**Autor**: Krzysztof Dąbrowski 293101 29.11.2018

SPRAWOZDANIE  
Kompresja LZW

# Cel zadania

Implementacja struktury danych drzewa o dowolnej liczbie dzieci w każdym węźle oraz przeszukiwań w głąb i w szerz.

# Repozytorium z kodem

<https://github.com/SiwyKrzysiek/Drzewo>

# Główne funkcje

* Struktura drzewa
* Przechodzenie w głąb
* Przechodzenie w szerz
* Wyszukiwanie w głąb
* Wyszukiwanie w szerz

# Przykładowe drzewo wykorzystane w programie



# Pliki źródłowe

## Program.cs

using System;

namespace Drzewo

{

class MainClass

{

public static void Main(string[] args)

{

Tree<int> tree = Tree<int>.CreateExampleTree();

//Przechodzenie przykładowego drzewa

Console.WriteLine("Przejście drzewa w głąb:");

Console.WriteLine(tree.DFSWalk());

Console.WriteLine("\nPrzejście drzewa w szerz:");

Console.WriteLine(tree.BFSWalk());

//Wyszukiwanie wskazanego elementu w drzewie

int found = tree.DFS(x => x == 1);

//Console.WriteLine("\n" + found);

}

}

}

## Tree.cs

using System.Collections.Generic;

using System.Text;

using System;

namespace Drzewo

{

public class Tree<T>

{

public T Data { get; set; } //Dana przechowywana w węźle

public List<Tree<T>> Children { get; set; } //Lista dzieci

/// <summary>

/// Konstruktor tworzący pojedyńczy węzeł

/// </summary>

/// <param name="value">Wartość wstawiana do drzewa</param>

public Tree(T value)

{

Data = value;

Children = new List<Tree<T>>(); //Na początku węzeł ma pustą listę dzieci

}

/// <summary>

/// Dodanie pojedyńczego dziecka do wybranego węzła

/// </summary>

/// <param name="value">Wartość do doddania</param>

public void AddOneChild(T value)

{

Tree<T> newNode = new Tree<T>(value); //Utworzenie nowego węzła do dodania

this.Children.Add(newNode); //Dopisanie utworzonego węzła jako dziecko aktualnego

}

/// <summary>

/// Dodanie dowolnej liczby dzieci

/// </summary>

/// <param name="values">Kolekcja dzieci do dodania</param>

public void AddManyChildren(IEnumerable<T> values)

{

foreach(T vale in values) //Dla każdej wartości dodajemy ją jako dziecko

{

this.Children.Add(new Tree<T>(vale));

}

}

/// <summary>

/// Zwraca i-te dziecko aktualnego węzła

/// </summary>

/// <param name="i">Numer dziecka</param>

public Tree<T> this[int i] => this.Children[i]; //Accesor zwracający odpowiednie dziecko

/// <summary>

/// Zwraca liczbę dzieci aktualnego węzła

/// </summary>

/// <value>Liczba dzieci</value>

int NumberOfChildren => this.Children.Count; //Accesor zwracający liczbę dzieci

/// <summary>

/// Zwraca ciąg wartości wszystkich dzieci danego węzła

/// </summary>

/// <returns>Wartości dzieci</returns>

public string GetChildrenAsText()

{

StringBuilder result = new StringBuilder();

foreach (Tree<T> child in this.Children) //Dla każdego dziecka dopisujemy jego wartość do wynikowego ciągu

result.Append(child.Data.ToString() + " ");

return result.ToString();

}

/// <summary>

/// Przechodzi drzewo w głąb i zwraca listę odwiedzonych węzłów

/// </summary>

/// <returns>Odwiedzone węzły</returns>

public string DFSWalk()

{

StringBuilder output = new StringBuilder();

DFSWalk(output);

return output.Remove(output.Length-2, 2).ToString(); //Pozbycie się ostatniego przecinka i spacji

}

/// <summary>

/// Funkja, która faktycznie przechodzi drzewo

/// </summary>

private void DFSWalk(StringBuilder result)

{

result.Append(this.Data.ToString() + ", ");

//System.Console.WriteLine(this.Data);

foreach (var child in this.Children)

{

child.DFSWalk(result);

}

}

/// <summary>

/// Przechodzi drzewo w szerz i zwraca listę odwiedzonych węzłów

/// </summary>

/// <returns>Odwiedzone węzły</returns>

public string BFSWalk()

{

StringBuilder output = new StringBuilder();

BFSWalk(output);

return output.Remove(output.Length - 2, 2).ToString(); //Pozbycie się ostatniego przecinka i spacji

}

/// <summary>

/// Funkja, która faktycznie przechodzi drzewo

/// </summary>

private void BFSWalk(StringBuilder result)

{

//System.Console.WriteLine(this.Data);

result.Append(this.Data.ToString() + ", ");

Queue<Tree<T>> nodesToVisit = new Queue<Tree<T>>(this.Children);

while (nodesToVisit.Count != 0)

{

Tree<T> currentNode = nodesToVisit.Dequeue();

result.Append(currentNode.Data.ToString() + ", ");

foreach (Tree<T> child in currentNode.Children)

nodesToVisit.Enqueue(child);

}

}

/// <summary>

/// Szukanie elementu w głąb drzewa.

/// </summary>

/// <param name="criterion">Funkcja zwracająca <c>true</c> dla szukanego elementu</param>

/// <exception cref="InvalidOperationException">Wyjątek zostaje rzucony w sytuacji gdy nie istnieje element spełniający kryterium</exception>

/// <returns>Szukany element</returns>

public T DFS(Predicate<T> criterion)

{

bool flag = false; //Przygotowanie flagi dla prawdziwej funkcji

return DFS(criterion, ref flag); //Wywołanie prawdziwej funkcji

}

/// <summary>

/// Rekurencyjna funkcja, która faktycznie szuka w głąb.

/// Jest niedostępna by mieć kontrolę nad dodatkowymi parametrami

/// </summary>

/// <param name="criterion">Funkcja zwracająca <c>true</c> dla szukanego elementu</param>

/// <param name="found">Flaga uzywana w rekurencji. Musi zostać zainicjowana zmienną o wartości <c>false</c></param>

/// <param name="lastChild">Flaga uzywana w rekurencji. Musi dostać wartośc <c>false</c> w oryginalnym wywołaniu</param>

/// <returns>Szukany element</returns>

private T DFS(Predicate<T> criterion, ref bool found, bool lastChild = true)

{

if (criterion(this.Data)) //Jeżeli aktualny element spełnia kryterium

{

found = true; //Ustawienie falagi by nie wchodzić w kolejne poziomy rekurencji

return this.Data; //Przekazanie znalezionego elementu

}

if (lastChild && this.Children.Count == 0) //Jeżeli algorytm doszedł do elementu, który jest ostatnim dzieckiem swojego rodzica i nie ma dzieci to oznacza to, że zostało przeszukane całe drzewo

throw new InvalidOperationException("Element not found");

for (int i = 0; i < this.Children.Count; i++) //Dla każdego dziecka

{

Tree<T> child = this.Children[i];

T result = child.DFS(criterion, ref found, i == this.Children.Count - 1 && lastChild); //Wywyołujemy rekurencyjnie szukanie

if (found) //Jeżeli flaga została ustawiona to nie ma sensu kontynułować szukania

return result;

}

return default(T); //Wartość domyślna jest zwracana, ponieważ funkcja musi coś zrwócić. Nie jest ona używana

}

/// <summary>

/// Szukanie elementu w szerz drzewa

/// </summary>

/// <param name="criterion">Funkcja zwracająca <c>true</c> dla szukanego elementu</param>

/// <exception cref="InvalidOperationException">Wyjątek zostaje rzucony w sytuacji gdy nie istnieje element spełniający kryterium</exception>

/// <returns>Szukany element</returns>

public T BFS(Predicate<T> criterion)

{

if (criterion(this.Data)) //Sprawdzenie korzenia

return this.Data;

Queue<Tree<T>> nodesToVisit = new Queue<Tree<T>>(this.Children); //Dzieci korzenia wstawiane są do kolejki

while (nodesToVisit.Count != 0) //Dopuki są węzły do sprawdzenia

{

Tree<T> currentNode = nodesToVisit.Dequeue(); //Wyjmowany jest pierwszy węzeł z kolejki

if (criterion(currentNode.Data)) //Sprawdzenie czy spełnia on kryterium

return currentNode.Data;

foreach (Tree<T> child in currentNode.Children) //Dodanie dzieci aktualnego węzła na koniec kolejki

nodesToVisit.Enqueue(child);

}

throw new InvalidOperationException("Element not found"); //Wszystkie węzły zostały sprawdzone

}

/// <summary>

/// Zwraca przykładowe drzewo int-ów

/// </summary>

/// <returns>Przykładowe drzewo</returns>

public static Tree<int> CreateExampleTree()

{

//Budowa przykładowego drzewa ("Drzewo.png")

//Pierwszy poziom

Tree<int> tree = new Tree<int>(7);

//Drugi poziom

tree.AddManyChildren(new int[] { 8, 3, 5 });

//Trzeci poziom

tree[0].AddManyChildren(new int[] { 14, 2 });

tree[1].AddOneChild(1);

tree[2].AddManyChildren(new int[] { -8, 30 });

//Czwarty poziom

tree[0][0].AddManyChildren(new int[] { 0, 1, 0 });

tree[1][0].AddManyChildren(new int[] { 3, 4, 2 });

tree[2][0].AddOneChild(13);

tree[2][1].AddManyChildren(new int[] { -10, 42 });

return tree;

}

}

}