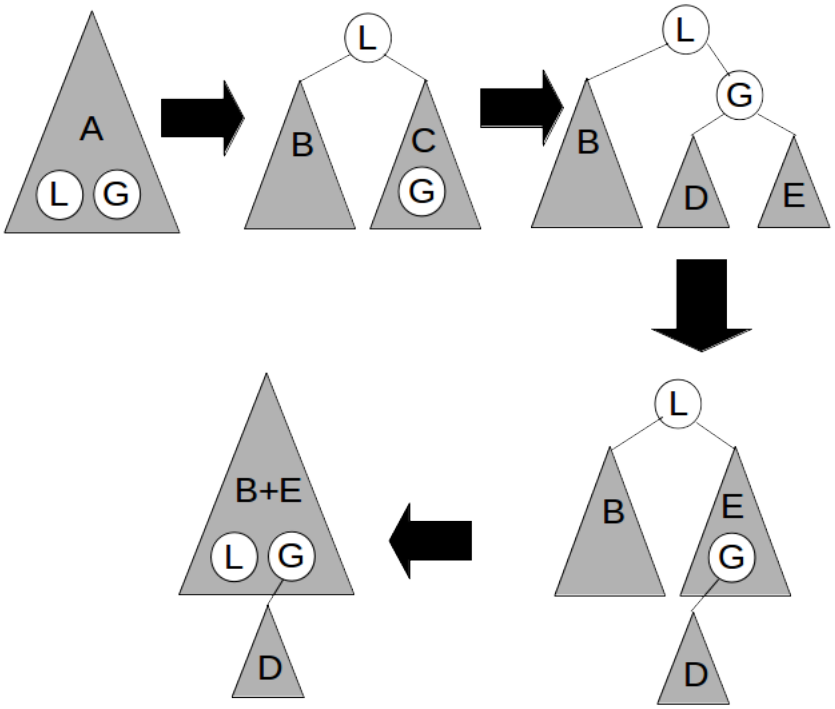
Do zdefiniowania operacji słownikowych na drzewie Tango będziemy korzystali z następujących operacji na drzewie czerwono-czarnym.

1. Split- Rozdzielenie drzewa czerwono-czarnego na dwa osobne zbilansowane drzewa, zawierające klucze mniejsze i większe od danego
2. Join- Połączenie dwóch drzew czerwono-czarnych w jedno

3.1.3. Łączenie i rozdzielanie drzew preferowanych ścieżek

Aby aktualizować preferowane ścieżki, potrzebujemy umieć połączyć dwa drzewa preferowanych ścieżek oraz rozdzielić drzewo preferowanej ścieżki. Do połączenia dwóch drzew wystarczy wykorzystać operację join. Aby rozdzielić ścieżkę w danym wierzchołku w, potrzebujemy znaleźć w drzewie pomocniczym wszystkie wierzchołki o głębokości większej niż $\mathcal{D}(w)$ i zrobić z nich osobne drzewo pomocnicze. Zauważmy,

że istnieje przedział kluczy, dla których wszystkie wierzchołki w drzewie pomocniczym są głębokośći większej niż $\mathcal{D}(w)$. Korzystając z maksymalnej głębokości, możemy znaleźć minimalny i maksymalny klucz o głębokości większej niż $\mathcal{D}(w)$. Następnie przy pomocy dwóch operacji Split wydzielamy dolną część ścieżki do osobnego drzewa pomocniczego i łączymy spowrotem pozostałe klucze przy pomocy operacji Join.



3.1.4. Aktualizacja preferowanych ścieżek

Mając wierzchołek v chcemy zaktualizować drzewo w taki sposób, aby ścieżka do wierzchołka v była ścieżką preferowaną całego drzewa. Niech wierzchołek v będzie częścią preferowanej ścieżki $\mathcal{P}$, jakiegoś poddrzewa drzewa referencyjnego. $\mathcal{P}$ może być pojedynczym wierzchołkiem. Zaczynamy od rozdzielenia $\mathcal{P}$ w wierzchołku v. Preferowany syn wierzchołka, jeżeli istnieje, staje się nie preferowanym synem. Następnie wykonujemy poniższe kroki, tak długo, aż nie dotrzemy do korzenia całego drzewa.

1. Niech v będzie ostatnim elementem preferowanej ścieżki z wierzchołka r. Znajdujemy ojca o wierzchołka r

2. Rozdzielamy ścieżkę preferowaną do której należy o w wierzchołku o. Preferowany syn wierzchołka staje się nie preferowanym.

3. Do górnej części ścieżki preferowanej dołączamy ścieżkę z r do v

3.1.5. Wyszukiwanie

Do wyszukiwania klucza k będziemy symulować wyszukiwanie na drzewie referencyjnym. Najpierw wyszukujemy k w drzewie pomocniczym. Jeśli nie znaleźliśmy klucza k w ścieżce preferowanej, wtedy kontynuujemy wyszukiwanie w poddrzewie niepreferowanego syna ostatniego wierzchołka, na którym zakończyliśmy wyszukiwanie. Na koniec aktualizujemy preferowane ścieżki.

3.2. Analiza złożoności

3.2.1. Złożoność czasowa

Oszacujmy złożoność operacji na drzewach pomocniczych.

Lemat 3..1. *Złożoność dowolnej operacji na drzewie pomocniczym to $O(loglogn)$ gdzie n to liczba elementów w strukturze.*

Dowód. Preferowane ścieżki, są ścieżkami zbilansowanego drzewa binarnego, zatem ich długość jest nie większa niż $\lceil logn \rceil$. Operacje na drzewach pomocniczych są implementowane przy pomocy drzewa czerwono-czarnego i są wykonywane w czasie logarytmicznym od rozmiaru drzewa. Zatem koszt operacji to $O(loglogn)$. □

Następnie oszacujmy koszt pojedynczego dostępu do elementu w drzewie Tango.

Lemat 3..2. *Złożoność pojedynczego dostępu do elementu x_i w drzewie Tango to $O((k+1)(1+loglogn))$, gdzie n to liczba elementów w strukturze, a k liczba nie preferowanych krawędzi na ścieżce z korzenia do x_i.*

Dowód. Koszt dostępu można rozdzielić na koszt wyszukiwania i koszt aktualizacji preferowanych ścieżek w drzewie. Przy wyszukiwaniu odwiedzimy nie więcej niż $k+1$ drzew pomocniczych. Z 3..1 wynika, że wyszukiwanie w drzewie pomocniczym ma złożoność $O(loglogn)$. Zatem sumaryczny koszty wyszukania to $O((k+1)(1+loglogn))$. Analogicznie przy aktualizacji dla każdego odwiedzonego drzewa pomocniczego wykonujemy jedna operację Split i dwie operacje Join, wiec sumaryczny koszt to również$O((k+1)(1+loglogn))$. □

Twierdzenie 3..3. *Pesymistyczny koszt pojedynczego dostępu to $O(log(n)log(log(n)))$.*

3.2.2. Złożoność pamięciowa

Dla każdego wierzchołka będziemy pamiętać maksymalnie dwa wskaźniki na synów w drzewie pomocniczym, dwa wskaźniki na nie preferowanych synów w drzewie referencyjnym, jeden wskaźnik na ojca w drzewie pomocniczym, oraz klucz. Zatem złożoność pamięciowa to $O(n)$.

3.3. Konkurencyjność

3.3.1. Przeplotowe ograniczenie dolne

Zdefiniujemy sobie model do szacowania ograniczenia dolnego na liczbę operacji na optymalnym drzewie binarnym. Stwórzmy na kluczach $x_i \in \{1, ..., n\}$ idealne drzewo binarne B. Jeśli $n \neq 2^k - 1$ wtedy nasze drzewo będzie pełne, a nie idealne. Struktura drzewa nie będzie się zmieniać w czasie.

Definicja 3..4 (Lewy i prawy obszar)**.** *Dla danego wierzchołka y i drzewa B jego lewym obszarem $L(y)$ będzie lewe poddrzewo y oraz y, a prawy obszar $P(y)$ to prawe poddrzewo y.*

Przy każdym zapytaniu x_i dla wszystkich wierzchołków na ścieżce od korzenia do x_i odnotowujemy czy x_i znajduje się w jego prawym czy lewym poddrzewie.

Definicja 3..5 (Przeplot)**.** *Dla danego wierzchołka y, drzewa B i ciągu zapytań X, przeplotem nazwiemy parę zapytań (x_i, x_j) spełniającą następujące warunki :*

1. $(x_i \in P(y) \wedge x_j \in L(y)) \vee (x_i \in L(y) \wedge x_j \in P(y))$
2. $\neg\exists_{k\in[i,j]}(x_k \in L(y) \vee x_k \in P(y)$

Zdefiniujmy, także funkcję $IB(X, i)$ jako liczbę przeplotów wprowadzonych przez i-te zapytanie, w ciągu zapytań X, a $IB(X) = \sum_i IB(X, i)$.

Będziemy starali oszacować oszacować liczbę operacji w optymalnym algorytmie BST, korzystając z liczby przeplotów. Niech T_i będzie stanem drzewa binarnego po wykonaniu zapytań $x_1, x_2, ..., x_i$, przez dowolny deterministyczny algorytm wyszukiwania i reorganizacji.

Definicja 3..6. *Punkt przejściowy wierzchołka y w momencie i to wierzchołek z , o najmniejszej głębokości w T_i, taki że ścieżka z korzenia T_i przechodzi przez jakiś wierzchołek zarówno z $L(y)$ jaki i $P(y)$ w pełnym drzewie B.*

Lemat 3..7. *Dla każdego wierzchołka y i momentu i istnieje dokładnie jeden punkt przejściowy.*

Dowód. Niech l i r będą najniższymi wspólnymi przodkami odpowiednio wierzchołków z $L(y)$ i z $P(y)$ w T_i . Ponieważ klucze w lewym obszarze stanowią spójny przedział posortowanego ciągu wszystkich kluczy, a w drzewie BST wspólny przodek dwóch wierzchołków ma wartość klucza pomiędzy wartościami synów, w takim razie l jest elementem lewego poddrzewa. Analogicznie r jest elementem prawego poddrzewa. Zauważmy również, że klucze całego poddrzewa y w B stanowią spójny

przedział posortowanego ciągu wszystkich kluczy, zatem najniższy wspólny przodek należy albo do lewego albo do prawego obszaru. Z tego wynika, że l lub r jest najniższym wspólnym przodkiem całego poddrzewa. Załóżmy bez straty ogólności, że jest to l. Zauważmy, że r będzie punktem przejściowym dla y w momencie i. Z definicji, najniższego wspólnego przodka, jest jedynym wierzchołkiem o najmniejszej głębokości, którego ścieżka przechodzi przez wierzchołek z $R(y)$. Ścieżka do r przechodzi też przez l, które jest elementem $L(y)$. □

Lemat 3..8. *Jeśli w momencie i z był punktem przejściowym wierzchołka y i algorytm BST obsługując zapytania $x_j, ..., x_k$ nie przechodzi przez z, ani nie wykonuje rotacji, która by zmniejszyła głębokość z, to z jest punktem przejściowym w każdym momencie z przedziału $[j, k]$.*

Dowód. Zdefiniujmy l i r jak w poprzednim dowodzie i załóżmy bez straty ogólności, że l jest wspólnym przodkiem wyszystkich wierzchołków z $L(y)$ i $P(y)$ w momencie j. Zatem r jest punktem przejsciowym y w momencie j. Skoro algorytm BST nie przechodzi w zapytaniach przez r to r pozostaje najniższym wspólnym przodkiem wierzchołków $P(Y)$ w T_i. Dodatkowo, skoro nie została wykonana, żadna rotacja zmniejszająca głębokość r to poddrzewo r w T_i pozostaje niezmienne. W czasie wykonywiania zapytań $x_j, ..., x_k$, żaden element z $L(y)$ nie stał się synem r, zatem jakiś element $L(Y)$ musi być wspólnym przodkiem wszystkich wierzchołków z $L(y)$ i $P(y)$. Z tego wynika, że r pozostaje punktem przejściowym y. □

Lemat 3..9. *Każdy wierzchołek z może być w momencie i punktem przejściowym co najwyżej jednego wierzchołka y.*

Dowód. Weźmy dowolne dwa wierzchołki y_1 i y_2, pokażemy, że ich punkty przejściowe w momencie i są różne. Zdefiniujmy dla nich podobnie jak w dowodach poprzednich lematów odpowiednio l_1, r_1 i l_2, r_2. Jeśli y_1 i y_2 nie znajdują się na jednej ścieżce do korzenia (żaden nie jest przodkiem drugiego), to ich lewe i prawe obszary są rozłączne, a zatem ich punkty przejściowe muszą być różne. Jeśli tak nie jest to załóżmy bez straty ogólności, że y_1, jest przodkiem y_2. Jeśli punkt przejściowy y_1 znajduje się w innym obszarze y_1 niż y_2, wtedy punkty przejściowe muszą być różne. W przeciwnym wypadku punkt przejściowy y_1 musi być wspólnym przodkiem elementów z $L(y_2)$ i $R(y_2)$. Zatem głębokość punktu przejściowego y_1 będzie nie większa niż głębokość l_2 i r_2. Z kolei punkt przejściowy y_2 ma głębokość nie mniejszą niż głebokości l_2 i r_2. Skoro l_2 i r_2 muszą mieć różne głębokości, to punkty przejściowe y_1 i y_2 muszą być różne. □

Twierdzenie 3..10. *$OPT(X) \geq \frac{IB(X)}{2} - n$, gdzie OPT to koszt działania optymalnego algorytmu BST na ciagu zapytań X.*

Dowód. Bedziemy szacować z dołu liczbę operacji wykonywanych przez optymalny algorytm przez liczbę punktów przejściowych odwiedzonych (przechodzi przez nie

ścieżka z korzenia do x_i lub wykonuje się na nich rotacje zmniejszające ich głębokość) w trakcie wykonywania zapytań. Korzystając z 3..9 będziemy zliczać te wydarzenia dla każdego wierzchołka y osobno, a następnie je zsumujemy. Niech $x_{a_1}, xa_2, ..., xa_k$ będzie maksymalnym podciągiem zapytań, w którym każde dwa kolejne zapytania są o elementy w różnych obszarach y. p będzie liczbą przeplotów w y. Załóżmy bez straty ogólności, że zapytania $x_{a_{2i}}$ będą zapytaniami a elementy lewego obszary, $x_{a_{2i+1}}$ prawego. Rozpatrzmy l i r takie jak w poprzednich dowodach. Wszystkie zapytania o prawy obszar y odwiedzają r, a o lewy odwiedzają l. Zatem przy dwóch kolejnych zapytaniach, albo algorytm albo będzie przechodził przez punkt przejściowy przy wyszukiwaniu, albo punkt przejściowy się zmieni, co również wymaga odwiedzienia punktu przejsciowego. Zatem dla dwóch kolejnych zapytań z naszego poddciągu algorytm przynajmniej raz odwiedza punkt przejściowy y. Dla danego wierzchoła otrzymujemy odwiedzamy $\lfloor \frac{p}{2} \rfloor \geq \frac{p}{2} - 1$. Sumując po wszystkich y otrzymyjemy $OPT(X) \geq \frac{IB(X)}{2} - n$. □

3.3.2. Zastosowanie przeplotów w oszacowaniu konkurencyjności

Lemat 3..11. *Liczba nie preferowanych krawędzi na ścieżce z korzenia do x_i jest równa $IB(X, i)$.*

Dowód. Obecność niepreferowanej krawędzi $\{a, b\}$ na ścieżce oznacza, że ostatnie zapytanie o element w poddrzewie wierzchołka y w innym poddrzewie niż x_i. To odpowiada przeplotowi. □

Twierdzenie 3..12. *Dla ciągu zapytań $X = \{x_1, x_2, ..., x_m\}$, takich że $\forall_i x_i \in \{1, 2, ..., n\}$ sumaryczny koszt działania drzewa Tango to $O((OPT(X)+n)(1+loglogn))$, gdzie $OPT(X)$ to koszt optymalnego dynamicznego drzewa BST działającego offline.*

Dowód. Z 3..11 i 3..2 wynika, że koszt obsługi pojedynczego zapytania na drzewie Tango to $O((IB(X, i) + 1)(1 + loglogn))$.Dodatkowo, dla każdego wierzchołka co najwyżej raz będziemy zmieniać jego stan z nie posiadania preferowanego syna na posiadanie jednego. Zatem sumaryczny koszt zapytania to $O((IB(X) + m + n)(1 + loglogn))$. Z 3..10 wiemy, że $OPT(X) \geq IB(X) - n$. Oczywiste też jest, że $OPT(X) \geq m$. Z tego wynika, że koszt algorytmu to $O(OPT(X) + n)(1 + loglogn))$. □

Bibliografia

[1] Daniel Dominic Sleator, Robert Endre Tarjan *Self-Adjusting Binary Search Trees*. Journal of the Association for Computing Machinery, Vol. 32, No. 3, July 1985.

[2] Erik D. Demaine , Dion Harmon , John Iacono , And Mihai Patrascu *Dynamic Optimality—Almost*. SIAM Journal on Computing, 2007, Vol. 37, No. 1 : pp. 240-251

[3] Jonathan Derryberry, Daniel Dominic Sleator, Chengwen Chris Wang, *A Lower Bound Framework for Binary Search Trees with Rotations*. November 2005.

[4] Thomas H. Cormen, Charles E. Leiseron, Ronald R. Riverst *Introduction to algorithms*. 1990.

[5] Prosenjit Bose, Karim Douïeb,Vida Dujmović, Rolf Fagerberg, *An O(log log n)-Competitive Binary Search Tree with Optimal Worst-Case Access Times*. February 2010