



Andrzej Pająk
A.Pajak@ii.pw.edu.pl
Andrzej.Pajak@pw.edu.pl

Struktury danych (cz.2)

Instytut Informatyki
Studia Podyplomowe
Java EE — produkcja oprogramowania

Grupy JA20Z - (JA2-A, B)
Warszawa 2021



- Notacja asymptotyczna
- Przykład wyznaczania asymptotyki
- Myślenie rekurencyjne
- Struktury liniowe: tablice
- Struktury liniowe: porządkowanie
- Wyszukiwanie binarne
- Wyszukiwanie binarne – bez rekursji
- Algorytmy sortowania
- Kopiec binarny
- Algorytmy selekcji



- **Precyzyjne określenie złożoności algorytmu jest trudne i zazwyczaj niemożliwe:**
 - algorytm może być **kompozycją algorytmów składowych** o zmieniającym się wpływie na złożoność całości
 - w praktyce, czas wykonania konkretnego algorytmu (programu) zależy od **czynników nie uwzględnianych podczas analizy** (np. losowe obciążenie systemu innymi zadaniami, sprawność platformy, itd.)
 - w modelach obliczeń przyjmuje się **idealizację operacji** uważanych za podstawowe (rzeczywiste operacje w różnym stopniu odbiegają od ideału)
- **Ergo: ocena złożoności poprzez asymptotyki dla „dostatecznie” dużych rozmiarów problemu**



Stosowane są 3 asymptotyki (n - rozmiar problemu):

$O(f(n))$ oszacowanie tempa wzrostu **od góry**; asymptotykę odczytujemy tak: **istnieje algorytm** rozwiązania problemu o złożoności **nie gorszej niż** $f(n)$

$\Omega(f(n))$ oszacowanie tempa wzrostu **od dołu**; asymptotykę odczytujemy tak: **nie istnieje algorytm** rozwiązania problemu o złożoności **lepszej niż** $f(n)$

$\Theta(f(n))$ złożoność „**wrodzona**” problemu jest określona funkcją $f(n)$ i znany jest algorytm optymalny o tej złożoności

Definicja formalna (wszystkie funkcje $N \rightarrow R^+$)

$$O(f(n)) = \{ g(n) \mid \exists (c > 0, n_0 \in N) \forall (n > n_0) : g(n) \leq c f(n) \}$$

$$\Omega(f(n)) = \{ g(n) \mid \exists (c > 0, n_0 \in N) \forall (n > n_0) : g(n) \geq c f(n) \}$$

$$\Theta(f(n)) = \Omega(f(n)) \cap O(f(n))$$



Przykład: $T(n)$ dla problemu maxSum

Wg algorytmu maxSum1(): $T(n) \in O(n^2)$

Wg algorytmu maxSum2(): $T(n) \in O(n \log n)$

Wg algorytmu maxSumOpt(): $T(n) \in O(n)$

Trywialne ograniczenie dolne: $T(n) \in \Omega(n)$

(wszystkie n elementy są istotne)

"Wrodzona" złożoność maxSum: $\Theta(n)$

Uwaga: Używając notacji asymptotycznej należy pozostawiać w zapisie **TYLKO** składnik decydujący o tempie wzrostu wartości z pominięciem współczynnika proporcjonalności. Na przykład (przykład "kolejowy"):
zamiast $O(3n^2 + 5n)$ pisać $O(n^2)$



Dla rozwiązania maxSum2 mamy równanie rekurencyjne:

$$T(n) = 2T(n/2) + O(n) = 2T(n/2) + n \quad [\text{pomijamy ukryty wsp.}]$$

Założmy, że rozmiar tablicy jest $n = 2^k$ (dla zachowania asymptotycznego to założenie jest neutralne). Oznaczając $t(k) = T(n)$, $k = \log n$ dostajemy rekurencję liniową:

$$t(k) = 2t(k-1) + 2^k ; \quad t(0) = T(1) = 1 \quad [\text{koszt stały dla } n = 1]$$

Dla takiego równania rekurencyjnego teoria podpowiada postać ogólną rozwiązania: **$t(k) = (Ak + B)2^k$** ,

gdzie współczynniki A , B trzeba wyznaczyć z warunków początkowych: $t(0) = 1$, $t(1) = 2t(0) + 2^1 = 4$

$$t(0): B = 1; \quad t(1): (A+1)2 = 4; \text{ czyli } A=1$$

Rozwiązanie ma zatem postać: $t(k) = (k + 1)2^k$,

Wracając do oryginału: **$T(n) = (\log n + 1)n \in O(n \log n)$**



Myślenie rekurencyjne



W matematyce i informatyce używa się dwu pokrewnych pojęć:

- **Indukcja** (np. definicja indukcyjna, dowód indukcyjny)
- **Rekurencja** (np. funkcja zdefiniowana rekurencyjnie)

Przykład

- **Definicja indukcyjna ciągu Fibonacciego F_n , $n \geq 0$ [0,1,1,2,3,5, ...]**
 - $F_0 = 0$, $F_1 = 1$ są liczbami Fibonacciego
 - Jeżeli F_{k-1} i F_k są kolejnymi liczbami Fibonacciego, to $F_{k+1} = F_k + F_{k-1}$
- **Funkcja rekurencyjna obliczania elementów ciągu Fibonacciego**

```
int Fibo(int n) {    // n>=0 – numer elementu ciągu
    if(n==0 || n==1) return n; //Bez rekursji
    return Fibo(n-1)+Fibo(n-2); //2 odwołania rekurencyjne
}
```

W praktyce definicję indukcyjną i rekurencyjną uważa się za równoważne.



Rekurencję stosuje się często szukając koncepcji rozwiązania problemu – jest to naturalna technika do wykorzystania w metodzie dekompozycji (p. maxSum2):

- **OBOWIĄZKOWO:** Jeżeli problem jest "mały", to rozwiąż go bezpośrednio i zwróć wynik (przypadek bazowy)
- W przeciwnym przypadku
 - podziel problem na podproblemy mniejsze
 - rozwiąż każdy z podproblemów (**rekurencja**)
 - komponuj z rozwiązań podproblemów rozwiązanie problemu oryginalnego

Definicja (procedury, funkcji, obiektu) jest rekurencyjna, jeżeli odwołuje się do samej siebie.



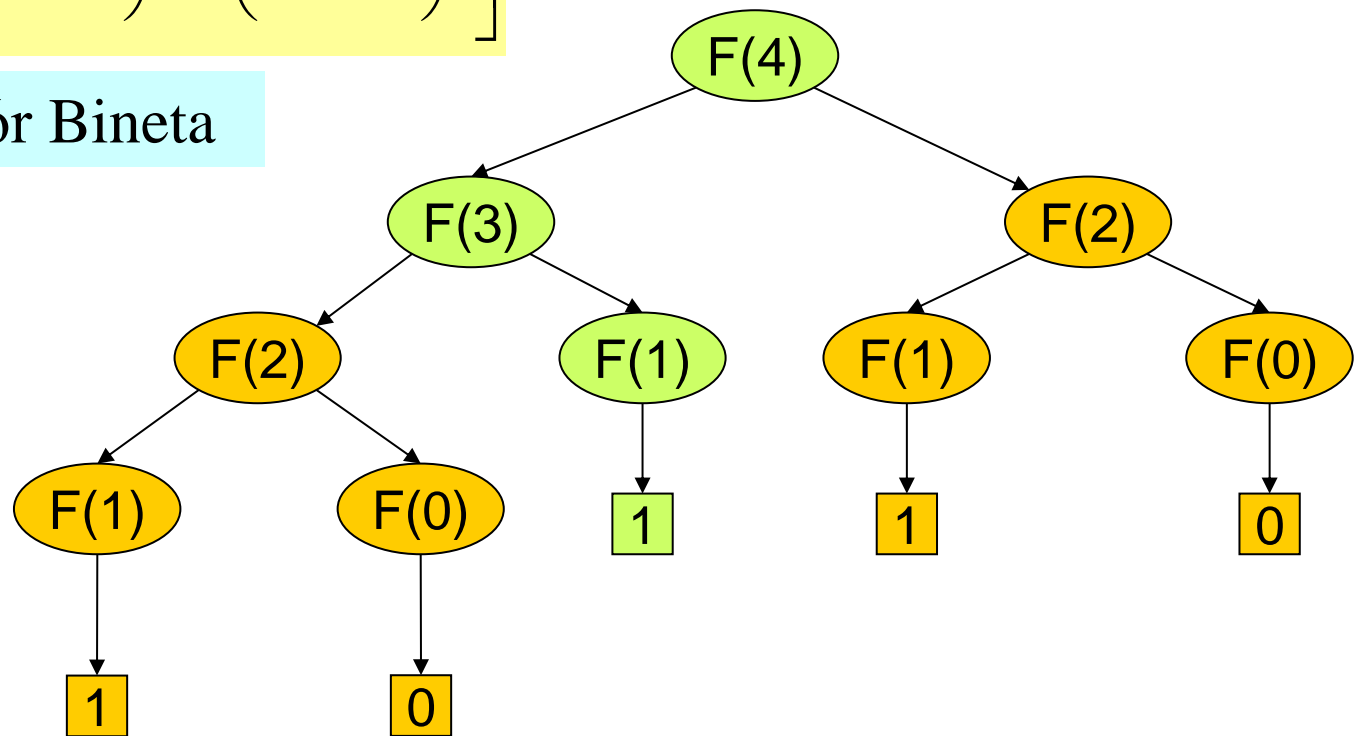
Rodzaje rekurencji

- **Bezpośrednia** (jak w `Fibo(.)`, `maxSum2(.)`)
- **Pojedyncza** (co najwyżej jedno odwołanie rekurencyjne)
– łatwo daje się zastąpić iteracją
- Rekurencja "**ogonowa**" (ostatnią czynnością jest wywołanie siebie)
- **Mnoga** (np. jak w funkcji `Fibo(.)`)
- **Pośrednia** – występuje łańcuch pośrednich odwołań, np.:
 $f(.) \rightarrow g(.) \rightarrow h(.) \rightarrow f(.)$
- **Sieciowa** - sieć zależności rekurencyjnych (sieć powiązań cyklicznych)

Rekursja mnoga może być kosztowna, jeżeli prowadzi do powtórzeń akcji (np. rekurencyjne obliczanie `Fibo(.)`).

$$f(n) = \frac{1}{\sqrt{5}} \left[\left(\frac{1 + \sqrt{5}}{2} \right)^n - \left(\frac{1 - \sqrt{5}}{2} \right)^n \right]$$

Wzór Bineta



Obliczenia wielokrotne wg definicji rekurencyjnej ($F \equiv \text{Fibo}$)



Przykład: Obliczyć, na ile sposobów można zadaną liczbę naturalną (>0) przedstawić w postaci sumy składników dodatnich nie większych od ustalonego maksimum.

Np. liczbę 7 można przedstawić jako sumę składników nie większych od 3 tak:

$$1+1+1+1+1+1+1$$

$$2+1+1+1+1+1$$

$$2+2+1+1+1$$

$$2+2+2+1$$

$$3+1+1+1+1$$

$$3+2+1+1$$

$$3+2+2$$

$$3+3+1$$



Oznaczmy $q(m, n)$ funkcję obliczającą liczbę rozkładów m na sumy składników nie większych od n (m, n naturalne):

Warunki początkowe

$q(1, n) = 1$; jedynie jako ona sama

$q(m, 1) = 1$; jeden rozkład: $1+1+\dots+1$ (m jedynek)

Obcięcie

$q(m, n) = q(m, m)$ jeżeli $n > m$; nie ma rozkładów ze składnikami większymi od liczby rozkładanej

Redukcje

$q(m, m) = q(m, m-1) + 1$ wydzielamy m jako "rozkład" m

$q(m, n) = q(m, n-1) + q(m-n, n)$ oddzielnie grupa ze składnikiem obowiązkowym n



W zapisie zwartym funkcja $q(m,n)$:

1. $q(1, n) = 1$;
2. $q(m, 1) = 1$;
3. $q(m, n) = q(m, m)$ $m < n$
4. $q(m, m) = q(m, m-1) + 1$ $m = n$
5. $q(m, n) = q(m, n-1) + q(m-n, n)$ $m > n$

=====

$$\begin{aligned} q(7,3) &= q(7,2) + q(4,3) \\ &= q(7,1) + q(5,2) + q(4,3) \\ &= 1 + q(5,1) + q(3,2) + q(4,3) \\ &= 2 + q(3,1) + q(1,2) + q(4,3) \\ &= 4 + q(4,2) + q(1,3) = 5 + q(4,2) \\ &= 5 + q(4,1) + q(2,2) = 6 + q(2,1) + 1 = 8 \end{aligned}$$

$$q(7, 3) = 8$$

$$1+1+1+1+1+1+1$$

$$2+1+1+1+1+1$$

$$2+2+1+1+1$$

$$2+2+2+1$$

$$3+1+1+1+1$$

$$3+2+1+1$$

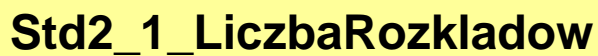
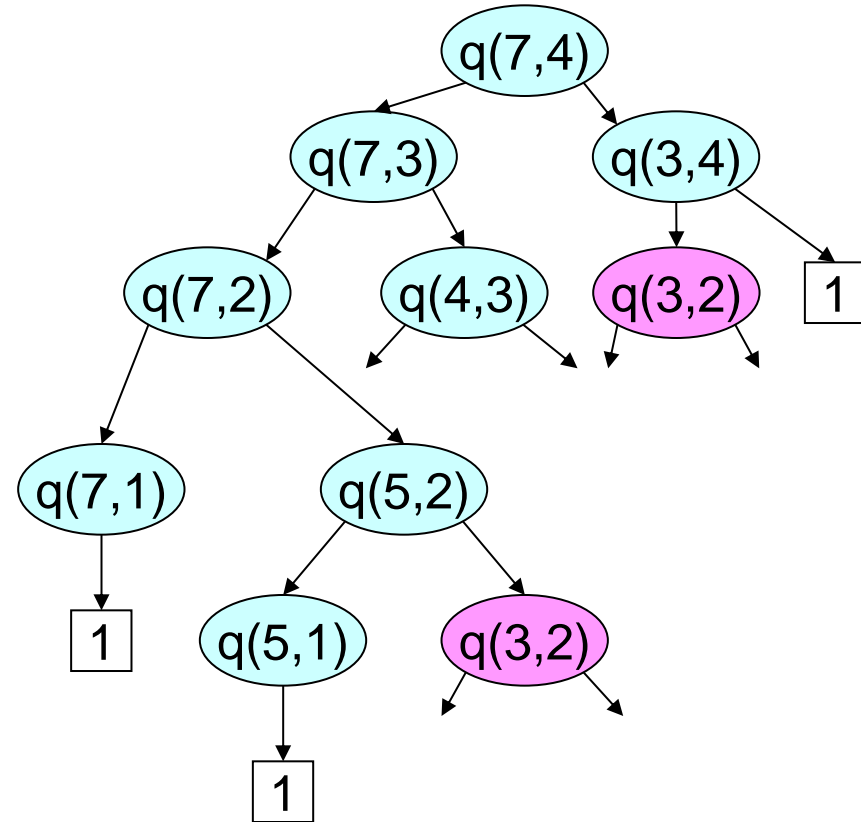
$$3+2+2$$

$$3+3+1$$



q(10, 10) = 42
Liczba wywołań q(): 71

Fragment drzewa wywołań dla $q(7, 4)$





- Tablica jest uznawana za strukturę wbudowaną języka programowania (agregacja indeksowana, *random access*)
- Cechy tablic w Javie:
 - Należą do typów **referencyjnych** – są obiektami w ogólnym sensie
`int[] A = new int[10]; // 10 elementów, indeks 0..9`
`Object ob = A; // OK`
 - **Rozmiar tablicy jest niezmienny**, ustalony w momencie tworzenia i dostępny poprzez atrybut **length** (`A.length == 10`)
 - **Inicjalizacja** tablicy odbywa się w czasie wykonania programu (nie podczas kompilacji); można na przykład napisać:
`int n;`
`// ...`
`int[] B = {prime(n), prime(n+1)};`
 - **Czas "życia"** wynika z ogólnych zasad Javy dla obiektów:
`A = new int[100]; // Porzucenie i nowa alokacja`
 - Klasa **java.util.Arrays** zawiera 155 metod statycznych do manipulacji tablicami (wypełnianie, konwersje, kopiowanie, wyszukiwanie, sortowanie, ...)



Struktury liniowe: **porządkowanie**



- Tablice często są kontenerami obiektów podlegających porządkowaniu wg określonej **relacji porządku**
- **Porządek naturalny obiektów**: relacja określona poprzez implementację interfejsu `Comparable<T>` w klasie

```
public interface Comparable<T> {  
    int compareTo(T t);  
} // this.compareTo(t); zwraca -:0:+  
// Porządek liniowy                < = >
```

- Wszystkie typy pierwotne (`byte`, `int`, ...) mają wbudowaną naturalną relację porządku; typy referencyjne "opakowujące" (`Byte`, `Int`, ...) realizują `Comparable<Typ>`
- Próba wymuszenia porządkowania dla tablicy obiektów bez określonej relacji porządku \Rightarrow **ClassCastException**



Struktury liniowe: **porządkowanie (cd1)**



- Jeżeli porządek naturalny nie jest zdefiniowany, albo porządkowanie ma dotyczyć specjalnej relacji, trzeba użyć interfejsu funkcyjnego **Comparator<T>**:

```
@FunctionalInterface
public interface Comparator<T> {
    int compare(T t1, T t2);
} // Porządek liniowy t1:t2 wg konwencji
// compareTo()
```

- Obiekt komparatora (**albo wyrażenie lambda** dla compare()) jest dostarczony do metody realizującej porządkowanie poprzez parametr wywołania.
- Ten wariant porządku nazywamy **wymuszonym przez komparator** (w przeciwieństwie do "naturalnego")



Przykład stosowania: Std2_7_GenericArrayHeap



Wyszukiwanie binarne (połówkowe)



Wyszukiwanie połówkowe

Dane: **uporządkowana** tablica $T[n]$; szukany element x

Wynik: jeśli x jest w T , zwróć indeks pierwszego wystąpienia; jeśli brak zwróć wartość -1

Algorytm: **Pseudokod**

$\text{Szukaj}(T, d, g, x)$ // zakres $[d..g)$

{ if($g==d$) return -1; // Brak

$m = (d+g)/2$; // indeks środka

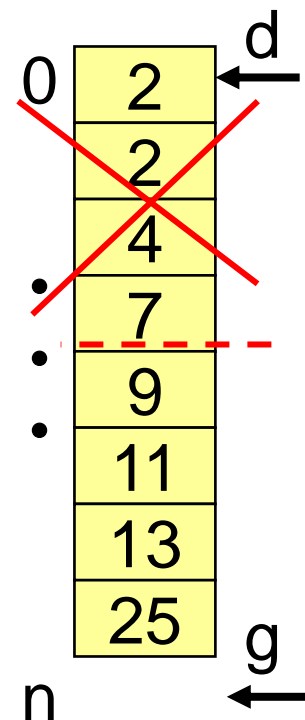
 if($x == T[m]$) return m ;

 if($x < T[m]$) // rekursja

 return $\text{Szukaj}(T, d, m, x)$;

 return $\text{Szukaj}(T, m+1, g, x)$;

}



Koszt rzędu $O(\log_2 n)$. Dla $n = 1024$ trzeba około 10 porównań.



Wyszukiwanie binarne (cd1)



```
// Java: Znajdź element x w tablicy
// uporządkowanej T w przedziale indeksów d..g-1
static int szukaj(int T[], int d, int g, int x) {
    if(g==d) return -1; // Pusta tablica, brak x

    int m = (d+g)/2;      // indeks środka
    if(x==T[m]) return m; // znaleziono x

    if(x < T[m])
        return szukaj(T, d, m, x); // rekursja pojedyncza
    return szukaj(T, m+1, g, x);
}
// Ten sam efekt: i = Arrays.binarySearch(T, d, g, x);
```





Wyszukiwanie binarne: przykład wykonania



	0	1	2	3	4	5	6	7	8	9
T:	2	4	6	9	12	20	22	30	35	

x=20

```
Szukaj(T[], 0, 9, 20)
|  Szukaj(T[], 5, 9, 20)
|  |  Szukaj(T[], 5, 7, 20)
|  |  |  Szukaj(T[], 5, 6, 20)
|  |  |  return
|  |  return
|  return
return
znaleziono: T[5] = 20
```

x=17

```
Szukaj(T[], 0, 9, 17)
|  Szukaj(T[], 5, 9, 17)
|  |  Szukaj(T[], 5, 7, 17)
|  |  |  Szukaj(T[], 5, 6, 17)
|  |  |  |  Szukaj(T[], 5, 5, 17)
|  |  |  |  return
|  |  |  return
|  |  return
|  return
return
Brak
```



Wyszukiwanie binarne – bez rekursji



```
//Szukanie połówkowe bez rekursji
//Znajdź element x w tablicy uporządkowanej T[n]
static int szukaj_br(int T[], int n, int x)
{
    int d=0, g=n, m;
    while(d < g)
    {
        m = (d+g)/2;           // indeks środka
        if(x == T[m]) return m; // znaleziono x

        if(x < T[m]) // szukaj w dolnej połowie: zmień g
            g = m;
        else // szukaj w górnej połowie: zmień d
            d = m+1;
    }
    return -1; // d>=g, brak x
}
```

Zmodyfikować metodę tak, aby w przypadku braku elementu x zwracała:
– **indeksPotencjalnegoWstawienia-1**,
(-n-1 jeśli na końcu).





Sortowanie przez wybieranie (selectionSort)

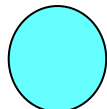
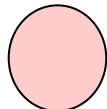
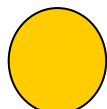
- Dane: dowolna tablica T , n obiektów (indeksy $0 \dots n-1$)
- Wynik: tablica uporządkowana niemalejąco; $T[i] \leq T[j]$, $i < j$
- Metoda: (idea w zapisie rekurencyjnym)

```
selSort(T, n) {  
    if(n == 1) return; // 1 element, nic do roboty  
    // n > 1: znajdź indeks elementu maks. w T  
    imax = max_index(T, n); // koszt  $O(n)$   
    // Zamień miejscami parę  $T[imax]$ ,  $T[n-1]$   
    swap(T, imax, n-1);  
    // Dokończ dzieło: sortuj pozostałe elementy  
    selSort(T, n-1); // Rekursja "ogonowa"  
} // złożoność:  $O(n^2)$ 
```



Sortowanie 1: przez wybieranie (cd1)



T:	41	66	15	61	91	11	32	33	59	77	
9:	41	66	15	61	77	11	32	33	59	91	aktualne maksimum
8:	41	66	15	61	59	11	32	33	77	91	
7:	41	33	15	61	59	11	32	66	77	91	
6:	41	33	15	32	59	11	61	66	77	91	element do wymiany
5:	41	33	15	32	11	59	61	66	77	91	
4:	11	33	15	32	41	59	61	66	77	91	
3:	11	32	15	33	41	59	61	66	77	91	maksimum na pozycji właściwej
2:	11	15	32	33	41	59	61	66	77	91	
1:	11	15	32	33	41	59	61	66	77	91	



Sortowanie przez wybieranie – wersja iteracyjna

```
static void selSort_iter(int[] T) {  
    int n = T.length;  
    int imax;  
    for(int i=n-1; i>0; --i){  
        // oblicz imax = maxIndex(T, i+1);  
        imax = 0;  
        for(int j=1; j<=i; ++j)  
            if(T[j]>T[imax]) imax = j;  
        swap(T, imax, i);  
    }  
} // złożoność  $O(n^2)$ 
```




Sortowanie bąbelkowe (bubbleSort)

- Dane: dowolna tablica T , n obiektów (indeksy $0 \dots n-1$)
- Wynik: tablica uporządkowana niemalejąco; $T[i] \leq T[j]$, $i < j$
- Metoda: skanuj tablicę w poszukiwaniu sąsiedztwa pary elementów **niespełniających** relacji porządku. Jeśli brak takiego sąsiedztwa, to tablica jest uporządkowana. Jeśli jest – zamień miejscami i skanuj dalej.

```
static void bubbleSort(int T[], int n) {  
    bool wymiana; // Czy była w ostatnim skanowaniu?  
  
    do {  
        wymiana = false;  
        for(int i=0; i<n-1; ++i) {  
            if(T[i]>T[i+1])  
                { swap(T, i, i+1); wymiana = true; }  
        }  
    } while(wymiana);  
} // Złożoność:  $O(n^2)$ 
```



Sortowanie 2: bąbelkowe (cd1)



skan#

1: 41 66 15 61 91 11 32 33 59 77

2: 41 15 61 66 11 32 33 59 77 91

3: 15 41 61 11 32 33 59 66 77 91

4: 15 41 11 32 33 59 61 66 77 91

5: 15 11 32 33 41 59 61 66 77 91

6: 11 15 32 33 41 59 61 66 77 91



wymiana
sąsiadów
podczas
skanowania



Sortowanie przez wstawianie (insertionSort)

- Dane: dowolna tablica T , n obiektów (indeksy $0 \dots n-1$)
- Wynik: tablica uporządkowana niemalejąco; $T[i] \leq T[j]$, $i < j$
- Metoda: Niech fragment tablicy do indeksu k będzie uporządkowany niemalejąco. Weźmy pod uwagę element $T[k+1]$. Posuwając się w kierunku malejących indeksów wstawiamy $T[k+1]$ na właściwe miejsca (zamieniając miejscami sąsiadów jeśli trzeba)

```
static void insertionSort(int T[]) {  
    int n = T.length, x, j;  
    for(int i=1; i<n; ++i) {  
        x = T[i];  
        for(j=i-1; j>=0 && T[j]>x; --j)  
            T[j+1] = T[j];  
        T[j+1] = x;  
    }  
} // złożoność:  $O(n^2)$ 
```



T: 41 66 15 61 91 11 32 33 59 77
 1: 41 66 15 61 91 11 32 33 59 77
 2: 15 41 66 61 91 11 32 33 59 77
 3: 15 41 61 66 91 11 32 33 59 77
 4: 15 41 61 66 91 11 32 33 59 77
 5: 11 15 41 61 66 91 32 33 59 77
 6: 11 15 32 41 61 66 91 33 59 77
 7: 11 15 32 33 41 61 66 91 59 77
 8: 11 15 32 33 41 59 61 66 91 77
 9: 11 15 32 33 41 59 61 66 77 91



fragment
posortowany



Sortowanie szybkie (quickSort, T. Hoare 1959)

- Dane: dowolna tablica T , n obiektów (indeksy $0 \dots n-1$)
- Wynik: tablica uporządkowana niemalejąco; $T[i] \leq T[j]$, $i < j$
- Metoda:
 - Wybierz pewien element s tablicy T (separator, *ang. pivot*).
 - Wg elementu s podziel T na 2 podtablice: A z elementami $\leq s$, B z elementami $\geq s$. [koszt $O(n)$]
 - Rekurencyjnie sortuj (quicksort) podtablice A i B
 - Otrzymujemy uporządkowaną całą tablicę T
- Koszt: zależy od wyboru s ; jeżeli s generuje w przybliżeniu zrównoważony podział na A i B , to $O(n \log n)$; niekorzystny podział daje ocenę pesymistyczną $O(n^2)$.





T:	d	A: $\leq s$	B: $\geq s$	g
-----------	----------	-------------------------------	-------------------------------	----------

```
static void quickSort(int[] T, int d, int g)// indeksy d..g
{ int i, j, s;
  s= T[(d+g)/2];// Przykładowy wybór separatora ze środka T
  i = d; j = g;
  do
  { while(T[i]<s) ++i;
    while(T[j]>s) --j;
    if(i<=j) { swap(T, i, j); ++i; --j; }
  } while(i<j);
  if(d<j) quickSort(T, d, j);
  if(i<g) quickSort(T, i, g);
}
```



Sortowanie 4: szybkie (cd2)



T:	d	A: $\leq s$	B: $\geq s$	g
----	---	-------------	-------------	---

0 1 2 3 4 5 6 7 8 9

42 54 98 68 63 83 94 55 35 12

quicksort(T, 0, 9)

42 54 12 68 63 83 94 55 35 98

swap 2, 9

i *j*

42 54 12 35 63 83 94 55 68 98

swap 3, 8

i *j*

42 54 12 35 55 83 94 63 68 98

swap 4, 7

i *j*

42 54 12 35 55 83 94 63 68 98

podtablica A i B

j *i*

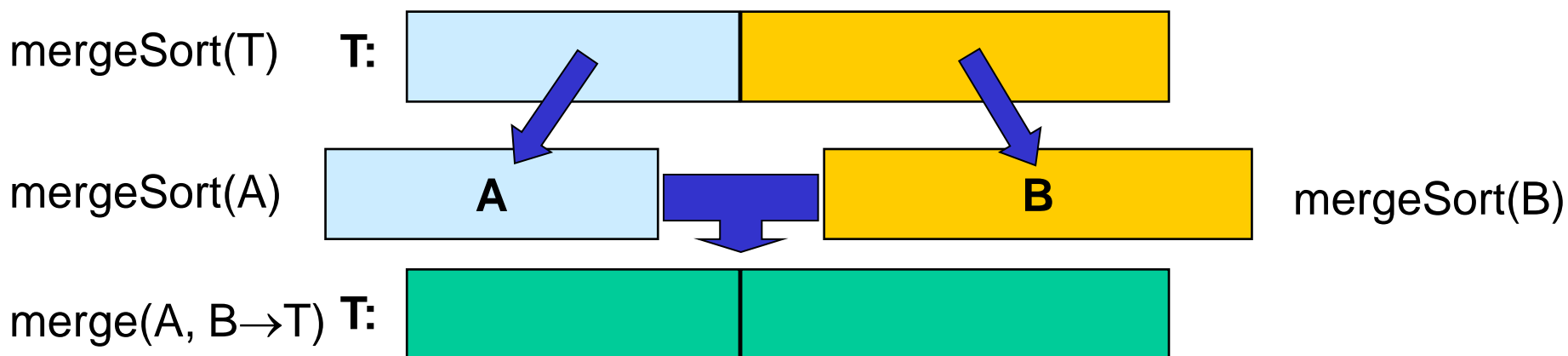
Rekursja

↙
quickSort(T, 0, 4)

↘
quickSort(T, 5, 9)

Sortowanie przez scalanie (mergeSort)

- Dane: dowolna tablica T , n obiektów (indeksy $0 \dots n-1$)
- Wynik: tablica uporządkowana niemalejąco; $T[i] \leq T[j]$, $i < j$
- Metoda:
 - Podziel tablicę T na dwie połówkowe podtablice: $T \Rightarrow A \cdot B$
 - Rekurencyjnie sortuj (mergesort) tablice A i B
 - Scal posortowane tablice A , B do T
- Złożoność: $T(n) = 2T(n/2) + O(n)$; $\Rightarrow T(n) \in O(n \lg n)$





Sortowanie 5: przez scalanie (cd1)



```
static void mergeSort(int T[]) {  
    int n = T.length;  
    if(n<=1) return; // Nie ma nic do roboty  
    int m1 = n/2, m2 = n-m1;  
    int[] A = new int[m1], B = new int[m2];  
    for(int i=0; i<m1; ++i) A[i] = T[i];  
    for(int i=0; i<m2; ++i) B[i] = T[i+m1];  
    mergeSort(A); mergeSort(B); // Rekursja  
  
    // Scalanie  
    int i1=0, i2=0, i=0; // i: indeks odbiorczy w T  
    while(i1<m1 && i2<m2)  
        if(A[i1]<B[i2]) T[i++] = A[i1++]; else T[i++] = B[i2++];  
  
    while(i1<m1) T[i++] = A[i1++]; // Kopiowanie remanentu  
    while(i2<m2) T[i++] = B[i2++];  
}
```



Kopiec (sterta, heap) binarny: struktura



Kopiec (heap)

Koncepcyjnie, jest to (**wyobrażone**)
drzewo binarne **lewostronnie**
pełne z lokalną relacją porządku w
każdym węźle wewnętrznym.

Reprezentacja kopca: tablica.

MinHeap: relacja porządku \geq ,

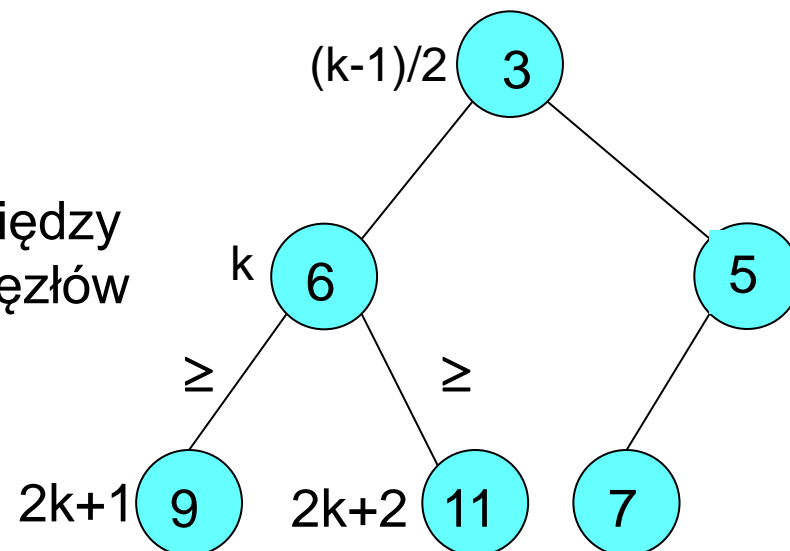
MaxHeap: relacja porządku \leq .

MinHeap w tablicy T[6]

	0	1	2	3	4	5
T:	3	6	5	9	11	7



Relacje pomiędzy
indeksami węzłów

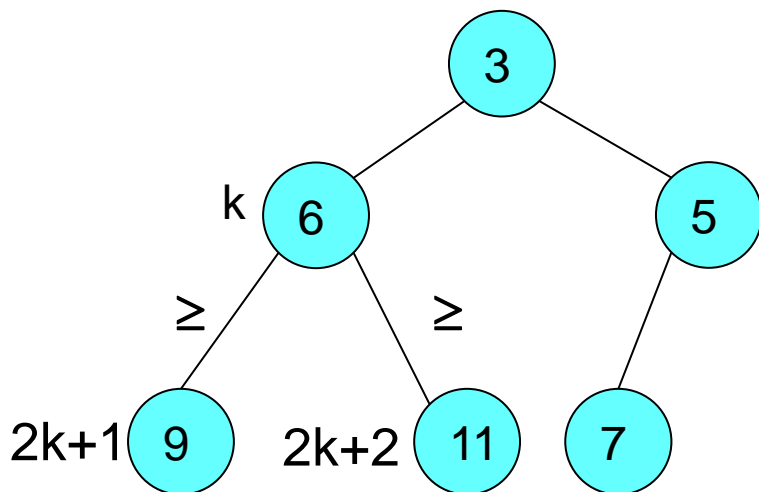




Kopiec binarny: operacje



```
class MinHeap {  
    private int [ ] H; // Podległa tablica  
    private int n;    // Liczba elementów  
  
    public MinHeap(size){/* */}  
    public void push(elem){/* */}  
    public int  getMin(){/* */}  
    public int  remMin(){/* */}  
    public void pushMinHeap(int [ ] T){/* */}  
    public void makeMinHeap(int [ ] T){/* */}  
    // ... inne składowe  
}
```



Operacje na kopcach (MinHeap)

push(x); // ++n (liczba elementów)

Dodaj x do kopca; **O(log n)**

getMin();

Zwraca min. – kopiec b.z.; **O(1)**

remMin(); // --n;

Zwraca i usuwa min. z kopca;

O(log n)

makeMinHeap(T);

Kopiec "hurtem" z tablicy T; **O(n)**

pushMinHeap(T);

Dodaje kolejne elementy T przy pomocy push(x); **O(n log n)**

heapsort(T);

Sortowanie z kopcowaniem;

O(n log n)



Kopiec binarny: operacje (cd2)



```
// Prywatna metoda 'przetaczania' elementu w dół kopca
private void moveDown(int k)
// Ustanowienie warunku kopca w węźle k pod warunkiem
// istnienia poprawnych kopców w węzłach potomnych
{ int e = H[k]; // Ten element może migrować w dół kopca
  int j=2*k+1; // indeks lewego potomka
  while(j<n)
  { if(j<n-1 &&
      H[j]>H[j+1]) ++j; // j: indeks mniejszego potomka
    if(e <= H[j])
      break; // Już OK.
    H[k] = H[j];
    k = j; // w dół
    j = 2*k+1;
  }
  H[k] = e; // Osadzenie obiektu w odpowiednim miejscu
}
```



Kopiec binarny: operacje (cd3)

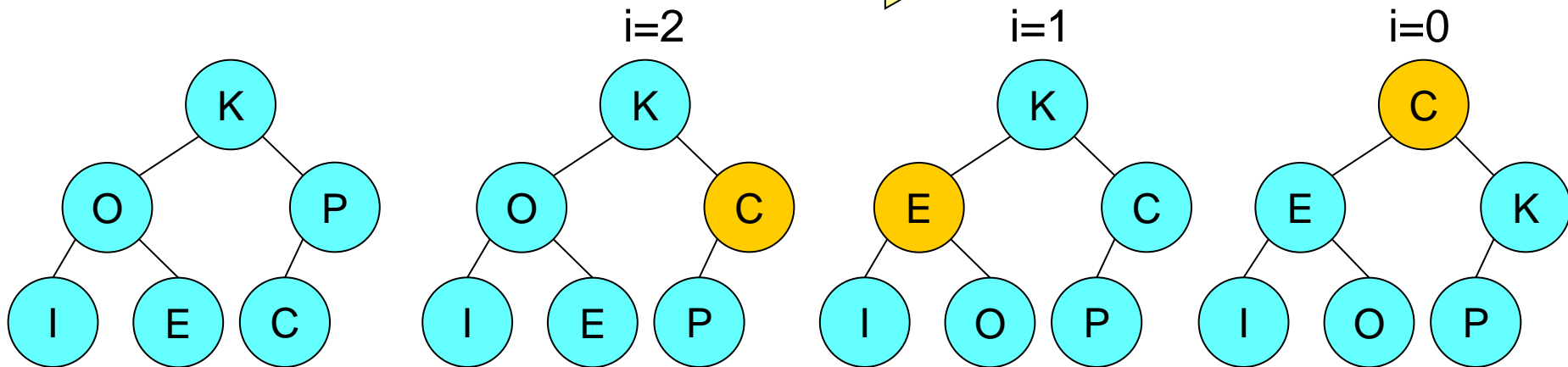


```
public void makeMinHeap(int [] A){ // Kopiec "hurtem".  
    H = Arrays.copyOf(A, A.length);  
    n = size = A.length;  
  
    // Wymuszanie warunków kopca od ostatniego węzła wew.  
    for(int i=(n-2)/2; i>=0; --i)  
        moveDown(i);  
} // Koszt O(n)
```

H: K O P I E C

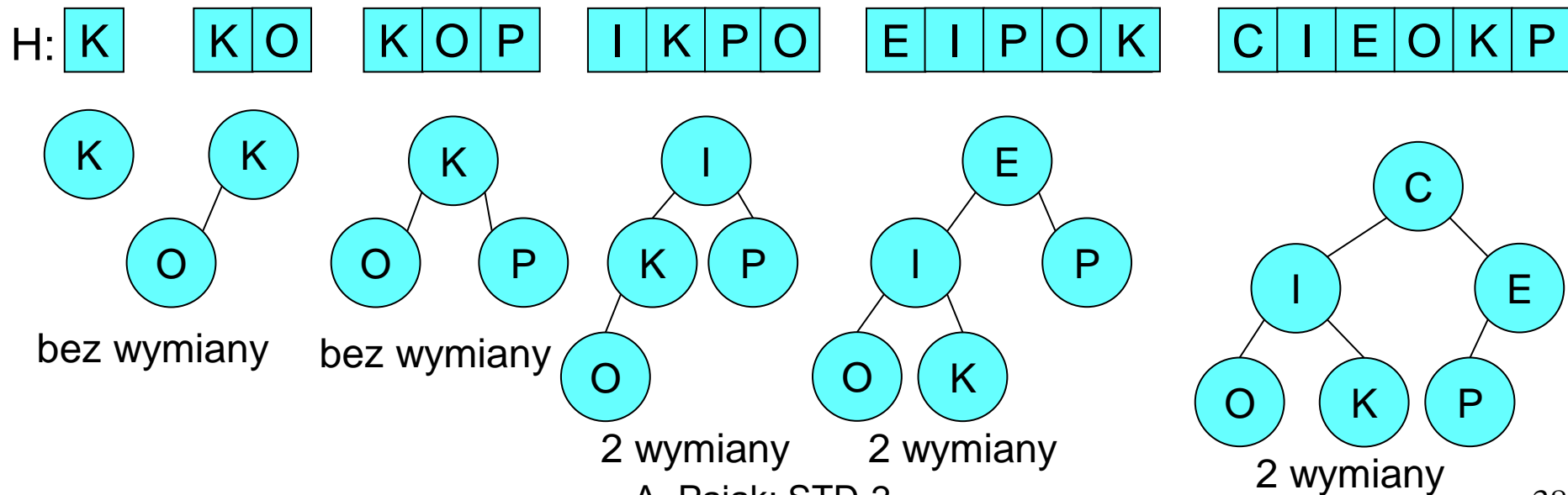
MakeMinHeap

H: C E K I O P



```
public void pushMinHeap(int [] A){
    // Dokładaj do kopca elementy z A[].
    // Pojemność kopca musi wystarczyć.
    for(int i=0;i<A.length;i++)
        push(A[i]);
} // koszt  $O(n \log n)$ 
```

Inkrementalne budowanie kopca





Kopiec binarny: operacje (cd5)



Kopiec można budować na dwa sposoby:

- "Hurtem" – dla wypełnionej tablicy T uruchamia się operację `MakeMinHeap()`; koszt tej operacji $O(n)$
- Inkrementalnie dodając po jednym elemencie `AddToHeap()`; koszt jest $O(n \log n)$

Dwie metody zwykle dają różne końcowe zawartości tablicy:

- K O P I E C "hurtem" daje: C E K I O P
- K O P I E C inkrementalnie daje: C I E O K P

Te różnice nie mają znaczenia dla głównej misji kopca – realizacji kolejki priorytetowej.



Sortowanie z użyciem kopca (heapsort)

- Dane: dowolna tablica T , n obiektów (indeksy $0 \dots n-1$)
- Wynik: tablica uporządkowana niemalejąco; $S[i] \leq S[j]$, $i < j$
- Metoda:
 - Utwórz kopiec wg tablicy T przy pomocy operacji `MakeMinHeap()` – na szczycie kopca będzie wystawiony element minimalny
 - Pobieraj z kopca kolejne elementy do tablicy wynikowej

```
static int[] heapsort(int[] T) {  
    MinHeap mh = new MinHeap(T.length); // o(1)  
    mh.makeMinHeap(T);                  // o(n)  
    int[] S = new int[T.length];        // o(n)  
    for(int i=0; i<T.length; ++i)       // o(n log n)  
        S[i] = mh.remMin();  
    return S;  
} // Koszt  $O(n \log n)$ 
```

Bardziej efektywny algorytm: budowanie kopca i sortowanie w tablicy źródłowej.



JCF zawiera klasę generyczną **PriorityQueue<E>**, która oferuje podobną funkcjonalność jak kopiec. Nazewnictwo jest specyficzne dla JCF (w innych językach zupełnie inne, np. biblioteka C++).

Ważniejsze metody:

- add(E e)** Wstawia element e do kolejki; zwraca true jeśli sukces
- offer(E e)** Wstawia element e do kolejki; zwraca true jeśli sukces
- peek()** Zwraca (bez usuwania) element priorytetowy albo null jeśli kolejka pusta
- poll()** Zwraca (i usuwa) element priorytetowy albo null jeśli kolejka pusta
- remove(Object o)** Usuwa jeden element podany w parametrze, jeśli jest w kolejce; zwraca true jeśli sukces
- toArray()** Zwraca tablicę zawierającą wszystkie elementy kolejki
- size()** Zwraca liczbę elementów w kolekcji



Problem: W tablicy $T[n]$ znaleźć k -te minimum.

Definicja (operacyjna)

$x \in T$ jest k -min(T), jeżeli po wykonaniu operacji $\text{sortuj}(T)$;
 $x == T[k]$ (indeks liczony od 1 albo od 0, wg konwencji języka).

- Koszt wyznaczania 1-min (0-min): $n-1$ porównań
- Koszt wyznaczania n -min (czyli max): $n-1$ porównań
- Koszt wyznaczenia 2-min: ??

Uwaga: Nie można wyznaczyć 2-min nie znając 1-min

- Koszt wyznaczenia mediany: ??



Rozwiązanie naiwne: $2n-3$ porównań

Wyznacz 1-min z n elementów; ($n-1$ porównań)

Wyznacz 1-min z pozostałych $n-1$ ($n-2$ porównania)

Rozwiązanie turniejowe w 2 etapach (elementy = gracze):

- 1) Rozgrywki pierwszej rundy (ewentualnie 1 gracz czeka)
Rozgrywki drugiej rundy (ewentualnie 1 gracz czeka)

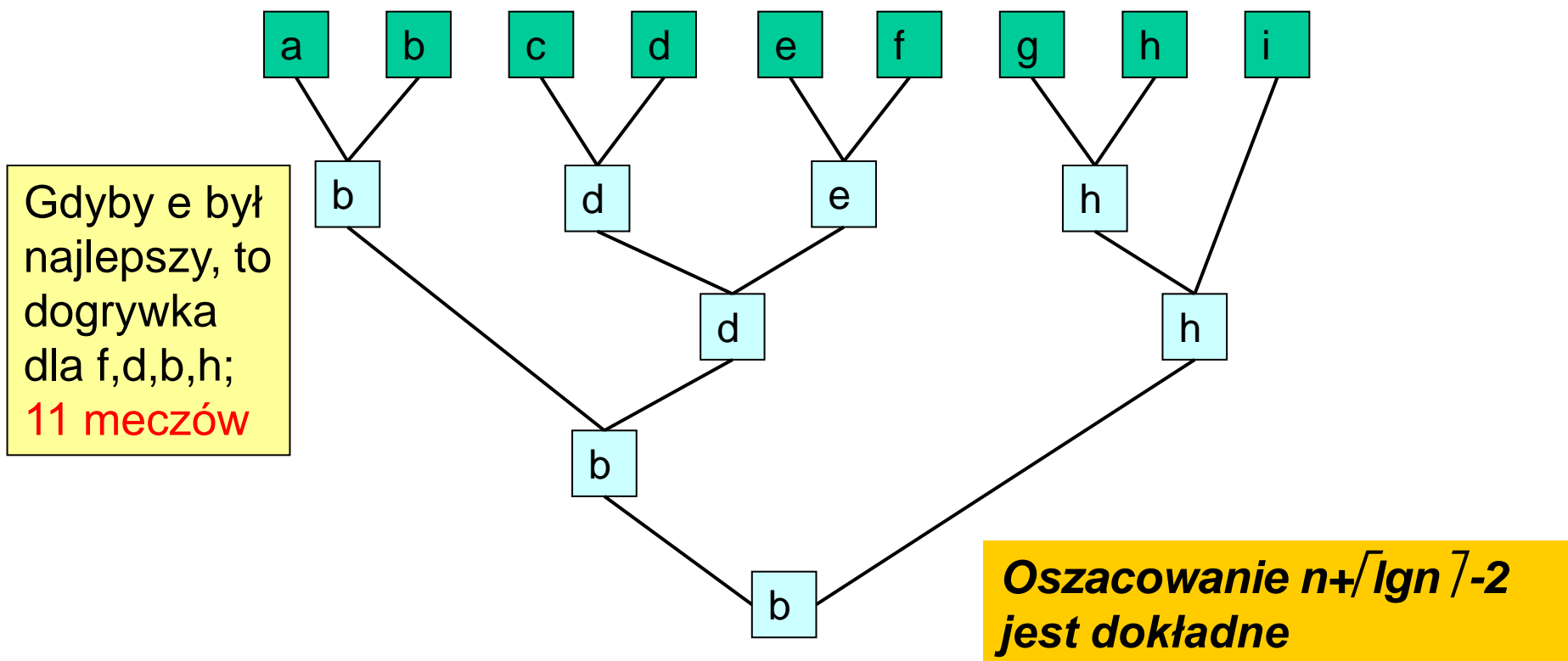
.....

Finał (runda $\lceil \lg n \rceil$) \Rightarrow wyłoniony zwycięzca $b(\text{est})$ (1-min)

Koszt etapu 1): $n-1$ porównań (meczy)

**2-min musi być wśród graczy, którzy przegrali
bezpośrednio ze zwycięzcą; jest ich co najwyżej $\lceil \lg n \rceil$**

- 2) Wśród przegranych z b wyłaniamy 2-min (koszt $\lceil \lg n \rceil - 1$)
Łączny koszt: co najwyżej $(n + \lceil \lg n \rceil - 2)$



Przykładowy turniej 9 graczy w 4 rundach z *najlepszym b*.
Do wyłonienia *drugiego* trzeba rozegrać subturniej pomiędzy graczami a,d,h, którzy przegrali z b; oznacza to dodatkowe 2 mecze; razem 10 meczów ($10 \leq n + \lceil \lg n \rceil - 2 = 11$)



Wg definicji k-min: sortuj(T); T[k]; koszt $O(n \lg n)$

Zastosowanie kopca

```
MinHeap kopiec = new MinHeap(n);
```

```
kopiec.makeMinHeap(T);           // koszt  $O(n)$ 
```

```
// Powtórz k razy
```

```
    x = kopiec.remMin();           // koszt  $O(\lg n)$ 
```

```
return x;                          Łączny koszt:  $O(n + k \lg n)$ 
```

Jeżeli $k < n/\lg n$, to łączny koszt wyznaczenia k-min jest $O(n)$.

Dla mediany (czyli $k = n/2$) koszt jest jednak $O(n \lg n)$

Istnieje liniowy algorytm wyznaczania mediany (algorytm z medianą median).

Jest przybliżone ograniczenie dolne: **$3(n-1)/2$ porównań**



Algorytm selekcji: mediana 5 elementów



Ograniczenie dolne dla mediany:
co najmniej $3(n-1)/2$ porównań

Np. mediana 5 kluczy a, b, c, d, e
wymaga 6 porównań

Pytanie Odp. Wniosek

$a ? b$ $>$ $a > b$

$c ? d$ $>$ $c > d$

(1) $a ? c$ $>$ $a > (b, c, d)$ - nie mediana

$b ? e$ $<$ $e > b$

(2) $c ? e$ $>$ $c > (b, d, e)$ - nie mediana

(3) $d ? e$ $<$ $e > (d, b)$ – e mediana

(3') $d ? e$ $>$ $d > (b, e)$ – d mediana

Uwaga:

**Dla 7 kluczy trzeba co najmniej 10
porównań (wg ograniczenia 9)**

