Zaawansowane struktury danych Zima 2010

vEB, x-fast & y-fast trees — 18 października, 2010

Paweł Rzechonek *Aleksander Balicki, Łukasz Zapart*

1 Problem następnika

Problem następnika polega na utrzymaniu zbioru S złożonego z n liczb z uporządkowanego uniwersum $\mathcal{U} = \{1, 2, \cdots, u\}$, na którym można wykonywać standardowo zdefiniowane operacje INSERT, UPDATE, oraz SUCCESSOR. Warto wspomnieć, że SUCCESSOR może jako parametr brać dowolny element $x \in \mathcal{U}$, niekoniecznie $x \in S$.

Należy sensownie ograniczyć pamięć zajmowaną przez strukturę danych – łatwo można przechowywać wszystkie możliwe odpowiedzi na pytania w pamięci rzędu $\Theta(2^u)$. Z tego powodu będziemy się zajmować tylko algorytmami zajmującymi $O(u \log^{O(1)} u)$ pamięci.

Klasycznie problem rozwiązuje się używając zbalansowanych binarnych drzew poszukiwań. Tym sposobem można uzyskać $O(\log n)$ dla wszystkich operacji.

Drzewa binarne nie wykorzystują jednak założenia o skończonym uniwersum. Pokażemy że można wykorzystać to założenie do zdefiniowania struktur danych wykonujących operacje w czasie $o(\log n)$.

2 Drzewa van Emde Boasa

Idea: Przeszukiwać binarnie bity liczby.

Niech $\text{high}(x)$ i $\text{low}(x)$ będą odpowiednio lewą i prawą połową bitów liczby w zapisie binarnym. Na przykład $x = 1100100011$, $\text{high}(x) = 11001$, $\text{low}(x) = 00011$. Zauważmy, że $x = \text{high}(x) \cdot \sqrt{u'} + \text{low}(x)$, gdzie u' jest u zaokrąglonym w górę do potęgi dwójki.

Drzewo vEB uniwersum rozmiaru u będzie składało się z:

- $\sqrt{u}$ poddrzew vEB rozmiaru $\sqrt{u}$: $S[0], S[1], \cdots, S[\sqrt{u}-1]$. Każde poddrzewo zawiera przedział długości $\sqrt{u}$. Klucz x należy do drzewa wtedy i tylko wtedy, gdy $\text{low}(x)$ należy do $S[\text{high}(x)]$.
- Jedno poddrzewo vEB rozmiaru $\sqrt{u}$: $S.summary$. Dla każdego i takiego, że $S[i]$ jest niepuste wkładamy i do $S.summary$.
- $S.min$ – minimum zbioru, trzymane niezależnie od poddrzew – to znaczy element nie może być jednocześnie $S.min$ i być w jakimś poddrzewie. Zauważmy, że żeby sprawdzić czy drzewo jest puste wystarczy sprawdzić czy $S.min$ jest puste.
- $S.max$ – maximum zbioru. W przeciwieństwie do minimum, maximum jest również trzymane w odpowiednim poddrzewie.
- operacji $S.successor$ i $S.insert$.

Algorithm 1 S.successor(x)

```
if x < S.min then
  return S.min
end if
if low(x) < S[high(x)].max then
  return high(x) ·√u + S[high(x)].successor(low(x))
else
  i = S.summary.successor(high(x))
  return i ·√u + S[i].min
end if
```

Algorithm 2 S.insert(x)

```
if S.min = ∅ then
  S.min = S.max = x
  return
end if
if x > S.max then
  S.max = x
end if
if x < S.min then
  swap(x, S.min)
end if
if S[high(x)].min = ∅ then
  S[high(x)].min = S[high(x)].max = low(x)
  S.summary.insert(high(x))
else
  S[high(x)].insert(low(x))
end if
```

Czas działania INSERT oraz SUCCESSOR ma równanie rekurencyjne

$$T(u) = T(\sqrt{u}) + O(1) \tag{1}$$

które ma rozwiązanie $O(\log \log u)$. Kluczowym jest osiągniecie tylko jednego wywołania rekurencyjnego poprzez trzymanie minimum osobno. Gdyby minimum było trzymane rekurencyjnie, INSERT potrzebowałby dwóch rekurencyjnych INSERTÓW – jednego do poddrzewa i jednego do $S.summary$, a wtedy złożoność czasowa byłaby $O(\log u)$. Dzięki trzymaniu minimum osobno INSERT do pustego drzewa jest trywialny.

2.1 Zmniejszenie zużycia pamięci

Problemem jest zużycie pamięci $O(u)$ – u może być bardzo duże. Żeby zaradzić temu problemowi wykorzystujemy tu dynamiczne tablice hashujące. Rozmiar takiej tablicy jest ograniczony przez ilość jej elementów, więc możemy zmniejszyć rozmiar vEB tree do O(n), gdy tablice SUMMARY i SUB będa tablicami hashującymi.

3 Hashowanie doskonałe

Doskonała funkcja hashująca h przeprowadza zbiór kluczy M o mocy $m = |M|$ na zbiór N o mocy $n = |N|$, bez kolizji, tzn $\forall_{x,y \in M} h(x) \neq h(y)$.

3.1 Przypadek statyczny

W przypadku statycznym mamy daną całą zawartość tablicy hashującej T na początku algorytmu. Musimy stworzyć taką strukturę, w której odpowiednio szybko będzie się wykonywać operację QUERY.

Okazuje się, że możemy zagwarantować O(1) w najgorszym przypadku na QUERY. Wykorzystujemy w tym celu hashowanie dwupoziomowe. Na początku bierzemy sobie funkcję $f : M \Rightarrow N$, która jest 2-uniwersalna, tj. $\forall_{x,y \in M} P(h(x) = h(y)) = \frac{1}{m}$. Ona przeprowadza nam zbiór kluczy na tzw. "kubełki". Niektóre kubełki pozostaja puste, a niektóre mają więcej elementów (z wysokim prawdopodobieństwem możemy stwierdzić, że nie będzie ich powyżej $O(\frac{\log n}{\log \log n})$). Dla każdego niepustego kubełka o indeksie i tworzymy nową funkcję (też 2-uniwersalną) $f_i : M_i \Rightarrow N_i$, gdzie M_i, to elementy zhashowane do kubełka i przez funkcję f, a $|N_i| = |M_i|^2$. Potrafimy przy takim założeniu, z prawdopodobieństwiem większym od $\frac{1}{2}$ wylosować funkcję, która będzie doskonała (nie będzie miała kolizji dla tego kubełka). Losujemy dopóki nie znajdziemy funkcji, która nie koliduje.

Taka konstrukcja gwarantuje nam czas wykonania QUERY$O(1)$ i ograniczenie na pamięć $O(n)$. Oczekiwany czas wylosowania funkcji dla kubełka jest stały, ale w najgorszym przypadku może zająć więcej czasu. Nie bierzemy jednak czasu konstrukcji słownika (zakładamy, że kiedyś znajdzie odpowiednie funkcje).

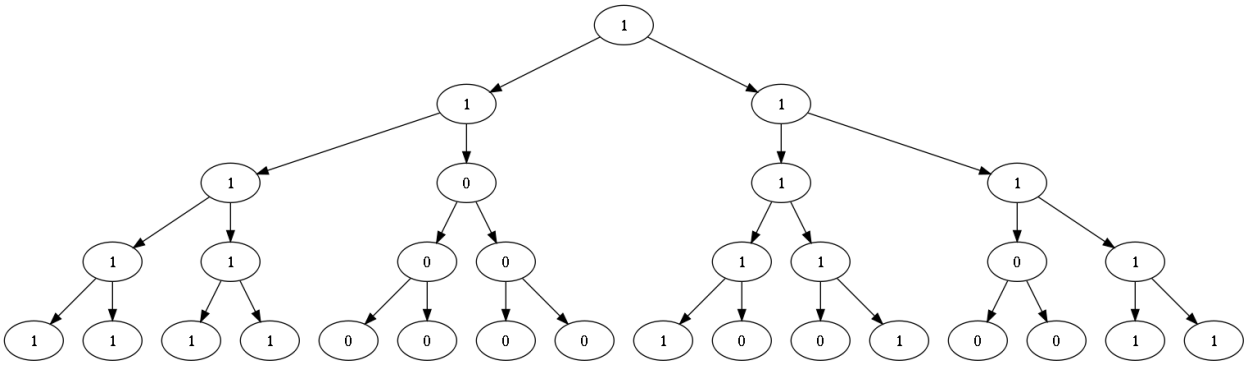
3.2 Przypadek dynamiczny

Ten przypadek jest podobny do przypadku statycznego, lecz mamy jeszcze operacje INSERT, UPDATE i DELETE, dla których też chcemy dobry czas wykonania. Tu już nie możemy dla nich zagwarantować czasu ściśle stałego.

Tworzymy tablice hashującą tak jak w przypadku statycznym, tylko tym razem $|N| = 2n$. Gdy nasza tablica rozrasta sie do $2n$ elementów, wtedy tworzymy 2 razy większą tablice hashującą, kopiując zawartość starej. Analogicznie, gdy ilość elementów zmniejsza się do $0.25n$, zmniejszamy dwukrotnie rozmiar tablicy. Zawsze po rozszerzeniu/zwężeniu do rozmiaru $2k$, tablica zawiera k elementów, czyli trzeba dodać lub usunąć $O(k)$, aby dokonać kolejnej zmiany rozmiaru – to nam amortyzuje czas operacji zmiany rozmiaru do $O(1)$. Czas staje się też oczekiwany, bo wylosowanie funkcji jest teraz wliczane w czas operacji zmiany rozmiaru.

4 Drzewa x-fast

Drzewa x-fast są kolejną strukturą danych, pomocną przy problemie następnika/poprzednika. Korzystają one z nowoprzypomnianej dynamicznej tablicy hashującej.



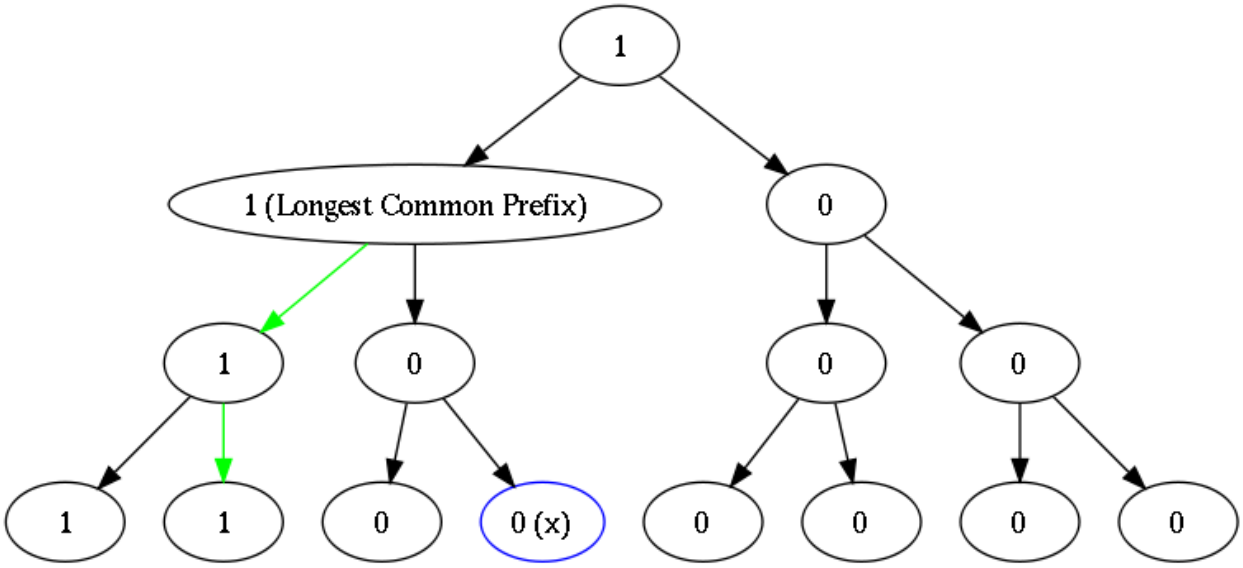
Rysunek 1: Przykład drzewa x-fast.

4.1 Konstrukcja

Drzewo x-fast ma w liściach wszystkie elementu uniwersum U o mocy $u = |U|$. Oznaczamy liście, które należą do naszego zbioru S jedynkami, a pozostałe zerami. Całe drzewo, jest drzewem OR, tzn. jeżeli któryś z synów ma jedynkę, to jego ojciec też ma jedynkę. Drzewo to jest wirtualne, nie trzymamy go w postaci drzewa w pamięci, za to trzymamy tablice hashującą, której kluczami są binarne prefiksy odpowiadające wierzchołkom w drzewie. Ponad to utrzymujemy też dwustronną listę elementów ze zbioru S.

4.2 Złożoność

- PRED



Rysunek 2: Działanie pred na drzewie x-fast.

Wyszukujemy binarnie najdłuższy wspólny podciąg jakiegoś elementu z S i naszego szukanego elementu x i znajdujemy wierzchołek v (wierzcholek symbolizuje najdłuższy wspólny prefiks) – czas $O(\log \log u)$. Jeżeli x jest w lewym poddrzewie v, to wchodzimy w prawe poddrzewo i schodzimy ścieżką jedynek jak najbardziej w lewo, znajdujemy wtedy następnik x. Jeżeli szukaliśmy poprzednika, to korzystamy z listy, żeby w $O(1)$ znaleść poprzednik (element przed nastepnikiem w liście). Analogiczna sytuacja, gdy x jest w prawym poddrzewie v, schodzimy w lewe poddrzewo i potem maksymalnie w prawo i znajdujemy poprzednik – czas $O(\log u)$, ale ta operacja jest deterministyczna dla każdego wierzchołka, więc można ją przeliczać wcześniej i trzymać te informacje w tablicy hashującej, co daje nam w rezultacie czas QUERY$O(\log \log u)$.

- UPDATE

  Dodajemy element do zbioru S, musimy wtedy dodać najwyżej $\log u$ prefiksów do tablicy hashującej – czas $O(\log u)$.

- SPACE

  Dla każdego z n elementów z S trzymamy wszystkie binarne prefiksy – pamięć $O(n \log u)$.

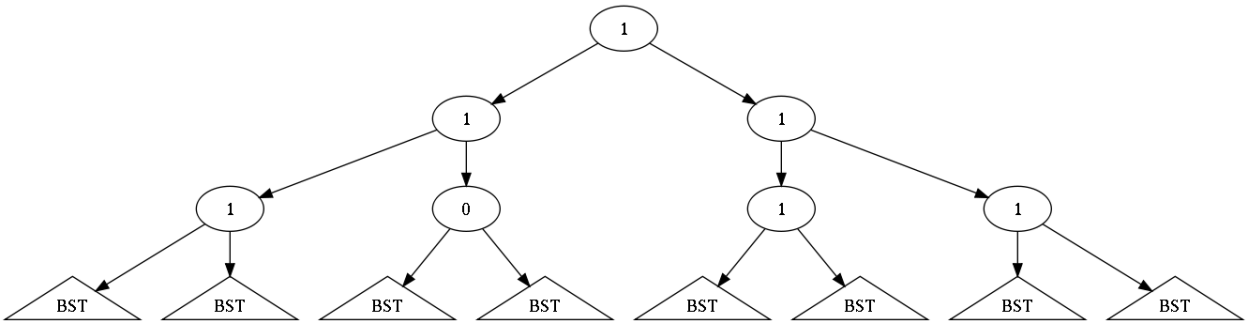
5 Drzewa y-fast

Drzewa y-fast są rozwinięciem drzew x-fast, poprzez dodanie pośredniości (ang. indirection).

5.1 Konstrukcja

Teraz nie trzymamy wszystkich elementów uniwersum w liściach drzewa, trzymamy za to $\frac{n}{\log u}$ zbalansowanych drzew BST (AVL lub RBT) o rozmiarze $O(\log u)$. Dla każdego z trzymanych

drzew wybieramy reprezentanta. Reprezentanci mają oddzielać wartościami poszczególne drzewa (wartości w ramach różnych drzew się nie nakładają).



Rysunek 3: Przykład drzewa y-fast.

5.2 Złożoność

- PRED

  Wyszukujemy binarnie po kluczach w tablicy, tak jak w drzewie x-fast. Zamiast następnika/poprzednika znajdujemy 2 drzewa, w których jest ewentualny poprzednik/następnik. Drzewa te mają rozmiar $O(\log u)$, czyli wyszukiwanie binarne w jednym takim, to $O(\log \log u)$, co jest też złożonością całej funkcji.

- UPDATE

  Wyszukujemy drzewo do którego mamy wstawić nasz element i wstawiamy go tam – obie operacje po $O(\log \log u)$. Problem tworzy się, gdy któreś drzewo zbytnio się rozrasta lub maleje. Jeżeli nasze drzewo urośnie do rozmiaru $2n \log u$, to rozbijamy je na 2 mniejsze drzewa i wybieramy nowych reprezentantów – czas $O(\log u)$. Jeśli zmaleje do $\frac{n}{4} \log u$, to łączymy drzewo z jego bocznym drzewem (potem ewentualnie rozłączamy znowu). Widać, żę po zmianie rozmiaru, ilość elementów w drzewie będzie pomiędzy $(\frac{2}{3} \log u; \frac{3}{2} \log u)$. Czyli do następnego rozłączenia upłynie jeszcze $O(\log u)$ kroków, czyli amortyzowany koszt to $O(1)$.

- SPACE

  Wszystkie prefiksy dla $\frac{n}{\log u}$ drzew, czyli $O(n)$ oraz $\frac{n}{\log u}$ drzew o rozmiarze $\log u$ daje w sumie $O(n)$.

5.3 Liczenie najmniejszego bitu ustawionego na 1

Problem: Dla słowa w policzyć indeks najmniejszego bitu ustawionego na 1.

Rozwiązania: Wszystkie rozwiązania poniżej działają w czasie $O(1)$.

- Omnipotentna RAM która ma taką instrukcję.
- Procesory Pentium, Cray, etc. posiadają taką instrukcję.

- Wyliczenie $a = (w \text{ AND } (w-1)) \text{ XOR } w$, w którym zostaje jedynie najmniejszy bit. Zamiana tego na wartość binarną jest możliwa w czasie $O(1)$.
- Lookup table.

5.4 Model RAMBO

RAMBO[5] (Random Access Machine with Bit Overlap) to model obliczeń w którym są słowa długości $\geqslant \log u$ bitów (innymi słowy na tyle duże by trzymać wskaźniki do danych), które mogą być manipulowane w czasie $O(1)$. W przeciwieństwie do zwykłej RAM, w RAMBO słowa mogą się częściowo pokrywać, t.j. bity mogą być współdzielone przez różne słowa.

5.5 Drzewa RAMBO vEB

Brodnik, Carlsson, Karlsson, and Munro pokazali[6], że jest możliwe zaimplementowanie drzew van Emde Boasa z operacjami w czasie $O(1)$ używając RAMBO. Potrzebne jest $2u$ bitów, jeden dla każdego węzła w pełnym drzewie binarnym. Słowa to drogi z korzenia do liścia. Ustawiamy bity na 1 w tych węzłach, które są przodkami elementu w zbiorze.

Dla pewnego zapytania $x \in \mathcal{U}$ możemy policzyć najmniejszego przodka ustawionego na 1 w czasie $O(1)$. By dostać następnik, moglibyśmy trzymać wskaźniki w każdym węźle na minimum i maksimum elementów w poddrzewie, którego korzeniem jest ten węzeł. Ale wtedy każdy liść mógłby mieć do $2 \log u$ wskaźników do siebie, i INSERT oraz DELETE wymagały by $\Omega(\log u)$ zmian wskaźników.

Będziemy zatem trzymać lewe i prawe wskaźniki w węzłach, które mają inne wskaźniki niż ich rodzic. Dokładniej, trzymamy wskaźniki w każdym *skaczącym* węźle, to znaczy takim, który ma elementy w obu poddrzewach. Każdy skaczący węzęł trzyma wskaźnik na maksimum w lewym poddrzewie i minimum w prawym poddrzewie.

Teraz poprzednik i następnik elementu x uniwersum można policzyć używając wskaźników w najmniejszym skaczącym przodku. Ta struktura zajmuje $2u$ bitów oraz $2n$ słów RAM-u na wskaźniki. Zużycie pamięci można dodatkowo zmniejszyć do u, zauważając, że liście nie potrzebują trzymać bitów.

6 Podsumowanie struktur danych

	struktura danych	czas/op.	rozmiar	model
1962	drzewa zbalansowane	$O(\log n)$	$O(n)$	BST
1975	drzewa vEB [1]	$O(\log w) = O(\log \log u)$	$O(u)$	word RAM
1984	drzewa x-fast [2]	$O(\log u)$	$O(n \log u)$	word RAM
1984	drzewa y-fast [2]	$O(\log w)$ z dużym ppb.	$O(n)$	word RAM
1989	drzewa RAMBO vEB [6]	$O(1)$	$O(u)$	RAMBO
1993	drzewa fusion [3]	$O(\log_w n) = O\left(\frac{\log n}{\log \log u}\right)$	$O(n)$	word RAM

7 Ograniczenia dolne

7.1 Model obliczeń

Model obliczeń, w którym będziemy udowadniać ograniczenia dolne to *model cell-probe*, w którym pamięć jest podzielona na słowa wielkości w, gdzie w jest parametrem modelu. Czytanie z i pisanie do komórek pamięci kosztuje 1 jednostkę; wszystkie inne operacje są za darmo.

Ten model jest słaby i mało realistycznie oddaje współczesne procesory, ale nadaje się świetnie do dowodzenia twierdzeń o ograniczeniach dolnych – wszystkie zachodzą również w modelu RAM.

7.2 Wyniki

- Ajtai 1988[7] – Dowód pierwszego nietrywialnego ograniczenia dolnego $\omega(1)$ – $\Omega(\sqrt{\log w})$.
- Miltersen 1994[8] – Dowód tego samego wyniku używając złożoności komunikacyjnej oraz $\Omega(\sqrt[3]{\log n})$.
- Miltersen, Nisan, Safra, Wigderson 1995[9] & JCSS 1998[10] – Wprowadzenie techniki eliminacji rund, czystszy dowód tego samego ograniczenia.
- Beame, Fich 1999[11] & JCSS2002[12] & manuskrypt 1994 – Dowód mocnych ograniczeń $\Omega(\frac{\log w}{\log\log w})$ oraz $\Omega(\sqrt{\frac{\log n}{\log\log n}})$. Autorzy dodatkowo zaprezentowali statyczną strukturę osiągającą $O(\min\{\frac{\log w}{\log\log w}, \sqrt{\frac{\log n}{\log\log n}}\})$, co pokazuje że te ograniczenia są optymalne biorąc pod uwagę tylko n i w.
- Xiao - praca doktorska 1992 na U.C. San Diego[13] – Niezależny, wcześniejszy, dowód tego samego ograniczenia co Beame i Fich.
- Sen - CCC 2003[14]; Sen, Venkatesh - JCSS2008[15] – Silniejsza wersja lematu eleminacji rund, dająca czystszy dowód.
- Patrascu, Thorup - STOC 2006[16]; SODA2007[17] – Wąskie ograniczenia dla $a = \log \frac{s}{n}$, gdzie s to użyta pamięć:

$$\Theta(min\{\log_w n, \log(\frac{w - \log n}{a}), \frac{\log \frac{w}{a}}{\log(\frac{a}{\log n}\log \frac{w}{a})}, \frac{\log \frac{w}{a}}{\log(\log \frac{w}{a} / \log \frac{\log n}{a})}\}) \tag{2}$$

  Ten kompromis pomiędzy n, w i s pokazuje, że biorąc pod uwagę tylko struktury danych używające $O(n \log^{O(1)} n)$ pamięci, optymalny czas operacji to $\Theta(min\{\log_w n, \frac{\log w}{\log \frac{\log w}{\log\lg n}}\})$. To implikuje że drzewa van Emde Boasa są optymalne jeśli $w = O(\log n)$, natomiast drzewa fusion są optymalne jeśli $\log w = \Omega(\sqrt{\log n}\log\log n)$

Przedstawione wyniki są właściwie dla prostszego problemu – *koloru przodka*. Każdy element jest czerwony lub niebieski, zapytanie na elemencie x zwraca kolor przodka x. Ponieważ można rozwiązać problem koloru przodka używając operacji PREDECESSOR, dolne ograniczenie dla koloru przodka da dolne ograniczenie dla problemu przodka.

7.3 Model złożoności komunikacyjnej

Rozpatrzymy problem w modelu złożoności komikacyjnej. Niech Alicja reprezentuje algorytm, a Bob pamięć. Dane wejściowe Alicji to zapytanie x; Boba struktura danych y. Alicja i Bob mogą komunikować się wysyłając sobie wiadomości o wielkości nie większej niż odpowiednio a i b. Założymy, że $a = O(\log n)$, by było możliwe adresowanie całej struktury danych (pamiętajmy o założeniu, że struktura danych ma nie przekraczać wielomianowej wielkości), i $b = w$, tak by Bob zwracał jedno słowo pamięci. Celem jest policzenie pewnej funkcji $f(x, y)$. W naszym przypadku funkcja f to problem koloru poprzednika. Parametrem, który nas interesuje jest liczba wiadomości wymienionych pomiędzy Alicją i Bobem. Jest ona niewiększa niż dwukrotność liczby zapytań w modelu $cell - probe$. Zauważmy, że model ten jest bardzo silny – Alicja i Bob mogą wykonywać dowolne obliczenia.

7.3.1 Ograniczenie dolne problemu następnika

Udowodnimy ograniczenie dolne $\Omega(\min\{\log_a w, \log_b n\})$ na ilość wysyłanych wiadomości w grze komunikacyjnej. Z tego można wyprowadzić ograniczenie dane przez Beame'a i Fich'a. Mamy $a = \Theta(\log n)$, tzn. użyta pamięć to $n^{O(1)}$, oraz $b = w$. Ograniczenie dolne jest najgorsze (najmniejsze) gdy:

$$\log_a w = \log_b n \Rightarrow \frac{\log w}{\log\log n} = \frac{\log n}{\log w} \Rightarrow \log^2 w = \log n \log\log n \tag{3}$$

W stosunku do n, $\log w = \sqrt{\log n \log\log n}$, więc ograniczenie wynosi $\log_a w = \sqrt{\frac{\log n}{\log logn}}$. W stosunku do w, $\log\log w = \Theta(\log\log n)$, więc ograniczenie wynosi $\log_b n = \frac{\log w}{\log\log w}$.

7.3.2 Eliminacja rund

Eliminacja rund może być użyta w dowolnej grze komunikacyjnej, niekoniecznie związanej z problemem następnika. Daje ona warunki dla których pierwsza runda komunikacji może być wyeliminowana.

Definicja 1. *Niech $f^{(k)}$ będzie wariantem f, w którym Alicja ma k wejściowych danych $x_1, \cdots, x_k$ a Bob ma wejścia $y, i \in \{1, \cdots k\}$ oraz $x_1, \cdots, x_{i-1}$. Celem jest policzenie $f(x_i, y)$.*

Załóżmy, że Alicja musi wysłać pierwszą wiadomość. Zauważmy, że ona musi wysłać tą wiadomość mimo, że nie wie jakie jest i. Jeśli $a << k$, z dużym prawdopodobieństwem Alicja nie wyśle nic ciekawego o x_i, co jest jedyną interesującą częścią jej odpowiedzi. A więc możemy traktować protokół jako zaczynający się od drugiej wiadomości, eliminując pierwszą.

Lemat 2 (Lemat o eliminacji rund)**.** *Załóżmy, że jest protokół dla $f^{(k)}$ w którym Alicja mówi pierwsza, który używa t wiadomości i jego prawdopodobieństwo błędu wynosi δ. Wtedy jest protokół dla f, gdzie Bob mówi pierwszy, który używa $t - 1$ wiadomości i ma prawdopodobieństwo błędu $\delta + O(\sqrt{a/k})$.*

7.3.3 Szkic dowodu ograniczenia

Niech t będzie liczbą rund komunikacji (czyli liczbą zapytań do pamięci) wykonanych przez algorytm. Naszym celem jest wykonanie t eliminacji rund, eliminując wszystkie wiadomości. W miarę eliminowania kolejnych rund, redukujemy n i w do jakichś n' i w'. Chcemy za każdym razem zwiększać prawdopodobieństwo błędu o co najwyżej $\frac{1}{3t}$, aby na końcu otrzymać nietrywialne prawdopodobieństwo sukcesu (co najmniej $\frac{2}{3}$).

7.3.4 Eliminacja Alicja ⇒ Bob

Wejście Alicji ma w' bitów (na początku $w = w'$). Podzielmy je na k równych kawałków $x_1, x_2, \cdots, x_k$, gdzie $k = \Theta(at^2)$. Każdy kawałek jest liczbą w'/k-bitową.

Możemy skonstruować drzewo, w którym każdy wierzchołek ma $2^{w'/k}$ synów, na w'-bitowych słowach odpowiadających możliwym wejściom Alicji, które są elementami struktury danych. Drzewo ma wtedy wysokość k. Ta technika jest podobna do drzew vEB, z tym że zamiast podzielić zapytanie na dwie części, dzielimy na k części.

Ponieważ udowadniamy ograniczenie dolne, możemy dobrać dane dowolnie by sprawić, że problem jest trudny. Niech elementy w strukturze danych mają wspólny prefiks długości i z zapytaniem Alicji i niech wszystkie różnią się na i-tym kawałku. A więc wszystkie elementy są rozgałęzione w i-tym kawałku (najtrudniejszy przypadek drzew vEB). Alicja i Bob znają strukturę wejść, więc Bob potrzebuje znać tylko i, wartość i-tego kawałka oraz $x_1, \cdots, x_i$ (bo wszystkie wartości Boba muszą mieć wspólny prefiks). Więc gdy wyeliminujemy wiadomość Alicji, w' jest zredukowane do $w'/k = \Theta(w'/at^2)$. Używając lematu można pokazać, że prawdopodobieństwo błędu zwiększa się o $O(1/t)$, czyli dokładnie tyle, na ile możemy sobie pozwolić w jednej rundzie.

7.3.5 Eliminacja Bob ⇒ Alicja

Wiadomość Alicji jest wyeliminowana, więc Bob mówi pierwszy, więc nie zna wartości zapytania. Wejście Boba to n' liczb długości w'. Podzielmy ten zbiór na k kawałków złożonych z n'/k liczb każdy, gdzie $k = \Theta(bt^2)$. To jest analogiczne do drzew fusion, gdzie drzewa te rozgałęziają się, zmniejszając n o współczynnik $w^{1/5}$.

Ponownie możemy skonstruować trudną instancję problemu. Prefiksujemy liczby w każdym kawałku wartością długości $\log k$, dając unikalny identyfikator każdemu kawałkowi (i-ty kawałek będzie zaczynał się prefiksem i). Zapytanie Alicji zaczyna się jakimś losowym prefiksem długości $\log k$, który decyduje o tym który kawałek jest interesujący, więc używając lematu możemy wyeliminować wiadomość Boba zwiększając prawdopodobieństwo błędu o $O(1/t)$.

Eliminacja redukuje n' do $n'/k = \Theta(n'/bt^2)$ oraz w' do $w' - \log k = \Theta(\log bt^2)$. Dopóki w jest niemałe ($\Omega(\log bt^2)$), ostatni składnik można pominąć (redukuje on w o rząd co najwyżej 2).

7.3.6 Zatrzymanie eliminacji

Każda runda redukuje n' do $\Theta(n'/bt^2)$ i w' do $\Theta(w'/at^2)$ oraz można uzyskać prawdopodobieństwo błędu mniejsze niż $\frac{1}{3}$ odpowiednio dobierając stałe.

Zatrzymujemy eliminację gdy $w' = O(\log bt^2)$ lub $n' = 2$. Stopień redukcji daje nam ograniczenie dolne $t = \Omega(\min\{\log_{at^2} w, \log_{bt^2} n\})$. Ponieważ $t = O(\log n)$, $a \geqslant \log n$ i $t = O(\log w)$, $b = w$, podstawy logarytmów są odpowiednio pomiędzy a i a^3 oraz b i b^3. A więc otrzymaliśmy ograniczenie dolne $t = \Omega(\min\{\log_a w, \log_b n\})$

7.4 Bez założenia skończonego uniwersum

W modelu drzew decyzyjnych, algorytm jest widziany jako drzewo. Węzły reprezentują dowolne obliczenie z wyjątkiem skoków, a krawędzie wychodzące z węzłów reprezentują decyzje. Każdy węzeł może mieć $O(1)$ synów. Długość drogi z korzenia do liścia w danej instancji algorytmu jest długością działania algorytmu. Zatem złożoność czasowa to wysokość drzewa.

Twierdzenie 3. *W modelu drzew decyzyjnych każda struktura danych implementująca operacje* INSERT *i* SUCCESSOR *wymaga* $\Omega(\log n)$ *zamortyzowanego czasu wykonania.*

Dowód: Redukcja liniowa z sortowania:

Algorithm 3 Sort$(a_1, a_2, \ldots, a_n)$

```
for i = 1 to n do
  INSERT(a_i)
end for
print x = SUCCESSOR(−∞)
for i = 1 to n − 1 do
  print x = SUCCESSOR(x)
end for
```

Powyższa procedura w oczywisty sposób sortuje elementy $a_1, a_2, \cdots, a_n$. Zatem SORT $\leqslant_{LIN}$ INSERT & SUCCESSOR. Sortowanie n elementów ma $n!$ różnych permutacji, więc jest co najmniej $n!$ liści w drzewie decyzyjnym sortowania. Zatem wysokość drzewa jest $\Omega(n \log n)$, a więc również złożoność sortowania.

INSERT & SUCCESSOR jest niełatwiejsze od sortowania, więc dla tego problemu dolne ograniczenie w modelu drzew decyzyjnych to również $\Omega(n \log n)$, więc amortyzacyjnie każda operacja potrzebuje $\Omega(\log n)$ czasu.

8 Zastosowania

Struktury danych implementujące słownik z uporządkowanego uniwersum z INSERT, DELETE i SUCCESSOR mają kilka zastosowań.

8.1 Union-split-find

Union-split-find:

- T - podział $\{1, 2, \cdots, u\}$ na przedziały.

- T.find(x) - zwraca nazwe przedziału zawierającego x.
- T.union(x) - łączy przedział zawierający x z następnym przedziałem.
- T.split(x) - dzieli przedział I zawierający x na przedziały $I \cap [1, x]$ oraz $I \cap [x + 1, u]$.

Implementacja: Przedziały można reprezentować przez ich największy element. Zatem podział $[1, x_1], [x_1 + 1, x_2], \cdots, [x_k + 1, u]$ można reprezentować przez zbiór $S = \{x_1, x_2, \cdots, x_k, u\}$. Zbiór S można przechować używając słownika z uporządkowanego uniwersum. Wtedy $T.find$ to SUCCESSOR, $T.union$ to DELETE a $T.split$ to INSERT.

8.2 Sortowanie liczb całkowitych ze skończonego uniwersum

Implemetacja jest identyczna jak w twierdzeniu 3.

Literatura

[1] P. van Emde Boas, *Preserving Order in a Forest in less than Logarithmic Time*, FOCS, 75-84, 1975.

[2] Dan E. Willard, *Log-Logarithmic Worst-Case Range Queries are Possible in Space* $\Theta(n)$, Inf. Process. Lett. 17(2): 81-84 (1983)

[3] M. Fredman, D. E. Willard, *Surpassing the Information Theoretic Bound with Fusion Trees*, J. Comput. Syst. Sci, 47(3):424-436, 1993.

[4] A. Andersson, P. B. Miltersen, M. Thorup, *Fusion Trees can be Implemented with* AC^0 *Instructions Only*, Theor. Comput. Sci, 215(1-2): 337-344, 1999.

[5] Andrej Brodnik, Svante Carlsson, Johan Karlsson, and J. Ian Munro. *Worst case constant time priority queue.* Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms, Washington, DC, 2001, 523–528.

[6] Michael L. Fredman and Michael E. Saks. *The cell probe complexity of dynamic data structures.* In Proceedings of the 21st ACM Symposium on Theory of Computing, Seattle, Washington, 1989, 345–354.

[7] Miklos Ajtai: *A lower bound for finding predecessors in Yao's cell probe model*, Combinatorica 8(3): 235-247, 1988.

[8] Peter Bro Miltersen: *Lower bounds for union-split-find related problems on random access machines*, Symposium on the Theory of Computing 1994: 625-634.

[9] Peter Bro Miltersen, Noam Nisan, Shmuel Safra, Avi Wigderson: *On data structures and asymmetric communication complexity*, Symposium on the Theory of Computing 1995: 103-111.

[10] Peter Bro Miltersen, Noam Nisan, Shmuel Safra, Avi Wigderson: *On Data Structures and Asymmetric Communication Complexity*, Journal of Computer and System Sciences, 57(1): 37-49 (1998)

[11] Paul Beame, Faith E. Fich: *Optimal Bounds for the Predecessor Problem*, Symposium on the Theory of Computing 1999: 295-304.

[12] Paul Beame, Faith E. Fich: *Optimal Bounds for the predecessor problem and related problems*, Journal of Computer and System Sciences, p.38-72, August 2002.

[13] Bing Xiao: *New bounds in cell probe model*, PhD thesis, University of California, San Diego, 1992.

[14] Pranab Sen: *Lower bounds for predecessor searching in the cell probe model*, IEEE Conference on Computational Complexity 2003, 73-83.

[15] Pranab Sen, Srinivasan Venkatesh:*lower bounds for predecessor searching in the cell probe model*, Journal of Computer and System Sciences 2008, 364-385.

[16] Mihai Patrascu, Mikkel Thorup: *Time-space trade-offs for predecessor research*, Symposium on Theory of computing 2006, 232-240.

[17] Mihai Patrascu, Mikkel Thorup: *Randomization does not help searching predecessors*, Symposium on Discreat Algorithms