**SPRAWOZDANIE**

|  |  |  |  |
| --- | --- | --- | --- |
| **Przedmiot** | Algorytmy i struktury danych | **Zadanie** | 8\_01 |
| **Autor** | Kamil Borkowski | **Grupa** | WCY22IY1S1 |
| **Temat** | B-Drzewo | | |

1. Treść

Napisz program, który umożliwi: Budowanie B-drzewa

Program powinien:

Umożliwić wczytywanie danych z pliku wejściowego

Umożliwić wprowadzanie danych z klawiatury

Dodawanie węzła

Kasowanie węzła

Szukanie węzła (ścieżka)

Kasowanie danych z całego drzewa

Narysowanie/ wyświetlenie drzewa (B-drzewa)

Wypisanie węzłów drzewa w kolejności przechodzenia

LVR – in-order, przejście poprzeczne, infiksowe

* 1. Metoda realizacji

Program wykorzystuje strukturę BdrzewoW, która reprezentuje węzeł drzewa. Operacje takie jak dodawanie, usuwanie, wyszukiwanie oraz są realizowane w funkcjach pomocniczych. Interfejs użytkownika umożliwia wykonywanie tych operacji poprzez wprowadzanie numerów operacji. Dodatkowo, program obsługuje wczytywanie danych z pliku i zwalnianie pamięci po zakończeniu działania.

* 1. Założenia / ograniczenia dotyczące danych:
     1. Dane wejściowe *(opis)*

Wartości węzłów – wczytywana z pliku/klawiatury

* + 1. Dane wyjściowe *(opis)*

Graficzne przedstawienie drzewa – wyświetlone na ekranie i w pliku .txt

Przejscie po węzłach metodami LVR – wyświetlone na ekranie i w pliku .txt

1. Realizacja
   1. Algorytm *(lista kroków lub schemat blokowy)*

1. Utworzenie struktury BdrzewoW.

2. Ustawienie początkowego korzenia (root) na NULL.

3. Implementacja funkcji DodajW do dodania węzła do drzewa.

4. Implementacja funkcji podzielPotomka do podzielenia węzła z drzewa.

5. Implementacja funkcji wstawNpW do wstawienia klucaz do niepelnego wezla.

6. Implementacja funkcji zlaczW do łącznia węzłów.

7. Implementacja funkcji odPoprzednika i odNastepnika do przenoszaca wartosci wezlow.

8. Implementacja funkcji najL i najP do szukania węzłów w poddrzewach.

9. Implementacja funkcji znajdzKlucz do szukania klucza.

10. Implementacja funkcji removeFromNonLeaf do usuwania klucza z węzła który nie jest liściem.

11. Implementacja funkcji uzupelnij do wypełniania luki w korzeniu.

12. Implementacja funkcji usunW do usuwania węzłów.

13. Implementacja funkcji wezlyLVR do wypisania wierzchołków w kolejności LVR.

14. Implementacja funkcji zwolnijDrzewo do zwalniania pamięci.

15. Implementacja funkcji drukujDrzewo do wizualizacji struktury drzewa.

16. Wyświetlenie prostego menu interakcji z użytkownikiem.

17. Oczekiwanie na wprowadzenie numeru operacji przez użytkownika.

18. Program utrzymuje pętlę, oczekując na kolejne operacje użytkownika.

19. Po zakończeniu działania programu zwalniana jest pamięć przy użyciu funkcji zwolnijDrzewo.

Kod źródłowy

//================================================

//Zadanie 8\_01 jezyk C++

//B-drzewo

//WCY22IY1S1 Borkowski Kamil

//================================================

#include <iostream>

#include <string>

#include <cstdlib>

#include <fstream>

using namespace std;

const int T = 2;//stala pomocnicza dla stopnia drzewa

//struktura B-drzewa 4 stopnia

typedef struct BdrzewoW {

int klucze[2\*T-1];

struct BdrzewoW \*potomek[2\*T];

int n;

int leaf;

} BdrzewoW;

//funkcja towrzaca nowe B-drzewo

BdrzewoW \*nowyW(int leaf) {

BdrzewoW \*node = (BdrzewoW \*) malloc(sizeof(BdrzewoW));

node->leaf = leaf;

node->n = 0;

return node;

}

//funkcja pomocnicza, dzielaca potomka

void podzielPotomka(BdrzewoW \*rodzic, int i, BdrzewoW \*child) {

BdrzewoW \*nowyPotomek = nowyW(child->leaf);//tworzymy nowego potomka

nowyPotomek->n = T - 1;

//druga polowe kluczy wstawiamy do nowego potomka

for (int j = 0; j < T - 1; j++) {

nowyPotomek->klucze[j] = child->klucze[j + T];

}

//jezeli potomek nie jest lisciem to kopiujemy do nowego potomka druga polowe potomkow

if (!child->leaf) {

for (int j = 0; j < T; j++) {

nowyPotomek->potomek[j] = child->potomek[j + T];

}

}

child->n = T - 1;//aktualizujemy ilosc kluczy potomka

//robimy miejsce na potomka

for (int j = rodzic->n; j >= i + 1; j--) {

rodzic->potomek[j + 1] = rodzic->potomek[j];

}

rodzic->potomek[i + 1] = nowyPotomek;

for (int j = rodzic->n - 1; j >= i; j--) {

rodzic->klucze[j + 1] = rodzic->klucze[j];

}

rodzic->klucze[i] = child->klucze[T - 1];

rodzic->n = rodzic->n + 1;//aktualizujemy ilosc kluczy rodzica

}

//funkcja wstawiajaca klucz do niepelnego wezla

void wstawNpW(BdrzewoW \*node, int key) {

int i = node->n - 1;

if (node->leaf) {

while (i >= 0 && node->klucze[i] > key) {

node->klucze[i + 1] = node->klucze[i];

i--;

}

node->klucze[i + 1] = key;

node->n = node->n + 1;

} else {

while (i >= 0 && node->klucze[i] > key) {

i--;

}

if (node->potomek[i + 1]->n == 2 \* T - 1) {

podzielPotomka(node, i + 1, node->potomek[i + 1]);

if (key > node->klucze[i + 1]) {

i++;

}

}

wstawNpW(node->potomek[i + 1], key);

}

}

//funkcja wstawiajaca nowy wezel

BdrzewoW \*DodajW(BdrzewoW \*root, int key) {

if (root == NULL) {

root = nowyW(1);

root->klucze[0] = key;

root->n = 1;

return root;

}

if (root->n == 2 \* T - 1) {

BdrzewoW \*newRoot = nowyW(0);

newRoot->potomek[0] = root;

podzielPotomka(newRoot, 0, root);

int i = 0;

if (newRoot->klucze[0] < key) {

i++;

}

wstawNpW(newRoot->potomek[i], key);

return newRoot;

} else {

wstawNpW(root, key);

}

return root;

}

//funkcja wyswietlajaca drzewo

void drukujDrzewo(BdrzewoW \*node) {

//wypisanie kluczy aktualnego wezla

printf("{ ");

for (int i = 0; i < node->n; i++) {

cout<<node->klucze[i]<<" ";

}

cout<<"} ";

//wywolanie tej funkcji dla potomkow

if (!node->leaf) {

cout<<endl;

for (int i = 0; i <= node->n; i++) {

drukujDrzewo(node->potomek[i]);

}

}

}

//funkcja laczaca wezly

BdrzewoW \*zlaczW(BdrzewoW \*root, int idx) {

BdrzewoW \*child = root->potomek[idx];

BdrzewoW \*sibling = root->potomek[idx + 1];

//przenosimy klucz z rodzica do potomka

child->klucze[T - 1] = root->klucze[idx];

//kopiowanie kluczy do potomka

for (int i = 0; i < sibling->n; ++i)

child->klucze[i + T] = sibling->klucze[i];

//kopiowanie wskaznikow do potomka

if (!child->leaf) {

for (int i = 0; i <= sibling->n; ++i)

child->potomek[i + T] = sibling->potomek[i];

}

for (int i = idx + 1; i < root->n; ++i)

root->klucze[i - 1] = root->klucze[i];

for (int i = idx + 2; i <= root->n; ++i)

root->potomek[i - 1] = root->potomek[i];

//aktualizacja ilosci kluczy korzenia i potomka

child->n += sibling->n + 1;

root->n--;

//zwalnianie pamieci po niepotrzebnym wezle

free(sibling);

return root;

}

//funkcja przenoszaca wartosci wezlow

BdrzewoW \*odPoprzednika(BdrzewoW \*root, int idx) {

BdrzewoW \*child = root->potomek[idx];

BdrzewoW \*sibling = root->potomek[idx - 1];

//przenoszenie kluczy potomka

for (int i = child->n - 1; i >= 0; --i)

child->klucze[i + 1] = child->klucze[i];

//przenoszenie wskaznikow potomka

if (!child->leaf) {

for (int i = child->n; i >= 0; --i)

child->potomek[i + 1] = child->potomek[i];

}

//przenoszenie kluczy z rodzica do potomka

child->klucze[0] = root->klucze[idx - 1];

if (!sibling->leaf)

child->potomek[0] = sibling->potomek[sibling->n];

//przenoszenie ostatniego klucza z potomka

root->klucze[idx - 1] = sibling->klucze[sibling->n - 1];

//aktualizacja ilosci kluczy potomkow

child->n++;

sibling->n--;

return root;

}

//funkcja przenoszaca wartosci wezlow

BdrzewoW \*odNastepnika(BdrzewoW \*root, int idx) {

BdrzewoW \*child = root->potomek[idx];

BdrzewoW \*sibling = root->potomek[idx + 1];

//przenoszenie klucza z rodzica do dziecka

child->klucze[child->n] = root->klucze[idx];

//przypisujemy pierwsze dziecko rodzica, jako ostatnie dziecko dziecka

if (!(child->leaf))

child->potomek[child->n + 1] = sibling->potomek[0];

//przenoszenie pierwszego klucza rodzenstwa do rodzica

root->klucze[idx] = sibling->klucze[0];

//przenoszenie wszytkich kluczy w rodzenstwie o jeden w lewo

for (int i = 1; i < sibling->n; ++i)

sibling->klucze[i - 1] = sibling->klucze[i];

//przenoszenie wskaznikow dziecka

if (!sibling->leaf) {

for (int i = 1; i <= sibling->n; ++i)

sibling->potomek[i - 1] = sibling->potomek[i];

}

//aktualizacja ilosci kluczy potomkow

child->n++;

sibling->n--;

return root;

}

//funkcja szukajaca najwiekszego klucza w lewym poddrzewie

int najL(BdrzewoW \*node, int idx) {

BdrzewoW \*cur = node->potomek[idx];

while (!cur->leaf)

cur = cur->potomek[cur->n];

return cur->klucze[cur->n - 1];

}

//funkcja szukajaca najmniejszego klucza w prawym poddrzewie

int najP(BdrzewoW \*node, int idx) {

BdrzewoW \*cur = node->potomek[idx + 1];

while (!cur->leaf)

cur = cur->potomek[0];

return cur->klucze[0];

}

//funkcja szukajaca danego klucza

int znajdzKlucz(BdrzewoW \*node, int k) {

int idx = 0;

while (idx < node->n && node->klucze[idx] < k) {

++idx;

}

return idx;

}

//zapowiedz funkcji usuwajacej wezel

void usunW(BdrzewoW \*root, int k);

void removeFromLeaf(BdrzewoW \*node, int idx) {

for (int i = idx + 1; i < node->n; ++i)

node->klucze[i - 1] = node->klucze[i];

node->n--;

}

//funkcja usuwajaca wezel z wezla ktory nie jest lisciem

void removeFromNonLeaf(BdrzewoW \*node, int idx) {

int k = node->klucze[idx];

//warunek jesli lewe poddrzewo ma co najmniej T kluczy

if (node->potomek[idx]->n >= T) {

int pred = najL(node, idx);

node->klucze[idx] = pred;

usunW(node->potomek[idx], pred);

}

//warunek jesli prawe poddrzewo ma co najmniej T kluczy

else if (node->potomek[idx + 1]->n >= T) {

int succ = najP(node, idx);

node->klucze[idx] = succ;

usunW(node->potomek[idx + 1], succ);

}

//jesli i prawe i lewe poddrzewo maja T kluczy to je laczymy

else {

zlaczW(node, idx);

usunW(node->potomek[idx], k);

}

}

//funkcja wypelniajaca luke w korzeniu

void uzupelnij(BdrzewoW \*root, int idx) {

//jesli lewy potomek ma wiecej kluczy zabierz jeden

if (idx != 0 && root->potomek[idx - 1]->n >= T)

odPoprzednika(root, idx);

//jesli prawy potomek ma wiecej kluczy zabierz jeden

else if (idx != root->n && root->potomek[idx + 1]->n >= T)

odNastepnika(root, idx);

//jesli nie to polacz ostaniego potomka z jego rodzicem

else {

if (idx != root->n)

zlaczW(root, idx);

else

zlaczW(root, idx - 1);

}

}

//funkjca usuwajaca wezel

void usunW(BdrzewoW \*root, int k) {

if (!root)

return;

int idx = znajdzKlucz(root, k);

//sprawdzenie czy klucz jest w drzewie

if (idx < root->n && root->klucze[idx] == k) {

if (root->leaf)

removeFromLeaf(root, idx);

else

removeFromNonLeaf(root, idx);

} else {

if (root->leaf) {

cout<<"Nie znaleziono danego klucza"<<endl;

return;

}

// Is the key in the subtree of the last child of this node?

bool flag = ((idx == root->n) ? true : false);

//zapelniamy klucze dziecka ktore ma ich mniej niz T

if (root->potomek[idx]->n < T)

uzupelnij(root, idx);

//usuwamy polaczonego juz potomka

if (flag && idx > root->n)

usunW(root->potomek[idx - 1], k);

else

usunW(root->potomek[idx], k);

}

//jesli korzen ma 0 kluczy to jego pierwszy potomek jest nowym korzeniem

if (root->n == 0) {

BdrzewoW \*tmp = root;

if (root->leaf)

root = NULL;

else

root = root->potomek[0];

free(tmp);

}

}

//funkcja drukujaca sciezke do danego wezla

bool sciezkaDoW(BdrzewoW \*node, int key, int sciezka[], int level) {

if (node == NULL) return false;

int i;

for (i = 0; i < node->n; i++) {

sciezka[level] = node->klucze[i]; //dodaj klucz do sciezki

if (key == node->klucze[i]) { //jesli klucz znaleziony

//drukowanie sciezki

for (int j = 0; j <= level; j++) {

if (key != sciezka[j]) {

cout<<sciezka[j]<<'\t';

} else {

cout<<sciezka[j]<<" ";

}

}

cout<<endl;

return true;//klucz odnaleziony

}

if (key < node->klucze[i]) break;//koniec petli gdy klucz jest mniejszy od aktualnego

}

if (node->leaf) return false;//jesli jest lisciem to klucza nie ma w drzewie

//szukanie w potomku

bool found = sciezkaDoW(node->potomek[i], key, sciezka, level + 1);

return found;

}

//funkcja szukajaca sciezki do wezla

void drukujSciezke(BdrzewoW \*root, int key) {

int sciezka[1000];

if (!sciezkaDoW(root, key, sciezka, 0)) {

cout<<"Klucz nie zostal odnaleziony"<<endl;

}

}

//funcja wypisujaca LVR

void wezlyLVR(BdrzewoW \*root) {

if (root == NULL) return;

int i;

for (i = 0; i < root->n; i++) {

//sprawdzanie lewego podrzewa

if (!root->leaf) {

wezlyLVR(root->potomek[i]);

}

cout<<root->klucze[i]<<'\t';

}

//sprawdzanie prawego podrzewa

if (!root->leaf) {

wezlyLVR(root->potomek[i]);

}

}

//funkjca zwalniajaca pamiec

void zwolnijDrzewo(BdrzewoW \*node) {

if (node == NULL) return;

if (!node->leaf) {

for (int i = 0; i <= node->n; i++) {

zwolnijDrzewo(node->potomek[i]);

}

}

free(node);

}

int main() {

BdrzewoW \*root = NULL;

int pom=1,n;//pom oblusguje interfejs, n= nr komendy z interfejsu

cout << "WCY22IY1S1 Borkowski Kamil" << endl;

cout << "Program do zarzadzania B-drzewem" << endl;

cout << "[0] - wyjscie z programu" << endl;

cout << "[1] - wczytaj wezly z pliku(we.txt)" << endl;

cout << "[2] - dodaj wezel przez klawiature" << endl;

cout << "[3] - usuwanie wezla o podanym kluczu" << endl;

cout << "[4] - wyswietlenie sciezki podanego wezla" << endl;

cout << "[5] - wyswietlenie drzewa" << endl;

cout << "[6] - przeglad wierzcholkow" << endl;

cout << "=========================================" << endl;

while(pom){

cout << "Wprowadz nr operacji: ";

cin >> n;

switch (n){

case 0:

cout<<"Wychodzenie..."<<endl;

pom = 0;

break;

case 1:

{

ifstream in;

in.open("we.txt");

int p;

while(in>>p)

{

root = DodajW(root, p);

}

cout<<"Drzewo zostalo usupelnione wezlami z pliku we.txt"<<endl<<endl;

in.close();

break;

}

case 2:

{

int k=0;//gdy wywola sie ponownie ta komende trzeba zmienic wartosc k z -1

cout<<"Podaj wezel(liczba,-1 konczy dodawanie wezlow)"<<endl;

while(k!=-1)

{

cin>>k;

if(k!=-1)

{

root = DodajW(root, k);

}

}

break;

}

case 3:

{

cout<<"Wskaz klucz wezla do usuniecia"<<endl;

int k;

cin>>k;

usunW(root, k);

cout<<"Usuwanie wskazanego wezla..."<<endl<<endl;

break;

}

case 4:

{

int k;

cout << "Podaj klucz wezla do wyszukania: ";

cin >> k;

cout << "Sciezka: ";

drukujSciezke(root, k);

break;

}

case 5:

{

if(root==NULL)

{

cout<<"Drzewo jest puste"<<endl<<endl;//sprawdzenie, czy drzewo jest puste

break;

}

drukujDrzewo(root); //wyswietlenie B-drzewa

cout<<endl;

break;

}

case 6:

{

cout << "Wezly w kolejnosci LVR: " << endl;

wezlyLVR(root);

cout<<endl;

break;

}

default:

{

cout<<"Podano bledny nr operacji"<<endl<<endl;

break;

}

}

}

zwolnijDrzewo(root);

return 0;

}

* 1. Dane wejściowe *(ekran uruchomienia programu)*

Obraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznie

* 1. Dane wyjściowe *(ekran uruchomienia programu)*

Obraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznie

1. Wnioski *(złożoność asymptotyczna algorytmu przy użyciu notacji* ***O lub innej i inne wnioski)***

Złożoność obliczeniowa programu opiera się na operacjach wykonywanych na B-drzewie. Złożoność zależy od operacji, ale w przypadku operacji wstawiania, usuwania i wyszukiwania w B-drzewie o stopniu

T, gdzie T to stała pomocnicza, złożoność wynosi O(logn), gdzie n to liczba kluczy w drzewie.