1 Struktury danych dla liczb całkowitych I

1.1 Modele obliczeń

Kilka kolejnych wykładów poświęcimy strukturom danych które umożliwiają wykonanywanie różnego rodzaju operacji na zbiorach liczb całkowitych.

Okazuje się, że jeśli ograniczymy się do przypadku liczb całkowitych (a nie, na przykład, elementów dowolnego liniowego porządku), to tradycyjnie używane modele, take jak RAM, czy drzewa decyzyjne, nie mają zbyt wiele sensu.

Przypomnijmy, że w modelu drzew decyzyjnych, algorytm jest drzewem porównań (można też rozpatrywać bogatsze modele, które umożliwiają wykonywanie operacji arytmetycznych na elementach danych wejściowych, tzw. algebraiczne drzewa decyzyjne). Z jednej strony ten model jest bardzo silny, ponieważ ignorujemy strukturalny aspekt algorytmu, w szczególności program może mieć wykładniczy rozmiar (!!). Z drugiej strony w modelu tym brakuje dostępu do lokacji w pamięci adresowanej przez zmienną (kluczowe w hashowaniu). Co więcej, gdybyśmy chcieli uwzględnić w tym modelu operacje możliwe tylko w dziedzinie liczb całkowitych, np. przesunięcia bitowe, tracimy najważniejszą zaletę tego modelu, tzn. względną łatwość uzyskiwania dolnych granic.

Model RAM jest jakby przeciwieństwem modelu drzew decyzyjnych. Bardzo dobrze oddaje on strukturalny aspekt algorytmów, dostępne w nim są wszystkie naturalne operacje, także te możliwe tylko w dziedzinie liczb całkowitych. Ceną, jaką za to płacimy jest oczywiście ogromna komplikacja modelu, a w konsekwencji problemy z dowodzeniem dolnych granic. Nawet model RAM nie opisuje jednak realistycznych obliczeń idealnie. Podstawowym zarzutem jaki można mu postawić jest jednostkowy czas wykonywania operacji arytmetycznych na argumentach dowolnych (!!) rozmiarów.

Przykład/Ćwiczenie 1 Pokaż, że w modelu RAM można posortować n liczb całkowitych w czasie $O(n)$.

Wskazówka Używając operacji przesunięcia bitowego, spróbuj zakodować wszystkie liczby w jednej wielkiej liczby, a następnie obliczyć rangi wszystkich elementów wejściowych.

Ten przykład pokazuje, że jeśli interesują nas liczby całkowite, musimy w jakiś sposób ograniczyć rozmiar argumentów, przy którym operacje arytmetyczne wykonują się w czasie jednostkowym. Problem ten z reguły rozwiązuje się nastepująco. Zakładamy, że nasz "komputer" może wykonywać operacje arytmetyczne w czasie stałym tylko na liczbach w-bitowych, dla pewnego ustalonego w, które będziemy nazywali długością słowa. Arytmetykę większych liczb musimy sobie sami zaimplementować, co oznacza, że jest ona odpowiednio droższa. Ten model nazywamy w-bitowym WordRAM. Zwróćmy uwagę, że zdefiniowaliśmy tu całą rodzinę modeli, sparametryzowaną przez w. Rozwiązując w modelu WordRAM konkretny problem, złożoność naszego algorytmu będziemy uzależniać zarówno od rozmiaru danych n jak i długości słowa w. Zakładamy przy tym, że $w \geq \log n$, co ma sens, bo musimy przecież jakoś adresować dane.

Uwaga Wiekszość znanych algorytmów, na przykład grafowych, ma w tym modelu złożoność taką samą jak w modelu RAM, niezależnie od parametru w.

Perwersyjny algorytm z Ćwiczenia 1 jest tu wyjątkiem.

Niekiedy rozpatruje się także nieco bardziej ograniczone modele. W szczególności, bardzo naturalnym modelem jest tzw. AC_0RAM. Model ten jest identyczny z WordRAM z jednym wyjątkiem, w czasie jednostkowym wykonują się w nim jedynie operacje typu AC_0, tzn. takie, które można zrealizować przy pomocy obwodów logicznych stałej głębokości i wielomianowego (względem w) rozmiaru (bramki mogą mieć dowolnie duży stopień wejściowy). Pozostałe, tzn. przede wszystkim mnożenie, musimy zrealizować sami. Dlaczego nakładamy takie ograniczenie? W jakimś sensie jest ono podyktowane przez rzeczywiste ograniczenia architektury komputerów. Rozmiar obwodów realizujących operacje typu AC_0 jest stały, niezależny od w. Tylko dla takich operacji ma sens założenie, że wykonują się one w czasie jednostkowym. Gdybyśmy budowali komputer 1.000.000-bitowy, to dodawanie liczb nie byłoby na nim czasowo dużo kosztowniejszy niż na komputerze 64-bitowym, ale już mnożenie tak.

1.2 Problem poprzednika

W problemie *p*oprzednika (ew. *n*astępnika, chcemy realizować następujące operacje na zbiorze liczb całkowitych S:

1. INSERT(x) – dodaje x do S,
2. DELETE(x) – usuwa x z S,
3. FIND(x) – sprawdza, czy x należy do S,
4. PREDECESSOR(x)/PRED(x) – zwraca poprzednik x w S,
5. SUCCESOR(x)/SUCC(x) – zwraca następnik x w S.

Uwaga: W operacjach PRED i SUCC argument x nie musi być elementem zbioru S.

Można powiedzieć, że problem poprzednika to uporządkowana wersja problemu słownika. Zwróćmy uwagę, że "wolne" zrównoważone drzewa BST pozwalają na wykonywanie operacji PRED w czasie $O(\log n)$, ale już "szybkie" tablice hashujące potrzebują czasu $O(n)$.

My będziemy zajmować się problemem poprzednika w modelu WordRAM/AC_0RAM, a zatem założymy, że nasze elementy są liczbami w-bitowymi, tzn. liczbami z przedziału $U = [0, \ldots, 2^w - 1]$. Jeśli zakładamy, że elementy S pochodzą ze skończonego uniwersum, tak jak w naszym przypadku, to mamy do czynienia z *p*roblemem poprzednika dla skończonego uniwersum (ang. FUP = finite universe predecessor). Jak zwykle, możemy też rozważać statyczną wersję problemu, w której nie realizujemy operacji modyfikujących S, tzn. INSERT i DELETE.

1.3 Drzewa van Emde Boasa

Nasz rozważania na temat problemu poprzednika rozpoczniemy od drzew van Emde Boasa. Drzewa ta realizują wszystkie operacje problemu poprzednika w czasie $O(\log w = \log\log |U|)$, a więc niezależnym od n a tylko od rozmiaru uniwersum.

Patrząc na złożoność jaką chcemy osiągnąć, można zgadnąć jak będzie działały operacje na naszej strukturze danych – będą one wykonywały "wyszukiwanie binarne" na reprezentacji bitowej x.

Dla dowolnej liczby k bitowej x, niech $\mathrm{high}(x)$ będzie $k/2$-bitową bardziej znaczącą częścią x, a $\mathrm{low}(x)$ $k/2$-bitową mniej znaczącą częścią x. Algorytm operacji $\mathrm{PRED}(x)$ mógłby wyglądać na przykład tak:

1. sprawdź czy S zawiera jakieś elementy y takie, że $\mathrm{high}(y) = \mathrm{high}(x)$, w szczególności czy najmniejszy z tych elementów jest mniejszy niż x,
2. jeśli nie, to musimy znaleźć $h = \mathrm{PRED}(\mathrm{high}(x))$ w zbiorze $\mathrm{high}(S)$ i jako odpowiedź zwrócić najwiekszy element $y \in S$ taki, że $\mathrm{high}(y) = h$,
3. jeśli tak, to szukamy $l = \mathrm{PRED}(\mathrm{low}(x))$ w zbiorze $\{\mathrm{low}(y) : y \in S, \mathrm{high}(y) = \mathrm{high}(x)\}$ i zwracamy $\mathrm{high}(x)l$.

Tak właśnie działają drzewa (kolejki) van Emde Boasa. Drzewo van Emde Boasa typu $\mathrm{VEB}(u)$ dla podzbioru S uniwersum $U = \{0, \ldots, u-1\}$ zawiera:

1. tablicę $L[0, \ldots, \sqrt{u}]$ drzew van Emde Boasa typu $\mathrm{VEB}(\sqrt{u})$,
2. H – drzewo typu $\mathrm{VEB}(\sqrt{u})$,
3. min – wartość najmniejszego elementu S,
4. max – wartość największego elementu S.

Drzewo H odpowiada zbiorowi $\mathrm{high}(S)$, natomiast drzewo $L[i]$ zbiorowi $\{\mathrm{low}(y) : y \in S, \mathrm{high}(y) = i\}$. Algorytm PRED możemy teraz zapisać następująco:

```
PRED(x)
if ((high(x) not in H) or (L[high(x)].min >= x))
   y = H.PRED(high(x))
   return sqrt(u)*y+L[y].max
else
   return sqrt(u)*high(x) + L[high(x)].PRED(low(x))
```

Ćwiczenie 2 Spróbuj zapisać algorytmy INSERT i DELETE. Jaką mają one złożoność?
Wskazówka Spróbuj "buforować" wstawienia.
Rozwiązanie Problem z operacjami INSERT i DELETE polega na tym, że jeśli dodajemy/usuwamy element z podstruktury H, to w zasadzie musimy wykonać dwa rekurencyjne wywołania – jedno dla H i jedno dla odpowiedniego L. Aby uniknąć tego problemu umawiamy się, że element zapamiętany jako min NIE JEST przechowywany w odpowiedniej podstrukturze L. W ten sposób pozbywamy się jednego wywołania rekurencyjnego.

Złożoność wszystkich operacji jest $O(\log w)$.

Ćwiczenie Jaka jest złożoność pamięciowa drzew van Emde Boasa?
Rozwiązanie

$$S(u) = 1 + \sqrt{u} + (\sqrt{u} + 1) * S(\sqrt{u}).$$

Sprawdzamy, że $S(u) \leq au + b$:

$$S(u) = 1 + \sqrt{u} + au + \sqrt{u}(b + 1) - b = au + \sqrt{u}(b + 2) + (1 - b).$$

Weźmy $b = -3$ i a wystarczająco duże, żeby dało się udowodnić podstawę indukcji.
Ćwiczenie Jaka będzie złożoność pamięciowa jeśli niepuste podstruktury będziemy trzymali w (dynamicznej) tablicy hashującej?
Rozwiązanie Tym razem dostajemy równanie:

$$S(w, n) = 1 + O(n) + S(w/2, k) + \sum_{i=1}^{k} S(w/2, n_i),$$

gdzie k jest liczbą elementów w H, a n_i jest liczbą elementów w i-tej niepustej podstrukturze. Sprawdzamy, że $S(w, n) \leq c(n - 1) \log w$:

$$S(w, n) = 1 + O(n) + c(k - 1)(\log w - 1) + c(n - k)(\log w - 1) \leq c(n - 1) \log w.$$

Aby przekonać się, że złożoność ta faktycznie jest $\Omega(n \log w)$ a nie, na przykład $O(n)$, wystarczy rozważyć zbiór elementów postaci $0^{2^i} 1^{w-2^i}$.

1.4 y-szybkie drzewa

Skonstruowane przez Willarda y-szybkie drzewa pokazują, że pomysł wyszukiwania binarnego w reprezentacji bitowej można zrealizować jeszcze bardziej dosłownie.

Rozważmy zbiór wszystkich prefiksów reprezentacji bitowych elementów z S. Prefiksy te tworzą drzewo binarne którego liściami są elementy S. Przypuśćmy, że trzymamy wszystkie wierzchołki tego drzewa w tablicy hashującej. Dodatkowo w każdym wierzchołku pamiętamy najmniejszy i największy liść z jego poddrzewa, a wszystkie liście łączymy w listę podwójną. Wtedy znalezienie poprzednika/następnika jest proste:

```
PRED/SUCC(x):
binarnie wyszukaj w drzewie
     najdłuższy prefiks p reprezentacji x
PRED/SUCC(x) = min/max z poddrzewa (jedynego!) syna p
skorzystaj z listy podwójnej, aby znaleźć
     drugi element z pary PRED(x), SUCC(x)
```

Ten algorytm działa oczywiście w czasie $O(\log w)$, pozostaje jednak kilka problemów. Po pierwsze, nie widać w jaki sposób uaktualniać opisaną strukturę danych, w pesymistycznym przypadku trzeba zmodyfikować $\Theta(w)$ wartości min/max. Po drugie, to drzewo ma nw wierzchołków. Oba problemy można rozwiązać stosując tzw. "indirection" (reprezentację pośrednią?).

1.5 Indirection

Zamiast przechowywać wszystkie elementy S w y-szybkim drzewie, postępujemy następująco. Dzielimy elementy S na spójne podzbiory $S_1, \ldots, S_k$ rozmiaru $\Theta(w)$, każdy podzbiór przechowujemy w zrównoważonym drzewie BST, a w y-szybkim drzewie pamiętamy tylko "reprezentantów" $r_1, \ldots, r_k$ podzbiorów. Od reprezenta r_k wymagamy jedynie, by $\max S_i <= r_i < \min S_{i+1}$, nie musi on należeć do S.

Aby znaleźć poprzednika x, przy pomocy y-szybkiego drzewa szukamy r_j – poprzednika x wśród $r_1, \ldots, r_i$, a następnie w drzewie odpowiadającym S_j oraz S_{j+1}.

Elementy dodajemy usuwamy tylko w drzewach BST, chyba że rozmiar modyfikowanego drzewa stanie się $> 4w$ lub $< w$. W pierwszym przypadku dzielimy drzewo na dwa mniejsze drzewa, w drugim łączymy odpowiednie drzewo z drzewem sąsiednim (i ew. dzielimy powstałe drzewo na dwa mniejsze). Obie operacje wymagają wybrania nowych reprezentantów i uaktualnienia y-szybkiego drzewa.

Jaka jest złożoność czasowa i pamięciowa tego algorytmu? Operacja poprzednika wykonuje się oczywiście w czasie $O(\log w)$. Operacje wstawiania/usuwanie podobnie, chyba że konieczna jest aktualizacja y-szybkiego drzewa. Wtedy wymagany jest czas $O(w)$. Zauważmy jednak, że sytuacja taka występuje tylko raz na $\Theta(w)$ operacji, a zatem zamortyzowany czas wstawień/usunięć jest $O(\log w)$.

Złożoność pamięciowa jest $O(n)$ – y-szybkie drzewo dla $\Theta(n/w)$ elementów wymaga pamięci $O(wn/w) = O(n)$, drzewa BST podobnie.

Zaprezentowana tu technika jest dość ogólna i ma wiele zastosowań.

Ćwiczenie Pokaż, w jaki sposób można operację PRED zrealizować w czasie $(\log\log(x - \mathrm{PRED}(x)))$ zamiast $O(\log\log|U|)$. Czy potrafisz w podobny sposób przyspieszyć kolejkę van Emde Boasa?