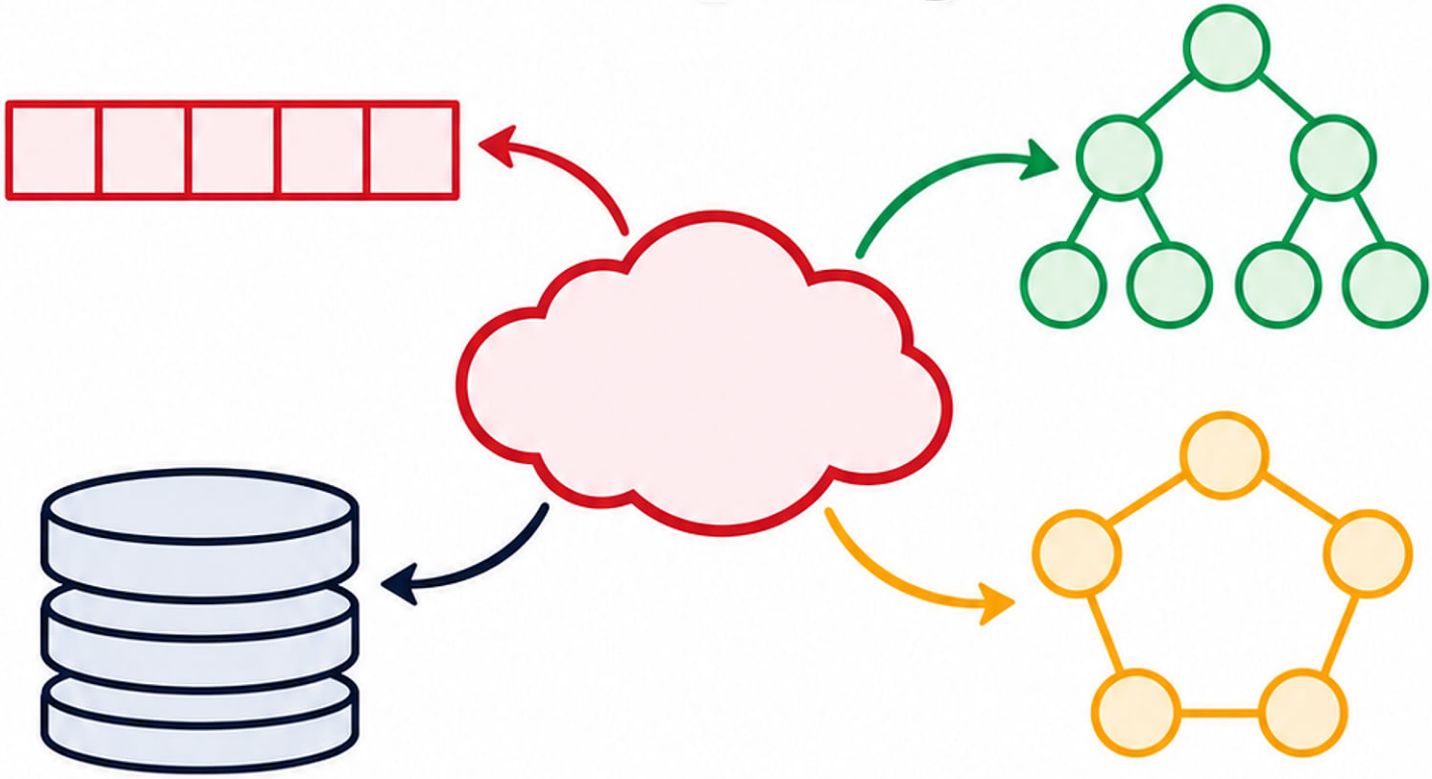


Java + struktury danych i algorytmy

50 map myśli



Wydanie rozszerzone: 64 mapy

Tablice, listy, stosy, kolejki, wyszukiwanie, sortowanie, drzewa i grafy

PRAWA AUTORSKIE

Java + struktury danych i algorytmy: 50 map myśli

Wydanie I, 2026.

Wszelkie prawa zastrzeżone. Materiał jest przeznaczony do nauki własnej. Możesz go czytać na telefonie, tablecie i komputerze oraz drukować na własny użytek. Kopiowanie, udostępnianie i sprzedaż całości lub części bez zgody autora są zabronione, chyba że pozwala na to zakupiona licencja.

Przykłady kodu zamieszczone w mapach możesz swobodnie wykorzystywać we własnych ćwiczeniach i projektach.

Materiał ma charakter edukacyjny. Efekty nauki zależą od samodzielnej pracy i regularnej praktyki.

Java jest zarejestrowanym znakiem towarowym firmy Oracle i jej podmiotów stowarzyszonych. Ten materiał jest niezależny i nie ma oficjalnego związku z firmą Oracle.

Spis treści

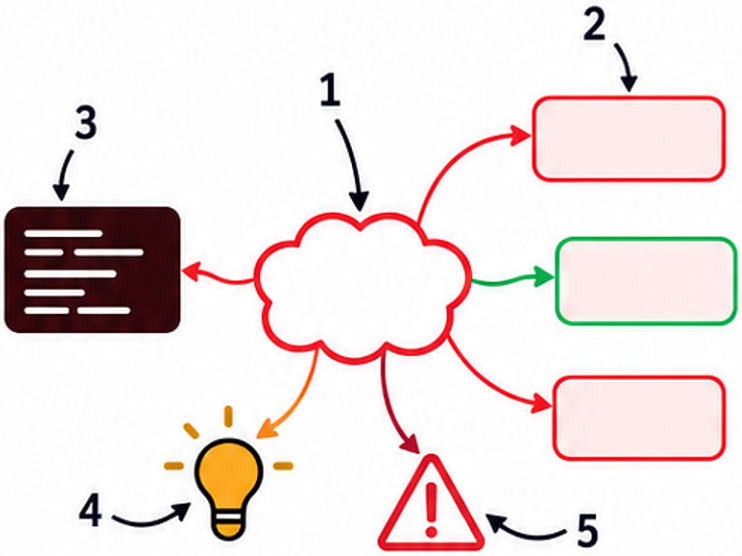
- **Jak korzystać z map** . . . . . — 03

- **Moduł 1. Złożoność obliczeniowa** (mapy 01–05) . . . . . — 04
- **Moduł 2. Tablice** (mapy 06–11) . . . . . — 10
- **Moduł 3. Listy** (mapy 12–19) . . . . . — 17
- **Moduł 4. Stosy** (mapy 20–25) . . . . . — 26
- **Moduł 5. Kolejki** (mapy 26–32) . . . . . — 33
- **Moduł 6. Wyszukiwanie** (mapy 33–39) . . . . . — 41
- **Moduł 7. Sortowanie** (mapy 40–48) . . . . . — 49
- **Moduł 8. Drzewa** (mapy 49–56) . . . . . — 59
- **Moduł 9. Grafy** (mapy 57–64) . . . . . — 68

- **Ściągawka 1:** koszt operacji w strukturach danych . . . . . — 77
- **Ściągawka 2:** która struktura do jakiego problemu . . . . . — 78
- **Co dalej?** Plan powtórek . . . . . — 79

Jak korzystać z map

Ten zestaw nie uczy składni od zera. Pokazuje, jak struktury danych działają od środka i ile kosztuje każda operacja. Dzięki temu łatwiej wybrać właściwe narzędzie i zrozumieć, dlaczego program działa szybko albo wolno.



Obiecaliśmy 50 map, a dostajesz 64. Każda mapa to jedno zagadnienie na jednej stronie. Moduły są ułożone od najprostszych struktur do grafów, ale do każdej mapy możesz wrócić osobno, na przykład w trakcie rozwiązywania zadania.

Elementy mapy

1. **Węzeł centralny** — temat i główne pytanie.
2. **Gałęzie** — najważniejsze pojęcia i zasady.
3. **Kod** — krótki przykład w Javie.
4. **Żarówka** — wskazówka do zapamiętania.
5. **Trójkąt** — typowy błąd, którego warto unikać.



Jak się uczyć

Przeczytaj mapę, przepisz kod w swoim edytorze i zmień jedną rzecz. Sprawdź, co się stanie. Mapa wyjaśnia, a praktyka utrwala.

MODUŁ 1



Złożoność obliczeniowa

Jak ocenić, czy algorytm poradzi sobie z dużą ilością danych

01 Czym jest złożoność algorytmu

02 Notacja O w praktyce

03 Złożoność pamięciowa

04 Jak policzyć kroki pętli

05 Przypadek najlepszy, średni i najgorszy

MAPA 01 | ZŁOŻONOŚĆ OBLICZENIOWA

Czym jest złożoność algorytmu

Jak rośnie koszt, gdy rośnie ilość danych?

Co mierzymy

Liczbę podstawowych kroków, np. porównań i przypisań. Nie mierzymy sekund, bo zależą od komputera.

Tempo wzrostu

Liczy się kształt wzrostu, a nie dokładna liczba. 3n + 5 i *n* rosną tak samo: liniowo.

Rozmiar danych n

n to liczba elementów: długość tablicy, liczba węzłów listy albo wierzchołków grafu.

Czas i pamięć

Złożoność czasowa mówi, ile kroków wykona algorytm. Pamięciowa mówi, ile dodatkowej pamięci zajmie.

Po co to wiedzieć

Żeby wybrać dobrą strukturę, zanim program zwolni przy dużej ilości danych.

```java
// suma elementów: jeden krok na element
int sum = 0;
for (int x : arr) {
    sum += x; // wykona się n razy
}
```

Zapamiętaj

Dla *n* = 10 każdy algorytm jest szybki. Różnica ujawnia się przy milionie elementów.

Notacja O w praktyce

Najważniejsze klasy złożoności od najszybszej



Ile kroków dla n = 1000?

O(log n)	O(n)	O(n log n)	O(n²)
około 10	1000	około 10 000	1 000 000

Pomijamy stałe

O(2n) to nadal O(n). Stała nie zmienia tempa wzrostu.

Zostaje najszybciej rosnący składnik

O(n² + n) to O(n²). Przy dużym n mniejszy składnik przestaje mieć znaczenie.

MAPA 03 | ZŁOŻONOŚĆ OBLICZENIOWA

Złożoność pamięciowa

Ile dodatkowej pamięci potrzebuje algorytm?

Pamięć dodatkowa

Liczymy tylko to, co algorytm tworzy oprócz danych wejściowych.

Rekurencja

Każde wywołanie zajmuje miejsce na stosie. Głębokość n oznacza O(n) pamięci.

O(1)

Kilka zmiennych, np. licznik i indeks, niezależnie od n.

Czas za pamięć

Często płacimy pamięcią za szybkość: HashSet przyspiesza sprawdzanie, ale zajmuje miejsce.

O(n)

Nowa tablica, lista lub mapa o rozmiarze danych wejściowych.

Rozmiary w Javie

int zajmuje 4 bajty. Referencja to zwykle 4 lub 8 bajtów, a każdy obiekt ma jeszcze nagłówek.

```java
// O(n) pamięci: nowa tablica
int[] copy = Arrays.copyOf(arr, arr.length);
```

```java
// O(1) pamięci: zamiana w miejscu
int tmp = arr[i];
arr[i] = arr[j];
arr[j] = tmp;
```

Uwaga

Zbyt głęboka rekurencja kończy się błędem StackOverflowError.

Jak policzyć kroki pętli

Czytaj kod od środka i mnóż liczby powtórzeń

```
for (int i = 0; i < n; i++) {     // n razy
   for (int j = 0; j < n; j++) {  // n razy dla każdego i
      count++;                    // razem n · n
   }
}
```

Pętle po kolei

Dwie pętle jedna po drugiej: n + n kroków, czyli O(n).

Pętla w pętli

Mnożymy: n · n kroków, czyli $O(n^2)$.

Pętla do i

for (j = 0; j < i; j++) daje 0 + 1 + ... + (n – 1) = n(n – 1)/2.
To nadal $O(n^2)$.

Dzielenie na pół

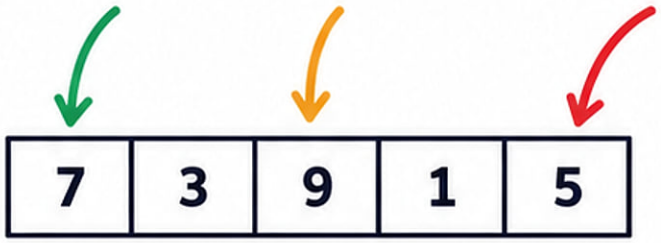
for (i = n; i > 1; i /= 2) wykona się około $\log_2 n$ razy: O(log n).

Metody w pętli

Licz koszt wywołań. list.contains(x) w pętli po n elementach to O(n) · O(n) = $O(n^2)$.

Ukryty koszt

Sklejanie tekstu przez += w pętli tworzy za każdym razem nowy obiekt String. Użyj StringBuilder.



Przypadek najlepszy, średni i najgorszy

Ten sam algorytm, różne dane, różny koszt

Najlepszy

Szukana liczba jest na pierwszym miejscu: 1 porównanie, O(1).

Średni

Liczba leży gdzieś w środku: około n/2 porównań, O(n).

Najgorszy

Liczba jest na końcu albo jej nie ma: n porównań, O(n).

Który podajemy?

Zwykle najgorszy przypadek, bo daje gwarancję.

Koszt zamortyzowany

ArrayList.add() zwykle trwa O(1), a rzadko O(n). Średnio na jedną operację wychodzi O(1).

Sortowanie szybkie

Średnio O(n log n), ale w najgorszym przypadku $O(n^2)$.

Dane mają znaczenie

Sortowanie przez wstawianie na prawie posortowanych danych działa niemal liniowo.

Zapamiętaj

Pytaj nie tylko „jak szybko?“, ale też „dla jakich danych?“.

MODUŁ 2



Tablice

Najprostsza struktura, na której opiera się wiele innych

06 Tablica w pamięci

07 Operacje na tablicy i ich koszt

08 Tablice dwuwymiarowe

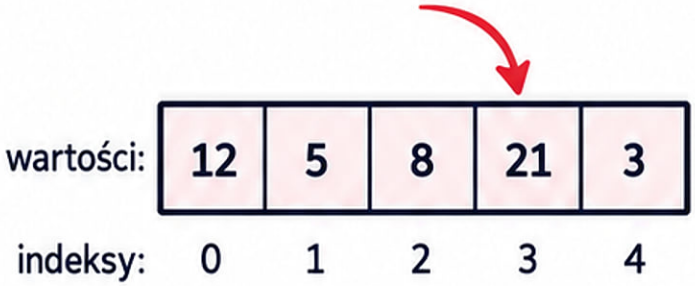
09 Klasa Arrays jako skrzynka narzędzi

10 Jak rośnie ArrayList

11 Dwa wskaźniki i okno przesuwne

Tablica w pamięci

Dlaczego odczyt arr[i] jest tak szybki?



Ciągły blok

Elementy leżą w pamięci jeden obok drugiego.

Indeks od zera

Pierwszy element to arr[0], ostatni to arr[arr.length - 1].

Dostęp O(1)

Adres = początek + indeks · rozmiar elementu.
Odczyt trwa tyle samo dla każdego i.

Stały rozmiar

Długości tablicy nie da się zmienić po jej utworzeniu.

Wartości domyślne

int: 0, boolean: false, obiekty: null.

Referencje

Tablica obiektów przechowuje referencje, a nie same obiekty.

```
int[] arr = new int[5]; // [0, 0, 0, 0, 0]
arr[0] = 12;
int last = arr[arr.length - 1];
```

⚠ Uwaga

arr[5] w tablicy o długości 5 rzuca ArrayIndexOutOfBoundsException.

Operacje na tablicy i ich koszt

Szybki odczyt, drogie wstawianie w środek

Dodanie na koniec

Jeśli jest wolne miejsce: O(1).

Przesuwanie

Wstawienie na indeks k przesuwa n – k elementów w prawo.

Tablica posortowana

Wyszukiwanie binarne skraca szukanie do O(log n).

Usuwanie bez przesuwania

Gdy kolejność nie ma znaczenia, wpisz ostatni element w miejsce usuwanego: O(1).



Operacja	Koszt	Dlaczego
Odczyt i zapis `arr[i]`	O(1)	bezpośredni adres
Wyszukiwanie wartości	O(n)	sprawdzamy kolejne elementy
Wstawienie lub usunięcie w środku	O(n)	przesuwanie elementów

```java
// wstawienie value na indeks k (size < arr.length)
for (int i = size; i > k; i--) {
    arr[i] = arr[i - 1];
}
arr[k] = value;
size++;
```

Tablice dwuwymiarowe

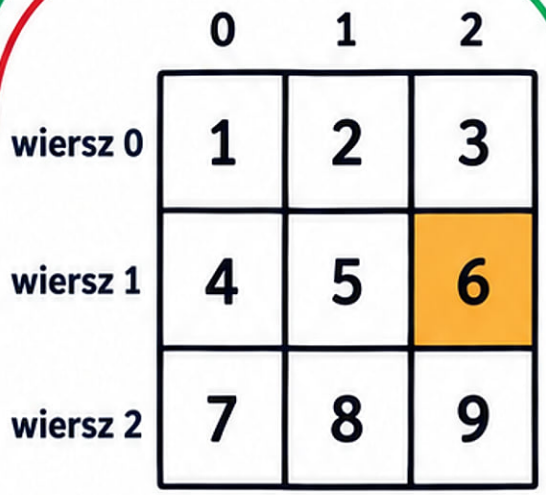
Wiersze i kolumny w jednej strukturze

TABLICA TABLIC

`int[][]` to tablica referencji do wierszy.

RÓŻNE DŁUGOŚCI

Każdy wiersz może mieć inną długość. To tak zwana tablica postrzępiona.



INDEKSY

`grid[r][c]`: najpierw wiersz, potem kolumna.

KOSZT PRZEJŚCIA

Dwie pętle: $O(r \cdot c)$ dla r wierszy i c kolumn.

WYMIARY

`grid.length` to liczba wierszy, `grid[0].length` to liczba kolumn.

KOLEJNOŚĆ

Przechodzenie wierszami jest zwykle szybsze, bo wiersz leży w jednym bloku pamięci.

```
int[][] grid = new int[3][3];
for (int r = 0; r < grid.length; r++) {
    for (int c = 0; c < grid[r].length; c++) {
        grid[r][c] = r * 3 + c + 1;
    }
}
```

SZYBKI PODGLĄD

`Arrays.deepToString(grid)` wypisze całą tablicę.

MAPA 09 | TABLICE

Klasa Arrays jako skrzynka narzędzi

Gotowe metody z pakietu java.util

sort

```
Arrays.sort(arr);
```

Sortuje tablicę w miejscu: O(n log n).

binarySearch

```
Arrays.binarySearch(arr, x);
```

Tylko na posortowanej tablicy: O(log n). Brak elementu daje wynik ujemny.

fill

```
Arrays.fill(arr, -1);
```

Wypełnia całą tablicę jedną wartością: O(n).

copyOf

```
Arrays.copyOf(arr, 10);
```

Tworzy nową tablicę, dłuższą lub krótszą: O(n).

equals

```
Arrays.equals(a, b);
```

Porównuje zawartość element po elemencie: O(n).

toString

```
Arrays.toString(arr);
```

Zwraca czytelny zapis, np. [1, 2, 3].

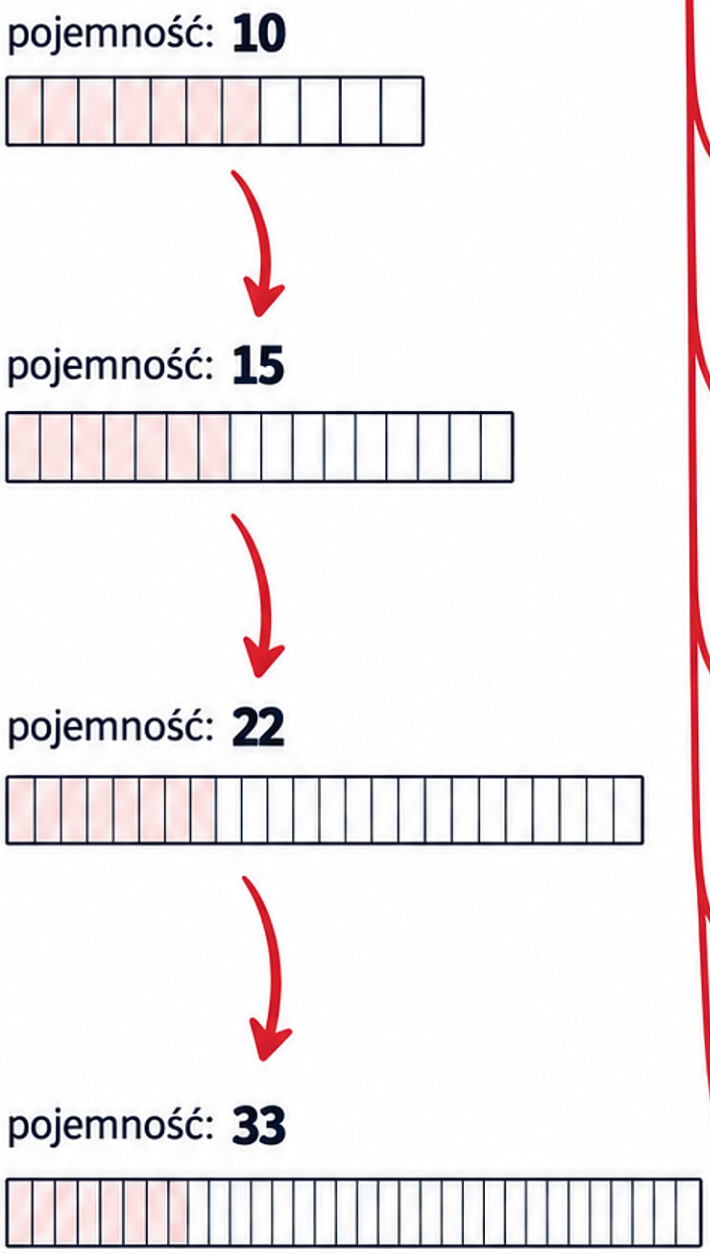
Częsty błąd

a == b porównuje referencje, a nie zawartość. Do porównania tablic użyj Arrays.equals(a, b).

MAPA 10 | TABLICE

Jak rośnie ArrayList

Tablica, która sama się powiększa



W środku tablica

ArrayList trzyma elementy w zwykłej tablicy **Object[]**.

size i pojemność

size to liczba elementów, pojemność to długość tablicy w środku.

Powiększanie

Gdy brakuje miejsca, powstaje tablica około 1,5 raza większa, a elementy są kopiowane: **O(n)**.

Zamortyzowane O(1)

Kopiowanie zdarza się rzadko, więc add() na koniec kosztuje średnio O(1).

Pojemność na start

new ArrayList<>(1000) pozwala uniknąć wielu powiększeń.

Usuwanie z przodu

remove(0) przesuwa wszystkie pozostałe elementy: O(n).

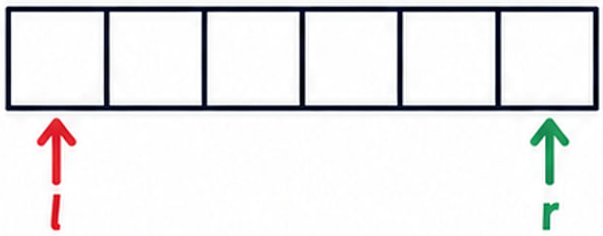
```
List<Integer> list = new ArrayList<>(1000);
list.add(7);    // O(1) zamortyzowane
list.get(0);    // O(1)
list.add(0, 5); // O(n): przesuwanie
```

Dwa wskaźniki i okno przesuwne

Dwie techniki, które zamieniają $O(n^2)$ w $O(n)$

Dwa wskaźniki

W posortowanej tablicy *l* startuje z lewej, *r* z prawej. Szukamy pary o sumie target.

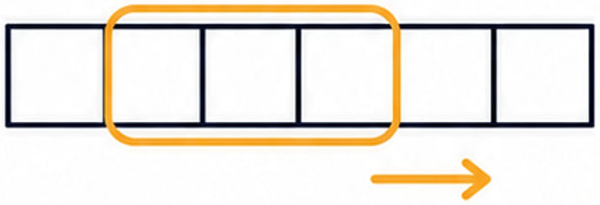



```java
int l = 0, r = arr.length - 1;
while (l < r) {
    int s = arr[l] + arr[r];
    if (s == target) return true;
    if (s < target) l++; else r--;
}
```

Koszt: $O(n)$ zamiast $O(n^2)$

Okno przesuwne

Największa suma *k* kolejnych elementów. Dodajemy element, który wchodzi do okna, i odejmujemy ten, który wychodzi.



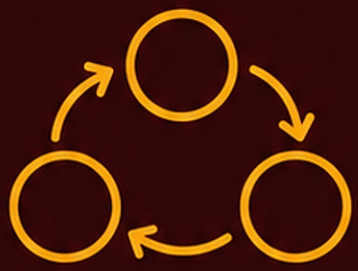
```java
int sum = 0;
for (int i = 0; i < k; i++) sum += arr[i];
int best = sum;
for (int i = k; i < arr.length; i++) {
    sum += arr[i] - arr[i - k];
    best = Math.max(best, sum);
}
```

Koszt: $O(n)$ zamiast $O(n \cdot k)$

Kiedy użyć

Słowa „para”, „podtablica” lub „kolejne elementy” w treści zadania to sygnał do tych technik.

MODUŁ 3



Listy

Elementy połączone referencjami zamiast jednego bloku pamięci

12 Lista jednokierunkowa

13 Wstawianie i usuwanie węzłów

14 Lista dwukierunkowa

15 LinkedList w Javie

16 ArrayList czy LinkedList?

17 Iterator i bezpieczne usuwanie

18 Odwracanie listy i szukanie środka

19 Wykrywanie cyklu i scalanie list

MAPA 12 | LISTY

Lista jednokierunkowa

Każdy węzeł wskazuje na następny



Węzeł

Przechowuje wartość i referencję next do następnego węzła.

head

Referencja do pierwszego węzła. Od niej zaczyna się każde przejście.

Koniec listy

Ostatni węzeł ma next równe null.

Rozproszenie

Węzły mogą leżeć w różnych miejscach pamięci.

Dostęp O(n)

Aby dojść do k-tego elementu, trzeba przejść k węzłów.

Na początek O(1)

Nowy węzeł wskazuje na stary head i sam staje się głową.

Zapamiętaj

Nigdy nie przesuwaj samego head podczas przeglądania. Użyj zmiennej pomocniczej cur.

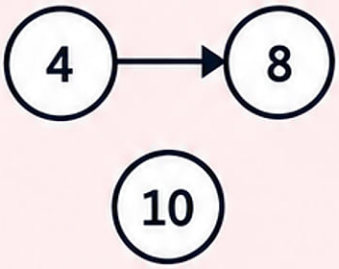
```
class Node {
   int value;
   Node next;
   Node(int value) { this.value = value; }
}

for (Node cur = head; cur != null; cur = cur.next) {
   System.out.println(cur.value);
}
```

Wstawianie i usuwanie węzłów

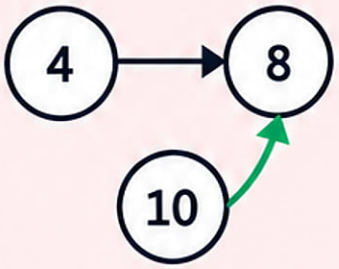
Przepinamy strzałki we właściwej kolejności

Wstawienie za węzłem a

1. 1. Utwórz nowy węzeł n.



2. 2. n.next = a.next



3. 3. a.next = n



Kolejność ma znaczenie

Jeśli najpierw ustawisz a.next = n, stracisz dostęp do reszty listy.

Koszt

Samo przepięcie to O(1), ale znalezienie miejsca to O(n).

Usuwanie

Omijamy węzeł: prev.next = prev.next.next. Resztą zajmie się garbage collector.

Pierwszy węzeł

Usuwanie head to osobny przypadek. Pomaga węzeł wartowniczy dummy przed głową.

```
Node n = new Node(10);
n.next = a.next;
a.next = n;
// usuwanie węzła za prev
prev.next = prev.next.next;
```

MAPA 14 | LISTY



Lista dwukierunkowa

Węzeł zna sąsiadów z obu stron

prev i next

Każdy węzeł ma referencję do poprzednika i do następnika.

head i tail

Szybki dostęp do obu końców: dodawanie i usuwanie na końcach w O(1).

Usuwanie znanego węzła

Mając referencję do węzła, przepinasz tylko dwie strzałki: O(1).

W obie strony

Listę można przeglądać od początku i od końca.

Cena

Dodatkowa referencja w każdym węźle to więcej pamięci i więcej przepięć.

```
// usunięcie węzła x ze środka listy
x.prev.next = x.next;
x.next.prev = x.prev;
```

⚠ Uwaga na końce

Przy pierwszym i ostatnim węźle prev lub next jest równe `null`. Sprawdź to, aby uniknąć NullPointerException.

MAPA 15 | LISTY

LinkedList w Javie

Gotowa lista dwukierunkowa z java.util

Dwa interfejsy

Implementuje List i Deque, więc działa jak lista i jak kolejka.

Końce w O(1)



addFirst, addLast, removeFirst, removeLast, getFirst i getLast.

get(i) w O(n)



Metoda idzie od bliższego końca, ale i tak przechodzi do n/2 węzłów.

add(i, x)

Samo wstawienie jest tanie, ale dojście do indeksu i kosztuje O(n).

Pamięć

Każdy element to osobny obiekt z trzema referencjami.

null

LinkedList pozwala przechowywać wartości null.

```
LinkedList<String> list = new LinkedList<>();
list.addFirst("B");
list.addLast("C");
list.addFirst("A");  // [A, B, C]
list.removeLast();   // [A, B]
```



Pułapka

Pętla for z *list*.get(*i*) na LinkedList to O(n²). Używaj pętli for-each albo iteratora.

ArrayList czy LinkedList?

Wybór zależy od tego, co robisz najczęściej

ArrayList

Częsty odczyt po indeksie.

Przechodzenie po kolei.

Dane w jednym bloku, więc lepiej działa pamięć podręczna procesora.

VS

LinkedList

Częste dodawanie i usuwanie na obu końcach.

Usuwanie przez iterator w środku listy.

Więcej pamięci na każdy element.

Operacja	ArrayList	LinkedList
get(i)	O(1)	O(n)
add na koniec	O(1)*	O(1)
add lub remove na początku	O(n)	O(1)

* koszt zamortyzowany

W praktyce

ArrayList wygrywa w większości zadań. Do kolejki dwustronnej lepiej wybrać ArrayDeque. W razie wątpliwości zmierz czas obu list na własnych danych, bo notacja O nie uwzględnia pamięci podręcznej.

MAPA 17 | LISTY

Iterator i bezpieczne usuwanie

Jak usuwać elementy podczas przeglądania kolekcji

Czym jest iterator

Obiekt, który przechodzi po kolekcji: `hasNext()`, `next()` i `remove()`.

for-each

Pętla for-each pod spodem używa iteratora.

removeIf

Najprostszy sposób usuwania według warunku.

```
Iterator<Integer> it = list.iterator();
while (it.hasNext()) {
   if (it.next() % 2 == 0) {
     it.remove(); // bezpieczne usuwanie
   }
}

// krócej:
list.removeIf(x -> x % 2 == 0);
```

ConcurrentModificationException

Pojawia się, gdy wywołasz `list.remove()` w pętli for-each.

remove() po next()

Wywołanie `remove()` bez wcześniejszego `next()` rzuca `IllegalStateException`.

Pętla z indeksem

Usuwając od przodu, pominiesz element. Idź od końca listy.

Także w mapach

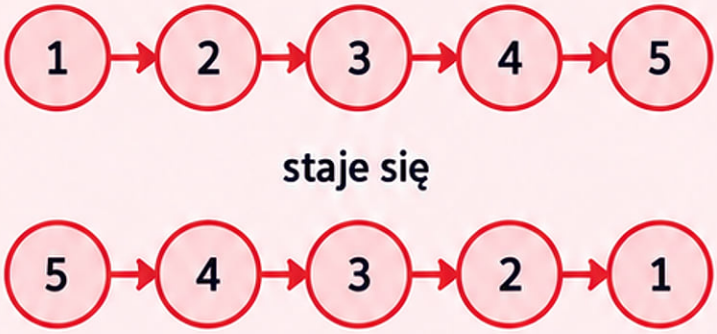
`map.entrySet().removeIf(e -> e.getValue() == 0)` bezpiecznie usuwa wpisy z HashMap.

Odwracanie listy i szukanie środka

Dwa klasyczne zadania w czasie O(n) i pamięci O(1)

Odwracanie

Trzy referencje: prev, cur i next. Każdy węzeł przestawiamy tak, aby wskazywał na poprzedni.

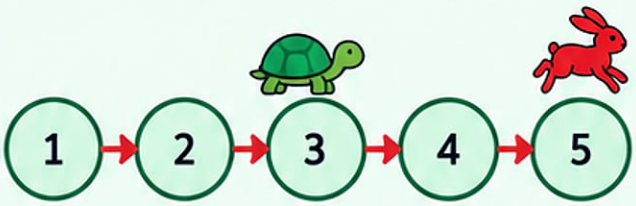



```
Node prev = null, cur = head;
while (cur != null) {
    Node next = cur.next;
    cur.next = prev;
    prev = cur;
    cur = next;
}
head = prev;
```

Środek listy

Wolny wskaźnik przesuwa się o 1 węzeł, szybki o 2. Gdy szybki dojdzie do końca, wolny jest w środku.

```
Node slow = head, fast = head;
while (fast != null && fast.next != null) {
    slow = slow.next;
    fast = fast.next.next;
}
// slow wskazuje środek
```



1 → 2 → 3 → 4 → 5: środek to 3

Zapamiętaj

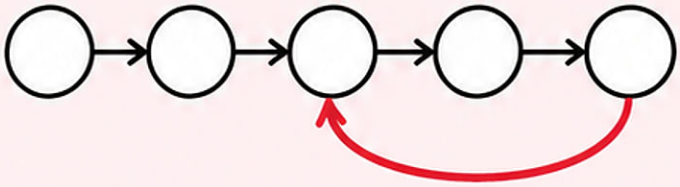
Zapisz cur.next, zanim go zmienisz. Inaczej zgubisz resztę listy.

MAPA 19 | LISTY

Wykrywanie cyklu i scalanie list

Żółw i zając oraz łączenie posortowanych list

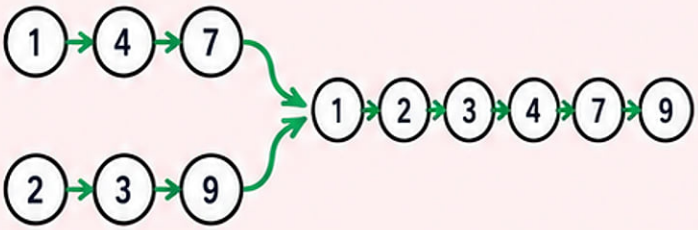
Cykl: algorytm Floyda



Wolny przesuwa się o 1, szybki o 2. Jeśli lista ma cykl, w końcu się spotkają. Jeśli szybki dojdzie do null, cyklu nie ma.

```
Node slow = head, fast = head;
while (fast != null && fast.next
    != null) {
    slow = slow.next;
    fast = fast.next.next;
    if (slow == fast) return true;
}
return false;
```

Scalanie dwóch list



Porównujemy głowy list a i b i dołączamy mniejszy węzeł. Węzeł dummy upraszcza początek.

```
Node dummy = new Node(0), tail =
  dummy;
while (a != null && b != null) {
   if (a.value <= b.value) {
      tail.next = a;
      a = a.next;
   } else {
      tail.next = b;
      b = b.next;
   }
   tail = tail.next;
}
tail.next = (a != null) ? a : b;
```

Koszt

Cykl: O(n) czasu i O(1) pamięci. **Scalanie:** O(n + m), bez kopiowania węzłów.

MODUŁ 4



Stosy

Ostatni wchodzi, pierwszy wychodzi

20 Stos i zasada LIFO

21 Stos na tablicy i na liście

22 ArrayDeque zamiast Stack

23 Sprawdzanie nawiasów

24 Stos wywołań i rekurencja

25 Odwrotna notacja polska

MAPA 20 | STOSY

Stos i zasada LIFO

Last In, First Out: ostatni wchodzi, pierwszy wychodzi

push(x)

Kładzie element na wierzch: O(1).

pop()

Zdejmuje i zwraca element z wierzchu: O(1).



wierzch: 3 — 2 — 1 :spód

peek()

Pokazuje wierzch bez zdejmowania: O(1). isEmpty() sprawdza, czy stos jest pusty.

Gdzie go spotkasz

Cofanie zmian w edytorze, przycisk „wstecz" w przeglądarce, stos wywołań metod, obliczanie wyrażeń.

Analogia

Stos talerzy: zawsze bierzesz ten z góry, czyli położony jako ostatni.

```
Deque<Integer> stack = new ArrayDeque<>();
stack.push(1);
stack.push(2);
stack.push(3);
stack.pop();  // 3
stack.peek(); // 2
```

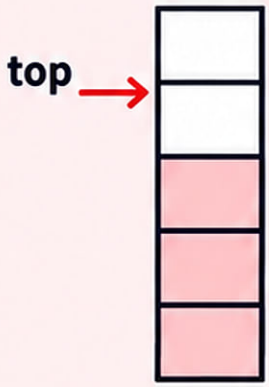
Pusty stos

`pop()` na pustym `ArrayDeque` rzuca `NoSuchElementException`. Najpierw sprawdź `isEmpty()`.

Stos na tablicy i na liście

Dwie własne implementacje tej samej idei

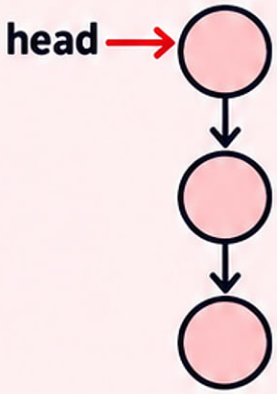
Na tablicy




```
int[] data = new int[100];
int top = 0;
void push(int x) { data[top++] = x; }
int pop() { return data[--top]; }
```

- **Plus:** dane w jednym bloku, bardzo szybko.
- **Minus:** stała pojemność albo powiększanie tablicy.

Na liście




```
Node head;
void push(int x) {
    Node n = new Node(x);
    n.next = head;
    head = n;
}
int pop() {
    int v = head.value;
    head = head.next;
    return v;
}
```

- **Plus:** brak limitu rozmiaru.
- **Minus:** każdy element to osobny obiekt.

Wspólny koszt

push i pop: O(1) w obu wersjach.

Uwaga

Przed pop() sprawdź, czy stos nie jest pusty.

ArrayDeque zamiast Stack

Dlaczego dokumentacja Javy poleca Deque

Cecha	Stack	ArrayDeque
Pochodzenie	Java 1.0, dziedziczy po Vector	Java 6, interfejs Deque
Synchronizacja	tak, więc wolniej	nie, więc szybciej
null	dozwolony	zabroniony

Stara klasa

Stack dziedziczy po Vector, więc pozwala sięgać po indeksie i łamać zasadę LIFO.

Tablica cykliczna

ArrayDeque trzyma elementy w tablicy, która rośnie w razie potrzeby.

Te same metody

push, pop i peek działają tak samo, więc zmiana jest prosta.

Kolejność iteracji

ArrayDeque przegląda od wierzchu, a Stack od spodu.

```
Deque<Character> stack = new ArrayDeque<>();
stack.push('a');
stack.push('b');
char top = stack.peek(); // 'b'
```

Zasada

W nowym kodzie deklaruj typ Deque, a twórz obiekt ArrayDeque.

Sprawdzanie nawiasów

Czy każdy nawias ma swoją parę?

1. **Nawias otwierający:** odłóż na stos oczekiwany nawias zamykający.
2. **Nawias zamykający:** zdejmij element ze stosu i porównaj.
3. **Pusty stos albo inny znak** oznacza błąd.
4. **Na końcu stos** musi być **pusty.**

```
boolean isValid(String s) {
  Deque<Character> st = new
   ArrayDeque<>();
  for (char c : s.toCharArray()) {
    if (c == '(') st.push(')');
    else if (c == '[') st.push(']');
    else if (c == '{') st.push('}');
    else if (st.isEmpty() ||
          st.pop() != c) return false;
  }
  return st.isEmpty();
}
```

Poprawnie

{[()]}

Błędnie

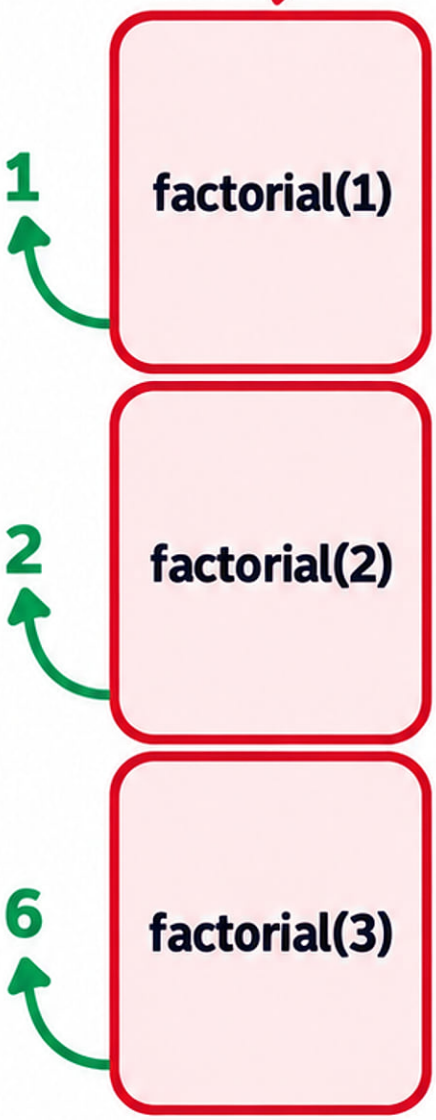
([)]

Koszt i założenie

O(n) czasu i O(n) pamięci. Kod zakłada, że napis zawiera tylko nawiasy.

Stos wywołań i rekurencja

Każde wywołanie metody to nowa ramka na stosie




```java
int factorial(int n) {
  if (n <= 1) return 1;          // warunek stopu
  return n * factorial(n - 1);  // rekurencja
}
```

RAMKA STOSU

Każde wywołanie dostaje własną ramkę: parametry, zmienne lokalne i adres powrotu.

WARUNEK STOPU

Bez niego metoda wywołuje się bez końca.

POWRÓT

Ramki są zdejmowane w odwrotnej kolejności, czyli zgodnie z zasadą LIFO.

PAMIĘĆ

`factorial(n)` zajmuje O(n) miejsca na stosie.

ZAMIANA NA PĘTLĘ

Każdą rekurencję da się zapisać pętlą z własnym stosem Deque. Unikasz wtedy StackOverflowError.

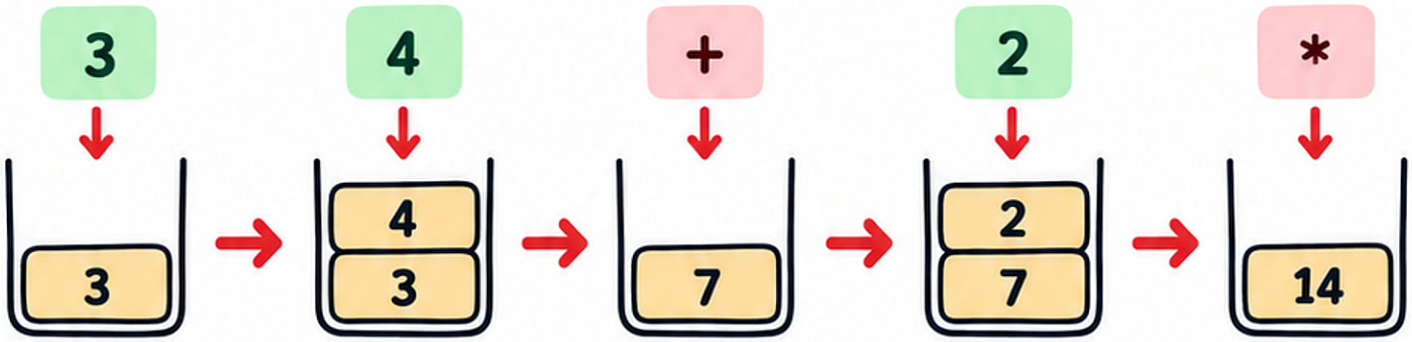
SPRAWDŹ SIĘ

Rozpisz ramki dla `factorial(4)`. Ile ramek będzie na stosie jednocześnie?

Odwrotna notacja polska

Wyrażenia bez nawiasów liczone stosem

3 4 + 2 * oznacza (3 + 4) * 2 = 14




```
int evalRpn(String[] tokens) {
  Deque<Integer> st = new ArrayDeque<>();
  for (String t : tokens) {
    switch (t) {
      case "+" -> st.push(st.pop() + st.pop());
      case "*" -> st.push(st.pop() * st.pop());
      case "-" -> { int b = st.pop(), a = st.pop();
                    st.push(a - b); }
      default -> st.push(Integer.parseInt(t));
    }
  }
  return st.pop();
}
```

Zasada

Liczba trafia na stos. Operator zdejmuje dwie liczby i odkłada wynik.

Kolejność argumentów

Przy odejmowaniu i dzieleniu pierwszy pop() to prawy argument.

Polski ślad

Notację polską opracował Jan Łukasiewicz. Wersję odwrotną stosuje się w kalkulatorach i kompilatorach.

Koszt Jedno przejście po symbolach: O(n) czasu i O(n) pamięci.

MODUŁ 5



Kolejki

Pierwszy wchodzi, pierwszy wychodzi

26 Kolejka i zasada FIFO

27 Kolejka cykliczna

28 Interfejs Queue

29 Deque, czyli kolejka dwustronna

30 Kolejka priorytetowa

31 Kolejka z dwóch stosów

32 Kolejki w praktyce

MAPA 26 | KOLEJKI

Kolejka i zasada FIFO

First In, First Out: kto pierwszy, ten lepszy



offer(x)

Dodaje element na koniec kolejki: O(1).

poll()

Usuwa i zwraca element z początku: O(1).

peek()

Pokazuje pierwszy element bez usuwania: O(1).

Dwa końce

Elementy wchodzą jednym końcem, a wychodzą drugim.

Gdzie ją spotkasz

Zadania dla drukarki, żądania do serwera, wiadomości między programami, przeszukiwanie wszerz.

Nie rób tak

ArrayList jako kolejka to zły pomysł: remove(0) kosztuje O(n).

```
Queue<String> queue = new ArrayDeque<>();
queue.offer("Ania");
queue.offer("Bartek");
queue.offer("Celina");
queue.poll(); // "Ania"
queue.peek(); // "Bartek"
```

Kolejka cykliczna

Tablica zamknięta w pierścień

Problem

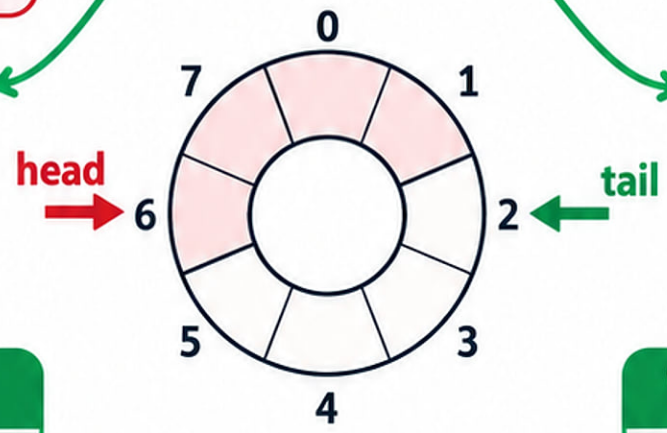
W zwykłej tablicy po usunięciu z przodu zostaje dziura albo trzeba przesuwać elementy: O(n).

Pełna czy pusta?

W obu przypadkach head == tail, dlatego liczymy size.

Zawijanie

Po ostatnim indeksie wracamy na 0 dzięki operacji modulo %.



Koszt

offer i poll: O(1), bez przesuwania.

head i tail

head wskazuje pierwszy element, a tail pierwsze wolne miejsce.

W Javie

Tak działa ArrayDeque. Gdy brakuje miejsca, tablica jest powiększana.

```
int[] data = new int[8];
int head = 0, tail = 0, size = 0;

void offer(int x) {
    data[tail] = x;
    tail = (tail + 1) % data.length;
    size++;
}

int poll() {
    int x = data[head];
    head = (head + 1) % data.length;
    size--;
    return x;
}
```

MAPA 28 | KOLEJKI

Interfejs Queue

Dwie rodziny metod do tych samych działań

Działanie	⚠ Rzuca wyjątek	✓ Zwraca wartość specjalną
Dodaj	`add(e)`	`offer(e)`: `false`
Usuń	`remove()`	`poll()`: `null`
Podejrzyj	`element()`	`peek()`: `null`

INTERFEJS, NIE KLASA

Queue opisuje zachowanie. Implementacje to m.in. ArrayDeque, LinkedList i PriorityQueue.

KTÓRĄ RODZINĘ WYBRAĆ

offer, poll i peek, gdy pusta kolejka to normalna sytuacja. add, remove i element, gdy to błąd programu.

```
Queue<Integer> q = new ArrayDeque<>();
q.offer(5);
Integer x = q.poll(); // 5
Integer y = q.poll(); // null: kolejka pusta
```

KOLEJKI Z LIMITEM

offer zwraca false, gdy kolejka jest pełna, np. w ArrayBlockingQueue.

BEZ NULL

Nie wstawiaj null: poll() używa go jako sygnału pustej kolejki.

Deque, czyli kolejka dwustronna

Dodawanie i usuwanie z obu końców



Przód

addFirst, pollFirst, peekFirst: O(1).

Tył

addLast, pollLast, peekLast: O(1).

Dwa w jednym

Deque zastępuje stos i kolejkę. Stos używa przodu, kolejka dodaje z tyłu, a usuwa z przodu.

Zastosowania

Sprawdzanie palindromów, maksimum w oknie przesuwnym, historia ostatnich działań.

```
Deque<Character> d = new ArrayDeque<>();
for (char c : "kajak".toCharArray()) d.addLast(c);
boolean ok = true;
while (d.size() > 1) {
    if (!d.pollFirst().equals(d.pollLast())) ok = false;
}
// ok == true: "kajak" to palindrom
```

Wybór klasy

ArrayDeque jest zwykle szybsza od LinkedList i zajmuje mniej pamięci.

Kolejka priorytetowa

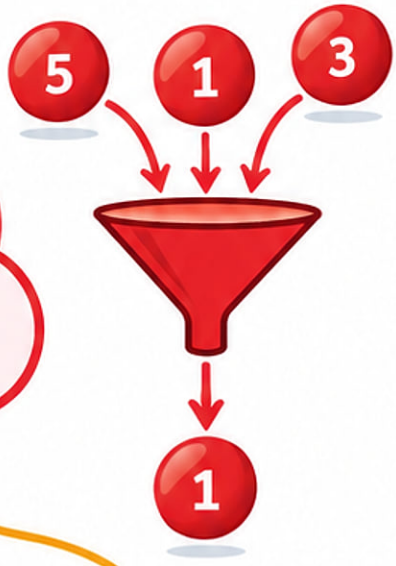
Najpierw wychodzi najważniejszy element

To nie FIFO

Wychodzi element o najwyższym priorytecie. Domyślnie jest to najmniejsza wartość.

Kopiec w środku

PriorityQueue to kopiec binarny w tablicy. Więcej w mapie 56.



Koszt

offer i poll: O(log n).
peek: O(1).

Comparator

Ustala własny priorytet, np. Comparator.reverseOrder() daje kolejkę od największej wartości.

Uwaga na iterację

Pętla for-each nie zwraca elementów według priorytetu. Posortowany wynik dają tylko kolejne wywołania poll().

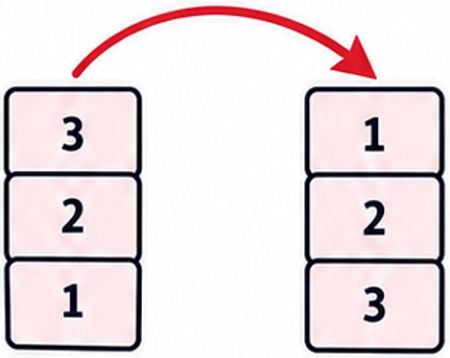
Zastosowania

Algorytm Dijkstry, szeregowanie zadań, k największych elementów.

```
PriorityQueue<Integer> pq = new PriorityQueue<>();
pq.offer(5);
pq.offer(1);
pq.offer(3);
pq.poll(); // 1

PriorityQueue<Task> tasks = new PriorityQueue<>(
    Comparator.comparingInt(Task::priority));
```

Kolejka z dwóch stosów

LIFO + LIFO = FIFO



in: 1, 2, 3 → out: 3, 2, 1 (na wierzchu 1)

```
class TwoStackQueue {
  Deque<Integer> in =
    new ArrayDeque<>();
  Deque<Integer> out =
    new ArrayDeque<>();

  void offer(int x) {
    in.push(x);
  }

  int poll() {
    if (out.isEmpty()) {
      while (!in.isEmpty())
        out.push(in.pop());
    }
    return out.pop();
  }
}
```

Pomysł

Przełożenie stosu odwraca kolejność elementów.

Kiedy przekładać

Tylko wtedy, gdy stos out jest pusty.

Koszt zamortyzowany

Każdy element trafia do in i do out tylko raz, więc średnio O(1).

Po co to znać

To częste zadanie na rozmowach rekrutacyjnych. Uczy liczenia kosztu zamortyzowanego.

Sprawdź się

Narysuj oba stosy po operacjach: offer(1), offer(2), poll(), offer(3), poll().

Kolejki w praktyce

Która kolejka do jakiego zadania?



Obsługa klientów

Zwykła kolejka FIFO: Queue i ArrayDeque. Kto pierwszy przyszedł, ten pierwszy zostaje obsłużony.

Izba przyjęć

Kolejka priorytetowa: PriorityQueue. Najpierw najpilniejszy przypadek, a nie najwcześniejszy.

Ostatnie działania

Deque z limitem: nowe na początek, najstarsze usuwamy z końca.

Wiele wątków

Kolejki z java.util.concurrent, np. LinkedBlockingQueue: jeden wątek dodaje zadania, drugi je pobiera.

```
// historia 5 ostatnich stron
Deque<String> history = new ArrayDeque<>();
void visit(String page) {
    history.addFirst(page);
    if (history.size() > 5) history.removeLast();
}
```

Zasada wyboru

Najpierw zapytaj, który element ma wyjść pierwszy: najstarszy, najnowszy czy najważniejszy. Odpowiedź od razu wskazuje właściwą kolejkę.

MODUŁ 6



Wyszukiwanie

Od przeglądania po kolei do haszowania w czasie O(1)

33 Wyszukiwanie liniowe

34 Wyszukiwanie binarne

35 Pułapki wyszukiwania binarnego

36 Haszowanie

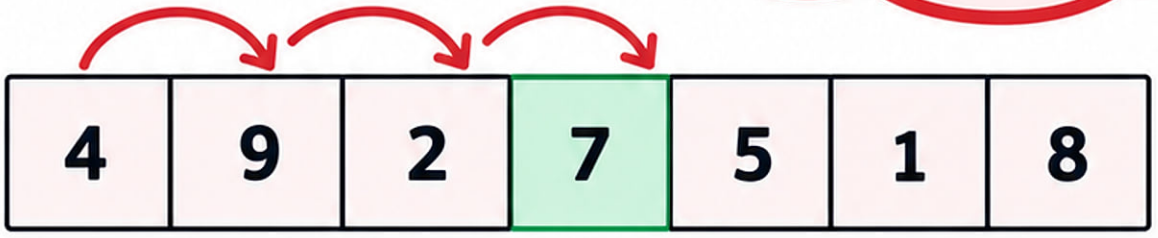
37 HashMap od środka

38 equals() i hashCode()

39 HashSet i szybkie sprawdzanie

Wyszukiwanie liniowe

Sprawdzamy element po elemencie




```java
int indexOf(int[] arr, int x) {
  for (int i = 0; i < arr.length; i++) {
    if (arr[i] == x) return i;
  }
  return -1; // brak elementu
}
```

Koszt

O(n) czasu i O(1) pamięci.

Zaleta

Działa na każdych danych, także nieposortowanych.

Wynik -1

Umowny sygnał, że elementu nie ma. Tak samo działa `list.indexOf(x)`.

Obiekty

Dla obiektów porównuj przez `equals()`, a nie przez `==`.

Kiedy wystarczy

Małe dane albo jednorazowe szukanie. Przy wielu zapytaniach lepiej posortować dane lub użyć HashSet.

W Javie

`list.contains(x)` i `list.indexOf(x)` to właśnie wyszukiwanie liniowe.

Wyszukiwanie binarne

Dzielimy posortowaną tablicę na pół

Szukamy 23

1. Środek to 16. 23 jest większe, więc odrzucamy lewą połowę.



2. Środek to 38. 23 jest mniejsze, więc odrzucamy prawą część.



3. Środek to 23. Znaleziono!



Warunek

Tablica musi być posortowana.

Koszt

O(log n). Milion elementów to najwyżej 20 porównań.

Pamięć

Wersja z pętlą: O(1).

```java
int lo = 0, hi = arr.length - 1;
while (lo <= hi) {
    int mid = lo + (hi - lo) / 2;
    if (arr[mid] == x) return mid;
    if (arr[mid] < x) lo = mid + 1;
    else hi = mid - 1;
}
return -1;
```

Gotowe metody

Arrays.binarySearch() i Collections.binarySearch() robią to samo.

Pułapki wyszukiwania binarnego

Sześć błędów, które psują prosty algorytm

Przepełnienie

(lo + hi) / 2 może przekroczyć zakres int. Pisz lo + (hi - lo) / 2.

Warunek pętli

Przy hi = arr.length - 1 używaj lo <= hi. Inaczej pominiesz ostatni element.

Pętla bez końca

Zawsze lo = mid + 1 lub hi = mid - 1, nigdy samo mid.

Brak sortowania

Na nieposortowanych danych wynik jest przypadkowy.

Ujemny wynik

Arrays.binarySearch zwraca -(miejsce wstawienia) - 1, gdy elementu nie ma.

Duplikaty

Przy powtórzeniach nie wiadomo, które wystąpienie zostanie znalezione.

Bezpieczny szablon

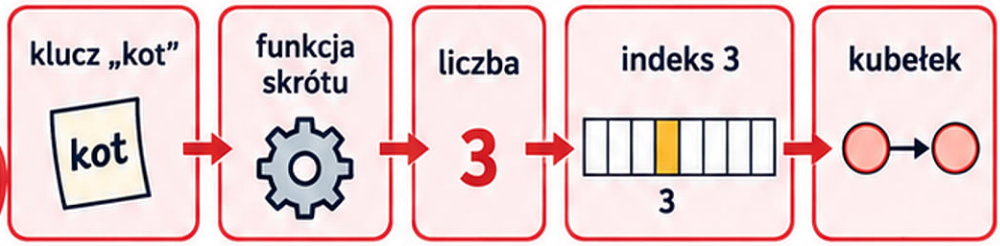
Pisz lo <= hi, licz mid = lo + (hi - lo) / 2 i zawsze przesuwaj granicę za mid.

```
int i = Arrays.binarySearch(arr, x);
if (i < 0) {
   int insertAt = -i - 1; // tu wstawić x
}
```

MAPA 36 | WYSZUKIWANIE

Haszowanie

Jak znaleźć element bez przeglądania wszystkiego



Funkcja skrótu

Zamienia klucz na liczbę. Ten sam klucz zawsze daje tę samą liczbę.

Indeks

Liczba jest zamieniana na numer kubełka w tablicy, np. przez resztę z dzielenia.

Kolizja

Dwa różne klucze trafiają do tego samego kubełka. To normalne i trzeba ją obsłużyć.

Łańcuchowanie

W kubełku trzymamy listę elementów. Tak działa HashMap.

Koszt

Średnio O(1) na wstawienie i wyszukiwanie. Przy wielu kolizjach koszt rośnie.

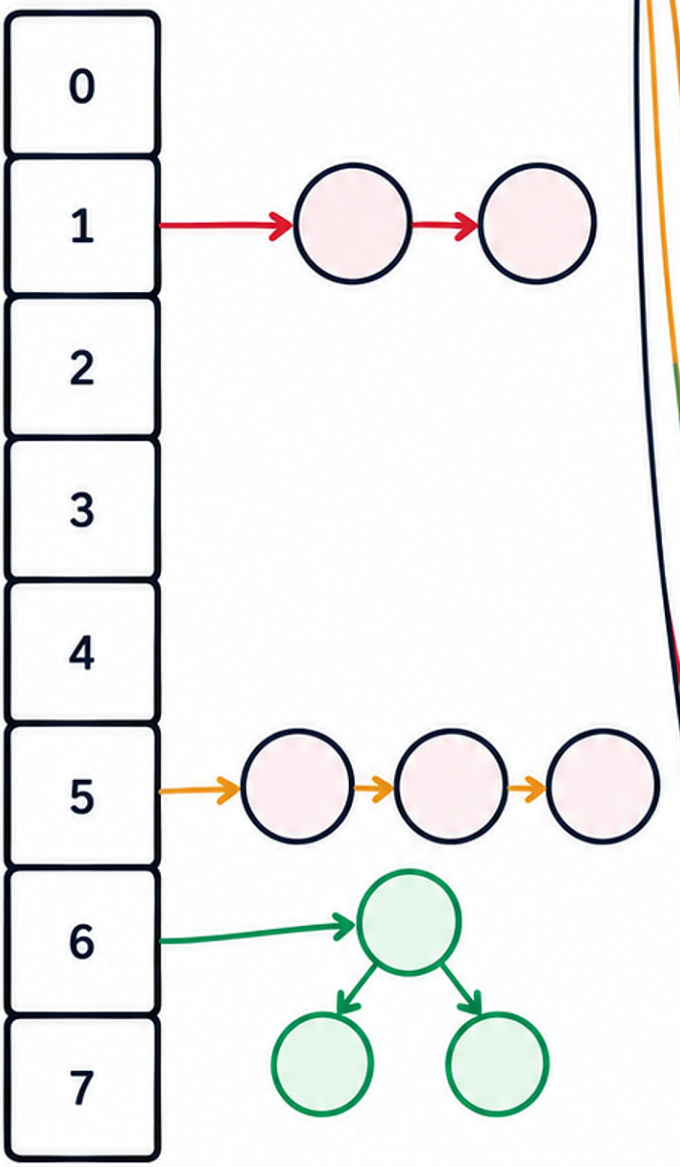
```
String key = "kot";
int h = key.hashCode();
int index = Math.floorMod(h, 16); // od 0 do 15
```

Uwaga

hashCode() może być ujemny. Dlatego używamy Math.floorMod zamiast zwykłego %.

HashMap od środka

Tablica kubełków z listami i drzewami



Tablica kubełków

Domyślnie 16 kubełków. Każdy przechowuje pary klucz–wartość.

Wybór kubełka

Indeks wynika z hashCode() klucza, wymieszanego i obciętego do rozmiaru tablicy.

Współczynnik 0,75

Gdy mapa jest zapełniona w 75%, tablica rośnie dwukrotnie, a elementy trafiają na nowe miejsca.

Drzewo w kubełku

Od Javy 8 lista w kubełku mająca co najmniej 8 elementów może zamienić się w drzewo: O(log n).

Koszt

put, get, remove: średnio O(1).

Bez kolejności

HashMap nie pamięta kolejności. LinkedHashMap pamięta kolejność dodawania.

```
Map<String, Integer> count = new HashMap<>();
for (String w : words) {
    count.merge(w, 1, Integer::sum);
}
count.getOrDefault("kot", 0);
```

equals() i hashCode()

Kontrakt, bez którego HashMap nie działa

equals()

- Porównuje zawartość obiektów.
- Musi być zwrotna i symetryczna.
- Domyślnie działa jak ==.

hashCode()

- Zwraca liczbę dla obiektu.
- Wybiera kubełek w HashMap i HashSet.
- Musi być stały, dopóki obiekt się nie zmienia.



Kontrakt

Jeśli a.equals(b), to a.hashCode() == b.hashCode().
Odwrotna zależność nie musi zachodzić.

```
class Point {
  final int x, y;
  Point(int x, int y) { this.x = x; this.y = y; }
  @Override public boolean equals(Object o) {
    return o instanceof Point p && x == p.x && y == p.y;
  }
  @Override public int hashCode() {
    return Objects.hash(x, y);
  }
}
```



Częsty błąd

Gdy nadpiszesz tylko equals(), set.contains() nie znajdzie równego obiektu.

HashSet i szybkie sprawdzanie

Czy ten element już był?

Operacja	ArrayList	HashSet
contains(x)	O(n)	O(1)
add(x)	O(1)*	O(1)
Kolejność	zachowana	brak

* koszt zamortyzowany, średnie wartości

W środku

HashSet to HashMap, w której liczą się tylko klucze.

Bez duplikatów

add(x) zwraca false, gdy element już jest w zbiorze.

Inne zbiory

LinkedHashSet pamięta kolejność dodawania. TreeSet trzyma elementy posortowane: O(log n).

Zastosowania

Usuwanie duplikatów, zbiór odwiedzonych wierzchołków, szybkie sprawdzanie słów.

```
Set<Integer> seen = new HashSet<>();
for (int x : arr) {
    if (!seen.add(x)) {
        System.out.println("Duplikat: " + x);
    }
}
```

Zapamiętaj

Elementy zbioru potrzebują poprawnych metod equals() i hashCode().

MODUŁ 7



Sortowanie

Od prostych pętli do O(n log n)

40 Sortowanie bąbelkowe

41 Sortowanie przez wybieranie

42 Sortowanie przez wstawianie

43 Sortowanie przez scalanie

44 Sortowanie szybkie

45 Sortowanie przez zliczanie

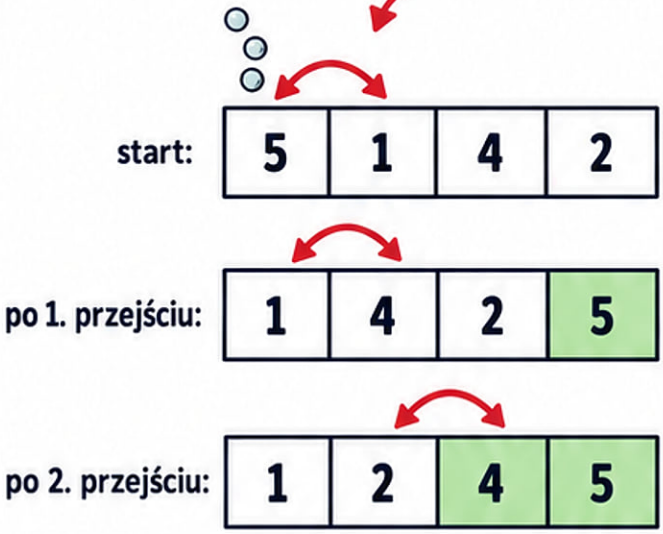
46 Porównanie algorytmów sortowania

47 Comparable i Comparator

48 Arrays.sort() i Collections.sort()

Sortowanie bąbelkowe

Największy element wypływa na koniec jak bąbelek




```java
for (int i = 0; i < n - 1; i++) {
   boolean swapped = false;
   for (int j = 0; j < n - 1 - i; j++) {
      if (arr[j] > arr[j + 1]) {
         int t = arr[j]; arr[j] = arr[j + 1];
         arr[j + 1] = t;
         swapped = true;
      }
   }
   if (!swapped) break;
}
```

Pomysł

Porównujemy sąsiadów i zamieniamy ich, gdy są w złej kolejności.

Koszt

$O(n^2)$. Z flagą swapped dla posortowanych danych wystarczy $O(n)$.

Pamięć

$O(1)$: sortuje w miejscu.

Stabilność

Tak. Równe elementy nie zmieniają kolejności.

Kiedy używać

Do nauki i bardzo małych danych. W projektach wybierz `Arrays.sort()`.

Sortowanie przez wybieranie

Wybierz najmniejszy i postaw go na początku

1. 29 | 10 | 14 | 37 | 13 — minimum 10



2. 10 | 29 | 14 | 37 | 13 — minimum 13



3. 10 | 13 | 14 | 37 | 29 — minimum 14



4. 10 | 13 | 14 | 37 | 29 — minimum 29, koniec: 10 | 13 | 14 | 29 | 37




```java
for (int i = 0; i < n - 1; i++) {
    int min = i;
    for (int j = i + 1; j < n; j++) {
        if (arr[j] < arr[min]) min = j;
    }
    int t = arr[i]; arr[i] = arr[min]; arr[min] = t;
}
```

Koszt

Zawsze O(n²) porównań, nawet dla posortowanych danych.

Mało zamian

Najwyżej n – 1 zamian. Przydaje się, gdy zapis jest drogi.

Stabilność

Nie. Zamiana może przestawić równe elementy.

Pamięć

O(1): sortuje w miejscu.



Sortowanie przez wstawianie

Jak układanie kart w dłoni

Pomysł

Lewa część jest już posortowana. Bierzemy kolejny element i wstawiamy go we właściwe miejsce.

Koszt

$O(n^2)$ w najgorszym przypadku, $O(n)$ dla danych prawie posortowanych.

Stabilność

Tak. Równych elementów nie przestawiamy.

```
for (int i = 1; i < n; i++) {
    int key = arr[i];
    int j = i - 1;
    while (j >= 0 && arr[j] > key) {
        arr[j + 1] = arr[j]; // przesuń w prawo
        j--;
    }
    arr[j + 1] = key;
}
```

Pamięć

O(1): sortuje w miejscu.

W Javie

`Arrays.sort()` używa go wewnętrznie dla bardzo małych fragmentów tablicy.

Przykład

Masz 2, 4, 7, 9 i nową kartę 5. Przesuwasz 9 i 7 w prawo, a 5 wstawiasz za 4.

Zapamiętaj

Przesuwamy elementy, a nie zamieniamy ich parami. To mniej zapisów.

Sortowanie przez scalanie

Dziel na pół, sortuj, scalaj

Dziel i zwyciężaj

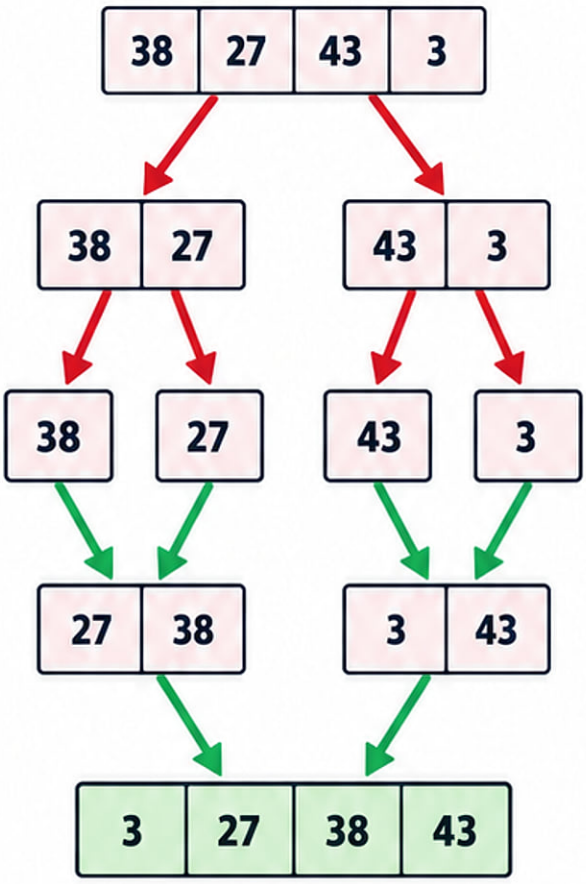
Dzielimy tablicę, aż zostaną jednoelementowe części.

Scalanie

Dwa wskaźniki łączą dwie posortowane połowy w czasie O(n).

Koszt

Zawsze O(n log n):
log n poziomów,
na każdym O(n) pracy.



Pamięć

O(n) na tablicę pomocniczą.

Stabilność

Tak, jeśli przy równych wartościach bierzemy element z lewej połowy.

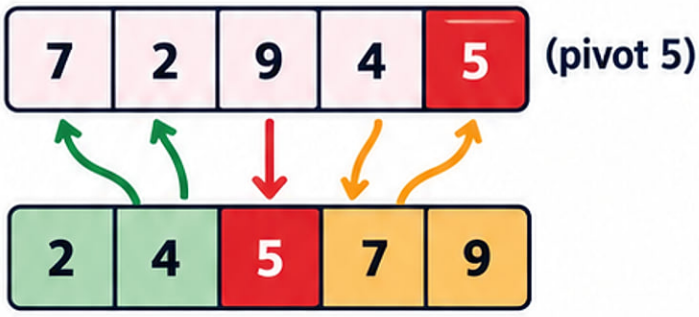
W Javie

Obiekty sortuje TimSort, ulepszona wersja sortowania przez scalanie.

```
void mergeSort(int[] a, int lo, int hi) { // hi poza zakresem
    if (hi - lo < 2) return;
    int mid = (lo + hi) >>> 1;
    mergeSort(a, lo, mid);
    mergeSort(a, mid, hi);
    merge(a, lo, mid, hi); // scala dwie posortowane połowy
}
```

Sortowanie szybkie

Pivot dzieli tablicę na mniejsze i większe



Pivot

Wybrany element, np. ostatni. Po partycji stoi na swoim miejscu.

Partycja

Mniejsze od pivota na lewo, pozostałe na prawo. Potem sortujemy obie części.

Koszt

Średnio O(n log n). Pesymistycznie $O(n^2)$, np. posortowane dane i ostatni element jako pivot.

Pamięć

Średnio O(log n) na stosie wywołań.

Stabilność

Nie.

W Javie

Arrays.sort( ) dla typów prostych używa wersji z dwoma pivotami (Dual-Pivot Quicksort).

```
void quickSort(int[] a, int lo, int hi) {
   if (lo >= hi) return;
   int p = partition(a, lo, hi);
   quickSort(a, lo, p - 1);
   quickSort(a, p + 1, hi);
}

int partition(int[] a, int lo, int hi) {
   int pivot = a[hi], i = lo;
   for (int j = lo; j < hi; j++) {
      if (a[j] < pivot) swap(a, i++, j);
   }
   swap(a, i, hi);
   return i;
}
```

Sortowanie przez zliczanie

Sortowanie bez ani jednego porównania

dane:

3	1	4	1	3	0

liczniki dla 0–4:

1	2	0	2	1
0	1	2	3	4

wynik:

0	1	1	3	3	4

Pomysł

Liczymy, ile razy występuje każda wartość, a potem wypisujemy je po kolei.

Koszt

O(n + k), gdzie k to największa wartość.

Pamięć

O(k) na tablicę liczników.

Warunek

Liczby całkowite z małego zakresu, np. oceny od 1 do 6 albo wiek.

Granica O(n log n)

Dotyczy tylko sortowania opartego na porównaniach. Zliczanie ją omija.

```
int[] count = new int[max + 1];
for (int x : arr) count[x]++;
int i = 0;
for (int v = 0; v <= max; v++) {
    while (count[v]-- > 0) arr[i++] = v;
}
```

Porównanie algorytmów sortowania

Czas, pamięć i stabilność w jednym miejscu

Algorytm	Czas średni	Pamięć	Stabilny
Bąbelkowe, wstawianie	O(n^2)	O(1)	✔ tak
Wybieranie	O(n^2)	O(1)	✘ nie
Scalanie	O(n log n)	O(n)	✔ tak
Szybkie	O(n log n)	O(log n)	✘ nie

Co to stabilność?

Równe elementy zostają w tej samej kolejności co przed sortowaniem.

Kiedy ma znaczenie

Gdy sortujesz kilka razy, np. najpierw po imieniu, a potem po klasie. Uczniowie z tej samej klasy zostają wtedy ułożeni alfabetycznie.

Małe dane

Sortowanie przez wstawianie jest często najszybsze dla kilkunastu elementów.

Wąski zakres

Sortowanie przez zliczanie: O(n + k) bez porównań.

Na co dzień

W prawdziwym projekcie użyj Arrays.sort() lub List.sort(). Własne sortowanie pisz po to, aby zrozumieć mechanizm.

MAPA 47 | SORTOWANIE



Comparable i Comparator

Jak powiedzieć Javie, który obiekt jest „mniejszy"

Comparable

- Porządek naturalny, wpisany w klasę.
- Jedna metoda: compareTo(other).
- Wynik ujemny, zero lub dodatni.

```
record Student(String name, int grade)
        implements Comparable<Student> {
  public int compareTo(Student o) {
    return name.compareTo(o.name);
  }
}
```

Comparator

- Porządek z zewnątrz, może być ich wiele.
- Łatwo łączyć kryteria.
- Nie trzeba zmieniać klasy.

```
students.sort(
  Comparator.comparingInt(Student::grade)
            .reversed()
            .thenComparing(Student::name));
```

Jak wybrać

Jeden oczywisty porządek: Comparable. Różne sposoby sortowania: Comparator.

Unikaj

Nie pisz a - b w compare(). Przy dużych liczbach wynik może się przepełnić. Użyj Integer.compare(a, b).

Krócej

Comparator.comparing(Student::name) tworzy komparator w jednej linii, bez pisania osobnej klasy.

MAPA 48 | SORTOWANIE

Typy proste

Arrays.sort(int[]) używa Dual-Pivot Quicksort: szybko, ale bez stabilności.

Obiekty

Arrays.sort(Object[]) i List.sort() używają TimSorta: O(n log n) i stabilnie.

Collections.sort()

Wywołuje list.sort(), więc działa tak samo jak sortowanie obiektów.

Arrays.sort() i Collections.sort()

Co dzieje się w środku gotowych metod

Fragment tablicy

Arrays.sort(arr, from, to) sortuje tylko zakres od from do to, bez elementu to.

Wiele rdzeni

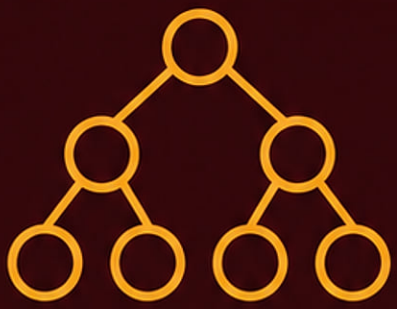
Arrays.parallelSort() dzieli pracę na wątki przy dużych tablicach.

```
int[] nums = {5, 2, 9};
Arrays.sort(nums);                    // [2, 5, 9]
List<String> names = new ArrayList<>(List.of("Ola", "Adam", "Ewa"));
Collections.sort(names);              // [Adam, Ewa, Ola]
names.sort(Comparator.reverseOrder()); // [Ola, Ewa, Adam]
```

Uwaga

List.of() tworzy listę niezmienną. Jej sortowanie rzuca UnsupportedOperationException.

MODUŁ 8



Drzewa

Struktury hierarchiczne i szybkie wyszukiwanie

49 Drzewo: korzeń, węzły, liście

50 Drzewo binarne

51 Przechodzenie w głąb

52 Przechodzenie wszerz

53 BST: wstawianie i wyszukiwanie

54 Usuwanie z BST

55 Drzewa zrównoważone, TreeMap i TreeSet

56 Kopiec binarny

MAPA 49 | DRZEWA

Drzewo: korzeń, węzły, liście

Słownik, który trzeba znać

Korzeń

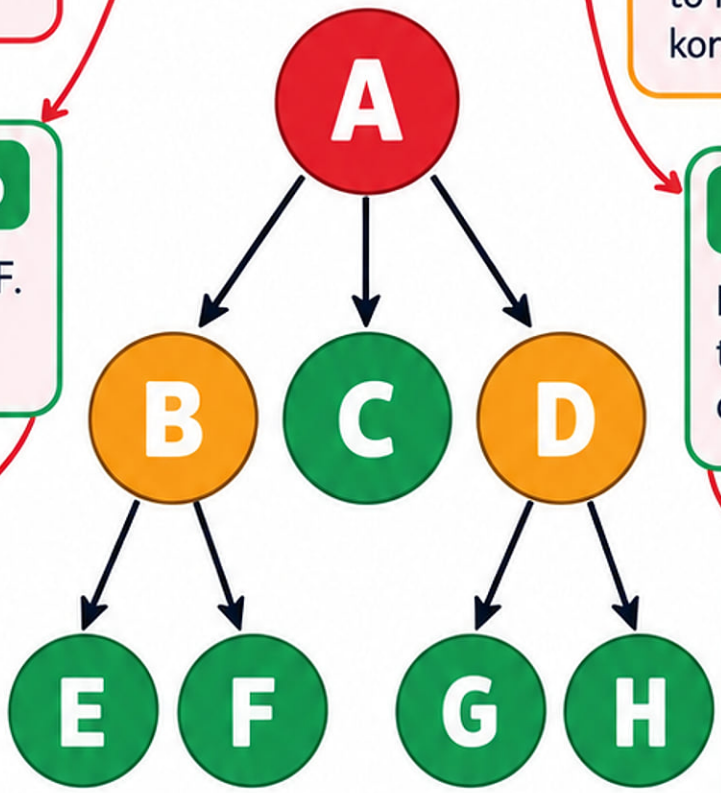
Jedyny węzeł bez rodzica. Tutaj: A.

Rodzic i dziecko

B jest rodzicem E i F.
E i F są dziećmi B.

Liść

Węzeł bez dzieci:
C, E, F, G i H.



Głębokość i wysokość

Głębokość to odległość od korzenia. Wysokość drzewa to najdłuższa droga od korzenia do liścia. Tutaj: 2.

Poddrzewo

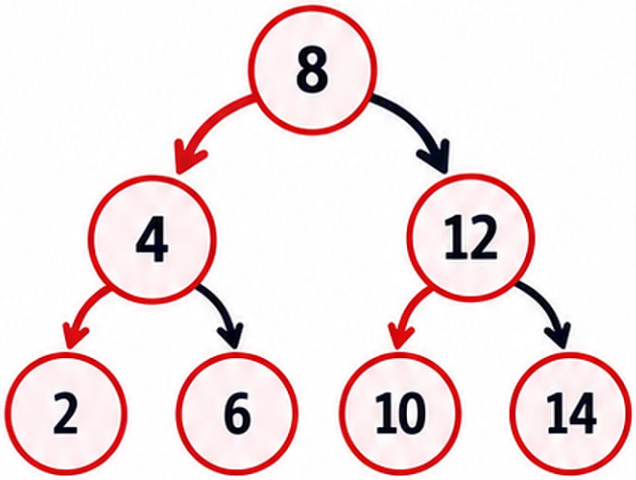
Każdy węzeł z potomkami tworzy własne małe drzewo.

Gdzie je spotkasz

Foldery na dysku, drzewo DOM w HTML, struktura firmy, drzewa decyzyjne.

```
class TreeNode {
    String value;
    List<TreeNode> children = new ArrayList<>();
}
```

MAPA 50 | DRZEWA



Drzewo binarne

Każdy węzeł ma najwyżej dwoje dzieci

Lewe i prawe

Dzieci mają swoje miejsca: left i right. Brak dziecka to null.

```
class Node {
  int value;
  Node left, right;
  Node(int value) {
    this.value = value;
  }
}

int height(Node n) {
  if (n == null) return -1;
  return 1 + Math.max(
     height(n.left),
     height(n.right));
}
```

Pełne

Każdy węzeł ma zero albo dwoje dzieci.

Kompletne

Wszystkie poziomy pełne, ostatni zapełniany od lewej. Tak wygląda kopiec.

Wysokość

Przy n węzłach wynosi od około $\log_2 n$ do $n - 1$. Od niej zależy szybkość operacji.

Rekurencja

Większość metod to: zrób coś z węzłem, potem wywołaj się dla left i right.

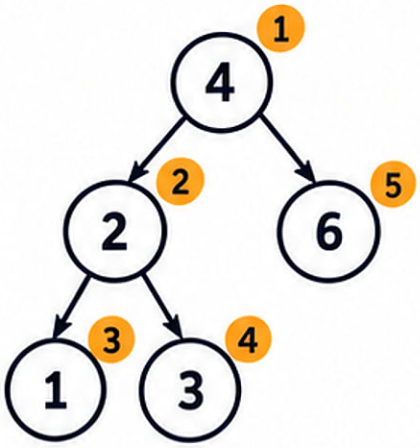
MAPA 51 | DRZEWA

Przechodzenie w głąb

Pre-order, in-order i post-order

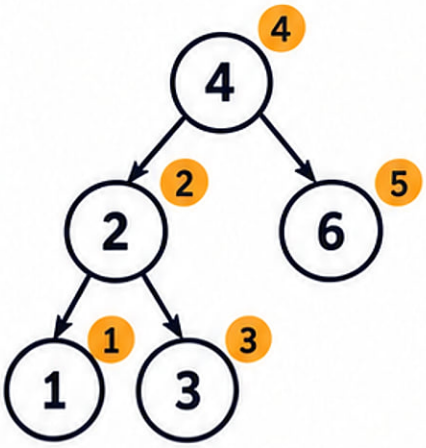
4 → 2, 6 | 2 → 1, 3

Pre-order



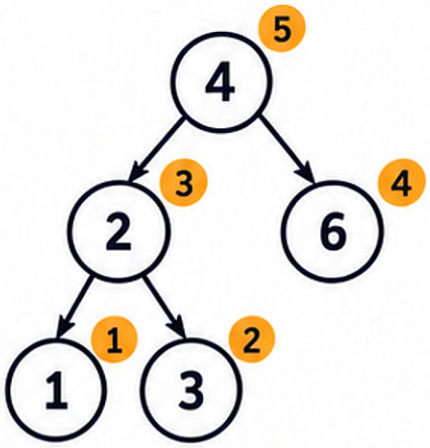
węzeł, lewe, prawe:
4, 2, 1, 3, 6

In-order



lewe, węzeł, prawe:
1, 2, 3, 4, 6

Post-order



lewe, prawe, węzeł:
1, 3, 2, 6, 4

```
void inOrder(Node n) {
    if (n == null) return;
    inOrder(n.left);
    System.out.print(n.value + " ");
    inOrder(n.right);
}
```

Koszt

Każdy węzeł raz: O(n).
Pamięć O(h), gdzie h to wysokość.

Zastosowania

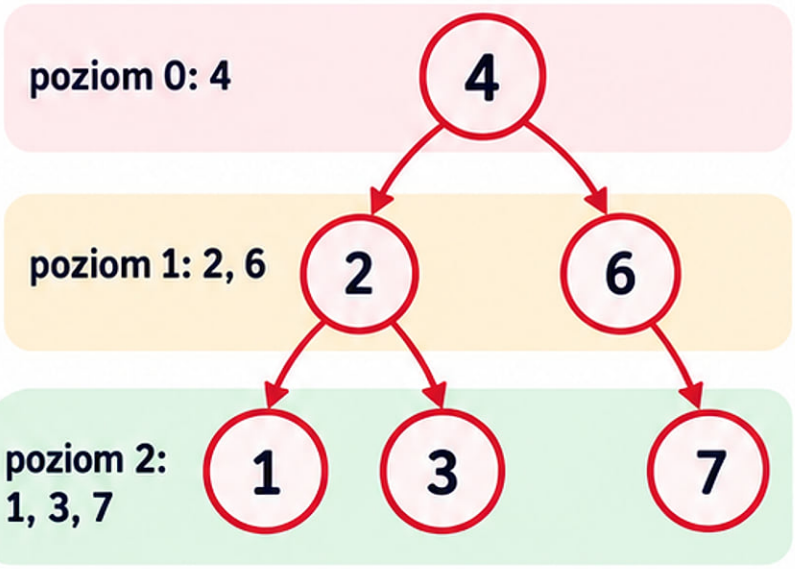
Pre-order: kopiowanie drzewa.
In-order: BST w kolejności rosnącej.
Post-order: usuwanie i liczenie rozmiaru folderów.

Sztuczka

Nazwa mówi, kiedy odwiedzamy węzeł: przed dziećmi, między nimi albo po nich.

Przechodzenie wszerz

Poziom po poziomie, z pomocą kolejki



Kolejność
4, 2, 6, 1, 3, 7




```
Queue<Node> q = new ArrayDeque<>();
q.offer(root);
while (!q.isEmpty()) {
  Node n = q.poll();
  System.out.print(n.value + " ");
  if (n.left != null) q.offer(n.left);
  if (n.right != null) q.offer(n.right);
}
```

Kolejka, nie stos

Kolejka FIFO pilnuje, aby płytsze węzły wyszły przed głębszymi.

Koszt

O(n) czasu. Pamięć zależy od najszerszego poziomu.

Poziomy osobno

Zapisz q.size() przed pętlą wewnętrzną, aby przetworzyć jeden cały poziom.

Zastosowania

Najmniejsza głębokość, widok drzewa z boku, wypisywanie poziomami.

BST: wstawianie i wyszukiwanie

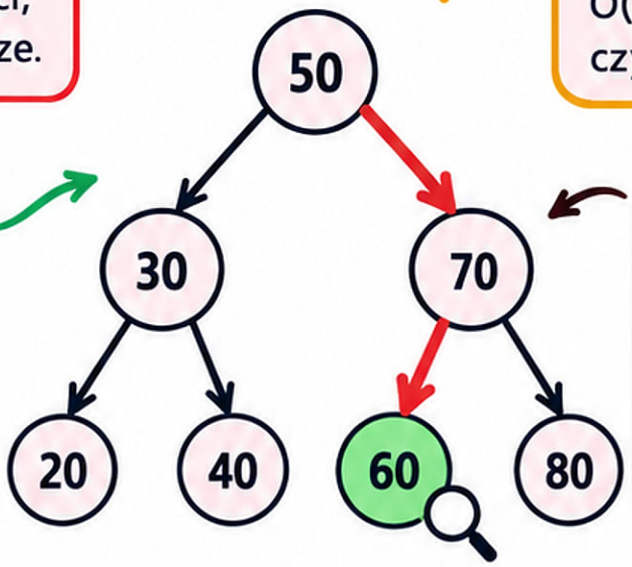
Mniejsze na lewo, większe na prawo

Zasada BST

W lewym poddrzewie są tylko mniejsze wartości, w prawym tylko większe.

Koszt O(h)

Zrównoważone drzewo: O(log n). Zdegenerowane, czyli „lista": O(n).



Bonus in-order

Przejście in-order zwraca wartości posortowane rosnąco.

Minimum i maksimum

Minimum to węzeł najbardziej na lewo, maksimum najbardziej na prawo.

50 → 30, 70 | 30 → 20, 40 | 70 → 60, 80 — szukamy 60: 50 → 70 → 60

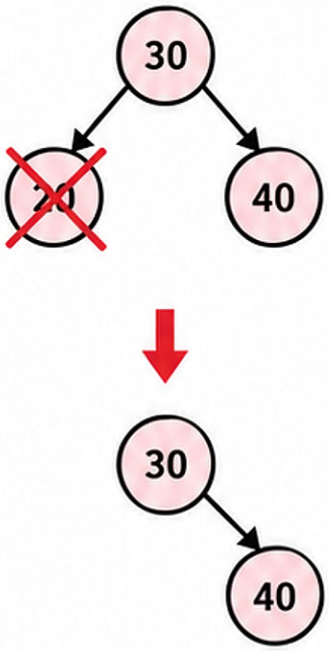
```
Node insert(Node n, int x) {
    if (n == null) return new Node(x);
    if (x < n.value) n.left = insert(n.left, x);
    else if (x > n.value) n.right = insert(n.right, x);
    return n; // duplikat pomijamy
}

boolean contains(Node n, int x) {
    while (n != null) {
        if (x == n.value) return true;
        n = x < n.value ? n.left : n.right;
    }
    return false;
}
```

Usuwanie z BST

Trzy przypadki, które trzeba rozróżnić

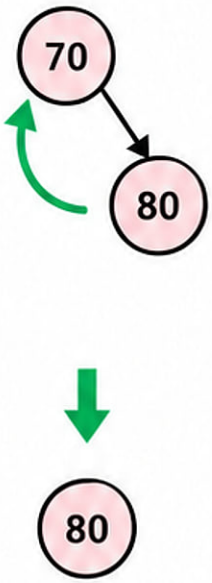
Liść

Po prostu usuwamy węzeł, np. 20.



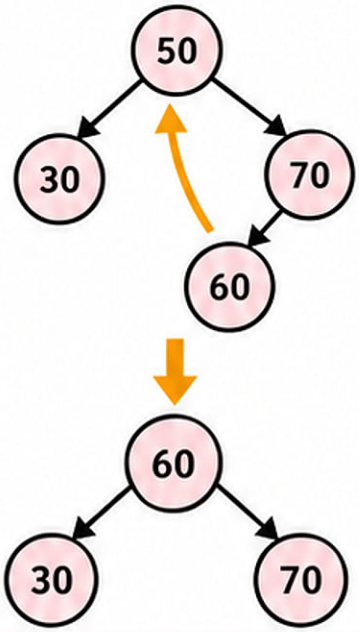
Jedno dziecko

Dziecko zajmuje miejsce usuwanego węzła.



Dwoje dzieci

Wstawiamy następnika, czyli najmniejszą wartość z prawego poddrzewa, a potem usuwamy jego stary węzeł. Usuwając 50, wstawiamy 60.



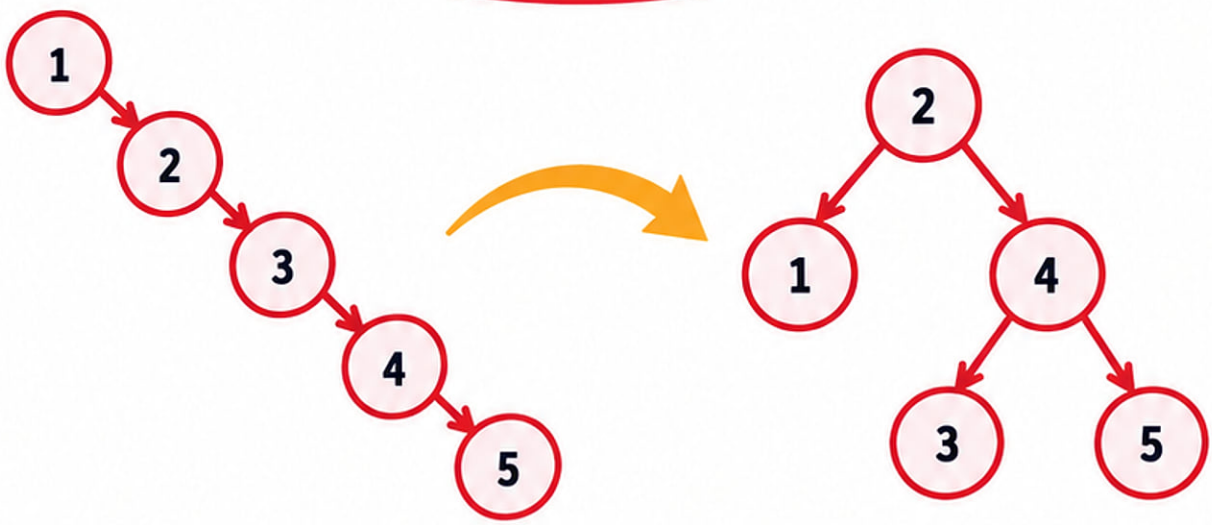

```java
Node delete(Node n, int x) {
  if (n == null) return null;
  if (x < n.value) n.left = delete(n.left, x);
  else if (x > n.value) n.right = delete(n.right, x);
  else {
    if (n.left == null) return n.right;
    if (n.right == null) return n.left;
    Node s = n.right;
    while (s.left != null) s = s.left;
    n.value = s.value;
    n.right = delete(n.right, s.value);
  }
  return n;
}
```

Koszt

O(h), tak jak wyszukiwanie. Zamiast następnika można użyć poprzednika: największej wartości z lewego poddrzewa.

Drzewa zrównoważone, TreeMap i TreeSet

Jak utrzymać O(log n) bez względu na dane



1 → 2 → 3 → 4 → 5 zamienia się w 2 → 1, 4 | 4 → 3, 5

Problem

Wstawianie posortowanych danych zamienia BST w listę: O(n).

Rotacje

Drzewa AVL i czerwono-czarne wykonują rotacje po każdej zmianie, aby wysokość pozostała bliska log n.

TreeMap i TreeSet

W Javie to drzewa czerwono-czarne. put, get i remove: O(log n). Klucze są zawsze posortowane.

Metody nawigacji

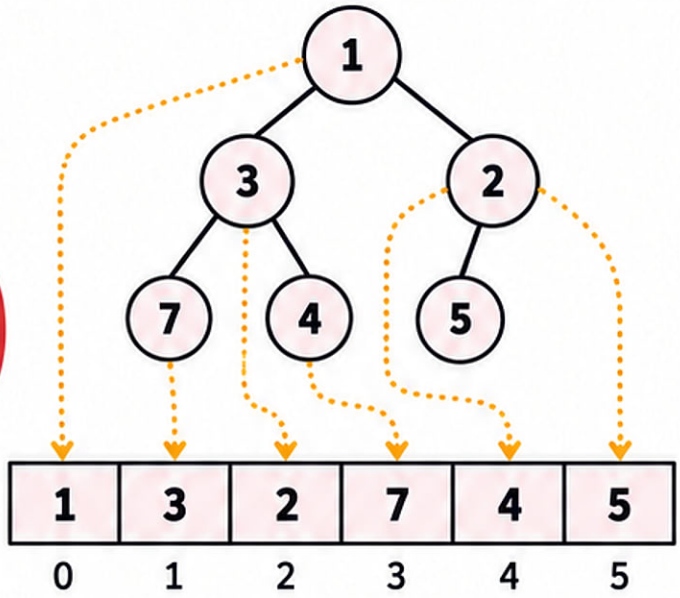
firstKey, floorKey, ceilingKey, headMap, subMap: najbliższy mniejszy, większy albo cały zakres.

```
TreeMap<Integer, String> scale = new TreeMap<>();
scale.put(0, "niedostateczny");
scale.put(50, "dopuszczający");
scale.put(70, "dobry");
scale.put(90, "celujący");
scale.floorEntry(76).getValue(); // "dobry"
```

MAPA 56 | DRZEWA

Kopiec binarny

Drzewo kompletne zapisane w tablicy



Własność kopca

W kopcu typu min rodzic jest zawsze mniejszy lub równy dzieciom. Minimum jest w korzeniu.

Indeksy

Dla węzła i: rodzic (i - 1) / 2, dzieci 2i + 1 i 2i + 2. Bez referencji.

Wstawianie

Dodaj na koniec i przesuwaj w górę, dopóki rodzic jest większy: O(log n).

Usuwanie minimum

Ostatni element trafia do korzenia i przesuwa się w dół: O(log n).

Więcej

Budowa kopca z tablicy: O(n). Sortowanie przez kopcowanie: O(n log n).

```
void siftUp(int[] h, int i) {
    while (i > 0 && h[(i - 1) / 2] > h[i]) {
        int p = (i - 1) / 2;
        int t = h[p]; h[p] = h[i]; h[i] = t;
        i = p;
    }
}
```

MODUŁ 9



Grafy

Sieci połączeń: mapy, znajomi, zależności

57 Graf: wierzchołki i krawędzie

58 Reprezentacja grafu

59 BFS, czyli przeszukiwanie wszerz

60 DFS, czyli przeszukiwanie w głąb

61 Spójne składowe i wykrywanie cykli

62 Algorytm Dijkstry

63 Sortowanie topologiczne

64 Minimalne drzewo rozpinające

MAPA 57 | GRAFY

Graf: wierzchołki i krawędzie

Najogólniejsza struktura połączeń

Wierzchołek i krawędź

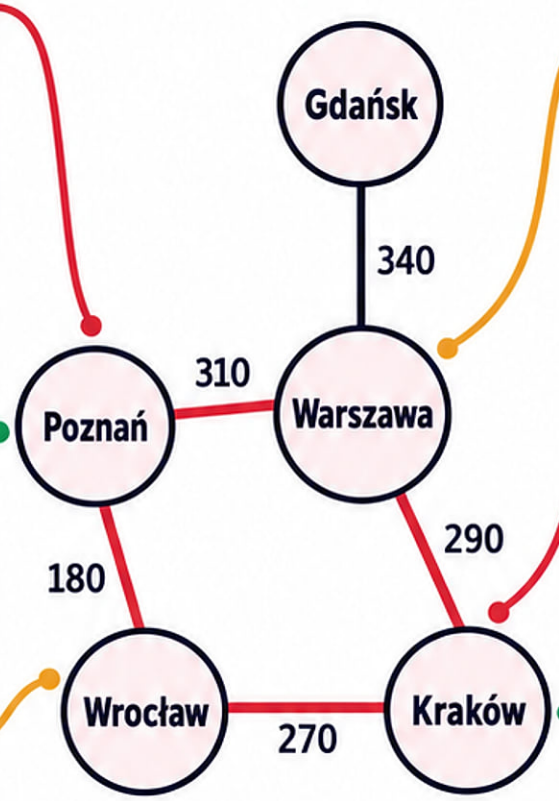
Wierzchołki to obiekty, np. miasta. Krawędzie to połączenia między nimi.

Skierowany czy nie

Znajomość działa w obie strony. Obserwowanie kogoś w sieci działa w jedną.

Ważony

Krawędź ma wagę, np. odległość w kilometrach.



Stopień

Liczba krawędzi wierzchołka. Warszawa ma stopień 3.

Ścieżka i cykl

Warszawa, Poznań, Wrocław, Kraków i z powrotem do Warszawy to cykl.

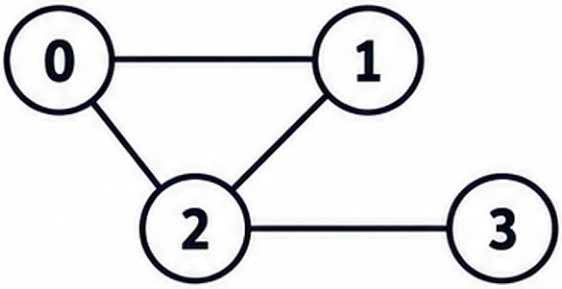
Oznaczenia

V to liczba wierzchołków, E to liczba krawędzi. Koszt algorytmów zapisujemy jako O(V + E).

```
record Edge(String from, String to, int weight) {}
List<Edge> edges = List.of(new Edge("Warszawa", "Kraków", 290));
```

Reprezentacja grafu

Macierz czy lista sąsiedztwa?



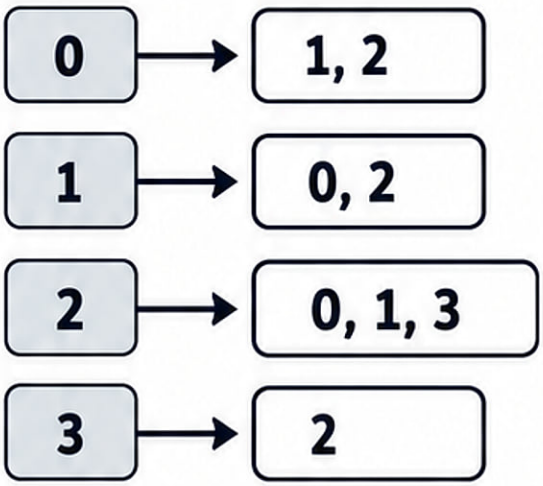
krawędzie: 0–1, 0–2, 1–2, 2–3

Macierz sąsiedztwa

	0	1	2	3
0	0	1	1	0
1	1	0	1	0
2	1	1	0	1
3	0	0	1	0

Pamięć $O(V^2)$. Sprawdzenie krawędzi $O(1)$. Dobra dla małych i gęstych grafów.

Lista sąsiedztwa



Pamięć $O(V + E)$. Szybkie przeglądanie sąsiadów. Najczęstszy wybór.

```
List<List<Integer>> adj = new ArrayList<>();
for (int i = 0; i < n; i++) adj.add(new ArrayList<>());
adj.get(0).add(1);
adj.get(1).add(0); // graf nieskierowany: w obie strony
```

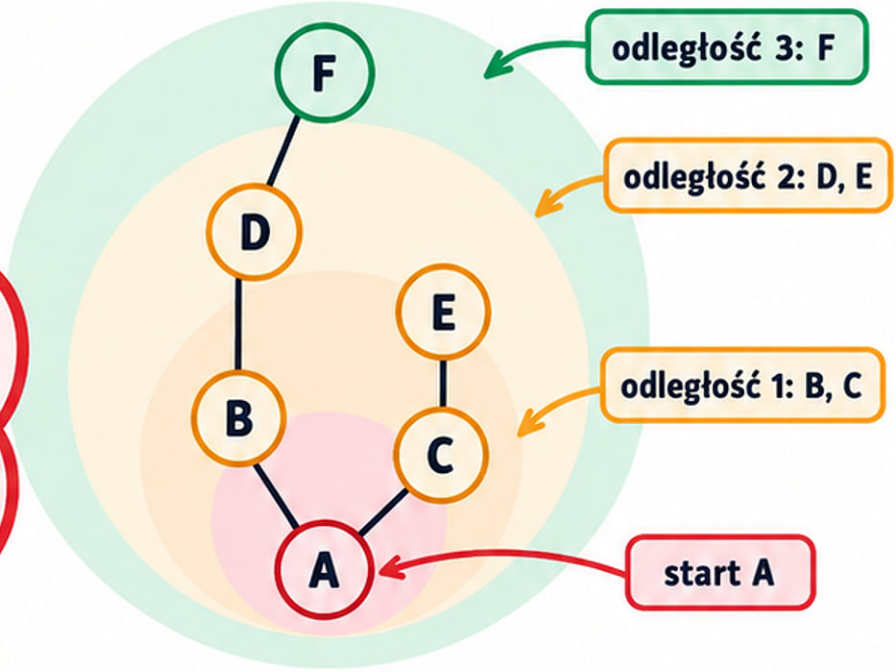
Zasada

Większość rzeczywistych grafów jest rzadka, więc wybieraj listę sąsiedztwa.

MAPA 59 | GRAFY

BFS, czyli przeszukiwanie wszerz

Fala, która rozchodzi się warstwami



A	B	C	D	E	F

```
int[] dist = new int[n];
Arrays.fill(dist, -1);
Queue<Integer> q = new ArrayDeque<>();
dist[start] = 0;
q.offer(start);
while (!q.isEmpty()) {
   int v = q.poll();
   for (int u : adj.get(v)) {
      if (dist[u] == -1) {
         dist[u] = dist[v] + 1;
         q.offer(u);
      }
   }
}
```

Kolejka

Najpierw wszyscy sąsiedzi, potem sąsiedzi sąsiadów.

Oznaczaj od razu

Oznacz wierzchołek przy dodawaniu do kolejki, a nie przy wyjmowaniu. Unikniesz duplikatów.

Najkrótsza droga

W grafie bez wag BFS daje najmniejszą liczbę krawędzi. Koszt: O(V + E).

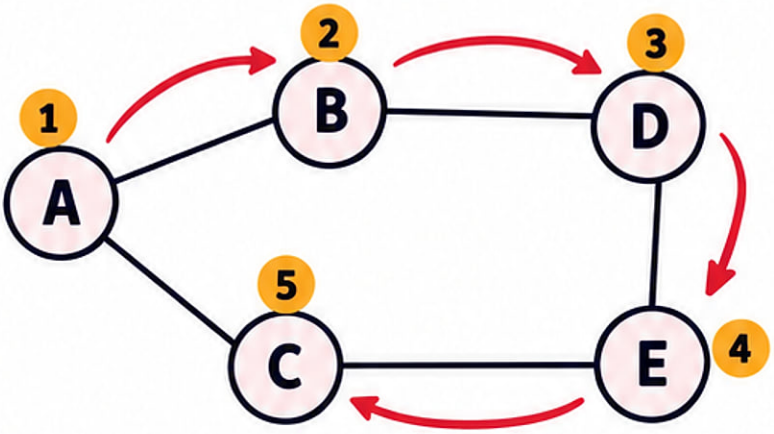
Zastosowania

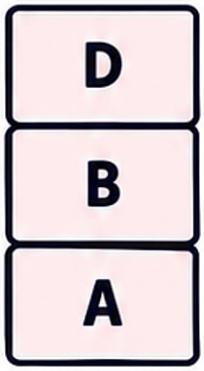
Wyjście z labiryntu, znajomi znajomych, najmniej przesiadek.

MAPA 60 | GRAFY

DFS, czyli przeszukiwanie w głąb

Idź jak najdalej, potem się cofnij





krawędzie: A–B, A–C, B–D, D–E, C–E — **kolejność odwiedzin:** A, B, D, E, C

```java
void dfs(int v, boolean[] visited) {
    visited[v] = true;
    for (int u : adj.get(v)) {
        if (!visited[u]) dfs(u, visited);
    }
}
```

Stos

Rekurencja korzysta ze stosu wywołań. Można też użyć własnego stosu Deque.

Koszt

O(V + E) czasu i O(V) pamięci na odwiedzone wierzchołki.

Głębokie grafy

Przy bardzo długich ścieżkach rekurencja grozi błędem StackOverflowError. Wtedy wybierz wersję z pętlą.

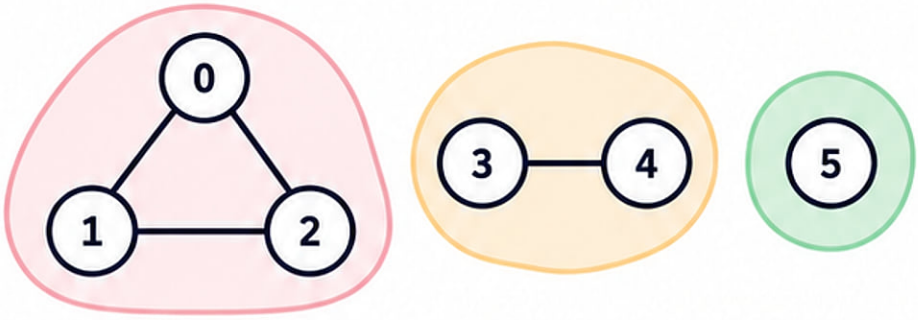
Zastosowania

Labirynty, wykrywanie cykli, spójne składowe, sortowanie topologiczne.

BFS czy DFS? | Najkrótsza droga bez wag: BFS. Pełne zbadanie struktury i zależności: DFS.

Spójne składowe i wykrywanie cykli

Dwa zadania rozwiązywane jednym DFS



{0, 1, 2} | {3, 4} | {5}: 3 składowe

Spójne składowe

Uruchamiamy DFS z każdego nieodwiedzonego wierzchołka. Każde uruchomienie to nowa składowa.

```
int components = 0;
boolean[] visited = new boolean[n];
for (int v = 0; v < n; v++) {
    if (!visited[v]) {
        dfs(v, visited);
        components++;
    }
}
```

Cykle

Graf nieskierowany: cykl jest wtedy, gdy DFS trafi na odwiedzonego sąsiada, który nie jest rodzicem.

Graf skierowany: używamy trzech kolorów. Biały to nieodwiedzony, szary jest na bieżącej ścieżce, czarny jest przetworzony. Krawędź do szarego oznacza cykl.



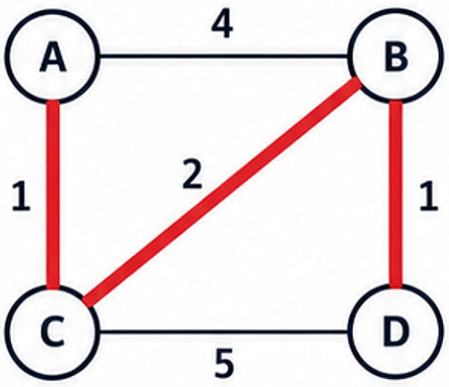
Alternatywa

Struktura Union-Find łączy zbiory i szybko sprawdza, czy dwa wierzchołki są w tej samej składowej.

MAPA 62 | GRAFY

Algorytm Dijkstry

Najkrótsze drogi w grafie z wagami



A–B 4 | A–C 1 | C–B 2 | B–D 1 | C–D 5 — start: A

Wynik dla startu w A

Wierzchołek	Odległość	Poprzednik
C	1	A
B	3	C
D	4	B

```
Arrays.fill(dist, Integer.MAX_VALUE);
dist[src] = 0;
PriorityQueue<int[]> pq =
    new PriorityQueue<>(Comparator.comparingInt(a -> a[1]));
pq.offer(new int[]{src, 0});

while (!pq.isEmpty()) {
    int[] cur = pq.poll();
    int v = cur[0];
    if (cur[1] > dist[v]) continue; // stary wpis
    for (int[] e : adj.get(v)) {    // e = {sąsiad, waga}
        int nd = dist[v] + e[1];
        if (nd < dist[e[0]]) {
            dist[e[0]] = nd;
            pq.offer(new int[]{e[0], nd});
        }
    }
}
```

ZACHŁANNOŚĆ

Zawsze bierzemy najbliższy nieodwiedzony wierzchołek.

KOSZT

O((V + E) log V) z kolejką priorytetową.

TYLKO WAGI NIEUJEMNE

Przy ujemnych wagach użyj algorytmu Bellmana-Forda.

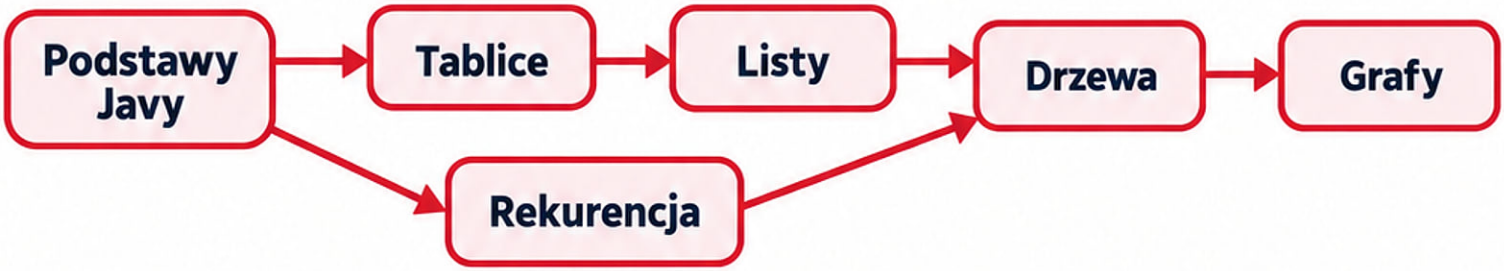
CAŁA TRASA

Zapamiętuj poprzednika każdego wierzchołka i idź od celu wstecz.

MAPA 63 | GRAFY

Sortowanie topologiczne

W jakiej kolejności wykonać zadania z zależnościami?



Jedna poprawna kolejność

1. Podstawy Javy
2. Tablice
3. Rekurencja
4. Listy
5. Drzewa
6. Grafy

TYLKO DAG

Działa w grafie skierowanym bez cykli.

STOPIEŃ WEJŚCIOWY

Liczba strzałek wchodzących. Zaczynamy od zadań, które mają 0.

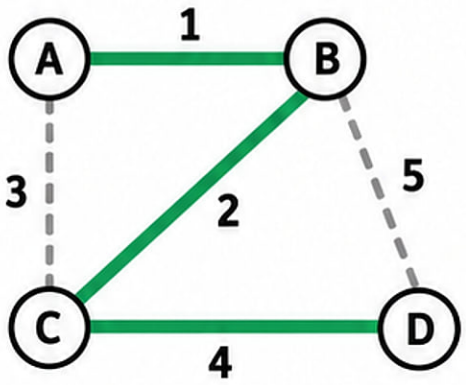
KOSZT

O(V + E). Kolejność nie musi być jednoznaczna.

ZASTOSOWANIA

Kompilacja modułów, zależności w Mavenie i Gradle'u, plan nauki.

```
int[] indeg = new int[n];
for (int v = 0; v < n; v++)
    for (int u : adj.get(v)) indeg[u]++;

Queue<Integer> q = new ArrayDeque<>();
for (int v = 0; v < n; v++) if (indeg[v] == 0)
    q.offer(v);

List<Integer> order = new ArrayList<>();
while (!q.isEmpty()) {
    int v = q.poll();
    order.add(v);
    for (int u : adj.get(v))
        if (--indeg[u] == 0) q.offer(u);
}

// order.size() < n oznacza cykl
```



Minimalne drzewo rozpinające

Połącz wszystko najmniejszym kosztem

A–B 1 | B–C 2 | A–C 3 | C–D 4 | B–D 5 — wybrane: A–B, B–C, C–D, suma 7

Definicja

Drzewo łączy wszystkie V wierzchołków za pomocą $V - 1$ krawędzi, bez cykli i z najmniejszą sumą wag.

Kruskal

Bierzemy krawędzie od najtańszej. Pomijamy te, które tworzą cykl. O(E log E).

Prim

Drzewo rośnie od jednego wierzchołka. Najtańszą krawędź wybiera PriorityQueue.

```
int find(int x) { // Union-Find
    while (parent[x] != x) {
        parent[x] = parent[parent[x]];
        x = parent[x];
    }
    return x;
}

edges.sort(Comparator.comparingInt(Edge::weight));
for (Edge e : edges) {
    int a = find(e.from()), b = find(e.to());
    if (a != b) { parent[a] = b; mst.add(e); }
}
```

Zastosowania

Najtańsza sieć kabli, rur lub dróg łącząca wszystkie punkty.

ŚCIĄGAWKA 1

Koszt operacji w strukturach danych

Średnie wartości dla n elementów

Sekwencje

Struktura	Odczyt i szukanie	Dodanie	Usunięcie
ArrayList	get **O(1)**, szukanie **O(n)**	koniec **O(1)*** środek **O(n)**	koniec **O(1)** środek **O(n)**
LinkedList	**O(n)**	końce **O(1)**	końce **O(1)**
ArrayDeque	końce **O(1)**	końce **O(1)***	końce **O(1)**

Szybkie wyszukiwanie

Struktura	Szukanie	Dodanie	Usunięcie
HashMap, HashSet	**O(1)**	**O(1)**	**O(1)**
TreeMap, TreeSet	**O(log n)**	**O(log n)**	**O(log n)**
PriorityQueue	minimum **O(1)**	**O(log n)**	minimum **O(log n)**

zielony: bardzo szybko bursztynowy: szybko różowy: zależy od n

* koszt **zamortyzowany**. Zwykła tablica: odczyt O(1), wstawienie w środek O(n). Przeszukanie grafu BFS lub DFS: O(V + E).

ŚCIĄGAWKA 2

Która struktura do jakiego problemu

Zacznij od pytania, a nie od kodu



Szybki odczyt po indeksie → **ArrayList**



Czy ten element już był? → **HashSet**



Słownik, liczenie wystąpień → **HashMap**



Dane zawsze posortowane, najbliższy klucz → **TreeMap lub TreeSet**



Cofanie, nawiasy, ostatni element → **stos: ArrayDeque**



Obsługa po kolei → **kolejka: ArrayDeque**



Zawsze najmniejszy lub największy, k najlepszych → **PriorityQueue**



Połączenia, trasy, zależności → **graf: lista sąsiedztwa**

Algorytmy na skróty

Szukanie w posortowanych danych: wyszukiwanie binarne. Najkrótsza droga bez wag: BFS. Najkrótsza droga z wagami: Dijkstra. Kolejność zadań: sortowanie topologiczne. Pary i kolejne elementy tablicy: dwa wskaźniki lub okno przesuwne.

Nie wiesz, co wybrać? Zacznij od ArrayList i HashMap.
Zmieniaj strukturę dopiero wtedy, gdy pomiar pokaże problem.

CO DALEJ? PLAN POWTÓREK

Masz już 64 mapy. Teraz zamień je w umiejętność. Ten czterotygodniowy plan pomoże Ci powtórzyć całość bez pośpiechu.

1

Tydzień 1

Moduły 1–2. Policz złożoność trzech własnych pętli.

2

Tydzień 2

Moduły 3–5. Napisz własny stos i kolejkę cykliczną.

3

Tydzień 3

Moduły 6–7. Zaimplementuj wyszukiwanie binarne i trzy sortowania.

4

Tydzień 4

Moduły 8–9. Zbuduj BST i uruchom BFS na liście sąsiedztwa.



Metoda powtórek

Zasłoń mapę i narysuj ją z pamięci na kartce. Porównaj z oryginałem. Wróć do niej po 1, 3 i 7 dniach.



Struktury danych to narzędzia. Im lepiej je znasz, tym prostszy staje się Twój kod. Powodzenia!