Algorytmy i struktury danych

sortowanie - pojęcia wstępne,
podstawowe alg. sortowania tablic,
zaawansowane alg. sortowania tablic,
kolejki priorytetowe oparte o kopiec
sortowanie w czasie liniowym,
statystyki pozycyjne

Sortowanie

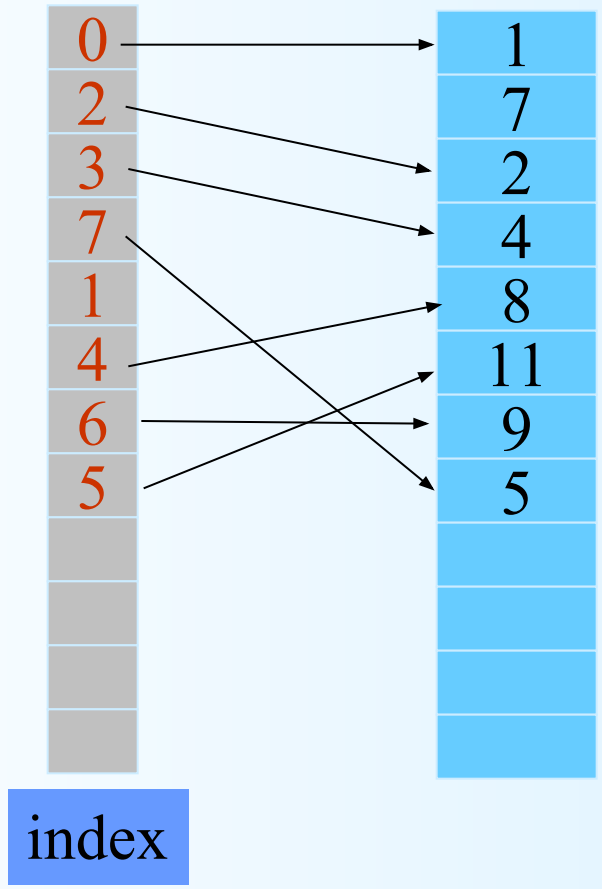
- ✓ Uporządkowanie elementów
- ✓ Klucz – fragment danych, dla którego jest określona relacja uporządkowania
- ✓ Złożoność problemu – najlepszy znany algorytm
- ✓ Sortowanie tablic czy plików
- ✓ Zachowanie uporzadkowania częściowego

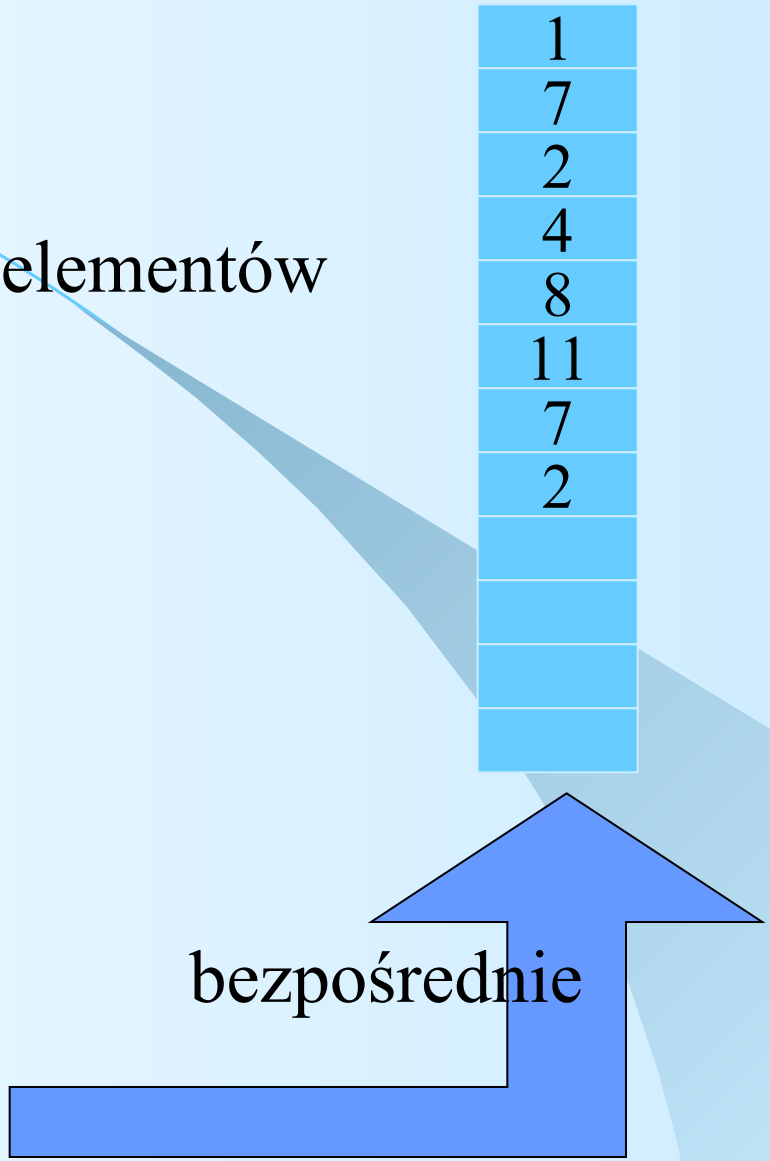
- ✓ Mediana

- ✓ Statystyka pozycyjna

Indeksy

- ✓ Bezpośrednie sortowanie elementów
- ✓ Sortowanie indeksu



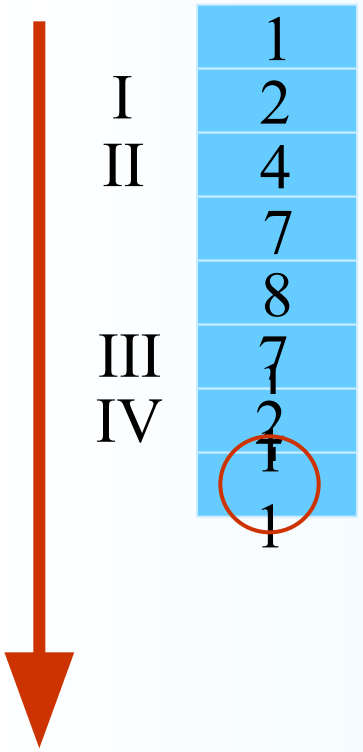


Klasyczne algorytmy dla tablic

- ✓ sortowanie przez wstawianie (Insertion Sort)
  - w kolejnym kroku następny element z części nie posortowanej wstawiany jest na odpowiednią pozycję części posortowanej
- ✓ sortowanie przez wybieranie (Selection Sort)
  - w kolejnym kroku największy (najmniejszy) element z części nie posortowanej wstawiany jest na koniec części posortowanej
- ✓ sortowanie przez zamianę (Exchange, Bubble Sort)
  - zamiana elementów miejscami aż do skutku

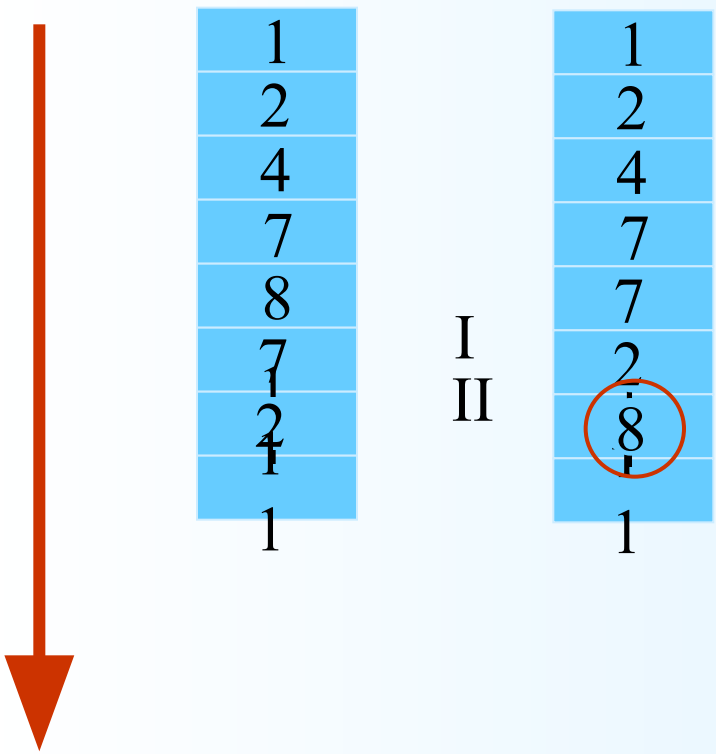
N. Wirth „Algorytmy+Struktury danych = programy”

Sortowanie bąbelkowe




```
for j in range(0,n):
  for i in range(0,n-1):
    if(A[i] > A[i+1]):
      A[i],A[i+1] = A[i+1],A[i]
```

Sortowanie bąbelkowe




```
for j in range(0,n):
  for i in range(0,n-1):
    if(A[i] > A[i+1]):
      A[i],A[i+1] = A[i+1],A[i]
```

Sortowanie bąbelkowe

//wariant podstawowy

```
for j in range(0,n):
  for i in range(0,n-1):
    if(A[i] > A[i+1]):
      A[i],A[i+1] = A[i+1],A[i]
```

//wariant ulepszony wewnętrzna pętla nie przegląda

//uporządkowanych elementów

```
for j in range(n-1,-1,-1):
  for i in range(0,j):
    if(A[i] > A[i+1]):
      A[i],A[i+1] = A[i+1],A[i]
```

Sortowanie bąbelkowe

```
//wariant ulepszony - detekcja konieczności powtórzeń
  for j in range(n-1,-1,-1):
    change = False
    for i in range(0,j):
      if(A[i] > A[i+1]):
        A[i],A[i+1] = A[i+1],A[i]
        change = True
    if not change: break
```

Sortowanie bąbelkowe

Właściwości:

- ✓ alg. prosty w implementacji
- ✓ złożoność $O(n^2)$
- ✓ zwięzły kod - małe stałe proporcjonalności
- ✓ zachowuje uporządkowania częściowe
- ✓ Stosunkowo najgorsze zachowanie spośród prostych metod

Sortowanie przez proste wstawianie

1
2
7
8
11
4
17
22

$X = 2$

```
for i in range (1, n)
    x = A[i]
    p = znajdz_miejsce_na(x)
    A[p] = x
```

- ✓ zachowuje uporządkowania częściowe
- ✓ pesymistyczna złożoność $O(N^2)$
- ✓ mniej zapisów niż przy sortowaniu przez zamianę parami

Sortowanie przez proste wstawianie

1
2
4
7
8
17
22

X = 4

```
for i in range (1, n)
    x = A[i]
    p = znajdz_miejsce_na(x)
    A[p] = x
```

- ✓ zachowuje uporządkowania częściowe
- ✓ pesymistyczna złożoność $O(N^2)$
- ✓ mniej zapisów niż przy sortowaniu przez zamianę parami

Sortowanie przez proste wstawianie

```
def ProsteWstawianie(A, n):
    for i in range (1, n):
        x = A[i]
        j=i
        while j>0 and x<A[j-1]
            A[j] = A[j-1]
            j=j-1
        A[j] = x
```

Sortowanie Shella

Zaawansowane sortownie przez wstawianie elementów

1. Wybieramy $k = k_p$
2. Sortujemy przez wstawianie proste podciagi całej tablicy (co k-ty element)
3. zmniejszamy k tak długo aż osiągnie 1

✓ o złożoności alg. decyduje k_p i sposób zmiany k

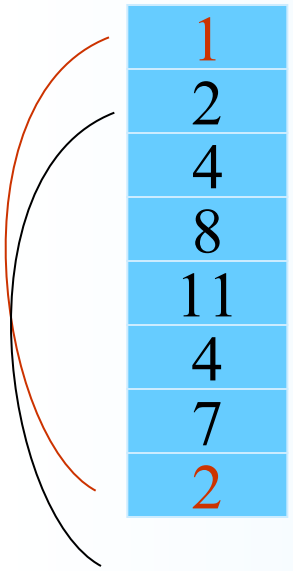
- $k_p = 1$ oznacza zwykłe sortowanie
- dla $k_i = 2^i - 1$ (....., 31, 15, 7, 3, 1)
  algorytm ma złożoność $O(N^{1.2})$
- dla $k_i = 2^r \cdot 3^q$ $(r,q \in \mathrm{N})$ (....., 20, 16, 12, 9, 8, 6, 4, 3, 2, 1)
  algorytm ma złożoność $O(N \log^2 N)$

✓ algorytm zachowuje uporządkowanie częściowe

Sortowanie Shella

```
for k in [ ..., 31, 15, 7, 3, 1] :
    sortuj_przez_wstawianie_k_podtablic(A)
#A[0],A[k],A[2k] ...
    #A[1],A[k+1],A[2k+1] ...
    #...
    #A[k-1],A[2k-1],A[3k-1] ...
```

k=7



- ✓ zachowuje uporządkowanie częściowe
- ✓ pesymistyczna złożoność $O(N^{1.2})$

Sortowanie Shella

```
for k in [ ..., 31, 15, 7, 3, 1] :
    sortuj_przez_wstawianie_k_podtablic(A)
#A[0],A[k],A[2k] ...
#A[1],A[k+1],A[2k+1] ...
#...
#A[k-1],A[2k-1],A[3k-1] ...
```

k=7

1
2
4
8
11
4
7
2

k=3

1
2
4
7
2
4
8
1
1

- ✓ zachowuje uporządkowanie częściowe
- ✓ pesymistyczna złożoność $O(N^{1.2})$

Sortowanie Shella

```
def ShellSort (A, n):
    k = 2^⌊log2 n⌋ - 1
    while k >= 1:
        for i in range(k,n):
            x = A[i]
            j = i
            while j >= k and x < A[j-k]:
                A[j] = A[j-k]
                j = j-k
            A[j] = x
        k = (k+1)/2 - 1
```

Sortowanie szybkie

Zaawansowane sortowanie przez zamianę elementów

1. Podziel tablicę A[p..r] na dwie A_1[p..q] i A_2[q+1..r], takie, że każdy element należący do A_1 jest mniejszy równy niż dowolny element A_2
2. Posortuj A_1 i A_2

Quicksort - implementacja

```
def QuickSort(A, p, r):
    if (p < r) :
        q = Partition(A, p, r)
        QuickSort (A, p, q)
        QuickSort (A, q+1, r)

QuickSort(Tablica, 0, N-1)
```

Partition

1. Wybieramy element x (granicę podziału)
2. Idąc od poczatku oprzedziału szukamy pierwszego elementu większego lub równego x.
3. Idąc od góry przedziału szukamy ostatniego elementu mniejszego lub równego x
4. Zamieniamy znalezione elementy.
5. Kończymy gdy poszukiwania dotrą to tego samego elementu

1. Warunki brzegowe:
   co będzie gdy wybierzemy min/max w przedziale?

Partition - implementacja

```
def Partition(A, l, r):
    x = A[l]
    l_m = l-1
    r_m = r+1
    while True:
        while True:
            l_m = l_m +1
            if A[l_m] >= x : break
        while True:
            r_m = r_m -1
            if A[r_m] <= x : break
        if l_m < r_m :
            A[l_m],A[r_m] = A[r_m],A[l_m]
        else:
            return r_m
```

Partition - implementacja

```
def Partition(A, p, r):
    m = p
    for i in range(p+1,r+1):
       if A[i]<tab[p]:
          m=m+1
          A[m],A[i] = A[i],A[m]
    A[p],A[m] = A[m],A[p]
    return m-1


def QuickSort(A, p, r):
    if (p < r) :
       q = Partition(A, p, r)
       QuickSort (A, p, q-1)
       QuickSort (A, q+1, r)

QuickSort(A, 0, MAX-1)
```

Randomized partition

- ✓ **Zapobiega umyślnemu wygenerowaniu najgorszego przypadku danych (ale dalej możliwe jest wystąpienie takiego przypadku)**

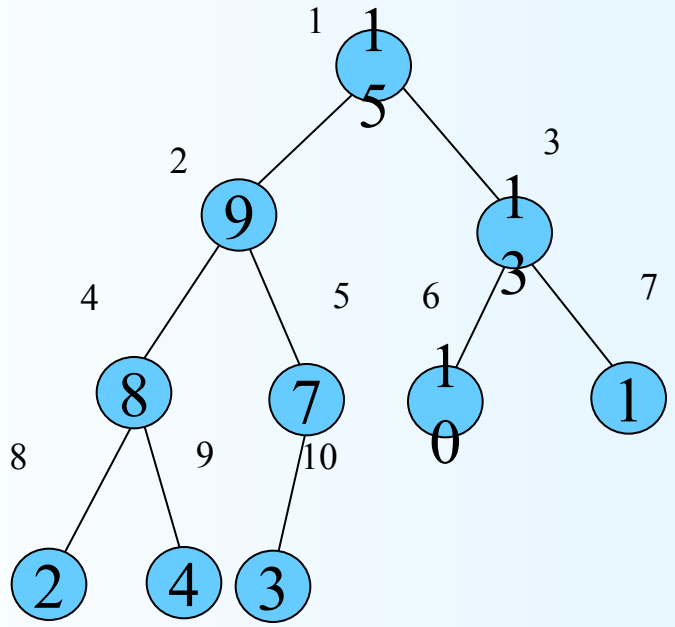
```
def RandomPartition(A, l, r):
    i = Random(l, r)
    A[i],A[l] = A[l],A[i]
    return Partition(A, l, r)
```

Quicksort - właściwości

- ✓ średni czas – O(N log N)
- ✓ pesymistyczny czas O(N^2)
- ✓ prosty algorytm – niewielki narzut
- ✓ najgorszy przypadek:
  Partition za każdym razem tworzy obszary
  o rozmiarach N-1 i 1
- ✓ Randomized Partition – brak możliwości podania najgorszego przypadku
- ✓ Aby zmniejszyć ew. zajętość stosu (w najgorszym przypadku ~N) można wybierać do sortowania najpierw mniejszy – a potem większy z przedziałów (co najwyżej log N)
- ✓ Zachowanie uporządkowania częściowego zależy od implementacji funkcji Partition

Kopiec

Każdy element jest nie mniejszy (większy) niż jego dzieci




```
A[Parent(i)] >= A[i]
```

15	9	13	8	7	10	1	2	4	3

```
def Parent(i): return i//2
def Left(i): return i*2
def Right(i): return i*2+1
```

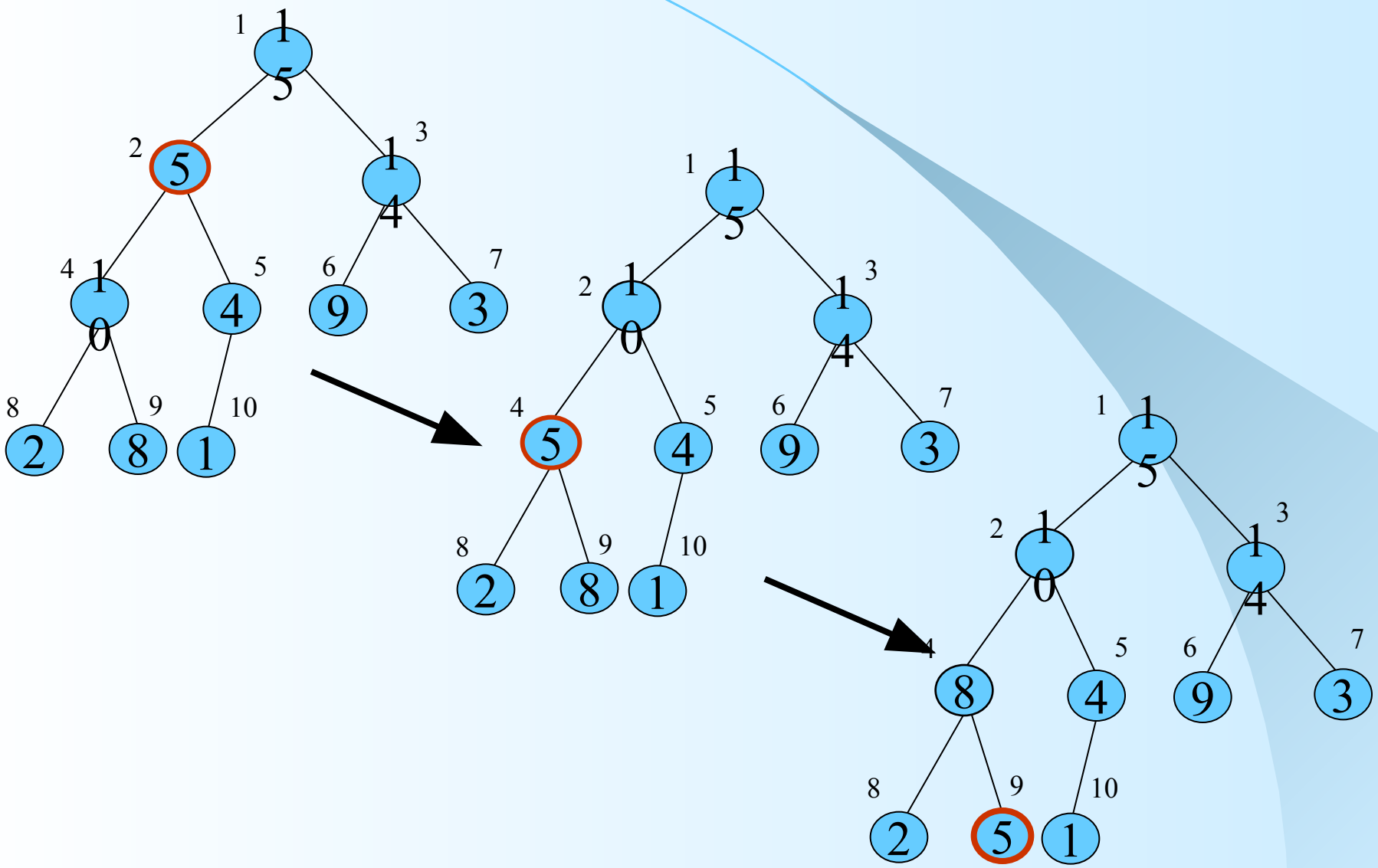
UWAGA index = 1..size

Przywracanie własności kopca

Z: właściwość kopca może nie być spełniona tylko dla korzenia

```
1) rozpocznij od korzenia (W=Root)
2) if jest spełniona właściwość kopca dla W i synów:
       koniec
   else
       zamień W z większym z synów W
       powtórz 2) dla nowego W
```

Przywracanie wł. kopca



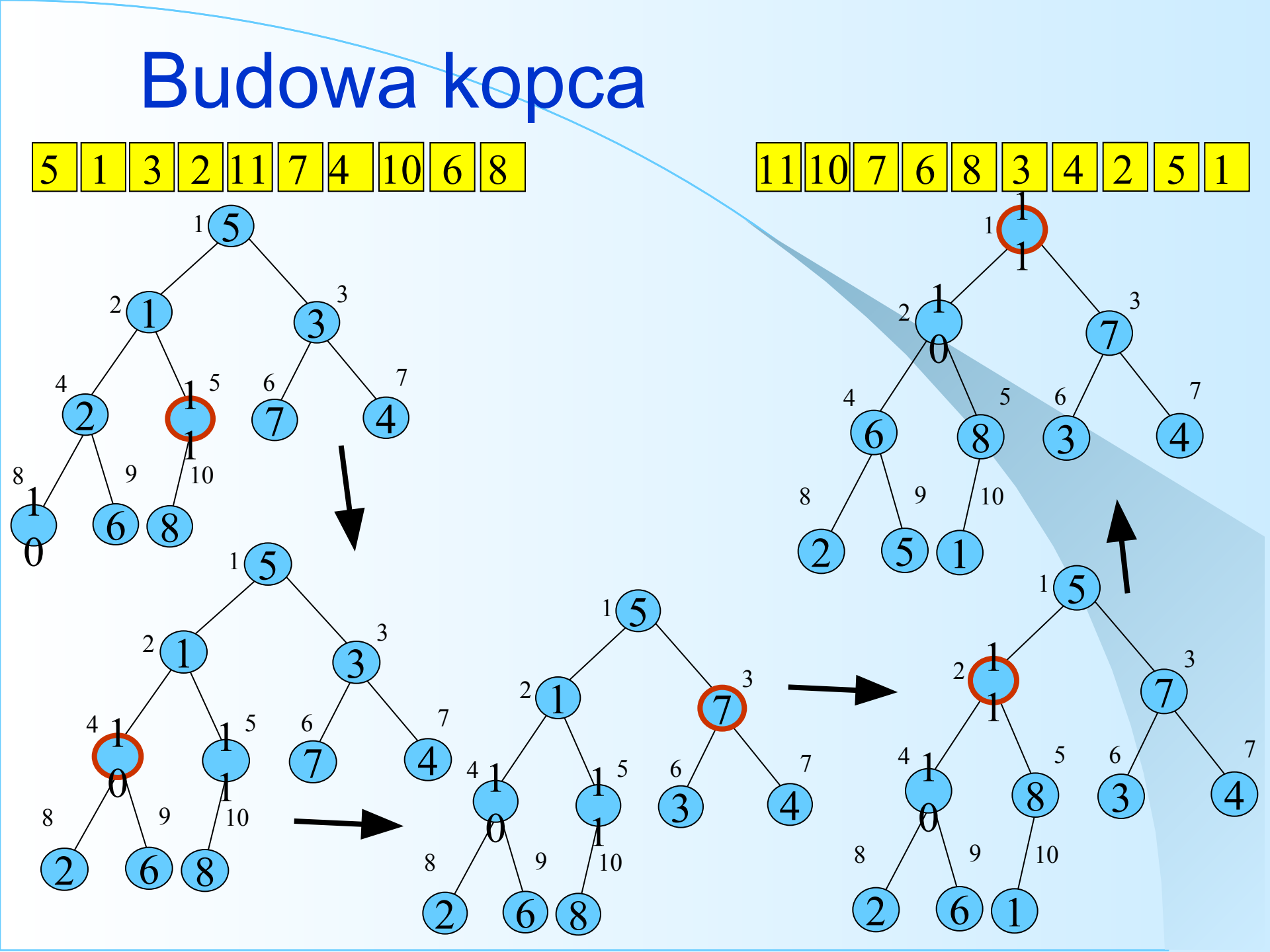
Przywracanie własności kopca

```
def Heapify(A, i, size):
    L = Left(i)
    R = Right(i)
    if L<=size and A[L-1]>A[i-1]:
        maxps = L
    else:
        maxps = i
    if R<=size and A[R-1]>A[maxps-1]:
        maxps = R
    if maxps != i:
        A[i-1],A[maxps-1] = A[maxps-1],A[i-1]
        Heapify(A, maxps, size)
```

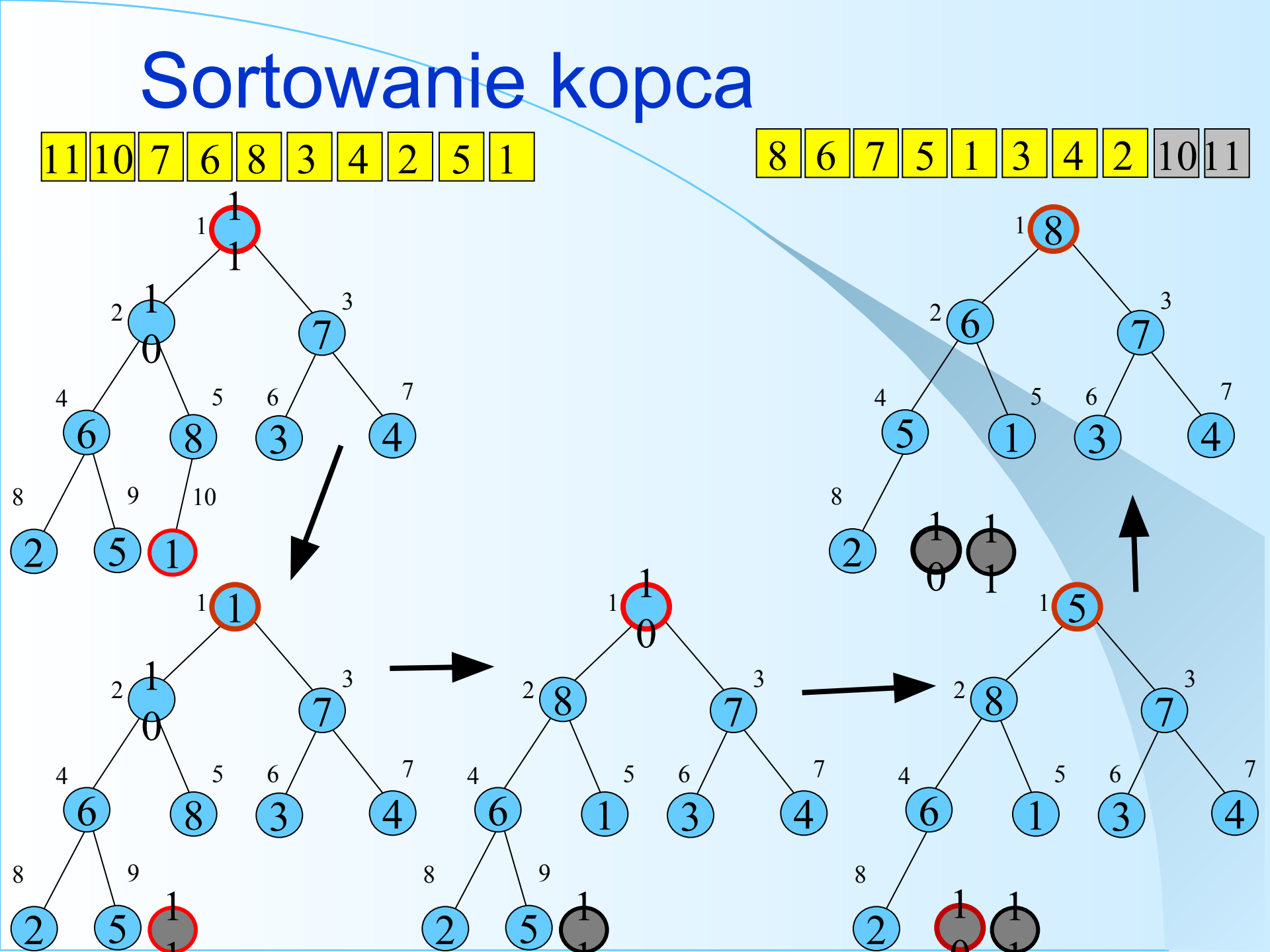
Sortowanie kopcowe

- ✓ Tworzenie kopca z całej tablicy
  - Przywracamy własność kopca dla coraz większych kopców
- ✓ Dla kolejnych (coraz mniejszych) kopców
  - **`Wybór największego elementu (korzeń kopca),`**
  - **`Zamiana z ostatnim,`**
  - **`Zmniejszenie rozmiarów kopca,`**
  - **`Przywrócenie własności kopca.`**

Budowa kopca



Sortowanie kopca



Sortowanie kopcowe

```
def BuildHeap(A, size):
    for i in range(Parent(size),0,-1):
        Heapify(A, i, size)

def HeapSort(A, size):
    BuildHeap(A, size)
    for i in range(size,1,-1) :
        A[i-1], A[0] = A[0], A[i-1]
        Heapify(A, 1, i-1)
```

Sortowanie przez scalanie

- ✓ nie jest wymagany dostęp do wszystkich elementów np. dla plików, list

podział ciągu A -> B,C

scalenie B+C -> ciąg par

ciąg par - > B,C

scalenie B+C -> ciąg czwórek

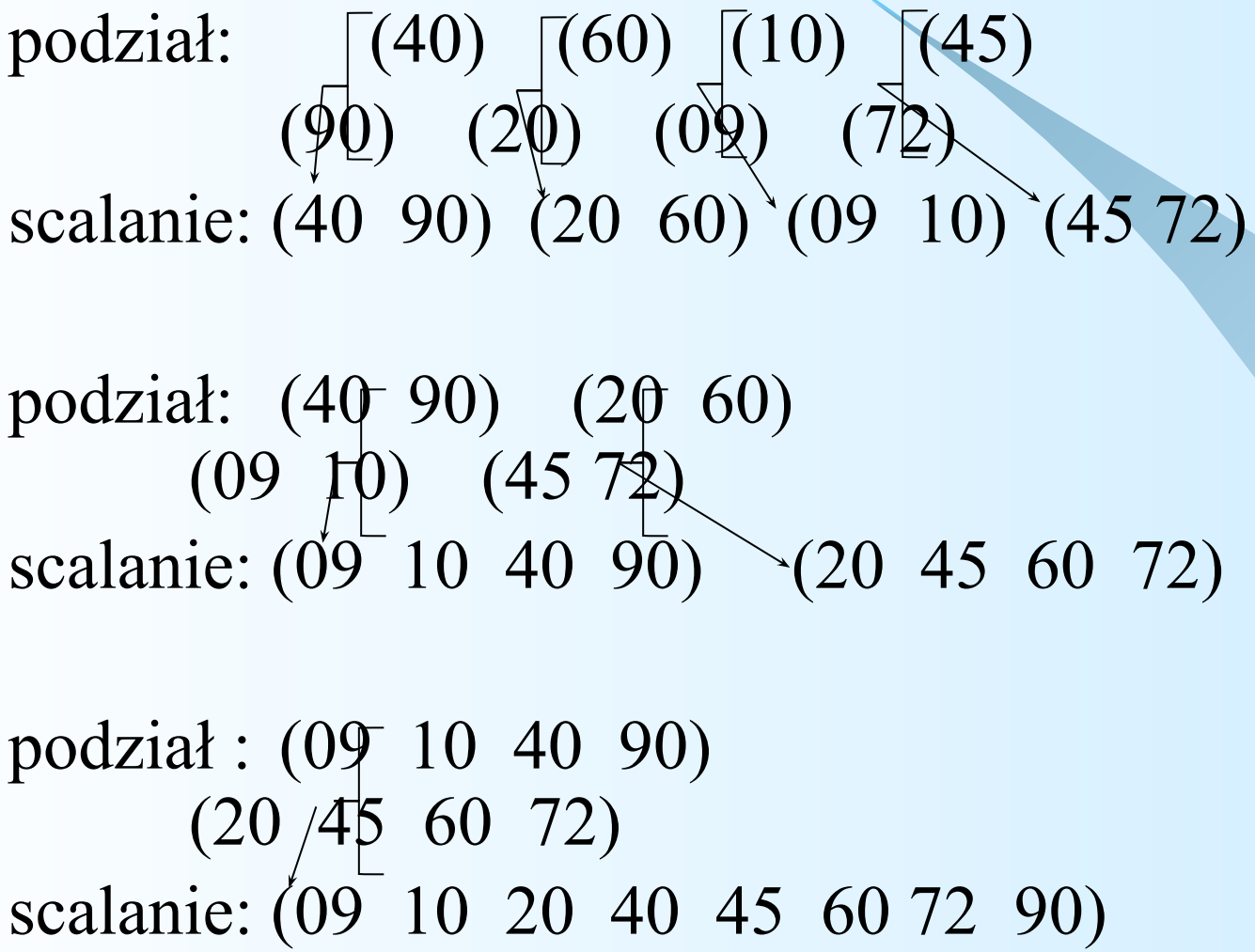
ciąg czwórek - > B,C

scalenie B+C -> ciąg ósemek

….

Sortowanie przez scalanie

we: 40, 60, 10, 45, 90, 20, 09, 72



Kolejka priorytetowa

Przykład: zadania do wykonania, zamówienia

Wymagane operacje:

- **Dodaj element o priorytecie X**
- **Pobierz element o najwyższym priorytecie**
- **Przeczytaj element o najwyższym priorytecie (bez usuwania)**

Przy realizacji kolejki priorytetowej przy pomocy kopca

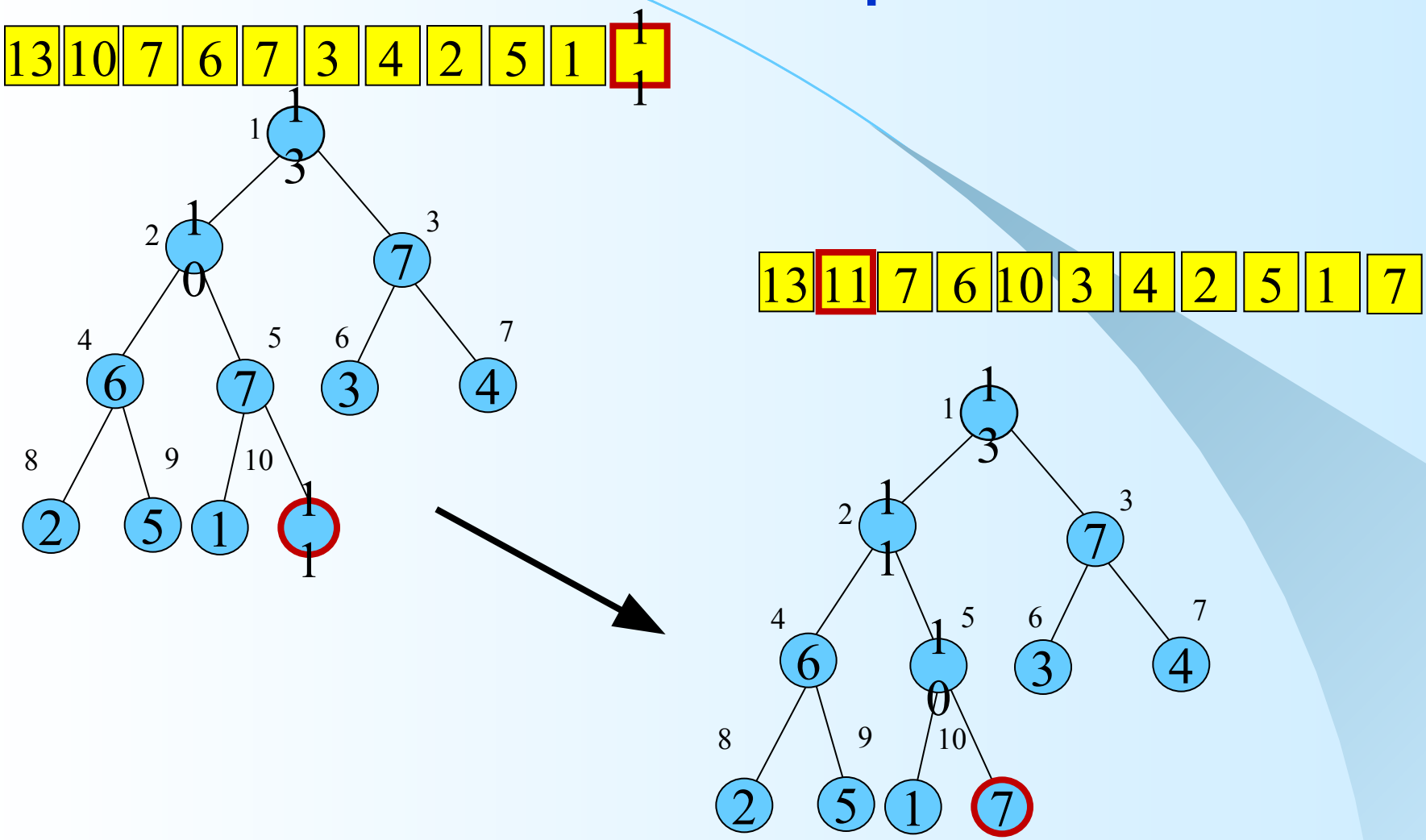
Zapis: **`dodawanie elementów do kopca`**

Odczyt: **`zwróć A[0] i przebuduj kopiec`**

Dodawanie do kopca

1) dodaj nowy element w na koniec kopca
2) if jest spełniona właściwość kopca dla *W*\
   i ojca *W*:
      koniec
   else:
      zamień *W* z ojcem *W*
      powtórz 2) dla *W* i nowego ojca *W*

Dodawanie do kopca



Usunięcie maksimum/minimum

Spostrzeżenie: *Maximum (Minimum) jest korzeń kopca*

1) **Usuwamy korzeń**
2) **Na miejsce korzenia wstawiamy ostatni element kopca**
3) **Wykonujemy Heapify(Root)**

Kolejka priorytetowa - realizacja

```
def HeapInsert(A, size, newElement):
    if size>=MAXSIZE:
        ERROR „przepelnienie kopca”
    else:
        size=size+1
    i = size
    while i>1 and A[Parent(i)-1]<newElement :
       A[i-1] = A[Parent(i)-1]
       i = Parent(i)
     A[i-1] = newElement
    return size
```

Kolejka priorytetowa - realizacja

```
def HeapGetMax(A, size):
    if size<1:
        ERROR "kopiec pusty"
    else:
        element max = A[0]
        A[0] = A[size-1]
        size = size-1
        Heapify(A, size)
        return max, size
```

Sortowanie liniowe?

Niekiedy dodatkowe informacje o kluczach lub charakterze elementów umożliwiają sortowanie w czasie liniowym

Sortowanie przez zliczanie

- ✓ Czas liniowy
- ✓ Założenie: wszystkie elementy są liczbami z przedziału 0..maxN
- ✓ maxN jest akceptowalnie duże (tablica mieści się w pamięci komputera)

```
1) zlicz (w tablicy A) wystąpienia poszczególnych\
    elementów
2) przygotuj tablice B z początkami sekwencji
   B[] = 0
   B[i] = B[i-1] + A[i-1]
3) przepisz elementy do nowej tablicy C na pozycję
    C[B[element.klucz]++] = element
```

Sortowanie przez zliczanie

```
def CountingSort(A, B, maxN, size):
    for i in range(0, maxN+1):
       C[i] = 0
    for i in range(0, size):
       C[A[i]] = C[A[i]]+1
    # C[i] zawiera liczbę elementów równych i
    for i in range(1, maxN+1):
       C[i] = C[i] + C[i-1]
    # C[i] zawiera liczbę elementów <= i
    for i in range(1,size):
        B[C[A[i]]] = A[i]
        C[A[i]] = C[A[i]]+1
```

Sortowanie przez zliczanie

We	I	A[i]	B[i]	Wy
A_3	0	0	0	E_1
B_2	1	2	0+1+1	I_1
C_3	2	2	2+1+1	B_2
D_5	3	2	4+1+1	F_2
E_1	4	2	6+1+1	A_3
F_2	5	1	8+1	C_3
G_4	6	0	9	G_4
H_4	7	0	9	H_4
I_1	8	0	9	D_5

Sortowanie kubełkowe

- ✓ Czas liniowy
- ✓ Założenie: wszystkie klucze są liczbami (rzeczywistymi) z przedziału x_0 .. x_1
- ✓ Równomierny rozkład wartości kluczy w przedziale

1) ***podziel dziedzinę kluczy na n równych przedziałów długości p = (x_1 - x_0) / n, gdzie i-ty przedział < x_0+(i-1)*p , x_0+i*p), dla i = 1..n, a ostatni (pierwszy) przedział jest obustronnie domknięty***
2) ***przepisz n elementów do n list (kubełków), tak aby elementy na i-tej miały klucze z należące do i-tego przedziału***
3) ***Posortuj elementy w kubełkach (dowolnym alg.)***
4) ***Wypisz zawartość poszczególnych kubełków od i = 1 do n***

Wyszukiwanie maksimum

```
def Minimum(A, size):
    m = A[0]
    for i in range(1,size):
        if (m>A[i]): m=A[i]
    return m
```

Wyszukiwanie min i maks

```
def MinMax(A, size):
    mi = A[0]
    ma = A[0]
    for i in range(2,size+2,2) :
        if A[i]<A[i-1]:
            mi = min (A[i], mi)
            ma = max (A[i-1], ma)
        else:
            mi = min (A[i-1], mi)
            ma = max (A[i], ma)
    return mi,ma
```

Zwracanie kilku wartości C/C++

```
struct MM {
    ELEMENT min, max;
    MM(ELEMENT emin, ELEMENT emax) {// konstruktor
        Min = emin;
        Max = emax;
    }
};

MM Minmax(ELEMENT A[], INDEX size)
{
    .....
    return MM(min, max);
    //bez konstruktora trzeba użyć zmiennej
     //chwilowej
}
```

Wyszukiwanie i-tej statystyki

- ✓ Oczekiwany czas działania = O(n)
- ✓ Pesymistyczny czas działania = $O(n^2)$

```
def RandomizedSelect(A, i, l, r):
    if l == r: return A[p]
    q = RandomizedPartition(A, l, r)
    k = q-l+1
    if i<=k: return RandomizedSelect (A, p, q, i)
    else: return RandomizedSelect (A, q+1, r, i-k)
```

Wyszukiwanie i-tej statystyki

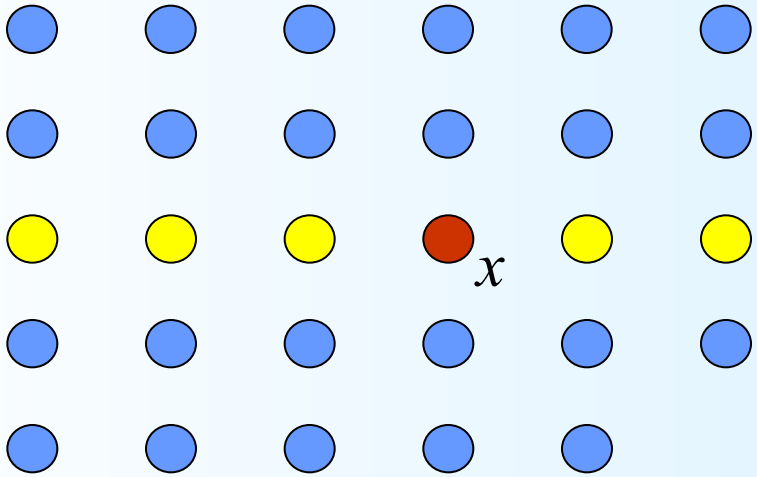
- ✓ Pesymistyczny czas działania = O(n)

INDEX PartitionX(ELEMENT A[], INDEX p, INDEX r, ELEMENT e)

ELEMENT Select(ELEMENT A[], INDEX p, INDEX r)

1. *Podziel elementy p do q na [n/5] grup po 5 elementów O(n)*
2. *Wyznacz medianę każdej z [n/5] grup, sortując je (czas stały). Jeśli (ostatnia) ma parzystą liczbę elementów wybierz większą z median O(n)*
3. *Wywołaj rekurencyjnie x = Select(...) na zbiorze median wyznaczonym w kroku 2 - b.d.*
4. *Podziel tablicę A względem mediany median tj. x i zmodyfikowanej PartitionX. Niech wynik będzie k - O(n)*
5. *Wywołaj rekurencyjnie Select(A, p, k) jeżeli i<=k lub Select(A, k+1, r) w przeciwnym przypadku - b.d.*

Wyszukiwanie i-tej statystyki



Uwagi:

- Co najmniej połowa median jest większa równa x
- Co najmniej $\lceil n/5 \rceil$ grup zawiera 3 elementy większe równe x, przy czym jedna z nich zawiera x, a druga może być niepełna
- Stąd liczba elementów większa (mniejsza) od x wynosi co najmniej 3 $(\lceil 1/2 \lceil n/5 \rceil \rceil - 2) \geq 3n/10 - 6$