Kopiec, wykorzystanie do Dijkstry

II kurs, 2. zajęcia

Marcin Andrychowicz

Wstęp

Porównanie złożoności

	wstawienie	usunięcie	minimum

- **wstawienie** — wstawienie elementu
- **usunięcie** — usunięcie elementu na podanej pozycji (a nie o danej wartości)
- **minimum** — znalezienie minimum

Wstęp

Porównanie złożoności

	wstawienie	usunięcie	minimum
tablica			

- **wstawienie** — wstawienie elementu
- **usunięcie** — usunięcie elementu na podanej pozycji (a nie o danej wartości)
- **minimum** — znalezienie minimum

Wstęp

Porównanie złożoności

	wstawienie	usunięcie	minimum
tablica	$O(1)$		

- **wstawienie** — wstawienie elementu
- **usunięcie** — usunięcie elementu na podanej pozycji (a nie o danej wartości)
- **minimum** — znalezienie minimum

Wstęp

Porównanie złożoności

	wstawienie	usunięcie	minimum
tablica	$O(1)$	$O(1)$	

- **wstawienie** — wstawienie elementu
- **usunięcie** — usunięcie elementu na podanej pozycji (a nie o danej wartości)
- **minimum** — znalezienie minimum

Wstęp

Porównanie złożoności

	wstawienie	usunięcie	minimum
tablica	$O(1)$	$O(1)$	$O(n)$

- **wstawienie** — wstawienie elementu
- **usunięcie** — usunięcie elementu na podanej pozycji (a nie o danej wartości)
- **minimum** — znalezienie minimum

Wstęp

Porównanie złożoności

	wstawienie	usunięcie	minimum
tablica	$O(1)$	$O(1)$	$O(n)$
posortowana tablica			

- **wstawienie** — wstawienie elementu
- **usunięcie** — usunięcie elementu na podanej pozycji (a nie o danej wartości)
- **minimum** — znalezienie minimum

Wstęp

Porównanie złożoności

	wstawienie	usunięcie	minimum
tablica	$O(1)$	$O(1)$	$O(n)$
posortowana tablica	$O(n)$		

- **wstawienie** — wstawienie elementu
- **usunięcie** — usunięcie elementu na podanej pozycji (a nie o danej wartości)
- **minimum** — znalezienie minimum

Wstęp

Porównanie złożoności

	wstawienie	usunięcie	minimum
tablica	$O(1)$	$O(1)$	$O(n)$
posortowana tablica	$O(n)$	$O(n)$	

- **wstawienie** — wstawienie elementu
- **usunięcie** — usunięcie elementu na podanej pozycji (a nie o danej wartości)
- **minimum** — znalezienie minimum

Wstęp

Porównanie złożoności

	wstawienie	usunięcie	minimum
tablica	$O(1)$	$O(1)$	$O(n)$
posortowana tablica	$O(n)$	$O(n)$	$O(1)$

- **wstawienie** — wstawienie elementu
- **usunięcie** — usunięcie elementu na podanej pozycji (a nie o danej wartości)
- **minimum** — znalezienie minimum

Wstęp

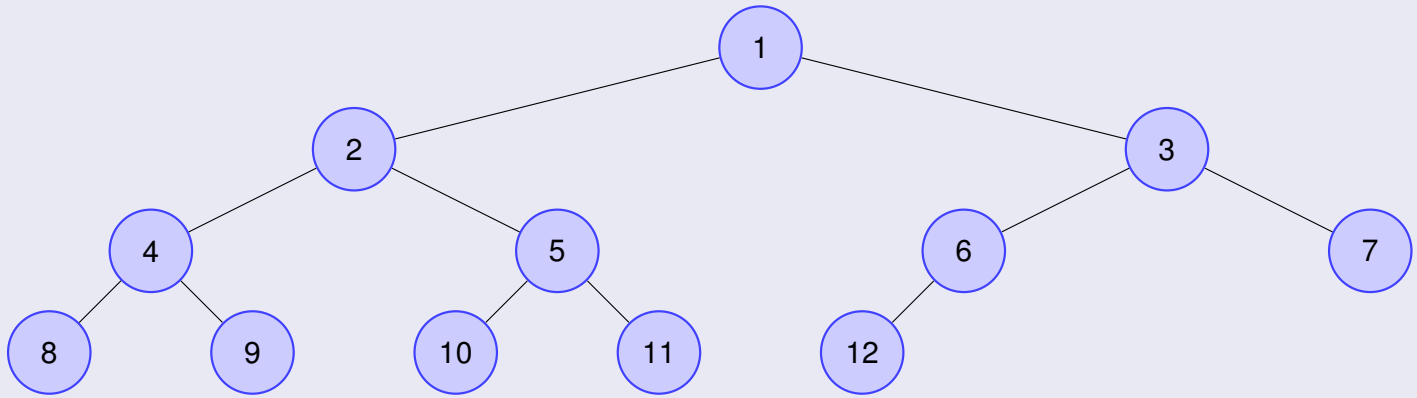
Porównanie złożoności

	wstawienie	usunięcie	minimum
tablica	$O(1)$	$O(1)$	$O(n)$
posortowana tablica	$O(n)$	$O(n)$	$O(1)$
kopiec	$O(\log n)$	$O(\log n)$	$O(\log n)$

- **wstawienie** — wstawienie elementu
- **usunięcie** — usunięcie elementu na podanej pozycji (a nie o danej wartości)
- **minimum** — znalezienie minimum

Pełne drzewo binarne

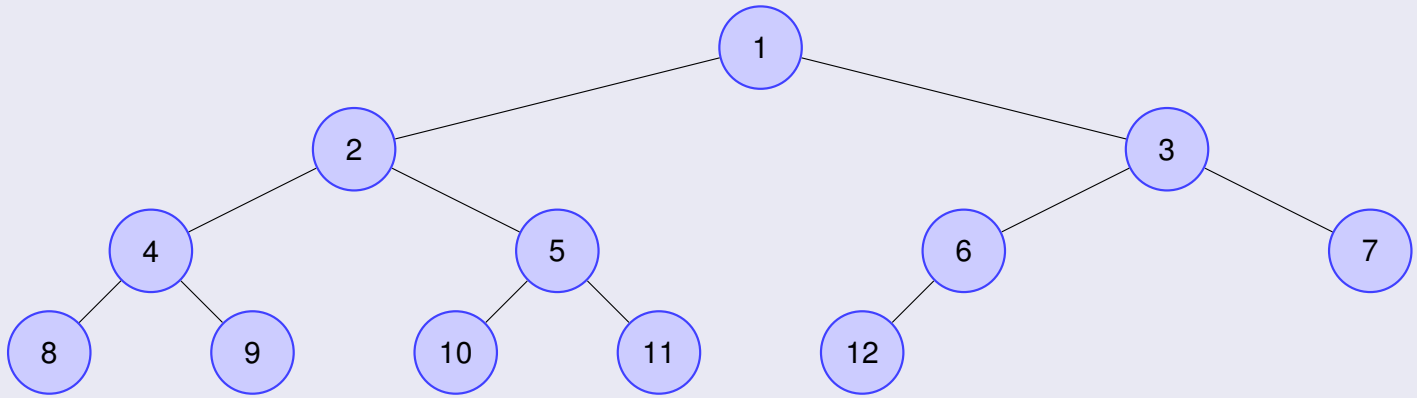
Pełne drzewo binarne



- wyjaśnienie nazwy:
  - **binarne** — każdy wierzchołek ma co najwyżej 2 synów
  - **pełne** — liście znajdują się tylko na 2 poziomach przy czym te na niższym poziomie są „z jednej strony”
- wierzchołki numerujemy jak na rysunku
- w wierzchołkach będziemy trzymali liczby
- `heap[i]` — wartość w i-tym wierzchołku

Pełne drzewo binarne

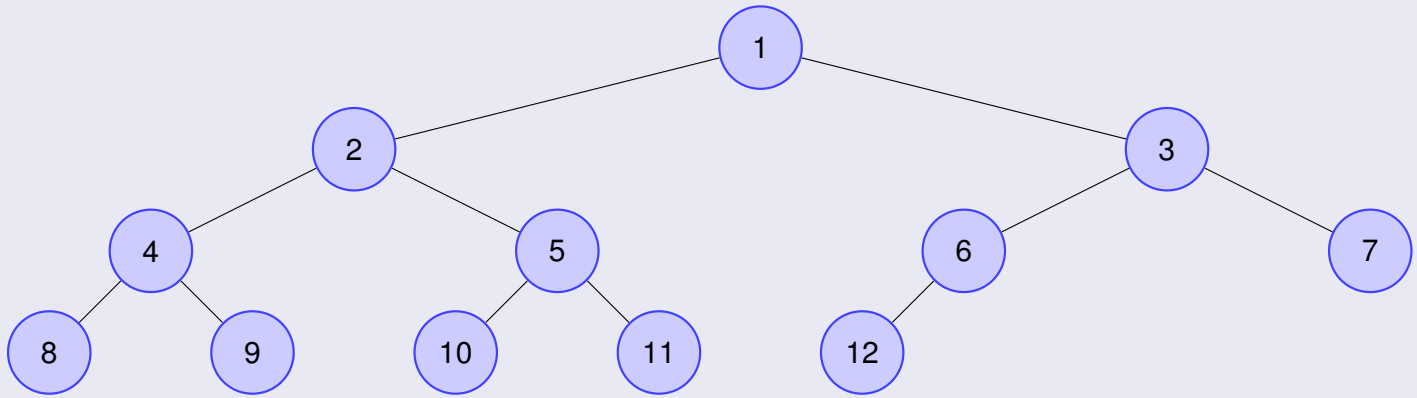
Pełne drzewo binarne



- wyjaśnienie nazwy:
  - **binarne** — każdy wierzchołek ma co najwyżej 2 synów
  - **pełne** — liście znajdują się tylko na 2 poziomach przy czym te na niższym poziomie są „z jednej strony”
- wierzchołki numerujemy jak na rysunku
- w wierzchołkach będziemy trzymali liczby
- `heap[i]` — wartość w *i*-tym wierzchołku

Pełne drzewo binarne

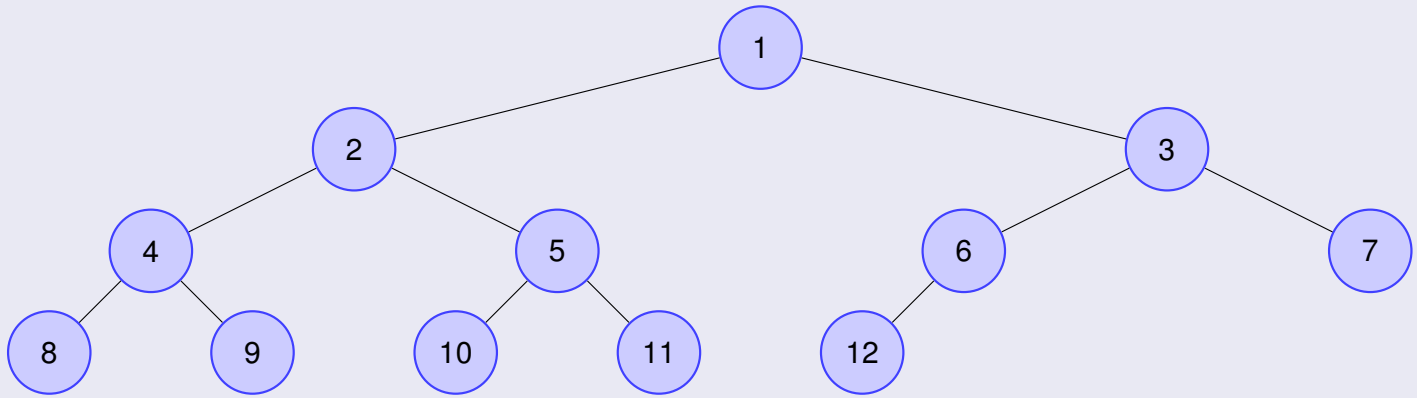
Pełne drzewo binarne



- wyjaśnienie nazwy:
  - **binarne** — każdy wierzchołek ma co najwyżej 2 synów
  - **pełne** — liście znajdują się tylko na 2 poziomach przy czym te na niższym poziomie są „z jednej strony”
- wierzchołki numerujemy jak na rysunku
- w wierzchołkach będziemy trzymali liczby
- `heap[i]` — wartość w *i*-tym wierzchołku

Pełne drzewo binarne

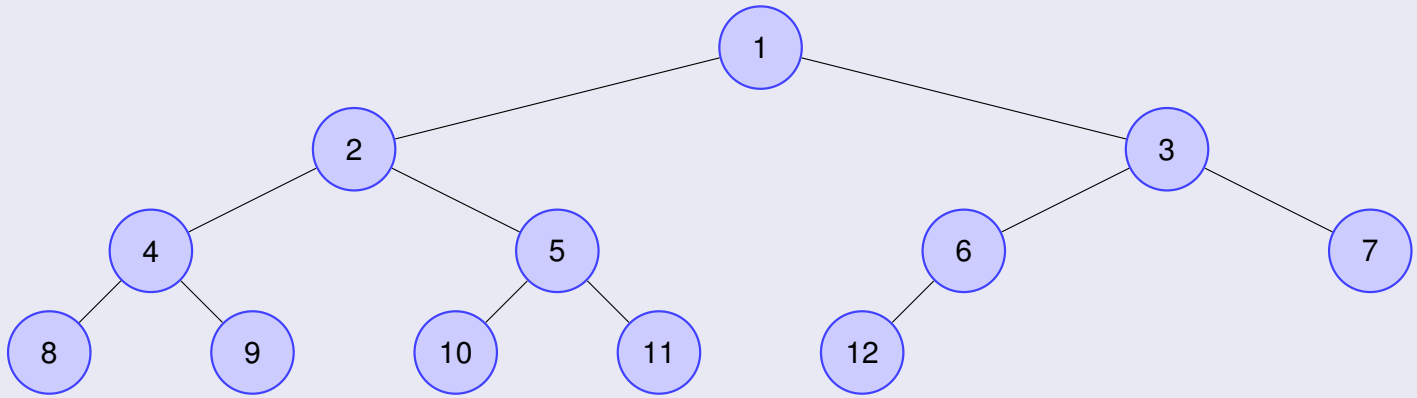
Pełne drzewo binarne



- wyjaśnienie nazwy:
  - **binarne** — każdy wierzchołek ma co najwyżej 2 synów
  - **pełne** — liście znajdują się tylko na 2 poziomach przy czym te na niższym poziomie są „z jednej strony”
- wierzchołki numerujemy jak na rysunku
- w wierzchołkach będziemy trzymali liczby
- `heap[i]` — wartość w *i*-tym wierzchołku

Pełne drzewo binarne

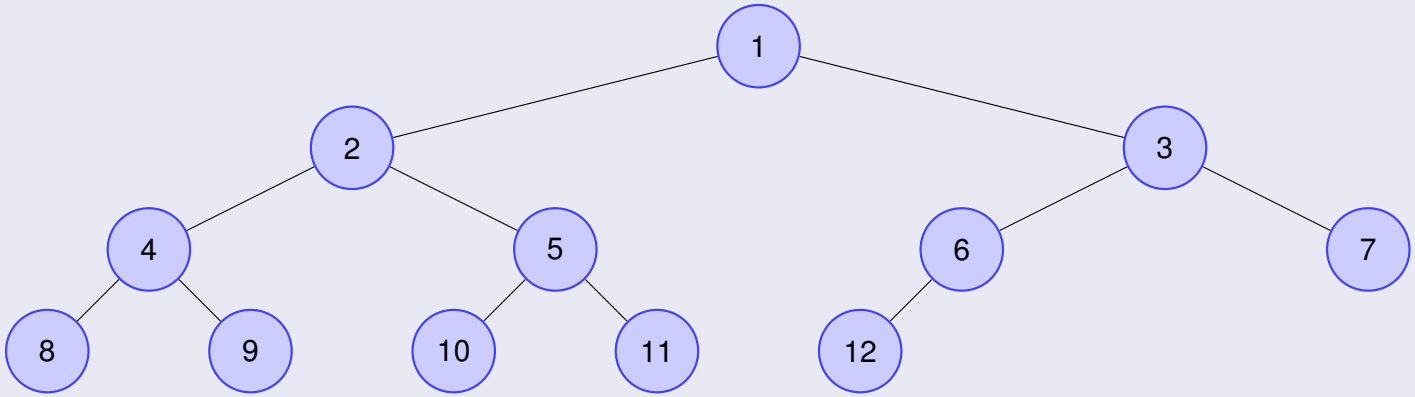
Pełne drzewo binarne



- wyjaśnienie nazwy:
  - **binarne** — każdy wierzchołek ma co najwyżej 2 synów
  - **pełne** — liście znajdują się tylko na 2 poziomach przy czym te na niższym poziomie są „z jednej strony”
- wierzchołki numerujemy jak na rysunku
- w wierzchołkach będziemy trzymali liczby
- `heap[i]` — wartość w *i*-tym wierzchołku

Pełne drzewo binarne

Pełne drzewo binarne

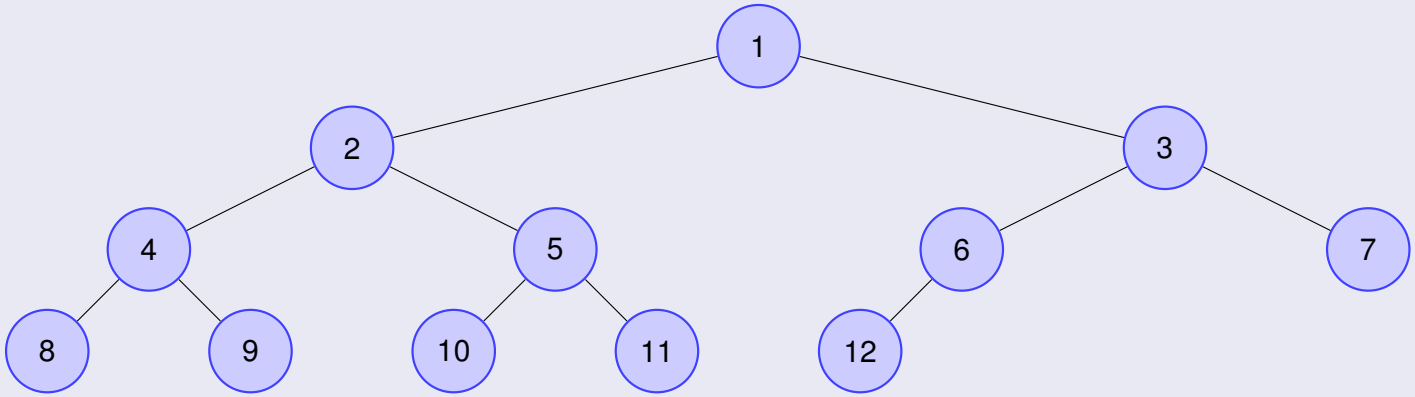


Dlaczego taka numeracja jest wygodna?

- lewy syn x to $2x$
- prawy syn x to $2x + 1$
- ojciec x to $\lfloor x/2 \rfloor$

Pełne drzewo binarne

Pełne drzewo binarne

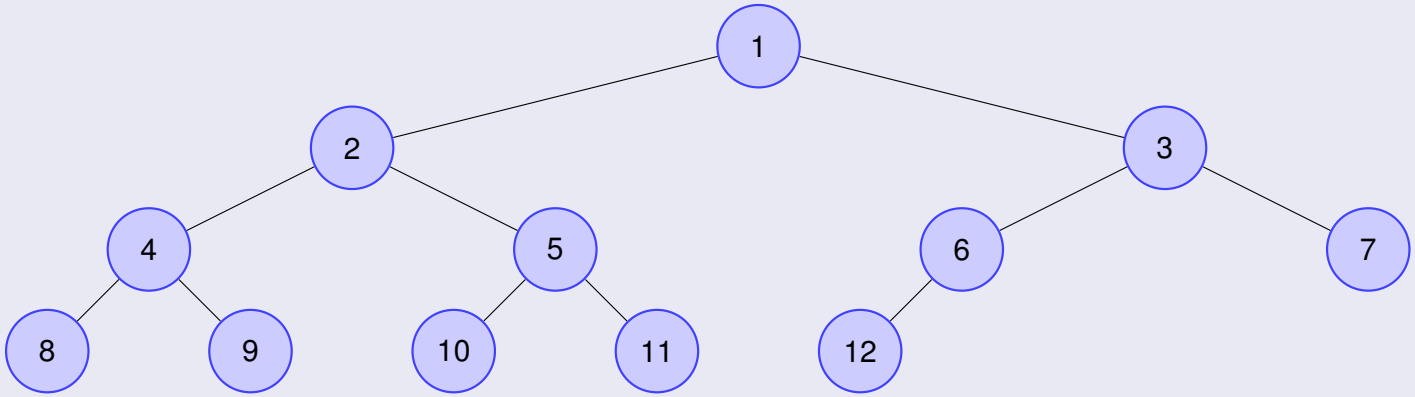


Dlaczego taka numeracja jest wygodna?

- lewy syn x to $2x$
- prawy syn x to $2x + 1$
- ojciec x to $\lfloor x/2 \rfloor$

Pełne drzewo binarne

Pełne drzewo binarne

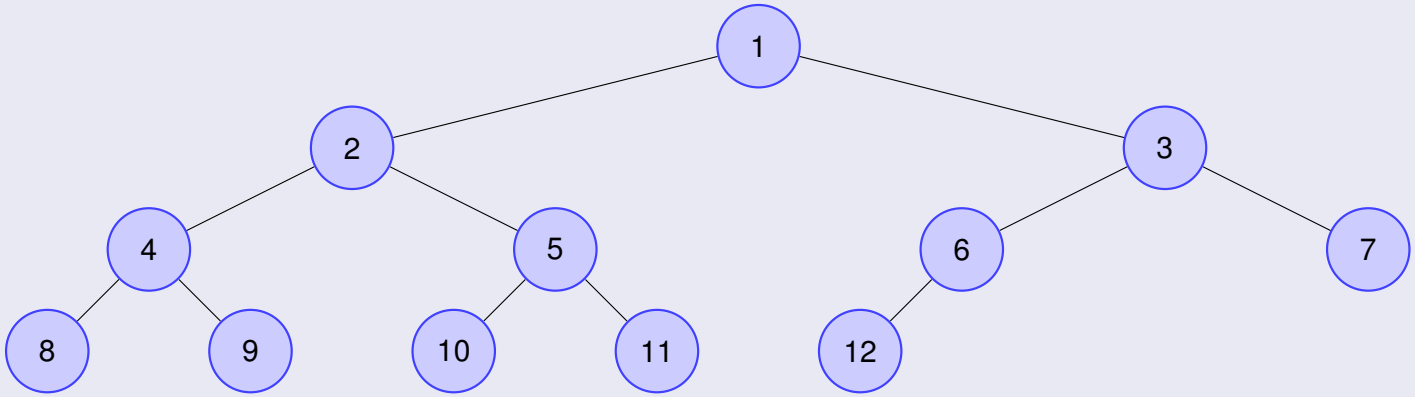


Dlaczego taka numeracja jest wygodna?

- lewy syn x to $2x$
- prawy syn x to $2x + 1$
- ojciec x to $\lfloor x/2 \rfloor$

Pełne drzewo binarne

Pełne drzewo binarne

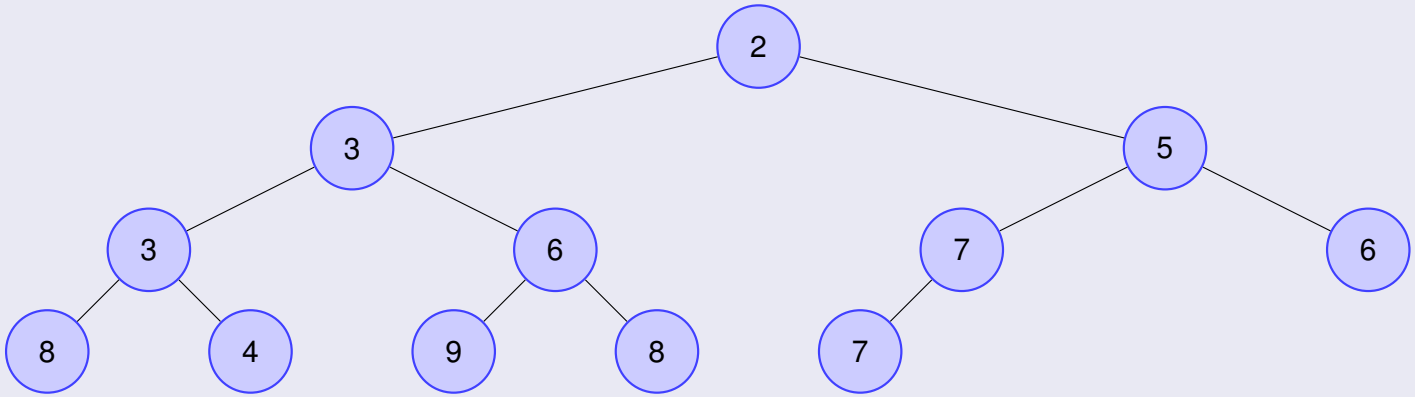


Dlaczego taka numeracja jest wygodna?

- lewy syn x to $2x$
- prawy syn x to $2x + 1$
- ojciec x to $\lfloor x/2 \rfloor$

Kopiec

Kopiec

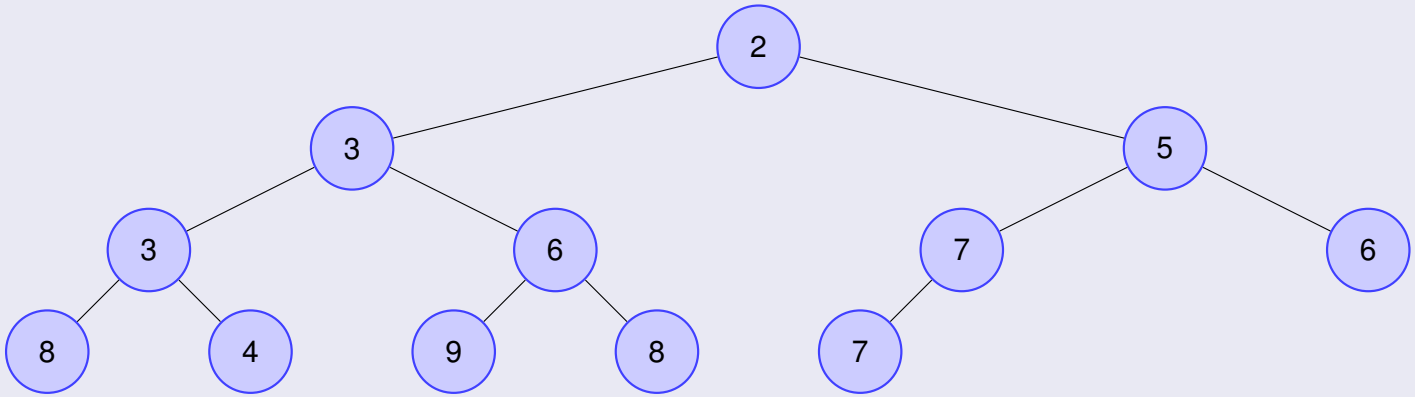


Własność kopca

`Kopcem` nazywamy pełne drzewo binarne, które posiada własność kopca, tzn. każdy wierzchołek ma przypisaną wartość nie większą niż jego synowie.
Innymi słowy, dla wszystkich x zachodzi $heap[\lfloor x/2 \rfloor] \leqslant heap[x]$.

Dodawanie elementu

Kopiec

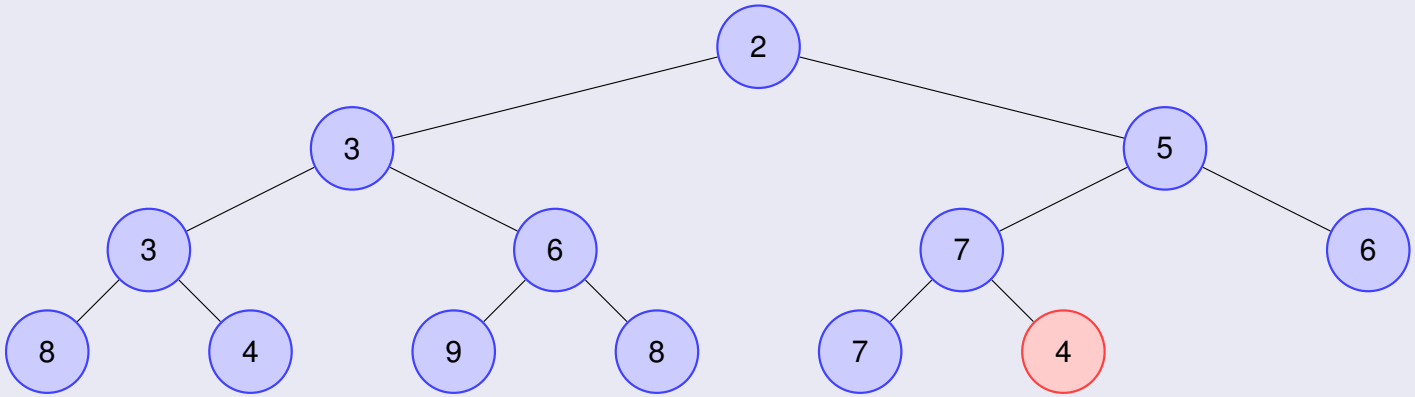


Jak dodać element do kopca?

1. dodajemy element na koniec tablicy (własność kopca może być zaburzona)
2. dopóki ojciec nowego elementu jest od niego większy, to zamieniamy wartości w obu wierzchołkach

Dodawanie elementu

Kopiec

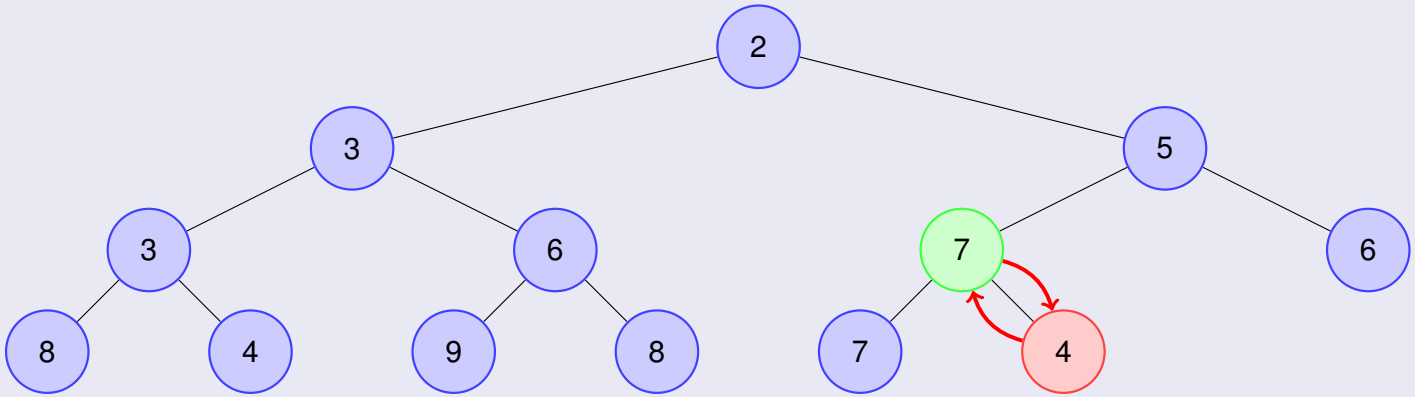


Jak dodać element do kopca?

1. dodajemy element na koniec tablicy (własność kopca może być zaburzona)
2. dopóki ojciec nowego elementu jest od niego większy, to zamieniamy wartości w obu wierzchołkach

Dodawanie elementu

Kopiec

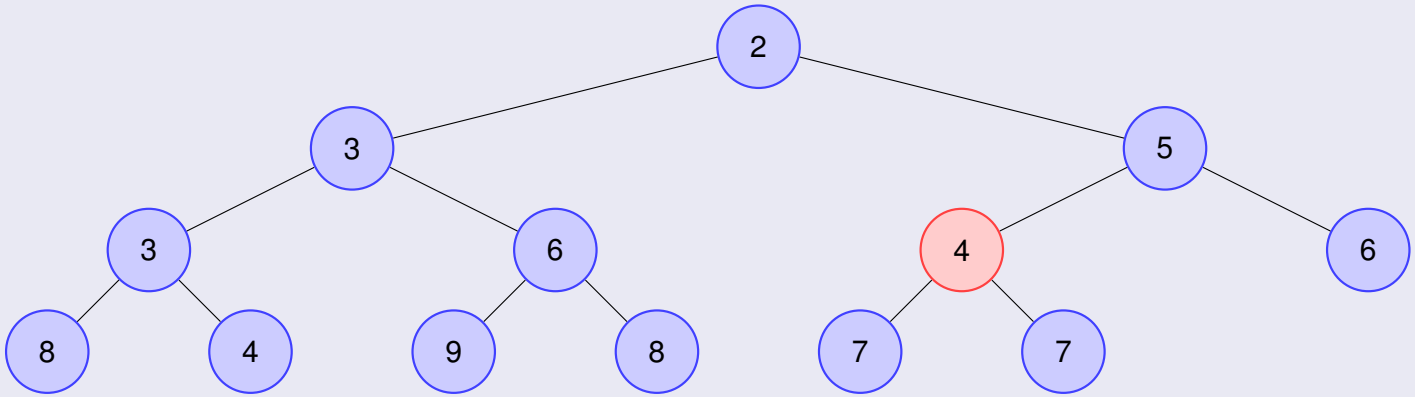


Jak dodać element do kopca?

1. dodajemy element na koniec tablicy (własność kopca może być zaburzona)
2. dopóki ojciec nowego elementu jest od niego większy, to zamieniamy wartości w obu wierzchołkach

Dodawanie elementu

Kopiec

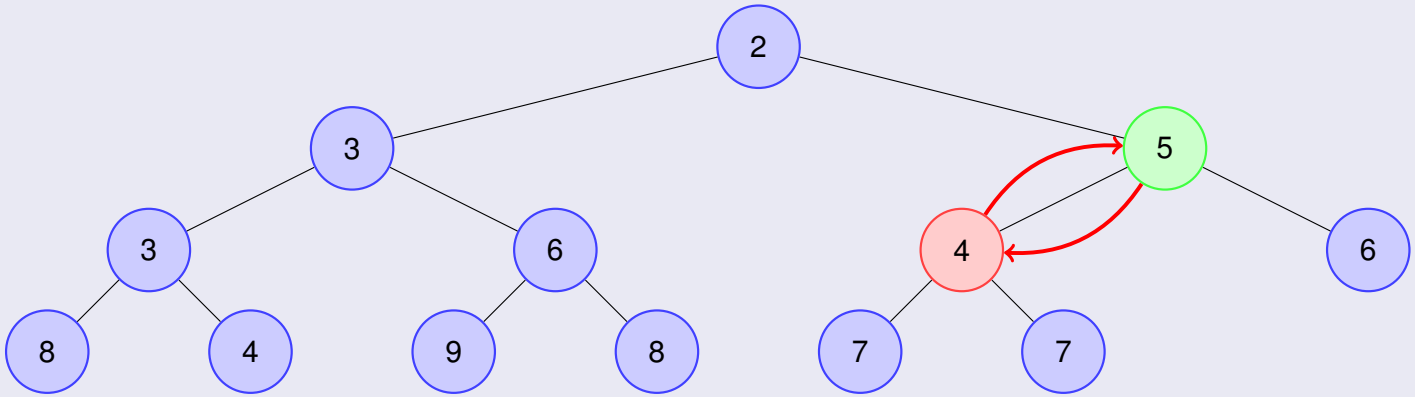


Jak dodać element do kopca?

1. dodajemy element na koniec tablicy (własność kopca może być zaburzona)
2. dopóki ojciec nowego elementu jest od niego większy, to zamieniamy wartości w obu wierzchołkach

Dodawanie elementu

Kopiec

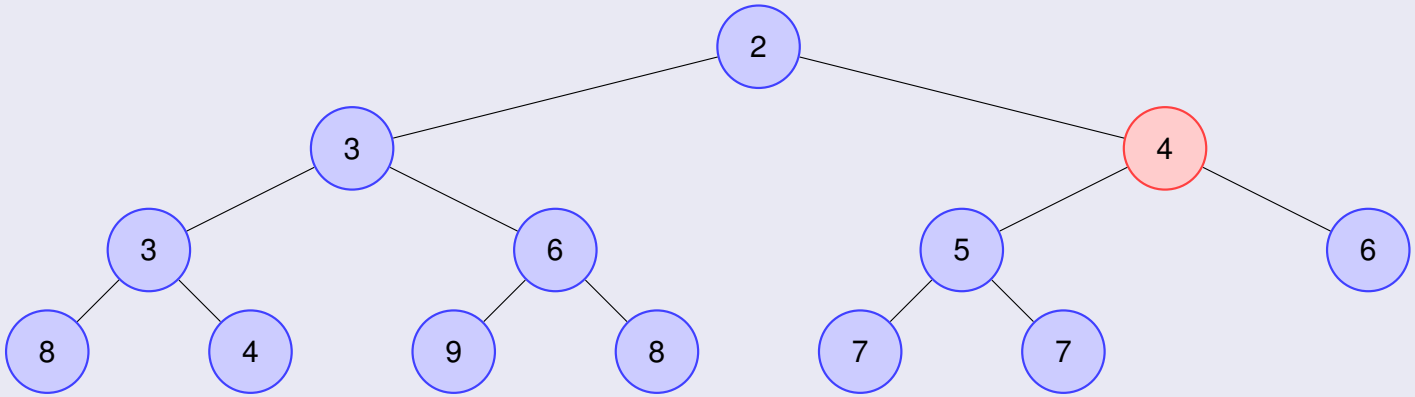


Jak dodać element do kopca?

1. dodajemy element na koniec tablicy (własność kopca może być zaburzona)
2. dopóki ojciec nowego elementu jest od niego większy, to zamieniamy wartości w obu wierzchołkach

Dodawanie elementu

Kopiec



Jak dodać element do kopca?

1. dodajemy element na koniec tablicy (własność kopca może być zaburzona)
2. dopóki ojciec nowego elementu jest od niego większy, to zamieniamy wartości w obu wierzchołkach

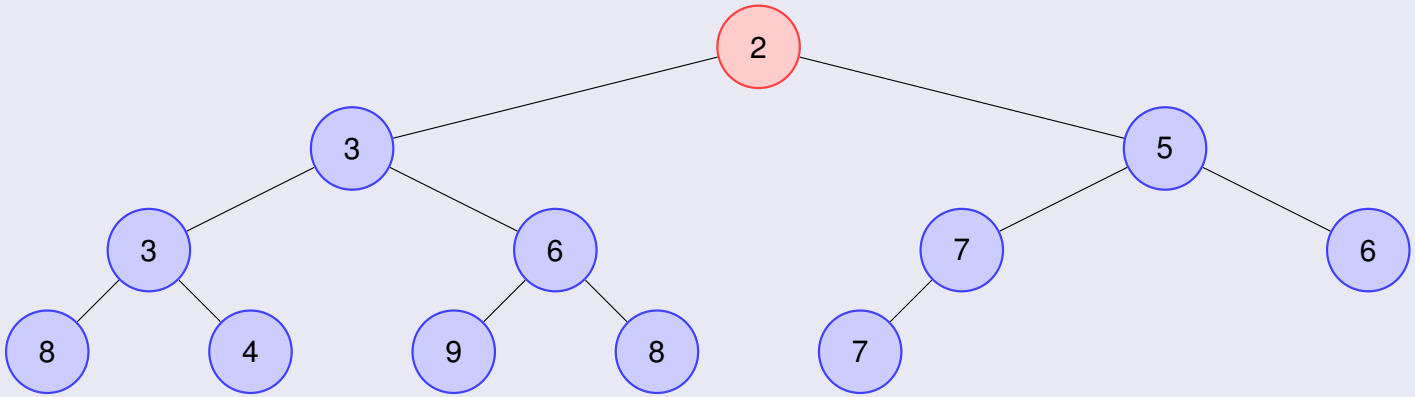
Jak dodać element do kopca?

HeapUp

Proces przenoszenia elementu kopca coraz wyżej w celu przywrócenia własności kopca nazywamy `kopcowaniem w górę (HeapUp)`. W notatkach znajduje się implementacja wstawiania elementu do kopca.

Szukanie minimum

Kopiec

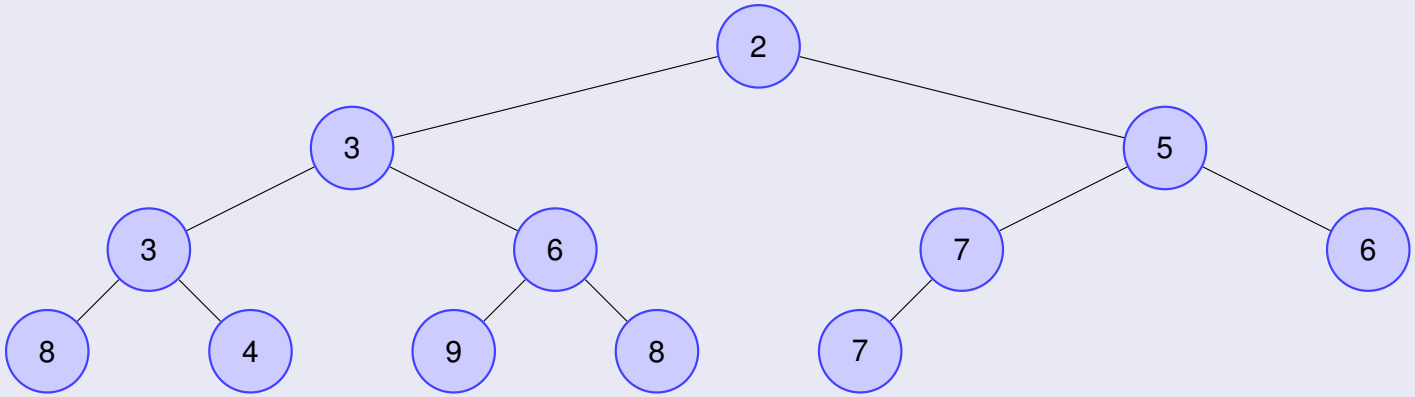


Jak znaleźć minimum w kopcu?

Minimum znajduje się oczywiście z korzeniu, czyli w *heap*[1].

Usuwanie korzenia

Kopiec

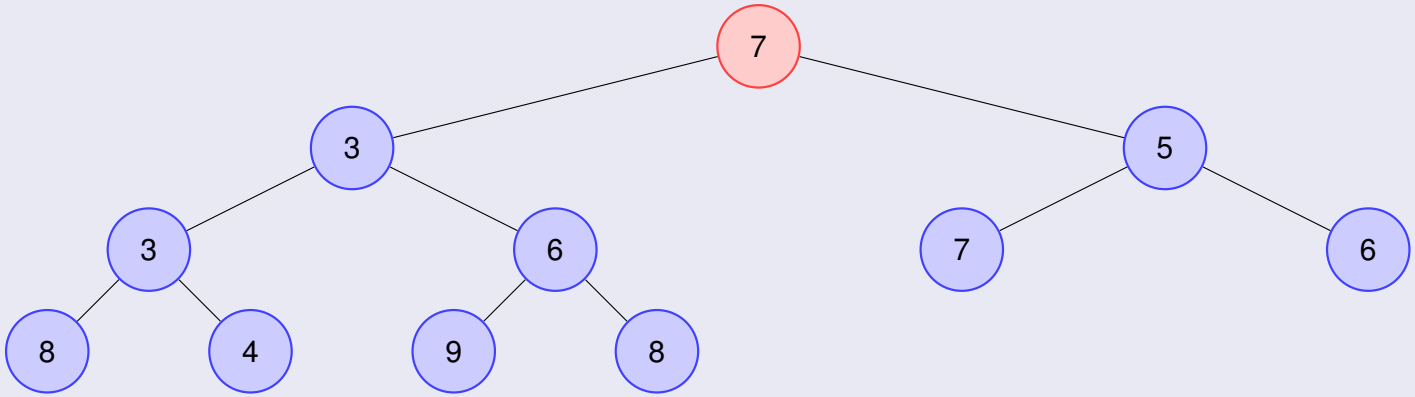


Jak usunąć korzeń?

1. w miejsce korzenia wstawiamy ostatni element (własność kopca może być zaburzona)
2. dopóki wstawiony element jest większy od któregoś ze swoich synów to zamieniamy go z synem o mniejszej wartości

Usuwanie korzenia

Kopiec

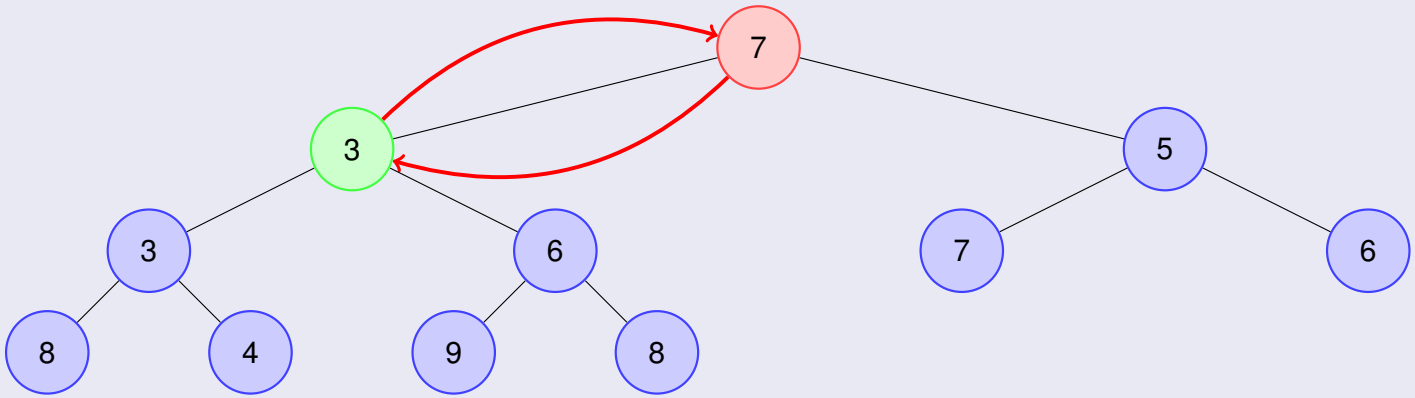


Jak usunąć korzeń?

1. w miejsce korzenia wstawiamy ostatni element (własność kopca może być zaburzona)
2. dopóki wstawiony element jest większy od któregoś ze swoich synów to zamieniamy go z synem o mniejszej wartości

Usuwanie korzenia

Kopiec

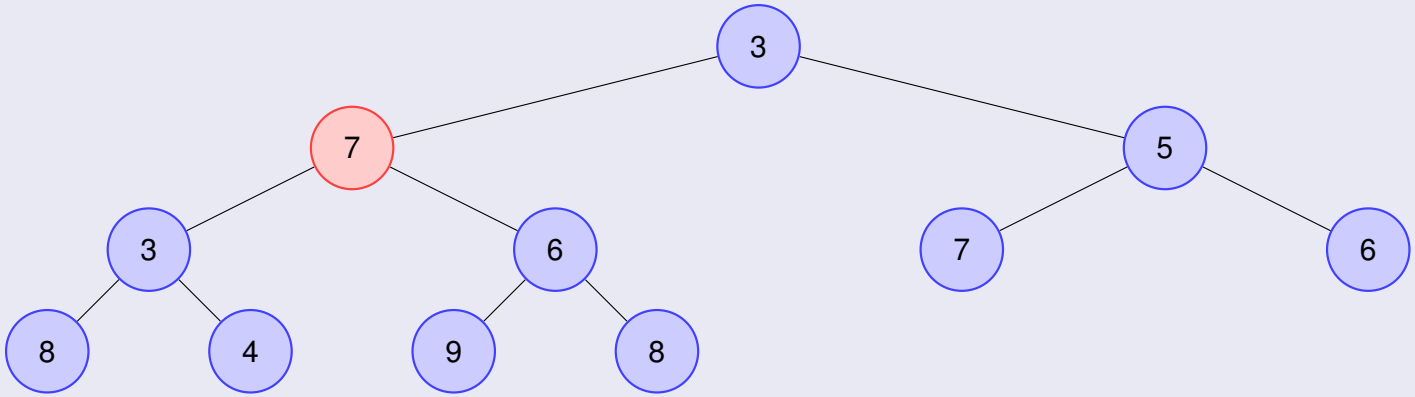


Jak usunąć korzeń?

1. w miejsce korzenia wstawiamy ostatni element (własność kopca może być zaburzona)
2. dopóki wstawiony element jest większy od któregoś ze swoich synów to zamieniamy go z synem o mniejszej wartości

Usuwanie korzenia

Kopiec

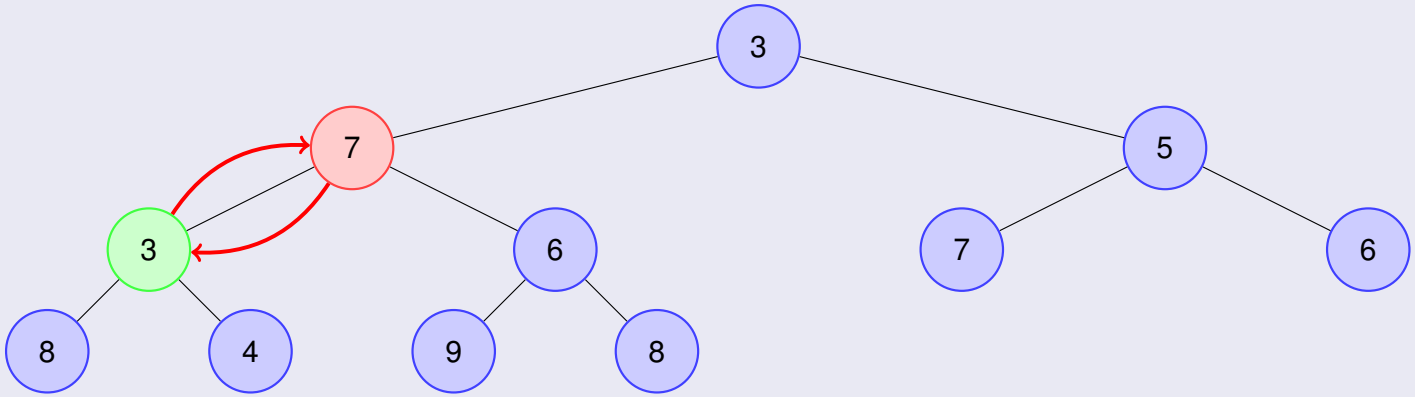


Jak usunąć korzeń?

1. w miejsce korzenia wstawiamy ostatni element (własność kopca może być zaburzona)
2. dopóki wstawiony element jest większy od któregoś ze swoich synów to zamieniamy go z synem o mniejszej wartości

Usuwanie korzenia

Kopiec

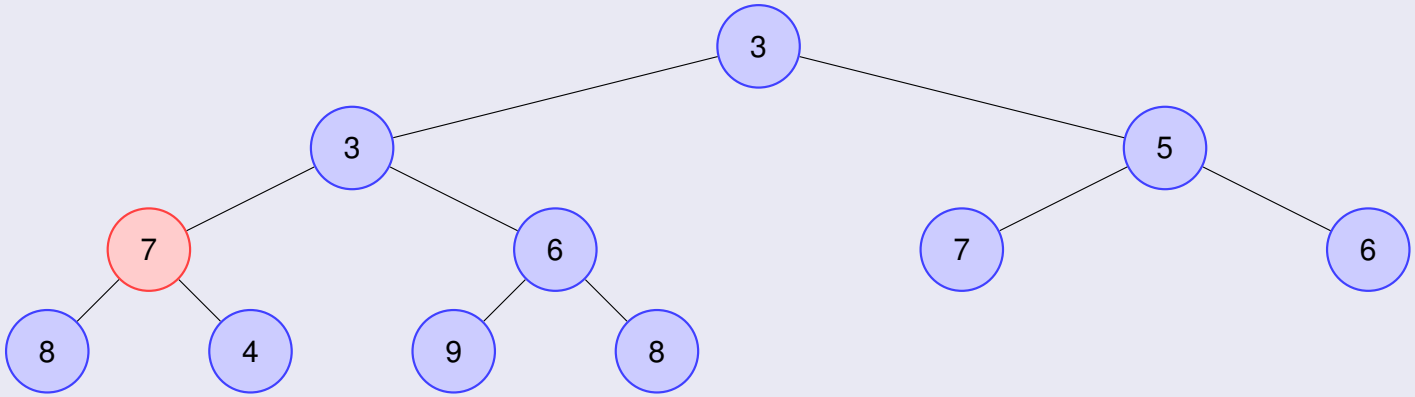


Jak usunąć korzeń?

1. w miejsce korzenia wstawiamy ostatni element (własność kopca może być zaburzona)
2. dopóki wstawiony element jest większy od któregoś ze swoich synów to zamieniamy go z synem o mniejszej wartości

Usuwanie korzenia

Kopiec

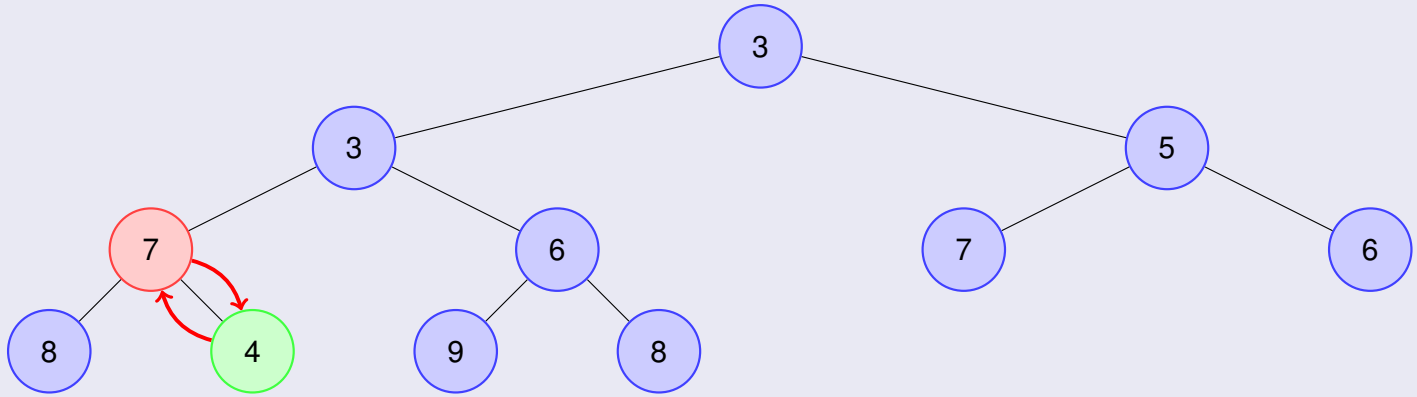


Jak usunąć korzeń?

1. w miejsce korzenia wstawiamy ostatni element (własność kopca może być zaburzona)
2. dopóki wstawiony element jest większy od któregoś ze swoich synów to zamieniamy go z synem o mniejszej wartości

Usuwanie korzenia

Kopiec

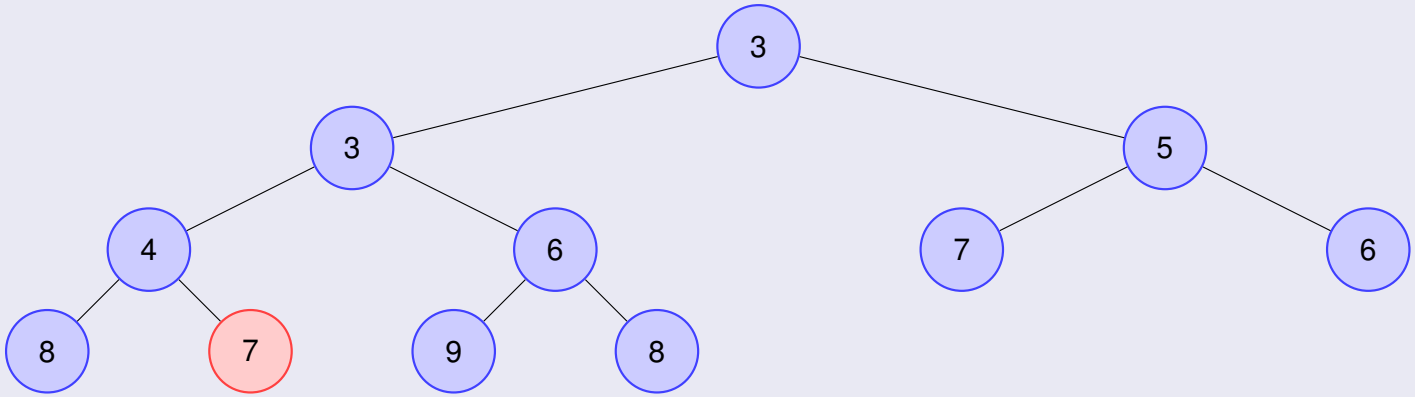


Jak usunąć korzeń?

1. w miejsce korzenia wstawiamy ostatni element (własność kopca może być zaburzona)
2. dopóki wstawiony element jest większy od któregoś ze swoich synów to zamieniamy go z synem o mniejszej wartości

Usuwanie korzenia

Kopiec



Jak usunąć korzeń?

1. w miejsce korzenia wstawiamy ostatni element (własność kopca może być zaburzona)
2. dopóki wstawiony element jest większy od któregoś ze swoich synów to zamieniamy go z synem o mniejszej wartości

Jak usunąć korzeń?

HeapDown

Proces przenoszenia elementu kopca coraz niżej w celu przywrócenia własności kopca nazywamy `kopcowaniem w dół (HeapDown)`. W notatkach znajduje się implementacja usuwania korzenia z kopca.

Usuwanie dowolnego elementu

Problem

W jaki sposób możemy usunąć element na danej pozycji (niekoniecznie korzeń)?

Rozwiązanie

Postąpimy podobnie jak w przypadku korzenia:

1. w miejsce usuwanego elementu wstawiamy ostatni element
2. kopcujemy nowy element w górę
3. kopcujemy nowy element w dół

Usuwanie dowolnego elementu

Problem

W jaki sposób możemy usunąć element na danej pozycji (niekoniecznie korzeń)?

Rozwiązanie

Postąpimy podobnie jak w przypadku korzenia:

1. w miejsce usuwanego elementu wstawiamy ostatni element
2. kopcujemy nowy element w górę
3. kopcujemy nowy element w dół

Usuwanie dowolnego elementu

Problem

W jaki sposób możemy usunąć element na danej pozycji (niekoniecznie korzeń)?

Rozwiązanie

Postąpimy podobnie jak w przypadku korzenia:

1. w miejsce usuwanego elementu wstawiamy ostatni element
2. kopcujemy nowy element w górę
3. kopcujemy nowy element w dół

Usuwanie dowolnego elementu

Problem

W jaki sposób możemy usunąć element na danej pozycji (niekoniecznie korzeń)?

Rozwiązanie

Postąpimy podobnie jak w przypadku korzenia:

1. w miejsce usuwanego elementu wstawiamy ostatni element
2. kopcujemy nowy element w górę
3. kopcujemy nowy element w dół

Usuwanie dowolnego elementu

Problem

W jaki sposób możemy usunąć element na danej pozycji (niekoniecznie korzeń)?

Rozwiązanie

Postąpimy podobnie jak w przypadku korzenia:

1. w miejsce usuwanego elementu wstawiamy ostatni element
2. kopcujemy nowy element w górę
3. kopcujemy nowy element w dół

Analiza złożoności

Analiza złożoności

Wszystkie opisane operacje działają w czasie proporcjonalnym do wysokości kopca, która jest logarytmiczna względem ilości elementów w kopcu.

Sortowanie przez kopcowanie (heapsort)

Problem

Jak można wykorzystać kopiec do uzyskania szybkiego — $O(n \log n)$ algorytmu sortowania?

Rozwiązanie

Elementy tablicy, którą chcemy posortować wrzucamy do kopca a następnie dopóki kopiec nie jest pusty, to wypisujemy minimum i je usuwamy.

Sortowanie przez kopcowanie (heapsort)

Problem

Jak można wykorzystać kopiec do uzyskania szybkiego — $O(n \log n)$ algorytmu sortowania?

Rozwiązanie

Elementy tablicy, którą chcemy posortować wrzucamy do kopca a następnie dopóki kopiec nie jest pusty, to wypisujemy minimum i je usuwamy.

Kopiec a algorytm Dijkstry

Inne zastosowanie kopca

Jednym z najczęściej spotykanych zastosowań kopca jest użycie go do przyspieszenia algorytmu Dijkstry. Na początku przypomnimy ten algorytm.

Kopiec a algorytm Dijkstry

Algorytm Dijkstry

1. oznacz wszystkie wierzchołki jako nieodwiedzone
2. dla każdego $v \in V$ ustaw $dis[v] = \infty$
3. ustaw $dis[s] = 0$
4. dopóki istnieje nieodwiedzony wierzchołek o skończonej odległości:
   1. niech v będzie wierzchołkiem nieodwiedzonym o najmniejszej odległości
   2. oznacz v jako odwiedzony
   3. zrelaksuj wszystkie krawędzie wychodzące z v

wąskie gardło: tą operację wykonujemy $O(|V|)$ razy a dotychczas zabierała ona czas $O(|V|)$

Kopiec a algorytm Dijkstry

Algorytm Dijkstry

1. oznacz wszystkie wierzchołki jako nieodwiedzone
2. dla każdego $v \in V$ ustaw $dis[v] = \infty$
3. ustaw $dis[s] = 0$
4. dopóki istnieje nieodwiedzony wierzchołek o skończonej odległości:
   1. niech v będzie wierzchołkiem nieodwiedzonym o najmniejszej odległości
   2. oznacz v jako odwiedzony
   3. zrelaksuj wszystkie krawędzie wychodzące z v

wąskie gardło: tą operację wykonujemy $O(|V|)$ razy a dotychczas zabierała ona czas $O(|V|)$

Kopiec a algorytm Dijkstry

Algorytm Dijkstry

1. oznacz wszystkie wierzchołki jako nieodwiedzone
2. dla każdego $v \in V$ ustaw $dis[v] = \infty$
3. ustaw $dis[s] = 0$
4. dopóki istnieje nieodwiedzony wierzchołek o skończonej odległości:
   1. niech v będzie wierzchołkiem nieodwiedzonym o najmniejszej odległości
   2. oznacz v jako odwiedzony
   3. zrelaksuj wszystkie krawędzie wychodzące z v

wąskie gardło: tą operację wykonujemy $O(|V|)$ razy a dotychczas zabierała ona czas $O(|V|)$

Kopiec a algorytm Dijkstry

Algorytm Dijkstry

1. oznacz wszystkie wierzchołki jako nieodwiedzone
2. dla każdego $v \in V$ ustaw $dis[v] = \infty$
3. ustaw $dis[s] = 0$
4. dopóki istnieje nieodwiedzony wierzchołek o skończonej odległości:
   1. niech v będzie wierzchołkiem nieodwiedzonym o najmniejszej odległości
   2. oznacz v jako odwiedzony
   3. zrelaksuj wszystkie krawędzie wychodzące z v

wąskie gardło: tą operację wykonujemy $O(|V|)$ razy a dotychczas zabierała ona czas $O(|V|)$

Kopiec a algorytm Dijkstry

Algorytm Dijkstry

1. oznacz wszystkie wierzchołki jako nieodwiedzone
2. dla każdego $v \in V$ ustaw $dis[v] = \infty$
3. ustaw $dis[s] = 0$
4. dopóki istnieje nieodwiedzony wierzchołek o skończonej odległości:
   1. niech v będzie wierzchołkiem nieodwiedzonym o najmniejszej odległości
   2. oznacz v jako odwiedzony
   3. zrelaksuj wszystkie krawędzie wychodzące z v

wąskie gardło: tą operację wykonujemy $O(|V|)$ razy a dotychczas zabierała ona czas $O(|V|)$

Kopiec a algorytm Dijkstry

Algorytm Dijkstry

1. oznacz wszystkie wierzchołki jako nieodwiedzone
2. dla każdego $v \in V$ ustaw $dis[v] = \infty$
3. ustaw $dis[s] = 0$
4. dopóki istnieje nieodwiedzony wierzchołek o skończonej odległości:
   1. niech v będzie wierzchołkiem nieodwiedzonym o najmniejszej odległości
   2. oznacz v jako odwiedzony
   3. zrelaksuj wszystkie krawędzie wychodzące z v

wąskie gardło: tą operację wykonujemy $O(|V|)$ razy a dotychczas zabierała ona czas $O(|V|)$

Kopiec a algorytm Dijkstry

Algorytm Dijkstry

1. oznacz wszystkie wierzchołki jako nieodwiedzone
2. dla każdego $v \in V$ ustaw $dis[v] = \infty$
3. ustaw $dis[s] = 0$
4. dopóki istnieje nieodwiedzony wierzchołek o skończonej odległości:
   1. niech v będzie wierzchołkiem nieodwiedzonym o najmniejszej odległości
   2. oznacz v jako odwiedzony
   3. zrelaksuj wszystkie krawędzie wychodzące z v

wąskie gardło: tą operację wykonujemy $O(|V|)$ razy a dotychczas zabierała ona czas $O(|V|)$

Kopiec a algorytm Dijkstry

Algorytm Dijkstry

1. oznacz wszystkie wierzchołki jako nieodwiedzone
2. dla każdego $v \in V$ ustaw $dis[v] = \infty$
3. ustaw $dis[s] = 0$
4. dopóki istnieje nieodwiedzony wierzchołek o skończonej odległości:
   1. niech v będzie wierzchołkiem nieodwiedzonym o najmniejszej odległości
   2. oznacz v jako odwiedzony
   3. zrelaksuj wszystkie krawędzie wychodzące z v

wąskie gardło: tą operację wykonujemy $O(|V|)$ razy a dotychczas zabierała ona czas $O(|V|)$

Kopiec a algorytm Dijkstry

Algorytm Dijkstry

1. oznacz wszystkie wierzchołki jako nieodwiedzone
2. dla każdego $v \in V$ ustaw $dis[v] = \infty$
3. ustaw $dis[s] = 0$
4. dopóki istnieje nieodwiedzony wierzchołek o skończonej odległości:
   1. niech v będzie wierzchołkiem nieodwiedzonym o najmniejszej odległości
   2. oznacz v jako odwiedzony
   3. zrelaksuj wszystkie krawędzie wychodzące z v

wąskie gardło: tą operację wykonujemy $O(|V|)$ razy a dotychczas zabierała ona czas $O(|V|)$

Kopiec a algorytm Dijkstry

Jak to przyspieszyć?

- chcielibyśmy trzymać w kopcu wierzchołki nieodwiedzone i móc szybko wyciągać z niego wierzchołek x o najmniejszej wartości $dis[x]$
- możemy trzymać w kopcu numery wierzchołków grafu
- przy kopcowaniu musimy wówczas porównywać odpowiednie wartości w tablicy *dis* a nie bezpośrednio wartości trzymane w kopcu ($dis[x]$ będziemy nazywać priorytetem wartości x)
- problem: priorytety mogą ulegać zmianie (ale tylko maleć), co może zaburzyć własność kopca

Kopiec a algorytm Dijkstry

Jak to przyspieszyć?

- chcielibyśmy trzymać w kopcu wierzchołki nieodwiedzone i móc szybko wyciągać z niego wierzchołek x o najmniejszej wartości $dis[x]$
- możemy trzymać w kopcu numery wierzchołków grafu
- przy kopcowaniu musimy wówczas porównywać odpowiednie wartości w tablicy *dis* a nie bezpośrednio wartości trzymane w kopcu ($dis[x]$ będziemy nazywać priorytetem wartości x)
- problem: priorytety mogą ulegać zmianie (ale tylko maleć), co może zaburzyć własność kopca

Kopiec a algorytm Dijkstry

Jak to przyspieszyć?

- chcielibyśmy trzymać w kopcu wierzchołki nieodwiedzone i móc szybko wyciągać z niego wierzchołek x o najmniejszej wartości $dis[x]$
- możemy trzymać w kopcu numery wierzchołków grafu
- przy kopcowaniu musimy wówczas porównywać odpowiednie wartości w tablicy *dis* a nie bezpośrednio wartości trzymane w kopcu ($dis[x]$ będziemy nazywać priorytetem wartości x)
- problem: priorytety mogą ulegać zmianie (ale tylko maleć), co może zaburzyć własność kopca

Kopiec a algorytm Dijkstry

Jak to przyspieszyć?

- chcielibyśmy trzymać w kopcu wierzchołki nieodwiedzone i móc szybko wyciągać z niego wierzchołek x o najmniejszej wartości $dis[x]$
- możemy trzymać w kopcu numery wierzchołków grafu
- przy kopcowaniu musimy wówczas porównywać odpowiednie wartości w tablicy *dis* a nie bezpośrednio wartości trzymane w kopcu ($dis[x]$ będziemy nazywać priorytetem wartości x)
- problem: priorytety mogą ulegać zmianie (ale tylko maleć), co może zaburzyć własność kopca

Jak znaleźć element w kopcu?

Problem

Za każdym razem, gdy priorytet jakiegoś elementu się zmniejszy musimy wykonać kopcowanie tego elementu w górę. Umiemy jednak wykonywać kopcowanie jedynie elementu na danej pozycji (a nie o danej wartości). Jak rozwiązać ten problem?

Rozwiązanie

Oprócz tablicy *heap*, będziemy trzymali tablicę *where*, gdzie $where[x]$ oznacza pozycję w kopcu wartości x, tzn. $heap[where[x]] = x$. Przyjmujemy $where[x] = 0$, jeśli w kopcu nie ma elementu x. Tablicę tę musimy uaktualniać za każdym razem, gdy przemieszczamy elementy w kopcu.

Jak znaleźć element w kopcu?

Problem

Za każdym razem, gdy priorytet jakiegoś elementu się zmniejszy musimy wykonać kopcowanie tego elementu w górę. Umiemy jednak wykonywać kopcowanie jedynie elementu na danej pozycji (a nie o danej wartości). Jak rozwiązać ten problem?

Rozwiązanie

Oprócz tablicy *heap*, będziemy trzymali tablicę *where*, gdzie $where[x]$ oznacza pozycję w kopcu wartości x, tzn. $heap[where[x]] = x$. Przyjmujemy $where[x] = 0$, jeśli w kopcu nie ma elementu x. Tablicę tę musimy uaktualniać za każdym razem, gdy przemieszczamy elementy w kopcu.

Algorytm Dijkstry z kopcem

Analiza złożoności

- nieodwiedzony wierzchołek o najmniejszej odległości możemy znaleźć teraz w czasie $O(\log |V|)$
- relaksacja krawędzi zajmuje teraz czas $O(\log |V|)$, więc relaksacja wszystkich krawędzi zajmuje czas $O(|E| \log |V|)$ i taka też jest złożoność całego algorytmu

Algorytm Dijkstry z kopcem

Analiza złożoności

- nieodwiedzony wierzchołek o najmniejszej odległości możemy znaleźć teraz w czasie $O(\log |V|)$
- relaksacja krawędzi zajmuje teraz czas $O(\log |V|)$, więc relaksacja wszystkich krawędzi zajmuje czas $O(|E| \log |V|)$ i taka też jest złożoność całego algorytmu

Wady takiego rozwiązania

Wady:

- musimy zaimplementować kopiec z operacją zmiany priorytetu, co może zająć dość dużo czasu
- gotowe implementacje kopca, o których dowiesz się na jednych z następnych zajęć, nie posiadają tej operacji

Wady takiego rozwiązania

Wady:

- musimy zaimplementować kopiec z operacją zmiany priorytetu, co może zająć dość dużo czasu
- gotowe implementacje kopca, o których dowiesz się na jednych z następnych zajęć, nie posiadają tej operacji

Inne rozwiązanie

Inne rozwiązanie

- przedstawimy teraz rozwiązanie, które korzysta ze zwykłego kopca (bez operacji zmiany priorytetu)
- do kopca będziemy wrzucać pary $(dis[x], x)$
- przyjmujemy porządek leksykograficzny na parach, tzn. $(x, y) < (a, b)$ wtw. $x < a \vee (x = a \wedge y < b)$
- najmniejszy element w kopcu, odpowiada wówczas wierzchołkowi o najmniejszej odległości

Inne rozwiązanie

Inne rozwiązanie

- przedstawimy teraz rozwiązanie, które korzysta ze zwykłego kopca (bez operacji zmiany priorytetu)
- do kopca będziemy wrzucać pary $(dis[x], x)$
- przyjmujemy porządek leksykograficzny na parach, tzn. $(x, y) < (a, b)$ wtw. $x < a \vee (x = a \wedge y < b)$
- najmniejszy element w kopcu, odpowiada wówczas wierzchołkowi o najmniejszej odległości

Inne rozwiązanie

Inne rozwiązanie

- przedstawimy teraz rozwiązanie, które korzysta ze zwykłego kopca (bez operacji zmiany priorytetu)
- do kopca będziemy wrzucać pary $(dis[x], x)$
- przyjmujemy porządek leksykograficzny na parach, tzn. $(x, y) < (a, b)$ wtw. $x < a \vee (x = a \wedge y < b)$
- najmniejszy element w kopcu, odpowiada wówczas wierzchołkowi o najmniejszej odległości

Inne rozwiązanie

Inne rozwiązanie

- przedstawimy teraz rozwiązanie, które korzysta ze zwykłego kopca (bez operacji zmiany priorytetu)
- do kopca będziemy wrzucać pary $(dis[x], x)$
- przyjmujemy porządek leksykograficzny na parach, tzn. $(x, y) < (a, b)$ wtw. $x < a \vee (x = a \wedge y < b)$
- najmniejszy element w kopcu, odpowiada wówczas wierzchołkowi o najmniejszej odległości

Problem zmiany priorytetu

Problem

Co zrobić jeśli wartość $dis[x]$ się zmieni?

Rozwiązanie

Wrzucamy do kopca nową parę $(dis[x], x)$. Przy takim podejściu w kopcu będą znajdowały się śmieci — nieaktualne pary postaci (d, x), gdzie $d > dis[x]$. Pary takie po prostu pomijamy przy wyciąganiu ich z kopca.

Problem zmiany priorytetu

Problem

Co zrobić jeśli wartość $dis[x]$ się zmieni?

Rozwiązanie

Wrzucamy do kopca nową parę $(dis[x], x)$. Przy takim podejściu w kopcu będą znajdowały się śmieci — nieaktualne pary postaci (d, x), gdzie $d > dis[x]$. Pary takie po prostu pomijamy przy wyciąganiu ich z kopca.

Nowa wersja algorytmu Dijkstry

Algorytm Dijkstry z kopcem:

1. dla każdego $v \in V$ ustaw $dis[v] = \infty$
2. ustaw $dis[s] = 0$
3. wrzuć do kopca parę $(0, s)$
4. dopóki kopiec nie jest pusty
   1. wyciągnij najmniejszą parę z kopca, oznaczmy ją (d, v)
   2. jeśli $d > dis[v]$ to powróć do pkt. 4
   3. zrelaksuj wszystkie krawędzie wychodzące z v, jeśli poprawia to odległość do wierzchołka x, to wrzuć parę $(dis[x], x)$ do kopca

Nowa wersja algorytmu Dijkstry

Algorytm Dijkstry z kopcem:

1. dla każdego $v \in V$ ustaw $dis[v] = \infty$
2. ustaw $dis[s] = 0$
3. wrzuć do kopca parę $(0, s)$
4. dopóki kopiec nie jest pusty
   1. wyciągnij najmniejszą parę z kopca, oznaczmy ją (d, v)
   2. jeśli $d > dis[v]$ to powróć do pkt. 4
   3. zrelaksuj wszystkie krawędzie wychodzące z v, jeśli poprawia to odległość do wierzchołka x, to wrzuć parę $(dis[x], x)$ do kopca

Nowa wersja algorytmu Dijkstry

Algorytm Dijkstry z kopcem:

1. dla każdego $v \in V$ ustaw $dis[v] = \infty$
2. ustaw $dis[s] = 0$
3. wrzuć do kopca parę $(0, s)$
4. dopóki kopiec nie jest pusty
   1. wyciągnij najmniejszą parę z kopca, oznaczmy ją (d, v)
   2. jeśli $d > dis[v]$ to powróć do pkt. 4
   3. zrelaksuj wszystkie krawędzie wychodzące z v, jeśli poprawia to odległość do wierzchołka x, to wrzuć parę $(dis[x], x)$ do kopca

Nowa wersja algorytmu Dijkstry

Algorytm Dijkstry z kopcem:

1. dla każdego $v \in V$ ustaw $dis[v] = \infty$
2. ustaw $dis[s] = 0$
3. wrzuć do kopca parę $(0, s)$
4. dopóki kopiec nie jest pusty
   1. wyciągnij najmniejszą parę z kopca, oznaczmy ją (d, v)
   2. jeśli $d > dis[v]$ to powróć do pkt. 4
   3. zrelaksuj wszystkie krawędzie wychodzące z v, jeśli poprawia to odległość do wierzchołka x, to wrzuć parę $(dis[x], x)$ do kopca

Nowa wersja algorytmu Dijkstry

Algorytm Dijkstry z kopcem:

1. dla każdego $v \in V$ ustaw $dis[v] = \infty$
2. ustaw $dis[s] = 0$
3. wrzuć do kopca parę $(0, s)$
4. dopóki kopiec nie jest pusty
   1. wyciągnij najmniejszą parę z kopca, oznaczmy ją (d, v)
   2. jeśli $d > dis[v]$ to powróć do pkt. 4
   3. zrelaksuj wszystkie krawędzie wychodzące z v, jeśli poprawia to odległość do wierzchołka x, to wrzuć parę $(dis[x], x)$ do kopca

Nowa wersja algorytmu Dijkstry

Algorytm Dijkstry z kopcem:

1. dla każdego $v \in V$ ustaw $dis[v] = \infty$
2. ustaw $dis[s] = 0$
3. wrzuć do kopca parę $(0, s)$
4. dopóki kopiec nie jest pusty
   1. wyciągnij najmniejszą parę z kopca, oznaczmy ją (d, v)
   2. jeśli $d > dis[v]$ to powróć do pkt. 4
   3. zrelaksuj wszystkie krawędzie wychodzące z v, jeśli poprawia to odległość do wierzchołka x, to wrzuć parę $(dis[x], x)$ do kopca

Nowa wersja algorytmu Dijkstry

Algorytm Dijkstry z kopcem:

1. dla każdego $v \in V$ ustaw $dis[v] = \infty$
2. ustaw $dis[s] = 0$
3. wrzuć do kopca parę $(0, s)$
4. dopóki kopiec nie jest pusty
   1. wyciągnij najmniejszą parę z kopca, oznaczmy ją (d, v)
   2. jeśli $d > dis[v]$ to powróć do pkt. 4
   3. zrelaksuj wszystkie krawędzie wychodzące z v, jeśli poprawia to odległość do wierzchołka x, to wrzuć parę $(dis[x], x)$ do kopca

Nowa wersja algorytmu Dijkstry

Algorytm Dijkstry z kopcem:

1. dla każdego $v \in V$ ustaw $dis[v] = \infty$
2. ustaw $dis[s] = 0$
3. wrzuć do kopca parę $(0, s)$
4. dopóki kopiec nie jest pusty
   1. wyciągnij najmniejszą parę z kopca, oznaczmy ją (d, v)
   2. jeśli $d > dis[v]$ to powróć do pkt. 4
   3. zrelaksuj wszystkie krawędzie wychodzące z v, jeśli poprawia to odległość do wierzchołka x, to wrzuć parę $(dis[x], x)$ do kopca

Analiza algorytmu

Analiza algorytmu

- w tym przypadku rozmiar kopca wynosi $O(|E|)$ a nie $O(|V|)$
- nie wpływa to jednak na złożoność, gdyż wynosi ona $O(|E| \log |E|) = O(|E| \log |V|^2) = O(2|E| \log |V|) = O(|E| \log |V|)$
- jest on nieznacznie prostszy w implementacji niż poprzedni
- nie wymaga kopca z operacją zmiany priorytetu

Analiza algorytmu

Analiza algorytmu

- w tym przypadku rozmiar kopca wynosi $O(|E|)$ a nie $O(|V|)$
- nie wpływa to jednak na złożoność, gdyż wynosi ona $O(|E| \log |E|) = O(|E| \log |V|^2) = O(2|E| \log |V|) = O(|E| \log |V|)$
- jest on nieznacznie prostszy w implementacji niż poprzedni
- nie wymaga kopca z operacją zmiany priorytetu

Analiza algorytmu

Analiza algorytmu

- w tym przypadku rozmiar kopca wynosi $O(|E|)$ a nie $O(|V|)$
- nie wpływa to jednak na złożoność, gdyż wynosi ona $O(|E| \log |E|) = O(|E| \log |V|^2) = O(2|E| \log |V|) = O(|E| \log |V|)$
- jest on nieznacznie prostszy w implementacji niż poprzedni
- nie wymaga kopca z operacją zmiany priorytetu

Analiza algorytmu

Analiza algorytmu

- w tym przypadku rozmiar kopca wynosi $O(|E|)$ a nie $O(|V|)$
- nie wpływa to jednak na złożoność, gdyż wynosi ona $O(|E| \log |E|) = O(|E| \log |V|^2) = O(2|E| \log |V|) = O(|E| \log |V|)$
- jest on nieznacznie prostszy w implementacji niż poprzedni
- nie wymaga kopca z operacją zmiany priorytetu