Algorytmy i struktury danych - Kopce binarne. Kolejki priorytetowe

Marcin Żurowski

21 maja 2026

Plan zajęć

Kopiec binarny

- **Kopiec binarny** – tablicowa struktura danych, którą interpretujemy jako prawie pełne drzewo binarne: drzewo pełne na wszystkich poziomach za wyjątkiem być może najniższego, który jest wypełniony od lewej strony do pewnego miejsca.
- Każdemu elementowi x tablicy A reprezentującej kopiec odpowiada w drzewie binarnym węzeł o kluczu x.

Kopiec binarny

- **Kopiec binarny** – tablicowa struktura danych, którą interpretujemy jako prawie pełne drzewo binarne: drzewo pełne na wszystkich poziomach za wyjątkiem być może najniższego, który jest wypełniony od lewej strony do pewnego miejsca.
- Każdemu elementowi x tablicy A reprezentującej kopiec odpowiada w drzewie binarnym węzeł o kluczu x.

Kopiec binarny

Tablica A reprezentująca kopiec posiada dwa atrybuty:

- $A.length$ – długość tablicy A,
- $A.heapsize$ – liczba elementów kopca przechowywanego w tablicy A.

Do kopca należą elementy $A[1], A[2], \ldots, A[A.heapsize]$.

Kopiec binarny

Tablica A reprezentująca kopiec posiada dwa atrybuty:

- $A.length$ – długość tablicy A,
- $A.heapsize$ – liczba elementów kopca przechowywanego w tablicy A.

Do kopca należą elementy $A[1], A[2], \ldots, A[A.heapsize]$.

Kopiec binarny

Tablica A reprezentująca kopiec posiada dwa atrybuty:

- $A.length$ – długość tablicy A,
- $A.heapsize$ – liczba elementów kopca przechowywanego w tablicy A.

Do kopca należą elementy $A[1], A[2], \ldots, A[A.heapsize]$.

Kopiec binarny

Tablica A reprezentująca kopiec posiada dwa atrybuty:

- $A.length$ – długość tablicy A,
- $A.heapsize$ – liczba elementów kopca przechowywanego w tablicy A.

Do kopca należą elementy $A[1], A[2], \ldots, A[A.heapsize]$.

Kopiec binarny

Kopiec jako drzewo binarne:

- Korzeniem drzewa jest węzeł odpowiadający $A[1]$.
- Dla zadanego indeksu i indeksy jego ojca $PARENT(i)$, lewego syna $LEFT(i)$ oraz prawego syna $RIGHT(i)$ obliczamy za pomocą następujących funkcji.

```
procedure PARENT(i)
 return i DIV 2
procedure LEFT(i)
 return 2 * i
procedure RIGHT(i)
 return 2 * i + 1
```

Kopiec binarny

Kopiec jako drzewo binarne:

- Korzeniem drzewa jest węzeł odpowiadający $A[1]$.
- Dla zadanego indeksu i indeksy jego ojca $PARENT(i)$, lewego syna $LEFT(i)$ oraz prawego syna $RIGHT(i)$ obliczamy za pomocą następujących funkcji.

```
procedure PARENT(i)
 return i DIV 2
procedure LEFT(i)
 return 2 * i
procedure RIGHT(i)
 return 2 * i + 1
```

Kopiec binarny

Kopiec jako drzewo binarne:

- Korzeniem drzewa jest węzeł odpowiadający $A[1]$.
- Dla zadanego indeksu i indeksy jego ojca $PARENT(i)$, lewego syna $LEFT(i)$ oraz prawego syna $RIGHT(i)$ obliczamy za pomocą następujących funkcji.

```
procedure PARENT(i)
 return i DIV 2
procedure LEFT(i)
 return 2 * i
procedure RIGHT(i)
 return 2 * i + 1
```

Kopiec binarny

Kopiec jako drzewo binarne:

- Korzeniem drzewa jest węzeł odpowiadający $A[1]$.
- Dla zadanego indeksu i indeksy jego ojca $PARENT(i)$, lewego syna $LEFT(i)$ oraz prawego syna $RIGHT(i)$ obliczamy za pomocą następujących funkcji.

```
procedure PARENT(i)
 return i DIV 2
procedure LEFT(i)
 return 2 * i
procedure RIGHT(i)
 return 2 * i + 1
```

Kopiec binarny

Kopiec jako drzewo binarne:

- Korzeniem drzewa jest węzeł odpowiadający $A[1]$.
- Dla zadanego indeksu i indeksy jego ojca $PARENT(i)$, lewego syna $LEFT(i)$ oraz prawego syna $RIGHT(i)$ obliczamy za pomocą następujących funkcji.

```
procedure PARENT(i)
 return i DIV 2
procedure LEFT(i)
 return 2 * i
procedure RIGHT(i)
 return 2 * i + 1
```

Kopiec binarny

Kopiec jako drzewo binarne:

- Korzeniem drzewa jest węzeł odpowiadający $A[1]$.
- Dla zadanego indeksu i indeksy jego ojca $PARENT(i)$, lewego syna $LEFT(i)$ oraz prawego syna $RIGHT(i)$ obliczamy za pomocą następujących funkcji.

```
procedure PARENT(i)
 return i DIV 2
procedure LEFT(i)
 return 2 * i
procedure RIGHT(i)
 return 2 * i + 1
```

Kopiec binarny

Kopiec jako drzewo binarne:

- Korzeniem drzewa jest węzeł odpowiadający $A[1]$.
- Dla zadanego indeksu i indeksy jego ojca $PARENT(i)$, lewego syna $LEFT(i)$ oraz prawego syna $RIGHT(i)$ obliczamy za pomocą następujących funkcji.

```
procedure PARENT(i)
 return i DIV 2
procedure LEFT(i)
 return 2 * i
procedure RIGHT(i)
 return 2 * i + 1
```

Kopiec binarny

Wyróżniamy kopiec typu *max* i kopiec typu *min*.

- Własność kopca typu max: $A[PARENT(i)] \geq A[i]$.
- Największy element całego kopca typu max umieszczony jest w korzeniu, a największy element dowolnego poddrzewa kopca typu max znajduje się w korzeniu tego poddrzewa.
- Własność kopca typu min: $A[PARENT(i)] \leq A[i]$.
- Najmniejszy element całego kopca typu min umieszczony jest w korzeniu, a najmniejszy element dowolnego poddrzewa kopca typu min znajduje się w korzeniu tego poddrzewa.

Kopiec binarny

Wyróżniamy kopiec typu *max* i kopiec typu *min*.

- Własność kopca typu max: $A[PARENT(i)] \geq A[i]$.
- Największy element całego kopca typu max umieszczony jest w korzeniu, a największy element dowolnego poddrzewa kopca typu max znajduje się w korzeniu tego poddrzewa.
- Własność kopca typu min: $A[PARENT(i)] \leq A[i]$.
- Najmniejszy element całego kopca typu min umieszczony jest w korzeniu, a najmniejszy element dowolnego poddrzewa kopca typu min znajduje się w korzeniu tego poddrzewa.

Kopiec binarny

Wyróżniamy kopiec typu *max* i kopiec typu *min*.

- Własność kopca typu max: $A[PARENT(i)] \geq A[i]$.
- Największy element całego kopca typu max umieszczony jest w korzeniu, a największy element dowolnego poddrzewa kopca typu max znajduje się w korzeniu tego poddrzewa.
- Własność kopca typu min: $A[PARENT(i)] \leq A[i]$.
- Najmniejszy element całego kopca typu min umieszczony jest w korzeniu, a najmniejszy element dowolnego poddrzewa kopca typu min znajduje się w korzeniu tego poddrzewa.

Kopiec binarny

Wyróżniamy kopiec typu *max* i kopiec typu *min*.

- Własność kopca typu max: $A[PARENT(i)] \geq A[i]$.
- Największy element całego kopca typu max umieszczony jest w korzeniu, a największy element dowolnego poddrzewa kopca typu max znajduje się w korzeniu tego poddrzewa.
- Własność kopca typu min: $A[PARENT(i)] \leq A[i]$.
- Najmniejszy element całego kopca typu min umieszczony jest w korzeniu, a najmniejszy element dowolnego poddrzewa kopca typu min znajduje się w korzeniu tego poddrzewa.

Kopiec binarny

Kopiec jako drzewo binarne – własności:

- Kopiec o wysokości h ma co najmniej 2^h i co najwyżej $2^{h+1} - 1$ elementów.
- Wysokość kopca zawierającego n elementów jest równa $\lfloor \lg n \rfloor$.
- Liście kopca to węzły o indeksach $\lfloor \frac{n}{2} \rfloor + 1, \lfloor \frac{n}{2} \rfloor + 2, \ldots, n$
- Węzły wewnętrzne kopca mają indeksy $1, 2, \ldots, \lfloor \frac{n}{2} \rfloor$

Kopiec binarny

Kopiec jako drzewo binarne – własności:

- Kopiec o wysokości h ma co najmniej 2^h i co najwyżej $2^{h+1} - 1$ elementów.
- Wysokość kopca zawierającego n elementów jest równa $\lfloor \lg n \rfloor$.
- Liście kopca to węzły o indeksach $\lfloor \frac{n}{2} \rfloor + 1, \lfloor \frac{n}{2} \rfloor + 2, \ldots, n$
- Węzły wewnętrzne kopca mają indeksy $1, 2, \ldots, \lfloor \frac{n}{2} \rfloor$

Kopiec binarny

Kopiec jako drzewo binarne – własności:

- Kopiec o wysokości h ma co najmniej 2^h i co najwyżej $2^{h+1} - 1$ elementów.
- Wysokość kopca zawierającego n elementów jest równa $\lfloor \lg n \rfloor$.
- Liście kopca to węzły o indeksach $\lfloor \frac{n}{2} \rfloor + 1, \lfloor \frac{n}{2} \rfloor + 2, \ldots, n$
- Węzły wewnętrzne kopca mają indeksy $1, 2, \ldots, \lfloor \frac{n}{2} \rfloor$

Kopiec binarny

Kopiec jako drzewo binarne – własności:

- Kopiec o wysokości h ma co najmniej 2^h i co najwyżej $2^{h+1} - 1$ elementów.
- Wysokość kopca zawierającego n elementów jest równa $\lfloor \lg n \rfloor$.
- Liście kopca to węzły o indeksach $\lfloor \frac{n}{2} \rfloor + 1, \lfloor \frac{n}{2} \rfloor + 2, \ldots, n$
- Węzły wewnętrzne kopca mają indeksy $1, 2, \ldots, \lfloor \frac{n}{2} \rfloor$

Kopiec binarny

Przywracanie własności kopca typu max – założenia:

- Dana jest tablica A i indeks i.
- Drzewa binarne o korzeniach $LEFT(i)$ i $RIGHT(i)$ są kopcami typu max, ale element $A[i]$ może naruszać własność kopca typu max, tzn. być mniejszy od co najmniej jednego ze swoich synów.
- Przywrócenie własności kopca typu max polega na przestawieniu węzłów w poddrzewie zaczepionym w węźle i w taki sposób, aby stało się ono kopcem typu max.

Kopiec binarny

Przywracanie własności kopca typu max – założenia:

- Dana jest tablica A i indeks i.
- Drzewa binarne o korzeniach $LEFT(i)$ i $RIGHT(i)$ są kopcami typu max, ale element $A[i]$ może naruszać własność kopca typu max, tzn. być mniejszy od co najmniej jednego ze swoich synów.
- Przywrócenie własności kopca typu max polega na przestawieniu węzłów w poddrzewie zaczepionym w węźle i w taki sposób, aby stało się ono kopcem typu max.

Kopiec binarny

Przywracanie własności kopca typu max – założenia:

- Dana jest tablica A i indeks i.
- Drzewa binarne o korzeniach $LEFT(i)$ i $RIGHT(i)$ są kopcami typu max, ale element $A[i]$ może naruszać własność kopca typu max, tzn. być mniejszy od co najmniej jednego ze swoich synów.
- Przywrócenie własności kopca typu max polega na przestawieniu węzłów w poddrzewie zaczepionym w węźle i w taki sposób, aby stało się ono kopcem typu max.

Kopiec binarny

Przywracanie własności kopca typu max – idea:

- Porównujemy elementy $A[i]$, $A[LEFT(i)]$ oraz $A[RIGHT(i)]$. Indeks największego z nich zapamiętujemy w zmiennej L.
- Jeśli $L = i$, to poddrzewo zaczepione w węźle i jest kopcem typu max, koniec algorytmu.
- Jeśli $L \neq i$, to zamieniamy elementy $A[i]$ i $A[L]$. Element $A[i]$ jest teraz nie mniejszy niż jego synowie, ale element $A[L]$ może być mniejszy od swoich synów. Wywołujemy rekurencyjnie przywracanie własności kopca dla tego węzła.
- W każdym kroku algorytmu schodzimy na niższy poziom kopca. Zatem przywracanie własności kopca na węźle o wysokości h zajmuje czas $O(h)$. Stąd czas przywracania własności kopca dla dowolnego ustalonego węzła w n-elementowym kopcu wynosi $O(\lg n)$.

Kopiec binarny

Przywracanie własności kopca typu max – idea:

- Porównujemy elementy $A[i]$, $A[LEFT(i)]$ oraz $A[RIGHT(i)]$. Indeks największego z nich zapamiętujemy w zmiennej L.
- Jeśli $L = i$, to poddrzewo zaczepione w węźle i jest kopcem typu max, koniec algorytmu.
- Jeśli $L \neq i$, to zamieniamy elementy $A[i]$ i $A[L]$. Element $A[i]$ jest teraz nie mniejszy niż jego synowie, ale element $A[L]$ może być mniejszy od swoich synów. Wywołujemy rekurencyjnie przywracanie własności kopca dla tego węzła.
- W każdym kroku algorytmu schodzimy na niższy poziom kopca. Zatem przywracanie własności kopca na węźle o wysokości h zajmuje czas $O(h)$. Stąd czas przywracania własności kopca dla dowolnego ustalonego węzła w n-elementowym kopcu wynosi $O(\lg n)$.

Kopiec binarny

Przywracanie własności kopca typu max – idea:

- Porównujemy elementy $A[i]$, $A[LEFT(i)]$ oraz $A[RIGHT(i)]$. Indeks największego z nich zapamiętujemy w zmiennej L.
- Jeśli $L = i$, to poddrzewo zaczepione w węźle i jest kopcem typu max, koniec algorytmu.
- Jeśli $L \neq i$, to zamieniamy elementy $A[i]$ i $A[L]$. Element $A[i]$ jest teraz nie mniejszy niż jego synowie, ale element $A[L]$ może być mniejszy od swoich synów. Wywołujemy rekurencyjnie przywracanie własności kopca dla tego węzła.
- W każdym kroku algorytmu schodzimy na niższy poziom kopca. Zatem przywracanie własności kopca na węźle o wysokości h zajmuje czas $O(h)$. Stąd czas przywracania własności kopca dla dowolnego ustalonego węzła w n-elementowym kopcu wynosi $O(\lg n)$.

Kopiec binarny

Przywracanie własności kopca typu max – idea:

- Porównujemy elementy $A[i]$, $A[LEFT(i)]$ oraz $A[RIGHT(i)]$. Indeks największego z nich zapamiętujemy w zmiennej L.
- Jeśli $L = i$, to poddrzewo zaczepione w węźle i jest kopcem typu max, koniec algorytmu.
- Jeśli $L \neq i$, to zamieniamy elementy $A[i]$ i $A[L]$. Element $A[i]$ jest teraz nie mniejszy niż jego synowie, ale element $A[L]$ może być mniejszy od swoich synów. Wywołujemy rekurencyjnie przywracanie własności kopca dla tego węzła.
- W każdym kroku algorytmu schodzimy na niższy poziom kopca. Zatem przywracanie własności kopca na węźle o wysokości h zajmuje czas $O(h)$. Stąd czas przywracania własności kopca dla dowolnego ustalonego węzła w n-elementowym kopcu wynosi $O(\lg n)$.

max-heapify

```
procedure MAX-HEAPIFY(A,i)
 l = LEFT(i)
 r = RIGHT(i)
 if l ≤ A.heapsize and A[l] > A[i]
  L = l
 else
  L = i
 if r ≤ A.heapsize and A[r] > A[L]
  L = r
 if L ≠ i
  A[i] ↔ A[L]
  MAX-HEAPIFY(A,L)
```

Kopiec binarny

- Procedurę MAX-HEAPIFY można wykorzystać do przekształcenia dowolnej tablicy $A[1..n]$ w kopiec typu max.
- Elementy $A[\lfloor \frac{n}{2} \rfloor + 1], A[\lfloor \frac{n}{2} \rfloor + 2], \ldots, A[n]$ są liśćmi drzewa binarnego odpowiadającego tablicy A, czyli kopcami rozmiaru 1.
- Wystarczy przejść pozostałe węzły $A[\lfloor \frac{n}{2} \rfloor], A[\lfloor \frac{n}{2} \rfloor - 1], \ldots, A[1]$ i dla każdego z nich wywołać procedurę MAX-HEAPIFY.
- Algorytm budowania kopca jest przykładem podejścia wstępującego (bottom-up).

Kopiec binarny

- Procedurę MAX-HEAPIFY można wykorzystać do przekształcenia dowolnej tablicy $A[1..n]$ w kopiec typu max.
- Elementy $A[\lfloor \frac{n}{2} \rfloor + 1], A[\lfloor \frac{n}{2} \rfloor + 2], \ldots, A[n]$ są liśćmi drzewa binarnego odpowiadającego tablicy A, czyli kopcami rozmiaru 1.
- Wystarczy przejść pozostałe węzły $A[\lfloor \frac{n}{2} \rfloor], A[\lfloor \frac{n}{2} \rfloor - 1], \ldots, A[1]$ i dla każdego z nich wywołać procedurę MAX-HEAPIFY.
- Algorytm budowania kopca jest przykładem podejścia wstępującego (bottom-up).

Kopiec binarny

- Procedurę MAX-HEAPIFY można wykorzystać do przekształcenia dowolnej tablicy $A[1..n]$ w kopiec typu max.
- Elementy $A[\lfloor \frac{n}{2} \rfloor + 1], A[\lfloor \frac{n}{2} \rfloor + 2], \ldots, A[n]$ są liśćmi drzewa binarnego odpowiadającego tablicy A, czyli kopcami rozmiaru 1.
- Wystarczy przejść pozostałe węzły $A[\lfloor \frac{n}{2} \rfloor], A[\lfloor \frac{n}{2} \rfloor - 1], \ldots, A[1]$ i dla każdego z nich wywołać procedurę MAX-HEAPIFY.
- Algorytm budowania kopca jest przykładem podejścia wstępującego (bottom-up).

Kopiec binarny

- Procedurę MAX-HEAPIFY można wykorzystać do przekształcenia dowolnej tablicy $A[1..n]$ w kopiec typu max.
- Elementy $A[\lfloor \frac{n}{2} \rfloor + 1], A[\lfloor \frac{n}{2} \rfloor + 2], \ldots, A[n]$ są liśćmi drzewa binarnego odpowiadającego tablicy A, czyli kopcami rozmiaru 1.
- Wystarczy przejść pozostałe węzły $A[\lfloor \frac{n}{2} \rfloor], A[\lfloor \frac{n}{2} \rfloor - 1], \ldots, A[1]$ i dla każdego z nich wywołać procedurę MAX-HEAPIFY.
- Algorytm budowania kopca jest przykładem podejścia wstępującego (bottom-up).

build-max-heap

```
procedure BUILD-MAX-HEAP(A)
 A.heapsize = A.length
 for i = A.length DIV 2 downto 1
  MAX-HEAPIFY(A,i)
```

- Procedura BUILD-MAX-HEAP składa się z $O(n)$ wywołań procedury MAX-HEAPIFY, działającej w czasie O(lg n). Zatem czas budowania kopca wynosi co najwyżej $O(n \lg n)$.
- Czas wykonania procedury MAX-HEAPIFY dla węzła o wysokości h wynosi $O(h)$. W kopcu binarnym większość węzłów ma wysokość znacznie mniejszą niż $\lg n$. Wykorzystując tę obserwację, można pokazać, że procedura BUILD-MAX-HEAP działa w czasie $\Theta(n)$.

build-max-heap

```
procedure BUILD-MAX-HEAP(A)
 A.heapsize = A.length
 for i = A.length DIV 2 downto 1
  MAX-HEAPIFY(A,i)
```

- Procedura BUILD-MAX-HEAP składa się z $O(n)$ wywołań procedury MAX-HEAPIFY, działającej w czasie O(lg n). Zatem czas budowania kopca wynosi co najwyżej $O(n \lg n)$.
- Czas wykonania procedury MAX-HEAPIFY dla węzła o wysokości h wynosi $O(h)$. W kopcu binarnym większość węzłów ma wysokość znacznie mniejszą niż lg n. Wykorzystując tę obserwację, można pokazać, że procedura BUILD-MAX-HEAP działa w czasie $\Theta(n)$.

Sortowanie przez kopcowanie

Sortowanie przez kopcowanie – idea:

- Z wejściowej tablicy $A[1..n]$ tworzymy kopiec typu max.
- Największy element kopca znajduje się w $A[1]$. Zamieniamy elementy $A[1]$ i $A[n]$, aby ustawić największy element na właściwym miejscu.
- W tablicy $A[1..n-1]$ pierwszy element jest jedynym, który może naruszać własność kopca typu max. Przywracamy własność kopca, tworząc kopiec z tablicy $A[1..n-1]$.
- Powtarzamy powyższe postępowanie umieszczając kolejne największe elementy w tablicy na pozycjach $n-1, n-2, \ldots, 2$.

Sortowanie przez kopcowanie

Sortowanie przez kopcowanie – idea:

- Z wejściowej tablicy $A[1..n]$ tworzymy kopiec typu max.
- Największy element kopca znajduje się w $A[1]$. Zamieniamy elementy $A[1]$ i $A[n]$, aby ustawić największy element na właściwym miejscu.
- W tablicy $A[1..n-1]$ pierwszy element jest jedynym, który może naruszać własność kopca typu max. Przywracamy własność kopca, tworząc kopiec z tablicy $A[1..n-1]$.
- Powtarzamy powyższe podejście, umieszczając kolejne największe elementy w tablicy na pozycjach $n-1, n-2, \ldots, 2$.

Sortowanie przez kopcowanie

Sortowanie przez kopcowanie – idea:

- Z wejściowej tablicy $A[1..n]$ tworzymy kopiec typu max.
- Największy element kopca znajduje się w $A[1]$. Zamieniamy elementy $A[1]$ i $A[n]$, aby ustawić największy element na właściwym miejscu.
- W tablicy $A[1..n-1]$ pierwszy element jest jedynym, który może naruszać własność kopca typu max. Przywracamy własność kopca, tworząc kopiec z tablicy $A[1..n-1]$.
- Powtarzamy powyższe podejście, umieszczając kolejne największe elementy w tablicy na pozycjach $n-1, n-2, \ldots, 2$.

Sortowanie przez kopcowanie

Sortowanie przez kopcowanie – idea:

- Z wejściowej tablicy $A[1..n]$ tworzymy kopiec typu max.
- Największy element kopca znajduje się w $A[1]$. Zamieniamy elementy $A[1]$ i $A[n]$, aby ustawić największy element na właściwym miejscu.
- W tablicy $A[1..n-1]$ pierwszy element jest jedynym, który może naruszać własność kopca typu max. Przywracamy własność kopca, tworząc kopiec z tablicy $A[1..n-1]$.
- Powtarzamy powyższe podejście, umieszczając kolejne największe elementy w tablicy na pozycjach $n-1, n-2, \ldots, 2$.

Sortowanie przez kopcowanie

Sortowanie przez kopcowanie – idea:

- Z wejściowej tablicy $A[1..n]$ tworzymy kopiec typu max.
- Największy element kopca znajduje się w $A[1]$. Zamieniamy elementy $A[1]$ i $A[n]$, aby ustawić największy element na właściwym miejscu.
- W tablicy $A[1..n-1]$ pierwszy element jest jedynym, który może naruszać własność kopca typu max. Przywracamy własność kopca, tworząc kopiec z tablicy $A[1..n-1]$.
- Powtarzamy powyższe podejście, umieszczając kolejne największe elementy w tablicy na pozycjach $n-1, n-2, \ldots, 2$.

Sortowanie przez kopcowanie

```
procedure HEAPSORT(A)
 BUILD-MAX-HEAP(A)
 for i = A.length downto 2
  A[1] ↔ A[i]
  A.heapsize = A.heapsize - 1
  MAX-HEAPIFY(A,1)
```

Sortowanie przez kopcowanie działa w czasie $O(n \lg n)$.

Sortowanie przez kopcowanie

```
procedure HEAPSORT(A)
 BUILD-MAX-HEAP(A)
 for i = A.length downto 2
  A[1] ↔ A[i]
  A.heapsize = A.heapsize - 1
  MAX-HEAPIFY(A,1)
```

Sortowanie przez kopcowanie działa w czasie $O(n \lg n)$.

Kolejka priorytetowa

- Podstawowym zastosowaniem kopca binarnego jest implementacja kolejki priorytetowej.
- Kolejka priorytetowa – struktura danych służąca do reprezentowania zbioru S zawierającego elementy, z których każdy ma określony priorytet (klucz).
- Podobnie jak w przypadku kopców binarnych, możemy mówić o kolejce priorytetowej typu max lub typu min.

Kolejka priorytetowa

- Podstawowym zastosowaniem kopca binarnego jest implementacja kolejki priorytetowej.
- Kolejka priorytetowa – struktura danych służąca do reprezentowania zbioru S zawierającego elementy, z których każdy ma określony priorytet (klucz).
- Podobnie jak w przypadku kopców binarnych, możemy mówić o kolejce priorytetowej typu max lub typu min.

Kolejka priorytetowa

- Podstawowym zastosowaniem kopca binarnego jest implementacja kolejki priorytetowej.
- Kolejka priorytetowa – struktura danych służąca do reprezentowania zbioru S zawierającego elementy, z których każdy ma określony priorytet (klucz).
- Podobnie jak w przypadku kopców binarnych, możemy mówić o kolejce priorytetowej typu max lub typu min.

Kolejka priorytetowa - operacje

Operacje na kolejce priorytetowej typu max:

- INSERT(S, x) – wstawia element x do zbioru S
- MAXIMUM(S) – zwraca element zbioru S o największym kluczu
- EXTRACT-MAX(S) – usuwa i zwraca element zbioru S o największym kluczu
- INCREASE-KEY(S, x, k) – zmienia wartość klucza elementu x na nową wartość k (nie mniejszą niż aktualna wartość klucza elementu x) (procedura pomocnicza, zazwyczaj nie jest udostępniana przez kolejki priorytetowe z gotowych bibliotek)

Kolejka priorytetowa - operacje

Operacje na kolejce priorytetowej typu max:

- INSERT(S, x) – wstawia element x do zbioru S
- MAXIMUM(S) – zwraca element zbioru S o największym kluczu
- EXTRACT-MAX(S) – usuwa i zwraca element zbioru S o największym kluczu
- INCREASE-KEY(S, x, k) – zmienia wartość klucza elementu x na nową wartość k, nie mniejszą niż aktualna wartość klucza elementu x (procedura pomocnicza, zazwyczaj nie jest udostępniana przez kolejki priorytetowe z gotowych bibliotek)

Kolejka priorytetowa - operacje

Operacje na kolejce priorytetowej typu max:

- INSERT(S, x) – wstawia element x do zbioru S
- MAXIMUM(S) – zwraca element zbioru S o największym kluczu
- EXTRACT-MAX(S) – usuwa i zwraca element zbioru S o największym kluczu
- INCREASE-KEY(S, x, k) – zmienia wartość klucza elementu x na nową wartość k, nie mniejszą niż aktualna wartość klucza elementu x (procedura pomocnicza, zazwyczaj nie jest udostępniana przez kolejki priorytetowe z gotowych bibliotek)

Kolejka priorytetowa - operacje

Operacje na kolejce priorytetowej typu max:

- INSERT(S, x) – wstawia element x do zbioru S
- MAXIMUM(S) – zwraca element zbioru S o największym kluczu
- EXTRACT-MAX(S) – usuwa i zwraca element zbioru S o największym kluczu
- INCREASE-KEY(S, x, k) – zmienia wartość klucza elementu x na nową wartość k, nie mniejszą niż aktualna wartość klucza elementu x (procedura pomocnicza, zazwyczaj nie jest udostępniana przez kolejki priorytetowe z gotowych bibliotek)

Kolejka priorytetowa - operacje

Operacje na kolejce priorytetowej typu max:

- INSERT(S, x) – wstawia element x do zbioru S
- MAXIMUM(S) – zwraca element zbioru S o największym kluczu
- EXTRACT-MAX(S) – usuwa i zwraca element zbioru S o największym kluczu
- INCREASE-KEY(S, x, k) – zmienia wartość klucza elementu x na nową wartość k, nie mniejszą niż aktualna wartość klucza elementu x (procedura pomocnicza, zazwyczaj nie jest udostępniana przez kolejki priorytetowe z gotowych bibliotek)

Kolejka priorytetowa - operacje

Operacje na kolejce priorytetowej typu min:

- INSERT(S, x) – wstawia element x do zbioru S
- MINIMUM(S) – zwraca element zbioru S o najmniejszym kluczu
- EXTRACT-MIN(S) – usuwa i zwraca element zbioru S o najmniejszym kluczu
- DECREASE-KEY(S, x, k) – zmienia wartość klucza elementu x na nową wartość k nie większą niż aktualna wartość klucza elementu x (procedura pomocnicza, zazwyczaj nie jest udostępniana przez kolejki priorytetowe z gotowych bibliotek)

Kolejka priorytetowa - operacje

Operacje na kolejce priorytetowej typu min:

- INSERT(S, x) – wstawia element x do zbioru S
- MINIMUM(S) – zwraca element zbioru S o najmniejszym kluczu
- EXTRACT-MIN(S) – usuwa i zwraca element zbioru S o najmniejszym kluczu
- DECREASE-KEY(S, x, k) – zmienia wartość klucza elementu x na nową wartość k, nie większa niż aktualna wartość klucza elementu x (procedura pomocnicza, zazwyczaj nie jest udostępniana przez kolejki priorytetowe z gotowych bibliotek)

Kolejka priorytetowa - operacje

Operacje na kolejce priorytetowej typu min:

- INSERT(S, x) – wstawia element x do zbioru S
- MINIMUM(S) – zwraca element zbioru S o najmniejszym kluczu
- EXTRACT-MIN(S) – usuwa i zwraca element zbioru S o najmniejszym kluczu
- DECREASE-KEY(S, x, k) – zmienia wartość klucza elementu x na nową wartość k, nie większa niż aktualna wartość klucza elementu x (procedura pomocnicza, zazwyczaj nie jest udostępniana przez kolejki priorytetowe z gotowych bibliotek)

Kolejka priorytetowa - operacje

Operacje na kolejce priorytetowej typu min:

- INSERT(S, x) – wstawia element x do zbioru S
- MINIMUM(S) – zwraca element zbioru S o najmniejszym kluczu
- EXTRACT-MIN(S) – usuwa i zwraca element zbioru S o najmniejszym kluczu
- DECREASE-KEY(S, x, k) – zmienia wartość klucza elementu x na nową wartość k, nie większa niż aktualna wartość klucza elementu x (procedura pomocnicza, zazwyczaj nie jest udostępniana przez kolejki priorytetowe z gotowych bibliotek)

Kolejka priorytetowa - operacje

Operacje na kolejce priorytetowej typu min:

- INSERT(S, x) – wstawia element x do zbioru S
- MINIMUM(S) – zwraca element zbioru S o najmniejszym kluczu
- EXTRACT-MIN(S) – usuwa i zwraca element zbioru S o najmniejszym kluczu
- DECREASE-KEY(S, x, k) – zmienia wartość klucza elementu x na nową wartość k, nie większa niż aktualna wartość klucza elementu x (procedura pomocnicza, zazwyczaj nie jest udostępniana przez kolejki priorytetowe z gotowych bibliotek)

Kolejka priorytetowa

Przykładowe zastosowania kolejki priorytetowej:

- szeregowanie zadań z określonymi priorytetami
- symulator zdarzeń o określonych czasach wystąpienia
- znajdowanie minimalnego drzewa rozpinającego w grafie
- znajdowanie w grafie z wagami na krawędziach najkrótszych ścieżek z jednym źródłem
- algorytm Huffmana kompresji danych

Kolejka priorytetowa

Przykładowe zastosowania kolejki priorytetowej:

- szeregowanie zadań z określonymi priorytetami
- symulator zdarzeń o określonych czasach wystąpienia
- znajdowanie minimalnego drzewa rozpinającego w grafie
- znajdowanie w grafie z wagami na krawędziach najkrótszych ścieżek z jednym źródłem
- algorytm Huffmana kompresji danych

Kolejka priorytetowa

Przykładowe zastosowania kolejki priorytetowej:

- szeregowanie zadań z określonymi priorytetami
- symulator zdarzeń o określonych czasach wystąpienia
- znajdowanie minimalnego drzewa rozpinającego w grafie
- znajdowanie w grafie z wagami na krawędziach najkrótszych ścieżek z jednym źródłem
- algorytm Huffmana kompresji danych

Kolejka priorytetowa

Przykładowe zastosowania kolejki priorytetowej:

- szeregowanie zadań z określonymi priorytetami
- symulator zdarzeń o określonych czasach wystąpienia
- znajdowanie minimalnego drzewa rozpinającego w grafie
- znajdowanie w grafie z wagami na krawędziach najkrótszych ścieżek z jednym źródłem
- algorytm Huffmana kompresji danych

Kolejka priorytetowa

Przykładowe zastosowania kolejki priorytetowej:

- szeregowanie zadań z określonymi priorytetami
- symulator zdarzeń o określonych czasach wystąpienia
- znajdowanie minimalnego drzewa rozpinającego w grafie
- znajdowanie w grafie z wagami na krawędziach najkrótszych ścieżek z jednym źródłem
- algorytm Huffmana kompresji danych

Kolejka priorytetowa

Implementacja kolejki priorytetowej typu max za pomocą kopca binarnego typu max

- Elementy kolejki priorytetowej (kopca) odpowiadają pewnym obiektom. Konieczne jest określenie, jaki obiekt odpowiada danemu elementowi kolejki (lub odwrotnie). Nie możemy więc przechowywać w kopcu wyłącznie kluczy (priorytetów).
- Każdy element kopca musi oprócz klucza (priorytetu) odpowiadającego mu obiektu przechowywać odniesienie do tego obiektu (wskaźnik lub identyfikator). Dla wygody w pseudokodzie nie zaznaczamy, że elementy kopca są rekordami. Porównywanie elementów (rekordów) oznacza porównywanie ich kluczy.

Kolejka priorytetowa

Implementacja kolejki priorytetowej typu max za pomocą kopca binarnego typu max

- Elementy kolejki priorytetowej (kopca) odpowiadają pewnym obiektom. Konieczne jest określenie, jaki obiekt odpowiada danemu elementowi kolejki (lub odwrotnie). Nie możemy więc przechowywać w kopcu wyłącznie kluczy (priorytetów).
- Każdy element kopca musi oprócz klucza (priorytetu) odpowiadającego mu obiektu przechowywać odniesienie do tego obiektu (wskaźnik lub identyfikator). Dla wygody w pseudokodzie nie zaznaczamy, że elementy kopca są rekordami. Porównywanie elementów (rekordów) oznacza porównywanie ich kluczy.

Kolejka priorytetowa

Implementacja operacji MAXIMUM

```
procedure HEAP-MAXIMUM(A)
 if A.heapsize = 0
  error "niedomiar"
 else
  return A[1]
```

Złożoność dla kolejki priorytetowej zawierającej n elementów: $\Theta(1)$.

Kolejka priorytetowa

Implementacja operacji MAXIMUM

```
procedure HEAP-MAXIMUM(A)
 if A.heapsize = 0
  error "niedomiar"
 else
  return A[1]
```

Złożoność dla kolejki priorytetowej zawierającej n elementów: $\Theta(1)$.

Kolejka priorytetowa

Implementacja operacji MAXIMUM

```
procedure HEAP-MAXIMUM(A)
 if A.heapsize = 0
  error "niedomiar"
 else
  return A[1]
```

Złożoność dla kolejki priorytetowej zawierającej n elementów: $\Theta(1)$.

Kolejka priorytetowa

Implementacja operacji EXTRACT-MAX

```
procedure HEAP-EXTRACT-MAX(A)
 if A.heapsize = 0
  error "niedomiar"
 else
  max = A[1]
  A[1] = A[A.heapsize]
  A.heapsize = A.heapsize - 1
  MAX-HEAPIFY(A,1)
  return max
```

Złożoność dla kolejki priorytetowej zawierającej n elementów: $O(\lg n)$.

Kolejka priorytetowa

Implementacja operacji EXTRACT-MAX

```
procedure HEAP-EXTRACT-MAX(A)
 if A.heapsize = 0
  error "niedomiar"
 else
  max = A[1]
  A[1] = A[A.heapsize]
  A.heapsize = A.heapsize - 1
  MAX-HEAPIFY(A,1)
  return max
```

Złożoność dla kolejki priorytetowej zawierającej n elementów: $O(\lg n)$.

Kolejka priorytetowa

Implementacja operacji EXTRACT-MAX

```
procedure HEAP-EXTRACT-MAX(A)
 if A.heapsize = 0
  error "niedomiar"
 else
  max = A[1]
  A[1] = A[A.heapsize]
  A.heapsize = A.heapsize - 1
  MAX-HEAPIFY(A,1)
  return max
```

Złożoność dla kolejki priorytetowej zawierającej n elementów: $O(\lg n)$.

Kolejka priorytetowa

Implementacja operacji INCREASE-KEY

```
procedure HEAP-INCREASE-KEY(A, i, key)
 if key < A[i]
  error
 else
  A[i] = key
  while i > 1 and A[PARENT(i)] < A[i]
   A[i] ↔ A[PARENT(i)]
   i = PARENT(i)
```

Złożoność dla kolejki priorytetowej zawierającej n elementów: $O(\lg n)$.

Kolejka priorytetowa

Implementacja operacji INCREASE-KEY

```
procedure HEAP-INCREASE-KEY(A, i, key)
 if key < A[i]
  error
 else
  A[i] = key
  while i > 1 and A[PARENT(i)] < A[i]
   A[i] ↔ A[PARENT(i)]
   i = PARENT(i)
```

Złożoność dla kolejki priorytetowej zawierającej n elementów: $O(\lg n)$.

Kolejka priorytetowa

Implementacja operacji INCREASE-KEY

```
procedure HEAP-INCREASE-KEY(A, i, key)
 if key < A[i]
  error
 else
  A[i] = key
  while i > 1 and A[PARENT(i)] < A[i]
   A[i] ↔ A[PARENT(i)]
   i = PARENT(i)
```

Złożoność dla kolejki priorytetowej zawierającej n elementów: $O(\lg n)$.

Kolejka priorytetowa

Implementacja operacji INSERT:

```
procedure MAX-HEAP-INSERT(A, key)
 if A.heapsize = A.length
  error "przepełnienie"
 else
  A.heapsize = A.heapsize + 1
  A[A.heapsize]= −∞
  HEAP-INCREASE-KEY(A, A.heapsize, key)
```

Złożoność dla kolejki priorytetowej zawierającej n elementów: $O(\lg n)$.

Kolejka priorytetowa

Implementacja operacji INSERT:

```
procedure MAX-HEAP-INSERT(A, key)
 if A.heapsize = A.length
  error "przepełnienie"
 else
  A.heapsize = A.heapsize + 1
  A[A.heapsize]= −∞
  HEAP-INCREASE-KEY(A, A.heapsize, key)
```

Złożoność dla kolejki priorytetowej zawierającej n elementów: $O(\lg n)$.