Wrocław
University
of Science
and Technology

Struktury danych

Wykład 6

Kopiec Fibonacciego, struktura zbiorów rozłącznych

dr inż. Jarosław Rudy

HR EXCELLENCE IN RESEARCH

Koszt amortyzowany – metoda księgowania (1)

- Jedną z alternatywnych metod analizy kosztu amortyzowanego jest metoda księgowania.
- Koszt zamortyzowany operacji oprócz kosztu rzeczywistego uwzględnia (nieujemny) „kredyt".
- Jeśli operacja ma koszt zamortyzowany większy niż rzeczywisty, to zwiększa kredyt.
- Operacja może mieć koszt zamortyzowany mniejszy niż rzeczywisty, o ile w momencie jej wykonywania mamy wystarczający kredyt.
- Ponieważ kredyt jest nieujemny, obliczony w ten sposób koszt zamortyzowany ogranicza od góry koszt rzeczywisty.

Koszt amortyzowany – metoda księgowania (2)

Przykład dodawania elementu na koniec tablicy dynamicznej:

- Każda operacja dodawania daje 1 kredyt.
- Rozważmy ciąg $k+1$ operacji:
  - Pierwsze k operacji nie wymaga rozszerzenia tablicy. Każda ma koszt rzeczywisty 1 i zamortyzowany 2, łącznie gromadząc kredyt wysokości k.
  - Ostatnia operacja musi rozszerzyć tablicę, więc jej rzeczywisty koszt to $k+1$ (czyli $O(k)$).
    - Jak każde dodawanie, operacja przydziela 1 kredyt... ale możemy wykorzystać zgromadzone k kredytu.
    - Ostatecznie koszt zamortyzowany to $1+1+k-k=2 \in O(1)$.
  - Każda operacja w ciągu ma więc koszt zamortyzowany równy $O(1)$!

Koszt amortyzowany – metoda potencjału (1)

- Strukturze danych przypisujemy potencjał.
- Potencjał jest funkcją od stanu struktury.
  - Potencjał mogą posiadać fragmenty struktury, potencjał całości jest wtedy sumą potencjałów częsci.
- Koszt zamortyzowany operacji i oblicza się jako koszt rzeczywisty plus różnica potencjałów (potencjał po operacji i i po operacji i_1):

$$k_{\text{amortized}} = k_{\text{real}} + \Phi_i - \Phi_{i-1}. \tag{1}$$

Koszt amortyzowany – metoda potencjału (2)

Przykład dodawania elementu na koniec tablicy dynamicznej:

- Zdefiniujmy potencjał tablicy jako $\mathrm{size} - \mathrm{capacity}$. W tym przypadku potencjał może być ujemny, ale niczemu to nie przeszkadza.
- Rozważmy ciąg $k+1$ operacji:
  - Każda z pierwszych k operacji ma koszt rzeczywisty równy 1, a potencjał wzrasta o 1.
    - Koszt zamortyzowany wynosi więc 2.
  - Ostatnia operacja ma koszt rzeczywisty $k+1$, ale potencjał spada o $k-1$.
    - Koszt zamortyzowany wynosi więc 2.
  - Każda operacja w ciągu ma więc koszt zamortyzowany równy $O(1)$!

Koszt amortyzowany – metoda potencjału (3)

i	capacity	size	Φ_i	$\Phi_i - \Phi_{i-1}$	k_{real}	$k_{\text{amortized}}$
init	4	0	-4	n/d	n/d	n/d
1	4	1	-3	1	1	2
2	4	2	-2	1	1	2
3	4	3	-1	1	1	2
4	4	4	0	1	1	2
5	8	5	-3	-3	$4 + 1 = 5$	2
6	8	6	-2	1	1	2
7	8	7	-1	1	1	2
8	8	8	0	1	1	2
9	16	9	-7	-7	$8 + 1 = 9$	2

Kopiec Fibonacciego (1)

- Kopiec binarny pozwala na wydajną realizację kolejki priorytetowej.
- Najważniejsze operacje (insert, find-min, extract-min, decrease-key) mają złożoność $O(\log n)$.
- Jednak tylko operacja find-min ma czas $O(1)$.
- Lepszą złożoność teoretyczną oferuje kopiec Fibonacciego.
  - Dalej opisujemy kopce typu min.

Kopiec Fibonacciego (2)

- Kopiec Fibonacciego ma formę lasu złożonego z kopców.
- Kształt kopców jest mniej rygorystyczny.
- Większa elastyczność pozwala na zostawienie niektórych czynności na później (tzw. lazy approach).
- W pewnym momencie potrzeba jednak narzucić większy porządek (kosztem dodatkowego czasu operacji).

Kopiec Fibonacciego (3)

- Stopień (liczba dzieci) każdego węzła wynosi co najwyżej $\log n$.
- Dla węzła o stopniu k rozmiar podrzewa zakorzenionego w nim wynosi co najmniej F_{k+2} ($k+2$-ta liczba Fibonacciego).
- Dla nie-korzenia x możemy odciąć tylko jednego syna. Gdy odcinamy kolejnego, x samo jest odcinane i staje się osobnym drzewem.
- Zwiększa to liczbę drzew, ale drzewa można łączyć podczas innej operacji.

Kopiec Fibonacciego (4)

- Do analizy złożoności kopca Fibonacciego stosowana jest metoda potencjału.
- Węzeł x jest znaczony (marked), jeśli co najmniej 1 jego syn został odcięty od czasu gdy x został czyimś synem.
  - Korzenie nigdy nie są znaczone.
- Potencjał kopca Fibonacciego to liczba kopców plus dwukrotność liczby znaczonych węzłów:

$$\Phi = t + 2m \tag{2}$$

- Czas zamortyzowany to czas rzeczywisty plus różnica potencjałów razy pewna dobrana wartość C.

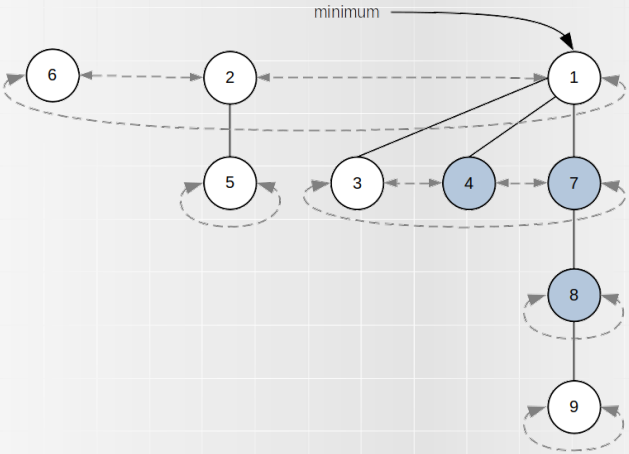
Kopiec Fibonacciego (5)

Dodatkowe założenia dotyczące realizacji struktury:

- Korzenie (las) połączone są cykliczną listą dwukierunkową.
- Rodzeństwo również połączone jest taką listą.
- Węzeł przechowuje stopień (liczbę dzieci) i to czy jest znaczony.
- Przechowujemy wskaźnik na najmniejszy korzeń.

Kopiec Fibonacciego (6)

Przykład kopca Fibonacciego:



Kopiec Fibonacciego (7)

Implementacja operacji:

- find-min() – zwracamy pamiętany wskaźnik na minimum.
  - Nie zmienia potencjału, czas zamortyzowany $O(1)$.
- merge() – operacja, którą można zdefiniować dla wielu ADT/struktur danych
  - Łączy dwie struktury (zwykle tego samego typu) w jedną.
  - Połączenie dwóch kopców Fibonacciego sprowadza się do połączenia dwóch list cyklicznych z korzeniami i wybrania nowego minimum z dwóch wartości.
  - Nie zmienia potencjału, czas zamortyzowany $O(1)$.

Kopiec Fibonacciego (8)

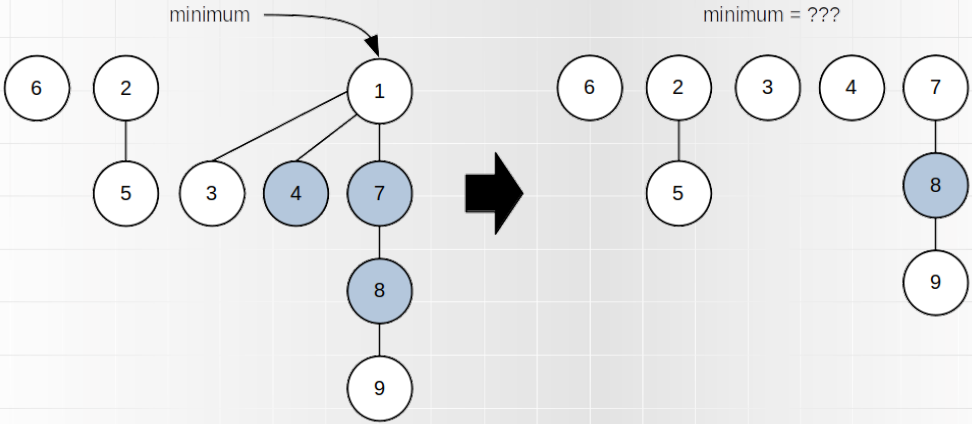
- insert() – dodaje nowy 1-elementowy kopiec (sam korzeń), konieczność wyboru nowego elementu minimalnego z dwóch możliwych.
  - Alternatywnie: tworzymy nowe drzewo i wykonujemy merge() z oryginalnym drzewem.
  - Czas rzeczywisty $O(1)$.
  - Potencjał zwiększa się o 1 (1 nowe drzewo bez węzłów znaczonych).
  - Czas zamortyzowany $O(1)$.

Kopiec Fibonacciego (9)

extract-min() składa się z kilku faz:

- Faza pierwsza:
  - Usuwamy minimalny węzeł (do którego mamy wskaźnik).
  - Dzieci usuniętego węzła (który był korzeniem) stają się osobnymi drzewami.
  - Jeśli dzieci było k, to potencjał wzrasta o $k - 1$ (1 korzeń znika, k się pojawia).
  - Czas to $O(k) \in O(\log n)$.

Kopiec Fibonacciego (10)



Kopiec Fibonacciego (11)

- Faza druga:
  - Redukcja liczby korzeni – korzenie o tym samym stopniu są łączone (to nie jest operacja merge!)
  - Jedno z łączonych drzew staje się synem drugiego (zgodnie z własnością kopca).
  - Powtarzane dopóki pozostałe korzenie mają różne stopnie (będzie ich więc $O(\log n)$.
  - Przechowujemy tablicę rozmiaru $O(\log n)$ przechowującą w indeksie i wskaźnik do korzenia o stopniu i, co pozwala na szybkie lokalizowanie i łączenie kopców.

Kopiec Fibonacciego (12)

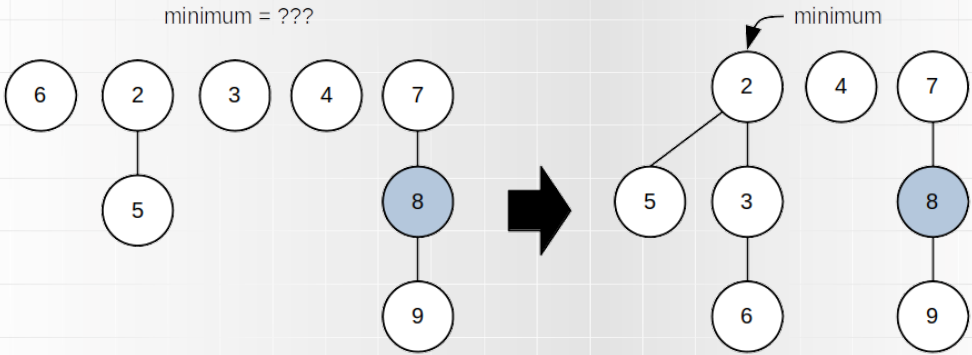
- Czas rzeczywisty: $O(\log n + m)$, gdzie m to liczba kopców (korzeni) na początku fazy.
- Zmiana potencjału: $O(\log n) - m$.
- Czas zamortyzowany: $O(\log n + m) + C(O(\log n) - m)$.
  - Przy odpowiednio dużym C wynik upraszcza się do $O(\log n)$.

- Faza trzecia – sprawdzamy wynikłe $O(\log n)$ korzeni, by znaleźć nowe minimum.
  - Czas $O(\log n)$, brak zmian potencjału, więc koszt zamortyzowany również $O(\log n)$.

Cała operacja extract-min ma więc zamortyzowany koszt $O(\log n)$.

Kopiec Fibonacciego (13)

Połączenie drzew (6) i (3), a następnie drzew $(3, 6)$ i $(2, 5)$.



Kopiec Fibonacciego (13)

Operacja decrease-key():

- Znajdujemy węzeł (z odpowiednim słownikiem trwa to $O(\log n)$).
- Zmniejszamy klucz węzła.
- Jeśli złamana jest zasada kopca, to odcinamy węzeł od rodzica i oznaczamy rodzica (chyba, że jest korzeniem).
- Jeśli rodzic był już oznaczony, to też go obcinamy i oznaczamy jego rodzica.
- Gdy proces się skończy (dotarliśmy do nieoznaczonego węzła), to określamy nowe minimum z dwóch wartości (zmniejszony klucz i nowe minimum).

Kopiec Fibonacciego (13)

- Załóżmy, że proces utworzył k nowych drzew (korzeni).
- Wśród nich co najmniej $k-1$ było oznaczonych (odcięty jako pierwszy w procesie mógł nie być).
- Ponieważ stają się korzeniami, przestają być oznaczone.
- Jeden węzeł mogł zostać oznaczony.
- Dochodzi k drzew oraz $-k+2$ oznaczonych węzłów.
- Potencjał zmienia się więc o $k+2(-k+2) = -k+4$.

Kopiec Fibonacciego (14)

- Czas rzeczywisty: $O(k)$.
- Czas zamortyzowany: $O(k) + C(-k + 4)$.
  - Dla odpowiedniego C otrzymujemy $O(1)$.

Operacja delete():

- Za pomocą decrease-key() ustawiamy wartość usuwanego węzła na $-\infty$.
- Za pomocą extract-min() usuwamy węzeł, który teraz stał się minimum.
- Czas $O(1) + O(\log n) \in O(\log n)$.

Kopiec Fibonacciego (15)

- Kopiec Fibonacciego poprawia teoretyczną złożoność niektórych algorytmów grafowych np. algorytmu Dijkstry i algorytmu Prima.
- Dobra teoretyczna złożoność okupiona jest jednak skomplikowaną implementacj i mniejszą skutecznośćią praktyczną (zwłaszcza w niektórych sytuacjach).
  - Niektóre pojedyncze operacje na kopcu wykonują się długo, zaś wyższa stała zmniejsza korzyść względem $O(\log n)$.
- Wskazówki praktyczne:
  - Gdy nie potrzeba operacji decrease-key(): stosujemy kopce binarne.
  - Gdy potrzeba decrease-key(): stosujemy kopce parujące (pairing heaps).
    - Asymptotycznie gorsze od kopca Fibonacciego, ale bardzo dobre w praktyce i prostsze w implementacji.

Kopiec Fibonacciego (16)

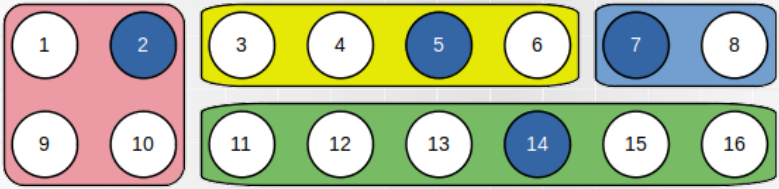
Podsumowanie złożoności niektórych odmian kopców:

Kopiec	insert()	extract-min	find-min()	decrease-key()	merge()
Binarny	$O(\log n)$	$\Theta(\log n)$	$\Theta(1)$	$O(\log n)$	$O(n)$
Dwumianowy	$\Theta(1)^a$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(\log n)$	$O(\log n)$
Parujący	$\Theta(1)$	$O(\log n)^a$	$\Theta(1)$	$o(\log n)^a$	$\Theta(1)$
Fibonacciego	$\Theta(1)$	$O(\log n)^a$	$\Theta(1)$	$\Theta(1)^a$	$\Theta(1)$

^{a} – czas zamortyzowany

Struktura zbiorów rozłącznych (1)

- Disjoint-set data structure – przechowuje elementy pogrupowane w zbiory tak, że każdy element należy do dokładnie jednego zbioru.
  - Z matematycznego punktu widzenia przechowuje pewne rozbicie zbioru.
- Każdy zbiór ma reprezentanta.
- Stosowana między innymi w algorytmie Kruskala.



Struktura zbiorów rozłącznych (2)

- Operacje:
  - insert() – dodanie nowego elementu.
    - Powstaje nowy zbiór 1-elementowy.
  - union() – łączenie dwóch zbiorów w jeden.
  - find() – znalezienie reprezenta zbioru.
    - Jeśli dwa elementy mają tego samego reprezentanta, to należą do tego samego zbioru.

Struktura zbiorów rozłącznych (3)

Implementacja naiwna z użyciem samych list:

- insert(x) – czas $O(1)$.
- union(x,y) – $O(\min\{|X|, |Y|\}) \in O(n)$.
- find(x) – czas $O(|X|) \in O(n)$.
- X i Y to zbiory do których należą odpowiednio x i y.

Struktura zbiorów rozłącznych (4)

W praktyce używamy implementacji lasu zbiorów rozłącznych (disjoint-set forest):

- Każdy zbiór reprezentowany jest przez osobne drzewo.
- Każdy węzeł pamięta rodzica (plus stopień węzła/rozmiar poddrzewa).
- Korzeń drzewa jest reprezentantem zbioru.
- Niektóre operacje mogą być długie, ale przekształcają strukturę na korzyść przyszłych operacji, dbając by drzewa miały małą wysokość.
- Czas zamortyzowany jest bardzo dobry.
- Implementacja jest zarówno szybka praktycznie jak i optymalna asymptotycznie (gorsza najwyżej o stałą od najlepszej możliwej).

Struktura zbiorów rozłącznych (5)

Implementacja insert(x):

- Stworzenie elementu x.
- $x.\mathrm{parent} \leftarrow x$.
- $x.rank \leftarrow 0$ (lub $x.size \leftarrow 1$).
- Dodanie x do listy drzew (zbiorów).
- Czas $O(1)$.

Struktura zbiorów rozłącznych (6)

Implementacja find(x):

- W teorii wystarczy jedynie podążać (iteracyjnie lub rekurencyjnie) za rodzicem, do korzenia i go zwrócić.
- Drzewa mogą jednak zrobić się duże, więc find() wykorzystuje tą okazję by zmniejszyć ich wysokość.
- Efekt uzyskuje się przez zmianę wskaźników, by zmniejszyć czas dotarcia do korzenia w kolejnych wywołaniach find() dla tego samego zbioru.

Struktura zbiorów rozłącznych (7)

Algorytm kompresji ścieżek (path compression):

- Dla każdego węzła na drodze od x do korzenia root wykonujemy:
  - $x.\mathrm{parent} \leftarrow \mathrm{root}$
- Przypisanie wymaga znajomości korzenia, więc trzeba wykonać 2 przejścia przez ścieżkę.
  - Podejście rekurencyjne – wymaga dodatkowej pamięci (zapis ścieżki na stosie).
  - Podejście iteracyjne – dwie pętle, pierwsza znajduje korzeń, druga przechodz ponownie ścieżkę i zapisuje go. Wymaga stałej pamięci.

Struktura zbiorów rozłącznych (8)

Algorytm dzielenia ścieżek (path spliting):

- Dla każdego węzła na drodze od x do korzenia wykonujemy:
  - $(x, x.\text{parent}) \leftarrow (x.\text{parent}, x.\text{parent.parent})$
- Węzeł zamiast ojca wskazywać będzie od teraz na dziadka – za każdym wykonaniem find() będą wskazywać bliżej korzenia.
- Pesymistycznie tak samo jak dla kompresji ścieżek, ale lepszy w praktyce.

Algorytm połowicznej ścieżki (path halving):

- Identycznie, ale zmiana jest co drugi węzeł:
  - $x.\text{parent} \leftarrow x.\text{parent.parent}$
  - $x \leftarrow x.\text{parent}$

Struktura zbiorów rozłącznych (9)

Implementacja union(x,y):

- Używamy find(x) i find(y) by znaleźć reprezentantów (nazwijmy ich odpowiednio r_x i r_y).
- Jeśli $r_x = r_y$ to x i y są w tym samym zbiorze i algorytm się kończy.
- Jeśli $r_x \neq r_y$, to łączymy zbiory, czyniąc jeden korzeń synem drugiego.
- Wybór bardzo wpływa na wysokość przyszłych drzew! Zapobiegamy zbyt wysokim drzewom za pomocą dodatkowych czynności.
- Konieczność przechowywania w węzłach rozmiaru drzewa (dodatkowe $O(\log n)$ bitów) lub rangi korzenia (dodatkowe $O(\log \log n)$ bitów).

Struktura zbiorów rozłącznych (10)

Łączenie przez rozmiar (union by size):

- Korzeniem złączonych drzew staje się korzeń drzewa o większym rozmiarze.
- Jeśli rozmiary są identycze, wybór jest dowolny.
- Aktualizujemy rozmiar nowego korzenia.

Struktura zbiorów rozłącznych (11)

Łączenie przez rangę (union by rank):

- Ranga jest górnym ograniczeniem na wysokość drzewa.
- Lepszy wskaźnik niż wysokość, bo nie zmienia się w czasie find().
- Jeśli łączone drzewa mają różne rangi, to ten z większą staje się rodzicem i rangi się nie zmieniają.
- Jeśli rangi są takie same, to dowolne drzewo może stać się rodzicem, ale jego rangę trzeba zwiększyć o 1.

Struktura zbiorów rozłącznych (12)

- find() i union() mają zamortyzowaną złożoność $O(\alpha(n))$, gdzie α jest odwrotnością bardzo szybko rosnącej funkcji Ackermanna.
- Ponieważ $O(1) \in O(\alpha(n))$, to dowolny ciąg k operacji na lesie zbiorów rozłącznych działa w czasie $O(k\alpha(n))$.
- „Fizycznie" wydaje się niemożliwe uzyskanie $\alpha(n) > 4$, w praktyce można więc przyjąć, że $O(\alpha(n)) \in O(1)$.
- Wszystkie operacje działają więc zasadniczo w czasie $O(1)$, ale jest to czas zamortyzowany – niektóre operacje w ciągu mogą zająć długi czas.