Uniwersytet Zielonogórski
Instytut Sterowania i Systemów Informatycznych

Algorytmy i struktury danych

Laboratorium 7

Drzewa poszukiwań binarnych

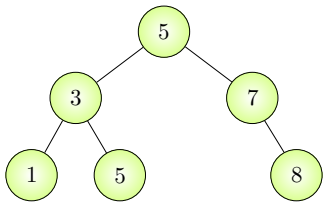
1 Cel ćwiczenia

Ćwiczenie ma na celu zapoznanie studentów z drzewami poszukiwań binarnych – BST (ang. *Binary Search Trees*). W szczególności omówione zostaną podstawowe operacje możliwe do wykonania na drzewach BST tj. przeszukiwanie, szukanie minimum i maksimum, wstawianie oraz usuwanie elementów drzewa.

2 Drzewa poszukiwań binarnych

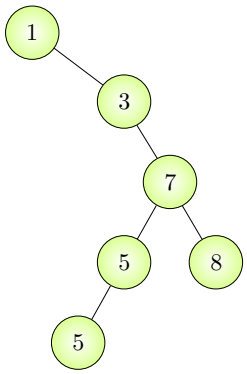
Do reprezentacji drzew BST można wykorzystać różne struktury tworzone w oparciu o zmienne dynamiczne. Podstawowe operacje na drzewach poszukiwań binarnych wymagają czasu proporcjonalnego do wysokości drzewa. W pełnym (tzw. zrównoważonym) drzewie binarnym o n węzłach takie operacje działają w najgorszym przypadku w czasie $\Theta(\lg n)$. Jeśli jednak drzewo składa się z jednej gałęzi o długości n, to te same operacje wymagają w pesymistycznym przypadku czasu $\Theta(n)$. Drzewo binarne spełnia następującą charakterystyczną własność, pozwalającą na szybkie wyszukiwanie jego elementów:

Niech x będzie węzłem drzewa BST. Jeśli y jest węzłem znajdującym się w lewym poddrzewie węzła x, to $klucz[y] \leq klucz[x]$. Jeśli y jest węzłem znajdującym się w prawym poddrzewie węzła x, to $klucz[x] \leq klucz[y]$. Inaczej mówiąc, dla każdego węzła x klucze znajdujące się w jego lewym poddrzewie są nie większe niż $klucz[x]$, a klucze w jego prawym poddrzewie są nie mniejsze niż $klucz[x]$ (Rys.1).



Rys. 1: Przykładowe drzewo BST

W korzeniu drzewa znajduje się klucz 5, klucze 1, 3 oraz 5 w jego lewym poddrzewie są nie większe niż 5, a klucze 7 oraz 8 znajdujące się w jego prawym poddrzewie są nie mniejsze niż 5. Ta sama własność jest spełniona w każdym węźle drzewa. Na przykład klucz 3 na rysunku jest nie mniejszy niż klucz 1 znajdujący się w jego lewym poddrzewie oraz nie większy niż klucz 5 znajdujący się w jego prawym poddrzewie. Ten sam zbiór wartości może być przedstawiony za pomocą wielu różnych drzew BST. Pesymistyczny czas działania większości operacji na drzewach BST jest proporcjonalny do wysokości drzewa. Na rysunku 1 przedstawiono drzewo BST o wysokości 2 składające się z 6 węzłów. Nieco mniej efektywne drzewo BST zawierające te same klucze może mieć wysokość 4 (Rys. 2). Wysokość obydwu drzew jest różna pomimo, iż są zbudowane z identycznych danych. Wysokość drzewa zależy od kolejności danych. Drzewo BST można zrealizować za pomocą struktury danych z dowiązaniami (np. listy), w której każdy węzeł jest obiektem. Oprócz pola *klucz* każdy węzeł zawiera pola *lewy*, *prawy* oraz *rodzic*, które wskazują odpowiednio na jego lewego syna, prawego syna oraz ojca. Jeśli węzeł nie ma następnika albo poprzednika, to odpowiednie pole ma wartość *nil*. Węzeł w korzeniu drzewa jest jedynym węzłem, którego pole wskazujące na ojca ma wartość *nil*.



Rys. 2: Przykładowe drzewo BST o wysokości 4

3 Przechodzenie drzewa BST

Własność drzewa BST umożliwia wypisanie wszystkich znajdujących się w nim kluczy w sposób uporządkowany za pomocą prostego algorytmu rekurencyjnego zwanego przechodzeniem drzewa metodą *INORDER*. Nazwa tego algorytmu wynika stąd, że klucz korzenia poddrzewa zostaje wypisany między wartościami z jego lewego poddrzewa a wartościami z jego prawego poddrzewa. Podobnie algorytm przechodzenia drzewa metodą *PREORDER* wypisuje klucz korzenia przed wypisaniem wartości znajdujących się w obu poddrzewach, a algorytm przechodzenia drzewa metodą *POSTORDER* wypisuje klucz korzenia po wypisaniu wartości znajdujących się w poddrzewach.

4 Wyszukiwanie węzła w drzewie BST

Do wyszukiwania w drzewie BST węzła, który zawiera dany klucz, można użyć następującej procedury. Mając dany wskaźnik do korzenia drzewa oraz klucz k, procedura `SZUKAJ` wyznacza wskaźnik do węzła zawierającego klucz k, jeżeli taki węzeł istnieje. W przeciwnym razie daje w wyniku wartość *nil*.

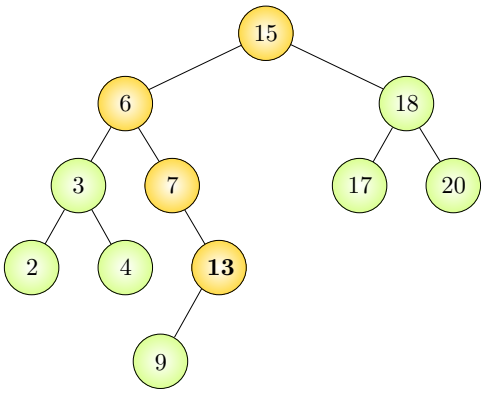
Algorytm 1: *Rekurencyjne wyszukiwanie węzła w drzewie*

```
SZUKAJ_REKURENCYJNIE(x,k)
1  if x = nil lub k = klucz[x]
2    then return x
3  if k < klucz[x]
4    then return SZUKAJ(lewy[x],k)
5    else return SZUKAJ(prawy[x],k)
```

Procedura ta rozpoczyna wyszukiwanie w korzeniu i schodzi po ścieżce w dół drzewa (Rys. 3). Dla każdego węzła x, który napotka, porównuje klucz k z wartością $klucz[x]$. Jeśli te wartości są równe, to wyszukiwanie zostaje przerwane. Jeśli $k < klucz[x]$, to dalsze wyszukiwanie przebiega już tylko w lewym poddrzewie węzła x, ponieważ z własności drzewa BST wynika, że k nie może znajdować się w prawym poddrzewie. Analogicznie, jeśli $k > klucz[x]$, to wyszukiwanie zostaje ograniczone do prawego poddrzewa. Liczba węzłów odwiedzonych podczas rekurencyjnego zejścia z korzenia drzewa, a zarazem czas działania procedury `SZUKAJ` wynosi $\Theta(h)$, gdzie h jest wysokością drzewa. Przedstawioną procedurę można zapisać iteracyjnie przez „rozwinięcie" rekursji w pętlę while. Na większości komputerów ta druga wersja jest efektywniejsza.

Algorytm 2: *Iteracyjne wyszukiwanie węzła w drzewie*

```
SZUKAJ_ITERACYJNIE(x,k)
1  while x <> nil i k <> klucz[x]
2     do if k < klucz[x]
3        then x = lewy[x]
4        else x = prawy[x]
5  return x
```



Rys. 3: Wyszukiwanie węzła w drzewie BST

Wyszukanie klucza 13 w drzewie na rysunku 3 wymaga przejścia po ścieżce 15-6-7-13. Najmniejszym kluczem w drzewie jest 2. Aby do niego dotrzeć, należy podążać zawsze według wskaźników *lewy*, począwszy od korzenia. Największy klucz (tutaj 20) w drzewie można znaleźć, podążając od korzenia według wskaźników *prawy*. Następnikiem węzła o kluczu 15 jest węzeł o kluczu 17, ponieważ jest to najmniejszy klucz w prawym poddrzewie węzła 15. Węzeł o kluczu 13 nie ma prawego poddrzewa, jego następnikiem jest więc najniższy przodek, którego lewy syn jest również przodkiem 13. Następnikiem węzła o kluczu 13 jest więc węzeł o kluczu 15.

5 Wyszukiwanie minimalnego i maksymalnego elementu w drzewie BST

Element w drzewie BST o najmniejszym kluczu można łatwo odszukać podążając według wskaźników *lewy*, począwszy od korzenia, aż napotkamy *nil* (Rys. 3). Następująca procedura wyznacza wskaźnik do minimalnego elementu poddrzewa o korzeniu w węźle x:

Algorytm 3: *Wyszukiwanie minimalnego elementu drzewa*

```
MINIMUM(x)
1  while lewy[x] <> nil
2     do x = lewy[x]
3  return x
```

Poprawność powyższej procedury wynika z własności drzewa BST. Jeśli węzeł x nie ma lewego poddrzewa, to najmniejszym elementem w poddrzewie o korzeniu x jest $klucz[x]$, ponieważ każdy klucz w jego prawym poddrzewie jest co najmniej tak duży jak $klucz[x]$. Jeśli natomiast węzeł x ma lewe poddrzewo, to najmniejszy klucz w poddrzewie o korzeniu x znajduje się na pewno w poddrzewie o korzeniu $lewy[x]$, bo każdy węzeł w lewym poddrzewie jest nie większy niż $klucz[x]$, a wszystkie klucze w prawym poddrzewie są nie mniejsze niż $klucz[x]$. Procedura `MAKSIMUM` wyszukująca maksymalny element drzewa BST jest analogiczna, różnica jest tylko taka, że przesuwamy się po wskaźnikach prawy.

Algorytm 4: *Wyszukiwanie maksymalnego elementu drzewa*

```
MAKSIMUM(x)
1  while prawy[x] <> nil
2     do x = prawy[x]
3  return x
```

Obie procedury działają na drzewie o wysokości h w czasie $\Theta(h)$, ponieważ przechodzą tylko po ścieżce w dół drzewa.

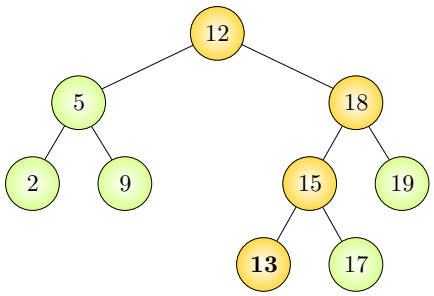
6 Wstawianie węzła do drzewa BST

Operacja wstawiania elementu zmienia zbiór dynamiczny reprezentowany przez drzewo BST. Ta zmiana wymaga przeorganizowania struktury drzewa, ale w taki sposób, aby została zachowana własność drzewa BST. Modyfikacja drzewa niezbędna przy wstawieniu nowego elementu jest dość łatwa, gorzej jest przy usuwaniu elementu. Nową wartość v można wstawić do drzewa poszukiwań binarnych T za pomocą procedury `WSTAW`. Do procedury przekazujemy jako argument węzeł z, w którym $klucz[z] = v$, $lewy[z] = nil$ oraz $prawy[z] = nil$. W wyniku wykonania procedury drzewo T oraz niektóre pola z są modyfikowane w sposób, który odpowiada wstawieniu z we właściwe miejsce w drzewie.

Algorytm 5: *Wstawianie węzła do drzewa*

```
WSTAW(T,z)
1  y = nil
2  x = korzen[T]
3  while x <> nil
4     do y = x
5        if klucz[z] < klucz[x]
6          then x = lewy[x]
7          else x = prawy[x]
8  rodzic[z] = y
9  if y = nil
10   then korzen[T] = z
11   else if klucz[z] < klucz[y]
12           then lewy[y] = z
13           else prawy[y] = z
```

Na rysunku 4 przedstawiony jest sposób działania procedury `WSTAW`. Podobnie jak w przypadku procedur `SZUKAJ` i `SZUKAJ_REKURENCYJNIE`, procedura `WSTAW` rozpoczyna przeglądanie w korzeniu, a następnie przebiega po ścieżce w dół drzewa. Wskaźnik x przebiega po ścieżce, a zmienna y zawiera zawsze wskazanie na ojca x. Po zainicjowaniu wartości zmiennych w pętli while w wierszach 3-7 wskaźniki x i y są przesuwane w dół drzewa w lewo lub w prawo, w zależności od wyniku porównania $klucz[z]$ z $klucz[x]$, aż do chwili, w której zmienna x przyjmie wartość nil. Ta właśnie wartość nil zajmuje miejsce w drzewie, w którym należy umieścić wskaźnik na węzeł z. Wstawienie z do drzewa (tzn. wiążące się z tym przypisania właściwych wartości odpowiednim wskaźnikom) odbywa się w wierszach 8-13 procedury.



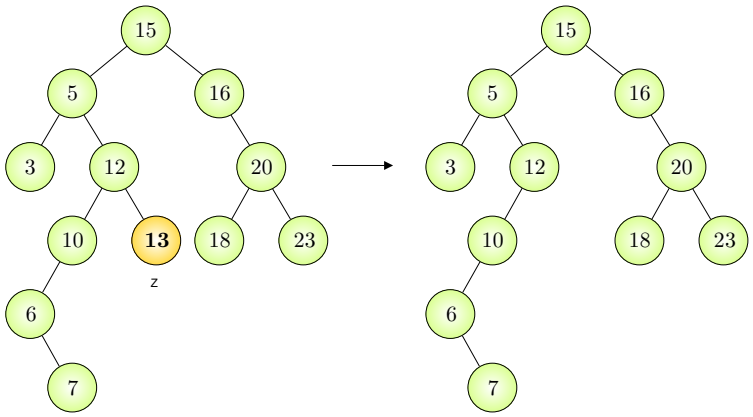
Rys. 4: Wstawianie węzła z kluczem 13 do drzewa BST

Na rysunku 4 kolorowe węzły wchodzą w skład ścieżki od korzenia do miejsca, w którym węzeł zostaje wstawiony. Kolorową linią jest oznaczony wskaźnik, który zostaje utworzony w wyniku dodania elementu do drzewa. Procedura `WSTAW`, podobnie jak inne elementarne operacje na drzewach poszukiwań, działa na drzewie o wysokości h w czasie $\Theta(h)$.

7 Usuwanie węzła z drzewa BST

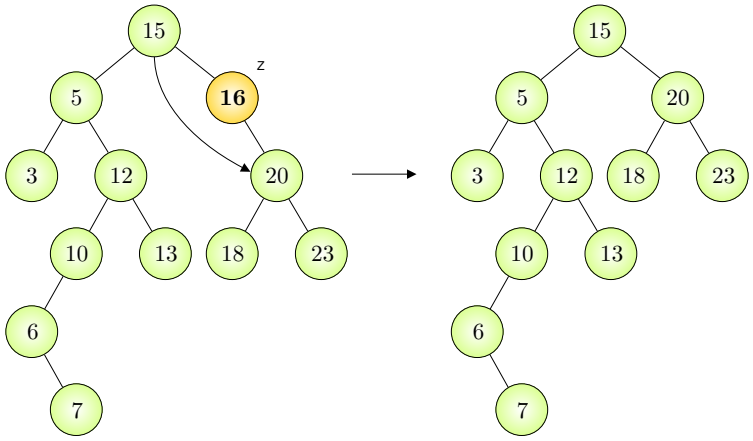
Argumentem procedury służącej do usuwania danego węzła z z drzewa poszukiwań binarnych jest wskaźnik do z. W procedurze tej rozpatrywane są trzy przypadki.

1. Jeśli z nie ma synów, to w jego ojcu $rodzic[z]$ zastępujemy wskaźnik do z wartością nil.



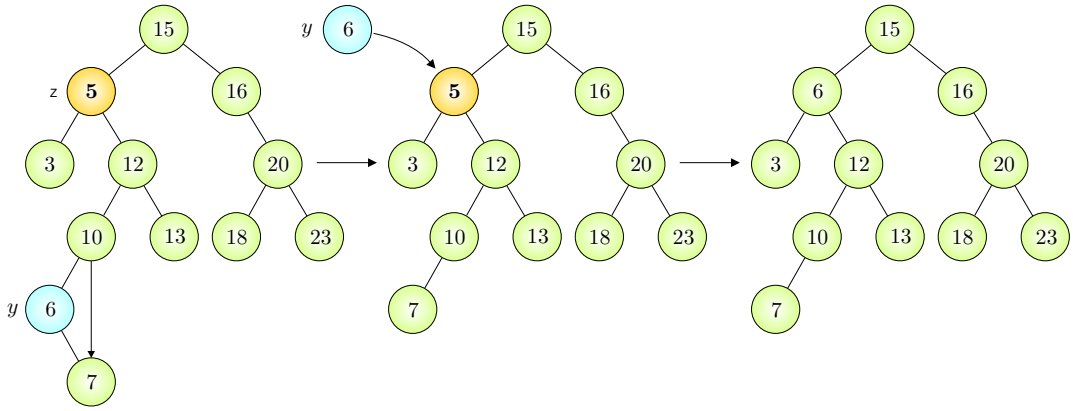
Rys. 5: Usuwanie węzła z z drzewa BST

2. Jeśli węzeł ma tylko jednego syna, to usuwamy z przez ustalenie wskaźnika między jego ojcem a jedynym synem.



Rys. 6: Usuwanie węzła z z drzewa BST

3. Jeśli węzeł ma dwóch synów, to usuwamy następnik y węzła z, o którym wiadomo, że nie ma lewego syna oraz zastępujemy zawartość z zawartością y.



Rys. 7: Usuwanie węzła z z drzewa BST

Procedura `USUN` realizuje wszystkie trzy wyżej przedstawione przypadki.

Algorytm 6: *Usuwanie węzła z drzewa*

```
USUN(T,z)
1  if lewy[z] = nil lub prawy[z] = nil
2    then y = z
3    else y = NASTEPNIK(z)
4  if lewy[y] <> nil
5    then x = lewy[y]
6    else x = prawy[y]
7  if x <> nil
8    then rodzic[x] = rodzic[y]
9  if rodzic[y] = nil
10   then korzen[T] = x
11   else if y = lewy[rodzic[y]]
12          then lewy[rodzic[y]] = x
13          else prawy[rodzic[y]] = x
14 if y <> z
15   then klucz[z] = klucz[y]
16   // Jeśli y ma inne pola,
        to je także należy skopiować.
17 return y
```

W wierszach 1-3 procedury zostaje wyznaczony węzeł y, który zostanie usunięty z drzewa. Węzeł y może być albo węzłem wejściowym z (jeśli z ma co najwyżej jednego syna), albo następnikiem z (jeśli z ma dwóch synów). Następnie w wierszach 4-6 zmiennej x zostaje przypisana różna od nil wartość y albo wartość nil, jeśli y nie ma żadnych synów. Węzeł y zostaje usunięty w wierszach 7-13 przez odpowiednią modyfikację wartości wskaźników $rodzic[y]$ i x. Pełna implementacja operacji usunięcia węzła y okazuje się dość skomplikowana, ponieważ trzeba osobno rozpatrzyć szczególne przypadki: gdy $x = nil$ lub gdy y jest korzeniem. Wreszcie w wierszach 14-16 zawartość y zostaje przepisana do z (niszcząc poprzednią zawartość z), jeżeli następnik z został usunięty. W ostatnim wierszu do procedury wołającej zostaje przekazany węzeł y. Procedura ta może go wstawić na listę wolnych pozycji. Czas działania procedury `USUN` na drzewie o wysokości h wynosi $\Theta(h)$.

8 Ćwiczenia do wykonania

1. Wykonać ćwiczenia zawarte w protokole.
2. Na podstawie literatury dokonać analizy zastosowań drzew BST.

Bibliografia

[1] L.Banachowski i in.: *Algorytmy i struktury danych*, WNT, 1996.

[2] T.H.Cormen i in.: *Wprowadzenie do algorytmów*, WNT, 2000.

[3] A.Drozdek: *Struktury danych w jezyku C*, WNT, 1996.

[4] D.E.Knuth: *Sztuka programowania*, WNT, 2002.

[5] R.Sedgewick: *Algorytmy w C++*, Wydawnictwo RM, 1999.

[6] P.Wróblewski: *Algorytmy, struktury danych i techniki programowania*, HELION, 1996.