Notatki do egzaminu teoretycznego z ASD

Koszt zamortyzowany

Koszt zamortyzowany to średni koszt wykonania pojedynczej operacji w zadanym ciągu operacji.

$\overline{c_i} = \frac{C(n)}{n}$, gdzie $\overline{c_i}$ to koszt zamortyzowany.

Metody analizy kosztu zamortyzowanego:

1. Metoda kosztu sumarycznego
   Po prostu obliczamy $C(n)$.
2. Metoda księgowania
   Elementom struktury przypisujemy żetony, z które potem możemy wydawać. Podczas niektórych operacji przypisujemy nowe żetony, zwiększając koszt operacji o liczbę przypisanych żetonów. Podczas innych operacji korzystamy z wcześniej przypisanych żetonów, zmniejszając koszt operacji o liczbę wykorzystanych żetonów. Koszt zamortyzowany operacji jest pesymistycznym kosztem operacji (w który to koszt wliczamy przypisywanie i wydawanie żetonów).
3. Metoda potencjału
   Wymyślamy funkcję potencjału $\Phi(O_i)$, gdzie O_i jest stanem struktury, na której wykonujemy operacje. Φ musi spełniać: $\Phi(O_i) \geqslant \Phi(O_0)$, przy czym najczęściej najwygodniej jest przyjąć, że $\Phi(O_0) = 0$. Okazuje się, że jak Φ to spełnia to nasza definicja
   Jak mamy to zapewnione to wtedy prawdą jest, że $\overline{c_i} \geqslant c_i + \Phi(O_i) - \Phi(O_{i-1})$. A więc chodzi o to, żeby dobrać odpowiednią funkcję potencjału, która zwiększa się przy wykonywaniu mało kosztownych operacji na strukturze, a zmniesza przy wykonywaniu kosztownych operacji.

Sortowanie

1. Algorytmy sortujące

- Insertion sort (sortowanie przez wstawianie)
  - Algorytm w skrócie: dla i-tego przebiegu pętli ciąg na pozycjach [0, i-1] jest posortowany, bierzemy i-ty element i przesuwamy go w lewo na odpowiednią pozycję.
  - Złożoność czasowa: $O(n^2)$
    * liczba porównań = $n - 1 + Inv(a)$, liczba przesunięć = $Inv(a)$, gdzie a to sortowany ciąg.
  - Własności:
    * w miejscu, stabilny, złożoność zależna od liczby inwersji
- Selection sort (sortowanie przez wybieranie)
  - Algorytm w skrócie: dla i-tego przebiegu pętli ciąg na pozycjach [n-i, n] jest posortowany, znajdujemy maksymalny element na pozycjach [0, n-i-1] i zamieniamy go miejscami z elementem na pozycji n-i-1.
  - Złożoność czasowa: $O(n^2)$
  - Własności:
    * w miejscu, mała liczba zamian (zawsze $n - 1$)
    * niestabilny, kwadratowa złożoność *niezależnie od danych*
- **TODO:** metoda Shella (usprawnienie Insertion sorta)
- Heap sort
  - Algorytm w skrócie: budujemy kopiec typu max na tablicy (patrz: Kolejki priorytetowe → Kopiec zupełny), a następnie n razy wykonujemy na nim del_max, wstawiając maksymalne elementy na koniec tablicy.

  - Złożoność czasowa: $O(n \log n)$
  - Własności:
    * w miejscu
    * niestabilny
- Merge sort (sortowanie przez scalanie)

  Podstawowa wersja
  - Algorytm w skrócie: dzielimy ciąg na dwie równe części, rekurencyjnie sortujemy każdą z nich, a następnie je scalamy.
  - Złożoność czasowa: $O(n \log n)$
  - Własności:
    * stabilny, bardzo mało porównań, sporo przypisań
    * nie w miejscu (tablica do scalania + rekursja)

  Wersja w miejscu
  - Algorytm w skrócie:
    * obserwacja: do scalenia dwóch tablic wystarczy pomocniczy bufor o rozmiarze mniejszej z tych tablic (najpierw ze scalonych dwóch tablic zamieniamy zawartość mniejszej z zawartością bufora, a potem wybieramy mniejszy element i albo przesuwamy początkową zawartość bufora w prawo albo zamieniamy liczbę z bufora z elementem z początkowej zawartości bufora)
    * algorytm w skrócie (pierwsze kroki, kolejne analogiczne aż do uzyskanie stałego rozmiaru nieposortowanej części): sortujemy [n/2, n] używając jako bufora [0, n/2], sortujemy [n/4, n/2] buforem [0, n/4], scalamy [n/4, n/2] z [n/2, n] buforem [0, n/4], sortujemy [n/8, n/4] buforem [0, n/8], scalamy [n/8, n/4] z [n/4, n] buforem [0, n/8], itd...
  - Złożoność czasowa: $O(c \cdot n \log n)$ (**TODO:** nwm czym jest c, ale jakaś stała)
  - Własności:
    * w miejscu
    * niestabilny
- Quick sort
  - Algorytm w skrócie: wybieramy z ciągu element dzielący, dzielimy pozostałe elementy na mniejsze i większe od niego, rekurencyjnie je sortujemy i łączymy lewy ciąg, element dzielący i prawy ciąg.
  - partition(l, r): funkcja dzieląca ciąg a[l..r] względem elementu a[l] - algorytm Hoare'a idziemy dwoma wskaźnikami od lewej i prawej strony aż napotkamy z lewej element większy od a[l], a z prawej mniejszy, wtedy zamieniamy je miejscami i kontynuujemy aż wskaźniki się spotkają
  - Złożoność czasowa: *pesymistyczna* - $O\left(n^2\right)$, *oczekiwana* - $\Theta(n \log n)$ (dla losowej permutacji)
  - Właściwości:
    * w miejscu (w podstawowej wersji nie w miejscu, bo rekursja, ale istnieje wersja ze stosem)
    * niestabilny
    * możemy uniknąć pesymistycznego czasu kwadratowego jako element dzielący wybierając medianę
- Count sort (sortowanie przez zliczanie)
  - Algorytm w skrócie: trzymamy tablicę b o wielkości m (= zakres wartości w tablicy wejściowej). Zliczamy ile razy występuje każda wartość w wejściowym ciągu i zapisujemy to w tablicy b. Updatujemy tablicę b, tak żeby $b[i]$ było równe ostatniej pozycji w posortowanym ciągu na której występuje element i. Przechodzimy po elementach wejściowej tablicy i za pomocą tablicy b znajdujemy miejsce w wyjściowej tablicy, na które należy włożyć dany element, odpowiednią wartość w tablicy b dekrementujemy.
  - Złożoność czasowa: $O(n + m)$, gdzie m to zakres wartości ciągu wejściowego. Dla $m = O(n)$ dostajemy złożoność liniową.
  - Własności:
    * stabilny, dla liniowego zakresu wartości liniowy
    * nie w miejscu

- Bucket sort (sortowanie kubełkowe)
  - Algorytm w skrócie: trzymamy m kubełków (czyli list); przechodzimy po wejściowym ciągu i każdy element dodajemy na koniec kubełka który odpowiada jego wartości. Następnie przechodzimy po wszystkich kubełkach od najmniejszej do największej wartości i konkatenujemy je do wynikowej listy.
  - Złożoność czasowa: $O(n+m)$, gdzie m to zakres wartości ciągu wejściowego. Dla $m = O(n)$ dostajemy złożoność liniową.
  - Własności:
    * stabilny
    * nie w miejscu
- Sortowanie leksykograficzne

  Wersja dla słów tej samej długości
  - Algorytm w skrócie: idziemy po słowach od końca i dla każdej pozycji sortujemy stabil słowa leksykograficznie i stabilnie (bucket sortem) według liter na tej pozycji. Po przejściu do początku, słowa są posortowane.
  - Złożoność czasowa: $O(R)$, gdzie $R = kn$ to suma długości słów, pod warunkiem że $m = O(n)$

  Wersja dla słów dowolnej długości
  - Algorytm w skrócie: **TODO** (sprytny preprocessing i bucket sort)
  - Złożoność czasowa: $O(R+m)$ $(= O(R)$ gdy $m = O(R))$

2. Dolna granica na liczbę porównań, statystyki pozycyjne i algorytm piątek

- Dolna granica na liczbę porównań dla sortowania przez porównania: $O(n \log n)$
  - W dowolnym drzewie decyzyjnym o m liściach, najdłuższa ścieżka od korzenia do liścia jest $\geqslant \lceil log(m) \rceil$. $m = n!$, gdy m jest równe liczbie permutacji ciągu n-elementowego.
- Algorytm "magicznych piątek"
  - Jest to algorytm wyznaczający "przybliżoną" medianę zbioru.
  - Algorytm w skrócie: dzielimy ciąg na 5-elementowe tablice (ostatnia może być mniejsza), i wszystkie tablice sortujemy. Wkładamy wszystkie środkowe elementy (mediany) do oddzielnej tablicy i wywołujemy na nim algorytm rekurencyjnie, aż dostaniemy pojedynczy element - naszą przybliżoną medianę.
- Selekcja Hoare'a
  - Jest to algorytm znajdujący k-ty co do wielkości element ciągu.
  - partition-3-way(l, r, k): liniowa funkcja pomocnicza dzieląca elementy tablicy a[l..r] elementy na mniejsze, równe i większe od zadanego elementu tej tablicy (nie wiem jak)
  - Algorytm (selekcja Hoare'a) w skrócie: wybieramy jako element dzielący medianę median z algorytmu "magicznych piątek", za pomocą partition-3-way() dzielimy ciąg względem mediany median. Sprawdzamy, w której części z trzech otrzymanych znajduje się element k-ty co do wielkości - jeśli w środkowej (równej elementowi dzielącemu), to go zwracamy; jeśli w lewej lub w prawej, to rekurencyjnie się na niej wywołujemy.
  - Złożoność czasowa: $O(n)$

Kolejki priorytetowe

- Kopiec zupełny
  - downheap(i: element, l: left subtree of i, r: right subtree of i):
    * zakładamy, że l i r spełniają warunek kopca!
    * schodzimy w dół elementem i wymieniając go z maksimum z jego dzieci, dopóki sam nie jest maksymalny lub jest liściem
    * złożoność czasowa: $O(log\ n)$
  - upheap(i)
    * zakładamy, że kopiec, poza elementem i spełnia warunek kopca

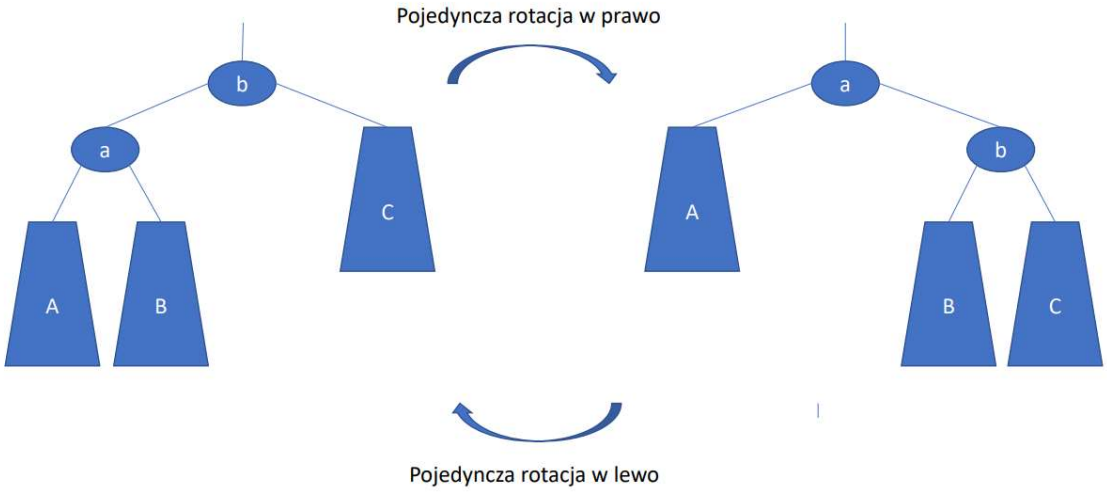
    * przywracamy warunek kopca wymieniając go z rodzicem jeśli jest większy od rodzica, w przeciwnym wypadku, kończymy.
    * złożoność czasowa: $O(log\ n)$
  - build_heap(a)
    * algorytm w skrócie: idziemy po węzłach od dołu do góry (od prawego końca tablicy), na każdym robiąc downheap()
    * złożoność czasowa: $O(n)$ (wychodzi jak obliczymy odpowiednią sumę)
  - del_min()
    * wstawiamy do korzenia (na miejsce minimum) wartość z ostatniego liścia i wykonujemy na niej downheap()
    * złożoność czasowa: $O(log\ n)$
  - insert(x)
    * dodajemy x jako nowego (ostatniego) liścia i wykonujemy na nim upheap()
    * złożoność czasowa: $O(log\ n)$
  - decrease_key(x, new_x)
    * problematyczne, bo trzeba najpierw znaleźć x w kopcu. Zakładając że trzymamy dodatkowo strukturę pozwalającą w czasie $O(log\ n)$ znaleźć miejsce x w kopcu, możemy wykonać decrease_key() zmieniając wartość węzła przechowującego x na new_x, a następnie wykonując na nim upheap().
    * złożoność czasowa: $O(log\ n)$
  - Własności: można go trzymać w tablicy, jeśli korzeń ma numer 1, to dla węzła o numerze i jego lewe dziecko ma numer $2i$, prawe dziecko - $2i + 1$, rodzic - $\lfloor i / 2 \rfloor$
- Kolejka dwumianowa
  - Drzewo dwumianowe B_i to drzewo B_{i-1} z podłączonym pod korzeń drugim drzewem B_{i-1}. B_0 składa się z jednego wierzchołka.
    * Własności: $|B_k| = 2^k$, $h(B_k) = k$, $childrenNum(root(B_k)) = k$
  - Kolejka dwumianowa z n elementami to las złożony z drzew dwumianowych o parami różnych rozmiarach, których łączny rozmiar to n. Elementy w drzewach są rozmieszczone w porządku kopcowym.
    * Dostęp do kolejki przez wskaźnik do korzenia z najmniejszym kluczem.
    * Każdy węzeł trzyma wskaźnik na rodzica, prawego i lewego sąsiada oraz którekolwiek dziecko.
  - join(T_1, T_2)
    * dwa drzewa dwumianowe o tym samym stopniu łączymy podpinając jedno pod korzeń drugiego, i przepinając wskaźniki
    * złożoność czasowa: $O(1)$
  - union(Q_1, Q_2)
    * dwie kolejki dwumianowe łączymy w jedną wykonując dodawanie binarne na ich drzewach dwumianowych (poprzez funkcję join())
    * złożoność czasowa: $O(log|Q_1| + log|Q_2|)$
  - init(Q, $\{e_1, ..., e_k\}$)
    * tworzymy kolejno jednoelementowe kolejki dwumianowe i wykonujemy operacje union()
    * złożoność czasowa: $O(k)$
  - min()
    * złożoność czasowa: $O(1)$
  - insert(Q, e)
    * tworzymy jednoelementową kolejkę z e i wykonujemy union()
    * złożoność czasowa: $O(log\ n)$
  - decrease_key(Q, v, new_key)
    * znajdujemy miejsce elementu v, nadajemu mu new_key i wykonujemy coś podobnego do upheap() dla kopca zupełnego

        * złożoność czasowa: $O(log\ n)$
    - delete_min(Q)
        * usuwamy z kolejki Q drzewo T z najmniejszym kluczem i usuwamy z niego korzeń. Pozostałe poddrzewa T (po usunięciu korzenia) tworzą nową kolejkę dwumianową Q'. Wykonujemy union(Q, Q').
        * złożoność czasowa: $O(log\ n)$
- Leniwa kolejka dwumianowa
    - Podstawowa różnica między leniwą kolejką dwumianową a zwykłą jest taka, że przy leniwej nie mamy wymagania, żeby rozmiary drzew wchodzących w skład kolejki były parami różne. Druga różnica jest taka, że przy operacji delete_min() wykonujemy "czyszczenie", które zbija nam czas zamortyzowany operacji delete_min().
    - złożoności czasowe (z "czyszczeniem"):
        * init_empty(): $O(1)$
        * min(): $O(1)$
        * delete_min: *pesymistyczny* $O(n)$, *zamortyzowany* $O(log\ n)$
        * insert(): *zamortyzowany* $O(1)$
        * decrease_key(): $O(log\ n)$
- Kopiec Fibonacciego
    - Dalsza modyfikacja kolejki dwumianowej, zapewniająca jeszcze lepszą złożoność czasową decrease_key() (jakieśtam ostrożne odcinanie poddrzewa węzła, na którym robimy decrease_key(), kolorowanie wierzchołków, dziwne rzeczy)
    - złożoności czasowe:
        * init_empty(): $O(1)$
        * min(): $O(1)$
        * delete_min(): *zamortyzowany* $O(log\ n)$
        * insert(): *zamortyzowany* $O(1)$
        * decrease_key(): *zamortyzowany* $O(1)$

Słowniki

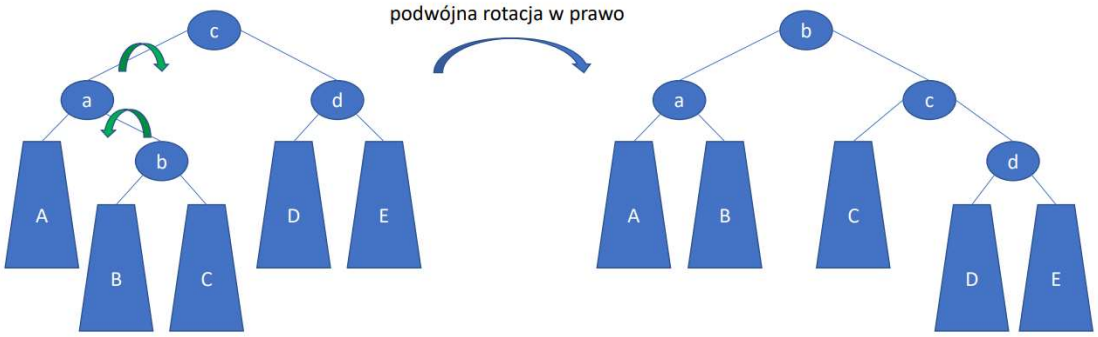
- Wzbogacanie drzew BST
    - Jest to trzymanie w wierzchołkach drzewa BST większej ilości informacji o poddrzewie. Trzeba zadbać o to, żeby na podstawie informacji z wierzchołków będacych korzeniami lewego i prawego poddrzewa dało się policzyć informacje dla rodzica.
- Drzewo BST (binary search tree)
    - search(root, k)
        * jeśli root = k, to zwracamy root; jeśli root > k, to szukamy w lewym poddrzewie; wpp. szukamy w prawym poddrzewie
        * złożoność czasowa: $O(h(T))$
    - insert(root, k)
        * robimy search(root, k) i dostajemy liść drzewa; podpinamy do niego k z odpowiedniej strony
        * złożoność czasowa: $O(h(T))$
    - delete(root, k)
        * robimy search(root, k). Jeśli znaleziony węzeł jest liściem, to go usuwamy. Jeśli ma jedno dziecko, to podpinamy to dziecko pod jego rodzica. Jeśli ma dwoje dzieci, to znajdujemy skrajnie lewy węzeł w prawym poddrzewie (jest on liściem oraz najmniejszą wartością większą od k, co sprawia, że po zamianie zachowany będzie porządek BST), usuwamy go, a jego wartość zapisujemy w węźle k.
        * złożoność czasowa: $O(h(T))$
- Drzewo AVL

– Zrównoważone drzewo BST, w którym dla każdego węzła wysokości jego poddrzew różnią się maksymalnie o 1. Dla AVL drzewa $h = log\ n$. Do balansowania drzewa AVL definiujemy operacje rotacji. W każdym wierzchołku trzymamy wskaźnik zrównoważenia $\in \{-1, 0, 1\}$
– rotate_left(), rotate_right()
    * operacje polegające na przepięciu kilku poddrzew i wierzchołków w stałym czasie, powodujące zmianę różnicy w wysokości lewego i prawego poddrzewa wierzchołka, a jednocześnie zachowujące porządek BST.
    * złożoność czasowa: $O(1)$



Rotacje zachowują porządek symetryczny w BST, numerację infiksową (inorder) węzłów drzewa binarnego!

– double_rotate_left(), double_rotate_right()
    * złożoność czasowa: $O(1)$

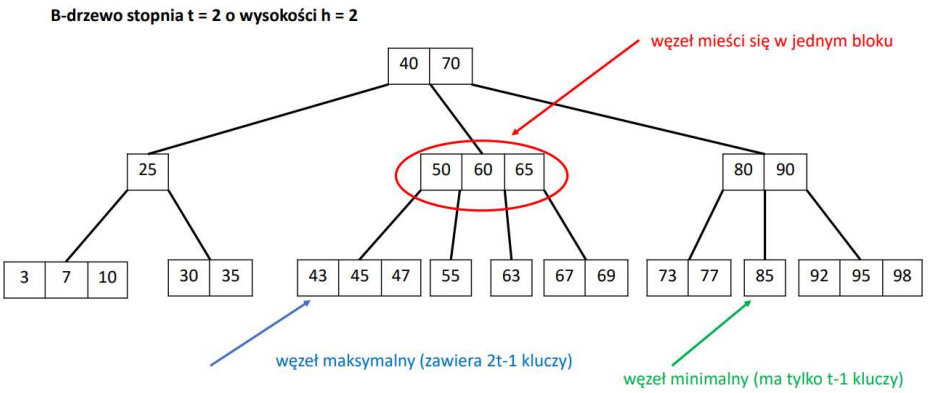


– insert() i delete() wykonujemy początkowo jak dla zwykłego drzewa BST, ale po wykonaniu ich musimy zadbać o zrównoważenie drzewa. W zależności od tego jaki był poprzednio wskaźnik zrównoważenia w poddrzewie, które modyfikujemy i czy zmienia się lewe czy prawe poddrzewo, musimy czasem wykonać: rotację pojedynczą, rotację podwójną, poprawienie wskaźników zrównoważenia na ścieżce od zmodyfikowanego węzła do korzenia poprzez rotacje.
– Złożoności czasowe:
    * search(): $O(log\ n)$
    * insert(): $O(log\ n)$
    * delete(): $O(log\ n)$

• Drzewo typu "splay"

- localsplay(x)
    * przenosimy węzeł x do korzenia, przy założeniu że jest 1 lub 2 poziomy pod korzeniem (inaczej: przenosimy węzeł x o 1 lub 2 poziomy w górę), zachowując porządek BST. Jeśli jest 1 poziom pod korzeniem, wystarczy wykonać jedną rotację (left_rotate/right_rotate)
    * złożoność czasowa: $O(1)$
- splay(x)
    * wykonujemy localsplay(x) dopóki nie znajdzie się w korzeniu
- operacje search, insert, delete wykonujemy tak jak w zwykłym drzewie BST, ale zawsze na koniec robimy splay() na węźle, którego dotyczyła operacja (lub jego rodzicu dla delete())
- Złożoności czasowe: search(), insert() i delete() mają *pesymistyczną złożoność* $O(n)$, ale *zamortyzowaną* $O(log\ n)$

• B-drzewo
- Nieco dziwne drzewo, w którym w każdym wierzchołku trzymamy wiele wartości i wiele wskaźników na dzieci. B-drzewo stopnia t charakteryzuje się takimi cechami:
    * W każdym wierzchołku trzymane jest m = liczba trzymanych kluczy, lista kluczy: Key_1, ..., Key_m oraz zmienna is_leaf = (węzeł jest liściem)
    * W wierzchołkach niebędących liśćmi trzymana jest lista m+1 wskaźników na dzieci: P_1, ..., P_{m+1}, dla liści są to NULLe.
    * Każdy klucz K w dziecku, na które wskazuje P_i spełnia warunek $\text{Key}_{i-1} < K < \text{Key}_i$.
    * Wszystkie liście leżą na tej samej głębokości.
    * Każdy węzeł różny od korzenia zawiera co najmniej t-1 i co najwyżej 2t-1 kluczy. Korzeń niepustego drzewa zawiera od 1 do 2t-1 kluczy.
- Tak wygląda B-drzewo stopnia 2:



- search(x)
    * algorytm podobny do search() w zwykłym drzewie BST, ale gdy wchodzimy do wierzchołka musimy przeiterować się po liście kluczy, która w nim jest i znaleźć pierwszy większy od x.
    * złożoność czasowa: $O(t \cdot log_t\ n)$
- insert(x)
    * gdybyśmy nie dbali o to, żeby węzły miały ograniczoną liczbę wierzchołków, to można by po prostu znaleźć miejsce na x i go tam dodać.
    * Ale dbamy, więc utrzymujemy następujący niezmiennik podczas schodzenia w dół drzewa: węzeł do którego wchodzimy nigdy nie jest maksymalny. Przy schodzeniu w dół drzewa jeśli chcemy wejść do węzła, który jest maksymalny, najpierw wykonujemy na nim operację:
    * Bierzemy środkowy klucz z maksymalnego węzła, wstawiamy go na odpowiednie miejsce (możemy to zrobić, bo mamy niezmiennik, że węzeł do którego weszliśmy nie był maksymalny), rozdzielamy

poprzednio maksymalny węzeł na dwa węzły po połowie kluczy oraz odpowiednio updatujemy wskaźniki P_i na dzieci.

* złożoność czasowa: $O(t \cdot log_t\ n)$

– delete(x)

* operacja czerpiąca z delete() na zwykłym drzewie BST, podobna też do powyższej operacji insert(), tylko że musimy dbać o to, żeby liczba kluczy nie spadła poniżej minimum. Trzymamy do tego niezmiennik: węzeł do którego wchodzimy, jeśli różny od korzenia, nigdy nie jest minimalny. Jest to jednak bardziej skomplikowane od insert().
* złożoność czasowa: $O(t \cdot log_t\ n)$

- Drzewo czerwono-czarne
  – **TODO**

Haszowanie

- Funkcja haszująca, uniwersalna rodzina funkcji haszujących
  – *Funkcja haszująca* to funkcja która pewnemu zbiorowi U przypisuje wartości ze zbioru {0, 1, ..., m-1}. Najczęściej |U| ≫ m i problem haszowania polega na tym, żeby nie znając pewnego zbioru wartości należących do U tak wybrać funkcję haszującą, żeby równomiernie przypisywała wartości od 0 do m.
  – Dwa różne klucze e, f są ze sobą w kolizji gdy h(e) = h(f)
  – Definicja: Rodzina funkcji haszujących $H = \left\{h_i : U \to \{0, 1,\ ..., m\}\right\}$ jest *uniwersalna*, jeśli:

    dla każdej pary różnych elementów $e, f \in U$ liczba funkcji $h_i \in H$ dla których $h_i(e) = h_i(f)$ (in. dla których e, f są w kolizji) wynosi $\leqslant \frac{|H|}{m}$.

    * Z tego wynika, że jeśli losowo wybierzemy funkcję haszującą h_i z H, to prawdopodobieństwo kolizji między dowolnymi elementami e, f jest $\leqslant \frac{1}{m}$.
  – Weźmy $\alpha = \frac{n}{m}$, gdzie n - liczba elementów w słowniku (m jak wyżej). Zauważmy, że taka definicja rodziny funkcji haszujących zapewnia nam, że oczekiwana długość a[i] (patrz: Słownik z haszowanie, poniżej) jest równa około α.
    Teraz, jeśli w słowniku jest $n = O(m)$ elementów, to $\alpha = O(1)$, a więc oczekiwana złożoność każdej z operacji słownikowych na takim słowniku to $O(1)$.
    Stąd, jeśli mamy ciąg n operacji słownikowych na początkowo pustym słowniku, to każda z nich ma oczekiwaną złożoność $O(1)$, a więc oczekiwana złożoność takiego ciągu operacji to $O(n)$.
  – Przykładowa uniwersalną rodziną funkcji haszujących:

    * Weźmy funkcję haszującą, która jest produktem z mnożenia pewnej macierzy złożonej z zer i jedynek przez wektor będący zapisem binarnym klucza. Wynikiem tego produktu jest wektor (o długości b, ustalonej przez nas), a naszym haszem jest liczba, której zapisem dwójkowym są kolejne zera i jedynki z tego wektora.
    * Zbiór takich funkcji dla ustalonego b i wszystkich możliwych macierzy jest uniwersalną rodziną funkcji haszujących.
  – Statyczne haszowanie doskonałe: **TODO**

    * Ogólna idea jest taka: mamy dany z góry zbiór n kluczy. Chcemy zaprojektować słownik jedynie z operacją search(), która ma pesymistyczną złożoność czasową O(n). Okazuje się, że da się to zrobić.
- Algorytm Karpa-Rabina (idea)
  – Jest to algorytm wyszukiwania wzorca w tekście, wykorzystujący ideę haszowania do przyspieszenia metody naiwnej (brute force).
  – Algorytm: mamy dany wzorzec x i tekst y, nad alfabetem = {0,1}. Definiujemy funkcję haszującą h(z) równą

liczbie, której zapisem binarnym jest z, braną modulo q, gdzie q jest pewną wybraną dużą liczbą pierwszą. Liczymy sobie na początek h(x) i hasz h(y[0,..,|x|-1]) (pierwsze podsłowo). Idziemy po kolei po wszystkich podsłowach y długości |x| i dla każdego przyrównujemy jego hasz z h(x). Zauważamy, że na podstawie haszu danego podsłowa y możemy łatwo policzyć hasz kolejnego podsłowa (odcięcie pierwszego bitu i dodanie nowego bitu na koniec, a potem wzięcie tego modulo q - jak się chwilę zastanowić, to można to policzyć). No więc po prostu idziemy po kolejnych podsłowach i porównujemy ich hasze do h(x). Jeśli są różne to wiemy, że te słowa nie są równe, a jeśli są równe to nie mamy takiej pewności (własność funkcji haszującej). A więc jeśli hasze są równe to brute-forcowo sprawdzamy czy słowa są równe.
  - Możemy zauważyć, że algorytm ten nie będzie szybko działał, jeśli jest dużo wystąpień wzorca w tekście, bo każde trzeba liniowo porównać.
- Słownik z haszowaniem
  - Słownik S implementujemy jako tablicę list a[0..m-1], gdzie a[i] to lista wszystkich kluczy należących do S którym funkcja haszująca przypisała wartość i.
  - Sprawdzanie czy klucz k jest w słowniku: obliczamy h(k) i przechodzimy po liście a[h(k)] sprawdzając czy któraś wartość na liście jest równa k.
  - Dodawanie klucza do słownika: dodajemy k na koniec listy a[h(k)]

Algorytmy grafowe

- Algorytm Dijkstry
  - Na wejściu dostajemy spójny graf, w którym krawędzie mają przyporządkowane nieujemne wagi. Algorytm oblicza ścieżki o najmniejszej łącznej wadze z wyróżnionego wierzchołka do wszystkich pozostałych.
  - Algorytm w skrócie: przechowujemy potencjalne wagi ścieżek od źródła w tablicy w' oraz zbiór wierzchołków z jeszcze nieobliczonymi ścieżkami - R. Na początku $R = V \setminus \{s\}$, w'[s] = 0, sąsiadom s ustawiamy w' równe wadze krawędzi s–v, a dla wszystkich innych wierzchołków ustawiamy w'[v] = $+\infty$. W pętli wybieramy wierzchołek v z R o najmniejszej wadze w', po czym usuwamy v z R, a następnie updatujemy w' dla jego sąsiadów: w'[v_neighbour] = min(w'[v_neighbour], w'[v] + w[$v - v$_neighbour]).
  - Własności:
    * podczas wykonywania algorytmu (zakładając że wykorzystujemy kolejkę priorytetową) wykonujemy n-1 operacji min(), n-1 operacji delete_min() i m operacji decrease_key()
  - Złożoność czasowa: $O(n \log n + m)$ przy wykorzystaniu kopca Fibonacciego jako kolejki priorytetowej
- Algorytm Floyda-Warshalla
  - Algorytm obliczający najkrótszą ścieżkę (= najmniejsza liczba krawędzi; krawędzie nie mają wag) dla każdej pary wierzchołków w zadanym grafie.
  - Algorytm:
    * Dostajemy graf z ponumerowanymi wierzchołkami od 1 do n. Wprowadzamy oznaczenie $W^k[i, j]$ = najlżejsza ścieżka z i do j, której wierzchołki wewnętrzne (czyli wszystkie poza $i,\ j$) mają numer $\leqslant k$.
    * Obserwacja 1: $W^k[i, j] = min\left(W^{k-1}[i, k] + W^{k-1}[k, j], W^{k-1}[i, j]\ \right)$. Innymi słowy, długość najlżejsza ścieżka z i do j po wierzchołkach $\leqslant$ k to albo najlżejsza ścieżka z i do j po wierzchołkach $\leqslant$ k-1 (czyli nieprzechodząca przez wierzchołek k), albo jest to ścieżka złożona z najlżejszej ścieżki z i do k połączonej z najlżejszą ścieżką z k do j (obie po wierzchołkach $\leqslant$ k) (czyli ścieżka przechodząca przez wierzchołek k).
    * Obserwacja 2: $W^0[i, j] = A[i, j]$, gdzie A to zwykła macierz sąsiedztwa z zerami wymienionymi na $+\infty$
    * Algorytm polega na tym, że najpierw tworzymy macierz W^0 z obserwacji 2, a potem iteracyjnie przechodzimy z W^i do W^{i+1} ze wzoru z obserwacji 1, aż dostaniemy W^n, gdzie n to liczba wierzchołków w grafie (czyli największy numer wierzchołka). W^n to właśnie wynik, którego szukamy.
  - Złożoność czasowa: $O\left(n^3\right)$

- Algorytm na domknięcie przechodnie grafu
  - Mamy dany graf skierowany. Mamy znaleźć domknięcie przechodnie tego grafu, czyli nowy graf skierowany, taki, że jest w nim krawędź $u \to v$ wtw. gdy w wejściowym grafie istnieje ścieżka (skierowana) od u do v.
  - Idea algorytmu: tworzymy (lub dostajemy na wejściu) macierz sąsiedztwa A wejściowego grafu. Definiujemy sobie taki specjalny nowy operator logiczny: $\circ$, działający na dwie macierze zer i jedynek (które traktujemy jak wartości logiczne 0=fałsz, 1=prawda), którego definicji nie chce mi się przepisywać. Następnie robimy przełomową obserwację, że dla macierzy sąsiedztwa A z wejścia mamy: $A^k[i, j] = 1$ wtw. gdy w wejściowym grafie istnieje ścieżka z i do j o długości dokładnie k. (nwm czemu to działa). A potem kolejna przełomowa obserwacja: jak w macierzy A na przekątnej wpiszemy jedynki to dostajemy: $A^{n-1}[i, j] = 1$ wtw. gdy w wejściowym grafie istnieje ścieżka z i do j (dowolnej długości). A więc rozwiązanie zadania sprowadza się do podniesienia macierzy do potęgi $n - 1$, a na to mamy szybkie algorytmy.
  - **TODO:** dokładniej jak działa ten algorytm i czemu działa
  - Złożoność czasowa: $O\left(n^{\omega}\ log\ n\right)$, gdzie ω to stała występująca w złożoności najszybszego algorytmu na potęgowanie macierzy jaki znamy.
- BFS (Breadth first search)
  - BFS to sposób przechodzenia grafu "wszerz". Mamy dany graf w formie listy sąsiedztwa, mamy znaleźć długości najkrótszych ścieżek od zadanego wierzchołka u w grafie do wszystkich pozostałych.
  - Algorytm: algorytm jest iteracyjny (bez rekurencji) i używamy do niego zwykłej kolejki Q (FIFO), tablicy odwiedzonych wierzchołków i tablicy $dist$ odległości wierzchołków od u. Na początek do Q dodajemy wierzchołek źródłowy u i oznaczamy go jako odwiedzonego, a potem dopóki Q nie jest pusta wyciągamy z niej wierzchołek v i przechodzimy się po wszystkich jego sąsiadach. Jeśli sąsiad nie był odwiedzony, to ustawiamy jego $dist$ na $dist[v] + 1$, oznaczamy jako odwiedzonego i dodajemy do Q.
  - Złożoność czasowa: $O(n + m)$
- DFS (Depth first search)
  - DFS to sposób przechodzenia grafu "w głąb". Mamy dany graf w formie listy sąsiedztwa i chcemy odwiedzić wszystkie jego wierzchołki w taki sposób. Algorytm jest rekurencyjny (można go też zaimplementować iteracyjnie używając stosu).
  - Algorytm w skrócie: trzymamy tablicę czy wierzchołek został odwiedzony. Przechodzimy po wszystkich wierzchołkach. Dla wierzchołka v oznaczamy, że został odwiedzony, a następnie dla każdego z jego sąsiadów wywołujemy algorytm rekurencyjnie.
  - Złożoność czasowa: $O(n + m)$
- Drzewo przeszukiwania DFS
  - Istotna własność: jest ma ono numerację preorder (w numeracji DFS), a więc numery wierzchołków w poddrzewie wierzchołka i tworzą zbiór $\{i + 1,\ i + 2,\ ...,\ i + k\}$.
- Algorytm sprawdzający dwuspójność grafu
  - Chodzi tutaj o dwuspójność wierzchołkową. Graf jest dwuspójny wierzchołkowo, kiedy nie ma w nim wierzchołka rozspójniającego, czyli wierzchołka, którego usunięcie zwiększa liczbę spójnych składowych.
  - Algorytm:
    * Definiujemy low[u] = najmniejszy numer wierzchołka w numeracji DFS do którego można dostać się z u najpierw chodząc po poddrzewie o korzeniu w u (w drzewie DFS) a potem przechodząc jedną krawędzią niedrzewową w górę (poza poddrzewo u).
    * Okazuje się, że wierzchołek v różny od korzenia (w drzewie DFS) jest rozdzielający wtw. gdy posiada dziecko u takie, że low[u] $\geqslant$ nr_dfs[v]. Korzeń jest wierzchołkiem rozdzielającym wtw. gdy ma więcej niż jedno dziecko.
    * Reszta algorytmu w skrócie: przechodzimy po grafie DFSem jednocześnie licząc wartości low i sprawdzamy czy powyższy warunek jest spełniony.
    * Złożoność czasowa: $O(n + m)$
- Algorytm na znalezienie dwuspójnych składowych

  - Dwuspójna składowa to maksymalny dwuspójny podgraf (nie można dodać do niego żadnego wierzchołka, żeby nadal był dwuspójny)
  - Algorytm w skrócie: **TODO** (korzystamy z poprzedniego algorytmu)
  - Złożoność czasowa: chyba $O(n+m)$, chociaż na slajdach jest $O(m)$ (**TODO**)
- Kolorowanie Brooks'a
  - Algorytm znajdujący $\Delta(G)$-kolorowanie grafu G, gdzie $\Delta(G)$ to maksymalny stopień wierzchołka w G. Zakłada, że G nie jest kliką ani cyklem nieparzystej długości (bo wtedy nie da się znaleźć takiego kolorowania z tw. Brooksa). Korzystamy z tego, że graf jest k-kolorowalny wtw. gdy każda jego dwuspójna składowa jest k-kolorowalna (twierdzenie, jak pokolorujemy każdą dwuspójną składową, to możemy cyklicznie przesunąć kolory tak żeby uzyskać poprawne kolorowanie całego grafu). Wystarczy więc znaleźć kolorowania dla każdej dwuspójnej składowej (na wykładzie pominięte było jak potem cyklicznie przesunąć kolory), i algorytm, który to robi nazwijmy Kolorowanie Brooks'a+
  - Kolorowanie Brooks'a+
    * Mamy dany graf dwuspójny G oraz trzy wierzchołki a, b, c, takie, że w G są krawędzie a--c, b--c i nie ma krawędzi a--b. Chcemy znaleźć kolorowanie opisane wyżej.
    * Algorytm w skrócie: kolorujemy wierzchołki a i b kolorem 1 (najmniejszym). Przeglądamy w głąb graf G\{a,b} poczynając od c i nadajemy wierzchołkom numerację DFS. Kolorujemy wierzchołki w kolejności od największego do najmniejszego numeru DFS, kolorując każdy wierzchołek najmniejszym kolorem, który nie został jeszcze użyty do pokolorowania jego sąsiadów w całym grafie G.
    * Złożoność czasowa: $O(n+m)$
  - Złożoność czasowa: $O(n+m)$
- Algorytm wyznaczanie st-numeracji
  - **TODO**
- Algorytm sprawdzania czy graf jest wielokątowy
  - **TODO**
- Algorytm testowania silnej spójności
  - Graf skierowany jest silnie spójny wtw. gdy dla każdej pary wierzchołków u, v istnieje ścieżka z u do v oraz z v do u.
  - Algorytm:
    * Trzymamy graf jako dwie listy sąsiedztwa L_+: $L_+[u]$ to lista wierzchołków do których można przejść 1 krawędzią z u, i L_-: $L_-[u]$ to lista wierzchołków z których można przejść 1 krawędzia do u.
    * Bierzemy dowolny wierzchołek s z grafu. Robimy DFS z s "w przód", czyli korzystając tylko z L_+ jako list sąsiedztwa i kolorując odwiedzone wierzchołki na biało. Następnie analogicznie robimy DFS z s "w tył", korzystając z L_- i kolorując odwiedzone wierzchołki na czarno. Graf jest silnie spójny wtw. gdy każdy wierzchołek został pokolorowany dwoma kolorami.
  - Złożoność czasowa: $O(n+m)$ (2xDFS)
- Algorytm znajdowania silnie spójnych składowych
  - Algorytm (przeklejone ze slajdu):
    * przeszukaj graf w przód metodą w głąb numerując wierzchołki w kolejności odwiedzania i obliczając dla każdego wierzchołka rozmiar poddrzewa w lesie przeszukiwania w głąb, o korzeniu w tym wierzchołku
    * przeglądaj wierzchołki w kolejności odwiedzania w przód (punkt 1) i jeśli aktualnie oglądany wierzchołek v nie został jeszcze przypisany do żadnej silnie spójnej składowej (s[v] nie jest jeszcze określone), uruchom przeszukiwanie w tył (metoda w głąb) z wierzchołka v, oznaczając wszystkie wierzchołku osiągane z v i należące do poddrzewa w przód o korzeniu w tym wierzchołku, jako należące do silnie spójnej składowej o etykiecie v
  - Złożoność czasowa: $O(n+m)$
- Algorytm Kruskala (minimalne drzewo rozpinające)
  - Mamy dany spójny graf nieskierowany z wagami na krawędziach, krawędzie dostajemy posortowane

niemalejąco. Mamy znaleźć drzewo rozpinające ten graf o najmniejszej sumie wag.
  - Algorytm: korzystamy ze struktury Find-Union (patrz: rozdział Find-Union), początkowo składającej się z pojedynczych wierzchołków. Trzymamy F będące zbiorem krawędzi wchodzących w skład wynikowego drzewa, początkowo pustym. Iterujemy się po krawędziach w kolejności niemalejącej i dla każdej krawędzi a--b sprawdzamy czy Find(a)==Find(b). Jeśli tak, to dodanie a--b spowodowałoby do F spowodowałoby powstanie cyklu, więc nie dodajemy jej do F. Wpp. zachłannie dodajemy ją do F i robimy Union(Find(a), Find(b)).

Find-Union

- Struktura Find-Union
  - Struktura reprezentująca zbiór zbiorów, które można ze sobą łączyć. Umożliwia szybkie sprawdzanie czy dwa elementy są w tym samym zbiorze.
  - Implementacja drzewiasta: trzymamy las w postaci listy p, gdzie p[i] = numer rodzica wierzchołka i, lub i, jeśli i jest korzeniem (to sposób na sprawdzenie, czy wierzchołek jest korzeniem). Trzymamy też listę d, gdzie d[i] = rozmiar poddrzewa o korzeniu w i, przy czym te wartości są poprawne kiedy i jest korzeniem któregoś drzewa, w przeciwnym wypadku nie muszą być poprawne, bo nie są nam potrzebne.
    * Ini(n)
      · Tworzymy las złożony z pojedynczych wierzchołków. Nadajemy odpowiednie wartości tablicy p i d.
      · Złożoność czasowa: $O(n)$
    * Find(i)
      · Przechodzimy za pomocą tablicy p od i do korzenia. Numer korzenia to wynik tej operacji. Po przejściu do korzenia, przechodzimy jeszcze raz tą samą ścieżką (od i do korzenia) i każdy wierzchołek na tej ścieżce podpinamy bezpośrednio pod ten korzeń. (kompresja ścieżek)
      · Złożoność czasowa: $O(log\ n)$ (niezamortyzowane)
    * Union(i, j)
      · Zakłada, że i, j to korzenie. Podpinamy drzewo mniejszego rozmiaru pod większe drzewo (rozmiary są w tablicy d).
      · Złożoność czasowa: $O(1)$
  - Zamortyzowana złożoność czasowa ciągu m operacji Find/Union, dla początkowego podziału na singletony: $O((m+n)\cdot\alpha(n))$, gdzie $\alpha(n)$ jest funkcją tak wolno rosnącą, że można ją uznać za stałą.

Algorytmy tekstowe

- Algorytm KMP (Knutha-Morrisa-Pratta)
  - Jest to algorytm wyszukiwania wzorca w tekście. Mamy dany wzorzec x i tekst y, mamy znaleźć wszystkie wystąpienia wzorca w tekście.
  - Tablica prefiksów-sufiksów P: P[i] to długość maksymalnego właściwego prefiksu, który jest jednocześnie sufiksem słowa x[0..i]. P[0] = 0, P[1] = 0.
  - Algorytm w skrócie (**TODO:** dokończyć): dodajemy do x znak $ na koniec i do y znak # (zakładamy, że te znaki nie występują w alfabecie). Daje to właściwość, że żaden sufiks nie jest właściwym prefiksem innego sufiksu. ...
  - Złożoność czasowa: $O(|y|)$
- Drzewo sufiksowe (zwarte)
  - Struktura o $O(n) \leqslant 2n+1$ węzłach, gdzie n to długość słowa s, dająca nam wygodny dostęp do wszystkich podsłów słowa s.
  - Można dzięki niej szybko rozwiązywać problemy tekstowe, które na pierwszy rzut oka wydają się mieć dużą złożoność
- Tablica sufiksowa

  - Tablica sufiksowa SA dla słowa s z dołączonym znakiem $\$$ na końcu: SA[i] = pozycja na której zaczyna się i-ty leksykograficznie sufiks s. Czyli SA wyznacza porządek leksykograficzny sufiksów.
  - Przydaje się też tablica odwrotna SA^{-1}, którą łatwo można wyznaczyć z SA.
- Tablica LCP
  - LCP[0..n] to tablica nieujemnych liczb całkowitych taka, że LCP[0] = 0, a dla każdego i>0 LCP[i] = długość najdłuższego wspólnego prefiksu sufiksów wyznaczonych przez SA[i] oraz SA[i-1].
- Algorytm tworzenia drzewa sufiksowego (z tablicy sufiksowej i LCP)
  - Algorytm w skrócie: **TODO**
  - Złożoność czasowa: $O(n)$, gdzie n - długość słowa