Podstawy Programowania 2

Drzewa BST - część pierwsza

Arkadiusz Chrobot

Katedra Systemów Informatycznych

20 kwietnia 2020

Plan

1. Wstęp
2. Definicje
3. Implementacja
   - Typ bazowy i wskaźnik na korzeń
   - Dodawanie elementu
   - Przechodzenie drzewa binarnego
   - Wypisanie zgodnie ze strukturą drzewa
   - Usuwanie wszystkich węzłów drzewa binarnego
4. Podsumowanie

Wstęp

Drzewa są nieliniowymi abstrakcyjnymi strukturami danych używanymi do reprezentowania danych uporządkowanych hierarchicznie. Są one szczególnym przypadkiem innych struktur danych nazywanych grafami, które zostaną zaprezentowane na kolejnych wykładach. Drzewo wyszukiwań binarnych (ang. *Binary Search Tree*) z kolei możemy uznać za rodzaj drzewa, w którym obowiązuje określony porządek elementów. Drzewa i grafy są realizacją pojęć wywodzących się z działu matematyki nazwanego topologią, dlatego najpierw poznamy kilka definicji z tej dziedziny odnoszących się do drzew.

Definicje

Drzewo

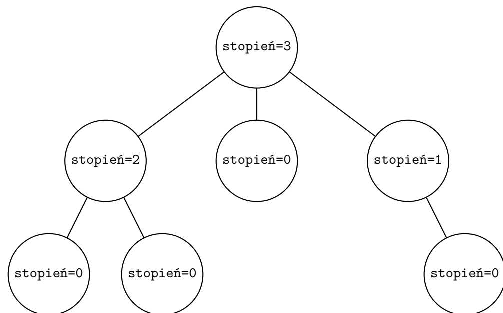
Drzewo (ang. *tree*) jest zbiorem T jednego lub więcej elementów nazywanych węzłami (ang. *node*), takich, że:

- istnieje jeden wyróżniony węzeł nazywany *korzeniem* (ang. *root*) drzewa, oraz
- pozostałe węzły (z wyłączeniem korzenia) są podzielone na $m \geqslant 0$ rozłącznych zbiorów $T_1, \ldots, T_m$, z których każdy jest drzewem. Drzewa $T_1, \ldots, T_m$ są nazywane *poddrzewami* korzenia.

Definicje

Stopień węzła

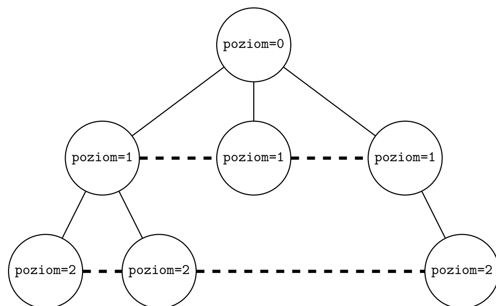
Stopniem węzła drzewa nazywamy liczbę jego poddrzew. Węzły o stopniu większym od zera nazywane są *węzłami wewnętrznymi*, a węzły o stopniu zerowym *węzłami zewnętrznymi* lub *liśćmi*.



Definicje

Poziom węzła

Poziom węzła jest wielkością zdefiniowaną rekurencyjnie: korzeń ma poziom równy 0, a poziom każdego innego węzła jest o jeden większy od poziomu korzenia w najmniejszym zawierającym go poddrzewie.



Definicje

Drzewa uporządkowane (płaskie)

Drzewo, w którym kolejność występowania poddrzew jest istotna jest *drzewem uporządkowanym* lub *drzewem płaskim*.

Definicje

Wysokość drzewa

Wysokość drzewa jest o jeden większa od maksymalnego poziomu jego węzłów.

Definicje

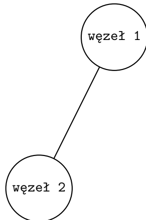
Drzewo binarne

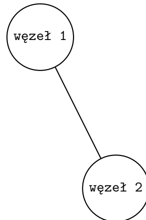
Drzewo binarne jest skończonym zbiorem węzłów, który albo jest pusty, albo składa się z korzenia i dwóch rozłącznych drzew binarnych nazywanych *lewym* i *prawym* poddrzewem. Zatem stopień każdego węzła w drzewie binarnym nie przekracza wartości dwa. Jeśli lewe i/lub prawe poddrzewo danego węzła nie jest puste, to jego korzeń nazywamy *potomkiem* danego węzła, a ten węzeł nazywamy *przodkiem* tego korzenia. Ciekawostka: według podanych definicji drzewo binarne nie jest drzewem, bo może nie zawierać żadnych elementów, a drzewo musi zawierać co najmniej jeden element.

Definicje

Różnice między drzewami i drzewami binarnymi

Jeśli założymy, że rysunki na slajdzie przedstawiają drzewa binarne, to będą to dwa różne drzewa binarne. Oba rysunki będą przedstawiały jedno drzewo, jeżeli przyjmiemy, że jest to zwykłe drzewo.

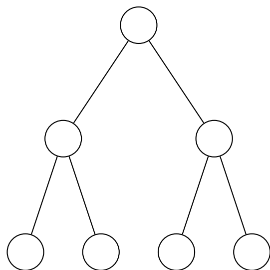




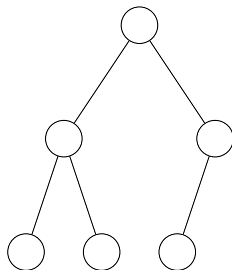
Definicje

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, nazywane od angielskiej nazwy drzewem BST jest drzewem binarnym, które służy do realizacji struktur słownikowych (ang. *dictionary*), w których każda wartość jest identyfikowana za pomocą klucza. W drzewie BST klucze uporządkowane są według następującej reguły:

Uporządkowanie kluczy w BST

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

Peter Braß[1] nazywa drzewa BST *pierwszym modelem drzew wyszukiwań*.

[1] Peter Brass "Advanced Data Structures", Cambridge University Press, Cambridge, 2008

Implementacja

Użyjemy struktury drzewa BST do przechowywania par klucz - wartość, gdzie kluczem będzie liczbowa wartość kodu ASCII znaku, zaś wartością sam znak. Jeśli przyjrzymy się definicji tego drzewa to dojdziemy do wniosku, że pozwala ona na to, aby klucz danego węzła powtarzał się zarówno po jego lewej, jak i prawej stronie. To niestety utrudnia implementację operacji wstawiania elementu (węzła) i tworzenia drzewa. Należy więc przyjąć jakieś rozwiązanie kompromisowe tej kwestii. Jedni informatycy uważają, że klucze w drzewie BST nie powinny się powtarzać, inni twierdzą, że mogą się powtarzać, ale należy przyjąć, że klucze powtarzające się będą wyłącznie po lewej lub wyłącznie po prawej stronie węzła, który zawiera taki sam klucz. Uniemożliwienie powtarzania kluczy wydaje się być dobrym pomysłem, dopóki nie będziemy chcieć przechowywać np. danych osobowych, w których klucz stanowiłoby nazwisko osoby. Wspomniany Peter Braß proponuje, aby klucze się powtarzały, ale wartości były zawarte wyłącznie w liściach drzewa. My przyjmiemy jeszcze inne rozwiązanie - klucze będą się powtarzać i każdy węzeł będzie przechowywał wartość.

Implementacja

Tak, jak w przypadku innych abstrakcyjnych struktur danych, BST możemy zaimplementować z użyciem tablicy lub w formie dynamicznej struktury danych. Nas będzie na tym wykładzie interesowało to drugie podejście. Podobnie jak w przypadku list i ich pochodnych, musimy zdefiniować typ bazowy BST (typ pojedynczego elementu) i operacje, które będą na tej strukturze wykonywane. W tym ostatnim przypadku ograniczymy się na tym wykładzie tylko do operacji wstawiania węzłów do drzewa, operacji wypisania zawartości drzewa zgodnie z algorytmami przechodzenia drzewa, operacji wypisania zawartości drzewa w sposób ilustrujący jego strukturę oraz operacji usunięcia wszystkich węzłów z drzewa. Zgodnie z przedstawioną definicją, puste drzewo binarne jest uznawane za istniejące drzewo, a więc nie istnieje operacja jego tworzenia, ani operacja jego usuwania. Operacja usunięcia pojedynczego elementu z drzewa jest dosyć skomplikowana, dlatego zostanie omówiona na następnym wykładzie, wraz z kilkoma innymi, dodatkowymi operacjami.

Implementacja

Pliki nagłówkowe

```
1 #include<stdlib.h>
2 #include<time.h>
3 #include<curses.h>
4 #include<locale.h>
```

Implementacja

Pliki nagłówkowe

Wśród instrukcji włączających pliki nagłówkowe, oprócz tych, które były używane w programach z poprzednich wykładów widzimy także te, które włączają pliki `curses.h` i `locale.h`. Funkcje w nich zadeklarowane posłużą do wypisywania zawartości drzewa na ekranie. Przypomnijmy, że jednym z naszych zadań będzie zdefiniowanie funkcji, która wypisze zawartość drzewa ilustrując jego strukturę. Do tego nadają się funkcje biblioteki `curses`. Z kolei funkcja `time()` zdefiniowana w pliku nagłówkowym `time.h` posłuży do inicjacji generatora liczb pseudolosowych, dzięki któremu będziemy wstawiać wartości do BST.

Typ bazowy i wskaźnik na korzeń

```
1  struct tree_node
2  {
3      int key;
4      char value;
5      struct tree_node *left_child, *right_child;
6  } *root;
```

Typ bazowy

Definicja typu bazowego przypomina tę, którą znamy np. z listy dwukierunkowej, jednak role wskaźników są w niej inne. Pole wskaźnikowe `left_child` będzie wskazywało na lewego potomka (i w rezultacie na lewe poddrzewo) danego węzła, a pole `right_child` na prawego potomka (i tym samym na prawe poddrzewo) danego węzła. Jeśli węzeł będzie liściem, to oba te wskaźnik będą miały wartość `NULL`. Jeśli będzie istniał tylko jeden potomek danego węzła, to jedno z tych pól będzie miało taką wartość. Istnieją także implementacje BST, w których każdy węzeł ma dodatkowe, trzecie pole wskaźnikowe. Wskazuje ono na przodka takiego węzła. Jedyny węzeł, który ma wartość `NULL` w takim polu, to korzeń drzewa. Pola `key` i `value` będą służyły, odpowiednio, do przechowywania klucza i związanej z nim wartości. Wraz z definicją typu zadeklarowana jest zmienna globalna o nazwie `root`, która będzie wskaźnikiem na korzeń drzewa.

Operacje na drzewie BST

Niektóre z operacji na drzewie BST mogą być prosto zrealizowane zarówno w postaci podprogramów rekurencyjnych, jak i iteracyjnych, inne implementuje się zazwyczaj wyłącznie jako funkcje rekurencyjne. Ich iteracyjna postać byłaby bardzo zawiła i niewiele wydajniejsza od rekurencyjnej. Na tym wykładzie, tam gdzie będzie to łatwe do osiągnięcia, przedstawię obie wersje danych operacji. Zaczniemy od operacji wstawiania nowego węzła do drzewa.

Dodawanie elementu - wersja rekurencyjna

```c
void add_node(struct tree_node **root, int key, char value)
{
    if(*root==NULL)
    {
        *root = (struct tree_node *)
                        malloc(sizeof(struct tree_node));
        if(*root) {
            (*root)->key = key;
            (*root)->value = value;
            (*root)->left_child = (*root)->right_child = NULL;
        }
    } else
    if((*root)->key >= key)
        add_node(&(*root)->left_child,key,value);
    else
        add_node(&(*root)->right_child,key,value);
}
```

Dodawanie elementu - wersja rekurencyjna

Funkcja `add_node()` nie zwraca żadnej wartości, ale posiada trzy parametry. Przez pierwszy, który jest podwójnym wskaźnikiem przekazywany będzie adres wskaźnika korzenia drzewa lub w przypadku wywołań rekurencyjnych potomka odwiedzanego węzła. Przez drugi parametr przekazywany jest klucz, a przez trzeci związana z nią wartość. Ta para zostanie zapisana w nowym węźle drzewa. Funkcja po wywołaniu sprawdza, czy przekazany jej wskaźnik ma wartość `NULL`. Jeśli to było pierwsze wywołanie funkcji i ten warunek jest spełniony, to znaczy to, że drzewo było puste i jest wstawiany jego pierwszy węzeł. Zatem w wierszach 5 i 6 przydzielana jest pamięć na nowy węzeł, a w wierszach 8, 9 i 10 inicjowane są jego pola. Zwróćmy uwagę, że oba pola wskaźnikowe zyskują wartość `NULL`. Jeśli warunek z wiersza trzeciego nie był spełniony to znaczy to, że drzewo już istniało. W takim wypadku funkcja wykonuje instrukcję z wiersza 13, tj. ustala relację między kluczem zawartym w istniejącym węźle, a kluczem, który zostanie zapisany w nowym węźle.

Dodawanie elementu - wersja rekurencyjna

Jeśli klucz w istniejącym elemencie jest większy lub równy nowemu kluczowi, to funkcja `add_node()` jest rekurencyjnie wywoływana dla lewego potomka tego tego węzła, który jest jednocześnie korzeniem jego lewego poddrzewa. Wszystkie wartości z kluczami równymi kluczowi w istniejącym węźle trafią zatem do jego lewego poddrzewa. Jeśli warunek z wiersza trzynastego nie jest spełniony, to funkcja jest rekurencyjnie wywoływana dla prawego potomka badanego węzła, który jest równocześnie korzeniem jego prawego poddrzewa. Proszę zwrócić uwagę, że w obu przypadkach wywołań rekurencyjnych podstawiany jest adres odpowiedniego pola wskaźnikowego badanego elementu, a wiec jeśli zajdzie taka potrzeba kolejne wywołanie może zmodyfikować zawartość tego pola. Wywołania rekurencyjne kończą się, kiedy jedno z nich zostanie uruchomione dla pola wskaźnikowego, o wartości równej `NULL`. Wówczas to wywołanie stworzy nowy węzeł drzewa, wykonując instrukcje w wierszach od 4 do 11, tak jak to było opisane na poprzednim slajdzie i zakończy swe działanie.

Dodawanie elementu - wersja rekurencyjna

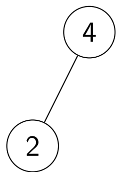
Po utworzeniu nowego węzła nastąpią powroty z poprzednich wywołań rekurencyjnych i ostatecznie zakończy się działanie funkcji `add_node()`. Kolejne slajdy ilustrują jak będzie budowana struktura BST przez tę funkcję, jeśli będą zapisywane w nim klucze w następującej kolejności: `4, 2, 1, 3, 5`.

Dodawanie elementu

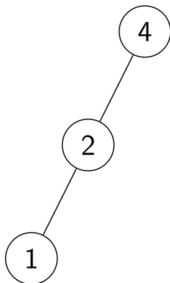
Dodawanie elementu

4

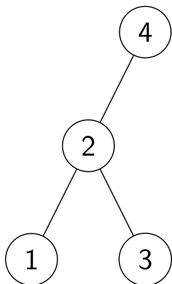
Dodawanie elementu



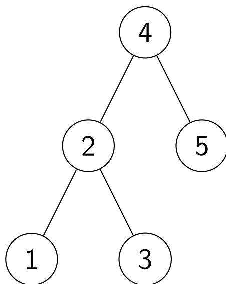
Dodawanie elementu



Dodawanie elementu



Dodawanie elementu



Dodawanie elementu - wersja iteracyjna

```
1  void add_node(struct tree_node **root, int key, int value)
2  {
3      while(*root!=NULL) {
4          if((*root)->key>=key)
5              root = &(*root)->left_child;
6          else
7              root = &(*root)->right_child;
8      }
9      *root = (struct tree_node *)
10                             malloc(sizeof(struct tree_node));
11     if(*root) {
12         (*root)->key = key;
13         (*root)->value = value;
14         (*root)->left_child = (*root)->right_child = NULL;
15     }
16 }
```

Dodawanie elementu - wersja iteracyjna

Okazuje się, że wersja iteracyjna funkcji `add_node()` jest równie prosta do napisania jak jej rekurencyjna odpowiedniczka. Prototyp (nagłówek) funkcji zostaje ten sam. Pierwszym elementem treści tej funkcji jest pętla `while` (wiersze od 3 do 8). Jeśli drzewo byłoby puste, to ta pętla nie wykonałaby się ani razu. Jeśli jednak zawiera ono węzły, to wewnątrz pętli badana jest relacja między kluczem danego węzła, a nowym kluczem. Jeśli klucz w węźle jest większy lub równy kluczowi nowemu, to podwójnemu wskaźnikowi `root` przypisywany jest adres pola-wskaźnika lewego potomka węzła (wiersz nr 5), a w przeciwnym przypadku adres pola-wskaźnika prawego potomka (wiersz nr 7). W rozważanej sytuacji pętla kończy się, gdy któreś z tych pól ma wartość `NULL`. Wówczas tworzony jest nowy węzeł (wiersze 9 i 10), którego adres zapisywany jest do pola wskazywanego przez wskaźnik `root`. W przypadku, gdy pętla `while` zakończyłaby się bez wykonywania ani jednej iteracji, adres nowego węzła byłby zapisany we wskaźniku korzenia drzewa - drzewo tym samym zyskałoby pierwszy węzeł. W wierszach od 12 do 14 inicjowane są pola nowego węzła i kończy się działanie funkcji `add_node()`.

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 zakłada się, że najpierw jest odwiedzane lewe poddrzewo korzenia, a następnie prawe.

Przechodzenie drzewa binarnego

Porządek *inorder*

Algorytm przechodzenia w porządku inorder jest następujący:

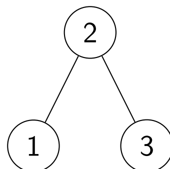
1. odwiedź lewe poddrzewo,
2. odwiedź korzeń,
3. odwiedź prawe poddrzewo.

Ten algorytm najprościej jest zaimplementować w postaci funkcji rekurencyjnej. Kolejny slajd zawiera funkcję, która używając tego algorytmu wypisuje wartości kluczy w drzewie BST. Oprócz tego, na tym slajdzie znajduje się przykład takiego drzewa i wynik jego przejścia w tym porządku.

Przechodzenie drzewa

Porządek *inorder*

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



Wynik

1, 2, 3

Przechodzenie drzewa binarnego

Porządek *preorder*

Algorytm przechodzenia w porządku preorder jest następujący:

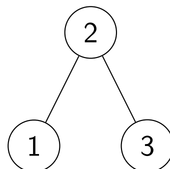
1. odwiedź korzeń,
2. odwiedź lewe poddrzewo,
3. odwiedź prawe poddrzewo.

Również ten algorytm jest najczęściej implementowany w postaci funkcji rekurencyjnej. Różnica między nim, a inorder polega na tym, że korzeń jest odwiedzany w nim na początku, a dopiero potem poddrzewa. Kolejny slajd zawiera funkcję go implementującą i ilustrację jego działania dla przykładowego drzewa - tak jak w przypadku inorder.

Przechodzenie drzewa

Porządek *preorder*

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



Wynik

2, 1, 3

Przechodzenie drzewa binarnego

Porządek *postorder*

Algorytm przechodzenia w porządku postorder jest następujący:

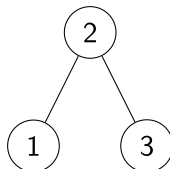
1. odwiedź lewe poddrzewo,
2. odwiedź prawe poddrzewo,
3. odwiedź korzeń.

Tak jak poprzednie algorytmy przechodzenia drzewa i ten jest implementowany w postaci funkcji rekurencyjnej. W tym algorytmie najpierw odwiedzane są poddrzewa, a na końcu korzeń. Tak jak w przypadku pozostałych algorytmów, następny slajd zawiera przykładową implementację oraz ilustrację działania postorder.

Przechodzenie drzewa

Porządek *postorder*

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



Wynik

1, 3, 2

Wypisanie zawartości drzewa

Zaprezentowane na poprzednich slajdach funkcje wypisywały klucze zgromadzone w drzewie zgodnie z określonym algorytmem jego przechodzenia. Nas będzie interesowała jednak cała zawartość węzłów drzewa (klucz i wartość), zatem na kolejnych trzech slajdach umieszczone są kody źródłowe funkcji, które ją wypisują w porządku inorder, preorder i postorder. Każda para klucz - wartość wypisywana jest w osobnym wierszu ekranu.

Wypisanie zawartości drzewa - porządek inorder

```
1  void print_inorder(struct tree_node *root)
2  {
3      if(root) {
4          print_inorder(root->left_child);
5          printw("klucz: %4d, wartość: %4c\n",
6                                                  root->key,root->value);
7          print_inorder(root->right_child);
8      }
9  }
```

Wypisanie zawartości drzewa - porządek preorder

```
1  void print_preorder(struct tree_node *root)
2  {
3      if(root) {
4          printw("klucz: %4d, wartość: %4c\n",
5                                  root->key,root->value);
6          print_preorder(root->left_child);
7          print_preorder(root->right_child);
8      }
9  }
```

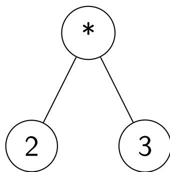
Wypisanie zawartości drzewa - porządek postorder

```
1  void print_postorder(struct tree_node *root)
2  {
3      if(root) {
4          print_postorder(root->left_child);
5          print_postorder(root->right_child);
6          printw("klucz: %4d, wartość: %4c\n",
7                                          root->key,root->value);
8      }
9  }
```

Drzewo arytmetyczne

Algorytmy przechodzenia drzewa mają o wiele więcej zastosowań niż tylko wypisywanie zawartości drzewa BST lub innego drzewa binarnego. Załóżmy, że udałoby nam się skonstruować drzewo binarne, które zawierałoby wyrażenie arytmetyczne $2 \cdot 3$, gdzie operator mnożenia byłby zapisany w korzeniu drzewa, a w liściach byłby zapisane jego argumenty. W ten sposób struktura drzewa określa, że dana operacja będzie wykonywana na określonych argumentach. Tego typu drzewo binarne nazywane jest *drzewem arytmetycznym*. Jeśli teraz zastosujemy dla tego drzewa algorytm inorder, to otrzymamy wyrażenie arytmetyczne w postaci wrostkowej, czyli takiej, jakiej używamy w matematyce na co dzień. Jeśli użyjemy algorytmu preorder to dostaniemy wyrażenie w notacji przedrostkowej, czyli w Notacji Polskiej, a algorytm postorder wygeneruje nam wyrażenie w notacji przyrostkowej, czyli Odwrotnej Notacji Polskiej. Kolejne slajdy zawierają ilustrację opisanego zagadnienia. Drzewa arytmetyczne nie muszą być takie proste, mogą opisywać bardziej skomplikowane wyrażenia.

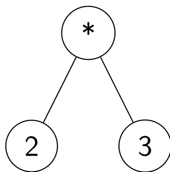
Drzewo arytmetyczne



Porządek inorder

2 * 3

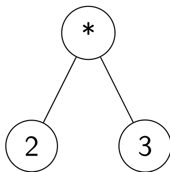
Drzewo arytmetyczne



Porządek preorder

* 2 3

Drzewo arytmetyczne



Porządek postorder

2 3 *

Wypisanie zgodnie ze strukturą drzewa

Interesujące byłoby wypisanie zawartości BST, w sposób ilustrujący na ekranie strukturę tego drzewa. W tym celu wykorzystamy możliwości jakie daje biblioteka `curses`. Aby struktura drzewa była łatwa do odczytania założymy, że kolejne poziomy drzewa[2] będą wypisywane co dwa wiersze. Użytym algorytmem przechodzenia drzewa będzie preorder, ponieważ zaczyna jego odwiedzanie od korzenia i korzeni jego poddrzew, które jako pierwsze muszą być wypisane na ekranie. Ponieważ wypisanie pary klucz - wartość na raz mogłoby zmniejszyć czytelność informacji na ekranie, to zostaną zaprezentowane dwie funkcje ilustrujące strukturę BST. Pierwsza z nich wypisuje na ekran wartości, a druga klucze.

[2] Poziom drzewa zawiera wszystkie jego węzły, które mają taki sam poziom.

Wypisanie zgodnie ze strukturą drzewa - wypisanie wartości

```
1  void print_values(struct tree_node *root, int x, int y,
2                                               unsigned int distance)
3  {
4      if(root) {
5        mvprintw(y,x,"%4c",root->value);
6        print_values(root->left_child,x-distance,y+2,distance/2);
7        print_values(root->right_child,x+distance,y+2,distance/2);
8      }
9  }
```

Wypisanie zgodnie ze strukturą drzewa - wypisanie wartości

Funkcja `print_values()` posiada cztery parametry. Pierwszym jest wskaźnik, pod który będą podstawiane adresy korzenia BST i korzeni jego poddrzew. Dwa kolejne parametry to współrzędne miejsca na ekranie, gdzie zostanie umieszczona wartość bieżąco odwiedzanego węzła, przed ostatni parametr przekazywana jest połowa odległości jaka dzieli dwa węzły będące rodzeństwem. Jest ona uwzględniana nawet wtedy, gdy istnieje tylko jeden węzeł z rodzeństwa. Funkcja najpierw sprawdza, czy została wywołana dla istniejącego węzła drzewa (wiersz nr 4). Jeśli tak, to wypisuje wartość tego węzła w miejscu ekranu określonym przez przekazane jej współrzędne, a następnie wywołuje się rekurencyjnie dla lewego i prawego potomka tego węzła. Proszę zwrócić uwagę, na użycie czwartego parametru w tych wywołaniach. W przypadku lewego potomka jest on odejmowany od współrzędnej poziomej jego przodka, a w przypadku prawego potomka jest on dodawany. Współrzędna pionowa jest zwiększana w obu przypadkach o 2. Odległość między tymi potomkami jest skracana o połowę.

Wypisanie zgodnie ze strukturą drzewa - wypisanie kluczy

```
1  void print_keys(struct tree_node *root, int x, int y,
2                                               unsigned int distance)
3  {
4      if(root) {
5          mvprintw(y,x,"%4d",root->key);
6          print_keys(root->left_child,x-distance,y+2,distance/2);
7          print_keys(root->right_child,x+distance,y+2,distance/2);
8      }
9  }
```

Wypisanie zgodnie ze strukturą drzewa - wypisanie kluczy

Funkcja wypisująca klucze jest zaimplementowana w ten sam sposób, jak funkcja wypisująca wartości, z dokładnością do wiersza nr 5, gdzie zamiast pola `value` wypisywane jest pole `key`.

Usuwanie wszystkich węzłów drzewa binarnego

```
1   void remove_tree_nodes(struct tree_node **root)
2   {
3       if(*root)
4       {
5           remove_tree_nodes(&(*root)->left_child);
6           remove_tree_nodes(&(*root)->right_child);
7           free(*root);
8           *root = NULL;
9       }
10  }
```

Usuwanie wszystkich węzłów drzewa binarnego

Funkcja usuwająca wszystkie węzły drzewa binarnego działa zgodnie z algorytmem postorder. Dzięki temu nie ma niebezpieczeństwa, że do jej rekurencyjnych wywołań będą przekazane adresy nieistniejących pól. Musimy 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 `root`. Taki zapis powoduje także, że tę wartość dostaje każde z pól wskaźnikowych węzłów, tuż przed ich usunięciem. Zastosowany algorytm powoduje, że węzły drzewa są usuwane począwszy od liści, a skończywszy na korzeniu.

Funkcja main()

Część pierwsza

```
1  int main(void)
2  {
3      if(setlocale(LC_ALL,"")==NULL) {
4          fprintf(stderr,"Błąd inicjacji ustawień językowych!\n");
5          return -1;
6      }
7      if(!initscr()) {
8          fprintf(stderr,"Błąd inicjacji biblioteki curses!\n");
9          return -1;
10     }
11     int i;
12     srand(time(0));
13     for(i=0;i<10;i++) {
14         int key = 0; char value = 0;
15         value='a'+rand()%('z'-'a'+1);
16         key = (int)value;
17         add_node(&root,key,value);
18     }
```

Funkcja `main()`

Część pierwsza

W pierwszej części funkcji `main()` wykonywana jest inicjacja pracy biblioteki `curses` oraz generatora liczb pseudolosowych, a następnie budowane jest drzewo o dziesięciu węzłach zawierających małe litery alfabetu łacińskiego (wiersze od 13 do 18).

Funkcja main()

Część druga

```
1         printw("Wartości w drzewie w porządku inorder:\n");
2         print_inorder(root);
3         refresh();
4         getch();
5         erase();
6         printw("Wartości w drzewie w porządku postorder:\n");
7         print_postorder(root);
8         refresh();
9         getch();
10        erase();
11        printw("Wartości w drzewie w porządku preorder:\n");
12        print_preorder(root);
13        refresh();
14        getch();
15        erase();
```

Funkcja `main()`

Część druga

W drugiej części funkcji `main()` wypisywana jest zawartość BST w każdym z poznanych porządków, czyli inorder, postorder i preorder. Po wykonaniu każdej z odpowiednich funkcji, działanie programu jest wstrzymywane, do czasu naciśnięcia przez użytkownika klawisza na klawiaturze, a następnie ekran jest czyszczony.

Funkcja main()

Część trzecia

```
1        printw("Struktura drzewa (wartości):");
2        print_values(root,COLS/2,1,20);
3        refresh();
4        getch();
5        erase();
6        printw("Struktura drzewa (klucze):");
7        print_keys(root,COLS/2,1,20);
8        refresh();
9        getch();
10       erase();
11       remove_tree_nodes(&root);
12       if(endwin()==ERR) {
13           fprintf(stderr,"Błąd funkcji endwin()!\n");
14           return -1;
15       }
16       return 0;
17   }
```

Funkcja `main()`

Część trzecia

W trzeciej części funkcji `main()` wypisywane są wartości i klucze zgromadzone w węzłach BST tak, aby zilustrować strukturę tego drzewa. Proszę zwrócić uwagę na współrzędne pierwszego wypisywanego węzła, czyli korzenia drzewa. Wartość współrzędnej pionowej wynosi `1`, czyli będzie on wypisany w drugim wierszu ekranu. Wartość współrzędnej poziomej to `COLS/2`, gdzie `COLS` to stała określająca liczbę kolumn ekranu. Oznacza to, że wartość lub klucz korzenia zostaną wypisane w połowie drugiego wiersza ekranu. Dzięki temu struktura BST powinna być w miarę dobrze widoczna, choć efekt końcowy zależy od tego jakie zawartości jego węzłów zostały wygenerowane. Tak jak poprzednio po wypisaniu kluczy i wartości wykonanie programu jest wstrzymywane do czasu naciśnięcia przez użytkownika klawisza na klawiaturze. Po tym zwartość ekranu jest czyszczona. Dodatkowo po wypisaniu kluczy z drzewa usuwane są wszystkie węzły, kończona jest praca biblioteki `curses` i program kończy swoją pracę.

Podsumowanie

Właściwości drzewa BST zostaną bliżej przedstawione na następnym wykładzie. Warto jednak zaznaczyć, że drzewa, nie tylko BST i binarne w ogólności, są bardzo elastycznymi strukturami, powszechnie stosowanymi w kompilatorach (wspomniane drzewa arytmetyczne), systemach operacyjnych (np. algorytm CFS w Linuksie), grafice komputerowej (drzewa opisujące tworzone sceny) i innych zagadnieniach (od programów dla księgowości do sztucznej inteligencji). W wielu zastosowaniach drzewa okazują się najbardziej efektywnymi strukturami danych.

Pytania

?

KONIEC

Dziękuję Państwu za uwagę!