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 „<<” dla wyświetlania elementów na ekranie (ostream, stringstream).

- 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/a/495056

Spełnione podpunkty

**Wymagania podstawowe:**

✅ 1 – spełniony

✅ 2 – spełniony

✅ 3 – spełniony

✅ 4 – spełniony

✅ 5 – spełniony

✅ 6 – spełniony

**Wymagania dodatkowe:**

✅ 1 – spełniony

● 2 –częściowo

✅ 3 – spełniony

✅ 4 – spełniony

✅ 5 – spełniony

Zdublowane klucze

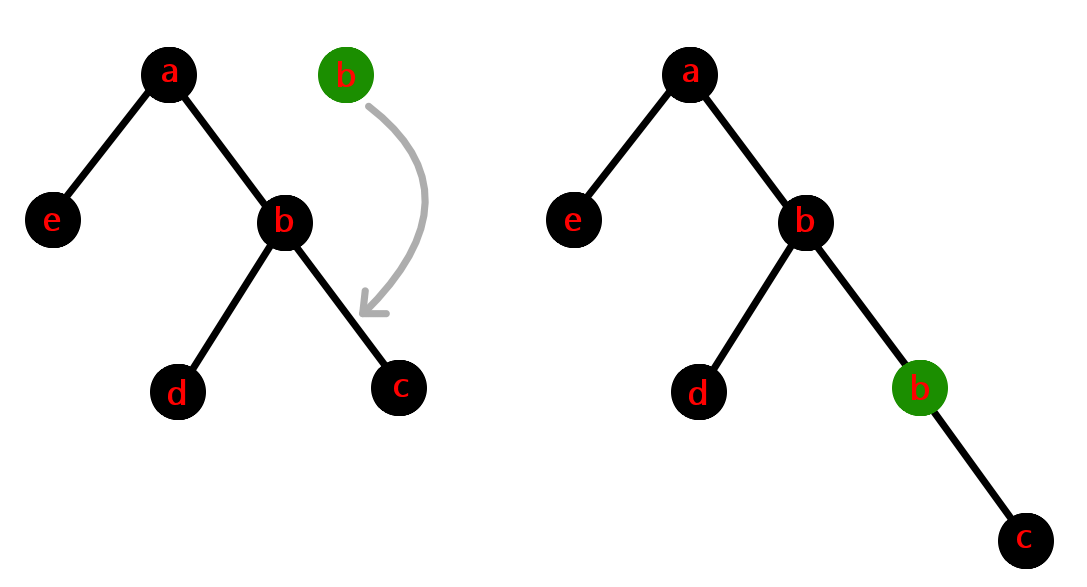
Uwaga! w naszej implementacji, dopuszczamy możliwość zdublowanych kluczy pod warunkiem niektórych wyjątków od lematów drzewa czerwono czarnego (prawy i lewy syn może mieć taki sam klucz). Dla zachowania jednak dokładnego drzewa czerwono czarnego zdublowane klucze są wyłączone i aby je odblokować należy wpisać komentarz w wskazane miejsce w kodzie.

Skąd taka decyzja? Pomysł o zdublowanych kluczach jest taki sam co w poprzednim projekcie, to znaczy rozstawiamy zdublowane klucze po prawych dzieciach (obrazek niżej). W drzewie czerwono czarnym następują jednak obroty podczas funkcji fix przez co ta kolejność może ulec zmianie. Kolejność minimum, maksimum i innych funkcji nie powinna ulec zmianie, ponieważ przyjmiemy konwencje, że klucze późniejsze będą jak najdalej prawej. Funkcje testujące mogą jednak zauważyć tą niespójność.

Domyślnie program będzie ignorować insert zdublowanych kluczy.

Schemat postępowania:

Zdublowany klucz jest zawsze po prawej stronie. Nowy węzeł otrzymuje prawy węzeł starego węzła, a lewy pozostaje bez zmian. W ten sposób zachowamy własności drzewa binarnego.



**Uwaga obrazek ten pokazuje wynik operacji PRZED wykonaniem funkcji FixInsert.**

Drzewo czerwono-czarne

Najistotniej chyba w tym projekcie będzie wspomnieć czym różni się ten program od poprzedniego projektu. Większość kodu została taka sama jedynie zastosowałem następujące zmiany:

Null’e (nie wyświetlane na drzewie) uznajemy za czarne

**enum Color** – zawiera się w każdym węźle (RBTNode->color). Może przyjmować RED,BLACK oraz odpowiednik „podwójnego czarnego” albo „siwego” w naszej implementacji nazwaliśmy różowym (PINK) dla funkcji Delete.

Funkcje:

**void ReplaceColor(Node\* subtree\_root,Color newColor)** – zamienia kolor węzła subtree na ten podany w drugim argumencie. Posiada zabezpieczenie by program nie próbował zamienić koloru dla węzła NIL.

**Color WhatColorIs(Node\* subtree\_root)** – Zwraca kolor węzłą subtree. Jeżeli węzeł nie istnieje to czarny.

**rotateLeft(Node\* subtree\_root)** – obraca drzewo wokół węzła subtree w lewo.

**rotateRight(Node\* subtree\_root)** – obraca drzewo wokół węzła subtree w prawo.

**Funkcja void FixInsert(Node\* subtree\_root)** - wywoływaną na końcu funkcji Insert, jest realizacją algorytmu przedstawionego na wykładzie na slajdzie 19.

Usuwanie:

W przypadku usuwania przyjęliśmy nieco inną metodę. Publiczna **Delete(key k)** zleca zadanie dla prywatnej funkcji **Delete(Node\* subtree,key k)** której celem jest znalezienie odpowiedniego węzła oraz podmianę istniejących a **fixDelete** dopiero usuwa odpowiedni węzeł. Tym pomysłem zainspirowałem się przeglądając przykładowe implementacje na githubie, między innymi tym [LINK](https://github.com/anandarao/Red-Black-Tree) albo tym [LINK](https://github.com/sandeepnmenon/Red-Black-Tree-Implementation).

Oczywiście zmieniłem też wszystkie „BST” w kodzie na „RBT”.

Wyświetlanie drzewa i zapis do pliku drzewa

Wyświetlenie drzewa jest dla wysokości max 4, gdy drzewo ma większą wysokość, to wszystkie elementy nie pokazują się. Natomiast ostanie elementy wyświetlają tylko wartość klucza. Tak samo jest zapis do pliku. Niekiedy niektóre elmenty mogą być trochę odsunięte z powodu, że nie dało mi się ogarnać dokładnie ilość spacji, kresek.

A zapis drzewa jako w formie testu, jest bez widoki drzewa, wyświetlają są od wartości lewej do prawej, wszystkie NULL nie pojawią się, Gdy rodzić jest nullem to nie pojawią się jego synowie (NULL) .

TESTY

Na początku są testy dla wykonywanie dla drzew małych do 3 elementów, sprawdzają czy wartość wskaźników i root jest prawidłowa ustalona. Następne są testy dla większych drzew, gdzie na początku jest losowo generowanie drzewo. Tak samo losowo jest dodawanie i usuwanie wartości drzewa. Za każdą operacją jest sprawdzianie czy wskaźniki są poprawnie ustawione oraz czy jest popranie ustawione elementy lewy po lewej stronie rodzica a większy po prawej. Także czy root jest czarny i liście są czarne (nil). Także czy czerwony element nie ma czerwonego potomka. Gdy wykryję jakiś błąd to wyskoczy informacja na ekran. W detailsTest jest wiele warunków, które odwoju się do metod, która sprawdza jedną z wielu opcji.

Funkcja randing() tworzy radonowe drzewo z i-elementami. A rand() dodaje tylko jeden element do drzewa, wartość przyjmuję liczbę, która później jest mnożono z root i dodania jako nowa wartość do drzewa.

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 typie.*

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,data\_t dataArg)** który nadpisuje dane key oraz data na podany w argumencie.

**enum Color** – zawiera się w każdym węźle (RBTNode->color). Może przyjmować RED,BLACK oraz odpowiednik „podwójnego czarnego” albo „siwego” w naszej implementacji nazwaliśmy różowym (PINK) dla funkcji Delete.

*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)

W naszym drzewie null’e też uznajemy za czarne,

**rotateLeft(Node\* subtree\_root)** – obraca drzewo wokół węzła subtree w lewo.

**rotateRight(Node\* subtree\_root)** – obraca drzewo wokół węzła subtree w prawo.

**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 się doubluje dodaje go po prawej stronie węzła o tym samym kluczu.

Jeżeli chcemy zablokować dodawanie elementów o tym samym kluczu, należy usunąć komentarz w wskazanym miejscu w kodzie.

Na końcu wywołuje FixInsert dla nowego węzła w argumencie.

Ta funkcja została zaimplementowana iteracyjnie (nie rekurencyjnie).

**Void FixInsert** – naprawia drzewo czerwono czarne po wstawieniu nowego elementu.

**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.

**void ReplaceColor(Node\* subtree\_root,Color newColor)** – zamienia kolor węzła subtree na ten podany w drugim argumencie. Posiada zabezpieczenie by program nie próbował zamienić koloru dla węzła NIL.

**Color WhatColorIs(Node\* subtree\_root)** – Zwraca kolor węzła subtree. Jeżeli węzeł nie istnieje to czarny.

**void Delete(const key\_t k) –**Wywołuje poniższą funkcje oraz funkcje FixDelete dla uzyskanego wskaźnika.

**Node\* Delete(Node\* subtree,key\_t k) –** Jego główne zadanie to podmiana odpowiedniego węzła i zwrócenie wskaźnika dla FixDelete.

Funkcja ta działa rekurentyjnie

**Void FixDelete(Node\* subtree) –** służy do naprawy kolorów węzła. Jest wywoływany przez Delete. Tak naprawdę to dopiero ta funkcja kasuje 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 NODE 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 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!

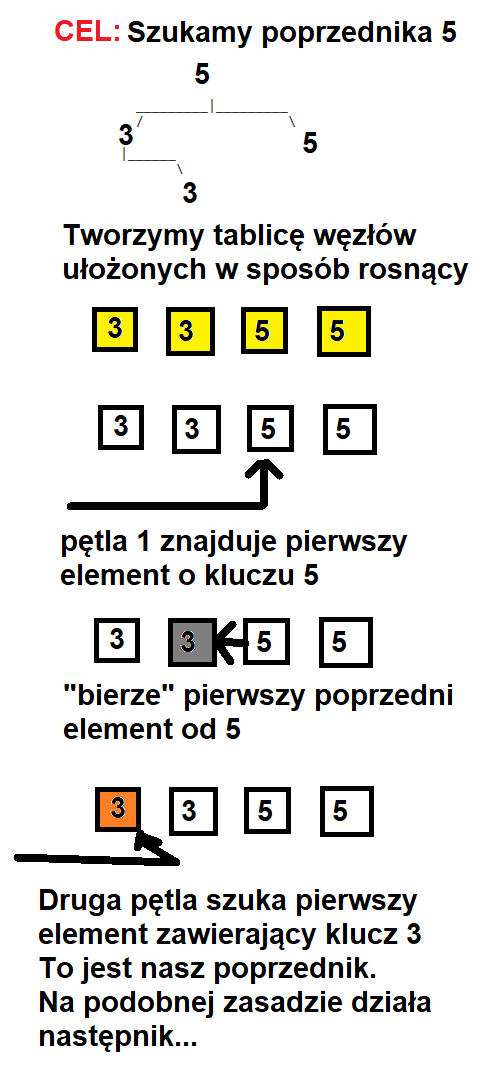
**BSTNode<key\_t,data\_t>\* Search(const key\_t k,BSTNode<key\_t,data\_t>\* subtree\_root=nullptr) –** Poszukiwanie węzła w poddrzewie subtree\_root. Przeszukuje wszystkie element I sprawdza czy klucz jest taki sam jak ten który szuka. Jak tak to zwraca ten wskaźnik . Subtree\_root jest argumentem opcjonalnym. 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, a jeżeli nie znajdzie, to zwraca ten sam. Wykorzystuje do tego VectorOfNodesAscending. Znajduje element o danym kluczu i wyszukuje odrobinę mniejszy (poprzedni element tablicy). Używamy tablicy węzłów ponieważ w ten sposób funkcje te zwracają konkretny poprzednik/sukcesor jednego poddrzewa a nie całości od roota. Nie mogłem się więc „cofać” po ojcach, ponieważ tak moglibyśmy wyjść poza poddrzewo. (stąd decyzja o tablicy)

**W momencie gdy nie istnieje poprzednik/następnik zwracany jest ten sam węzeł poddrzewa który podaliśmy w argumencie. Łatwo można to zmienić w wskazanych miejscach w kodzie (dodałem komentarze w kodzie w odpowiednich miejscach; wystarczy usunąć komentarz), uznaliśmy to jednak za wygodne, bo w ten sposób wiemy że BST nie jest puste, a klucz jednak istnieje.**

Co prawda jest czas szukania takich danych się wydłuży (bo musimy zbudować tablicę), ale jest to najprostrzy sposób jaki wymyśliłem w tym przypadku.

Poniższy rysunek przedstawia przykład działania funkcji Predecessor (dla lepszego zrozumienia). Jeszcze raz przypomnę, że szukamy węzła „najwyżej” o kluczu poprzednika.

**BSTNode<key\_t,data\_t>\* Successor(const key\_t k,BSTNode<key\_t,data\_t>\* subtree\_root=nullptr) –** tak samo jak poprzednia funkcja tylko znajduje następnika (następny element tablicy). Działa na podobnej zasadzie jak poprzednia funkcja tylko drugą pętle umieściłem w środku pierwszej.

**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 podobnej do znajdywania poprzednika i następnika)

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 podobnej do znajdywania poprzednika i następnika)

Funkcja ta działa iteracyjnie.

Funkcje wyświetlające w oknie CMD

**void PrintAscending(BSTNode<key\_t,data\_t> \*node=nullptr,int deep=0)** – Funkcja wyświetla na konosli wartości klucza (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.

Jest tu mała aktualizacja względem poprzedniego projektu. Teraz dla podanego w argumencie węzła dla jego ojca i dzieci funkcja pokazuje w konsoli informacje o ich danych, kluczu i kolorze (dla każdego z nich).

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 czyli od góry w głąb.

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 aby ta funkcjonalność działała. Natomiast o tym będę jeszcze wspominać później przy funkcjach konwertujących.

**PRZYKŁADOWY FORMAT PLIKU**

|  |
| --- |
| [0] //KOLEJNE INDEKSY KTÓRE BĘDĄ SCZYTYWANE W TAKIEJ KOLEJNOŚCI DO INSERTION  key=10 //TREŚĆ KLUCZA  data=abc//TREŚĆ DANYCH  [1]  key=2  data=def  […]//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=1) –** Zwraca wysokość drzewa binarnego dla poddrzewa subtree.

Funkcja działa rekurencyjnie. Jeżeli zostawimy same wartości domyślne to sprawdzona zostanie wysokość dla roota.

W argumencie sum decydujemy od jakiej liczby chcemy zacząć zliczać wysokość. Czy mamy liczyć wysokość drzewa od 1 (tak jest domyślnie wtedy wysokość samego roota będzie wynosić 1), czy np. od 0. Dodatkowo ten argument pomaga też mi w komunikacji między rekurencjami tej funkcji, gdyż ta działa rekurencyjnie.

**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. Tak naprawdę zwraca wielkość tablicy z funkcji VectorOfNodes(subtree\_root).

**int nodesCountOnLevel(const int level,BSTNode<key\_t,data\_t>\* subtree\_root=nullptr) –** korzysta z funkcji ListOfNodesInLevel() I zwraca wielkość otrzymanej tablicy. Inaczej zlicza ile jest węzłów na danym poziomie.

Uwaga: Poziomy liczone są od 0

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 podłogę „\_”.