POLITECHNIKA
RZESZOWSKA
im. IGNACEGO ŁUKASIEWICZA

Karolina Magdoń

Inżynieria i analiza danych
I rok

Sprawozdanie z projektu implementacji struktury drzewa binarnego (C++)

Praca przygotowana na zajęcia
„Algorytmy i struktury danych”
w roku akademickim 2022/2023

Rzeszów, styczeń 2023

Spis treści

1. Wstęp i opis zagadnień projektu

Zadanie projektowe polegało na implementacji struktury danych typu drzewo binarne. Implementacja miała zawierać potrzebne operacje charakterystyczne dla tej struktury. Określone one zostały jako kolejno: inicjowanie struktury, dodawanie i usuwanie elementów, wyświetlanie elementów, zliczanie elementów, sprawdzanie głębokości drzewa i wyszukiwanie czy element się w nim znajduje. Do kolejnych zaleceń należało: przedstawienie podstaw teoretycznych zagadnienia oraz pseudokod algorytmu (również schemat blokowy). Zobrazować rezultaty działania programu. Ukazać złożoność czasową, obliczeniową.

2. Podstawy teoretyczne zagadnienia

Drzewo to popularna struktura danych, która ma charakter nieliniowy. W przeciwieństwie do innych struktur danych, takich jak tablica, stos, kolejka i połączona lista, które mają charakter liniowy, drzewo reprezentuje strukturę hierarchiczną. Informacje o kolejności drzewa nie są ważne. Drzewo zawiera węzły i 2 wskaźniki. Te dwa wskaźniki to lewe dziecko i prawe dziecko węzła nadrzędnego. Zapoznajmy się szczegółowo z warunkami drzewa.

-*Korzeń*: Korzeń drzewa to najwyższy węzeł drzewa, który nie ma węzła nadrzędnego. W każdym drzewie jest tylko jeden węzeł główny.

-*Węzeł nadrzędny*: Węzeł, który jest poprzednikiem węzła, nazywany jest węzłem nadrzędnym tego węzła.

-*Węzeł potomny*: Węzeł, który jest bezpośrednim następcą węzła, nazywany jest węzłem potomnym tego węzła.

-*Rodzeństwo*: dzieci tego samego węzła nadrzędnego nazywane są rodzeństwem.

-*Krawędź*: Krawędź działa jako łącze między węzłem nadrzędnym a węzłem podrzędnym.

-*Liść*: Węzeł, który nie ma potomka, jest nazywany węzłem liścia. To ostatni węzeł drzewa. W drzewie może być wiele węzłów liścia.

-*Poddrzewo*: Poddrzewo węzła to drzewo, które traktuje ten konkretny węzeł jako węzeł główny.

-*Głębokość*: Głębokość węzła to odległość od węzła głównego do tego konkretnego węzła.

-*Wysokość*: Wysokość węzła to odległość od tego węzła do najgłębszego węzła tego poddrzewa.

-*Wysokość drzewa*: Wysokość drzewa to maksymalna wysokość dowolnego węzła. Jest to to samo, co wysokość węzła głównego.

3. Cechy programu

1. Definicja struktury drzewo z miejscem na dane typu int i dwojga dzieci.

2. Dodawanie i usuwanie węzłów z zainicjowanego drzewa.

3. Odczytywanie drzewa.

4. Zliczanie ilości węzłów.

5. Liczenie ilości poziomów drzewa.

6. Szukanie czy element znajduje się w drzewie.

Do wykonania tego projektu stworzyłam kod w języku C++ zawierający:

- deklaracje struktury drzewa binarnego,
- konstruktor dla drzewa binarnego,
- funkcje dodawania do drzewa binarnego,
- funkcje odczytywania i zliczania ilości węzłów,
- funkcje sprawdzania ilości poziomów drzewa,
- funkcje szukania czy element znajduje się w drzewie,
- testy funkcjonalności.

4. Opisy funkcji

4.1. Dodawanie

Funkcja dodawanie szuka pierwszego wolnego miejsca, gdzie nie ma już węzła. Jeśli w obecnym węźle nie ma miejsca, to funkcja wchodzi głębiej i tam ponownie próbuje go stworzyć. W tym celu do zmiennej, która określa przeszukiwany węzeł wpisujemy ten już głębszy. W praktyce jest to użycie podejścia preorder traversal.

4.2. Odczytywanie i zliczanie

Poczynając od węzła podanego jako argument do funkcji sprawdzamy, czy nie jest on pusty, jeśli jest, to zwracamy wartość poprzedniej ilości. W ten sposób możemy wywoływać funkcję rekurencyjnie i zawsze będziemy tyko dodawać 1, jeśli węzeł nie jest pusty. W ten sposób szybko liczymy ile węzłów ma drzewo, a za razem pozwalamy funkcji wykonywać się coraz głębiej i szukać kolejnych węzłów.

4.3. Ilość poziomów

Podobnie jak w innych funkcjach rekurencyjnie wywołujemy program, który będzie nadpisywał swoją wartość i schodził tak głęboko w drzewo, jak się da. Jako, że idziemy w dwóch kierunkach, to wybieramy wyższą wartość.

4.4. Szukanie

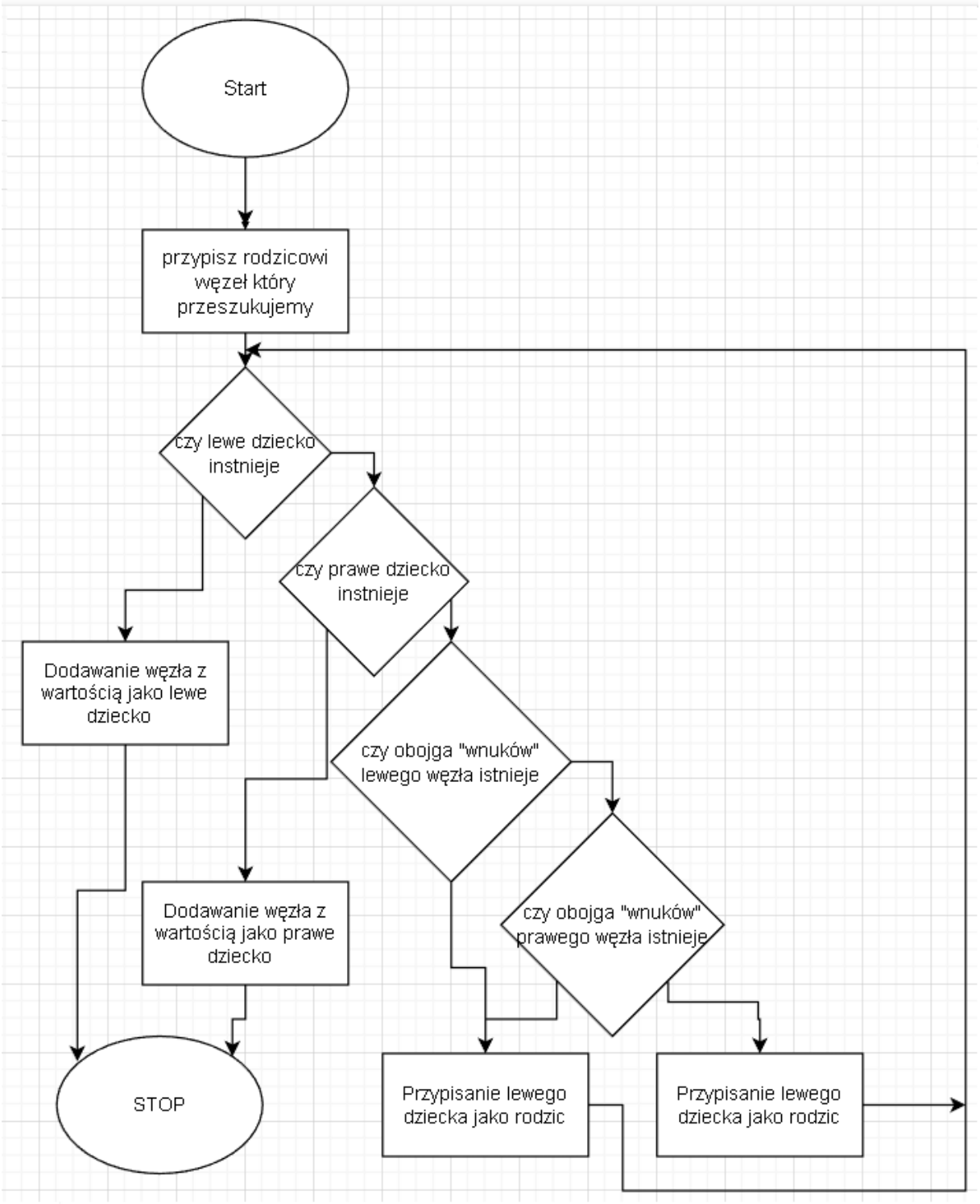
Ponownie funkcja rekurencyjna, sprawdza czy odwiedzany węzeł ma wartość zgodną szukanej, jeśli nie, to schodzi głębiej, jeśli na maksymalnej głębokości nie ma wartości, to zwrócone zostanie 0. Dlatego wartość dla znalezienia równa się 1, by zawsze zdominowała to, że na innej gałęzi nie znaleziono osoby.

5. Złożoność obliczeniowa

Złożoność obliczeniowa algorytmu jest bardzo wydajna, w każdym wypadku wynosi O(n), gdyż wystarczy, że tylko 2 razy przeanalizujemy wszystkie liczby z tabeli, aby wyznaczyć wynik.

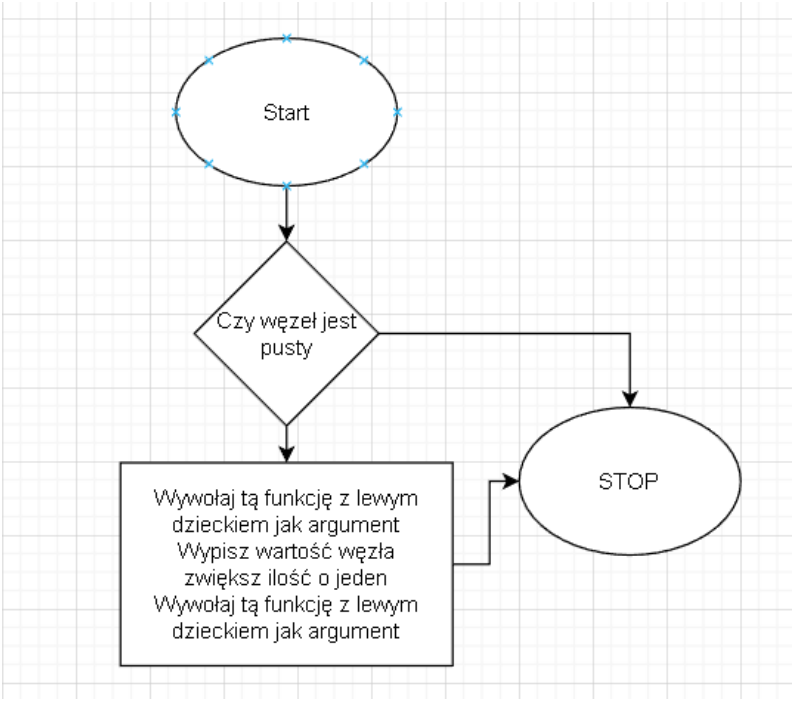
6. Schemat blokowy

6.1. Dodawanie



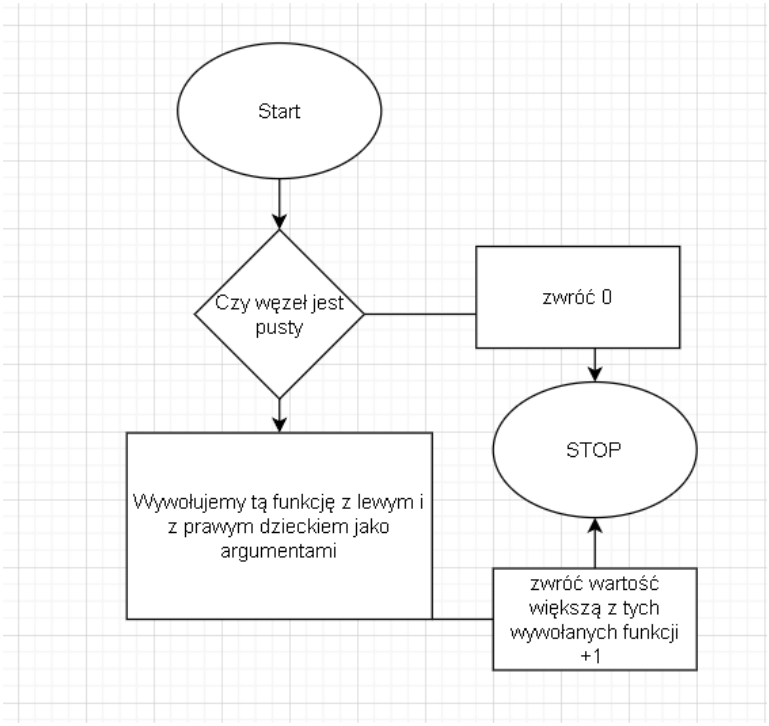
Rys.1. Schemat blokowy funkcji dodawania

6.2. Odczytywanie i zliczanie



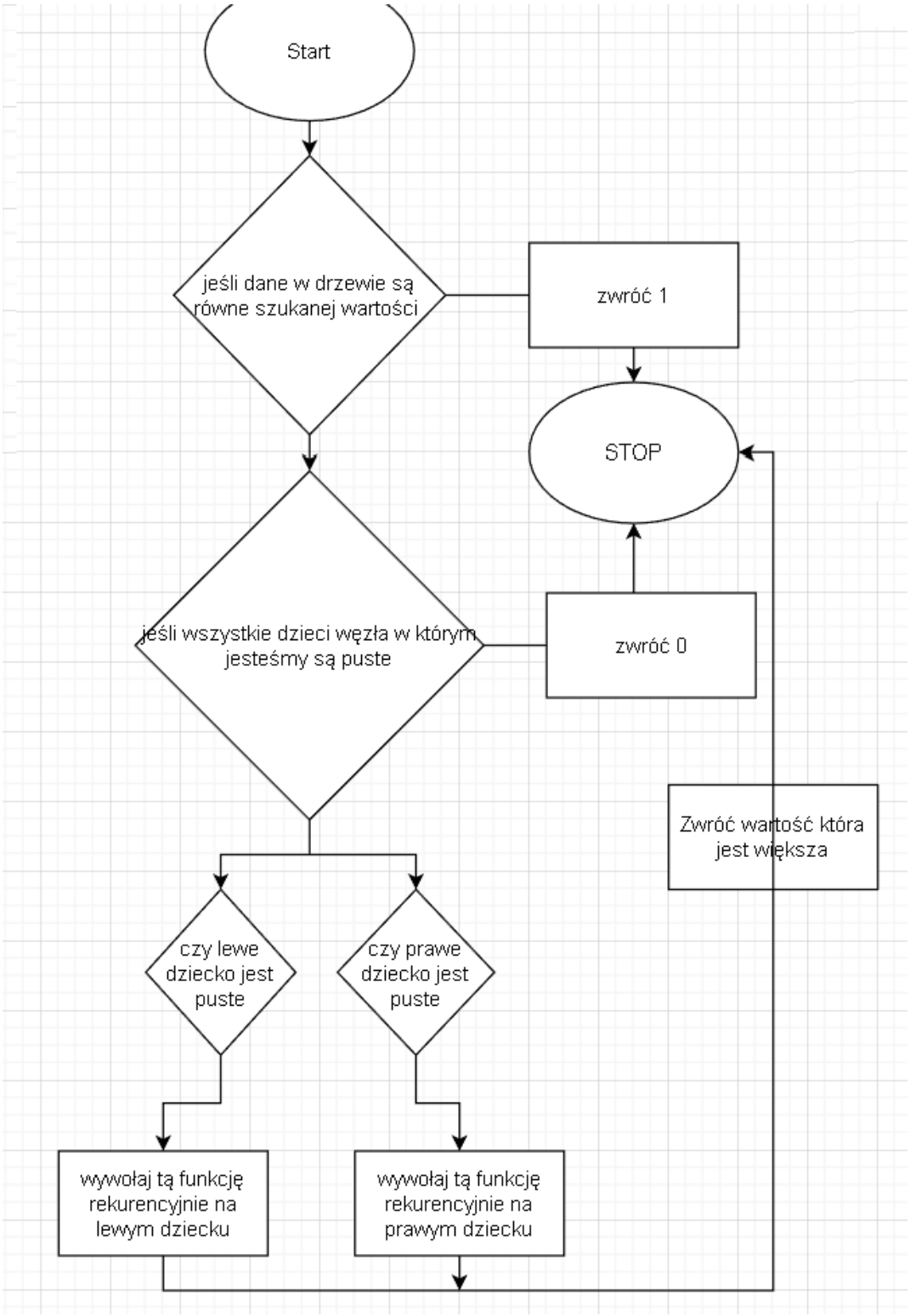
Rys.2. Schemat blokowy funkcji odczytywania i zliczania

6.3. Ilość poziomów



Rys.3. Schemat blokowy funkcji ilości poziomów

6.4. Szukanie



Rys.4. Schemat blokowy funkcji szukania

7. Pseudokod

7.1.

K01: Wczytaj drzewo, do którego dodajemy

K02: Ustaw wartość zmiennej wskazującej rodzica na wczytane drzewo:

K03: Powtarzaj kroki: od K04 do K012

K04: Jeśli lewe dziecko jest zapełnione przejdź do kroku K05, jeśli nie, K011

K05: Jeśli prawe dziecko jest zapełnione przejdź do kroku K06, jeśli nie, K010

K06: Jeśli „wnuki" lewego węzła są zapełnione przejdź do kroku K07, jeśli nie, K08

K07: Jeśli „wnuki" prawego węzła są zapełnione przejdź do kroku K08, jeśli nie, K09

K08: Ustaw wartość zmiennej wskazującej rodzica na lewy węzeł, przejdź do kroku K012

K09: Ustaw wartość zmiennej wskazującej rodzica na prawy węzeł, przejdź do kroku K012

K10: Dodajemy po prawej stronie kolejny węzeł z wartością, którą mieliśmy wstawić, przejdź do kroku K013

K11: Dodajemy po lewej stronie kolejny węzeł z wartością, którą mieliśmy wstawić, przejdź do kroku K013

K12: Wróć do kroku K04

K13: Koniec

7.2.

K01: Wczytaj drzewo

K02: Jeśli węzeł jest pusty to zwróć już naliczoną ilość węzłów bez zmian

K03: Rekurencyjnie odpal tą funkcję dla lewej gałęzi i jej wynik przypisz jako naliczona ilość elementów

K04: Wyświetl wartość dla obecnego węzła

K05: Rekurencyjnie odpal tą funkcję dla prawej gałęzi i jej wynik przypisz jako naliczona ilość elementów

K06: Zwróć naliczoną ilość

7.3.

K01: Wczytaj drzewo
K02: Jeśli węzeł jest pusty to zwróć 0
K03: Stwórz dwie zmienne pomocnicze
K04: Bierzemy dla nich wartości z odpalanych rekurencyjnie funkcji dla lewych i prawych gałęzi
K05: Zwróć większą z dwóch zmiennych +1

7.4.

K01: Wczytaj drzewo
K02: Stwórz dwie zmienne pomocnicze
K03: Jeśli dane z danego węzła są równe szukanej wartości zwróć 1, inaczej kontynuuj
K04: Jeśli i lewe i prawe dziecko jest puste, zwróć 0
K05: Jeśli istnieją węzły lewe lub prawe dziecko, wywołaj na danych z nich funkcję szukanie
K06: Zwróć większą z dwóch zmiennych

8. Dane we/wy i zmienne pomocnicze

8.1.

- Dane wejściowe: Drzewo binarne, dodawana wartość
- Dane wyjściowe: Dodany węzeł;

8.2.

- Dane wejściowe: Drzewo binarne, zmienna do zapisywania ilości;
- Dane wyjściowe: Wyświetlanie tablicy, ilość węzłów;

8.3.

- Dane wejściowe: Drzewo binarne;
- Dane wyjściowe: Ilość poziomów;

- Zmienne pomocnicze: wysokości podgałęzi wychodzącej z lewego węzła i analogicznie dla prawego

8.4.

- Dane wejściowe: Drzewo binarne, szukana wartość;
- Dane wyjściowe: Wiadomość czy udało się zaleźć liczbę;
- Zmienne pomocnicze: Użyta do zwrotów by zapisać wyniki.

9. Podsumowanie i wnioski

Stworzono w pełni funkcjonalną strukturę drzewa binarnego. Funkcje do jej obsługiwania także działają bez problemu. Jest to ciekawa struktura, która pozwala na specyficzny sposób dostania się do danych. Jak widać, prawie każda funkcja działa rekurencyjnie, przypisując coraz głębszych kandydatów na rodzica. Oczywiście jest dużo różnych przykładów rozwiązania tego. Ja skorzystałam z wysokiej modularności dużej ilości małych prostych w obsłudze elementów i wykonywałam tę samą funkcję wielokrotnie, tylko kończąc gdzieś głęboko, jak już osiągane było, co miało być i wtedy wartości były przekazywane coraz wyżej, aż rozplątały się wszystkie wywołania funkcji. Uważam to za efektywne korzystanie z tej struktury i widać, że ma ona swoje unikalne zastosowania.

Podsumowując:

1. Program odczytuje dane liczbowe z pliku i zapisuje do tablicy.
2. Program zapisuje wynik do pliku.
3. Wykonano testy sprawdzające działanie algorytmu.
4. Kod opatrzono komentarzami.
5. Sporządzono schematy blokowe oraz pseudokody.

10. Appendix: kod program

```
#include <iostream>

#include <fstream>

using namespace std;


 fstream zapisywanie; //Zapisywanie

struct drzewo_binarne //Deklaracja struktury drzewa binarnego

{

   int dane; //Zmienna do wartosci danego wezla

   struct drzewo_binarne* lewe_dziecko; //Deklarowanie dla danego wezla lewego dziecka tego samego typu struktury

   struct drzewo_binarne* prawe_dziecko;//Deklarowanie dla danego wezla prawego dziecka tego samego typu struktury

   int wypisz() //Funkcja do wypisywania danych z wezla

   {

      return dane; //Zwraca dane

   }

};

struct drzewo_binarne* nowe_drzewo_binarne(int dane_wejsciowe) //Kostruktor dla drzewa, przyjmuje int

{

   struct drzewo_binarne* drzewo_binarne = new struct drzewo_binarne; //Inicjacja nowego drzewa

   drzewo_binarne->dane = dane_wejsciowe; //Dane przyjmuje wartosc podana podczas tworzenia instancji

   drzewo_binarne->lewe_dziecko = NULL; //Miejsca na dzieci zaczynaja puste, potem moga zostac wypelnione

   drzewo_binarne->prawe_dziecko = NULL;

   return drzewo_binarne; //Zwrot stworzonego drzewa

};

void dodawanie(drzewo_binarne* drzewo_do_ktorego_dodajemy, int dodwana_wartosc) //Funkcja dodawnia, przyjmuje
drzewo do ktorego mamy dodac wartosc i wartosc

{

  drzewo_binarne *rodzic = drzewo_do_ktorego_dodajemy; //Deklaracja zmiennej rodzic która bedzie wskazywala jakiego
wezla dzieci badamy pod katem mienia wolnego miejsca na dane. Pozniej bedzie Ona nadpisywana by schodzic coraz glebiej
w drzewo

 while(true){ //While dziala bez warunku tylko bedzie sie konczyl przy break'ach

   if (rodzic->lewe_dziecko != NULL){ //Jesli lewe dziecko jest zapelnione

      if (rodzic->prawe_dziecko != NULL){ //Jesli prawe dziecko jest zapelnione

         if (rodzic->lewe_dziecko->lewe_dziecko != NULL && rodzic->lewe_dziecko->prawe_dziecko != NULL){ //Sprawdzanie
czy obojga dzieci danego wezla tez jest zajete
```

```
            if (rodzic->prawe_dziecko->lewe_dziecko != NULL && rodzic->prawe_dziecko->prawe_dziecko != NULL){
//Sprawdzanie czy obojga dzieci danego wezla tez jest zajete

                rodzic = rodzic->lewe_dziecko; cout<<"idziemy w lewo"<<endl; //Jesli idac od rodzica oboje jego dzieci jest pelne,
a takze ich dzieci, to przyjmujemy lewe dziecko jako rodzica

            }else{

                rodzic = rodzic->prawe_dziecko; cout<<"idziemy w  prawo"<<endl; //Jesli idac od rodzica oboje jego dzieci a
takze dzieci idace od lewego dziecka sa pelne, to przyjmujemy prawe dziecko jako rodzica

            }

          }else{

            rodzic = rodzic->lewe_dziecko; cout<<"idziemy w  lewo"<<endl; //Jesli idac od rodzica oboje jego dzieci jest pelne,
ale dzieci lewego dziecka sa puste, to przyjmujemy lewe dziecko jako rodzica

          }

       }else{

       rodzic->prawe_dziecko = nowe_drzewo_binarne(dodwana_wartosc); cout<<"dodane po prawej"<<endl<<endl; break;
//Jesli prawe dziecko tworzymy tam nowa strukture z dodawana wartoscia, wychodzimy z while'a, co konczy funkcje

       }

    }else{

    rodzic->lewe_dziecko = nowe_drzewo_binarne(dodwana_wartosc); cout<<"dodane po lewej"<<endl<<endl; break; //Jesli
lewe dziecko tworzymy tam nowa strukture z dodawana wartoscia, wychodzimy z while'a, co konczy funkcje

    }

  }

}



int odczytywanie_i_zliczanie(drzewo_binarne* drzewo_do_odczytania, int ilosc){ //Funkcja do odczytywania i zliczania ilosci
wezlow, przyjmuje



    if (drzewo_do_odczytania == NULL){return ilosc;}else{ilosc++;} //Jesli wezel ktory ma byc odczytany jest pusty, zwracamy
obecna ilosc wezlow (warto zaznaczyc ze przy rekurencyjnym wywolywaniu funkcji dostaniemy spowrotem wartosc
przekazana do funkcji)



    ilosc = odczytywanie_i_zliczanie(drzewo_do_odczytania->lewe_dziecko, ilosc); //Wywolanie rekurencyjne dla lewego
dziecka. W ten sposob bedzie funkcja idzie coraz glebiej i zawsze gdy znajduje pelny wezel dodaje do ilosci wezlow, a jesli
znajdzie pusty, to ta ilosc wroci sporotem do najwyzszego wywolania funkcji



     cout << drzewo_do_odczytania->dane << " "; //Wypisanie wartosci obecnego wezla  (jesli nie ma to funkcja nie dochodzi
do tego momentu bo juz wczesniej byl return)



    ilosc = odczytywanie_i_zliczanie(drzewo_do_odczytania->prawe_dziecko, ilosc); //Wywolanie rekurencyjne dla prawego
dziecka.



    return ilosc; //zwracamy ilosc by wydostac ja z funkcji dla korzenia
```

```
}

void zapisywanie_do_pliku(drzewo_binarne* drzewo_do_odczytania){ //Funkcja praktycznie identyczna do odczytywania ale nie zlicza i ma plik zamiast cout


   if (drzewo_do_odczytania == NULL){return;} //Jesli wezel ktory ma byc odczytany jest pusty, koniec funkcji

   zapisywanie_do_pliku(drzewo_do_odczytania->lewe_dziecko); //Wywolanie rekurencyjne dla lewego dziecka.


   zapisywanie << drzewo_do_odczytania->dane << " "; //Wypisanie wartosci obecnego wezla do pliku


   zapisywanie_do_pliku(drzewo_do_odczytania->prawe_dziecko); //Wywolanie rekurencyjne dla prawego dziecka.

}

int ile_poziomow(drzewo_binarne* drzewo_do_odczytania){ //Funkcja sprawdzajaca ile poziomow ma drzewo, przyjmuje drzewo


   if (drzewo_do_odczytania == NULL){return 0;} //Sprawdzamy czy wezel jest pusty, jesli tak zwracamy 0 bo nie dodajemy kolejnego poziomu do zliczania


   int lewe_dziecko_wysokosc,prawe_dziecko_wysokosc; //Tworzymy zmienne pomocnicze by moc porownywac ktora galaz idzie glebiej


   lewe_dziecko_wysokosc=ile_poziomow(drzewo_do_odczytania->lewe_dziecko); //Wywolanie funkcji rekurencyjnie na lewej galezi. Wartosc wywolania czyli ilosc poziomow obliczona z tej galezi zapisana do zmiennej

   prawe_dziecko_wysokosc=ile_poziomow(drzewo_do_odczytania->prawe_dziecko); //Wywolanie funkcji rekurencyjnie na prawej galezi.


   return max(lewe_dziecko_wysokosc,prawe_dziecko_wysokosc) + 1; //Zwracamy wartosc z tej galezi ktora byla glebsza +1 bo jeszcze liczymy poziom korzenia

}




int szukanie(drzewo_binarne* drzewo_do_odczytania, int szukana_wartosc){ //Funkcja szukanie pozwala znalezc czy jakas wartosc znajduje sie w drzewie, przyjmuje drzewo i szukana wartosc


   int zwrotlewy=0,zwrotprawy=0; //Definicja zmiennych pomocniczych potrzebnych do wiedzenia czy juz znaleziono liczbe


   if (drzewo_do_odczytania->dane == szukana_wartosc){ //Sprawdzamy czy wartosc danych w wezle jest rowna szukanej wartosci, jesli tak to zwracamy jedynke ktora wraca to pierwotnego wywolania funkcji

      return 1;

   }else{
```

```
    if (drzewo_do_odczytania->lewe_dziecko == NULL && drzewo_do_odczytania->prawe_dziecko == NULL){return 0;}
//Jesli kolejne galezie sa puste to zwracamy 0 i konczymy poszukiwania w tym kierunku

  }


  if (drzewo_do_odczytania->lewe_dziecko != NULL){ //Jesli lewe dziecko nie jest puste to wywolujemy rekurencyjnie na
nim szukanie

  zwrotlewy = szukanie(drzewo_do_odczytania->lewe_dziecko, szukana_wartosc);}


  if (drzewo_do_odczytania->prawe_dziecko != NULL){ //Jesli prawe dziecko nie jest puste to wywolujemy rekurencyjnie na
nim szukanie

  zwrotprawy = szukanie(drzewo_do_odczytania->prawe_dziecko, szukana_wartosc);}


  return max(zwrotlewy,zwrotprawy); //Brany jest max z wartosci tak by jesli gdzies sie pojawila 1 bo znalezlismy liczbe
zostala ona przekazana poza funkcje

}

string szukaj_czy_jest_element(drzewo_binarne* drzewo_do_odczytania, int szukana_wartosc){ //Funkcja pomocnicza dla
szukania gdzie konwertujemy wyniki 1 albo 0 w tekst


if (szukanie(drzewo_do_odczytania,szukana_wartosc)==1){return "znaleziono";}else{return "nie znaleziono";} //Jesli wynik
szukania to jeden to oznacza to ze znaleziono element o szukanej wartosci

}

int main()

{

  struct drzewo_binarne *przyklad = nowe_drzewo_binarne(10); //Inicjacja struktury, bierzemy wartosc 10 by root
wyroznial sie od reszty wezlow


  cout<<"1. Dodawanie wezlow z wartosciami:"<<endl<<endl;


  fstream wczytywanie; //Otwieranie pliku

  wczytywanie.open("Galezie.txt", ios::in);

  while(!wczytywanie.eof()){ //Aż do końca pliku

    int p; //Deklaracja zmiennej pomocniczej do której będą zapiswyane po kolei wartości

    wczytywanie >> p;

    dodawanie(przyklad, p); //Wywołanie funkcji dodawania z wartoscia wczytana z pliku

  }

  wczytywanie.close();


  cout<<endl<<"2. Wyswietlanie drzewa: ";
```

```
   int  ilosc=0; //Deklaracja zmiennej potrzebnej do zliczania

   ilosc = odczytywanie_i_zliczanie(przyklad,ilosc); //Funkcja wyswietla drzewo i za razem jej zwracana wartosc jest iloscia wezlow


   zapisywanie.open("Wyswietlone_drzewo.txt", ios::out); //Otwieranie nowego pliku tekstowego do ktorego zapisywane bedzie drzewo

   zapisywanie_do_pliku(przyklad); //Wywołanie funkcji zapisywania

   zapisywanie.close(); //Zamkniecie pliku


   int poziomy = ile_poziomow(przyklad); //Deklaracja zmiennej do ktorej przypisujemy wynik funkcji sprawdzajacej ilosc poziomow

   cout<<endl<<endl<<"Ilosc poziomow: "<<poziomy<<endl<<"Ilosc wezlow: "<<ilosc<<endl<<endl; //Wyswietlanie uzyskanych wartosci


   przyklad->lewe_dziecko = NULL; //Usuwanie nie potrzebuje pisania nowej funkcji, przypisanie wezlowi wartosci NULL likwiduje wszystkie wychodzace z niego wezly


   cout<<"3. Drzewo po usunieciu wezla: ";


   odczytywanie_i_zliczanie(przyklad,ilosc); //Wyswietlanie drzewa po usunieciu


   cout<<endl<<endl<<"4. Szukanie czy element znajduje sie w drzewie: ";


   cout<<szukaj_czy_jest_element(przyklad, 7)<<endl<<endl;


   return 0;

}
```