1 Definicja

Definicja 1 *Niech T będzie drzewem binarnym o wysokości d, którego wierzchołki zawierają klucze z liniowo uporządkowanego zbioru. Drzewo T nazywamy* kopcem *iff $d \leq 1$ lub T spełnia następujące warunki:*

(1) Struktura drzewa

- *wszystkie jego liście znajdują się na głębokości d lub $d-1$;*
- *wszystkie liście z poziomu $d-1$ leżą na prawo od wszystkich wierzchołków wewnętrznych z tego poziomu;*
- *położony najbardziej na prawo wierzchołek wewnętrzny z poziomu $d-1$ jest jedynym wierzchołkiem wewnętrznym w T, który może mieć jednego syna (co implikuje, że pozostałe wierzchołki wewnętrzne mają po dwóch synów);*

(2) Uporządkowanie

- *klucz w każdym wierzchołku wewnętrznym jest nie mniejszy od kluczy w jego potomkach.*

Warunki określające strukturę kopca mogą się wydać nieco skomplikowane. W rzeczywistości mówią one, że dobrą strukturę mają drzewa binarne powstałe przez dopisywanie do początkowo pustego drzewa wierzchołków do kolejnych poziomów drzewa, zapełniając każdy poziom od lewej strony do prawej strony .

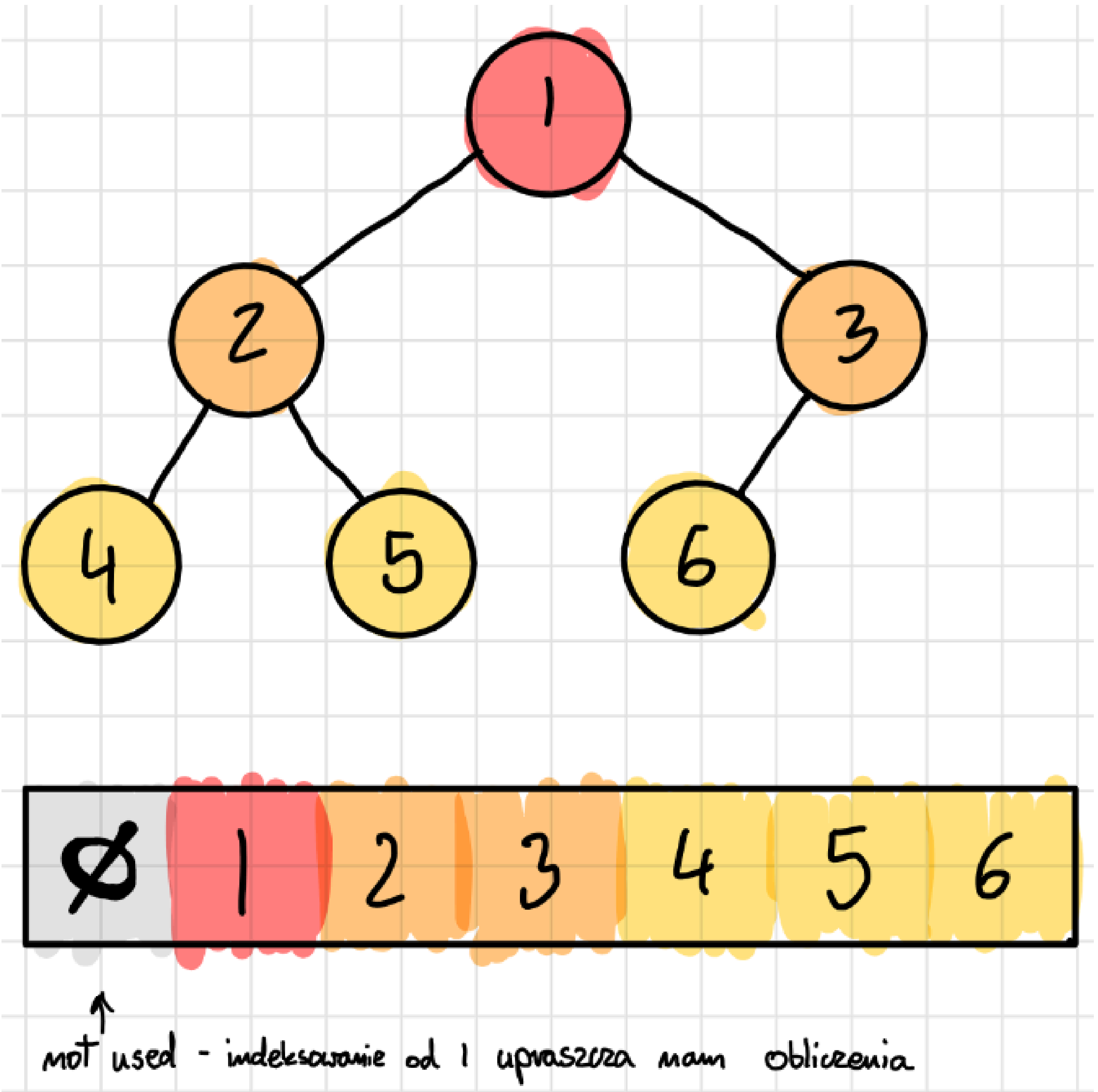
2 Implementacja kopców

Kopce w bardzo efektywny sposób mogą być pamiętane w tablicach. Do pamiętania kopca n-elementowego używamy n-elementowej tablicy K:

- korzeń kopca pamiętany jest w $K[1]$,
- lewy syn korzenia pamiętany jest w $K[2]$, prawy syn korzenia - w $K[3]$, itd ... .

Uogólniając: wierzchołki z poziomu k-tego pamiętane są kolejno od lewej do prawej w $K[2^k], K[2^k+1], \ldots, K[2^{k+1}-1]$.

Fakt 1 *Ojciec wierzchołka pamiętanego w $K[i]$ znajduje się w $K[i$ **div** $2]$ zaś jego dzieci (o ile istnieją) w $K[2i]$ i $K[2i+1]$.*



Ważniejsze procedury

procedure *buduj-kopiec* $(K[1..n])$
  for $i \leftarrow (n$ **div** $2)$ **downto** 1 *przesuń-niżej* (K, i)

procedure *przesuń-niżej* $(K[1..n], i)$
  $k \leftarrow i$
  repeat
    $j \leftarrow k$
    if $2j \leq n$ **and** $K[2j] > K[k]$ **then** $k \leftarrow 2j$
    if $2j < n$ **and** $K[2j+1] > K[k]$ **then** $k \leftarrow 2j+1$
    $K[j] \leftrightarrow K[k]$
  until $j = k$

```
def build_heap_fast(self):  # O(n)
    for i in range(len(self.heap) // 2, 0, -1):
        self.move_down(i)
```

```
def move_down(self, i):
    iterator = i
    condition = True

    while condition:
        max_child = iterator
        left_child = 2 * iterator
        right_child = 2 * iterator + 1

        if left_child < len(self.heap) and self.heap[left_child] < self.heap[max_child]:
            max_child = left_child
        if right_child < len(self.heap) and self.heap[right_child] < self.heap[max_child]:
            max_child = right_child

        if max_child != iterator:
            self.change_elements(iterator, max_child)
            iterator = max_child
        else:
            condition = False
```

procedure *przesuń-wyżej* $(K[1..n], i)$
  $k \leftarrow i$
  repeat
    $j \leftarrow k$
    if $j > 1$ **and** $K[j$ **div** $2] < K[k]$ **then** $k \leftarrow j$ **div** 2
    $K[j] \leftrightarrow K[k]$
  until $j = k$

```
def move_up(self, i):
    iterator = i
    condition = True
    while condition:
        parent = iterator // 2
        if parent > 0 and self.heap[parent] > self.heap[iterator]:
            self.change_elements(iterator, parent)
            iterator = parent
        else:
            condition = False
```

3 Zastosowania kopców

3.1 Heapsort - sortowanie przy użyciu kopca

```
procedure heapsort(K[1..n])
  buduj − kopiec(K)
  for i ← n  step −1  to 2  do
    K[1] ↔ K[i]
    przesuń-niżej (K[1..i − 1], 1)
  return K
```

Twierdzenie 2 *Algorytm heapsort działa w czasie* $O(n \log n)$.

3.1.1 Przyspieszenie *heapsortu*

Po usunięciu maksimum na szczycie kopca powstaje dziura, w którą *heapsort* wstawia element z dołu kopca. Element taki jest, z dużym prawdopodobieństwem, mały i zostanie przez procedurę *przesuń-niżej* zsunięty z powrotem nisko. Przesuwając go o jeden poziom w dół *przesuń-niżej* wykonuje dwa porównania. Tak więc z dużym prawdopodobieństwem potrzeba będzie 2·`wysokość kopca` porównań na przywrócenie własności kopca.

Można postępować nieco oszczędniej. Otóż można najpierw przesunąć dziurę na dół kopca, następnie wstawić w nią ostatni element kopca i używając procedury *przesuń-wyżej* znaleźć dla niego odpowiednie miejsce w kopcu. Oszczędność wynika z tego, że na przesunięcie dziury o jedno miejsce w dół potrzeba tylko jednego porównania oraz z tego, że w średnim przypadku *przesuń-wyżej* będzie przesuwać element o nie więcej niż 2 poziomy w górę.

3.2 Kolejka priorytetowa

Kolejka priorytetowa jest strukturą danych przeznaczoną do pamiętania zbioru S (elementów z jakiegoś uporządkowanego uniwersum) i wykonywania operacji wstawiania elementów do S oraz znajdowania i usuwania największego elementu z S.

Wprost idealnie do implementacji kolejek priorytetowych nadają się kopce.

3.2.1 Procedury realizujące operacje kolejki priorytetowej

```
function find − max(K[1..n])
  return K[1]

procedure delete − max(K[1..n])
  K[1] ← K[n]
  przesuń-niżej (K[1..n − 1], 1)

procedure insert − node(K[1..n], v)
  K[n + 1] ← v
  przesuń-wyżej (K[1..n + 1], n + 1)
```

Treść

Jaka jest złożoność poniższej funkcji tworzenia kopca w tablicy K:

```
procedure buduj-kopiec(K[1...n])
    for i <- 2 to n: przesun-wyzej(K,i)
```

gdzie przesun-wyzej przywraca porządek kopcowy w kopcu zapan

▼ Rozwiązanie

$\sum_{n=1}^{\infty} \log i = \log n! <= \log n^n = n \log n$

Treść

Napisz w pseudokodzie szybką procedurę budowy kopca jaki ma czas działania ?

▼ Rozwiązanie

O(n)

```
buduj_szybko():
  dla i od n/2 do 1:
    przesuń_niżej(i)
```

Treść

W kopcu umieszczono n kluczy 1, 2, 3,..., n.

W korzeniu znajduje się 1.

Na jakiej maksymalnie wysokości znajduje się klucz k? 1<=k<=n

---

▼ Rozwiązanie

k, jeśli k < logn

wpp. logn

Treść

Opisz algorytm tworzenia kopca, którego złożoność określa:

$$T(n) = \sum_{i=1}^{n} \log i$$

Czy to najszybszy algorytm tworzenia kopca? Odpowiedź uzasadnij.

▼ Rozwiązanie

Ta suma przedstawia `n` operacji insert zaczynając na pustym kopcu. Jest to algorytm budujący w czasie `O(nlogn)`.

Istnieje szybszy algorytm, w którym na początkowej tablicy wartości uruchamiamy na elementach od `podłoga(n/2)` do 1 procedurę `przesuń_niżej`. Ten algorytm ma złożoność `O(n)`.

Treść

Udowodnij, że przynajmniej jedna z operacji min, deletemin lub insert wykonywanych na kolejce priorytetowej wymaga w najgorszym przypadku czasu Ω(logn)

▼ Rozwiązanie

Trzeba pokazać, że jeśli moglibyśmy zrobić wszystkie operacje poniżej logn, to mielibyśmy złożoność sortowania nlogn, co jest sprzeczne z dolną granicą dla tego problemu w comparison model.

Treść

W której wersji deletemin na kopcu spodziewamy się wykonać mniej porównań i dlaczego? (usuwany element zastępujemy skrajnie prawym liściem z ostatniego poziomu vs przesuwanie dziury na dół).

▼ Rozwiązanie

Pierwsze rozwiązanie wykonuje 2logn operacji((porównanie dwóch synów oraz porównanie z mniejszym synem) logn razy)

Przesuwanie dziury w dół wykona >= logn i z dużym prawdopodobieństwem będzie to <2logn (porównanie dwóch synów logn razy + naprawienie kopca)