SPRAWOZDANIE

Autor: Michał Deć

Numer indeksu: 169771

Grupa projektowa numer 1

Zadanie projektowe numer 3

Opiekun pracy: **dr inż. prof. PRz Mariusz Borkowski**

Rzeszów, rok 2022

1. Opis problemu

Program ma za zadanie dokonać implementacji struktury danych typu drzewo binarne wraz z wszelkimi potrzebnymi operacjami charakterystycznymi dla tej struktury, takich jak: inicjowanie struktury, dodawanie lub usuwanie elementów, wyświetlanie elementów oraz zliczanie elementów i wyszukiwanie zadanego elementu.

2. Opis teoretyczny zagadnienia

2.1 Czym jest drzewo binarne

Drzewo binarne jest hierarchiczną strukturą danych, którego elementy nazywa się węzłami (nodes) lub wierzchołkami. W drzewie binarnym każdy węzeł posiada co najwyżej dwa następniki (stąd jego nazwa, bo binarny = dwójkowy, zawierający dwa elementy). Następniki te nazywamy potomkami, dziećmi lub węzłami potomnymi (child node) danego węzła – ojca (parent node). Węzeł nie posiadający rodzica nazywany jest korzeniem drzewa binarnego (root node).

2.2 Reguła drzewa binarnego

Wszystkie węzły drzewa spełniają tak zwaną regułę drzewa binarnego. Elementy znajdujące się w lewym poddrzewie są mniejsze od swojego ojca, natomiast elementy prawego poddrzewa są większe od ojca. Reguła ta obowiązuje wszystkie poddrzewa.

2.3 Zasada działania drzewa binarnego

Dzięki przestrzeganiu reguły drzewa binarnego można użyć metody dziel i zwyciężaj (divide and conquer). Biorąc pod uwagę dowolny węzeł to w jego lewym poddrzewie zawsze są wartości mniejsze od rodzica, zaś w prawym poddrzewie wartości większe od rodzica. Jeśli szukamy dowolnej liczby w drzewie, to najpierw sprawdzimy wartość korzenia i dzięki temu ocenimy, w którym poddrzewie należy tej wartości szukać, przez co odrzucimy połowę węzłów do przeszukania.

Drzewo binarne najlepiej jest wykonać na implementacji wskaźnikowej. Do lewego potomka węzła k-tego dostajemy się lewym wskaźnikiem ojca, zaś do prawego potomka węzła k-tego dostajemy się prawym wskaźnikiem ojca. Jeśli węzeł potomny nie istnieje, to odpowiedni wskaźnik ojca wskazuje na NULL.

Funkcje obsługujące drzewo:

- add_node() - dodaje element do drzewa.
- if_node_exsists() - znajduje element w drzewie.
- del_node_key() - usuwa dany element.
- display() - wydruk elementów drzewa na ekranie.
- count_all_nodes() - zlicza ilość węzłów

2.4 Diagram drzewa binarnego

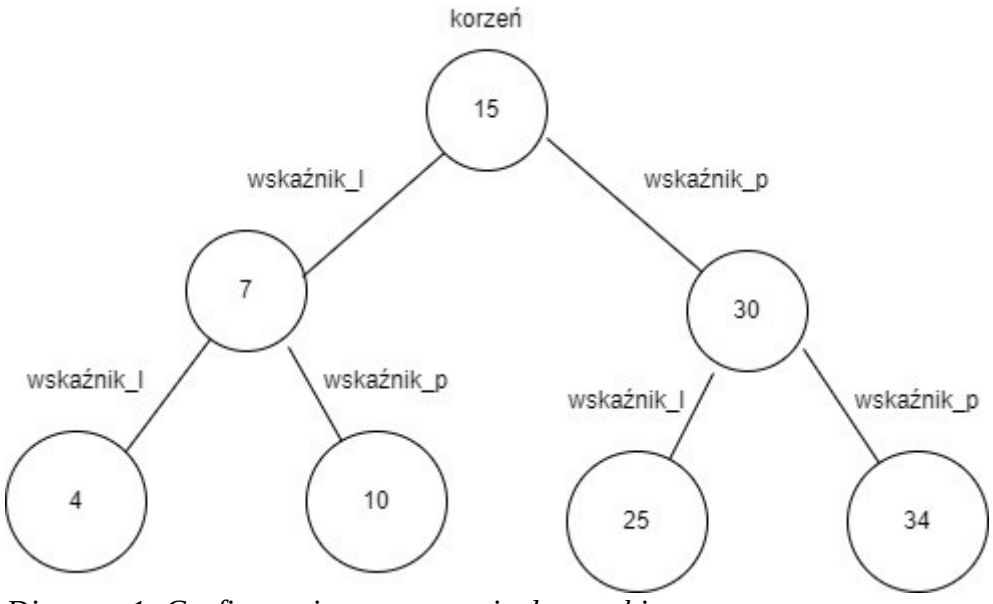
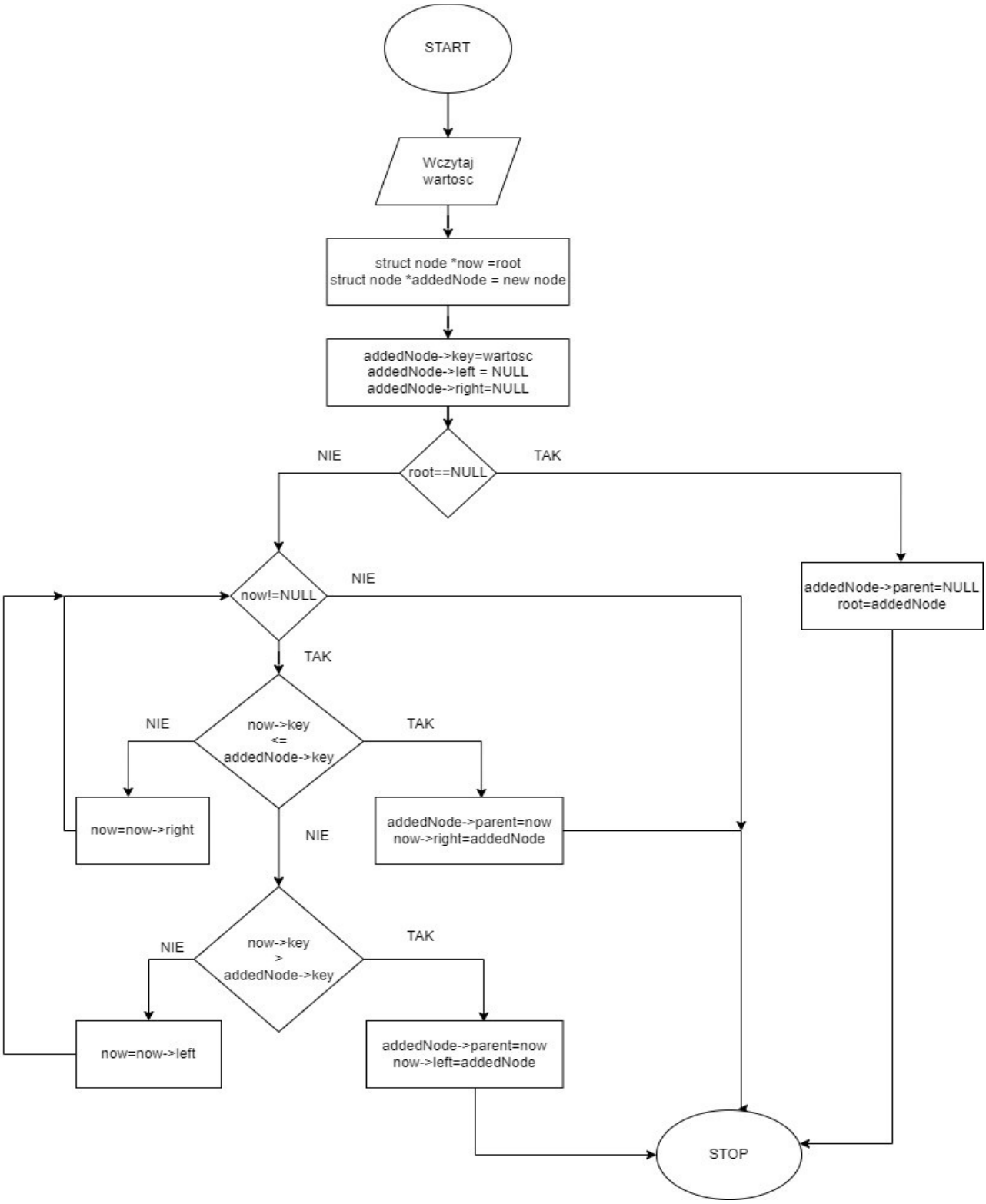


Diagram 1: Graficzne zinterpretowanie drzewa binarnego

Na powyższym diagramie możemy zauważyć zachowanie reguły drzewa binarnego. Dzięki temu, jeśli chcemy wyszukać dowolną liczbę, na przykład 10, to algorytm na początku sprawdzi wartość korzenia i dzięki temu będzie wiedział, w którym poddrzewie ma szukać liczby. Na starcie zaoszczędzimy czasu, ponieważ 10<15, więc przy wyszukiwaniu liczby algorytm odrzuci całą prawą stronę poddrzewa od korzenia, ponieważ wie, że tam jej nie znajdzie. Następnie program sprawdzi wartość w lewym poddrzewie. Skoro 10>7, to algorytm odrzuci z kolei lewą stronę poddrzewa od wartości 7. Przekłada się to na metodę dziel i zwyciężaj.

3. Schemat blokowy funkcji dodającej węzeł



Schemat 1: Schemat blokowy funkcji dodającej węzeł

4. Pseudokod funkcji dodającej węzeł

Wczytaj wprowadzoną wartość.

Utwórz wskaźniki wskazujące na aktualnie rozpatrywany węzeł i nowy węzeł.

Przypisz wartości do węzła.

Sprawdź czy korzeń ma wartość NULL, jeśli tak to przypisz wprowadzoną wartość do korzenia.

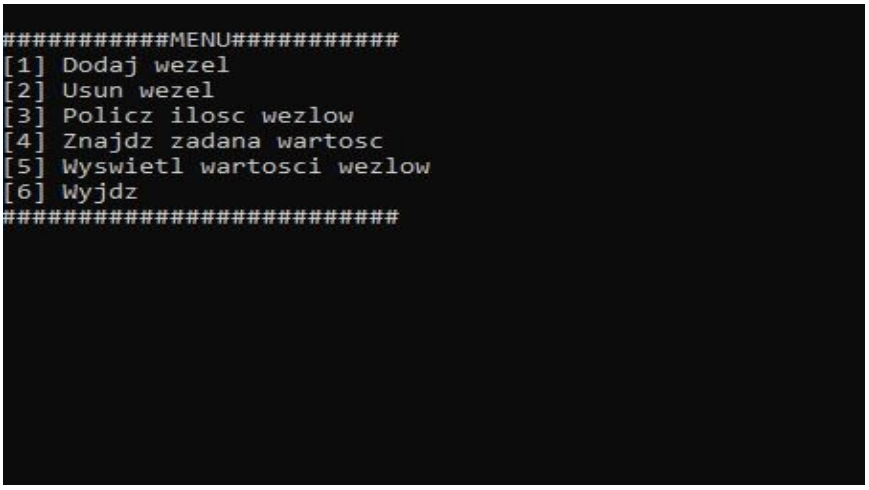
W przeciwnym wypadku wykonuj dopóki aktualny węzeł ma wartość inną, niż NULL:

> Jeśli wartość nowego węzła jest większa bądź równa od wskazywanego, to sprawdź czy po prawej stronie jest miejsce i dodaj w nie węzeł. W innym wypadku przejdź w głąb drzewa.
>
> Jeśli wartość nowego węzła jest mniejsza od wskazywanego, to sprawdź czy po lewej stronie jest miejsce. Jeśli tak to dodaj w nie aktualny węzeł. W przeciwnym wypadku przejdź w głąb drzewa.

Zakończ działanie funkcji.

5. Przedstawienie możliwości programu

5.1 Prezentacja menu głównego



Obraz 1: Wygląd menu głównego

5.2 Inicjowanie struktury

Według zasady drzewa binarnego każdy z węzłów musi przechowywać przynajmniej: zmienną wartości węzła, wskaźnik na rodzica, wskaźnik na lewe dziecko i wskaźnik na prawe dziecko. Jeśli dany węzeł nie ma dziecka po danej stronie, to wskaźnik należy ustawić na wartość NULL. Dzięki temu jesteśmy w stanie zweryfikować czy dany wskaźnik wskazuje na dziecko czy na pustą przestrzeń. Przy dodawaniu węzłów, jeśli wskaźnik na korzeń ma wartość NULL to nowy węzeł staje się automatycznie korzeniem.

```
 7
 8  struct node {
 9      int key;
10      struct node *left;
11      struct node *right;
12      struct node *parent;
13  };
14
15  struct node *root = NULL;
16
```

Obraz 2: Inicjowanie struktury.

5.3 Opis działania menu głównego

W programie mamy zdefiniowany panel sterowania. W zależności od wyboru jaki się wprowadzi program będzie wykonywał różne funkcje. Dla wyboru o numerze program doda nowy węzeł. Dla wyboru o numerze dwa program usunie dany węzeł z drzewa. Dla wyboru o numerze trzy program policzy ilość węzłów w drzewie. Dla wyboru o numerze cztery program wyszuka zadaną wartość w drzewie. Dla wyboru o numerze pięć program wyświetli aktualne wartości w drzewie. Dla wyboru o numerze sześć program zakończy swoje działanie.

```
208         switch (answ) {
209         case 1: {
210             cout << "Wporwadz wartosc wezla: ";
211             cin >> inpu;
212             add_node(inpu);
213             system("PAUSE");
214             break;
215         }
216         case 2: {
217             cout << "Wprowadz wartosc do usuniecia: ";
218             cin >> inpu;
219             if (root == NULL) {
220                 cout << "Brak korzenia - drzewo jest puste" << endl;
221             }
222             else{
223                 del_node_key(root, inpu);
224             }
225             system("PAUSE");
226             break;
227         }
228         case 3: {
229             cout << "Ilosc wezlow = " << count_all_nodes() << endl;
230             system("PAUSE");
231             break;
232         }
233         case 4: {
234             cout<<"Wprowadz wartosc do wyszukania: "<<endl;
235             cin>>inpu;
236 
237             if(if_node_exsists(root, inpu))
238             {
239                 cout<<"Odnaleziono poszukiwana wartosc"<<endl;
240             }
241             else
242             {
243                 cout<<"Nie odnaleziono poszukiwanej wartosci"<<endl;
```

```
244             }
245             system("PAUSE");
246             break;
247         }
248         case 5: {
249             display(root);
250             system("PAUSE");
251             break;
252         }
253         case 6: {
254             isRunning = false;
255             break;
256         }
257         default: {
258             cout << "Zly wybor!" << endl;
259             system("PAUSE");
260             break;
261         }
262         }
263     }
```

Obraz 3: Kod źródłowy panelu sterowania

6. Kod źródłowy funkcji działających w drzewie

6.1 Funkcja dodająca węzeł

```
/*
Funkcja dodawania wezla - funkcja ta przechodzi przez drzewo
w poszukiwaniu wolnego miejsca na wezel. Jesli wartosc nowego wezla
jest mniejsza od aktualnego, przechodzi w lewo, w przeciwnym przypadku
przechodzimy w prawo. Gdy natkniemy sie na wartosc null wstawiamy tam
nowy wezel pamietajac o przypisaniu wskaznikow.
*/
void add_node(int val) {
    //tworzymy wskazniki wskazujace na aktualnie rozpatrywany wezel i nowy wezel
    struct node *now = root;
    struct node *addedNode = new node;

    //Przypisujemy wartosci do wezla
    addedNode->key = val;
    addedNode->left = NULL;
    addedNode->right = NULL;

    //Jesli korzen ma wartosc null, nowy wezel staje sie korzeniem
    if (root == NULL) {
        addedNode->parent = NULL;
        root = addedNode;
        return;
    }
    else {

        while (now!= NULL) {
            //wartosc nowego wezla jest wieksza badz rowna od wskazywanego
            if (now->key <= addedNode->key) {
                //jesli po prawej jest wolne miejsce dodajemy w nie wezel
                if (now->right == NULL) {
                    addedNode->parent = now;
                    now->right = addedNode;
                    return;
                }
                //w innym przypadku przechodzimy w glab drzewa
                else {
                    now = now->right;
```

```
55                  }
56              }
57              //wartosc nowego wezla jest mniejsza od wskazywanego
58              if (now->key > addedNode->key) {
59                  //jesli po lewej jest wolne miejsce dodajemy w nie wezel
60                  if (now->left == NULL) {
61                      addedNode->parent = now;
62                      now->left = addedNode;
63                      return;
64                  }
65                  //w innym przypadku przechodzimy w glab drzewa
66                  else {
67                      now = now->left;
68                  }
69              }
70          }
71      }
72  }
```

Kod 1: Dodanie węzła

6.2 Funkcja usuwająca węzeł

```
74  /*
75  Funkcja usuwania wezla - funkcja przechodzi po drzewie w poszukiwaniu
76  wezla o zadanej wartosci. Jesli szukana wartosc jest mniejsza od wartosci
77  aktualnie wskazywanego wezla, przechodzi w lewo, w przeciwnym przypadku
78  przechodzimy w prawo. Jesli znajdziemy wezel o zadanej wartosci, usuwamy go.
79  W przypadku gdy posiadal jakiekolwiek dzieci, wstawiamy je w wolne miejsce.
80  Jesli zadana wartosc nie znajduje sie w drzewie, zwracamy NULL.
81  */
82  struct node* del_node_key(struct node* node, int find) {
83
84      if (node == NULL)
85          return node;
86
87      if (find < node->key) {
88          return node->left = del_node_key(node->left, find);     //Rekurencyjne przejscie do szukanej wartosci
89      }
90
91      else if (find > node->key) {
92          return node->right = del_node_key(node->right, find);
93      }
94
95      if (find == node->key) {
96          //Jezeli drzewo sklada sie z samego korzenia
97          if (root->left == NULL && root->right == NULL) {
98              delete(root);
99              root = NULL;
100             return NULL;
101         }
102         //Wsuniecie prawego syna na miejsce aktualnie wskazywanego
103         if (node->left == NULL) {
104             struct node *tmp = node->right;
105             delete(node);
106             return tmp;
107         }
108         //Wsuniecie lewego syna na miejsce aktualnie wskazywanego
109         else if (node->right == NULL) {
110             struct node *tmp = node->left;
111             delete(node);
```

```
112             return tmp;
113         }
114
115         //Przypadek gdy wskazywany wezel ma dwóch synow
116         struct node* tmp = node->right;
117
118         node->key = tmp->key;
119         while (tmp->left != NULL) {
120             tmp = tmp->left;
121         }
122
123         //Znalezienie nastepcy
124         node->right = del_node_key(node->right, tmp->key);
125     }
126     return node;
127 }
```

Kod 2: Usuwanie węzła

6.3 Funkcja zliczająca ilość węzłów

```
129 //Zlicz wezly w sposob rekurencyjny
130 int count_node(struct node *now)
131 {
132     int count = 1;
133     if (now->left != NULL) {
134         count += count_node(now->left);
135     }
136     if (now->right != NULL) {
137         count += count_node(now->right);
138     }
139     return count;
140 }
141
142 //Zlicz wszystkie wezly
143 int count_all_nodes()
144 {
145     int count = 0;
146     if (root != NULL) {
147         count = count_node(root);
148     }
149     return count;
150 }
```

Kod 3: Zliczanie węzłów

6.4 Funkcja wyszukująca wartość

```
152   //Wyszukaj zadana wartosc w drzewie
153   bool if_node_exsists(struct node *now, int find)
154   {
155       if (now == NULL)
156           return false;
157
158       if (now->key == find)
159           return true;
160
161       // przejdz do lewego wezla
162       bool res1 = if_node_exsists(now->left, find);
163       // odnaleziono wezel z poszukiwana wartoscia
164       if(res1) return true;
165
166       //jesli nie odnaleziono wartosci po lewej stronie, to sprawdz prawy wezel
167       bool res2 = if_node_exsists(now->right, find);
168
169       return res2;
170   }
```

Kod 4: Wyszukanie elementu

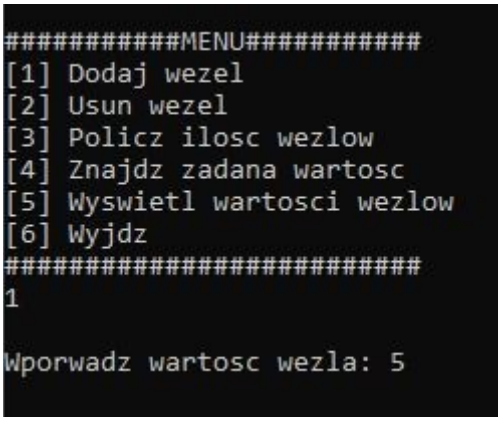
6.5 Funkcja wyświetlająca wartości

```
172   //wyswietl elementy drzewa
173   void display(struct node *now){
174
175       if(now != NULL){
176
177           //przejdz do wezla na lewo
178           display(now->left);
179
180           //wyswietl wartosc wezla
181           cout<<"Wartosc wezla wynosi: "<< now->key<<endl;
182
183           //przejdz do wezla na prawo
184           display(now->right);
185       }
186   }
```

Kod 5: Wyświetlanie elementu

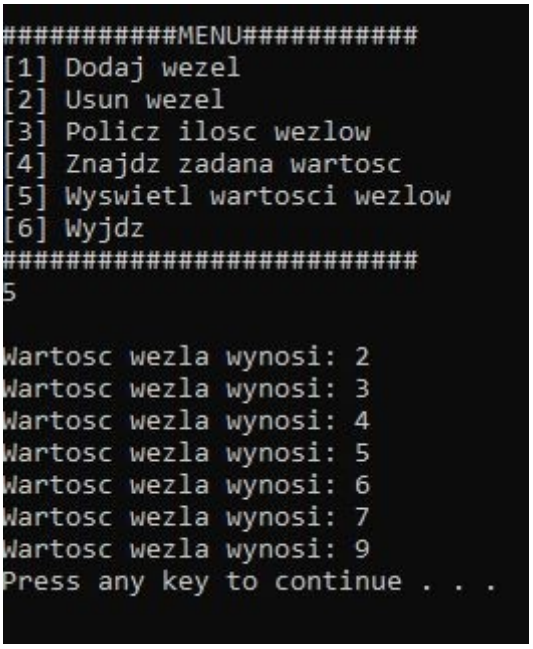
7. Testy działania programu

7.1 Test pierwszy



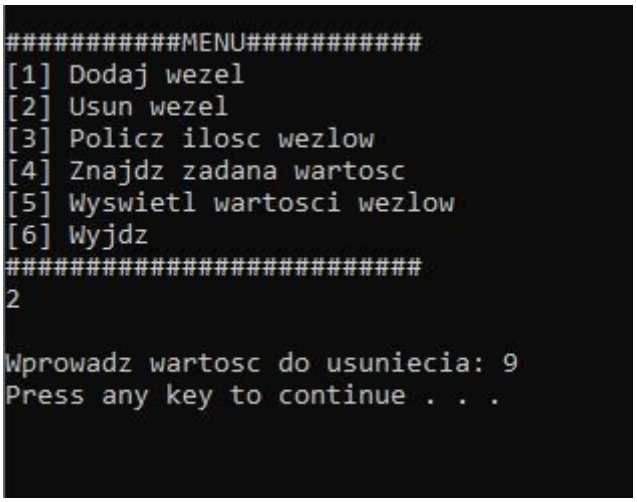
Test 1: Dodawanie węzła

7.2 Test drugi



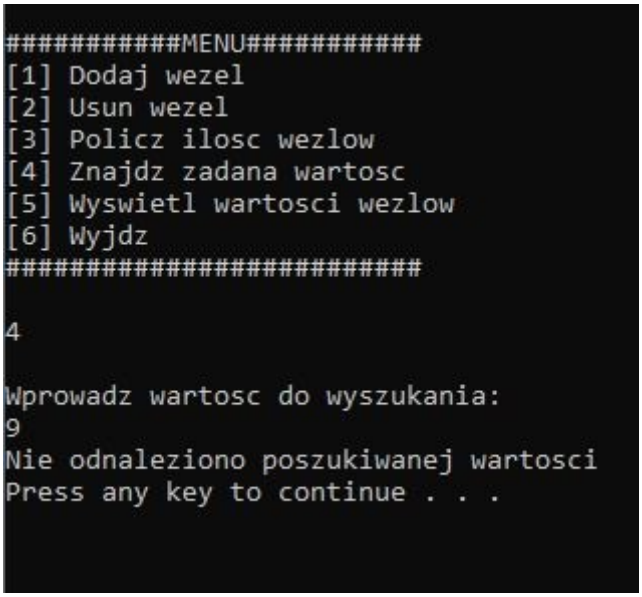
Test 2: Wyświetlenie węzłów

7.3 Test trzeci



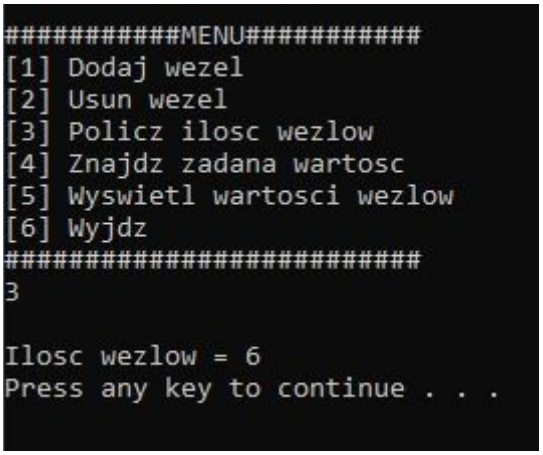
Test 3: Usuń węzeł

7.4 Test czwarty



Test 4: Wyszukanie wartości

7.5 Test piąty



Test 5: Zliczenie węzłów

8. Podsumowanie

Drzewa binarne w informatyce to niezwykle popularny rodzaj struktur danych. Ułatwiają one i przyspieszają wyszukiwanie, a także pozwalają w łatwy sposób operować na posortowanych danych. Dzięki zachowaniu reguły drzewa binarnego można skorzystać z tak zwanej metody dziel i zwyciężaj (divide and conquer), która pozwala w bardzo krótkim czasie wyszukiwać dane. Struktura katalogów w systemie operacyjnym jest właśnie strukturą drzewiastą o określonej hierarchii. Istnieje bardzo wiele zastosowań drzew w przemyśle informatycznym.

Spis treści

Diagram

Schemat blokowy

Spis obrazów

Spis kodu

Spis testów