Laboratorium AiSD

Lista 9

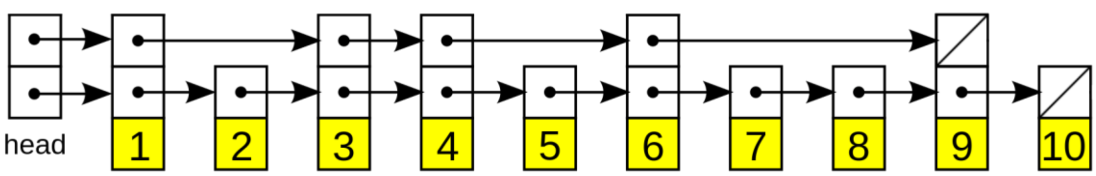
Struktury drzewiaste cz. 2

Proszę pamiętać, że **część rozwiązania** zadania stanowi również **zestaw testów** zaimplementowanych algorytmów i/lub struktur danych. Dodatkowo, proszę zwracać uwagę na **powtarzające się fragmenty** kodu i wydzielać je do osobnych funkcji/klas.

Wstęp – listy z przeskokami

Jak było omawiane na wykładzie nr 7, **binarne drzewa przeszukiwań** pozwalają na szybkie, w porównaniu do pozostałych poznanych struktur danych, wstawianie, wyszukiwanie oraz usuwanie elementów w czasie ***O(log N)***, zależnym od wysokości drzewa. Niestety, ta podstawowa struktura bardzo łatwo potrafi zostać zdegradowana, co redukuje jej efektywność do czasu liniowego. Istnieją dedykowane **mechanizmy zachowywania balansu** implementowane w strukturach takich jak: drzewa AVL, B-drzewa, czy drzewa czerwono-czarne, ale są one znacznie bardziej skomplikowane, przez co przeważnie wykorzystywane są już istniejące implementacje. Istnieją jednak struktury danych, które pozwalają na uzyskanie **średniej logarytmicznej złożoności** operacji będąc przy tym prostsze do napisania. Przykładem takiej struktury danych jest lista z przeskokami.

Lista z przeskokami (*ang. Skip List*) jest **probabilistyczną strukturą danych** stanowiącą modyfikację podstawowej, **uporządkowanej listy jednokierunkowej**. **Modyfikacja** polega na rozszerzeniu wierzchołka o **dodatkowe referencje** do dalszych węzłów listy. Dzięki temu, uwzględniając fakt, że lista jest zawsze uporządkowana, można szybko przeskakiwać duże jej fragmenty. Przykładową, najprostszą strukturę listy z przeskokami przedstawiono na Rysunku 1.



Rysunek 1 Przykładowa lista z przeskokami o dwóch poziomach [1]

Jak pokazano na Rysunku 1, każdy z węzłów listy przechowuje wartość oraz **przynajmniej jedną referencję** na kolejny węzeł. Przykładowa lista zawiera **dwa poziomy** (licząc od dołu) – poziom 0, będący **zwykłą listą jednokierunkową** oraz poziom 1 zawierający: głowę oraz elementy o wartościach 1, 3, 4, 6 i 9. Każdy wyższy poziom zawiera **co najwyżej** tyle elementów, co poziom niższy, więc pozwala on **szybsze przeskoczenie fragmentu** listy.

Przykładowo, chcąc znaleźć element o wartości 4 można przejść poziomem nr 1 do węzłów: 1, 3 i 4 pomijając przy tym węzeł o wartości 2.

Jaki otrzymujemy zysk w kontekście złożoności? Dla **listy uporządkowanej** dowolna operacja na niej ma złożoność liniową ***O(N)***. W **przykładowej** liście z przeskokami, można sterować **długością przeskoku** oraz ich **liczbą** na poziomie 1. Balans uzyskuje się przyjmując długość przeskoku równą $\sqrt{N}$, co daje jednocześnie $\sqrt{N}$ węzłów na poziomie 1 (liczba przeskoków razy ich długość musi dać liczbę elementów listy). Oznacza to, że taka lista ma złożoność operacji rzędu ***O(***$\sqrt{N}$***)***. Dodając **większą liczbę poziomów**, dla dużej liczby elementów, uzyskamy **średni czas** operacji rzędu ***O(log N)***.

Poniższe sekcje opisują sposób wykonywania operacji na liście z przeskokami.

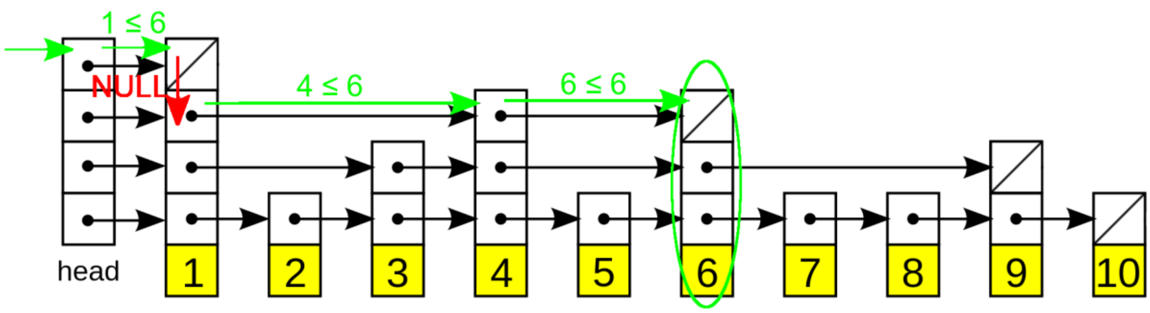
Wyszukiwanie

Podstawową operacją wykonywaną na listach z przeskokami jest wyszukiwanie. Jego idea sprowadza się do pozostawania na możliwie najwyższym poziomie listy do czasu, aż przeskok do kolejnego węzła nie spowodował by pominięcia fragmentu z szukaną wartością.

Algorytm wyszukiwania składa się z poniższych kroków:

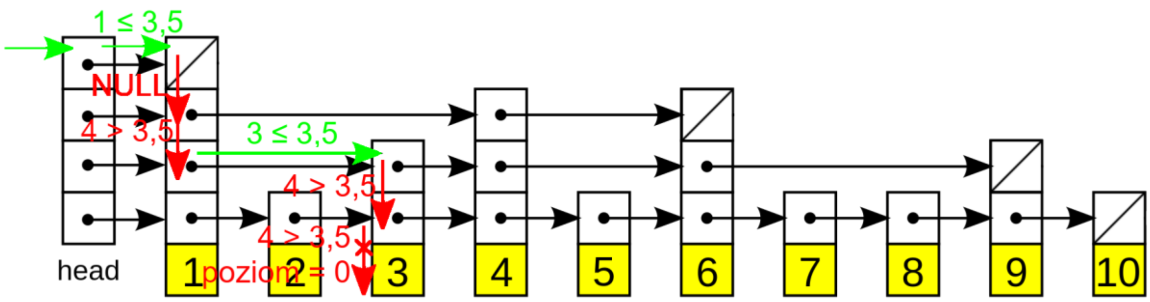
1. Rozpocznij na najwyższym poziomie w głowie listy,
2. Powtarzaj dopóki nie znaleziono węzła lub poziom >= 0:
    a. Jeżeli referencja na kolejny węzeł jest pusta, zejdź na niższy poziom,
    b. Jeżeli wartość w kolejnym węźle jest większa (większa równa) od szukanej, zejdź na niższy poziom,
    c. Jeżeli wartość w kolejnym węźle jest mniejsza lub równa (mniejsza) szukanej, przejdź do tego węzła.
3. Jeżeli znaleziono węzeł o szukanej wartości – zwróć wartość, w przeciwnym przypadku zgłoś brak szukanej wartości.

Przykład przeszukania dla wartości 6 przedstawiono na Rysunku 2.



Rysunek 2 Wyszukiwanie elementu o wartości 6

Rysunek 3 przedstawia sytuację, gdy szukana wartość (3,5) nie znajduje się na liście.



Rysunek 3 Wyszukanie nieistniejącej wartości 3,5

Należy zwrócić uwagę, że próba wyszukania wartości 3,5 kończy się niepowodzeniem, gdyż algorytm starał się zejść **poniżej poziomu 0**.

Powyższa wersja algorytmu wyszukiwania zawiera pewną **optymalizację**. W **podstawowej wersji**, wyszukiwanie zawsze schodzi na poziom 0, jednak jak widać na Rysunku 2 wartość 6 znaleziono będąc już na poziomie 2. **Wersję podstawową** można otrzymać **zamieniając nierówności** w punktach b i c kroku 2 algorytmu, na te podane w nawiasach. Różnica bierze się stąd, że podstawowa wersja wyszukiwania może być używana w pozostałych operacjach. Niniejszy opis zawiera jednak wersje operacji, które takiego wymogu nie mają.

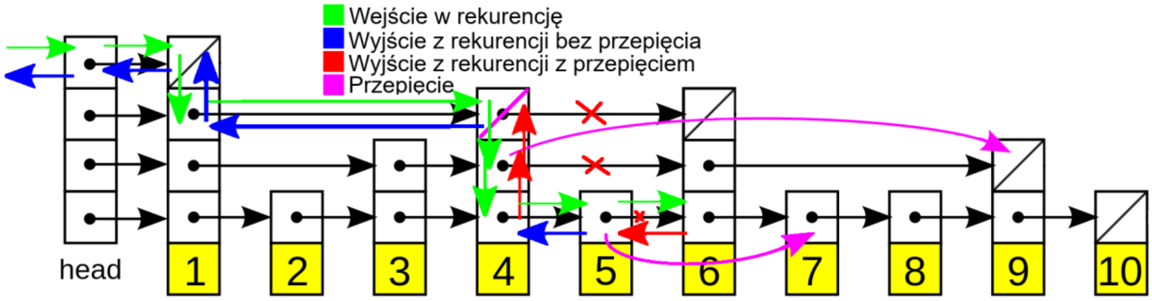
Usuwanie

Algorytm usuwania węzła jest w zasadzie identyczny do swojego odpowiednika dla list jednokierunkowych – **znajdź węzeł** i **przepnij** do jego poprzednika węzeł kolejny. Oczywiście **trudność** leży w tym, że lista z przeskokami posiada **wielokrotne połączenia**, które należy przepiąć do odpowiednich węzłów. Algorytm można zrealizować na kilka sposobów, np. istnieje wersja z pomocniczą tablicą przechowującą referencje na węzły do aktualizacji. [2] W niniejszym dokumencie przedstawiono jednak wersję rekurencyjną, która takiej dodatkowej pamięci nie wymaga.

Algorytm usuwania elementu:

1. Znajdź element na liście wykorzystując podstawową wersję algorytmu wyszukiwania (schodzącą na najniższy poziom),
2. Jeżeli węzła nie ma na liście – zakończ,
3. W przeciwnym przypadku, wracając z rekurencji metody wyszukiwania wykonuj następujące kroki:
    a. Jeżeli wracasz do węzła, który jest pierwszym węzłem odwiedzanym na danym poziomie podczas powrotu – przepnij referencję z danego poziomu usuwanego węzła do odwiedzanego węzła,
    b. Jeżeli usuwany węzeł nie ma już więcej poziomów lub poziom się nie zmienił – nic nie rób.

Powyższy algorytm kończy się odpięciem danego węzła z listy na wszystkich zajmowanych przez niego poziomach. Przykładowe wykonanie dla wartości 6 przedstawiono na Rysunku 4.



Rysunek 4 Przykład usunięcia węzła o wartości 6

Zwróć uwagę, że na Rysunku 4 przepięcie następuje jedynie w momencie odwiedzenia pierwszego węzła na danym poziomie. Pozostałe węzły nie uczestniczą w przepinaniu.

Wstawianie

W odróżnieniu od przeszukiwania i usuwania, wstawianie **nie** jest algorytmem. Wynika to z faktu zastosowania **losowości** w trakcie tworzenia węzła. Ta losowość daje nam **statystyczną** gwarancję na **średnią** złożoność ***O(log N)***.

Dla danej listy z przeskokami ustalany jest parametr $p \in \langle 0, 1)$ określający **prawdopodobieństwo dodania kolejnego poziomu** do węzła. Na jego podstawie ustalana jest liczba poziomów dla nowo tworzonego węzła, przy czym szansa na „wyższe" węzły maleje wykładniczo (p^n, n – liczba poziomów). Wybór wysokości uzyskiwany jest na drodze rzutu monetą – poziomy dodawane są do czasu, aż nie wypadnie wynik o prawdopodobieństwie $1 - p$. Metoda opisana jest poniżej:

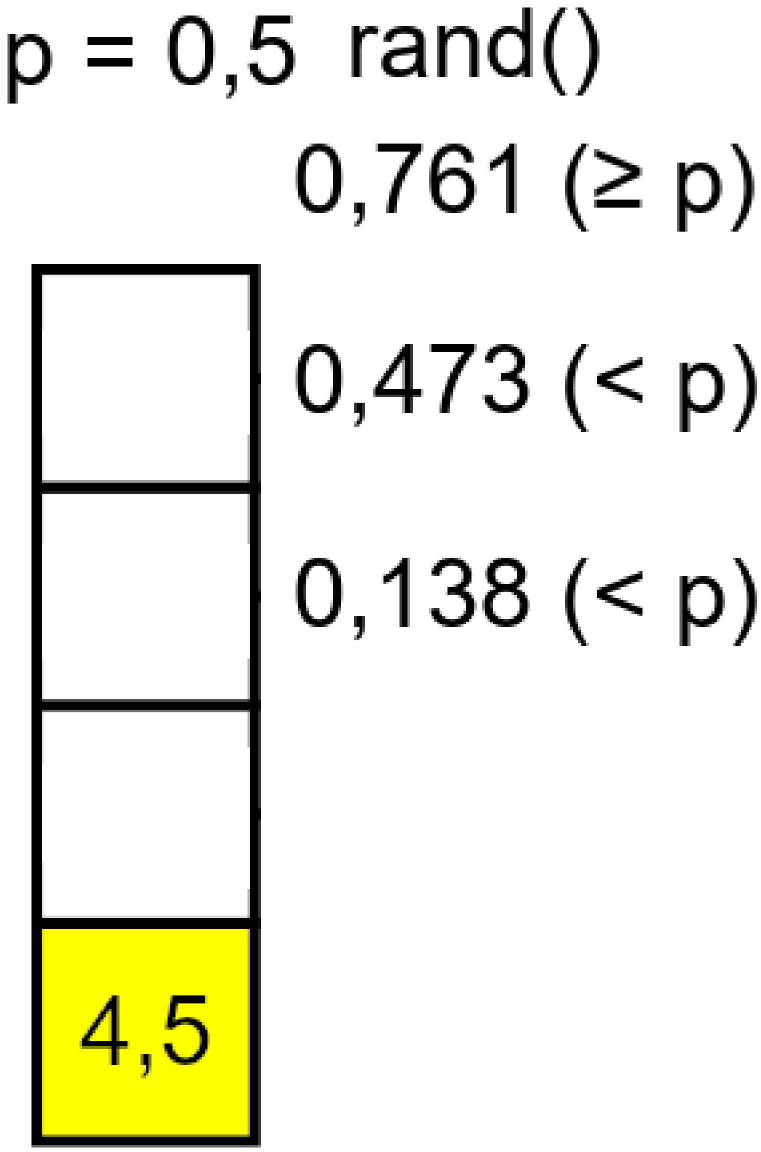
1. Liczba poziomów := 1
2. Dopóki rand() < p: Liczba poziomów := Liczba poziomów + 1
3. Zwróć Liczba poziomów

Funkcja rand() jest generatorem liczb pseudolosowych z przedziału $\langle 0, 1)$.

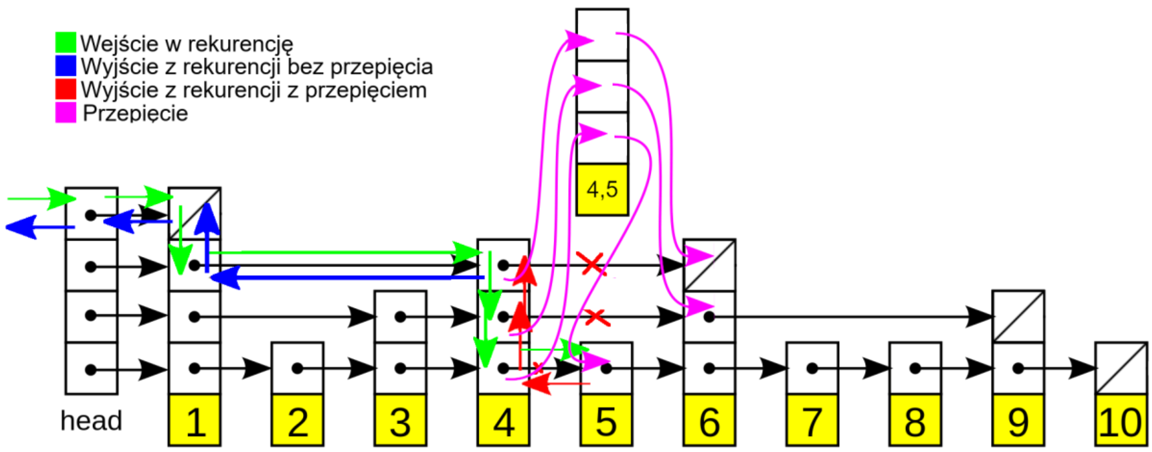
Mając ustaloną liczbę poziomów węzła, taki węzeł jest tworzony i zostaje wstawiony zgodnie z poniższym algorytmem:

1. Znajdź miejsce na liście, w którym element ma zostać wstawiony, wykorzystując <u>podstawową</u> wersję algorytmu wyszukiwania (schodzącą na najniższy poziom),
2. Jeżeli znaleziono węzeł o danej wartości – zgłosić błąd,
3. Jeżeli węzła nie znaleziono, to wracając z rekurencji wykonuj następujące czynności:
   a. Jeżeli wracasz do węzła, który jest pierwszym węzłem odwiedzanym na danym poziomie podczas powrotu – wepnij nowy węzeł między węzeł odwiedzany, a jego następnik na danym poziomie,
   b. Jeżeli poziom się nie zmienił – nic nie rób.
4. Jeżeli wrócono do głowy listy, a wstawiony węzeł ma jeszcze niepołączone poziomy – dodaj poziomy do głowy i podepnij do nich utworzony węzeł.

Jak widać, operacja wstawiania węzła jest analogiczna do usuwania węzła z listy. Przykład wstawienia wartości 4,5 przedstawiono na Rysunkach 4 i 5.



Rysunek 5 Wyznaczenie liczby poziomów nowego węzła



Rysunek 6 Przykład wstawienia elementu o wartości 4,5 do listy

Podsumowanie

Lista z przeskokami jest prostą, ale ciekawą strukturą danych pozwalającą na efektywne wstawianie, wyszukiwanie i usuwanie elementów. Wykorzystuje ona statystykę do zagwarantowania **średniej** złożoności O(log N). Oznacza to jednak, że efektywność **nie** jest zapewniona w każdym przypadku i ostatecznie może się zdarzyć, że operacje osiągną górną granicę ***O(N)***. Ze względu na wykorzystanie podstaw statystycznych, listy z przeskokami najlepiej działają pracy w sytuacji, gdy elementów jest bardzo dużo. Dzięki temu znalazły zastosowania w systemach zarządzania bazami danych.

Należy podkreślić, że zysk czasowy wiąże się jednak z **narzutem pamięciowym**. W średnim przypadku mamy złożoność ***O(N)***, a w najgorszym ***O(N log N)*** [1].

Zadania do wykonania

Celem listy jest **porównanie pod względem czasu** efektywności operacji wybranych struktur – **drzewa BST**, **drzewa czerwono-czarnego** oraz **listy z przeskokami**. Implementacje mają realizować **strukturę zbioru** – kolekcji przechowującej elementy i pozwalającej na sprawdzenie, czy dany element znajduje się w kolekcji, dodanie elementu oraz jego usunięcie.

1. Zdefiniuj interfejs ***ISet<T>*** dla kolekcji zboru. Ma on zawierać operacje:
   - *void insert(T element)* – operacja wstawienia elementu. Dodanie istniejącego elementu do zbioru **NIE** jest błędem,
   - *bool contains(T element)* – operacja sprawdzenia, czy element znajduje się w zbiorze,
   - *bool remove(T element)* – operacja usunięcia element ze zbioru. Wartość zwracana informuje, czy element znajdował się w zbiorze.
2. Wykorzystaj implementację drzewa ***BST<T>*** z poprzedniej listy do stworzenia klasy ***BSTSet<T>*** implementującą interfejs ***ISet<T>***, (10 pkt.)
3. Stwórz klasę ***RBSet<T>*** implementującą interfejs ***ISet<T>*** za pomocą **drzewa czerwono-czarnego**. Wykorzystaj **gotową implementację** tej kolekcji z biblioteki standardowej języka JAVA – ***TreeSet<T>*** [3], (10 pkt.)
4. Stwórz klasę ***SkipList<T>*** implementującą strukturę listy z przeskokami. Operacje listy mają być zrealizowane **<u>zgodnie z opisem</u>** podanym w części wprowadzającej tj. **nie zawierać** pomocniczych kolekcji węzłów do aktualizacji. Konstruktor klasy ma przyjmować parametr ***p***. Na jej podstawie stwórz kolekcję ***SkipSet<T>*** implementującą interfejs ***ISet<T>***, (50 pkt.)
5. Zmierz i przedstaw **czasy wykonania** operacji dla różnych rodzajów przygotowanych zbiorów. Przetestuj wpływ parametru $p \in \{0; 0{,}25; 0{,}5; 0{,}8\}$. W testach uwzględnij różną liczbę elementów umieszczanych w kolekcjach (zarówno dużą, jak i małą). Przygotuj wykresy. (30 pkt.)

Materiały

[1] https://en.wikipedia.org/wiki/Skip_list

[2] https://www.geeksforgeeks.org/skip-list/

[3] https://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html