Zespół Szkół Mechanicznych nr 1 im. Franciszka Siemiradzkiego
Technikum nr 10 Mechaniczne z Oddziałami Mistrzostwa Sportowego
Branżowa Szkoła I Stopnia nr 10 Mechaniczna
ul. Św. Trójcy 37, 85-224 Bydgoszcz
zsm1.bydgoszcz.pl

Materiały dydaktyczne

Przedmiot: Projektowanie oprogramowania/Pracownia projektowania oprogramowania
Opracował: Mirosław Miciak
Ćwiczenie: C6
Tematy: Cyfrowy zapis informacji, Algorytm, Sposoby przedstawiania algorytmu, Analiza algorytmu, Rekurencja, Funkcje rekurencyjne, Algorytm w postaci pseudokodu, Analiza algorytmów, Algorytm Euklidesa, Złożoność obliczeniowa algorytmów, Języki programowania, Kompresja, Wyszukiwanie binarne, Wyszukiwanie w zaawansowanych strukturach, Szyfr Cezara, Algorytm szyfrujący ROT-13, Kontrola błędów, Sortowanie bąbelkowe, Sortowanie przez wybór, Sortowanie szybkie, Implementacja quick sort, Sortowanie przez zliczanie (sortowanie klasy n), Porównanie algorytmów sortujących, Algorytmy sortowania podsumowanie, Struktury danych, Struktury danych sterta, Tworzenie kopca, Sortowanie przez kopcowanie, Drzewa binarne, Czysty kod, Sztuczna inteligencja, Komputerowe rozpoznawanie obrazów, Języki formalne, Wyrażenia regularne, Projektowanie klas UML, Wzorce projektowe, Wzorzec budowniczy, Wzorzec adapter, Wzorzec fabryka, Wzorzec fasada, Metoda szablonowa, Metodologie prowadzenia projektu, Model kaskadowy, Model prototypowy, Metodyki zwinne, Scrum, Role w metodyce SCRUM, Historyjki użytkownika, Kanban, Scrumban, System repozytoriów GIT, Programowanie zespołowe, Projekt programistyczny.

Struktury danych to sposób organizacji i przechowywania danych w taki sposób, aby umożliwić efektywny dostęp i modyfikację. Ważne struktury danych obejmują tablice, listy, stosy, kolejki, drzewa i kopce.

Tablice (Arrays)

Tablice to jedne z najbardziej podstawowych struktur danych. Są to zbiory elementów o stałym rozmiarze, które są tego samego typu i są indeksowane od zera.
Przykład w C#:

```
namespace StrukturyDanych
{
    internal class Program
    {
        static void Main(string[] args)
        {
            // Tworzenie i inicjalizacja tablicy
            int[] numbers = new int[] { 1, 2, 3, 4, 5 };

            // Przykład iteracji przez tablicę
            for (int i = 0; i < numbers.Length; i++)
            {
                Console.WriteLine(numbers[i]);
            }

        }
    }
```

```
}
```

Listy (Lists)

Teoria: Listy są dynamicznymi tablicami, które mogą zmieniać swój rozmiar w trakcie działania programu. Są one częścią przestrzeni nazw System.Collections.Generic.

Przykład w C#:

```
namespace StrukturyDanych
{
    internal class Program
    {
        static void Main(string[] args)
        {
            // Tworzenie i inicjalizacja listy
            List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };

            // Dodawanie elementów do listy
            numbers.Add(6);

            // Przykład iteracji przez listę
            foreach (int number in numbers)
            {
                Console.WriteLine(number);
            }


        }
    }
}
```

Stosy (Stacks)

Stos to struktura danych działająca na zasadzie LIFO (Last In, First Out), co oznacza, że ostatni dodany element jest pierwszym usuwanym.

Przykład w C#:

```
// Tworzenie stosu
Stack<int> stack = new Stack<int>();

// Dodawanie elementów na stos
stack.Push(1);
stack.Push(2);
stack.Push(3);

// Usuwanie elementu ze stosu
int top = stack.Pop();
Console.WriteLine(top); // Wyświetli 3
```

Kolejki (Queues)

Kolejka to struktura danych działająca na zasadzie FIFO (First In, First Out), co oznacza, że pierwszy dodany element jest pierwszym usuwanym.

Przykład w C#:

```
namespace StrukturyDanych
{
    internal class Program
    {
        static void Main(string[] args)
        {
            // Tworzenie kolejki
            Queue<int> queue = new Queue<int>();

            // Dodawanie elementów do kolejki
            queue.Enqueue(1);
            queue.Enqueue(2);
            queue.Enqueue(3);
```

```
            // Usuwanie elementu z kolejki
            int front = queue.Dequeue();
            Console.WriteLine(front); // Wyświetli 1

        }
    }
}
```

Drzewa binarne (Binary Trees)

Teoria: Drzewa binarne to struktura danych, w której każdy węzeł ma maksymalnie dwoje dzieci: lewe i prawe. tym przykładzie klasa TreeNode reprezentuje pojedynczy węzeł drzewa binarnego, a klasa BinaryTree zarządza wstawianiem węzłów i przeszukiwaniem drzewa w trzech różnych metodach (preorder, inorder i postorder). Metoda Main demonstruje tworzenie drzewa i wyświetlanie jego węzłów w różnych kolejnościach.

Przykład w C#:

```
namespace StrukturyDanychDrzewoString
{
    // Klasa reprezentująca węzeł drzewa binarnego
    public class TreeNode
    {
        public int Value { get; set; }
        public TreeNode Left { get; set; }
        public TreeNode Right { get; set; }

        public TreeNode(int value)
        {
            Value = value;
            Left = null;
            Right = null;
        }
    }

    // Klasa reprezentująca drzewo binarne
    public class BinaryTree
    {
        public TreeNode Root { get; set; }

        public void Insert(int value)
        {
            Root = Insert(Root, value);
        }

        private TreeNode Insert(TreeNode node, int value)
        {
            if (node == null)
            {
                return new TreeNode(value);
            }

            if (value < node.Value)
            {
                node.Left = Insert(node.Left, value);
            }
            else
            {
                node.Right = Insert(node.Right, value);
            }

            return node;
        }

        // Metoda usuwająca węzeł
        public TreeNode Delete(TreeNode root, int key)
        {
            if (root == null) return root;

            if (key < root.Value)
```

```
        {
            root.Left = Delete(root.Left, key);
        }
        else if (key > root.Value)
        {
            root.Right = Delete(root.Right, key);
        }
        else
        {
            // Węzeł z jednym dzieckiem lub bez dzieci
            if (root.Left == null)
                return root.Right;
            else if (root.Right == null)
                return root.Left;

            // Węzeł z dwoma dziećmi
            root.Value = MinValue(root.Right);
            root.Right = Delete(root.Right, root.Value);
        }

        return root;
    }

    // Metoda pomocnicza do znajdowania minimalnej wartości w prawym poddrzewie
    private int MinValue(TreeNode node)
    {
        int minv = node.Value;
        while (node.Left != null)
        {
            minv = node.Left.Value;
            node = node.Left;
        }
        return minv;
    }

    // Przeszukiwanie preorder
    public void PreOrderTraversal(TreeNode node)
    {
        if (node == null)
        {
            return;
        }
        Console.WriteLine(node.Value);
        PreOrderTraversal(node.Left);
        PreOrderTraversal(node.Right);
    }

    // Przeszukiwanie inorder
    public void InOrderTraversal(TreeNode node)
    {
        if (node == null)
        {
            return;
        }
        InOrderTraversal(node.Left);
        Console.WriteLine(node.Value);
        InOrderTraversal(node.Right);
    }

    // Przeszukiwanie postorder
    public void PostOrderTraversal(TreeNode node)
    {
        if (node == null)
        {
            return;
        }
        PostOrderTraversal(node.Left);
        PostOrderTraversal(node.Right);
        Console.WriteLine(node.Value);
    }
}
```

```
    public class Program
    {
        public static void Main(string[] args)
        {
            BinaryTree tree = new BinaryTree();

            // Dodawanie węzłów do drzewa
            tree.Insert(5);
            tree.Insert(3);
            tree.Insert(7);
            tree.Insert(2);
            tree.Insert(4);
            tree.Insert(6);
            tree.Insert(8);

            Console.WriteLine("Przeszukiwanie preorder:");
            tree.PreOrderTraversal(tree.Root);

            Console.WriteLine("\nPrzeszukiwanie inorder:");
            tree.InOrderTraversal(tree.Root);

            Console.WriteLine("\nPrzeszukiwanie postorder:");
            tree.PostOrderTraversal(tree.Root);

            // Usuwanie węzła
            tree.Delete(tree.Root, 3);

            Console.WriteLine("\nPrzeszukiwanie inorder po usunięciu 3:");
            tree.InOrderTraversal(tree.Root);
        }
    }

}
```

ZADANIA

1. Na podstawie przykładu Drzewa binarne utwórz drzewo i dodaj do niego pięć imion. Następnie usuń trzy imiona i wyświetl pozostałe. Można porównać łańcuch za pomocą string.Compare(value, node.Value) <0 zamiast (value < node.Value) jak to miało miejsce dla liczb.

2. Użyj drzewa binarnego do sortowania liczb tak jak to miało miejsce w ćwiczeniu z algorytmów sortowań: wygeneruj duży zbiór danych liczbowych np. 100K elementów, zmierz czas sortowania wybraną metodą np. QuickSort a następnie zmierz czas wyświetlania elementów za pomocą przeszukiwania inorder, porównaj czasy sformułuj wnioski.

3. Na podstawie przykładu Listy utwórz listę zawierającą imiona pięciu osób. Następnie dodaj dwa nowe imiona na końcu listy i usuń jedno z początkowych imion. Możesz użyć metody Remove() lub RemoveAt().

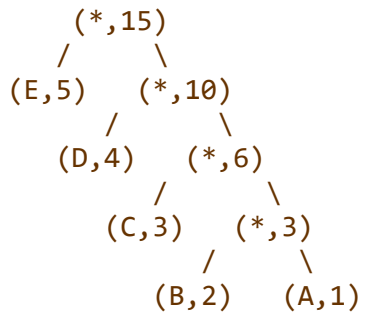
4. Na podstawie przykładu Stosy utwórz stos i dodaj do niego pięć imion. Następnie usuń trzy elementy ze stosu i wyświetl pozostałe.

5. Na podstawie przykładu Kolejki utwórz kolejkę i dodaj do niej pięć imion. Następnie usuń trzy elementy i wyświetl pozostałe.

... po godzinach

Drzewa binarne są używane w algorytmie Huffmana do kompresji danych. Algorytm Huffmana jest stosowany w kodowaniu znaków, aby zmniejszyć ilość danych potrzebną do przechowywania informacji. Dzięki temu, bardziej powszechnie używane znaki mają krótsze kody binarne, a rzadziej używane znaki - dłuższe kody binarne.

Przykład: Załóżmy, że mamy tekst do skompresowania: „ABBCCCDDDDEEEEE". Znaki występują w nim z różną częstotliwością:
A: 1 raz
B: 2 razy
C: 3 razy
D: 4 razy
E: 5 razy
Algorytm Huffmana buduje drzewo binarne na podstawie częstotliwości występowania znaków. Wynikowe drzewo może wyglądać następująco:



W tej strukturze, liście drzewa reprezentują znaki, a ich ścieżki od korzenia do liści reprezentują kody binarne. Kody binarne dla każdego znaku mogłyby wyglądać tak:
A: 111
B: 110
C: 10
D: 01
E: 00
Dzięki temu, zakodowany tekst „ABBCCCDDDDEEEEE" wyglądałby następująco:
111 110 110 10 10 10 01 01 01 01 00 00 00 00 00
Oznacza to, że zamiast przechowywać 15 znaków (każdy np. po 8 bitów), mamy 40 bitów kodu binarnego, co oszczędza miejsce.
implementacja algorytmu Huffmana w C#. Program będzie kompresował ciąg znaków i wyświetlał zakodowane wyniki.

```
namespace HuffmanCode
{
    using System;
    using System.Collections.Generic;

    public class HuffmanNode
```

```
    {
        public char Character { get; set; }
        public int Frequency { get; set; }
        public HuffmanNode Left { get; set; }
        public HuffmanNode Right { get; set; }

        public HuffmanNode(char character, int frequency)
        {
            Character = character;
            Frequency = frequency;
            Left = null;
            Right = null;
        }
    }

    public class HuffmanCoding
    {
        public HuffmanNode BuildHuffmanTree(Dictionary<char, int> frequencyDict)
        {
            PriorityQueue<HuffmanNode, int> priorityQueue = new PriorityQueue<HuffmanNode,
int>();

            foreach (var item in frequencyDict)
            {
                priorityQueue.Enqueue(new HuffmanNode(item.Key, item.Value), item.Value);
            }

            while (priorityQueue.Count > 1)
            {
                var left = priorityQueue.Dequeue();
                var right = priorityQueue.Dequeue();
                var sumFrequency = left.Frequency + right.Frequency;
                var newNode = new HuffmanNode('*', sumFrequency) { Left = left, Right = right
};
                priorityQueue.Enqueue(newNode, sumFrequency);
            }

            return priorityQueue.Dequeue();
        }

        public void GenerateCodes(HuffmanNode root, string code, Dictionary<char, string>
huffmanCode)
        {
            if (root == null)
            {
                return;
            }

            if (root.Left == null && root.Right == null)
            {
                huffmanCode[root.Character] = code;
            }

            GenerateCodes(root.Left, code + "0", huffmanCode);
            GenerateCodes(root.Right, code + "1", huffmanCode);
        }

        public string Encode(string text, Dictionary<char, string> huffmanCode)
        {
            string encodedText = string.Empty;
            foreach (var character in text)
            {
                encodedText += huffmanCode[character];
            }
            return encodedText;
        }

        public string Decode(string encodedText, HuffmanNode root)
        {
            string decodedText = string.Empty;
            HuffmanNode currentNode = root;
```

```
            foreach (var bit in encodedText)
            {
                currentNode = bit == '0' ? currentNode.Left : currentNode.Right;

                if (currentNode.Left == null && currentNode.Right == null)
                {
                    decodedText += currentNode.Character;
                    currentNode = root;
                }
            }

            return decodedText;
        }
    }

    public class Program
    {
        public static void Main(string[] args)
        {
            string text = "ABBCCCDDDDEEEEE";
            Dictionary<char, int> frequencyDict = new Dictionary<char, int>();

            foreach (var character in text)
            {
                if (frequencyDict.ContainsKey(character))
                {
                    frequencyDict[character]++;
                }
                else
                {
                    frequencyDict[character] = 1;
                }
            }

            HuffmanCoding huffmanCoding = new HuffmanCoding();
            HuffmanNode root = huffmanCoding.BuildHuffmanTree(frequencyDict);

            Dictionary<char, string> huffmanCode = new Dictionary<char, string>();
            huffmanCoding.GenerateCodes(root, string.Empty, huffmanCode);

            Console.WriteLine("Huffman Codes:");
            foreach (var item in huffmanCode)
            {
                Console.WriteLine($"{item.Key}: {item.Value}");
            }

            string encodedText = huffmanCoding.Encode(text, huffmanCode);
            Console.WriteLine($"\nEncoded Text: {encodedText}");

            string decodedText = huffmanCoding.Decode(encodedText, root);
            Console.WriteLine($"\nDecoded Text: {decodedText}");
        }
    }

}
```

HuffmanNode reprezentuje węzeł drzewa Huffmana.
HuffmanCoding zawiera metody do budowania drzewa Huffmana, generowania kodów, kodowania oraz dekodowania tekstu.
Program klasy głównej tworzy przykładowy tekst, generuje drzewa Huffmana i wyświetla zakodowany oraz odkodowany tekst.