Damian Szopiński

Maciej Pestka

Użyte biblioteki

Uwaga. W projekcie użyłem biblioteki

(github.com) <https://github.com/pulzed/mINI/blob/master/src/mini/ini.h>

Pliki .cpp są już w projekcie w folderze simpleini. Użyłem go, gdyż jest to multiplatformowy odpowiednik i musieliśmy zrezygnować z wbudowanych windowsowych funkcji.

Istotne kwestie

# Mieliśmy duży dylemat nad podpunktem 1 wymagań dodatkowych.

W naszym przypadku ani wariant A, ani wariant B nie był najlepszym rozwiązaniem. Ostatecznie zdecydowaliśmy się wybrać wariant C (napisany białym drukiem).

- U nas to użytkownik może decydować jakie dane będą przechowywane w drzewie binarnym (**template<typename typ\_danych>**). Problem staną się jednak klucze. Skoro typ danych nie jest określny, to na jakich podstawach dane mogły by determinować klucze.

Ostatecznie stwierdziliśmy, że najlepiej będzie jak drzewo będzie pobierać od nas dwa typy danych. Jeden który będzie typem klucza, a drugi typem danych jaki będzie przechowywać drzewo. (**template<typename typ\_klucza,typename typ\_danych>**)

W ten sposób to użytkownik będzie mógł determinować o interpretacji klucza. Można utworzyć klasę, która będzie przeciążać operatory mniejszości „<” i większości „>” oraz opcjonalnie „<<” dla wyświetlania elementów na ekranie.

- Sposób implementacji template, też stawiał delikatny problem. Chodzi o komunikacji typów danych między plikami h oraz cpp. Znaleźliśmy jednak rozwiązanie na stronie: <https://stackoverflow.com/questions/495021/why-can-templates-only-be-implemented-in-the-header-file>

Spełnione podpunkty

**Wymagania podstawowe:**

✅ 1 – spełniony

✅ 2 – spełniony

✅ 3 – spełniony

✅ 4 – spełniony

✅ 5 – spełniony

✅ 6 – spełniony

**Wymagania dodatkowe:**

✅ 1 – spełniony

✅ 2 –spełniony

✅ 3 – spełniony

✅ 4 – spełniony

✅ 5 – spełniony

Zdublowane klucze

To nie do końca jest tak, że to drzewo przechowuje zdublowane klucze, a przynajmniej od strony technicznej. Od strony użytkownika jednak będzie to działać bardzo podobnie. Węzeł zamiast samych danych przechowuje tablicę danych. Kiedy Insert wykryje, że klucz się powtarza, to ten znajduje istniejący węzeł i dokłada do tablicy dane które ten węzeł by przechowywał. Nie powstaje kolejna gałąź, tylko do tej samej gałęzi dokładane są kolejne informacje. W takim przypadku Delete nie kasuje całego węzła a tylko te informacje co najpóźniej się pojawiły. Cały węzeł jest kasowany, wyłącznie wtedy kiedy wszystkie dane „data” o tym samym kluczu zostałyby puste.

Węzeł

# Klasa BSTNode<typ\_klucza,typ\_danych>:

**concept RightType** – zastrzeżenie, że typ zmiennej który korzysta z BST, musi:

- Potrafić wypisać informacje w wierszu poleceń (cout<<dane)

- Współpracować z stringstream w celu zapisu i odczytu z pliku

- Musi dać się porównać (mniejszość, większość) oraz przyrównać (=)

*Klasa zawiera* ***key*** *która jest kluczem determinującym o pozycji na drzewie. Jest on o typie zmiennej podanej w template ORAZ* ***data*** *o typie zmiennej podanej w drugim templacie.*

Inaczej mówiąc. Key determinuje od pozycji na drzewie binarnym a jego typ zależy od wyboru użytkownika, o ile odpowiednie operatory funkcjonują. Pole Data również zależy od wyboru użytkownika, jednakże nie ma ono innego zastosowania niż czysto przechowywania danych i wyświetlania ich na ekranie. Zostało ono stworzone by spełniać wszystkie podpunkty z zadania.

**Konstruktor BSTNode(key\_t k)** który nadpisuje dane key na podany w argumencie.

**string Data()** – zwraca string z wszystkimi danymi „data” o tym samym kluczu.

*Oprócz tego posiada 3 wskaźniki wskazujące odpowiednio na ojca, lewego syna oraz prawego syna. Odpowiednik u nas* ***NIL*** *to* ***nullptr****.*

Drzewo binarne i podstawowe operacje

# Klasa BST<typ\_klucza,typ\_danych>:

Będzie przechowywać drzewo binarne i tworzyć odpowiednie BSTNode o takich samych typach zmiennych jakich podano przy tworzeniu tej klasy. Posiada jeden wskaźnik na BSTNode root.

(Wszystkie elementy poddrzewa będą operować na takich samych typach podanych przy deklaracji tej klasy)

**void Insert(const key\_t k,data\_t data)** – służy do dodawania nowego elementu na drzewie. Tworzy BSTNode o typach zmiennych podanych przy tworzeniu drzewa BST. Te 2 argumenty to oczywiście dane klucza oraz data jaki będzie posiadać nowy BSTNode. (Gdy klucz już istnieje to dokładamy do węzła kolejne dane)

Ta funkcja została zaimplementowana iteracyjnie (nie rekurencyjnie).

**void Transplant(BSTNode<key\_t,data\_t>\* u,BSTNode<key\_t,data\_t>\* v)** – funkcja przeszczepiająca węzły (węzeł z argumentów u z v). Przyda nam się do implementacji funkcji Delete.

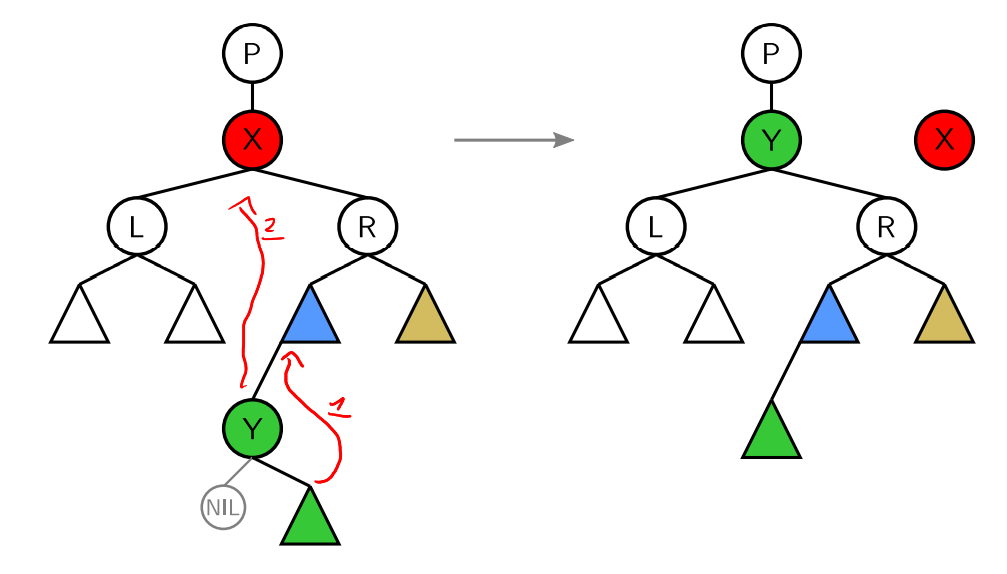
*Na następnej stronie Delete…*

**void Delete(const key\_t k) –** słyży do usuwania elementu z drzewa o danym kluczu. Posługuje się funkcją Search po odnaleźć dany element do skasowania.

Ta funkcja obsługuje 4 przypadki, które odpowiadają kolejnym po sobie if’om.

1. Jeżeli węzeł o kluczu k już istnieje, to usuń ostatnio otrzymane dane. Zakończ działanie funkcji.
2. Gdy wcześniejsze nie zachodzą i węzeł nie posiada dzieci, więc wystarczy by ojciec zapomniał o swoim dziecku. W tym fragmencie poszukujemy czy jest to lewe czy prawe dziecko ojca oraz ustawiamy odpowiedni wskaźnik na nullptr.
3. Gdy wcześniejsze nie zachodzą i węzeł posiada prawe dziecko. Więc trzeba transplanować dziecko „do tyłu”.
4. Analogicznie dla lewego dziecka.
5. Gdy wcześniejsze nie zachodzą i węzeł ma prawe i lewe dziecko. Postępuje zgodnie z rysunkiem przedstawionym na wykładzie.

Na koniec usuwamy węzeł.



**void Clear(BSTNode<key\_t,data\_t>\* node=nullptr) –** Kasuje poddrzewo i wszystkie jego pod węzły. Argument domyślny to jak zawsze root (kasowanie całego drzewa). Tak samo usuwana jest relacja ojca z synem.

Funkcja ta jest zaimplementowana rekurencyjnie.

Funkcje dodające do tablicy odpowiednie wskaźniki

**void VectorOfNodes(BSTNode<key\_t,data\_t> \*node,vector<BSTNode<key\_t,data\_t>\*>& vec) –** Wykorzystuje wskaźnik na tablice z argumentu vec i zapisuje na nim (push\_back) wszystkie wskaźniki do BSTNodes dla poddrzewa node *(TUTAJ ARGUMENT NIE JEST DOMYŚLNY).*

*Uwaga: Wcześniej utworzona tablica vec musi posiadać takie same typy zmiennych co dane poddrzewo BSTNode. Funkcja nic nie zwraca. Jedyny produkt to zmiana tablicy vec.*

Funkcja jest zaimplementowana rekurencyjnie.

**Void VectorOfNodesAscending (BSTNode<key\_t,data\_t>\*node,std::vector<BSTNode<key\_t,data\_t>\*>& vec) –** Działa tak samo jak poprzednia funkcja, różni się wyłącznie tym, że nie wektor zostanie zwrócony w kolejności rosnących kluczy.

*Panuje tu takie rozdzielenie, gdyż VectorOfNodesAscending służy mi do znajdywania Predecessor i Successor natomiast VectorOfNodes słyży do zapisywania elementów do pliku w odpowiedniej kolejności.*

**void ListOfNodesInLevel(const int level,std::vector<AiSD::BSTNode<key\_t,data\_t>\*>& table,AiSD::BSTNode<key\_t,data\_t>\* node=nullptr) –** Analogicznie, tylko dla danego poziomu głębokości (podanych w obowiązkowym argumencie level) poziom 0 to root. Tutaj wskaźnik (na poddrzewo node jako argumet funkcji) jest już opcjonalny bo treść zadania tego wymagała.

Funkcje zwracające odpowiednie węzły

Jeżeli jakiś klucz był „zdublowany”, to zwrócone będą wszystkie dane zdublowanego klucza!

**BSTNode<key\_t,data\_t>\* Search(const key\_t k,BSTNode<key\_t,data\_t>\* subtree\_root=nullptr) –** Poszukiwanie węzła w poddrzewie subtree\_root. Przeszukuje wszystkie element I sprawdza czy klucz jest taki sam jak ten który szuka. Jak tak to zwraca ten wskaźnik . Subtree\_root jest argumentem opcjonalnym. Stosuje tutaj taką praktykę, że domyślny nullptr najpierw sprawdza czy root jest również nullptr, jeżeli nie, to subtree\_root zamieina się na root. Tą praktykę będę powtarzał również w kolejnych funkcjach.

Ta funkcja zaimplementowana jest iteracyjnie.

**BSTNode<key\_t,data\_t>\* Predecessor(const key\_t k,BSTNode<key\_t,data\_t>\* subtree\_root=nullptr)** - próbuje znaleźć element mniejszy. Wykorzystuje do tego VectorOfNodesAscending. Znajduje element o danym kluczu i wyszukuje odrobinę mniejszy (poprzedni element tablicy).

**BSTNode<key\_t,data\_t>\* Successor(const key\_t k,BSTNode<key\_t,data\_t>\* subtree\_root=nullptr) –** tak samo jak poprzednia funkcja tylko wyszukuje odrobinę większy (następny element tablicy).

**BSTNode<key\_t,data\_t>\* Min(BSTNode<key\_t,data\_t>\* subtree\_root=nullptr) –** Tutaj ponownie mamy argument domyślny który działa na takiej samej zasadzie co poprzednio. Funkcja zwraca wskaźnik na węzeł o „najmniejszym” kluczu (cokolwiek to znaczy dla danego typu danych). Argument to podwęzeł w którym będziemy szukać minimum. (Działa na zasadzie poruszania się po lewych stronach węzłach)

Funkcja ta działa iteracyjnie.

**BSTNode<key\_t,data\_t>\* Max(BSTNode<key\_t,data\_t>\* subtree\_root=nullptr) –** działa analogicznie jak poprzednia funkcja tylko zamiast tego szuka maximum. (Działa na zasadzie poruszania się po prawych stronach węzłach)

Funkcja ta działa iteracyjnie.

Funkcje drukujące

**void PrintAscending(BSTNode<key\_t,data\_t> \*node=nullptr,int deep=0)** – Funkcja rysująca na ekranie wartości klucza (jak nazwa wskazuje rosnąco) dla danego poddrzewa node albo jeżeli zastosujemy argument domyślny, dla roota.

Uwaga: Widoczne kropki przy wartościach kluczy to nic innego jak głębokość „zanurzenia” w węzłach. Bardzo to pomaga w rozstrzygnięciu kolejności elementów na drzewie. (każda kropka to kolejne wejście w kolejne poddrzewo) Właśnie tym jest argument Deep;

Funkcja jest napisana rekurencyjnie.

**void pointersInfo(BSTNode<key\_t,data\_t>\* subtree\_root) –** to jest dodatkowa funkcja testująca którą postanowiliśmy zostawić. Służy do wyświetlania w oknie CMD informacji o wskaźnikach dla danego BSTNode. W ten sposób sprawdzaliśmy czy węzeł wskazywał na odpowiednie inne węzły.

Odczyt/Zapis do pliku

**void Save(string src) –** zapisywanie danych drzewa do pliku INI o ścieżce src. Wykorzystuje tutaj bibliotekę o której wspomniałem wcześniej. Korzysta z **VectorOfNodes** do zapisu danych w takiej kolejności jaka powinna być.

Tutaj zastosowałem sprytną sztuczkę do zapisu za pomocą sstringa i operatorów << i >>. W ten sposób większość podstawowych typów jak np. float, string itp. Będą ze sobą współpracowały. Jeżeli będziemy operować na swojej klasie to będziemy musieli przeciążyć te operatory z const char[] aby ta funkcjonalność działała. Natomiast o tym będę jeszcze wspominać później przy funkcjach konwertujących.

**PRZYKŁADOWY FORMAT PLIKU**

|  |
| --- |
| [0] //KOLEJNE DANE KTÓRE BĘDĄ SCZYTYWANE W TAKIEJ KOLEJNOŚCI DO INSERTION  key=10 //TREŚĆ KLUCZA  0=wartość klucza 10//TREŚĆ DANYCH  [1]  key=2  0=Wartość klucza 2  1=Wartość zdublowanego klucza 2  […]//I TAK DALEJ |

**void Load(string src) –** analogicznie do poprzedniego punktu. Funkcja sczytuje wszystkie grupy ini oraz ich itemy dla ścieżki o danym src. Nie ma tutaj wartości domyślnych dla zawartości pliku jak to często ma miejsce w plikach ini, dlatego trzeba być ostrożny co wpisujemy do pliku.

Funkcje Zliczające

**int height(BSTNode<key\_t,data\_t>\* subtree\_root=nullptr,int sum=0) –** Zwraca wysokość drzewa binarnego dla poddrzewa subtree.

Funkcja działa rekurencyjnie. (Dlatego jest ten opcjonalny argument sum (do komunikacji między rekurencjami). Użytkownik nie powinien zmieniać domyślnej wartości argumentu sum!). Jeżeli zostawimy same wartości domyślne to sprawdzona zostanie wysokość dla roota.

**int leavesCount(BSTNode<key\_t,data\_t>\* subtree\_root=nullptr) –** zlicza liczbę liści poddrzewa subtree. Działanie analogiczne do poprzedniego tylko nie potrzebowałem tutaj dodatkowego argumentu sum.

**int nodesCount(BSTNode<key\_t,data\_t>\* subtree\_root=nullptr) –** zlicza liczbę węzłów. Analogicznie do poprzedniego.

**int nodesCountOnLevel(const int level,BSTNode<key\_t,data\_t>\* subtree\_root=nullptr) –** korzysta z funkcji ListOfNodesInLevel() I zwraca wielkość otrzymanej tablicy.

Uwaga: Poziomy liczone są od 0, a nie od 1. Zupełnie inaczej było z funkcją height który roota uważał jako wysokość 1.

Operacje na typach danych

Dodatkowe funkcje ułatwiające konwertujące – służą one głównie do konwersji danych do pliku, ale w założeniu wspomagają też funkcje rysujące w oknie CMD.

**template <typename Type>Type convert(string str) –** Korzystając z sstringa zamienia stringa z argumentu na zmienną o typie podaną w template oraz zwraca tą zmienną.

(String->Typ danych)

**template <typename Type>string convertString(const Type val) –** sytuacja odwrotna. Zamienia dany typ danych w templacie na string.

(Typ danych -> String)

Uwaga: Spacja nie będzie obsługiwana i automatycznie będzie ona zamieniana na „\_”.