Opracowanie zadań z zakresu II kolokwium

Kolokwium 2016/2017

Zadanie 1

Student chce wypuścić n różnych pokémonów (numerowanych od 0 do $n-1$) z klatek (pokéball'i). Wypuszczony Pokémon natychmiast atakuje swojego wybawiciela, chyba że (a) jest spokojny, lub (b) w okolicy znajdują się co najmniej dwa uwolnione pokémony, na które ten pokémon poluje. Proszę zaimplementować funkcję:

```
int* releaseThemAll( HuntingList* list, int n ),
```

gdzie `list` to lista z informacją, które pokémony polują na które (lista nie zawiera powtórzeń):

```
struct HuntingList {
  HuntingList* next; // następny element listy
  int predator;      // numer pokemona, który poluje
  int prey; }        // numer pokemona, na którego poluje
```

Funkcja powinna zwrócić `n` elementową tablicę z numerami pokémonów w kolejności wypuszczania (tak, żeby wypuszczający nie został zaatakowany) lub `NULL` jeśli taka kolejność nie istnieje. Każdy wypuszczony pokémon zostaje "w okolicy". Jeśli pokémon nie występuje na liście jako `predator` to znaczy, że jest spokojny. Zaimplementowana funkcja powinna być możliwie jak najszybsza. Proszę krótko oszacować jej złożoność.

Rozwiązanie:

Tworzymy strukturę pokemon, mającą pola:

1. Licznik released - czy już wypuszczono danego pokemona (0 albo 1, może być też bool).
2. Lista my_prey - lista pokemonów, na który dany pokemon poluje (węzły zawierają pojedynczy int z numerem pokemona).
3. Lista hunting_me - lista pokemonów, które polują na danego pokemona (węzły zawierają pojedynczy int z numerem pokemona).

Tworzymy tablicę pokemonów długości n, gdzie każdy element początkowo ma licznik 0 i obie listy puste (numer indeksu to numer pokemona, czyli pole predator z danej w zadaniu struktury). Tworzmy też tablicę int release_order[n] - to jest wynik, ją będziemy zwracać, ustawiamy wszędzie -1.

Na początek musimy ustalić, które pokemony są spokojne, a które są łowcami, oraz które polują na które. Przechodzimy po liście predatorów, za każdym razem:

1. Wrzucamy rozpatrywanemu w tej chwili łowcy do jego listy my_prey jego ofiarę, czyli tworzymy nowego node'a dla listy my_prey z numerem ofiary.
2. Wrzucamy ofierze danego łowcy do listy hunting_me jego łowcę, czyli tworzymy nowego node'a dla listy hunting_me z numerem w tej chwili rozpatrywanego łowcy.

W ten sposób mamy uzupełnione listy, kto poluje na kogo i kto jest czyją ofiarą. Teraz możemy wypuścić wszystkie spokojne pokemony: przechodzimy po naszej tablicy pokemonów i wypuszczamy każdego, który ma pustą listę my_prey, wpisując ich numery (czyli numery ich indeksów) do release_order oraz zwiększając ich licznik released z 0 na 1.

Możemy teraz przejść do wypuszczania łowców. Będzie nam tutaj potrzebna rekurencyjna funkcja pomocnicza void release_hunters, która dostaje indeks (numer pokemona) jako argument. Idziemy w niej najpierw pętlą po liście ofiar danego pokemona, sprawdzając, czy pod indeksem jego ofiary w polu released jest 0 czy 1. Zliczamy wypuszczone ofiary, aż dojdziemy do 2, wtedy przerywamy pętlę (bo czy jest więcej, to nas nie interesuje). Jeśli przeszliśmy całą listę ofiar i nie udało nam się znaleźć 2 wypuszczonych, to kończymy - nie mamy dość ofiar. Jeśli natomiast uda nam się znaleźć 2 wypuszczone ofiary, to wypuszczamy tego łowcę, którego teraz rozpatrujemy. Musimy potem przejść po jego liście hunting_me, wywołując tę funkcję rekurencyjnie dla pokemonów, które polują na rozpatrywanego w tej chwili - w końcu jego wypuszczenie mogło sprawić, że można wypuścić tamte.

Mając tę funkcję, wystarczy, że przejdziemy po kolei po liście pokemonów i wywołamy ją dla każdego pokemona, który ma licznik released=0, czyli jeszcze go nie wypuściliśmy. Ze względu na działanie funkcji rekurencyjnej wystarczy tylko 1 przejście po liście pokemonów.

Na koniec sprawdzamy, czy ostatni element tablicy release_order jest równy -1, czy ma jakąś wartość. W pierwszym przypadku nie udało się wypuścić wszystkich n pokemonów i zwracamy NULL, w przeciwnym wypadku zwracamy wskaźnik na release_order.

Przewidywana złożoność to O(n) - przechodzenie po listach to zawsze n, a funkcja rekurencyjna tego zwykle nie zwiększy. Pesymistyczny przypadek to O(n^2), który nastąpi, gdy każdy pokemon będzie polował na każdego (lub liczba ofiar dla każdego pokemona będzie niewiele mniejsza), bo wtedy rekurencja wywoła się (n-1)*n razy, jednak jest to bardzo rzadki przypadek.

Zadanie 2

Dana jest struktura `struct HT{ int* table; int size; }`, która opisuje tablicę haszującą rozmiaru `size`, przechowującą liczby nieujemne. Tablica korzysta z funkcji haszującej `int hash(int x)` i liniowego rozwiązywania konfliktów (ujemne wartości w tablicy `table` oznaczają wolne pola). Doskonałością takiej tablicy nazywamy liczbę elementów `x` takich, że pozycja `x` w tablicy to `hash(x) mod size` (a więc `x` jest na "swojej" pozycji). Proszę napisać funkcję:

```
void enlarge( HT* ht);
```

która powiększa tablicę dwukrotnie i wpisuje elementy w takiej kolejności, by doskonałość powstałej tablicy była jak największa. Funkcja powinna być możliwie jak najszybsza.

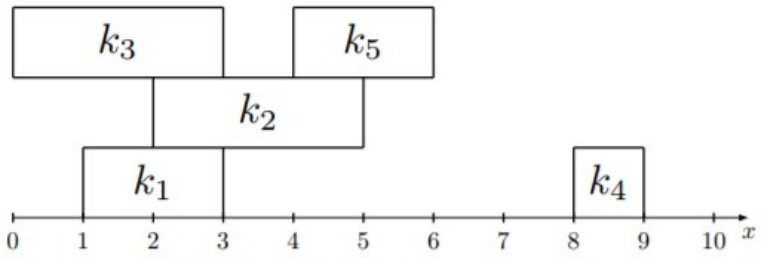
Rozwiązanie:

Przechodzimy po naszej danej tablicy, sprawdzamy po kolei hash dla danego pola w nowej tablicy. Jeśli w nowej tablicy dany element po zhaszowaniu byłby na "swojej" pozycji, to go wpisujemy tam i zmieniamy wartość w danej tablicy np. na -1 (żeby jakoś oznaczyć te, które już przepisaliśmy). W ten sposób osiągamy najlepszą możliwą doskonałość.

Potem przechodzimy jeszcze raz po danej tablicy, wrzucając po kolei te wartości, których nie wpisaliśmy wcześniej. Nie ma możliwości, żeby jeszcze bardziej zwiększyć doskonałość, można je wrzucać jakkolwiek, jak funkcja haszująca nam wyliczy.

Zadanie 3

Dany jest ciąg klocków $(k_1, ..., k_n)$. Klocek k_i zaczyna się na pozycji a_i i ciągnie się do pozycji b_i (wszystkie pozycje to liczby naturalne) oraz ma wysokość 1. Klocki układane są po kolei. Jeśli klocek nachodzi na któryś z poprzednich, to jest przymocowywany na szczycie poprzedzającego klocka. Na przykład dla klocków o pozycjach (1,3), (2,5), (0,3), (8,9), (4,6) powstaje konstrukcja o wysokości trzech klocków (vide rysunek). Proszę podać możliwie jak najszybszy algorytm, który oblicza wysokość powstałej konstrukcji (bez implementacji) oraz oszacować jego złożoność obliczeniową.



Rysunek 1: Przykład konstrukcji.

Rozwiązanie:

Rozwiązanie korzysta ze skiplisty. Będziemy na niej (na najniższym poziomie) przechowywać struktury, które mają początek przedziału, koniec przedziału i wysokość wieży z klocków na tym przedziale. Wysokość w strukturze nie ma nic (sic!) wspólnego z wysokością w skipliście. Skiplista wykorzystywana jest tylko po to, żeby wstawianie było w czasie O(log(n)).

Najpierw przechodzimy po liście i sprawdzamy lewy i prawy koniec przedziału klocków (lewy koniec przedziału klocka "najbardziej na lewo" i analogicznie dla prawej strony), tworzymy z tego element dla skiplisty (wysokość 0) i wrzucamy do skiplisty jako jedyny element. Ma to złożoność O(n), bo trzeba przejść po liście.

Następnie dla każdego klocka tworzymy nową strukturę i wstawiamy ją do skiplisty w taki sposób, żeby żaden przedział na siebie nie nachodził, to znaczy:

1. Jeżeli początek wstawianego klocka znajduje się w środku jakiegoś przedziału (czyli w środku jakiegoś już położonego klocka, a nie pomiędzy jakimiś dwoma), to musimy go "przyciąć", czyli zmniejszyć odpowiednio koniec starego klocka.
2. Przechodzimy po kolejnych przedziałach, które mają jakąś część wspólną z naszym klockiem: szukamy największej wysokości, która do tej pory tam była, usuwamy stare klocki w całości zawierające się w naszym nowym i szukamy klocka, do którego należy koniec naszego przedziału.
3. Jeśli klocek, do którego należy koniec naszego przedziału istnieje, to go odpowiednio "przycinamy" analogicznie do przycinania początku.
4. Ustawiamy wysokość naszego nowego klocka na dotychczasową największą wysokość +1.

Złożoność:

- każdy z n danych klocków trzeba wstawić do skiplisty, co zajmuje O(n*log(n)),
- jeżeli do znalezienia końca klocka musimy przejść więcej niż 1 przedział, to będzie trzeba usuwać istniejące klocki, pesymistycznie O(n). Łącznie daje to amortyzowane O(n).
- łącząc powyższe, złożoność to O(nlog(n)).

Kolokwium 2015/2016

Zadanie 1

Dane są struktury danych opisujące SkipListę:

```
struct SLNode {
    int      value; // wartość przechowywana w węźle
    int      level; // poziom węzła
    SLNode** next;  // level-elementowa tablica wskaźników na następniki
};
struct SkipList {
    SLNode* first;  // wartownik przedni (pole value ma wartość -∞)
    SLNode* last;   // wartownik tylny (pole value ma wartość +∞)
};
const int MAX_LEVEL = ... ; // maksymalny poziom węzłów
```

Proszę zaimplementować funkcję `SkipList merge( SkipList A, SkipList B )`, która scala dwie otrzymane SkipListy (w efekcie powstaje nowa SkipLista składająca się z dokładnie tych samych węzłów co poprzednie; nie należy zmieniać poziomów węzłów). Węzły wartowników mają poziom `MAX_LEVEL`. Funkcja powinna działać możliwie jak najszybciej. Proszę oszacować złożoność czasową.

Rozwiązanie:

Tworzymy nową, pustą skiplistę, jej maksymalna wysokość jest większą z wysokości danych skiplist A i B. Potem idziemy od najwyższego poziomu aż do zerowego włącznie:

1. Ustawiamy wskaźniki na początek nowej skiplisty, początek A i początek B,
2. Jeśli element z A jest mniejszy, to wpinamy go do nowej skiplisty i idziemy do przodu na skipliście A i tej nowej. W przeciwnym wypadku robimy to samo z B.
3. Kiedy dojdziemy do końca którejś z list, przepinamy resztę drugiej i dopinamy jej koniec do końca z nowej skiplisty. Obniżamy poziom o 1 i powtarzamy pętlę.

Złożoność: jedyne, co robi pętla, to przechodzi tyle razy po liście, ile jest poziomów, więc złożoność to O(n).

Zadanie 2

Proszę opisać (bez implementacji) algorytm, który otrzymuje na wejściu pewne drzewo BST T i tworzy nowe drzewo BST T', które spełnia następujące warunki: (a) T' zawiera dokładnie te same wartości co T, oraz (b) drzewo T' jest drzewem czerwono-czarnym (w związku z tym powinno zawierać kolory węzłów).

Rozwiązanie:

Mając tylko tyle informacji, najlepiej będzie, jeśli rozwalimy istniejące drzewo i zbudujemy drzewo czerwono-czarne od zera, korzystając z listy dwukierunkowej.

Wykorzystamy do tego funkcję rekurencyjną: weź root, wepnij go do listy, na jego lewo wepnij jego lewe dziecko (wywołanie rekurencyjne), na jego prawo wepnij jego prawe dziecko (wywołanie rekurencyjne). Dzięki własności drzewa BST, że na lewo wartości muszą być mniejsze, a na prawo większe, to wpinając w ten sposób otrzymamy posortowaną listę dwukierunkową.

Budowanie drzewa wygląda następująco: lista ma długość n, bierzemy z niej medianę (element n/2) i tworzymy z niego root'a nowego drzewa RB, kolorujemy go na czarno. Lewym dzieckiem staje się mediana z lewej części listy, analogicznie z prawym dzieckiem. Rekurencyjnie powtarzamy ten proces, aż zbudujemy całe drzewo.

Ze względu na to, jak zbalansowane jest to drzewo dzięki takiemu sposobowi budowy, to wszystkie node'y poza tymi z najniższego poziomu mogą być czarne, a tylko najniższy poziom będzie czerwony. Dzięki temu drzewo spełnia warunek drzewa RB.

Zadanie 3

W ramach obchodów międzynarodowego święta cyklistów organizatorzy przewidzieli górską wycieczkę rowerową. Będzie się ona odbywać po bardzo wąskiej ścieżce, na której rowery mogą jechać tylko jeden za drugim. W ramach zapewnienia bezpieczeństwa każdy rowerzysta będzie miał numer identyfikacyjny oraz małe urządzenie, przez które będzie mógł przekazać identyfikator rowerzysty przed nim (lub -1 jeśli nie widzi przed sobą nikogo). Należy napisać funkcję, która na wejściu otrzymuje informacje wysłane przez rowerzystów i oblicza rozmiar najmniejszej grupy (organizatorzy obawiają się, że na małe grupy mogłyby napadać dzikie zwierzęta). Dane są następujące struktury danych:

```
struct Cyclist {
    int id, n_id; // identyfikator rowerzysty oraz tego, którego widzi
};
```

Funkcja powinna mieć prototyp `int smallestGroup( Cyclist cyclist[], int n )`, gdzie `cyclist` to tablica n rowerzystów. Funkcja powinna być możliwie jak najszybsza. Można założyć, że identyfikatory rowerzystów to liczby z zakresu 0 do 10^8 (osiem cyfr napisanych w dwóch rzędach na koszulce rowerzysty) i że pamięć nie jest ograniczona. Rowerzystów jest jednak dużo mniej niż identyfikatorów (nie bez powodu boją się dzikich zwierząt). Należy zaimplementować wszystkie potrzebne struktury danych.

Rozwiązanie:

Głównym problemem w zadaniu jest to, że gdyby grupy rowerzystów były listami, to możemy poruszać się nimi tylko w jedną stronę za pomocą n_id jak po liście jednokierunkowej. Przydałoby się natomiast, żeby móc iść w strugą stronę, po p_id - numerze poprzedniego rowerzysty, jadącego za aktualnie rozpatrywanym. Do realizacji tego celu posłuży nam struktura BetterCyclist, zawierająca wszystkie pola zwykłego cyklisty i dodatkowo pole na indeks poprzednika p_id.

Treść zadania wyraźnie sugeruje użycie tablicy haszującej, w związku z czym tworzymy tablicę haszującą zawierającą struktury BetterCyclist, rozmiaru 2n (mogłaby być mniejsza, ale w ten sposób minimalizujemy ryzyko kolizji) i ustawiamy wszędzie numer id na -1 (później rozpoznamy w ten sposób puste, niewykorzystane pola). Haszowanie może wykorzystywać dowolną metodę rozwiązywania konfliktów, w tym wypadku nawet metoda listowa sprawdzi się bardzo dobrze, jednak w takim wypadku trzeba by stworzyć strukturę node'a zawierającego informacje BetterCyclista. Można też użyć metody liniowej czy kwadratowej albo ulepszyć haszowanie o mnożenie przez jakąś liczbę pierwszą.

Przechodzimy po danej tablicy cyklistów, zmieniając każdego na BetterCyclista i haszując ich: tworzymy nową strukturę, jako p_id ustawiamy -1, haszujemy po id i wstawiamy do naszej tablicy haszującej.

Teraz trzeba uzupełnić odpowiednio pola p_id naszych BetterCyclistów. Przechodzimy po danej tablicy cyklistów, szukamy ich następników w tablicy haszującej (haszujemy ich n_id) i wpisujemy następnikowi jako jego p_id numer id naszego obecnie rozpatrywanego rowerzysty (bo jest jego poprzednikiem, a id poprzednika to p_id jego następnika). Po przejściu w ten sposób całej danej tablicy wszyscy cykliści w tablicy haszującej BetterCyclist mają uzupełnione pole p_id.

Co jednak z ostatnimi rowerzystami z grup, którzy nie mieli poprzedników? Ci mają dalej wartość pola p_id równą -1, tak, jak ustawiliśmy na początku. Dzięki temu w tablicy haszującej są 2 szczególne rodzaje BetterCyclistów: ci, którzy mają n_id równe -1 (ci jadą pierwsi w swoich grupach) oraz ci, którzy mają p_id równe -1 (ci jadą ostatni w swoich grupach). Dzięki tej własności możemy rozróżnić końce i początki każdej grupy.

Teraz mamy już ustawione wszystkie wartości, których potrzebujemy i możemy przejść do szukania najmniejszej z grup. Tworzymy licznik smallestGroup, który będzie sprawdzał rozmiar najmniejszej grupie (na początku równy INT_MAX) oraz licznik size, który będzie sprawdzał rozmiar aktualnie rozpatrywanej grupy.

Przechodzimy po tablicy haszującej BetterCyclistów: jeśli dane pole nie jest puste (id!=-1) i jego p_id=-1, to spotkaliśmy ostatniego rowerzystę z danej grupy. Wtedy przechodzimy pętlą po danej grupie:

1. Zwiększamy licznik osób w danej grupie
2. Bierzemy n_id obecnego rowerzysty i haszujemy. Otrzymujemy indeks tablicy haszującej, gdzie jedzie rowerzysta jadący przed tym aktualnie rozpatrywanym. Idziemy tam.
3. Powtarzamy kroki 1-2, dopóki n_id!=-1, bo wtedy napotykamy lidera grupy, jadącego na jej początku.
4. Kiedy dojdziemy do pierwszego w grupie, to porównujemy rozmiary: size (danej grupy) i smallestGroup (najmniejszej grupy do tej pory). Jeśli size<smallestGroup, to smallestGroup=size.

Po przejściu w ten sposób całej tablicy haszującej (pętla uruchomi się tylko dla ostatnich w grupach, więc przez większość tablicy przejdziemy nic nie robiąc) otrzymujemy najmniejszy rozmiar grupy ze wszystkich. Na koniec wystarczy zwrócić smallestGroup.

Kolokwium 2014/2015

Zadanie 1

Pewna firma przechowuje dużo iczb pierwszych w postaci binarnej jako stringi "10101...". Zaimplementuj strukturę danych `Set` do przechowywania tych danych. Napisz funkcje: `Set createSet( string A[], int n )`, która tworzy `Set` z n-elementowej tablicy stringów oraz `bool contains( Set a, string s )`, która sprawdza czy dana liczba jest w `Secie`. Oszacuj złożoność czasową i pamięciową powyższych funkcji.

Rozwiązanie:

Wykorzystujemy strukturę podobną do radix tree, sama znajomość tej struktury nie jest jednak niezbędna. Tworzymy po prostu dość specyficzne drzewo BST, gdzie root to wartownik bez wartości. Każdy node trzyma wartość 0 (jeśli jest lewym dzieckiem) lub 1 (jeśli jest prawym dzieckiem) oraz bool is_end (domyślnie false). Ostatnia wartość jest ważna dlatego, że niektóre ciągi napisów mogą się w sobie zawierać - bool sprawdza, czy jakiś napis kończy się w danym miejscu.

Operacja wstawiania do drzewa przebiega następująco:

1. Idziemy po kolei po napisie, przesuwając się jednocześnie po drzewie.
2. Jeśli w napisie jest 0, to jeśli lewy syn istnieje, to idziemy w lewo. Jeśli nie, to go tworzymy i idziemy do niego.
3. Jeśli w napisie jest 1, to jeśli prawy syn istnieje, to idziemy w prawo. Jeśli nie, to go tworzymy i idziemy do niego.
4. Powtarzamy proces, dopóki nie dojdziemy do końca napisu. Wtedy oznaczamy węzeł, w którym jesteśmy, jako is_end=true.
5. Powtarzamy powyższe kroki dla każdego napisu z danej tablicy.

Odczytywanie jest operacją prawie identyczną:

1. Idziemy po kolei po napisie, przesuwając się jednocześnie po drzewie.
2. Jeśli w napisie jest 0, to jeśli lewy syn istnieje, to idziemy w lewo. Jeśli nie, to zwracamy false.
3. Jeśli w napisie jest 1, to jeśli prawy syn istnieje, to idziemy w prawo. Jeśli nie, to zwracamy false.
4. Kiedy dojdziemy do końca napisu, to zwracamy wartość is_end.

Złożoność tworzenia Setu:

- czasowa: n razy musimy wstawić napis, ilość operacji wstawienia jest proporcjonalna do jego długości. Jeśli m to przeciętna długość napisu, to oczekiwana złożoność czasowa to O(n*m),
- pamięciowa: pesymistycznie potrzebne będzie całe drzewo wysokości h, gdzie h to długość najdłuższego napisu. Drzewo takie ma $O(2^h)$ węzłów, a to jest złożoność pamięciowa.

Złożoność sprawdzenia w Secie:

- czasowa: O(h), gdzie h to długość sprawdzanego napisu, bo trzeba przejść pionowo po drzewie,
- pamięciowa: O(1), w miejscu.

Kolokwium 2013/2014

Zadanie 1A i 1B

Proszę zaimplementować funkcję wstawiającą zadaną liczbę do SkipListy przechowującej dane typu `int`. Proszę zadeklarować wszystkie potrzebne struktury danych i krótko (2-3 zdania) opisać zaimplementowany algorytm.

Proszę zaimplementować funkcję usuwającą zadaną liczbę ze SkipListy przechowującej dane typu `int`. Proszę zadeklarować wszystkie potrzebne struktury danych i krótko (2-3 zdania) opisać zaimplementowany algorytm.

Rozwiązanie:

Patrz W. Pugh "Skip lists" lub W. Pugh "Skip list cookbook".

Zadanie 2A

Proszę opisać jak zmodyfikować drzewa czerwono-czarne (przechowujące elementy typu `int`) tak, by operacja `int sum(T, x, y)` obliczająca sumę elementów z drzewa o wartościach z przedziału $[x, y]$ działała w czasie $O(\log n)$ (gdzie n to rozmiar drzewa T). Pozostałe operacje na powstałym drzewie powinny mieć złożoność taką samą jak w standardowym drzewie czerwono-czarnym. (Podpowiedź: Warto w każdym węźle drzewa przechowywać pewną dodatkową informację, która upraszcza wykonanie operacji `sum` i którą można łatwo aktualizować).

Rozwiązanie:

Dodatkową informacją są sumy elementów w lewym i prawym poddrzewie danego węzła.

Jeśli x=y, to zwracamy tylko x. W przeciwnym wypadku idziemy wskaźnikiem po drzewie tak długo, jak wartość w węźle jest różna od x i y, a także jest jednocześnie większa od x i y lub jednocześnie mniejsza od x i y. W ten sposób kiedy te warunki nie zostaną spełnione, to nasz wskaźnik będzie na szczególnym węźle - jest on rootem najmniejszego poddrzewa zawierającego x i y.

Jeśli x jest tym węzłem, to dodajemy x do sumy i dodajemy tylko sumę y (czym ona jest - wyjaśnienie poniżej). Jeśli y jest tym węzłem, to dodajemy y do sumy i dodajemy tylko sumę x. Jeśli żaden z tych wypadków nie nastąpił, to x musi znajdować się w lewym poddrzewie, a y w prawym poddrzewie. Suma końcowa składa się z: suma x dla root->left, suma y dla root->right, root->val.

Sumowanie od strony x przebiega następująco:

1. Jeśli x znajduje się w lewym poddrzewie, to dodajemy do sumy wartość z aktualnego węzła, sumę prawego poddrzewa (ptr->right->leftSubtreeSum + ptr->right->val + ptr->right->rightSubtreeSum) i idziemy w lewo. Robimy tak, gdyż wszystkie wartości w prawym poddrzewie muszą być większe od x (a mniejsze od y z własności BST).
2. Jeśli x znajduje się w prawym poddrzewie, to idziemy w prawo.
3. Kiedy dojdziemy do x, to dodajemy x, sumę jego prawego poddrzewa i kończymy.

Sumowanie od strony y przebiega podobnie:

1. Jeśli y znajduje się w prawym poddrzewie, to dodajemy do sumy wartość z aktualnego węzła, sumę lewego poddrzewa (ptr->left->leftSubtreeSum + ptr->left->val + ptr->left->rightSubtreeSum) i idziemy w prawo. Robimy tak (symetrycznie do sytuacji z x), gdyż wszystkie wartości w lewym poddrzewie muszą być mniejsze od y (a większe od x).
2. Jeśli y znajduje się w lewym poddrzewie, to idziemy w lewo.
3. Kiedy dojdziemy do y, to dodajemy y, sumę jego lewego poddrzewa i kończymy.

Zadanie 2B

Proszę opisać jak zmodyfikować drzewa AVL (przechowujące elementy typu `int`) tak, by operacja `int findRandom(T)` zwracająca losowo wybrany element z drzewa T działała w czasie $O(\log n)$. Funkcja `findRandom` powinna zwracać każdy element z drzewa z takim samym prawdopodobieństwem. Do dyspozycji mają Państwo funkcję `int random( int k )`, która zwraca liczbę ze zbioru $\{0, ..., k-1\}$ zgodnie z rozkładem jednostajnym. Pozostałe operacje na powstałym drzewie powinny mieć złożoność taką samą jak w standardowym drzewie AVL. (Podpowiedź: Warto w każdym węźle drzewa przechowywać pewną dodatkową informację, która upraszcza wykonanie operacji `findRandom` i którą można łatwo aktualizować).

Rozwiązanie:

Dodatkową informacją jest rozmiar poddrzewa tworzonego przez dany node, włącznie z nim (czyli rozmiar node'a bez dzieci to 1).

Funkcja dostaje jako argument root. Ważne jest, żeby ustalić, jak numerujemy elementy: czy od 0, czy od 1 - w praktyce rzecz jasna nie ma różnicy, bo i tak wskażemy dany element, ale jest to ważne w implementacji. Można łatwo policzyć, którym z kolei elementem jest root: jeśli nie ma lewego poddrzewa, to jest pierwszy; jeśli nie ma prawego poddrzewa, to root->size; jeśli są oba, to root->right->size - root->size. Wystarczy przejść po drzewie, szukając naszego pożądanego node'a, bo wiemy, czy jest w lewej, czy prawej części drzewa oraz który w kolejności jest ten rozpatrywany w danej chwili.

Idąc w lewo, nadal szukamy k-tego node'a. Idąc natomiast w prawo, k zmienia się - w końcu odrzuciliśmy już wszystkie wcześniejsze node'y z lewego poddrzewa. Dlatego idąc w prawo szukamy już elementu k - (numer w kolejności roota, z którego idziemy w prawo).

Egzamin 2016/2017

[2pkt.] Zadanie 2. Dana jest SkipLista przechowująca liczby z przedziału (0,1). Proszę opisać (bez implementacji) algorytm, który z takiej SkipListy usuwa wszystkie liczby z zadanego przedziału (x, y) (gdzie $0 < x < y < 1$). Proszę oszacować złożoność obliczeniową zaproponowanego algorytmu jako funkcję liczby n elementów w SkipLiście (przed operacją usuwania) oraz liczby d elementów, które zostaną usunięte. Proszę uzasadnić przedstawione oszacowanie złożoności. Algorytm powinien być możliwie jak najszybszy. Ocenie podlega poprawność i efektywność algorytmu (1pkt) oraz poprawność oszacowania złożoności czasowej (1pkt).

Rozwiązanie:

Tworzymy wskaźnik A na najwyższym poziomie skiplisty i idziemy nim do przodu, dopóki następnym element nie będzie wpadał do przedziału (x,y), który mamy usunąć. Wtedy tworzymy drugi wskaźnik B i przechodzimy nim do przodu w obrębie tego przedziału (x,y), dopóki następny element nie będzie już poza nim, czyli w [y,1) - ten element już nie ma być usunięty. Następnikiem A ma być następny element B, ten, który już nie wpada do (x,y) - przepinamy odpowiednio wskaźnik na następny w A.

Schodzimy następnie oboma wskaźnikami o 1 poziom w dół i sprawdzamy, czy któregoś z nich nie da się przesunąć do przodu. Może się tak stać, gdyż każdy niższy poziom ma więcej node'ów, więc jest "gęstszy" - może się okazać, że między A i przedział (x,y) wcisnął się element, który nie ma być usunięty, albo też po B jest teraz element, który jeszcze wpada do przedziału (x,y) i trzeba go usunąć. Po przesunięciu wskaźników przepinamy wskaźnik na następny element z A i schodzimy o poziom w dół. Proces powtarzamy, aż zrealizujemy go na poziomie 0 - wtedy skończyliśmy.

Złożoność: przejście po skipliście to O(log(n)), należy też uwzględnić długość przedziału (x,y), czyli d - o tyle łącznie musimy przesunąć wskaźniki. Złożoność algorytmu to zatem O(log(n)+d).

Uwaga: w treści zadania nie ma wyraźnie zaznaczone, czy pamięć należy zwolnić, algorytm usuwa tylko te node'y z listy, a nie z pamięci. Jeśli chcielibyśmy po sobie posprzątać, to na poziomie 0, jeszcze przed

przepięciem wskaźnika na następny z A trzeba zrobić wskaźnik tmp, przelecieć nim po przedziale (x,y) aż do B i usuwać kolejne node'y. Nie zmieni to złożoności, gdyż jest to tylko +d.

Egzamin 2014/2015, termin I

Zadanie 1.2

Zadanie:
Drzewo BST jest opisane przez strukturę:

```
struct BST
{
    BST *left;
    BST *right;
    int value;
};
```

Proszę zaimplementować funkcję **double average (BST *T)**, która oblicza średnią wartość elementów w drzewie T.

Rozwiązanie:
Wykorzystywana jest pomocnicza funkcja rekurencyjna przechodząca przez drzewo in-order, zliczająca (dzięki przekazaniu przez referencję) sumę elementów i liczbę elementów. Sama funkcja average musi tylko wtedy wyliczyć iloraz sumy i liczby elementów, które dostaje z funkcji rekurencyjnej.

Zadanie 2

Zadanie:
Dana jest struktura danych opisująca tablicę haszującą, która przechowuje liczby typu int indeksowane napisami:

```
struct HT
{
    string *key;   //tablica na klucze danych
    int *data;     //tablica na dane
    bool *free;    //pole wolne czy zajęte
    int size;      //rozmiar tablicy
};
```

Tablica wykorzystuje funkcję haszującą **int hash (string key, int size)**, która zwraca pozycję w tablicy, na której powinny się znaleźć dane o kluczu **key**. Stosowane jest liniowe rozwiązywanie kolizji. Proszę zaimplementować funkcję **double averageAccess (HT *ht)**, która oblicza, jaka jest średnia ilość pól w tablicy, które musi sprawdzić standardowy algorytm wyszukujący, gdy poszukuje losowo wybranego klucza znajdującego się w tablicy **ht**.

Rozwiązanie:
Najłatwiej jest zrobić nowy HT, po kolei haszować dane z data, które mieliśmy dane i zliczać przy tym kroki. Potem wystarczy zwrócić iloraz liczby kroków i rozmiaru tablicy.

Egzamin 2014/2015, termin II

Zadanie 1.2

[**4pkt.**] Drzewo BST jest opisane przez strukturę `struct BST{ BST *left, *right; int value; };`. Proszę zaimplementować funkcję `int countInterval( BST* T, int a, int b )`, która oblicza ile liczb z zadanegego przedziału domkniętego $[a, b]$ znajduje się w drzewie T.

Rozwiązanie:

Zadanie jest podobne zadania 2A z kolokwium 2013/2014. Tutaj będzie nam jednak potrzebna funkcja pomocnicza int get_size (node *root), która zwraca rozmiar drzewa z korzeniem w danym węźle (włącznie z nim). Jest to funkcja rekurencyjna zwracająca sumę 1+leftSubtreeSize+rightSubtreeSize.

Jeśli x=y, to zwracamy 1. W przeciwnym wypadku idziemy wskaźnikiem po drzewie tak długo, jak wartość w węźle jest różna od x i y, a także jest jednocześnie większa od x i y lub jednocześnie mniejsza od x i y. W ten sposób kiedy te warunki nie zostaną spełnione, to nasz wskaźnik będzie na szczególnym węźle - jest on rootem najmniejszego poddrzewa zawierającego x i y.

Jeśli x jest tym węzłem, to dodajemy 1 do licznika i dodajemy tylko zliczenie y (czym ono jest - wyjaśnienie poniżej). Jeśli y jest tym węzłem, to dodajemy 1 do licznika i dodajemy tylko zliczenie x. Jeśli żaden z tych wypadków nie nastąpił, to x musi znajdować się w lewym poddrzewie, a y w prawym poddrzewie. Zliczenie końcoweto: 1 + zliczenie x dla root->left + zliczenie y dla root->right.

Zliczanie od strony x przebiega następująco:

1. Jeśli x znajduje się w lewym poddrzewie, to zwiększamy licznik o 1 i get_size(ptr->right), a następnie idziemy w lewo. Robimy tak, gdyż wszystkie wartości w prawym poddrzewie muszą być większe od x (a mniejsze od y z własności BST).
2. Jeśli x znajduje się w prawym poddrzewie, to idziemy w prawo.
3. Kiedy dojdziemy do x, to zwiększamy licznik o 1 i get_size(ptr->right) i kończymy.

Zliczanie od strony y przebiega podobnie:

1. Jeśli y znajduje się w prawym poddrzewie, to zwiększamy licznik o 1 i get_size(ptr->left), a następnie idziemy idziemy w prawo. Robimy tak (symetrycznie do sytuacji z x), gdyż wszystkie wartości w lewym poddrzewie muszą być mniejsze od y (a większe od x).
2. Jeśli y znajduje się w lewym poddrzewie, to idziemy w lewo.
3. Kiedy dojdziemy do y, to zwiększamy licznik o 1 i get_size(ptr->right) i kończymy.

Zadanie 2

[10pkt.] **Zadanie 2.** Dana jest struktura danych opisująca tablicę haszującą, która przechowuje liczby typu int indeksowane napisami:

```
struct HT{
  string* key;    // tablica na klucze danych
  int   * data;   // tablica na dane
  bool  * free;   // pole wolne czy zajete
  int     size;   // rozmiar tablicy
};
```

Tablica wykorzystuje funkcję haszującą `int hash( string key, int size)`, która zwraca pozycję w tablicy, na której powinny się znaleźć dane o kluczu `key`. Stosowane jest liniowe rozwiązywanie kolizji. Niestety możliwe, że tablica zawiera błędne dane. Proszę zaimplementować funkcję `bool checkHT( HT* ht )`, która sprawdza czy dla każdego klucza umieszczonego w tablicy faktycznie możliwe jest jego odszukanie standardowym algorytmem używanym w tablicach haszujących z liniowym rozwiązywaniem konfliktów.

Rozwiązanie:

Wystarczy po kolei haszować klucze i sprawdzać, czy otrzymany wynik da się znaleźć w tablicy haszującej. Przy pierwszej napotkanej niezgodności zwracamy false; jeśli nie napotkaliśmy żadnej do końca, to zwracamy true.

Egzamin 2014/2015, termin III

Zadanie 3

[10 pkt.] **Zadanie 3**. Dana jest następująca struktura opisująca drzewo:

```
struct Tree{
        Tree *parent;          //rodzic, lub NULL jeśli to korzeń
        Tree *left, *right;    //lewe I prawe dziecko
        Int w_left, w_right    //wagi krawędzi do lewego i do prawego dziecka (dodatnie)
}
```

Proszę opisać (bez implementacji) możliwie jak najszybszy algorytm, ktory mając na wejściu drzewo opisane przez strukturą Tree znajduje długość najdłuższej ścieżki między dwoma węzłami drzewa (prosta ścieżka to taka, która nie odwiedza żadnego węzła więcej niż raz). (Na potrzeby algorytmu mogą państwo uzupełnić strukturę Tree o dalsze pola.)

Rozwiązanie:

Będziemy wykorzystywać funkcję rekurencyjną, która będzie zwracała zawsze 2 informacje o węźle:

- max_length: długość najdłuższej ścieżki od dołu drzewa do node'a,
- path_length: długość najdłuższej ścieżki pomiędzy node'ami w obrębie drzewa, którego rootem jest node.

Pierwsza informacja mówi, jaka jest długość pionowej ścieżki, kiedy chcemy iść od liści do tego node'a, a druga, jaka jest maksymalna odległość między 2 węzłami w całym poddrzewie, którego rootem jest node. Potrzebne są obie z nich, żeby obsłużyć wszystkie możliwe wyjątki.

Funkcja zaczyna w roocie, w każdym wywołaniu rekurencyjnym wykonując kroki:

1. Wywołuje siebie rekurencyjnie dla lewego i prawego syna, uzyskując informacje:
   - maksymalna długość od liści do lewego syna,
   - maksymalna odległość między 2 węzłami w poddrzewie lewego syna,
   - maksymalna długość od liści do prawego syna,
   - maksymalna odległość między 2 węzłami w poddrzewie prawego syna.
2. Dzięki tym informacjom możemy sprawdzić maksymalną odległość między 2 dowolnymi węzłami z całego drzewa tworzonego przez node, w którym teraz jest funkcja, biorąc maksimum z:
   - maksymalna odległość między 2 węzłami w poddrzewie lewego syna - może się okazać, że tam jest największa i niezależnie od tego, co byśmy do tego dodali z pozostałej części drzewa, to nie dostaniemy nic więcej. Ścieżka ta **nie** przechodzi przez obecnie rozpatrywanego roota,
   - maksymalna odległość między 2 węzłami w poddrzewie lewego syna - powód analogiczny do opisanego powyższego, ta ścieżka także **nie** przechodzi przez obecnie rozpatrywanego roota,
   - suma: maksymalnej długości od liści do lewego syna, odległości między lewym synem a rootem, odległości między prawym synem a rootem i maksymalnej długości od liści do prawego syna - te 4 informacje dają nam odległość najdłuższej ścieżki, która **przechodzi** przez obecnie rozpatrywanego roota.
3. Brakuje nam jeszcze tylko jednej informacji, którą mamy zwracać: maksymalnej długości od liści do obecnie rozpatrywanego roota. Obliczamy ją, biorąc maksimum z:
   - maksymalna długość od liści do lewego syna + długość od lewego syna do obecnego roota,

- maksymalna długość od liści do prawego syna + długość od prawego syna do obecnego roota.

4. Zwracamy obie informacje.

Przechodzimy w ten sposób po drzewie, uzyskując potrzebne informacje dla poddrzew i finalnie dla roota całego drzewa. Wtedy wystarczy, że funkcja zwróci maksymalną odległość pomiędzy 2 dowolnymi węzłami drzewa.

Egzamin 2013/2014

Zadanie 2

Zadanie:
Proszę opisać algorytm (bez implementacji) dla następującego problemu: dana jest tablica **A** zawierająca n struktur typu struct Interval { int x,y; };. Elementy tablicy **A** opisują przedziały otwarte. Dana jest także liczba **int t**. Zadanie poleca na wypisaniu **t** lub mniej przedziałów, których suma daje spójny przedział o maksymalnej długości. Proszę podać złożoność czasową algorytmu i uzasadnić jego poprawność.
Rozwiązanie:

Na początek sortujemy daną tablicę po początkach indeksów, np. quicksortem, jest to nlog(n). Później idziemy od całej długości tablicy pętlą, która będzie szukać grup indeksów, których przedziały w sumie dają przedział spójny - rozróżnimy w ten sposób rozłączne sumy przedziałów. Indeksy lewy i prawy takich grup możemy trzymać w strukturze i przechowywać wyniki np. w liście. Po wykonaniu tej operacji na całej tablicy mamy przygotowane dane i możemy przystąpić do wykorzystywania drugiej funkcji, pomocniczej, dla każdego z tych przedziałów. Funkcja ta będzie zwracać strukturę, która zawierać będzie: listę indeksów pól w tablicy, które stanowią wynik i które trzeba wypisać oraz sumę najdłuższego spójnego przedziału. Funkcję tę wywołujemy dla każdego z przedziałów indeksów z naszej listy, za każdym razem sprawdzając, czy suma najdłuższego spójnego przedziału jest większa, niż największa do tej pory napotkana, jeśli tak - aktualizujemy ją oraz listę indeksów związaną z daną sumą. Na koniec wystarczy wypisać wszystkie przedziały pod indeksami tablicy zapisanymi w tej liście.

Funkcja pomocnicza będzie rozpatrywać kombinacje przedziałów z każdego zakresu indeksów z listy. Nie będzie rozpatrywać jednak wszystkich kombinacji - w pesymistycznym przypadku mogłyby to kombinacje bez powtórzeń z każdego z tych zbiorów indeksów, a taka złożoność jest absolutnie nieakceptowalna nawet w przypadku pesymistycznym. Rozpatrywane kombinacje mają 3 wyraźne ograniczenia: co najwyżej t-elementowe, suma przedziałów każdej musi być spójna, a także biorąc przedział do kombinacji wystarczy wziąć najdalej sięgający przedział mający część wspólną z dotychczasową sumą - krótsze dadzą mniejsze sumy i nie ma co się nimi przejmować.

Działanie funkcji jest następujące: rozważa kombinacje (najlepiej rekurencyjnie, choć można też iteracyjnie) przedziałów z danego zakresu indeksów, za każdym wywołaniem pamiętając o sprawdzeniu 3 warunków dla obecnie rozpatrywanego przedziału. Zawsze, kiedy dojdzie do t przedziałów (lub gdy w ogóle zakres indeksów to t lub mniej) w tymczasowej sumie, porównuje ją z największą napotkaną na tym przedziale indeksów do tej pory - jeśli jest większa, aktualizuje ją i listę indeksów, których przedziałów użyliśmy do jej stworzenia. Jeśli funkcja bierze kolejny przedział, to bierze taki z kolejnych przedziałów, który jest ma część wspólną z sumą i sięga najdalej (ma największy prawy koniec przedziału), dopisuje go do listy wykorzystywanych przedziałów i wywołuje się dla wziętego przedziału (jeśli po wzięciu go dalej spełnia warunki). Jeśli nie bierze przedziału (możemy rozpatrywać kombinacje, które nie biorą np. kilku środkowych przedziałów, a mimo tego spełniają warunki, albo biorą tylko przedziały z prawego końca danego zakresu indeksów), to po prostu idzie dalej.

Egzamin 2012/2013

Zadanie 2A

Zadanie:

[10 pkt.] Zadanie 2. Proszę zaimplementować algorytm, który mając na wejściu dwa drzewa BST (przechowujące liczby typu int; proszę zadeklarować odpowiednie struktury danych) zwraca nowe drzewo BST, zawierające wyłącznie te liczby, które występują w obu drzewach. Algorytm powinien być jak najszybszy i wykorzystywać jak najmniej pamięci. Jaka jest złożoność zaproponowanego algorytmu? Co można powiedzieć o zrównoważeniu drzew tworzonych przez zaproponowany algorytm?

Rozwiązanie:

Mając tylko tyle informacji, najlepiej będzie, jeśli rozwalimy oba istniejące drzewa i zbudujemy z nich nowe drzewo od zera, korzystając z listy dwukierunkowej.

Wykorzystamy do tego funkcję rekurencyjną: weź root, wepnij go do listy, na jego lewo wepnij jego lewe dziecko (wywołanie rekurencyjne), na jego prawo wepnij jego prawe dziecko (wywołanie rekurencyjne). Dzięki własności drzewa BST, że na lewo wartości muszą być mniejsze, a na prawo większe, to wpinając w ten sposób otrzymamy posortowaną listę dwukierunkową. Powtarzamy tę operację dla każdego z danych drzew, otrzymując 2 posortowane listy dwukierunkowe.

Teraz należy spełnić warunek zadania, czyli wziąć z obu drzew tylko te liczby, które występują w obu drzewach. W tym celu tworzymy trzecią listę dwukierunkową, którą będziemy uzupełniać w następujący sposób:

1. Ustawiamy wskaźniki na początek każdej z list.
2. Jeśli elementy są identyczne, to przepinamy jeden z nich do listy wynikowej i przesuwamy oba wskaźniki do przodu, usuwając też ten jeden, którego nie przepięliśmy.
3. Powtarzamy proces, aż skończymy którąś z list, wtedy usuwamy resztę tej drugiej.

Budowanie drzewa z trzeciej listy wygląda następująco: lista ma długość n, bierzemy z niej medianę (element n/2) i tworzymy z niego root'a nowego drzewa RB, kolorujemy go na czarno. Lewym dzieckiem staje się mediana z lewej części listy, analogicznie z prawym dzieckiem. Rekurencyjnie powtarzamy ten proces, aż zbudujemy całe drzewo.

Złożoność:

- pamięciowa: O(1), bo ilość wykorzystywanych przez nas węzłów jest co najwyżej równa ilości węzłów drzew, które mieliśmy dane,
- czasowa: O(n), bo każda z operacji (zamiana drzew na listy, wybranie elementów do trzeciej listy, zbudowanie drzewa) zajmuje O(n).

Drzewo wynikowe jest idealnie zrównoważone, gdyż budujemy je, korzystając zawsze z mediany - nie ma lepszej metody na zrównoważenie drzewa, mając z góry narzucone dane. W związku z tym jest nie tylko drzewem BST, ale także AVL.

Zadanie 2B

Zadanie:

[10 pkt.] Zadanie 2. Proszę zaimplementować algorytm, który mając na wejściu dwa drzewa BST (przechowujące liczby typu int; proszę zadeklarować odpowiednie struktury danych) zwraca nowe drzewo BST zawierające wyłącznie te liczby, które występują w dokładnie jednym z drzew (ale nie w obu). Algorytm powinien być jak najszybszy i wykorzystywać jak najmniej pamięci. Jaka jest złożoność czasowa zaproponowanego algorytmu? Co można powiedzieć o zrównoważeniu drzew tworzonych przez zaproponowany algorytm?

Rozwiązanie:

Analogicznie do zadania 2A, różni się tylko część z przeglądaniem list:

1. Ustawiamy wskaźniki na początek każdej z list.
2. Jeśli elementy są identyczne, to przesuwamy się dalej i je usuwamy.
3. Jeśli elementy są różne, to przepinamy mniejszy z nich do listy wynikowej i przesuwamy wskaźnik na jego liście o 1 do przodu.
4. Powtarzamy proces, aż wyczerpiemy jedną z list. Pozostałą część drugiej listy przepinamy do listy wynikowej.

Egzamin 2010/2011

Zadanie 1

Zadanie:

Masz za zadanie usprawnić pracę bazy danych, która ze względu na ilość osób z niej korzystających jest niewydolna czasowo. Proszę zaproponować i zaimplementować rozwiązanie, które będzie wspomagało bazę danych. Proszę uzasadnić wybór rozwiązania. Wiadomo, że użytkownik pobiera rekordy z bazy danych o kluczu imię i nazwisko / data urodzenia. Ilość rekordów jaka dziennie jest pobierana to około 10000, każdy rekord jest jednak pobierany kilkaset razy. Oszacuj złożoność czasową dostępu do danych w porównaniu do samej bazy oraz złożoność pamięciową zaproponowanego rozwiązania.

Rozwiązanie:

Treść zadania wyraźnie sugeruje użycie haszowania, odpowiednio według grupy po imieniu i nazwisku lub po dacie urodzenia. Zgodnie z treścią zadania wystarczyłoby użyć takiej funkcji haszującej każdej nocy, aby uporządkować ewentualne nowe dane. Oznacza to jednak, że poza tym, że zgodnie z treścią zadanie pobieranie rekordu z tablicy haszującej musi być szybkie, to jeszcze funkcja haszująca musi być na tyle szybka, żeby każdej nocy można było zhaszować całą bazę danych od nowa.

Ze względu na żądanie jak największej wydajności powinniśmy użyć metody liniowej, kwadratowej albo haszowania podwójnego. Najlepsza byłaby ta ostatnia, choć wymaga napisania kolejnej funkcji haszującej, to zapewnia potencjalnie najmniejszą liczbę kolizji (chociaż trochę wydłuża haszowanie całej bazy danych). Kwadratowa natomiast da średnio mniej kolizji od liniowej, która jest najprostsza z tych trzech, ale najmniej efektywna.

Sama funkcja haszująca powinna wykorzystywać efektywne metody haszowania: operację XOR, przesunięcia bitowe czy mnożenia o dość duże liczby pierwsze. Zmniejszy to liczbę kolizji, jeśli będzie ona dobrze napisana.

Złożoność czasowa to zwykle O(1), w pesymistycznym przypadku O(n), ale występuje on bardzo rzadko. Złożoność pamięciowa to O(k), gdzie k to ilość rekordów przechowywanych w bazie, ponieważ wielkość tablicy haszującej rośnie liniowo względem ilości tych danych.

Opracował Jakub Adamczyk