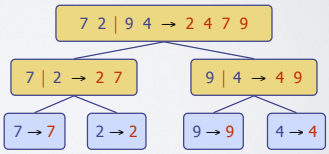
PROJEKTOWANIE I ANALIZA ALGORYTMÓW

DRZEWA, KOPCE, ALGORYTMY SORTOWANIA

Wykład 4&5

dr inż. Łukasz Jeleń
Na podstawie wykładów dr. T. Fevensa

SORTOWANIE PRZEZ SCALANIE



TECHNIKA DZIEL I ZWYCIĘŻAJ

Dziel i zwyciężaj jest ogólnym paradygmatem projektowania algorytmów:

- Podział: podziel dane wejściowe S na dwa rozłączne podzbiory S1 i S2
- Rekurencja: rozwiąż problem dla S1 i S2
- Scalanie: połącz rozwiązania dla S1 i S2 w jedno rozwiązanie dla S

Krokiem podstawowym rekurencji są podproblemy o rozmiarze 0 lub 1

Sortowanie przez scalanie jest algorytmem sortującym bazującym na technice dziel i zwyciężaj

Podobnie jak dla sortowania kopcowego

- wykorzystuje komparator
- Posiada złożoność O(n log n)

Inaczej niż dla sortowania kopcowego

- Nie wykorzystuje zewnętrznej kolejki priorytetowej
- Pobiera dane w sposób sekwencyjny (odpowiedni do sortowania danych na dysku zewnętrznym)

SORTOWANIE PRZEZ SCALANIE

Sortowanie przez scalanie listy S zawierającej n elementów składa się z trzech kroków:

- Podział: podział S na dwie sekwencje S1 i S2 zawierającymi ok. n/2 elementów każda
- Rekurencja: posortuj rekurencyjnie S1 i S2
- Scalanie: połącz S1 i S2 w jedną posortowaną listę

Algorytm *mergeSort(S, C)*
Wejście lista S z n elementami, komparator C
Wyjście lista S posortowana zgodnie z komparatorem C
if $S.size() > 1$
$(S_1, S_2) \leftarrow podziel(S, n/2)$
$mergeSort(S_1, C)$
$mergeSort(S_2, C)$
$S \leftarrow$ **pusta lista**
$połącz(S_1, S_2, S)$
return S

$$|S_1| = \left\lceil \frac{n}{2} \right\rceil \text{ and } |S_2| = \left\lfloor \frac{n}{2} \right\rfloor$$

ŁĄCZENIE DWÓCH POSORTOWANYCH SEKWENCJI

Ostatni krok sortowania przez scalanie składa się ze scalania dwóch posortowanych sekwencji S1 i S2 zaimplementowanych jako lista w jedną posortowaną sekwencję S zawierającą połączenie elementów z S1 i S2

Scalanie dwóch posortowanych sekwencji zawierających po n/2 elementów i zaimplementowane z zastosowaniem listy dwukierunkowej zabiera O(n) czasu

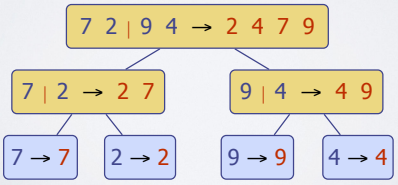
Algorytm *merge*(S_1, S_2, S)
Wejście sekwencje S_1 i S_2 zawierające po $n/2$ elementów, pusta sekwencja S
Wyjście posortowana sekwencja S: $S_1 \cup S_2$

while $\neg S_1.isEmpty() \wedge \neg S_2.isEmpty()$
if $S_1.first().element() \leq S_2.first().element()$
$S.addLast(S_1.remove(S_1.first()))$
else
$S.addLast(S_2.remove(S_2.first()))$
while $\neg S_1.isEmpty()$
$S.addLast(S_1.remove(S_1.first()))$
while $\neg S_2.isEmpty()$
$S.addLast(S_2.remove(S_2.first()))$

DRZEWO SORTOWANIA PRZEZ SCALANIE

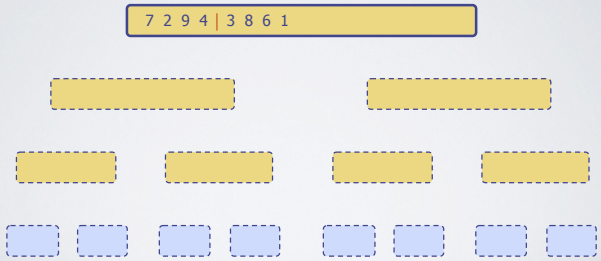
Działanie sortowania przez scalanie jest zobrazowane przez drzewo binarne

- każdy węzeł reprezentuje wywołanie rekurencyjne sortownia i zawiera
  - nieposortowaną sekwencję przez wywołaniem i podziałem
  - posortowaną sekwencję po zakończeniu wywołania
- korzeń jest początkowym wywołaniem
- liście są wywołaniami podsekwencji o rozmiarze 0 lub 1



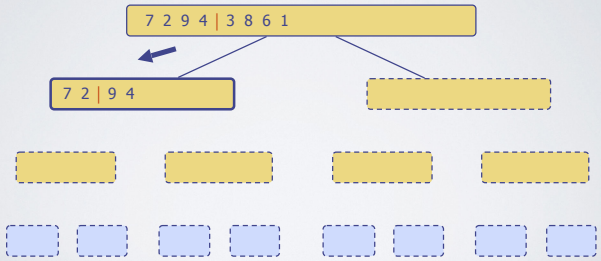
PRZYKŁAD DZIAŁANIA

Podział



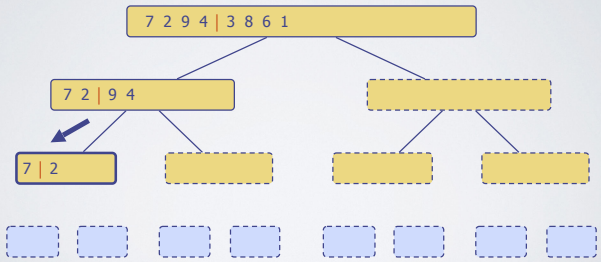
PRZYKŁAD DZIAŁANIA

Wywołanie rekurencyjne, podział



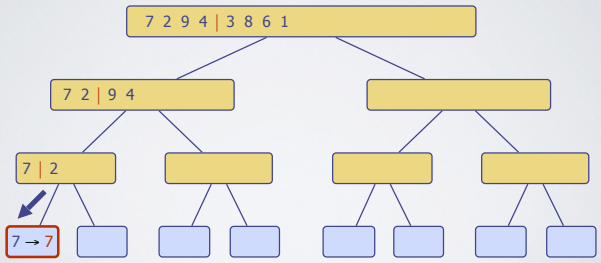
PRZYKŁAD DZIAŁANIA

Wywołanie rekurencyjne, podział



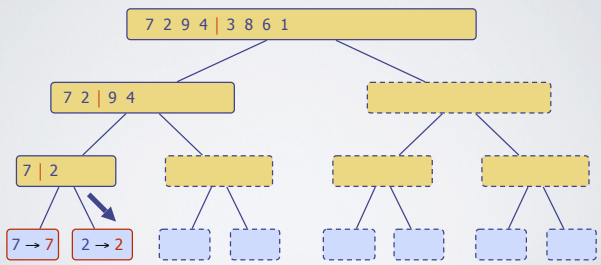
PRZYKŁAD DZIAŁANIA

Wywołanie rekurencyjne, podział



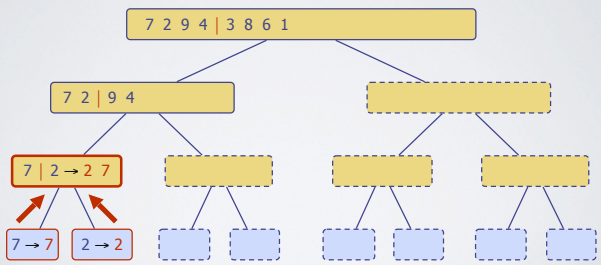
PRZYKŁAD DZIAŁANIA

Wywołanie rekurencyjne, krok podstawowy



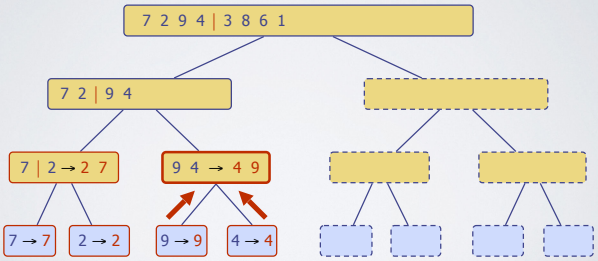
PRZYKŁAD DZIAŁANIA

Scalanie



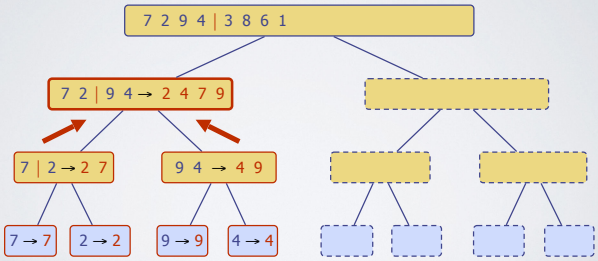
PRZYKŁAD DZIAŁANIA

Wywołanie rekurencyjne, …, krok podstawowy, scalanie



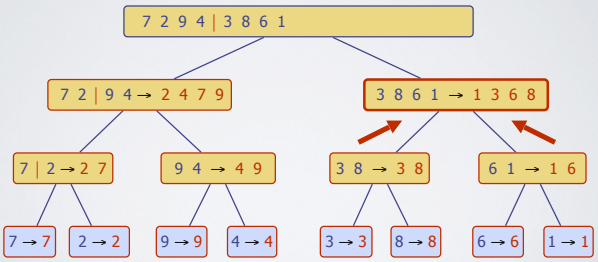
PRZYKŁAD DZIAŁANIA

Scalanie



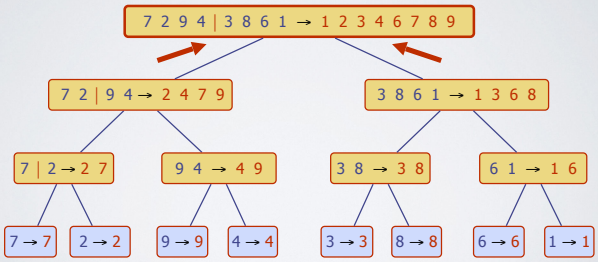
PRZYKŁAD DZIAŁANIA

Wywołanie rekurencyjne, …, krok podstawowy, scalanie



PRZYKŁAD DZIAŁANIA

Scalanie



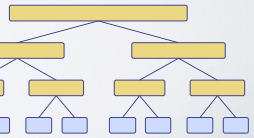
ANALIZA SORTOWANIA PRZEZ SCALANIE

Wysokość drzewa sortowania to O(log n)
- przy każdym wywołaniu rekurencyjnym dzielimy sekwencję na pół

Ilość operacji wykonywanych na węzłach na poziomie i to O(n)
- dzielimy i scalmy 2^i sekwencji o rozmiarze $n/2^i$
- wykonujemy 2^{i+1} wywołań rekurencyjnych

Zatem, całkowita złożoność obliczeniowa sortownia przez scalanie wynosi O(n log n)

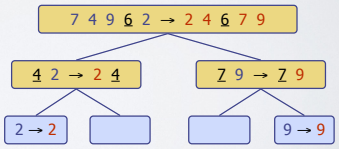
poziom	#sekw	rozmiar
0	1	n
1	2	$n/2$
i	2^i	$n/2^i$
…	…	…



PODSUMOWANIE ALGORYTMÓW SORTOWANIA

Algorytm	Złożoność	Uwagi
przez wybór	$O(n^2)$	wolne in-situ dla małych tablic (< 1K)
przez wstawianie	$O(n^2)$	wolne in-situ dla małych tablic (< 1K)
przez kopcowanie	$O(n \log n)$	szybkie in-situ (impl. na tablicy) dla średnich tablic (1K — 1M)
przez scalanie	$O(n \log n)$	szybkie sekwencyjny dostęp do danych dla b. dużych tablic (> 1M)

SORTOWANIE SZYBKIE



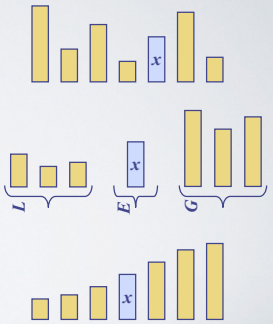
SORTOWANIE SZYBKIE

Sortowanie szybkie jest (losowym) algorytmem sortującym bazującym na technice dziel i zwyciężaj:

Podział: wybierz losowy element x (nazywany piwotem) i podziel S na
- L - elementy mniejsze od x
- E - elementy równe x
- G - elementy większe od x

Rekurencja: posortuj L i G

Scalanie: połącz L, E i G



PODZIAŁ

Dzielimy sekwencję wejściową w następujący sposób:
- Usuwamy element y z S i
- Wstawiamy y do L, E lub G w zależności od porównania z piwotem x

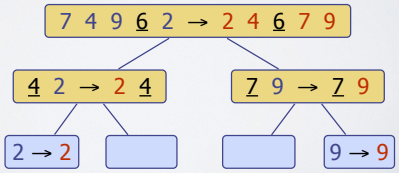
Wszystkie operacje wstawiania i usuwania wykonywane są na początku lub na końcu sekwencji, a zatem ich czas działania wynosi O(1)

Zatem czas działania kroku dzielącego sortowania szybkiego wyniesie O(n)

Algorytm *partition(S, p)*
Wejście sekwencja *S*, pozycja *p* piwota
Wyjście podsekwencje *L, E, G* z elementami z *S* mniejszymi, równymi, lub większymi od piwota
L, E, G ← puste sekwencje
x ← *S.remove(p)*
while ~*S.isEmpty*()
y ← *S.remove*(*S.first*())
if *y* < *x*
L.addLast(*y*)
else if *y* = *x*
E.addLast(*y*)
else { *y* > *x* }
G.addLast(*y*)
return *L, E, G*

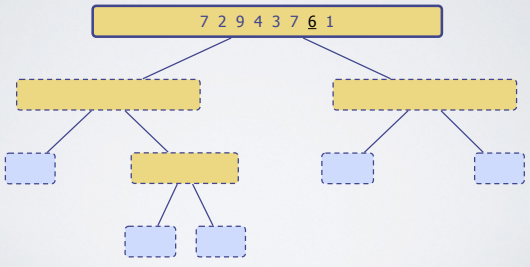
DRZEWO SORTOWANIA

Działanie sortowania szybkiego może być zilustrowane za pomocą drzewa binarnego
- Każdy węzeł reprezentuje wywołanie rekurencyjne i przechowuje:
  - Nieposortowaną sekwencję przed wywołaniem oraz piwot
  - Posortowaną sekwencję po wywołaniu
- Korzeń jest wywołaniem pierwotnym
- Liście są wywołaniami podsekwencji o rozmiarze 0 lub 1



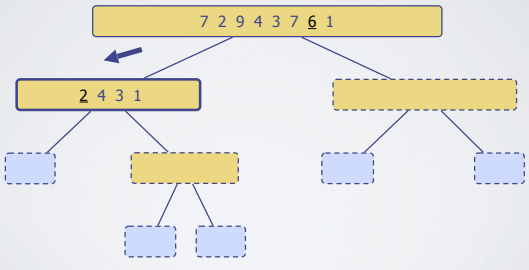
PRZYKŁAD DZIAŁANIA

Wybór piwota



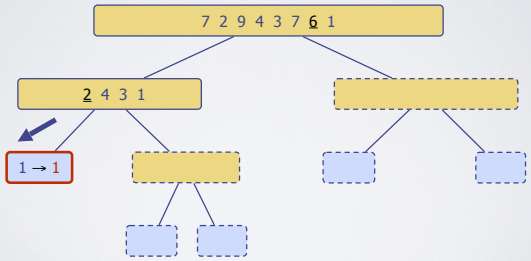
PRZYKŁAD DZIAŁANIA

Podział, wywołanie rekurencyjne, wybór piwota



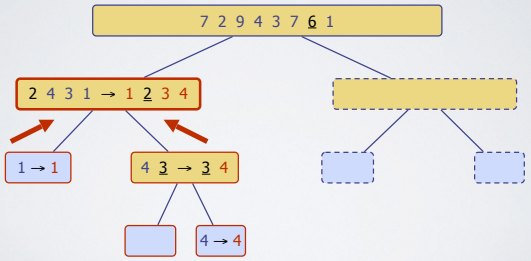
PRZYKŁAD DZIAŁANIA

Podział, wywołanie rekurencyjne, wybór piwota



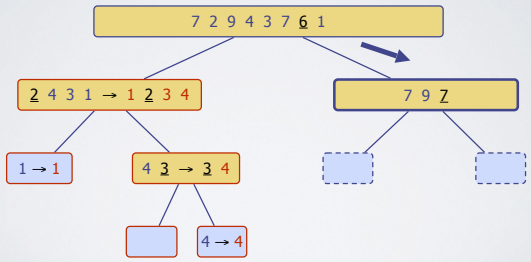
PRZYKŁAD DZIAŁANIA

Wywołanie rekurencyjne, …, krok podstawowy, scalanie



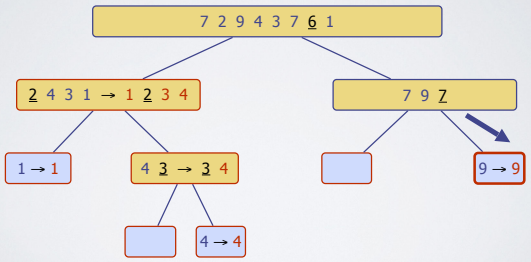
PRZYKŁAD DZIAŁANIA

Wywołanie rekurencyjne, wybór piwota



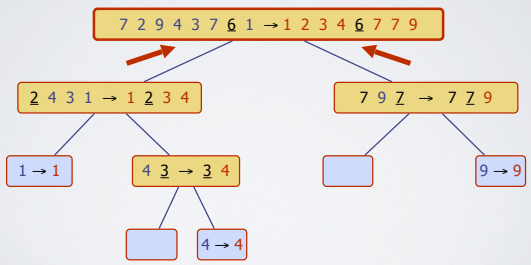
PRZYKŁAD DZIAŁANIA

Podział, wywołanie rekurencyjne, wybór piwota



PRZYKŁAD DZIAŁANIA

Scalanie, scalanie



NAJGORSZA ZŁOŻONOŚĆ OBLICZENIOWA

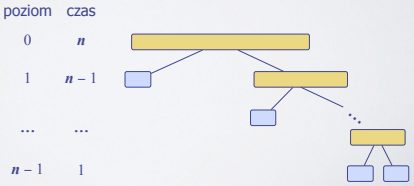
Z najgorszym czasem sortowania szybkiego mamy do czynienia, gdy piwot jest unikalnym minimalnym lub maksymalnym elementem

Jedna z list L i G ma rozmiar n − 1, a druga ma rozmiar 0

Złożoność obliczeniowa jest proporcjonalna do sumy:

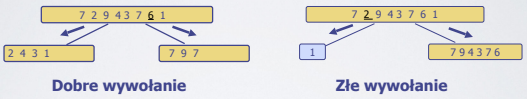
$$n + (n - 1) + \ldots + 2 + 1$$

Zatem najgorszy czas tego sortowania to $O(n^2)$



OCZEKIWANA ZŁOŻONOŚĆ OBLICZENIOWA

Rozważmy wywołanie rekurencyjne sortowania szybkiego na sekwencji o rozmiarze s
- Dobre wywołanie: rozmiary L i G są mniejsze od 3s/4
- Złe wywołanie: jedna z sekwencji ma rozmiar większy niż 3s/4



Dobre wywołanie **Złe wywołanie**

Wywołanie jest dobre z prawdopodobieństwem 1/2
- 1/2 z możliwych piwotów skutkuje dobrym wywołaniem

Złe piwoty **Dobre piwoty** **Złe piwoty**

OCZEKIWANA ZŁOŻONOŚĆ OBLICZENIOWA

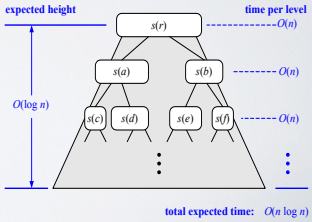
Fakt probabilistyczny: Oczekiwana ilość rzutów monetą wymagana do wyrzucenia k reszek wynosi 2k

Dla węzła na poziomie i, możemy oczekiwać:
- i/2 potomków będzie dobrym wywołaniem
- Rozmiar sekwencji wejściowej dla danego wywołania wynosi najwyżej $(3/4)^{i/2}n$

Mamy zatem:
- Dla węzła o poziomie $2\log_{4/3}n$, oczekiwany rozmiar wejścia to jeden
- Oczekiwana wysokość drzewa sortowania to O(log n)

Ilość obliczeń wykonywanych w węzłach o tym samym poziomie wynosi O(n)

Zatem oczekiwana złożoność obliczeniowa sortowania szybkiego jest w O(n log n)



SORTOWANIE SZYBKIE IN-SITU

Sortowanie szybkie może być zaimplementowane techniką in-situ.
W kroku dzielącym zmodyfikujemy operacje tak, aby przeorganizować elementy w tablicy w taki sposób, że
- elementy <= piwot będą posiadały indeks < l
- piwot posiada indeks l
- elementy >= piwot będą posiadały indeks > l

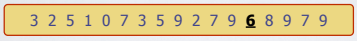
Wywołanie rekurencyjne będzie zawierać
- elementy z indeksami < l
- elementy z indeksami > l

Algorytm *inPlaceQuickSort(S, a, b)*
Wejście lista *S*, indeksy *a* i *b*
Wyjście lista *S* z elementami o indeksach między *a* i *b* przeorganizowana w sposób rosnący
if $a \geq b$
return
$i \leftarrow$ losowy integer między *a* i *b*
S.swapElements(i, b) {piwot na końcu}
$l \leftarrow$ *inPlacePartition(a, b)*
inPlaceQuickSort(S, a, l − 1)
inPlaceQuickSort(S, l + 1, b)

PODZIAŁ IN-SITU

Najpierw należy wybrać piwot na indeksie i między a i b.

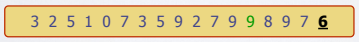
a i b



(piwot = 6)

Zamień piwot z elementem na indeksie b.
Umieść indeks startowy l na indeksie a,
a indeks r na indeksie b-1.

l r

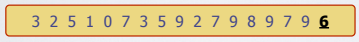


(piwot = 6)

PODZIAŁ IN-SITU

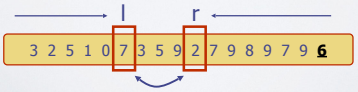
Dokonaj podziału z zastosowaniem dwóch indeksów do rozdzielenia S na L i G (pozostałe wartości = piwot mogą się znajdować zarówno w L jak i w G).

l r



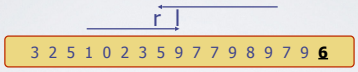
(piwot = 6)

Powtarzaj dopóki l i r się nie przetną:
- Przeskanuj indeksy w prawo aż do odnalezienia elementu > x.
- Przeskanuj r w lewo aż do odnalezienia elementu < x.
- Jeśli l < r, zamień elementy na indeksach l i r



PODZIAŁ IN-SITU

Kiedy l i r miną się w taki sposób, że l > r, możemy zamienić piwot z elementem na pozycji l

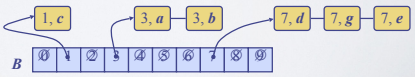


Wywołaj krok rekurencyjny dla podsekwencji od indeksu a do l-1 i podsekwencji dla indeksów l+1 do b

PODSUMOWANIE ALGORYTMÓW SORTOWANIA

Algorytm	Czas	Uwagi
przez wybór	$O(n^2)$	in-situ wolne (dobre dla małych wejść)
przez wstawianie	$O(n^2)$	in-situ wolne (dobre dla małych wejść)
szybkie	$O(n \log n)$ oczekiwany $O(n^2)$ najgorszy	in-situ, losowe najszybsza (dobra dla dużych danych)
przez kopcownie	$O(n \log n)$	in-situ szybka (dobra dla dużych danych)
przez scalanie	$O(n \log n)$	sekwencyjny dostęp do danych szybka (dobra dla b. dużych danych)

SORTOWANIE KUBEŁKOWE I POZYCYJNE



SORTOWANIE KUBEŁKOWE

Niech S będzie sekwencją n wpisów (klucz, wartość) z kluczami w przedziale [0, N − 1]
Sortowanie kubełkowe (Bucket-sort) wykorzystuje klucze jako indeksy w zewnętrznej tablicy B sekwencji (kubełki)

Faza 1: Przenosimy wszystkie wpisy (k,o) S do kubła B[k]

Faza 2: Dla i = 0, …, N − 1, przenieś wpisy kubła B[i] na koniec sekwencji S

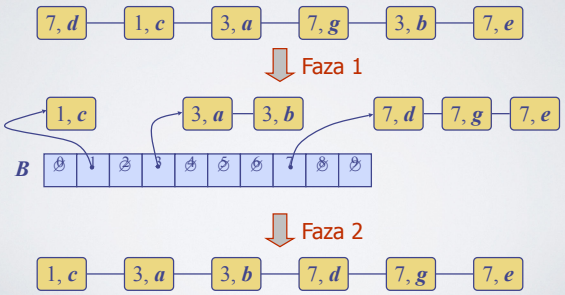
Analiza:
- Faza 1 zabiera O(n)
- Faza 2 zabiera O(n + N)

Sortowanie kubełkowe zabiera czas O(n + N)

Algorytm *bucketSort*(*S*, *N*)
Wejście lista *S* z wpisami o kluczach w przedziale [0, *N* − 1]
Wyjście sekwencja *S* posortowana rosnąco względem kluczy
B ← tablica z *N* pustymi sekwencjami
for każda pozycja *p* w *S* **do**
e ← *S*.*remove*(*p*)
k ← *e*.*getKey*()
B[*k*].*addLast*(*e*)
for *i* ← 0 **to** *N* − 1
for każdy wpis *e* w *B*[*i*] **do**
p ← *B*[*i*].*first*()
e ← *B*[*i*].*remove*(*p*)
S.*addLast*(*e*)

PRZYKŁAD

Zakres kluczy [0, 9]



WŁAŚCIWOŚCI I ROZSZERZENIA

Właściwości typu kluczy
- Klucze są wykorzystywane jako indeksy tablicy i nie mogą być dowolnymi obiektami
- Brak zewnętrznego komparatora

Właściwość stałości sortowania
- Relatywne uporządkowanie dowolnych dwóch wpisów o tym samym kluczu jest zachowane po zakończeniu działania algorytmu

Rozszerzenia
- Klucze są integerami w przedziale [a, b]
  - Umieść wpis (k, o) w kuble B[k − a]
- Klucze z łańcuchami znakowymi ze zbioru D możliwych łańcuchów, gdzie D ma stały rozmiar (e.g., nazwy krajów członkowskich UE)
  - Posortuj D i wyznacz indeks r(k) dla każdego łańcucha k w D w posortowanej sekwencji
  - Umieść wpis (k,o) w kuble B[r(k)]

UPORZĄDKOWANIE LEKSYKOGRAFICZNE

d-krotka jest sekwencją d kluczy ($k_1, k_2, \ldots, k_d$), gdzie klucz k_i będzie i-tym wymiarem krotki
Przykład:
Współrzędne Kartezjańskie punktu w przestrzeni są 3-krotką
Uporządkowanie leksykograficzne dwóch d-krotek jest zdefiniowane rekurencyjnie w następujący sposób:

$$(x1, x2, \ldots, xd) < (y1, y2, \ldots, yd)$$
$$\Leftrightarrow$$
$$x1 < y1 \ \vee \ x1 = y1 \wedge (x2, \ldots, xd) < (y2, \ldots, yd)$$

Tzn., krotki są porównywane z przez pierwszy wymiar, potem przez drugi, itd.

SORTOWANIE LEKSYKOGRAFICZNE

Niech C_i będzie komparatorem, który porównuje dwie krotki
Niech stableSort(S, C) będzie stałym algorytmem sortowania wykorzystującym komparator C
Sortowanie leksykograficzne sortuje sekwencję d-krotek w sposób leksykograficzny poprzez wywołanie d razy algorytmu stableSort. Raz dla każdego wymiaru.
Sortowanie leksykograficzne działa w czasie O(dT(n)), gdzie T(n) jest czasem działania stableSort

Algorytm *lexicographicSort*(***S***)
Wejście sekwencja ***S*** z ***d***-krotkami
Wyjście sekwencja ***S*** posortowana leksykograficznie

for $i \leftarrow d$ **downto** 1
stableSort(***S***, C_i)

Przykład:
(7,4,6) (5,1,5) (2,4,6) (2, 1, 4) (3, 2, 4)
(2, 1, 4) (3, 2, 4) (5,1,5) (7,4,6) (2,4,6)
(2, 1, 4) (5,1,5) (3, 2, 4) (7,4,6) (2,4,6)
(2, 1, 4) (2,4,6) (3, 2, 4) (5,1,5) (7,4,6)

SORTOWANIE POZYCYJNE

Sortowanie pozycyjne jest specyficzną odmianą sortowania leksykograficznego, które wykorzystuje sortowanie kubełkowe jako algorytm sortowania stałego.
Sortowanie pozycyjne ma zastosowanie do krotek, których klucze w każdym wymiarze są integerami w przedziale [0, N − 1]
Sortowanie pozycyjne działa w czasie O(d(n + N))

Algorytm *radixSort*(***S***, ***N***)
Wejście sekwencja ***S*** z ***d***-krotkami takie, że $(0, \ldots, 0) \le (x_1, \ldots, x_d)$ i $(x_1, \ldots, x_d) \le (N-1, \ldots, N-1)$ dla każdej krotki $(x_1, \ldots, x_d)$ w ***S***
Wyjście sekwencja ***S*** posortowana leksykograficznie
for $i \leftarrow d$ **downto** 1
bucketSort(***S***, ***N***)

Tutaj 18.04

PRZYKŁAD

Posortuj podane wartości z zastosowaniem sortowania pozycyjnego:
{21, 38, 241, 973, 100, 333}

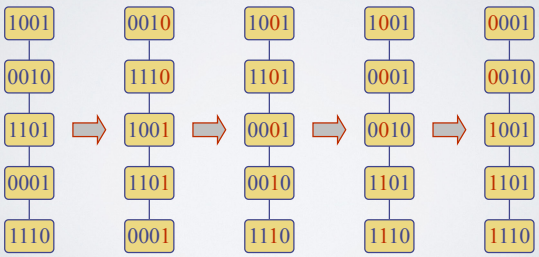
SORTOWANIE POZYCYJNE DLA LICZB BINARNYCH

Rozważ sekwencję n-bitowych integerów
$x = x_{b-1} \ldots x_1 x_0$
Reprezentujemy każdy element jako b-krotkę integerów z przedziału [0, 1] i zastosujemy sortowanie pozycyjne z N = 2
Ta modyfikacja algorytmu pozycyjnego działa w czasie O(bn)
Dla przykładu, możemy posortować sekwencję 32-bitowych integerów w liniowym czasie

Algorytm *binaryRadixSort*(***S***)
Wejście sekwencja ***S*** z ***b***-bitowymi integerami
Wyjście posortowana sekwencja ***S***
zamień każdy element x z ***S*** z elementem (0, ***x***)
for $i \leftarrow 0$ **to** $b - 1$
zamień klucz ***k*** każdego elementu (***k***, ***x***) w ***S*** z bitem x_i
bucketSort(***S***, 2)

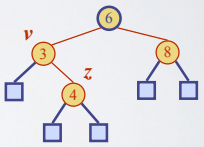
PRZYKŁAD

Sortowanie sekwencji 4-bitowych integerów



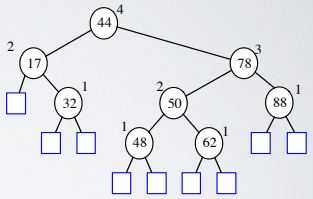
DRZEWA AVL

Adel'son-Vel'skii & Landis



DEFINICJA DRZEW AVL

- Drzewa AVL są zbalansowane
- Drzewo AVL jest binarnym drzewem przeszukiwań, w którym dla każdego węzła wewnętrznego v wysokość jego potomków może się różnić najwyżej o 1.



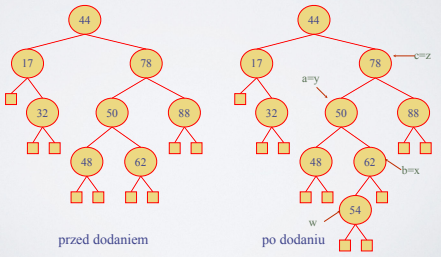
Przykład drzewa AVL z zaznaczonymi wysokościami

WYSOKOŚĆ DRZEWA AVL

- **Fakt**: Wysokość drzewa AVL przechowującego n kluczy wynosi O(log n)
- **Dowód**: Wprowadźmy ograniczenie **n(h)**: najmniejsza ilość węzłów wewnętrznych drzewa AVL o wysokości h.
  - Widzimy zatem, że n(1) = 1 i n(2) = 2
  - Dla n > 2, drzewo AVL o wysokości h zawiera korzeń, jedno poddrzewo AVL o wys. h-1 i drugie podrzewo o wys. h-2
  - zatem n(h) = 1 + n(h-1) + n(h-2)
  - Wiedząc, że n(h-1) > n(h-2), mamy n(h) > 2n(h-2). Zatem
    - n(h) > 2n(h-2), n(h) > 4n(h-4), n(h) > 8n(h-6), … (przez indukcję)
    - $n(h) > 2^i(h-2i)$
  - rozwiązując krok podstawowy (h-2i = 2) otrzymamy $n(h) > 2^{h/2-1}$
  - nakładając logarytm h < 2log n(h) + 2
  - Zatem wysokość drzewa AVL jest w O(log n)

DODAWANIE ELEMENTU DO DRZEWA AVL

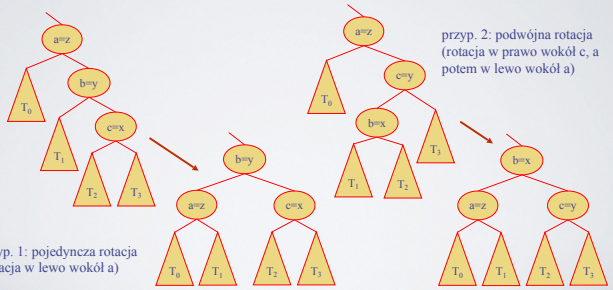
- Dodawanie elementu jest takie samo jak w binarnym drzewie przeszukiwań
- Zawsze wykonywane poprzez rozszerzenie węzła zewnętrznego
- Przykład:



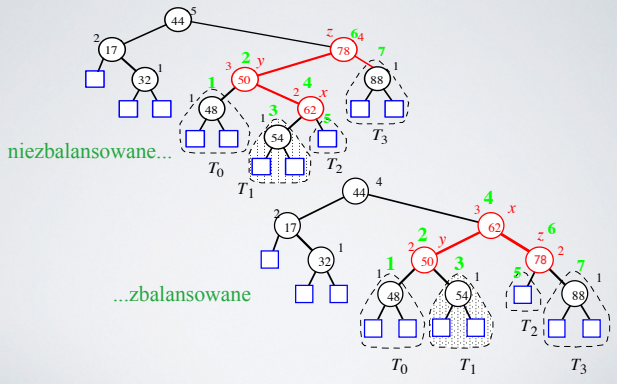
przed dodaniem po dodaniu

ROTACJA WĘZŁÓW AVL

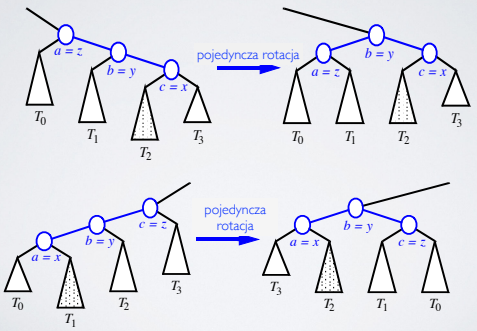
- niech (a, b, c) będzie listą ścieżki inorder dla x, y, z
- wykonaj rotacje niezbędne do umieszczenia b na górze drzewa



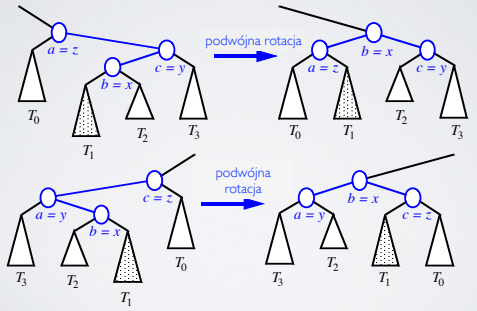
DODAWANIE - C.D.



RESTRUKTURYZACJA - JAKO POJEDYNCZA ROTACJA

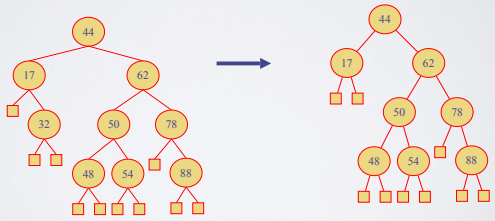


RESTRUKTURYZACJA - JAKO PODWÓJNA ROTACJA



USUWANIE Z DRZEWA AVL

- Usuwanie rozpoczyna się w taki sam sposób jak w binarnym drzewie przeszukiwań
  - usunięty węzeł będzie pustym zewnętrznym węzłem
    - jego ojciec, w, może spowodować brak zbalansowania
- Przykład:

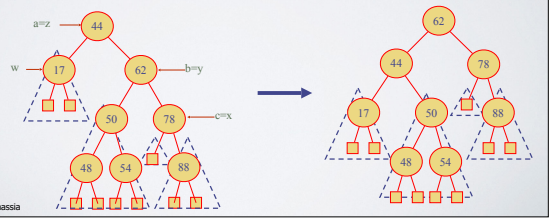


przed usunięciem 32

po usunięciu

BALANSOWANIE DRZEWA PO USUNIĘCIU

- Niech z będzie pierwszym niezbalansowanym węzłem napotkanym przy trawersowaniu drzewa w górę począwszy od węzła w. Jednocześnie niech y będzie synem z o większej wysokości i niech x będzie synem y o większej wysokości
- wywołujemy restructure(x) do przywrócenia balansu w z.
- Ponieważ restrukturyzacja może naruszyć balans w węzłach na wyższych poziomach drzewa musimy kontynuować sprawdzanie aż do osiągnięcia korzenia.



ZŁOŻONOŚĆ OBLICZENIOWA DRZEW AVL

- pojedyncza zmiana - O(1)
  - stosując strukturę listy dla drzewa binarnego
- szukanie - O(log n)
  - wysokość drzewa - O(log n), brak restrukturyzacji
- wstawianie - O(log n)
  - wstępne szukanie - O(log n)
  - Restrukturyzacja dla zachowania wysokości - O(log n)
- usuwanie - O(log n)
  - wstępne szukanie - O(log n)
  - Restrukturyzacja dla zachowania wysokości - O(log n)

PORÓWNANIE IMPLEMENTACJI SŁOWNIKÓW

Metoda	Tablica Haszująca (później)	Tablica przeszukiwań (Lista uporządkowana)	BST $\log n \le h \le n$	AVL
size, isEmpty	$O(1)$	$O(1)$	$O(1)$	$O(1)$
entries	$O(n)$	$O(n)$	$O(n)$	$O(n)$
find	$O(1)$ oczekiwany, $O(n)$ przyp. najgorszy	$O(\log n)$ oczekiwany, $O(\log n)$ przyp. najgorszy	$O(h)$	$O(\log n)$
findAll	$O(1 + s)$ oczekiwany, $O(n)$ przyp. najgorszy	$O(\log n + s)$ oczekiwany, $O(\log n + s)$ przyp. najgorszy	$O(h + s)$	$O(\log n + s)$
insert	$O(1)$ oczekiwany, $O(n)$ przyp. najgorszy	$O(n)$ oczekiwany, $O(n)$ przyp. najgorszy	$O(h)$	$O(\log n)$
remove	$O(1)$ oczekiwany, $O(n)$ przyp. najgorszy	$O(n)$ oczekiwany, $O(n)$ przyp. najgorszy	$O(h)$	$O(\log n)$