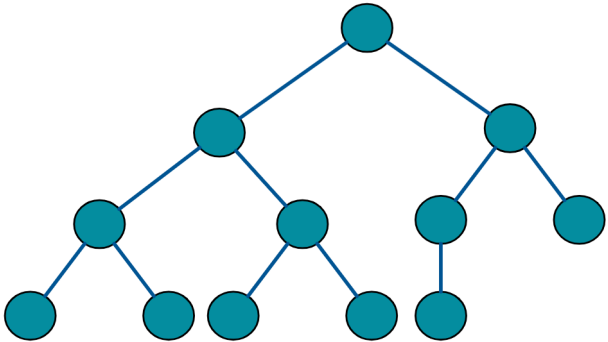
Kopce

Drzewo binarne T nazywamy **zupełnym** (complete tree), gdy na każdym *poziomie* (wyłączając być może ostatni) T ma wszystkie możliwe węzły, a węzły na ostatnim poziomie są możliwie najbardziej z lewej strony.



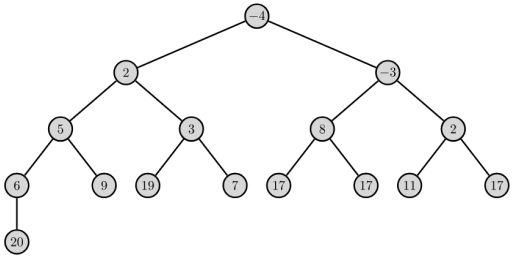
Wysokość takiego drzewa zależy logarytmicznie od liczby węzłów.

Kopce

Od teraz pozwalamy, aby drzewa mogły być **puste**.
Binarne drzewo zupełne (V, S), którego węzły przechowują parami porównywalne klucze nazywamy **kopcem binarnym** (binary heap), gdy

$$\forall x, y \in V \ \ (xSy \implies \text{klucz w } x \text{ jest nie większy od klucza w } y)\,.$$

Przykład:



Inne nazwy: sterta, stóg, kopiec minimum (min-heap). Jeśli odwrócimy nierówność: kopiec maksimum (max-heap).

Kopce

Operacje na kopcu:

- $\mathrm{insert\_key}(\mathrm{key})$: włóż nowy węzeł o zadanym kluczu.
- $\mathrm{extract\_min}()$: usuń węzeł z minimalnym kluczem i zwróć wartość klucza.
- (opcjonalnie) $\mathrm{decrease\_key}(\mathrm{key}, \mathrm{new\_value})$: zmniejsz wartość podanego klucza.
- (opcjonalnie) $\mathrm{increase\_key}(\mathrm{key}, \mathrm{new\_value})$: zwiększ wartość podanego klucza.

Wszystkie operacje muszą zachować własność kopca: po dodaniu/usunięciu węzła lub zmianie wartości klucza trzeba dokonać „naprawy".

W operacjach $\mathrm{decrease\_key}$, $\mathrm{increase\_key}$ zakładamy, że klucze są unikalne.

Kopce

Rozważmy kopiec, w którym klucz w węźle t zastępujemy innym tak, że niszczymy własność kopca. Są dwie możliwości:

(a) Nowy klucz w t jest mniejszy, niż klucz w rodzicu t, lub

(b) Klucz w lewym lub prawym dziecku t jest mniejszy, niż nowy klucz w t.

Uwaga: Skoro przed operacją podmiany klucza drzewo było kopcem, (a) i (b) nie zajdą jednocześnie.

W sytuacjach (a) i (b) naprawiamy kopiec przez „kopcowanie" odpowiednio „w górę" lub „w dół".

Kopce

Naprawianie kopca (przykłady – tablica).

Kopcowanie w górę: kroki $\mathrm{heapify\_up}(t)$:

1. Jeśli t jest korzeniem kopca, koniec.
2. Jeśli klucz w t jest mniejszy, niż klucz w rodzicu t:
   - 2a. Zamień miejscami klucze w t i rodzicu t
   - 2b. Wróć do kroku 1 zastępując t przez jego rodzica.

Kopcowanie w dół: kroki $\mathrm{heapify\_down}(t)$:

1. Jeśli klucz w t jest nie większy, niż klucz w dowolnym dziecku t, koniec.
2. W przeciwnym wypadku, niech s będzie dzieckiem t o minimalnym kluczu.
   - 2a. Zamień miejscami klucze w s i t.
   - 2b. Wróć do kroku 1 zastępując t przez s.

Kopce

Kroki $\mathrm{insert\_key}$ dla klucza key:

1. Wstaw nowy liść t o kluczu key, tak, aby drzewo wciąż było zupełne.
2. Wykonaj $\mathrm{heapify\_up}$ na t.

Kroki $\mathrm{extract\_min}$:

1. Zapisz klucz z korzenia kopca w x.
2. Przepisz klucz z ostatniego liścia kopca do korzenia i usuń ten liść.
3. Wykonaj $\mathrm{heapify\_down}$ na korzeniu kopca.
4. Zwróć x.

(Przykłady – animacja)

Kopce

Kroki increase_key(key, new_value):

1. Niech t: węzeł z kluczem key. Zastąp klucz w t przez new_value.
2. Wykonaj heapify_down na t.

Kroki decrease_key(key, new_value):

1. Niech t: węzeł z kluczem key. Zastąp klucz w t przez new_value.
2. Wykonaj heapify_up na t.

Złożoność operacji.

Fakt. Operacje heapify_down, heapify_up, insert_key, extract_min, increase_key, decrease_key mają pesymistyczną złożoność $\Theta(\log n)$, gdzie n to ilość węzłów w kopcu.

Dowód: Każdy krok algorytmu kopcowania w górę polega na porównaniu kluczy w pewnych węzłach i dokonaniu zamiany kluczy. Z każdą zamianą, węzeł przeczący własności kopca przesuwa się o jeden poziom w górę, lub algorytm kończy działanie. Kroków będzie zatem nie więcej niż wysokość drzewa. Podobnie dla kopcowania w dół. Wszystkie pozostałe działania na kopcu wykonują stałą liczbę operacji* i jedno wywołanie heapify_down albo heapify_up.

* Stały czas tych operacji uzasadnimy za chwilę.

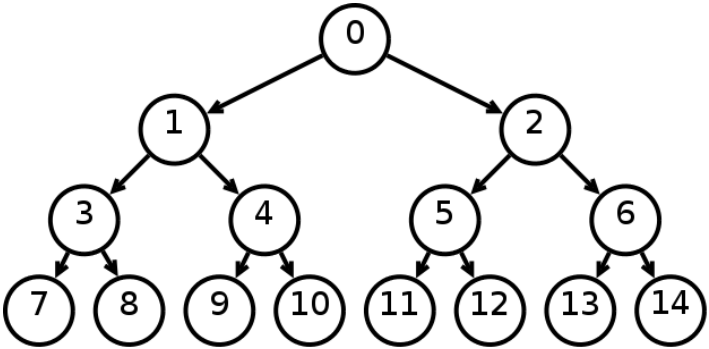
Złożoność operacji, c.d..

Inne operacje na kopcach: tworzenie kopca z listy, łączenie dwóch kopców.

Tworzenie i łączenie: u nas będzie (pesymistycznie) $\Theta(n \log n)$ – metoda Williamsa. Optymalnie: da się w $\Theta(n)$.

Istnieją inne implementacje kopców o lepszych złożonościach, np. *kopiec Fibonacciego*: $\mathrm{extract\_min}$ w czasie $\Theta(\log n)$, pozostałe operacje $\Theta(1)$.

Kopce jako listy.
Zadajemy naturalne numerowanie (indeksowanie) węzłów w zupełnym drzewie binarnym.



Kopce

Indeks x-tego węzła (od lewej) na poziomie y: $i = 2^y + x - 1$. Przyporządkowanie indeksu jest unikalne: w drzewie zupełnym o n węzłach węzły otrzymują indeksy $0, 1, \ldots n-1$.
Z indeksu i można odzyskać x i y.

Jeśli węzeł t ma indeks i, wtedy:

1. Rodzic t (jeśli istnieje) ma indeks $\lfloor (i-1)/2 \rfloor$.
2. Lewe dziecko (jeśli istnieje) ma indeks $2 \cdot i + 1$.
3. Prawe dziecko (jeśli istnieje) ma indeks $2 \cdot i + 2$.

To pozwala na reprezentowanie kopca jako listę.

Zaleta: dostawienie nowego liścia (potrzebne przy $\mathrm{insert\_key}$): to samo, co dostawienie elementu do listy (czas stały – a to uzasadnia dowód złożoności operacji kopcowych).

(Implementacja: Pycharm...)

Zastosowania kopców

Heapsort (sortowanie przez kopcowanie): stwórz kopiec *heap* z listy *lst*, następnie wyciągaj minimalny element z *heap* aż do wyczerpania.

Złożoność: $\Theta(n \log n)$ w przypadkach średnim i pesymistycznym, $\Theta(n)$ w optymistycznym (gdy klucze są równe).

Złożoność pamięciowa: $\Theta(n)$ dla pamiętania kopca. Jeśli wykorzystamy oryginalną listę jako kopiec, jest to sortowanie w miejscu.

Stabilność: NIE!

Bardzo wydajny algorytm, m.in. element Introsorta (hybrydy Quicksorta i Heapsorta).

Zastosowania kopców

Kolejka priorytetowa: struktura danych z operacjami „włóż" i „wyjmij". Każdy element jest wkładany do kolejki wraz z **priorytetem**. Elementy są wyciągane z kolejki zgodnie z ich priorytetami.
Priorytetami mogą być dowolne parami porównywalne wartości (liczby, napisy, etc.).

U nas: najpierw wyjmujemy elementy o **najmniejszym** (sic) priorytecie!

Przykłady kolejek priorytetowych:

1. Kolejka: elementy wyciągamy w kolejności wkładania (priorytet: czas włożenia).
2. Stos: elementy wyciągamy w kolejności odwrotnej (priorytet: minus czas włożenia).

Kolejka priorytetowa, c.d.
Zakładamy, że dane wkładane w kolejce są unikalne i porównywalne (można łatwo pozbyć się tego założenia, np. tworząc klasę przechowującą dane z przeładowanymi operatorami $<$, $==$).
W typowej sytuacji dane będą liczbami, napisami etc.

Implementacja kolejki priorytetowej przez kopiec: kluczami w kopcu są pary $(priorytet, dane)$. Te pary są uporządkowane leksykograficznie i mamy

$$priorytet_1 < priorytet_2 \implies (priorytet_1, dane_1) < (priorytet_2, dane_2).$$

Zatem $\mathrm{extract\_min}$ zawsze wyciąga parę o minimalnym priorytecie.

Algorytm Dijkstry

Najkrótsza ścieżka w grafie ważonym.

Niech $G = (V, E, f)$ będzie grafem skierowanym ważonym. W takim grafie długość ścieżki $(e_1, e_2, \ldots, e_n)$ definiujemy jako $\sum_{i=1}^{n} f(e_i)$ (suma wag krawędzi).

Załóżmy, że wszystkie krawędzie w G mają wagi dodatnie[1] (graf jest **dodatnio ważony**). Zadanie: dla wierzchołków u, v znaleźć najkrótszą ścieżkę z u do v.

Szczególny przypadek: $\forall e \in E \; f(e) = 1$. Wtedy długość ścieżki jest tożsama z długością ścieżki w grafie nieważonym i można zastosować BFS.

[1] W istocie wystarczy założyć tylko nieujemność wag .

Algorytm Dijkstry

Idea algorytmu Dijkstry (u – wierzchołek źródłowy):

1. Trawersujemy graf w pewnej kolejności, odwiedzając wierzchołki. Odwiedzone wierzchołki zaznaczamy.
2. Wierzchołkom w grafie przypisujemy *tymczasową odległość* od wierzchołka początkowego u. Jest ona zawsze nie mniejsza, niż odległośc faktyczna. Początkowo ta odległość wynosi 0 dla u i ∞ dla pozostałych wierzchołków.
3. Odwiedzamy wierzchołki priorytetyzując te o niższej odległości tymczasowej. Gdy odwiedzamy wierzchołek, jego odległość tymczasowa jest równa prawdziwej odległości.
4. Odwiedzając wierzchołek, zmniejszamy odległość tymczasową niektórych jego sąsiadów (tj. *relaksujemy* ją). Po wykonaniu algorytmu odległość tymczasowa równa się faktycznej.

Algorytm Dijkstry

Kroki $\mathrm{dijkstra}$ dla grafu (dodatnio ważonego) G i wierzchołka początkowego u:

1. Oznacz wszystkie wierzchołki jako nieodwiedzone.
2. Przypisz każdemu wierzchołkowi odległość tymczasową ∞ i 0 dla u.
3. Dopóki są nieodwiedzone wierzchołki (o skończonej odległości tymczasowej):
   1. Wybierz nieodwiedzony wierzchołek t o najmniejszej tymczasowej odległości.
   2. Oznacz t jako odwiedzony.
   3. Dla każdego nieodwiedzonego sąsiada t' wierzchołka t:
      1. Niech $l =$ (odległość tymczasowa t) + (waga krawędzi (t, t')).
      2. Jeśli l jest mniejsze niż odległość tymczasowa do t', zmień tę odległość na l.

Śledzenie odleglości tymczasowych i wybór wierzchołka do odwiedzin używa kolejki priorytetowej. (Implementacja – Pycharm)

Algorytm Dijkstry

Fakt: po wykonaniu algorytmu, odległość tymczasowa wierzchołka jest równa odległości faktycznej.
Dowód: skrypt, indukcja względem n, gdzie $n = |B|$, B – zbiór odwiedzonych wierzchołków.

Implementacja algorytmu Dijkstry: używamy kolejki priorytetowej do śledzenia odległości tymczasowych. Używamy $\mathrm{decrease\_key}$ do ich uaktualniania i $\mathrm{extract\_min}$ do pobrania węzła o minimalnej takiej odległości.

Relaksując odległość tymczasową wierzchołka możemy pamiętać, który odwiedzony wierzchołek zaświadczał o tej odległości. Wtedy po wykonaniu algorytmu możemy odtworzyć najkrótszą ścieżkę z u do wybranego wierzchołka.