Podstawy Programowania 2

Drzewa BST - część druga

Arkadiusz Chrobot

Zakład Informatyki

12 maja 2019

Plan

1. Wstęp
2. Wyszukiwanie w BST
   - Minimalny i maksymalny klucz
   - Wskazany klucz
3. Usuwanie węzła
4. Zmiany w funkcji `main()`
5. Podsumowanie

Wstęp

W tej części wykładu kontynuujemy definiowanie operacji dla BST. Większość funkcji realizujących te operacje będzie przedstawiona zarówno w postaci rekurencyjnej, jak i iteracyjnej. Wyjątkiem będą funkcje związane z operacją usuwania pojedynczego węzła z BST. One będą przedstawione wyłącznie w postaci iteracyjnej. W końcowej części wykładu będą przedstawione zmiany jakie należy wprowadzić do programu zaprezentowanego na poprzednim wykładzie, aby wywołać przedstawione funkcje.

Wyszukiwanie w BST

Rozpoczniemy przedstawieniem funkcji, które znajdują węzły BST zawierające odpowiednio minimalny, maksymalny i określony klucz. Funkcje te będą zwracały adresy odpowiednich węzłów, a w przypadku kiedy one nie będą istniały zwrócona zostanie przez nie wartość `NULL`.

Wyszukiwanie w BST

Minimalny i maksymalny klucz

Uporządkowanie węzłów w BST ułatwia zlokalizowanie tego, który zawiera minimalny lub maksymalny klucz. Węzeł z minimalnym kluczem jest skrajnie lewym węzłem BST, a węzeł z maksymalnym kluczem jest skrajnie prawym węzłem BST. Takie węzły nie istnieją tylko w jednym przypadku - kiedy drzewo BST jest puste. Nawet drzewo z jednym węzłem zawiera element maksymalny i minimalny. Jest to ten sam element, który równocześnie jest korzeniem drzewa.

Wyszukiwanie w BST

Minimalny klucz

Znalezienie klucza minimalnego polega na podążaniu od korzenia BST w lewą stronę, celem zlokalizowania elementu, którego wskaźnik na lewego potomka (lewe poddrzewo) ma wartość `NULL`. Następny slajd zawiera rekurencyjną wersję funkcji realizującej tę operację. Jeśli ta operacja zostanie zastosowana, nie dla korzenia, ale dla innego istniejącego węzła BST, to znajdzie ona element BST o minimalnym kluczu w poddrzewie, dla którego ten węzeł jest korzeniem.

Wyszukiwanie w BST - minimalny klucz

Wersja rekurencyjna

```
1  struct tree_node *find_minimum(struct tree_node *root)
2  {
3      if(root && root->left_child)
4          return find_minimum(root->left_child);
5      else
6          return root;
7  }
```

Wyszukiwanie w BST - minimalny klucz

Wersja rekurencyjna

Przedstawiona na poprzednim slajdzie funkcja zwraca adres węzła BST zawierającego klucz o minimalnej wartości lub wartość `NULL`, jeśli taki węzeł nie istnieje. Ma ona tylko jeden parametr, przez który przekazywany jest adres korzenia, a w przypadku wywołań rekurencyjnych adresy kolejnych odwiedzanych węzłów. W wierszu nr 3 funkcja sprawdza, czy węzeł wskazywany przez parametr `root` istnieje i czy jego lewy potomek istnieje. Jeśli pierwszy warunek nie jest spełniony, to funkcja została wywołana dla pustego drzewa i zakończy swe działanie zwracając wartość `NULL`. Jeśli drugi warunek nie jest spełniony, to znaczy, że bieżąco wskazywany przez parametr węzeł BST nie jest tym, który jest poszukiwany. W takim przypadku funkcja wywoła się rekurencyjnie dla lewego potomka bieżącego węzła (wiersz nr 4). Te wywołania będą powtarzane do momentu, aż funkcja znajdzie skrajnie lewy węzeł BST. Adres tego elementu będzie ostatecznie zwrócony przez funkcję.

Wyszukiwanie w BST - minimalny klucz

Wersja iteracyjna

```
1  struct tree_node *find_minimum(struct tree_node *root)
2  {
3      while(root && root->left_child)
4          root = root->left_child;
5      return root;
6  }
```

Wyszukiwanie w BST - minimalny klucz

Wersja iteracyjna

Wersja iteracyjna funkcji `find_minimum()` ma taki sam nagłówek, jak jej wersja rekurencyjna. Treść tej funkcji składa się z pętli `while`, w której parametr wskaźnikowy `root` jest używany do odwiedzania kolejnych węzłów BST, począwszy od korzenia do tego, który nie posiada lewego potomka. Jeśli funkcja zostanie wywołana dla pustego drzewa, to instrukcja zawarta w pętli nie wykona się nawet raz. Po zakończeni pętli funkcja kończy swe działanie zwracając adres zawarty w zmiennej `root`.

Wyszukiwanie w BST - maksymalny klucz

Operacja wyszukiwania węzła BST o maksymalnym kluczu ma analogiczny przebieg do operacji wyszukiwania węzła o kluczu minimalnym. Różnica polega na tym, że szukany jest węzeł położony na prawo od korzenia, który nie posiada prawego potomka. Innymi słowy, będzie to skrajnie prawy węzeł BST. Szukanie takiego węzła może zakończyć się niepowodzeniem tylko wtedy, gdy operacja będzie wykonywana na pustym drzewie. Jeśli operacja zostanie przeprowadzona dla innego węzła drzewa, niż korzeń, to znajdzie ona element BST o maksymalnym kluczu dla poddrzewa, w którym ten węzeł jest korzeniem. Na kolejnych dwóch slajdach zamieszczona jest wersja rekurencyjna i iteracyjna funkcji wyszukującej w BST element o maksymalnym kluczu. Ze względu na ich podobieństwo do funkcji szukającej węzła o minimalnym kluczu nie będą one dokładniej opisywane.

Wyszukiwanie w BST - maksymalny klucz

Wersja rekurencyjna

```
1  struct tree_node *find_maximum(struct tree_node *root)
2  {
3          if(root && root->right_child)
4              return find_maximum(root->right_child);
5          else
6              return root;
7  }
```

Wyszukiwanie w BST - maksymalny klucz

Wersja iteracyjna

```
1  struct tree_node *find_maximum(struct tree_node *root)
2  {
3      while(root && root->right_child)
4          root = root->right_child;
5      return root;
6  }
```

Wyszukiwanie w BST - wskazany klucz

Do wyszukiwania węzła o zadanym kluczu w BST możemy użyć podobnej metody, jak przy wyszukiwaniu miejsca do wstawienia nowego węzła w tym drzewie. Porównując wartość klucza z bieżąco odwiedzanym węzłem można ustalić, w którym poddrzewie (prawym lub lewym) powinien znajdować się szukany klucz. Operacja wyszukiwania kończy się, jeśli bieżąco odwiedzany węzeł ma taki sam klucz, jak ten, który jest poszukiwany, lub gdy nie ma węzłów, które należałoby przeszukać. W tym ostatnim przypadku wyszukiwanie kończy się niepowodzeniem.

Wyszukiwanie w BST - wskazany klucz

Wersja rekurencyjna

```
1  struct tree_node *find_node(struct tree_node *root, int key)
2  {
3          if(root && root->key > key)
4              return find_node(root->left_child,key);
5          else if(root && root->key < key)
6              return find_node(root->right_child,key);
7          else
8              return root;
9  }
```

Wyszukiwanie w BST - wskazany klucz

Wersja rekurencyjna

Przedstawiona na poprzednim slajdzie funkcja zwraca wskaźnik na węzeł drzewa i posiada dwa parametry. Przez pierwszy (wskaźnikowy) przekazywany jest do niej, przy wywołaniu nierekurencyjnym, adres korzenia drzewa, a w wywołaniach rekurencyjnych adres prawego lub lewego potomka bieżąco odwiedzanego węzła. Drugi parametr służy do przekazywania poszukiwanego klucza. W trzecim wierszu funkcja sprawdza, czy przekazany jej przez parametr `root` adres ma wartość różną od `NULL` i jeśli tak jest, czy klucz zapisany w węźle wskazywanym przez ten parametr jest większy od poszukiwanego klucza. Jeśli pierwszy warunek nie jest spełniony podczas pierwszego, nierekurencyjnego wywołania tej funkcji, to oznacza to, że została ona wywołana dla pustego BST. Jeśli ten warunek będzie niespełniony w wywołaniu rekurencyjnym, to znaczy to, że nie ma węzła o zadanym kluczu w całym drzewie. Jeśli oba warunki są spełnione, to funkcja jest wywoływana rekurencyjne, a jako pierwszy argument jej wywołania podawany jest wskaźnik na lewego potomka bieżąco odwiedzanego węzła (wiersz nr 4).

Wyszukiwanie w BST - wskazany klucz

Wersja rekurencyjna

Jeśli nie jest spełniony drugi warunek z wiersza nr 3 funkcji, to funkcja sprawdza warunki w wierszu nr 5. Ponownie, najpierw sprawdza ona, czy parametr `root` nie ma wartości `NULL`. Jest to konieczne, gdyż podczas sprawdzania tych warunków funkcja nie ma żadnej informacji, który z warunków w wierszu nr 3 nie był spełniony. Jeśli istnieje węzeł, na który wskazuje wspomniany parametr wskaźnikowy, to funkcja sprawdza, czy klucz w nim zapisany jest mniejszy od poszukiwanego. Jeśli tak, wywołuje się rekurencyjnie dla prawego potomka tego węzła. Wywołania rekurencyjne mogą zakończyć się z dwóch powodów. Pierwszym jest, to że parametr `root` uzyska wartość `NULL`. Oznacza to, że nie ma w BST węzła o poszukiwanym kluczu, w związku z tym funkcja zwróci wartość `NULL` (wiersz nr 8). Drugi powód, to ten, że bieżący węzeł nie ma klucza ani mniejszego, ani większego od poszukiwanego. Pozostaje zatem tylko jedna możliwość - węzeł zawiera poszukiwany klucz i funkcja zwróci adres tego węzła.

Wyszukiwanie w BST - wskazany klucz

Wersja iteracyjna

```
1  struct tree_node *find_node(struct tree_node *root, int key)
2  {
3      while(root) {
4          if(root->key == key)
5              return root;
6          if(root->key > key)
7              root = root->left_child;
8          else
9              root = root->right_child;
10     }
11     return root;
12 }
```

Wyszukiwanie w BST - wskazany klucz

Wersja iteracyjna

W wersji iteracyjnej funkcji `find_node()` prototyp został zachowany ten sam, co w wersji rekurencyjnej. Przeszukiwanie drzewa odbywa się wewnątrz pętli `while`. Pętla ta powtarzana jest tak długo, jak długo parametr `root` ma wartość różną od `NULL`. Zmiana wartości tego parametru następuje wewnątrz pętli. Jeśli klucz zawarty w węźle bieżąco wskazywanym przez `root` jest równy kluczowi poszukiwanemu (wiersz nr 4), to funkcja zwraca adres tego węzła i kończy swoje działanie (wiersz nr 5). Jeśli tak nie jest, ale klucz węzła jest większy od klucza szukanego (wiersz nr 6), to wspomniany wskaźnik jest „przestawiany” na lewego potomka wskazywanego węzła (wiersz nr 7). Jeśli i ten warunek nie jest spełniony, to do wskaźnika zapisywany jest adres prawego potomka węzła, na który wskazuje `root`. Jeśli pętla `while` zakończy się z powodu niespełnienia warunku z wiersza nr 3, to znaczy to, że szukanego klucza nie ma w całym BST i funkcja w wierszu 11 zwraca `NULL` oraz kończy swoje działanie.

Usuwanie węzła

Operacja usuwania pojedynczego węzła z BST usuwa węzeł posiadający określony klucz. Implementując ją należy rozważyć i oprogramować 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

Wspomniany w ostatnim punkcie na poprzednim slajdzie inny węzeł, to węzeł będący następnikiem lub poprzednikiem węzła, którego pierwotnie miała dotyczyć operacja usuwania. Następnik, to węzeł posiadający klucz bezpośrednio większy (lub równy) od klucza zawartego w oryginalnym węźle. Poprzednik zawiera zaś klucz bezpośrednio mniejszy (lub równy) od tego, który stanowi kryterium usunięcia. Poprzednikiem danego węzła jest skrajnie prawy węzeł w jego w lewym poddrzewie, zaś następnikiem skrajnie lewy węzeł w prawym poddrzewie. Przed usunięciem należy dane 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.

Usuwanie węzła

Funkcja isolate_predecessor()

```
1  struct tree_node *isolate_predecessor(struct tree_node **root)
2  {
3          while(*root && (*root)->right_child)
4              root = &(*root)->right_child;
5          struct tree_node *predecessor = *root;
6          if(*root)
7              *root = (*root)->left_child;
8          return predecessor;
9  }
```

Usuwanie węzła

Funkcja `isolate_predecessor()`

Funkcja ta zwraca adres poprzednika węzła BST, którego adres został jej przekazany przez podwójny wskaźnik `root`. Powinna ona być wywoływana wyłącznie z poziomu funkcji usuwającej węzeł z BST, wtedy i tylko wtedy, kiedy musi być usunięty węzeł posiadający dwóch potomków. Jako argument jej wywołania musi zostać przekazany wskaźnik na lewego potomka (korzeń lewego poddrzewa) węzła do usunięcia. W pętli `while` (wiersze 3 i 4) funkcja będzie przeszukiwać lewe poddrzewo tego węzła kierując się w prawą stronę, aż znajdzie skrajnie prawy węzeł tego poddrzewa. Proszę zwrócić uwagę, na sposób użycia w niej podwójnego wskaźnika `root`. 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. Sprawdzanie w wierszu nr 3, czy istnieje węzeł wskazywany przez `root` jest nadmiarowe.

Usuwanie węzła

Funkcja isolate_predecessor()

Po znalezieniu poprzednika usuwanego elementu funkcja zapamiętuje jego adres w zmiennej lokalnej predecessor. Następnie, po sprawdzeniu, czy ten poprzednik istnieje (wiersz nr 6), co też jest czynnością nadmiarową, funkcja zapisuje wartość znajdującą się w jego polu left_child w zmiennej wskazywanej przez root. Czynność ta jest wykonywana z dwóch powodów. Poprzednik na pewno nie ma prawego potomka, ale nie oznacza to, że nie ma także lewego potomka. Jeśli taki potomek istnieje, to jego adres musi być zapisany w polu przodka poprzednika, które na niego wskazywało. Inaczej stracimy dostęp do jego lewego potomka i całego związanego z nim poddrzewa. Jeśli lewy potomek poprzednika nie istnieje, to w jego polu left_child jest zapisana wartość NULL. Taka sama wartość powinna się znaleźć w polu przodka, które wskazywało na poprzednika, po wyłączeniu tego ostatniego z drzewa. Zatem instrukcja z wiersza nr 7 uwzględnia oba przypadki. Po wyłączeniu poprzednika z BST funkcja zwraca jego adres i kończy swe działanie (wiersz nr 8).

Usuwanie węzła

```
1   void delete_node(struct tree_node **root, int key)
2   {
3       while(*root && (*root)->key!=key) {
4           if((*root)->key>key)
5               root = &(*root)->left_child;
6           if((*root)->key<key)
7               root = &(*root)->right_child;
8       }
9       if(*root) {
10          struct tree_node *node = *root;
11          if(!node->left_child)
12              *root = (*root)->right_child;
13          else if(!node->right_child)
14              *root = (*root)->left_child;
15          else {
16              node = isolate_predecessor(&(*root)->left_child);
17              (*root)->key = node->key;
18              (*root)->value = node->value;
19          }
20          free(node);
21      }
22  }
```

Usuwanie węzła

Funkcja `delete_node()` nie zwraca żadnej wartości, ale posiada parametr będący podwójnym wskaźnikiem. Jest to konieczne ze względu na to, że w tej funkcji będą dokonywane zmiany zawartości wskaźnika korzenia lub pól wskaźnikowych poszczególnych węzłów. Przez drugi parametr tej funkcji przekazywany jest klucz, który powinien zawierać usuwany węzeł. W pętli `while` (wiersze 3 - 8) drzewo jest przeszukiwane w celu zlokalizowania tego węzła. Nie możemy w tym miejscu użyć funkcji `find_node()`, bo oprócz adresu węzła potrzebny jest nam również adres zmiennej/pola, które zawiera ten adres. Gwarantuje to nam podwójny wskaźnik. Pętla `while` sprawdza również (wiersz nr 3), czy zmienna `*root` nie zawiera wartości `NULL`. Może ona zakończyć się znalezieniem szukanego węzła lub wtedy, gdy nie będzie już węzłów BST, które można by przeszukać. Ostatni przypadek zachodzi również wtedy, gdy funkcja zostanie wywołana dla pustego drzewa. Wykonanie dalszych czynności uzależnione jest od tego, czy wskaźnik `*root` wskazuje na istniejący węzeł drzewa, co sprawdzane jest w wierszu nr 9 funkcji.

Usuwanie węzła

Jeśli wskaźnik `*root` zawiera wartość równą `NULL`, to znaczy to, że nie istnieje węzeł, który należałoby usunąć. W przeciwnym przypadku, funkcja najpierw zapamiętuje adres węzła wskazywanego przez `*root` w lokalnym wskaźniku `node` (wiersz nr 10), a następnie sprawdza, czy *nie* istnieje lewy potomek węzła. Jeśli tak, to oznacza to, że do usunięcia jest węzeł, z co najwyżej jednym potomkiem - może istnieć jeszcze prawy potomek. Adres tego potomka jest zapisywany w zmiennej wskazywanej przez `root` (może to być wskaźnik na korzeń drzewa lub odpowiednie pole wskaźnikowe potomka usuwanego węzła). Dzięki temu, ten potomek z całym swoim poddrzewem nie zostanie odłączony od BST. Jeśli prawy potomek nie istniał, to w polu `right_child` węzła usuwanego znajduje się wartość `NULL`, która teraz powinna znaleźć się w zmiennej wskazywanej przez `root`, a więc i w takim przypadku instrukcja z wiersza nr 12 jest poprawna.

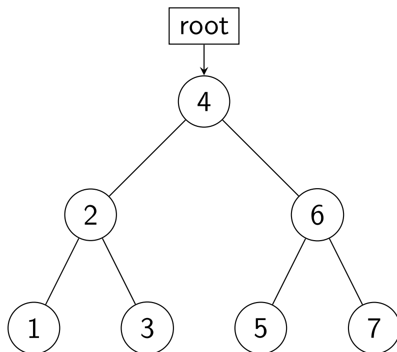
Usuwanie węzła

Jeśli istnieje lewy potomek, to funkcja sprawdza, czy *nie* istnieje prawy potomek usuwanego węzła. Jeśli jest to prawdą, to podstępuje ona analogicznie do przypadku, kiedy nie istnieje lewy potomek, z tym, że tym razem ma pewność, że w polu `left_child` jest wartość różna od `NULL` i należy ją zapamiętać w zmiennej wskazywanej przez `root` (wiersz nr 14). Inaczej od BST zostałby odłączony lewy potomek usuwanego węzła z całym swoim poddrzewem. Jeśli obaj potomkowie usuwanego węzła istnieją (wiersz nr 16), to funkcja wywołuje funkcję `isolate_predecessor()`, która odnajduje poprzednika węzła, wyłącza go z drzewa i zwraca jego adres, który zapamiętywany jest w zmiennej `node`. Następnie funkcja `delete_node()` przepisuje dane z poprzednika do węzła, który miał oryginalnie zostać usunięty (wiersze 17 i 18). Przed zakończeniem funkcja zwalania pamięć przeznaczoną na węzeł wskazywany przez `node`. Ponieważ w obu wcześniej opisywanych przypadkach w tej zmiennej zapisywany był adres węzła do usunięcia (wiersz nr 10), to dla wszystkich trzech przypadków zwalniany jest prawidłowy węzeł.

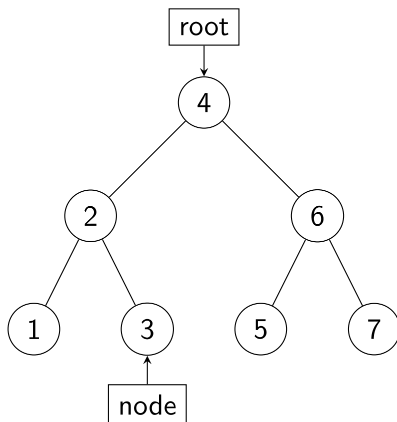
Usuwanie węzła

Kolejne slajdy ilustrują operację usuwania z BST węzła posiadającego dwóch potomków (w tym przypadku jest to także korzeń drzewa).

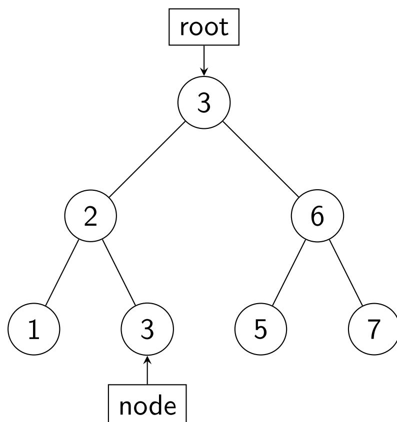
Usuwanie węzła



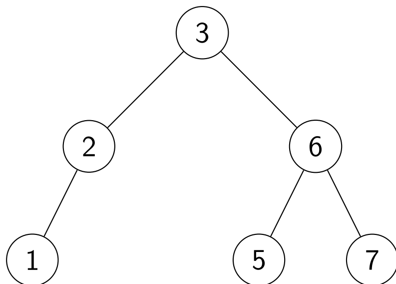
Usuwanie węzła



Usuwanie węzła



Usuwanie węzła



Zmiany w funkcji main()

Aby przetestować działanie zdefiniowanych wcześniej funkcji należy je wywołać w funkcji `main()` między wywołaniem funkcji `add_node()`, a `remove_tree_nodes()`. Kolejne slajdy zawierają przykładowy kod wywołujący te funkcje.

Zmiany w funkcji main()

```
1  if(root) {
2      printw("Najmniejsza wartość w drzewie: %c\n",
3                                  find_minimum(root)->value);
4      printw("Największa wartość w drzewie: %c\n",
5                                  find_maximum(root)->value);
6      refresh();
7      getch();
8  }
```

Zmiany w funkcji main()

Aby zademonstrować działanie funkcji find_minimum() i find_maximum() należałoby zapamiętać wynik ich wywołania dla korzenia drzewa w różnych zmiennych wskaźnikowych, zbadać czy te zmienne zawierają adres różny od NULL i wypisać wartość zawartą w węzłach BST wskazywanych przez te zmienne. Można też postąpić tak, jak w przypadku kodu zamieszczonego na poprzednim slajdzie - upewnić się (wiersz nr 1), że drzewo, dla którego te funkcje są wywoływane, nie jest puste. Jeśli tak jest, to te funkcje na pewno zwrócą adresy istniejących węzłów. Proszę zwrócić też uwagę na sposób ich wywołania (wiersz nr 3 i wiersz nr 5). Ponieważ zwracają one adres węzła, to od razu możemy przy pomocy tego adresu sięgnąć do jego pola value.

Zmiany w funkcji main()

```
1   int key;
2   scanw("%d",&key);
3   struct tree_node *result = find_node(root,key);
4   if(result) {
5       printw("Szukana wartość dla klucza %d to %c\n",
6                                               result->key, result->value);
7       refresh();
8       getch();
9       erase();
10      delete_node(&root,key);
11      print_keys(root,COLS/2,1,20);
12      refresh();
13      getch();
14  }
```

Zmiany w funkcji `main()`

W zamieszczonym na poprzednim slajdzie kodzie użytkownik programu proszony jest o podanie klucza, który ma posiadać wyszukiwany węzeł (wiersz nr 2). Jeśli zostanie on odnaleziony przez funkcję `find_node()`, to jego klucz i wartość są wypisywane na ekranie (wiersze 5 i 6), a następnie jest on usuwany (wiersz nr 10) i wypisywane są klucze zawarte w BST z uwzględnieniem struktury tego drzewa (wiersz nr 11).
W kodzie programu, który został udostępniony na stronie przedmiotu, używane jest makro - znacznik preprocesora o nazwie `RECURSIVE`. Jeśli zostanie ono zdefiniowane na początku programu lub w opcjach jego kompilacji, to zostaną użyte funkcje rekurencyjne, implementujące niektóre z operacji przedstawionych na wykładach dotyczących BST. Jeśli nie, to zostaną użyte ich iteracyjne odpowiedniczki.

Podsumowanie

Najbardziej czasochłonną operacją na BST, spośród tych które zostały zaprezentowane, jest operacja wyszukiwania węzła w drzewie. Dotyczy to także wyszukiwania poprzednika/następnika oraz elementu o maksymalnym/minimalnym kluczu. Czas jej trwania jest proporcjonalny do wysokości tego drzewa. W przypadku drzewa pełnego tę wartość można obliczyć ze wzoru $log_2(n)$, gdzie n jest liczbą węzłów w drzewie. Większość drzew, w których klucze mają rozkład losowy ma kształt zbliżony do drzewa pełnego, a więc BST jest bardzo efektywną strukturą danych. Na poprzednim wykładzie wspominaliśmy o możliwości zaimplementowania drzewa z użyciem tablicy. Przykładem takiej implementacji BST jest ... posortowana tablica. Podobnie jak w tym drzewie, klucze w niej (i być może wartości) są uporządkowane niemalejąco od strony lewej do prawej. Jeśli zastosujemy do poszukiwania kluczy w tej tablicy algorytm wyszukiwania binarnego, to jego czas działania będzie taki sam, jak wyszukiwania w BST. Można przy użyciu tablicy zaimplementować też inne drzewa. Przykład takiej implementacji zostanie podany na następnym wykładzie.

Podsumowanie

Nie każde drzewo BST posiada kształt drzewa pełnego lub nawet do niego zbliżony. Skrajnymi przypadkami są drzewa, które są tworzone na podstawie zbioru kluczy, który jest od początku uporządkowany niemalejąco lub nierosnąco. Wówczas te drzewa mają taki sam kształt jak zwykła lista liniowa, a czas wyszukiwania węzła w takim drzewie jest wprost proporcjonalny do liczby jego węzłów. Możemy mówić w takich przypadkach o drzewach zdegenerowanych. Aby uniknąć powstawania drzew, o takich kształtach stosuje się operację *wyważania*. Powstałe w ten sposób drzewa BST nazywamy *drzewami wyważonymi*. Przykładami takich struktur są drzewa AVL i drzewa czerwono-czarne, które nie będą omawiane na tym wykładzie.

Pytania

?

KONIEC

Dziękuję Państwu za uwagę!