Drzewa

Na poprzednich zajęciach została implementacja książki telefonicznej w oparciu o listę jednokierunkową, sortowaną przez wstawianie w trakcie jej wypełniania. Rozwiązanie to ma wadę, polegającą na tym, że w celu znalezienia zadanego elementu należy zawsze przejrzeć całą listę od początku – dla elementów, znajdujących się w pobliżu końca listy oznacza to konieczność przeanalizowania prawie wszystkich elementów listy. Aby wyeliminować tę niedogodność, można zastosować inną strukturę danych do przechowywania informacji o abonentach. Tą strukturą danych będzie jeden z rodzajów *drzew, drzewo poszukiwań binarnych* lub inaczej *drzewo BST*.

Listę charakteryzowało to, że każdy jej element miał co najwyżej jednego poprzednika i co najwyżej jeden element następny. Drzewo to struktura danych, której elementy, zwane *wierzchołkami* albo *węzłami* mają, podobnie jak listy, co najwyżej jeden element poprzedzający (rodzica), ale mogą mieć więcej niż jeden element następny (syna). Dwa wierzchołki drzewa połączone są *krawędzią*, oznaczającą, że te wierzchołki znajdują się w relacji rodzic-syn względem siebie. Ponadto mówi się, że wierzchołek, nie mający rodzica jest *korzeniem drzewa* (wierzchołek taki rysowany jest na diagramach u góry), a wierzchołek nie mający synów *liściem*. Każdy wierzchołek drzewa musi być osiągalny z korzenia poprzez jednoznaczną ścieżkę, czyli ciąg krawędzi, łączących się ze sobą. Warunek jednoznaczności ścieżki może być interpretowany jako zakaz występowania zamkniętych ścieżek (cykli) w drzewie.

Drzewo, które pozwala na istnienie maksymalnie dwóch synów każdego z wierzchołków nazywane jest *drzewem binarnym*. Synowie danego wierzchołka w drzewie binarnym określani są jako *lewy* i *prawy*. Jeżeli wierzchołek drzewa przechowuje informacje, pozwalające na dokonywanie porównań, oraz dla dowolnego wierzchołka drzewa binarnego spełniony jest warunek, że informacja, przechowywana przez lewego syna jest mniejsza (lub równa) od informacji w wierzchołku oraz informacja, przechowywana przez prawego syna, jest większa (lub równa) od informacji w wierzchołku to drzewo takie nazywamy drzewem poszukiwań binarnych lub drzewem *BST* (od ang. *Binary Search Tree*). W literaturze różnie traktowane jest zagadnienie unikalności kluczy w drzewie *BST*; pewne pozycje dopuszczają istnienie w drzewie kilku kluczy o tej samej wartości, oznacza to w powyższej definicji użycie nierówności nieostrych ("mniejszy lub równy", "większy lub równy") zamiast ostrych ("mniejszy", "większy"). Struktura drzewa *BST* nadaje się idealnie do przechowywania zbiorów danych, które powinny pozwalać na szybkie wyszukiwanie zadanych elementów, proces wyszukiwania jest podobny do algorytmu wyszukiwania binarnego. Ograniczeniem jest to, że wybrane kryterium oceniające relacje informacji w wierzchołkach może być tylko jedno dla danego drzewa.

Z racji potencjalnego istnienia więcej niż jednego następnika wierzchołka w drzewach (dowolnego typu) procedury, wymagające przeglądu wszystkich elementów drzewa (co ma miejsce np. podczas usuwania drzewa) zazwyczaj wymagają skorzystania z mechanizmu rekurencji bądź z jawnej implementacji operacji na stosie, celem odkładania adresów poszczególnych wierzchołków do późniejszego wykorzystania. Ma to związek z tym, że z każdego wierzchołka wychodzi potencjalnie więcej niż jedna krawędź, co nie miało miejsca w listach, gdzie istniała tylko jedna ścieżka, pozwalająca odwiedzić wszystkie pozycje listy.

Procedura wyszukująca element na podstawie klucza, względem którego rozłożone są elementy drzewa *BST*, powinna skorzystać z jego własności, dotyczących wartości kluczy synów. Celem wyszukania wartości, rozpoczynamy analizę od korzenia drzewa. Rozważając w każdej chwili dokładnie jeden wierzchołek drzewa, po sprawdzeniu jego istnienia, analizujemy przechowywaną przez niego wartość klucza. Jeśli wartość ta jest równa poszukiwanej – znaleźliśmy właściwy wierzchołek drzewa. Jeśli tak nie jest, do analizy w kolejnym kroku wybieramy lewy następnik wierzchołka, jeśli szukany klucz jest mniejszy niż klucz zawarty w aktualnie rozważanym wierzchołku, a w przeciwnym razie wybieramy prawy następnik wierzchołka. Procedurę kontynuujemy do momentu znalezienia właściwego elementu lub napotkania wierzchołka, który nie posiada istniejącego następnika, w którego kierunku należy się przemieścić.

Procedura tworzenia drzewa *BST* jest bardzo podobna do procedury wyszukiwania. Dodanie elementu do pustego drzewa oznacza umieszczenie go jako korzenia. Jeśli drzewo nie jest puste, dokonujemy analogicznego przeglądu wierzchołków, jak podczas wyszukiwania, z dwoma wyjątkami: znalezienie w drzewie wierzchołka z kluczem, który chcemy dodać, jest błędem, o ile nie dopuszczamy duplikujących się kluczy (próba złamania unikalności klucza), oraz procedura dodająca wierzchołek do drzewa poszukuje wierzchołka, w którym nie ma następnika po właściwej stronie (i dołącza tam nowy element), a nie kończy się w tym punkcie, jak procedura wyszukująca.

Przykładowa aplikacja, przechowująca książkę telefoniczną w formie drzewa *BST* o unikalnych kluczach, umożliwiająca wyświetlanie alfabetyczne listy abonentów, wyszukanie abonenta po nazwisku oraz dodawanie abonentow do bazy wygląda następująco (plik *drzewo.cpp*):

```
#include <iostream>
#include <string>
#include <new>
using namespace std;

struct abonent
{
    string imie;
    string nazwisko;
    string numer;

    abonent *lewy;
    abonent *prawy;
};

// prototypy
bool dodajAbonenta(abonent **korzen, abonent *dane);
abonent *szukajAbonenta(abonent *korzen, string nazwisko, unsigned long *kroki = NULL);
void usunDrzewo(abonent **korzen);
void wyswietlDrzewo(abonent *korzen);
abonent *wczytajAbonenta();
void wypiszAbonenta(abonent *co);

int main()
{
    cout << "Program utworzy baze numerow telefonicznych" << endl << endl;
    cout << "Podaj czynnosc do wykonania:" << endl;
    cout << "d - dodaj wpis do bazy" << endl;
    cout << "s - szukaj osoby o podanym nazwisku" << endl;
    cout << "w - wypisz liste rekordow w bazie" << endl;
    cout << "z - zakoncz dzialanie programu" << endl << endl;

    abonent *drzewo = NULL;
    unsigned long licznik = 0;
    char o;
    do {
        cout << "Podaj czynnosc do wykonania [d/s/w/z]: ";
        cin >> o; cout << endl;
        cin.ignore();

        // Zamień na dużą literę
        o = toupper(o);
        switch(o)
        {
            case 'D': {
                    cout << "Wpisz dane nowej osoby: " << endl <<endl;
                    abonent *nowy = wczytajAbonenta();
                    if(nowy != NULL) {
                            if(dodajAbonenta(&drzewo, nowy)) licznik++;
                            else cout << "Blad (konflikt nazwisk)" << endl;
                    }
                    else cout << "Blad utworzenia nowego abonenta" << endl;
                    break;
                }
```

```
            case 'S': {
                    cout << "Podaj szukane nazwisko: ";
                    string szukany;
                    getline(cin, szukany);

                    unsigned long ile;
                    abonent *znaleziony = szukajAbonenta(drzewo, szukany, &ile);
                    if(znaleziony == NULL)
                            cout << "Abonenta nie ma w bazie, przeglad objal " << ile << " z "
                                    << licznik  << " rekordow" << endl << endl;
                    else {
                            cout << "Znalezienie abonenta wymagalo przegladu " << ile << " z "
                                    << licznik << " rekordow bazy" << endl << endl;
                            cout << "Znaleziono nastepujaca osobe:" << endl;
                            wypiszAbonenta(znaleziony);
                            cout << endl;
                    }
                    break;
                }

            case 'W':
                cout << "Baza zawiera obecnie " << licznik << " abonentow:" << endl << endl;
                wyswietlDrzewo(drzewo);
                cout << endl;
                break;

            case 'Z': break;
            default: cout << "Dopuszczalne opcje to d [dodaj], s [szukaj], w [wypisz] lub z [zakoncz]" << endl;
        }
    } while(o != 'Z');
    usunDrzewo(&drzewo);
}

bool dodajAbonenta(abonent **korzen, abonent *dane)
{
    if(korzen == NULL || dane == NULL) return false;
    if(*korzen == NULL) {
        // W wypadku pustego drzewa dodajemy nowy element jako korzen
        *korzen = dane;
        return true;
    } else {
        abonent *aktualny = *korzen;
        while(aktualny != NULL)
        {
            if(aktualny->nazwisko == dane->nazwisko) return false;
            if(dane->nazwisko < aktualny->nazwisko)
                if(aktualny->lewy != NULL) aktualny = aktualny->lewy;
                else {
                    aktualny->lewy = dane;
                    return true;
                }
            else // dane->nazwisko > aktualny->nazwisko
                if(aktualny->prawy != NULL) aktualny = aktualny->prawy;
                else {
                    aktualny->prawy = dane;
                    return true;
                }
        }
        return false;
    }
}

abonent *szukajAbonenta(abonent *korzen, string nazwisko, unsigned long *kroki)
{
    unsigned long i = 0;
    abonent *aktualny = korzen;
    while(aktualny != NULL) {
        i++;
        if(aktualny->nazwisko == nazwisko) {
            if(kroki != NULL) *kroki = i;
```

```
            return aktualny;
        }
        if(nazwisko < aktualny->nazwisko) aktualny = aktualny->lewy;
            else aktualny = aktualny->prawy;
    }
    if(kroki != NULL) *kroki = i;
    return NULL;
}

void wyswietlDrzewo(abonent *korzen)
{
    if(korzen == NULL) cout << "Puste drzewo wyszukiwania" << endl;
    else {
        if(korzen->lewy != NULL) wyswietlDrzewo(korzen->lewy);
        wypiszAbonenta(korzen); cout << endl;
        if(korzen->prawy != NULL) wyswietlDrzewo(korzen->prawy);
    }
}

void usunDrzewo(abonent **korzen)
{
    if(korzen != NULL && *korzen != NULL) {
        // rekurencyjnie usun oba poddrzewa
        usunDrzewo( &(*korzen)->lewy );
        usunDrzewo( &(*korzen)->prawy );
        // Usun aktualny wierzcholek
        delete *korzen;
        *korzen = NULL;
    }
}

abonent *wczytajAbonenta()
{
    abonent *nowy;
    try {
        nowy = new abonent;
    }
    catch(bad_alloc) {
        return NULL;
    }

    cout << "Podaj imie: ";
    getline(cin, nowy->imie);
    cout << "Podaj nazwisko: ";
    getline(cin, nowy->nazwisko);
    cout << "Podaj numer: ";
    getline(cin, nowy->numer);
    nowy->lewy = nowy->prawy = NULL;
    return nowy;
}

void wypiszAbonenta(abonent *co)
{
    if(co != NULL) {
        cout << "imie: " << co->imie << endl;
        cout << "nazwisko: " << co->nazwisko << endl;
        cout << "numer: " << co->numer << endl;
    }
}
```