Podstawy Programowania 2

Drzewa wyszukiwań binarnych

Arkadiusz Chrobot

Katedra Systemów Informatycznych

15 maja 2024

Plan

Wstęp

Drzewa i *drzewa binarne* są nieliniowymi strukturami danych używanymi do przechowywania danych uporządkowanych hierarchicznie. Nieliniowość w ich przypadku oznacza, że każdy węzeł może mieć co najwyżej jednego poprzednika, nazywanego *rodzicem* lub *przodkiem* i kilku następników, nazywanych *potomkami*. Dla drzew binarnych liczba potomków pojedynczego węzła jest ograniczona do dwóch. Warto zwrócić uwagę, że drzewa binarne i drzewa są *dwoma różnymi strukturami danych*. Są one także podklasą *grafów*, które będą przedmiotem kolejnego wykładu.

Dzisiejszy wykład poświęcony jest *drzewom wyszukiwań binarnych* (ang. *binary search trees*), które w polskiej literaturze informatycznej nazywane są również *drzewami przeszukiwań binarnych* lub *drzewami poszukiwań binarnych*. W skrócie można je nazywać BST. Następne slajdy przedstawiają kilka definicji związanych z pojęciem drzewa binarnego i BST.

Definicje

Drzewo binarne

Drzewo binarne jest skończonym zbiorem węzłów, który albo jest pusty, albo składa się z węzła nazywanego *korzeniem* i dwóch rozłącznych drzew binarnych nazywanych *lewym* i *prawym* poddrzewem. Jeśli te poddrzewa nie są puste, to ich korzenie nazywamy odpowiednio *lewym potomkiem* i *prawym potomkiem* korzenia, a korzeń nazywamy ich *przodkiem*. *Stopień* każdego węzła (czyli liczba jego potomków) w drzewie binarnym nie przekracza dwa. Węzły o stopniu różnym od zera, to *węzły wewnętrzne*, a te o stopniu równym zero, to *liście*. Każdy węzeł ma również własność, którą nazywa się *poziomem*. Poziom korzenia wynosi 0, a poziom każdego innego węzła jest o 1 większy od poziomu korzenia w najmniejszym zawierającym go poddrzewie. *Wysokość* drzewa jest o jeden większa od maksymalnego poziomu jego węzłów. Z definicji wynika także, że drzewo binarne jest *drzewem płaskim* lub *drzewem uporządkowanym*, bo kolejność jego poddrzew jest istotna.

Definicje

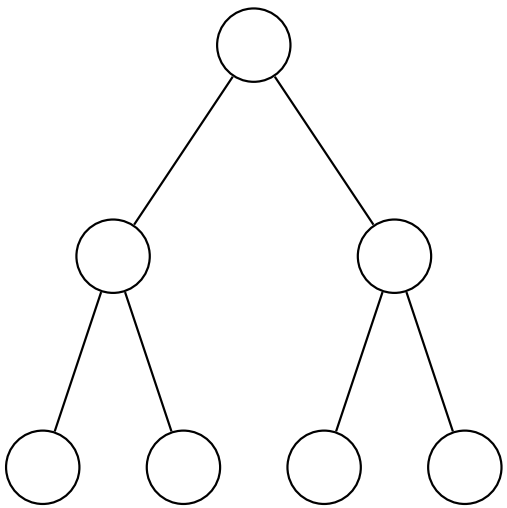
Drzewo binarne, a drzewo

Drzewo różni się od drzewa binarnego tym, że *zawsze* zawiera co najmniej jeden węzeł, a stopień każdego węzła nie jest ograniczony.

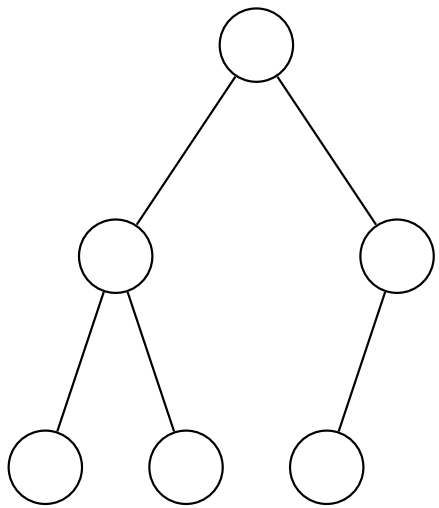
Definicje

Drzewa pełne i zupełne

Jeśli drzewo binarne o danej wysokości zawiera wszystkie możliwe węzły, to jest nazywane *drzewem pełnym*. Jeśli drzewo binarne o danej wysokości zawiera wszystkie możliwe węzły, poza kilkoma o maksymalnym poziomie, to jest to *drzewo zupełne*.



Drzewo binarne pełne



Drzewo binarne zupełne

Warto zauważyć, że w informatyce drzewa są rysowane korzeniem do góry.

Definicje

Drzewo BST

Drzewo wyszukiwań binarnych (BST) jest drzewem binarnym, w którym każdy węzeł zawiera daną nazywaną *kluczem*, z którą *może* być skojarzona *wartość*. Takie drzewo może posłużyć do budowy struktur słownikowych (ang. *dictionary*), w których klucz identyfikuje wartość. W BST klucze uporządkowane są według następującej reguły:

Uporządkowanie kluczy w BST

Niech x będzie węzłem BST. Jeśli y jest węzłem znajdującym się w lewym poddrzewie węzła x, to $\boldsymbol{klucz}(x) \geqslant \boldsymbol{klucz}(y)$. Jeśli y jest węzłem znajdującym się w prawym poddrzewie węzła x, to $\boldsymbol{klucz}(x) \leqslant \boldsymbol{klucz}(y)$.

Implementacja BST

Przedstawimy implementację BST w postaci dynamicznej struktury danych, która zgodna jest z umieszczoną na poprzednim slajdzie definicją. Zrobimy to na przykładzie programu, który tworzy BST zawierające jedynie klucze. Proszę zwrócić uwagę, że wspomniana definicja jest niejednoznaczna. Jeśli chcielibyśmy na niej oprzeć działanie operacji dodającej węzły do BST, to napotkalibyśmy na problem przy powtarzających się kluczach — czy węzeł zawierający taki sam klucz, jak inny, znajdujący się już w drzewie węzeł, ma być jego lewym, czy prawym potomkiem? Dlatego postąpimy tak, jak radzi większość opracowań na temat BST — stworzymy implementację, w której klucze są unikatowe.

Implementacja BST

Typ danych węzła BST

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #include<time.h>
4
5  struct bst_node
6  {
7          int key;
8          struct bst_node *left_child, *right_child;
9  } *root;
```

Implementacja BST

Typ danych węzła BST

Pliki nagłówkowe dodane na początku programu (wiersze 1–3) dostarczają deklaracji funkcji, które pozwalają obsługiwać standardowe wyjście, zarządzać pamięcią na stercie oraz używać generatora liczb pseudolosowych.

Zwróćmy uwagę, że definicja typu danych dla pojedynczego węzła BST (wiersze 5–6) przypomina podobny element programu, który używał listy dwukierunkowej. Zmieniło się jednak znaczenie składowych. Pole `key` służy do przechowywania wartości klucza, który będzie liczbą całkowitą typu `int`. W wierszu nr 8 zadeklarowano dwa pola wskaźnikowe o nazwach `left_child` i `right_child`. Będą one przechowywały adresy lewego i prawego potomka węzła, lub miały wartość `NULL`, jeśli ci potomkowie nie będą istnieć. Zwróćmy uwagę, że w przypadku, gdy oba te pola mają taką wartość, to węzeł jest liściem. W węzłach wewnętrznych co najwyżej tylko jedno z tych pól może mieć wartość `NULL`.

Implementacja BST

Typ danych węzła BST

W niektórych implementacjach BST występuje jeszcze dodatkowe pole wskaźnikowe, zazwyczaj nazywane `parent`, które wskazuje na przodka danego węzła. Jedynie dla korzenia drzewa to pole ma wartość `NULL`.
W wierszu nr 9, na slajdzie nr 9 zadeklarowana jest zmienna globalna o nazwie `root`. Jest to wskaźnik, który będzie wskazywał na korzeń BST. Jego wartość początkowa wynosi `NULL`, co również oznacza, że BST jest początkowo puste.

Funkcja add_node()

```
1  void add_node(struct bst_node **node, int number)
2  {
3          while(*node && (*node)->key != number)
4                  if((*node)->key > number)
5                          node = &(*node)->left_child;
6                  else
7                          node = &(*node)->right_child;
8          if(!*node) {
9                  *node = (struct bst_node
   ↪   *)malloc(sizeof(struct bst_node));
10                 if(*node) {
11                         (*node)->key = number;
12                         (*node)->left_child =
   ↪   (*node)->right_child = NULL;
13                 }
14         }
15 }
```

Funkcja `add_node()`

Funkcja `add_node()` jest odpowiedzialna za dodanie nowego węzła z określonym kluczem do BST. Nie zwraca ona żadnej wartości, ale ma dwa parametry. Pierwszy jest wskaźnikiem na wskaźnik, o nazwie `node`. Argumentem podstawianym za ten parametr jest adres zmiennej `root`. Drugim parametrem funkcji jest zmienna typu `int`, o nazwie `number`. Przez ten parametr do funkcji będzie przekazywany klucz, który zostanie zapisany w nowym węźle.

Funkcja `add_node()`

Zadaniem pętli **`while`** (wiersze 3–7) jest wyszukanie miejsca w BST dla nowego węzła. Kryterium wyszukiwania jest klucz, który zostanie w nim zapisany. Pętla wykonuje się tak długo, jak długo wskaźnik wskazywany przez **`node`** nie jest pusty i węzeł wskazywany przez ten wskaźnik ma klucz inny, niż ten, który będzie w nowym węźle. Jeśli oba te warunki są spełnione, to w ciele pętli sprawdzane jest, czy klucz zawarty w węźle wskazywanym przez wskaźnik, którego adres znajduje się w **`node`**, jest większy od klucza, który ma być umieszczony w nowym węźle (wiersz nr 4). Jeśli tak, to do **`node`** przypisywany jest adres pola **`left_child`** węzła wskazywanego przez wskaźnik wskazywany przez **`node`** (wiersz nr 5). W lewym poddrzewie są mniejsze klucze i tam powinien trafić nowy. Jeśli zaś warunek z wiersza nr 4 nie jest spełniony, to do **`node`** zapisywany jest adres pola **`right_child`** węzła wskazywanego przez wskaźnik, którego adres znajduje się w **`node`**. Oznacza to, że nowy klucz powinien się znaleźć w prawym poddrzewie, gdzie są większe klucze.

Funkcja `add_node()`

Po zakończeniu pętli `while` funkcja `add_node()` sprawdza, czy wskaźnik wskazywany przez `node` **jest pusty** (wiersz nr 8). Jeśli nie, to będzie to znaczyło, że pętla zlokalizowała węzeł posiadający taki sam, klucz, jaki jest zawarty w parametrze `number`, a ponieważ przyjęliśmy, że klucze w BST nie będą się powtarzać, to `add_node()` zakończy działanie, nie podejmując żadnych dalszych działań. Jeśli jednak ten warunek jest spełniony, to ta funkcja spróbuje przydzielić pamięć na nowy węzeł (wiersz nr 9) i jeśli się to powiedzie, co jest sprawdzane w wierszu nr 10, to zapisze do nowego węzła klucz zawarty w parametrze `number` (wiersz nr 11) i zainicjuje oba jego pola wskaźnikowe wartością `NULL` (wiersz nr 12).

Zwróćmy uwagę, że jeśli pętla `while` od razu się zatrzyma, to będzie to oznaczało, że drzewo jest puste i `add_node()` doda pierwszy węzeł do BST. Jeśli jednak pętla wykona kilka iteracji i zakończy się, gdy wyrażenie `*node` będzie fałszywe, to będzie to znaczyło, że `node` zawiera adres jednego z pól wskaźnikowych węzła, który powinien być przodkiem nowego.

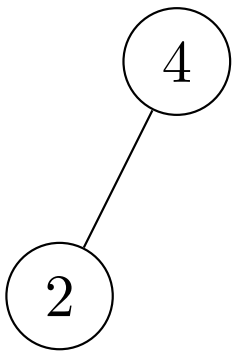
Dodawanie węzłów do BST

Następny slajd zawiera animację, która schematycznie pokazuje operację dodawania do BST węzłów, których klucze mają następującą kolejność: 4, 2, 1, 3, 5.

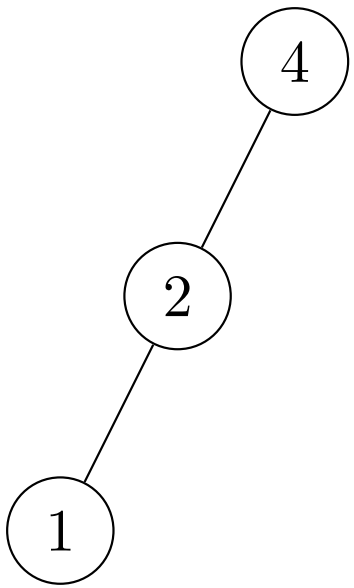
Dodawanie węzłów do BST

Dodawanie węzłów do BST

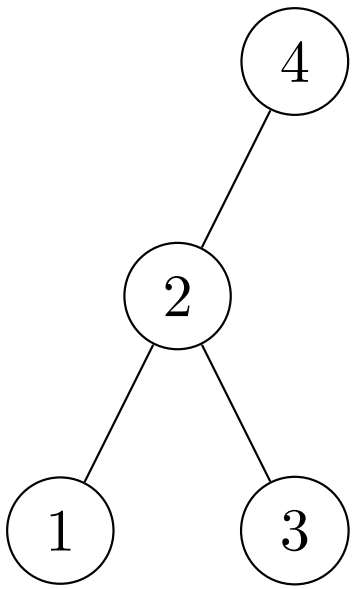
Dodawanie węzłów do BST



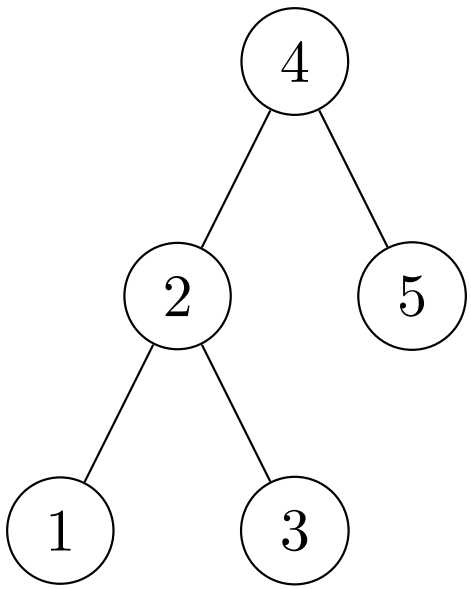
Dodawanie węzłów do BST



Dodawanie węzłów do BST



Dodawanie węzłów do BST



Przechodzenie drzewa binarnego

Istnieją trzy algorytmy rekurencyjne, według których realizowane jest odwiedzanie kolejnych poddrzew i w konsekwencji węzłów drzewa. Są to:

1. przechodzenie w porządku *inorder*, nazywanym inaczej porządkiem *wrostkowym*,
2. przechodzenie w porządku *preorder*, nazywanym inaczej porządkiem *przedrostkowym*,
3. przechodzenie w porządku *postorder*, nazywanym inaczej porządkiem *przyrostkowym*.

We wszystkich tych algorytmach względna kolejność odwiedzania poddrzew jest następująca: najpierw lewe poddrzewo, a następnie prawe. Wymienione algorytmy stanowią podstawę do opracowania innych algorytmów dotyczących drzew binarnych, w tym BST. Wszystkie z nich najprościej jest zaimplementować przy użyciu funkcji rekurencyjnych.

Przechodzenie drzewa binarnego

Porządek *inorder*

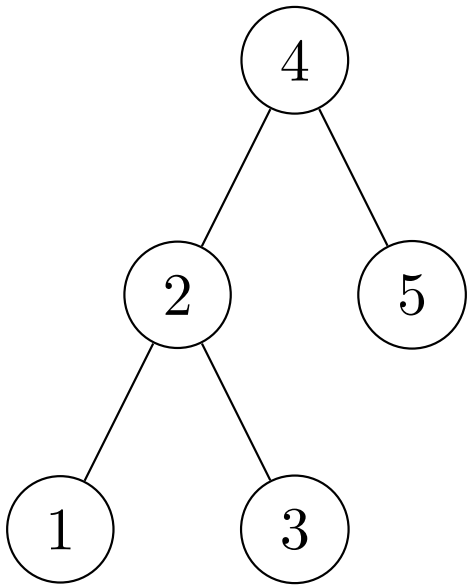
Algorytm przechodzenia drzewa binarnego w porządku inorder jest następujący:

1. odwiedź lewe poddrzewo,
2. odwiedź korzeń,
3. odwiedź prawe poddrzewo.

Następny slajd zawiera ilustrację przykładowego BST i wynik działania funkcji, która korzystając z algorytmu przechodzenia w porządku *inorder* wypisuje klucze na ekranie.

Przechodzenie drzewa

Porządek *inorder*



Wynik

1, 2, 3, 4, 5

Funkcja print_bst_inorder()

```
1  void print_bst_inorder(struct bst_node *node)
2  {
3          if(node) {
4                  print_bst_inorder(node->left_child);
5                  printf("%4d ",node->key);
6                  print_bst_inorder(node->right_child);
7          }
8  }
```

Funkcja `print_bst_inorder()`

Poprzedni slajd zawiera definicję funkcji `print_bst_inorder()`, która używając algorytmu przechodzenia BST w porządku *inorder* wypisuje zawarte w tym drzewie klucze na ekranie. Nie zwraca ona żadnej wartości, ale posiada parameter `node` będący wskaźnikiem na węzeł drzewa. Argumentem dla tej funkcji będzie adres korzenia drzewa.

W ciele funkcji sprawdzane jest najpierw, czy `node` nie jest wskaźnikiem pustym (wiersz nr 3). Jeśli ten warunek jest spełniony, to funkcja wywołuje się rekurencyjnie dla lewego potomka węzła wskazywanego przez `node` i w konsekwencji dla całego poddrzewa z nim związanego (wiersz nr 4). Po powrocie z tego wywołania rekurencyjnego funkcja wypisuje na ekranie klucz węzła wskazywanego przez `node` (wiersz nr 5) i ponownie wywołuje się rekurencyjnie (wiersz nr 6), ale tym razem dla prawego potomka węzła wskazywanego przez `node` i związanego z nim poddrzewa.

Funkcja `print_bst_inorder()`

Warto dodać, że wywołania rekurencyjne kończą się, gdy funkcja zostanie wywołana rekurencyjnie dla nieistniejącego węzła. W takim przypadku warunek w wierszu nr 3 nie będzie spełniony i ta instancja funkcji się zakończy, a sterowanie wróci do wcześniejszej instancji.

Przechodzenie drzewa binarnego

Porządek *preorder*

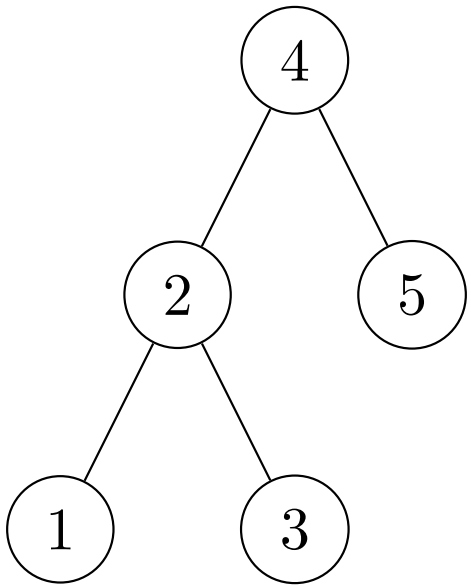
Algorytm przechodzenia drzewa binarnego w porządku *preorder* jest następujący:

1. odwiedź korzeń,
2. odwiedź lewe poddrzewo,
3. odwiedź prawe poddrzewo.

Różnica między nim, a *inorder* polega na tym, że korzeń jest odwiedzany w nim na początku, a dopiero potem poddrzewa. Kolejny slajd zawiera ilustrację z przykładowym BST i wynik działania funkcji, która korzystając z tego algorytmu wypisuje na ekranie klucze zawarte we wspomnianym drzewie.

Przechodzenie drzewa

Porządek *preorder*



Wynik

4, 2, 1, 3, 5

Funkcja print_bst_preorder()

```
1  void print_bst_preorder(struct bst_node *node)
2  {
3          if(node) {
4                  printf("%4d ",node->key);
5                  print_bst_preorder(node->left_child);
6                  print_bst_preorder(node->right_child);
7          }
8  }
```

Funkcja `print_bst_preorder()`

Funkcja `print_bst_preorder()` jest bardzo podobna do funkcji `print_bst_inorder()`. Jedyne dwie różnice między nimi, to nazwa oraz to, że wywołania rekurencyjne tej pierwszej (wiersze 5–6) następują po wypisaniu klucza węzła bieżąco wskazywanego przez `node` (wiersz nr 4).

Przechodzenie drzewa binarnego

Porządek *postorder*

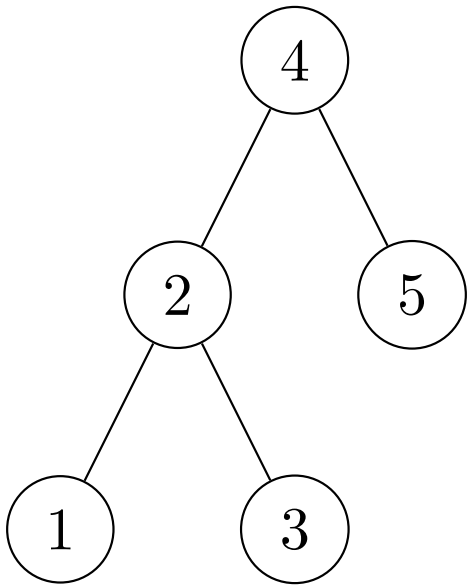
Algorytm przechodzenia drzewa binarnego w porządku *postorder* jest następujący:

1. odwiedź lewe poddrzewo,
2. odwiedź prawe poddrzewo,
3. odwiedź korzeń.

Różnica między nim, a poprzednimi polega na tym, że najpierw odwiedzane są oba poddrzewa (najpierw lewe, potem prawe), a na końcu korzeń. Kolejny slajd zawiera ilustrację z przykładowym BST i wynik działania funkcji, która korzystając z opisanego algorytmu wypisuje na ekranie klucze zawarte we wspomnianym drzewie.

Przechodzenie drzewa

Porządek *postorder*



Wynik

1, 3, 2, 5, 4

Funkcja print_bst_postorder()

```
1  void print_bst_postorder(struct bst_node *node)
2  {
3          if(node) {
4                  print_bst_postorder(node->left_child);
5                  print_bst_postorder(node->right_child);
6                  printf("%4d ",node->key);
7          }
8  }
```

Funkcja `print_bst_preorder()`

Funkcja `print_bst_postorder()` jest również bardzo podobna do dwóch zaprezentowanych wcześniej. Tutaj różnice również są w nazwie, oraz w tym, że funkcja wywołuje się rekurencyjnie dla potomków węzła wskazywanego przez `node` (wiersze 4–5), zanim wypisze klucz zapisany w tym węźle.

Liczba węzłów w BST

Wypisywanie na ekranie kluczy zawartych w drzewie, to nie jedyne zastosowanie algorytmów przechodzenia drzewa. Zastanówmy się jak policzyć ile węzłów jest w BST. Okazuje się, że pomóc nam może metoda dziel i zwyciężaj:

- jeśli drzewo jest puste, to liczba jego węzłów wynosi zero,
- jeśli drzewo nie jest puste, to liczba jego węzłów jest równa liczbie węzłów w jego lewym poddrzewie, korzeniowi (jeden węzeł) i liczbie węzłów w jego prawym poddrzewie.

Zwróćmy uwagę, że ostatni podpunkt odpowiada algorytmowi przechodzenia drzewa w porządku *inorder*, ale ponieważ dodawanie jest przemienne, to możemy zastosować dowolny z opisanych porządków.

Funkcja count_nodes()

```
1  unsigned int count_nodes(struct bst_node *node)
2  {
3          if(node)
4                  return count_nodes(node->left_child) +
    ↪  1 + count_nodes(node->right_child);
5          else
6                  return 0;
7  }
```

Funkcja `count_nodes()`

Zaprezentowana na poprzednim slajdzie funkcja wylicza liczbę węzłów BST używając algorytmu opisanego na slajdzie nr 32. Zwraca ona wartość typu `unsigned int` (liczba węzłów zawsze jest liczbą naturalną), a jako argument przyjmuje adres korzenia BST.

Funkcje find_minimum() i find_maximum()

```
1  struct bst_node *find_minimum(struct bst_node *node)
2  {
3          while(node && node->left_child)
4                  node = node->left_child;
5          return node;
6  }
7
8  struct bst_node *find_maximum(struct bst_node *node)
9  {
10         while(node && node->right_child)
11                 node = node->right_child;
12         return node;
13 }
```

Funkcje `find_minimum()` i `find_maximum()`

Zlokalizowanie w BST węzła o najmniejszym kluczu jest proste. To skrajnie lewy węzeł. Podobnie jest ze znalezieniem węzła o największym kluczu — to skrajnie prawy węzeł. Pierwsza z zaprezentowanych funkcji wyszukuje węzeł o najmniejszym kluczu i zwraca jego adres. Jako argument przyjmuje przez parameter `node` adres korzenia BST. Zawarta w niej pętla `while` sprawdza, czy parameter `node` nie jest wskaźnikiem pustym i czy pole `left_child` węzła przez ten parameter wskazywanego, też nie ma wartości `NULL` (wiersz nr 3). Jeśli oba wyrażenia są prawdziwe, to do `node` jest przypisywany adres zawarty we wspomnianym polu (wiersz nr 4), czyli po wykonaniu tej instrukcji `node` będzie wskazywał lewego potomka tego węzła. Pętla zakończy się, gdy znajdzie węzeł, który nie będzie miał lewego potomka. Fukcja `find_minimum()` zwróci jego adres, bo będzie to skrajnie lewy węzeł BST (wiersz nr 5). Zwróćmy uwagę, że może ona zwrócić `NULL` tylko w jednym przypadku — gdy zostanie wywołana dla pustego BST.

Funkcje `find_minimum()` i `find_maximum()`

Funkcja `find_maximum()` działa podobnie, ale w pętli `while` sprawdza, czy węzeł potencjalnie wskazywany przez `node` ma pole wskaźnikowe `right_child`, które nie jest pustym wskaźnikiem. Jeśli tak jest, to w wierszu nr 11 przypisuje do `node` adres przechowywany w tym polu. Po zakończeniu pętli funkcja zwraca adres węzła wskazywanego przez `node`, gdyż jest to adres skrajnie prawego węzła BST, który zawiera maksymalny klucz.

Funkcja `locate()`

```
1  struct bst_node *locate(struct bst_node *node, int
    ↪  number)
2  {
3          while(node && node->key != number)
4                  if(node->key > number)
5                          node = node->left_child;
6                  else
7                          node = node->right_child;
8          return node;
9  }
```

Funkcja `locate()`

Zadaniem funkcji `locate()` jest odnalezienie węzła o zadanym kluczu. Ten klucz jest przekazywany do niej przez parameter `number`, natomiast adres korzenia BST podstawiany jest pod parameter `node`. Funkcja zwraca adres znalezionego węzła lub wartość `NULL`, jeśli ten węzeł nie istnieje. Proszę zwrócić uwagę na podobieństwo pętli `while` zawartej w tej funkcji (wiersze 3–7), do analogicznej pętli z funkcji `add_node()`. Tym razem jednak ta pętla posługuje się wskaźnikiem pierwszego poziomu. Jeśli ten wskaźnik nie jest pusty i wskazuje on na węzeł, który nie zawiera szukanego klucza (wiersz nr 3), to funkcja sprawdza (wiersz nr 4), czy klucz zawarty w tym węźle jest większy od poszukiwanego. Jeśli tak, to w `node` zapisywany jest adres lewego potomka węzła (lub `NULL` jeśli on nie istnieje), a w przeciwnym przypadku zapisywany jest adres prawego potomka (lub `NULL` jeśli on nie istnieje). Po zakończeniu pętli wartość wskaźnika `node` jest zwracana przez funkcję.

Wyszukiwanie węzła o zadanym kluczu

Efektywność

Zaletą BST jest czas wyszukiwania węzła o zadanym kluczu, który jest proporcjonalny do wysokości tego drzewa. Jeśli *kształt* BST jest zbliżony lub taki sam jak kształt pełnego drzewa binarnego, to jego wysokość można wyliczyć ze wzoru $log_2(n)$, gdzie n oznacza liczbę wszystkich węzłów BST.

Funkcja remove_bst_nodes()

```
1  void remove_bst_nodes(struct bst_node **node)
2  {
3          if(*node) {
4                  remove_bst_nodes(&(*node)->left_child);
5                  remove_bst_nodes(&(*node)->right_child);
6                  free(*node);
7                  *node=NULL;
8          }
9  }
```

Funkcja `remove_bst_nodes()`

Funkcja usuwająca wszystkie węzły drzewa binarnego działa zgodnie z algorytmem przechodzenia BST w porządku *postorder*, bo tylko on gwarantuje, że usuwanie rozpocznie się od liści. Dzięki temu nie ma niebezpieczeństwa, że do rekurencyjnych wywołań funkcji będą przekazane adresy nieistniejących pól. Należy również zadbać o to, aby we wskaźniku korzenia drzewa znalazła się wartość `NULL` po zakończeniu działania tej funkcji. Stąd w wierszu nr 8 funkcja przypisuje tę wartość do zdereferencjonowanego wskaźnika `node`. Taki zapis powoduje także, że ta wartość jest przypisywana wszystkim polom wskaźnikowym węzła, przed jego usunięciem, a ostatecznie zostanie przypisana także wskaźnikowi `root`.

Usuwanie węzła z BST

Celem operacji usunięcia węzła z BST jest tak naprawdę pozbycie się z tej struktury danych określonego klucza. Wbrew pozorom jest to skomplikowana czynność. Funkcja, która ją implementuje musi uwzględniać cztery przypadki:

1. nie ma węzła o zadanym kluczu — ta sytuacja zazwyczaj nie wymaga dodatkowych działań,
2. usuwany węzeł nie ma potomków — węzeł można usunąć, ale należy pamiętać, aby odpowiedni wskaźnik w jego przodku ustawić na wartość `NULL`; jeśli jest on lewym potomkiem swojego przodka, to należy tę wartość nadać polu `left_child` przodka, w przeciwnym razie, polu `right_child`,
3. usuwany węzeł posiada jednego potomka — należy przed usunięciem tego węzła zapisać adres jego potomka w odpowiednim polu wskaźnikowym przodka usuwanego węzła, według metody opisanej w poprzednim punkcie,
4. usuwany węzeł ma dwóch potomków — jest to najtrudniejsza sytuacja; takiego węzła nie można po prostu usunąć; należy znaleźć inny węzeł w BST, który zostanie usunięty w jego zastępstwie.

Usuwanie węzła z BST

Wspomniany w ostatnim punkcie na poprzednim slajdzie inny węzeł, to węzeł będący następnikiem lub poprzednikiem tego, którego pierwotnie miała dotyczyć operacja usuwania. Następnik, to węzeł posiadający klucz bezpośrednio większy od klucza zawartego w oryginalnym węźle. Poprzednik zawiera zaś klucz bezpośrednio mniejszy od tego, który stanowi kryterium usunięcia. Poprzednikiem danego węzła jest skrajnie prawy węzeł w jego lewym poddrzewie, zaś następnikiem skrajnie lewy węzeł w prawym poddrzewie. Przed usunięciem należy klucz z poprzednika/następnika przenieść do węzła, który oryginalnie miał być usunięty.
W implementacji, która zostanie zaprezentowana usuwany będzie poprzednik węzła posiadającego dwóch potomków. Opis usuwania węzła z drzewa rozpoczniemy od funkcji, której zadaniem będzie wyizolowanie poprzednika węzła z drzewa. Następnie opisana zostanie funkcja implementująca obsługę wszystkich wymienionych wcześniej przypadków.

Funkcja isolate_predecessor()

```
1  struct bst_node *isolate_predecessor(struct bst_node
    ↪  **node)
2  {
3          while((*node)->right_child)
4                  node = &(*node)->right_child;
5          struct bst_node *predecessor = *node;
6          *node = (*node)->left_child;
7          return predecessor;
8  }
```

Funkcja `isolate_predecessor()`

Funkcja ta zwraca adres poprzednika węzła przeznaczonego do usunięcia. Jest ona wywoływana wyłącznie z poziomu funkcji usuwającej węzeł z BST, wtedy i tylko wtedy, gdy powinien być usunięty węzeł posiadający dwóch potomków. Jako argument jej wywołania jest przekazywany adres wskaźnika na lewego potomka (korzeń lewego poddrzewa) węzła do usunięcia. W pętli `while` (wiersze 3–4) funkcja przeszukuje lewe poddrzewo tego węzła kierując się w prawą stronę, aż znajdzie skrajnie prawy węzeł tego poddrzewa. Proszę zwrócić uwagę, na sposób użycia w niej wskaźnika na wskaźnik `node`. W każdej iteracji pętli zapisywany jest do niego adres pola, w którym przechowywany jest adres prawego potomka bieżąco wskazywanego węzła. Pętla zakończy się po znalezieniu węzła, który nie ma prawego potomka. Po znalezieniu poprzednika usuwanego węzła funkcja zapamiętuje jego adres w zmiennej lokalnej o nazwie `predecessor` (wiersz nr 5). Następnie, zapisuje ona adres znajdujący się w jego polu `left_child` w zmiennej wskazywanej przez `node` (wiersz nr 6).

Funkcja `isolate_predecessor()`

Poprzednik na pewno nie ma prawego potomka, ale może mieć lewego. Jeśli ten potomek istnieje, to jego adres zostanie zapisany w polu wskaźnikowym rodzica poprzednika, które do tej pory wskazywało na niego. Dzięki temu lewy potomek poprzednika nie zostanie utracony, być może wraz z całym swoim poddrzewem. Jeśli ten potomek jednak nie istnieje, to w polu wskaźnikowym przodka poprzednika zostanie zapisana wartość `NULL`, to także będzie prawidłowe — po usunięciu poprzednika jego rodzic nie będzie miał tego potomka.
Po odłączeniu poprzednika z BST funkcja `isolate_predecessor()` zwraca jego adres i kończy działanie (wiersz nr 7).

Funkcja delete_node()

```
1  void delete_node(struct bst_node **node, int number)
2  {
3          while(*node && (*node)->key != number)
4                  if ((*node)->key > number)
5                          node = &(*node)->left_child;
6                  else
7                          node = &(*node)->right_child;
8          if (*node) {
9                  struct bst_node *node_to_delete = *node;
10                 if(!node_to_delete->left_child)
11                         *node = (*node)->right_child;
12                 else if(!node_to_delete->right_child)
13                         *node = (*node)->left_child;
14                 else {
15                         node_to_delete =
   ↪  isolate_predecessor(&(*node)->left_child);
16                         (*node)->key = node_to_delete->key;
17                 }
18                 free(node_to_delete);
19         }
20 }
```

Funkcja `delete_node()`

Funkcja `delete_node()` jest odpowiedzialna za usunięcie z BST węzła o określonym kluczu. Nie zwraca ona żadnej wartości, a jako argumenty przyjmuje adres wskaźnika na korzeń BST (przez pierwszy argument) oraz klucz, który ma mieć węzeł do usunięcia (drugi parameter). Proszę zwrócić uwagę, że pętla `while` (wiersze 3–7) jest dokładnie taka sama jak w funkcji `add_node()`. Ma ona też takie samo zadanie — zlokalizować węzeł o zadanym kluczu. Jednakże, po jej zakończeniu, w wierszu nr 8 funkcja `delete_node()`, w przeciwieństwie do `add_node()`, sprawdza czy ten węzeł istnieje. Jeśli nie, to kończy zadanie — w BST nie ma węzła o zadanym kluczu i niczego nie trzeba usuwać. Jeśli jednak taki węzeł istnieje, to funkcja zapamiętuje w zmienne lokalnej `node_to_delete` jego adres (wiersz nr 9), a następnie sprawdza, czy **nie istnieje** jego lewy potomek (wiersz nr 10). Jeśli ten ostatni warunek jest spełniony, to może istnieć jeszcze jego prawy potomek, dlatego `delete_node()` przypisuje do wskaźnika wskazywanego przez `node` adres przechowywany w polu `right_child` węzła do usunięcia (wiersz nr 11).

Funkcja `delete_node()`

W ten sposób odpowiednie pole wskaźnikowe jego rodzica (jeśli ma on rodzica) zacznie wskazywać na tego potomka. Jeśli jednak ten potomek w ogóle nie istniał, to odpowiednie pole wskaźnikowe rodzica węzła do usunięcia uzyska wartość `NULL`, która w tym przypadku też jest prawidłowa — rodzic po usunięciu węzła z poszukiwanym kluczem nie będzie miał już tego potomka.

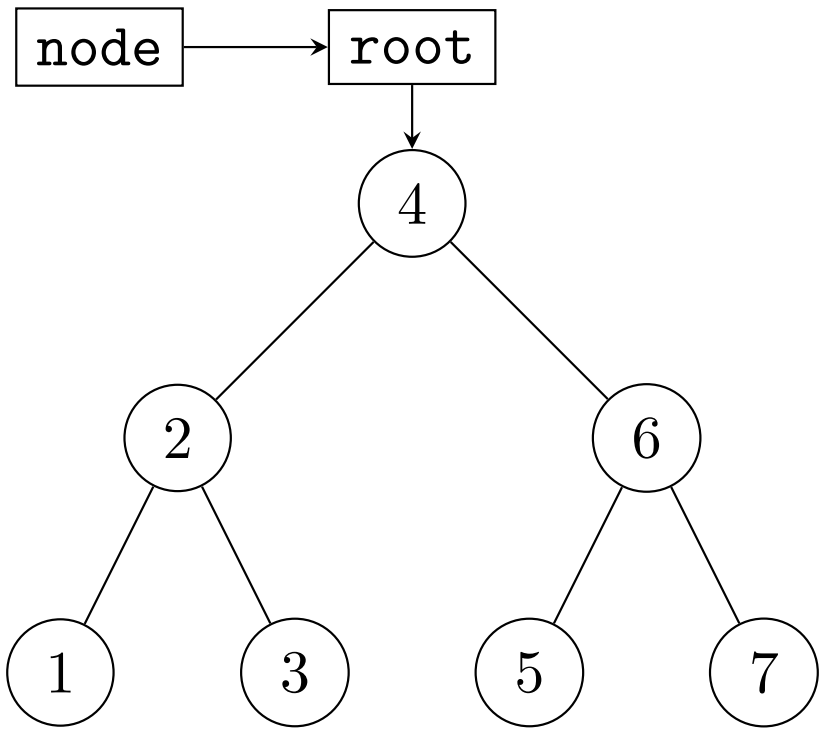
Jeśli warunek w wierszu nr 10 nie był spełniony, to funkcja sprawdza warunek w wierszu nr 12, czyli czy **nie istnieje** prawy potomek węzła do usunięcia. Jeśli ten warunek jest spełniony, to wiadomo, że na pewno istnieje jego lewy potomek (bo nie był spełniony warunek w wierszu nr 10). W związku z tym funkcja `delete_node()` zapisuje we wskaźniku, wskazywanym przez `node`, adres tego potomka (wiersz nr 13).

Jeśli oba warunki z wierszy nr 10 i 12 nie są spełnione, to oznacza, że węzeł do usunięcia ma dwóch potomków. W takim przypadku musi go zastąpić jego poprzednik.

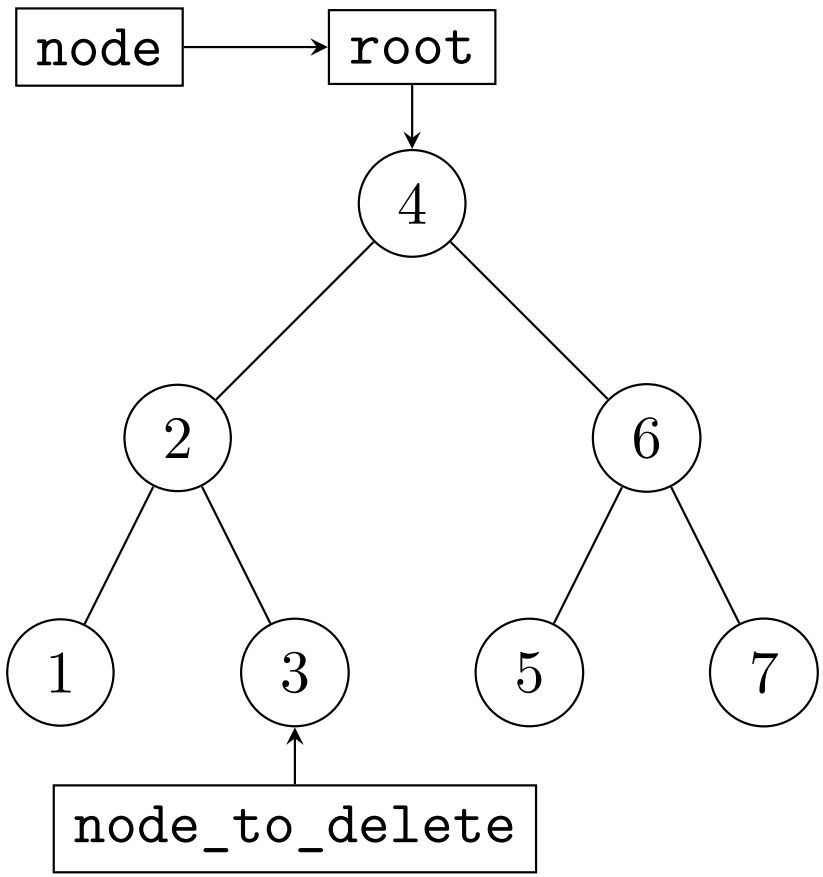
Funkcja `delete_node()`

Dlatego w wierszu nr 15 funkcja `delete_node()` wywołuje funkcję `isolate_predecessor()`, przekazując jej jako argument wywołania adres pola `left_child` węzła, który oryginalnie miał zostać usunięty i zapisując jej wynik we wskaźniku `node_to_delete` (wiersz nr 15). Tym wynikiem jest adres poprzednika węzła do usunięcia. Następnie klucz z poprzednika zapisywany jest w węźle, który oryginalnie miał zostać usunięty (wiersz nr 16).
Niezależnie, który z opisanych przypadków wystąpi, `delete_node()` wywołuje `free()` dla węzła wskazywanego przez `node_to_delete` i kończy działanie.
Następny slajd ilustruje prosty przypadek usuwania klucza, który znajduje się w węźle posiadającym dwóch potomków.

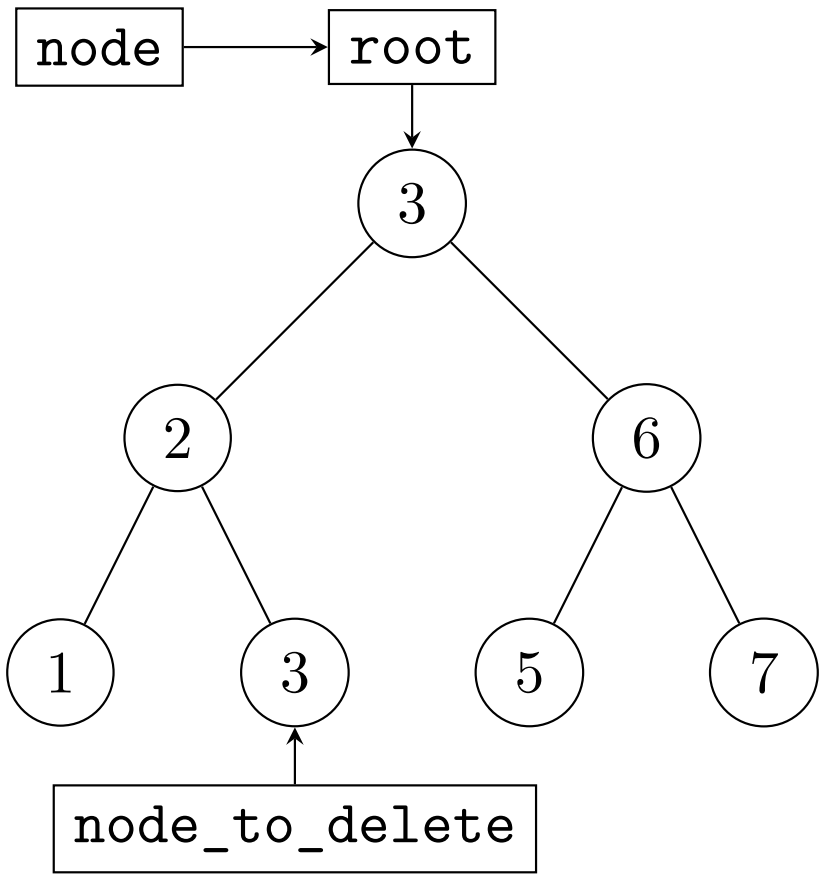
Usuwanie klucza zapisanego w węźle z dwoma potomkami



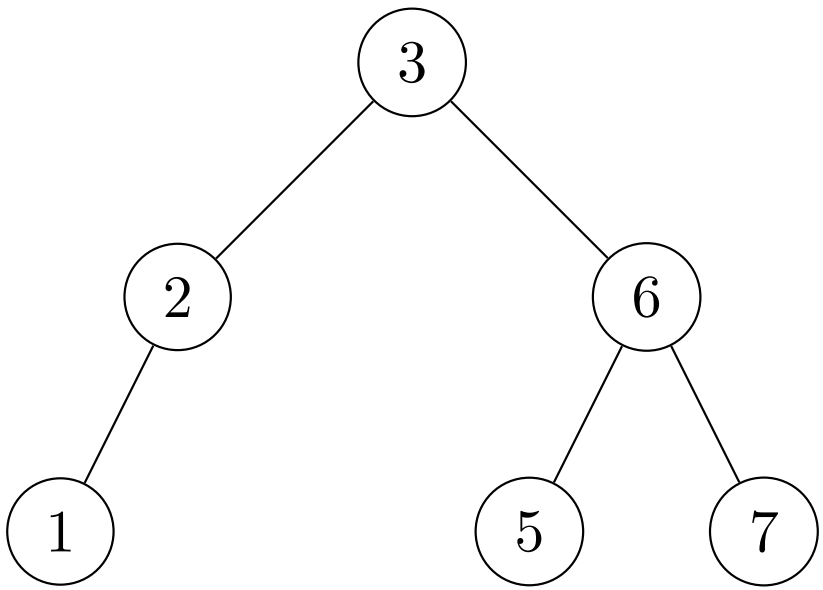
Usuwanie klucza zapisanego w węźle z dwoma potomkami



Usuwanie klucza zapisanego w węźle z dwoma potomkami



Usuwanie klucza zapisanego w węźle z dwoma potomkami



Funkcja main()

Część pierwsza

```
1  int main(void)
2  {
3          srand(time(0));
4          for(int i=0;i<10;i++)
5                  add_node(&root, -10 + rand() % 21);
6          printf("Liczba węzłów w drzewie wyszukiwań binarnych:
   ↪  %u\n", count_nodes(root));
7          print_bst_inorder(root);
8          puts("");
9          print_bst_preorder(root);
10         puts("");
11         print_bst_postorder(root);
12         puts("");
```

Funkcja `main()`

Część pierwsza

Na początku funkcji `main()` inicjowany jest generator liczb pseudolosowych (wiersz nr 3), a następnie w pętli `for` następuje próba dodania do drzewa 10 węzłów, których klucze są liczbami całkowitymi losowanymi z przedziału $[-10, 10]$ (wiersze 4–5). Potem, w wierszu nr 6 wypisywana jest na ekranie liczba węzłów w BST, uzyskana dzięki funkcji `count_nodes()`. W kolejnych wierszach uruchamiane są funkcje, które wypisują klucze przechowywane w BST w porządku, odpowiednio, *inorder*, *preorder* i *postorder*. Proszę zwrócić uwagę, że po każdym z tych wywołań uruchamiana jest funkcja `puts()` celem przeniesienia kursora do następnego wiersza na ekranie.

Funkcja main()

Część druga

```
1         if(root) {
2                 printf("Minimum: %d\n",
    ↪  find_minimum(root)->key);
3                 printf("Maximum: %d\n",
    ↪  find_maximum(root)->key);
4         }
5         int number_to_delete = -10+rand()%21;
6         printf("Liczba do usunięcia: %d\n",
    ↪  number_to_delete);
7         struct bst_node *result = locate(root,
    ↪  number_to_delete);
8         if(result)
9                 printf("Liczba znajduje się w drzewie
    ↪  wyszukiwań binarnych: %d\n", result->key);
10        else
11                puts("Liczby nie ma w drzewie wyszukiwań
    ↪  binarnych.");
```

Funkcja `main()`

Część druga

W funkcji `main()` uruchamiane są także funkcje wyszukujące węzły o największym i najmniejszym kluczu (wiersze 1–4). Ponieważ zwracają one wartość `NULL` tylko wtedy, gdy zostaną wywołane dla psutego drzewa, to w wierszu nr 1 `main()` sprawdza, czy to drzewo nie jest puste. Jeśli warunek jest spełniony, to wskaźniki zwracane przez te funkcje mogą zostać od razu zdereferencjonowane i dzięki temu bezpośrednio można odczytać klucze przechowywane w węzłach przez nie wskazywanych (wiersze nr 2 i 3). Potem funkcja `main()` losuje klucz do usunięcia, który zapisuje w zmiennej lokalnej `number_to_delete` (wiersz nr 6). Ta liczba jest następnie wypisywane na ekranie i uruchamiana jest funkcja `locate()`, która stara się znaleźć węzeł zawierający taki klucz (wiersz nr 7). Jej wynik zapisywany jest w lokalnym wskaźniku `result`. Program sprawdza, czy nie jest on pusty (wiersz nr 8) i jeśli tak jest, to na potwierdzenie wypisuje klucz z węzła przez niego wskazywanego, a jeśli nie, to stosowany komunikat.

Funkcja main()

Część trzecia

```
1           delete_node(&root, number_to_delete);
2           printf("Drzewo po próbie usunięcia liczby
    ↪   %d.\n",number_to_delete);
3           printf("Liczba węzłów w drzewie wyszukiwań binarnych:
    ↪   %u\n", count_nodes(root));
4           puts("Liczby w drzewie.");
5           print_bst_inorder(root);
6           puts("");
7           remove_bst_nodes(&root);
8           return 0;
9   }
```

Funkcja `main()`

Część trzecia

Kolejną operacją na drzewie BST wykonywaną w funkcji `main()` jest usunięcie węzła o zadanym kluczu (wiersz nr 1). Potem program wypisuje komunikaty, o tym jaki klucz próbował usunąć (wiersz nr 2) i ile węzłów jest obecnie w drzewie (wiersz nr 3). Następnie wypisuje wszystkie klucze z BST w porządku *inorder* (wiersz nr 5). Dzięki temu porządkowi najłatwiej jest zauważyć, który klucz został z drzewa usunięty. Na koniec funkcja `main()` wywołuje `remove_bst_nodes()` celem usunięcia wszystkich węzłów z drzewa (wiersz nr 7) i kończy działanie (wiersz nr 8).

Podsumowanie

Zaletą BST jest to, że operacje z nim związane mają czasową złożoność obliczeniową proporcjonalną do jego wysokości. Jeśli kształt tego drzewa jest zbliżony do drzewa pełnego, to wynosi ona $log_2(n)$, gdzie n jest liczbą węzłów w BST. Niestety, jeśli do takiego drzewa zostaną wstawione klucze, które już są posortowane rosnąco lub malejąco, to stanie się ono listą, a jego wysokość będzie wynosiła n.
Aby uniknąć takich skrajnych przypadków stosuje się *drzewa wyważone*, takie jak AVL lub czerwono-czarne. Większość operacji na BST można łatwo zdefiniować zarówno w postaci rekurencyjnej, jak i iteracyjnej (z użyciem pętli). Wyjątkiem jest przechodzenie przez drzewo i inne z nim powiązane operacje, które najwygodniej zrealizować w postaci rekurencyjnej.

Podsumowanie

Zwróćmy uwagę, że drzewa binarne mogą reprezentować także wyrażenia arytmetyczne i wtedy nazywamy je *drzewami arytmetycznymi*. Jeśli wypiszemy zawartość węzłów takiego drzewa w porządku *inorder*, to otrzymamy wyrażenie w notacji infiksowej. Jeśli zrobimy to samo używając porządku *preorder*, to uzyskamy wyrażenie w Notacji Polskiej, a jeśli z użyciem porządku *postorder*, to wyrażenie w Odwrotnej Notacji Polskiej.

Pytania

?

KONIEC

Dziękuję Państwu za uwagę!