**Міністерство освіти і науки України**

**Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського"**

**Факультет інформатики та обчислювальної техніки**

**Кафедра інформатики та програмної інженерії**

**Звіт**

з лабораторної роботи № 2 з дисципліни

«Проектування алгоритмів»

„ **Проектування структур даних**”

**Виконав(ла)**

(шифр, прізвище, ім'я, по батькові)

*ІП-01, Черпак Андрій Вадимович*

**Перевірив**

(прізвище, ім'я, по батькові)

*Головченко М.Н.*

Київ 2021

Зміст

[1 Мета лабораторної роботи 3](#_Toc91363051)

[2 Завдання 4](#_Toc91363052)

[3 Виконання 5](#_Toc91363053)

[3.1 Псевдокод алгоритмів 5](#_Toc91363054)

[3.1.1 Псевдокод алгоритму додавання запису 5](#_Toc91363055)

[3.1.2 Псевдокод алгоритму пошуку 5](#_Toc91363056)

[3.2 Часова складність пошуку 7](#_Toc91363057)

[3.3 Програмна реалізація 7](#_Toc91363058)

[3.3.1 Вихідний код 7](#_Toc91363059)

[3.3.2 Приклади роботи 16](#_Toc91363060)

[3.4 Тестування алгоритму 18](#_Toc91363061)

[3.4.1 Часові характеристики оцінювання 18](#_Toc91363062)

[Висновок 19](#_Toc91363063)

[Критерії оцінювання 20](#_Toc91363064)

# Мета лабораторної роботи

Мета роботи – вивчити основні підходи проектування та обробки складних структур даних.

# Завдання

Відповідно до варіанту (таблиця 2.1), записати алгоритми пошуку, додавання, видалення і редагування запису в структурі даних за допомогою псевдокоду (чи іншого способу по вибору).

Записати часову складність пошуку в структурі в асимптотичних оцінках.

Виконати програмну реалізацію невеликої СУБД, з функціями пошуку (алгоритм пошуку у вузлі структури згідно варіанту таблиця 2.1, за необхідності), додавання, видалення та редагування записів (запис складається із ключа і даних, ключі унікальні і цілочисельні, даних може бути декілька полів для одного ключа, але достатньо одного рядка фіксованої довжини). Для зберігання даних використовувати структуру даних згідно варіанту (таблиця 2.1).

Заповнити базу випадковими значеннями до 10000 і зафіксувати середнє (із 10-15 пошуків) число порівнянь для знаходження запису по ключу.

Зробити висновок з лабораторної роботи.

Таблиця 2.1 – Варіанти алгоритмів

|  |  |
| --- | --- |
| **№** | **Структура даних** |
| 29 | B-дерево t=50, метод Шарра |

# Виконання

## Псевдокод алгоритмів

### Псевдокод алгоритму додавання запису

**Insert**(requiredKey, newValue)

**якщо** Root == null **то**

Root = new BNode(true)

Root.Data[0] = (requiredKey, newValue);

Root.CountKeys = 1;

**інакше**

**якщо** Root.CountKeys == \_maxKeys **то**

upperNode = new BNode(false)

upperNode.Children[0] = Root

upperNode.SplitChild(0, Root)

insertTo = upperNode.Data[0].Key < requiredKey ? 1 : 0

upperNode.Children[insertTo].InsertionToNotFull(requiredKey, newValue)

Root = upperNode

**інкше**

Root.InsertionToNotFull(requiredKey, newValue)

**все якщо**

**все якщо**

\_maxKeyValue += 1

**кінець**

### Псевдокод алгоритму пошуку

**GetRecordByKey**(requiredKey)

nodeWithKey = Root.Search(requiredKey)

index = nodeWithKey.FindIndex(requiredKey)

**повернути** nodeWithKey.Data[index].Value

**кінець**

**Search**(key)

lastGreaterThanKey = CountKeys;

index = SharrSearch(key, ref lastGreaterThanKey);

**якщо** index != -1 **і** Data[index].Key == key **то**

**повернути** поточний вузол

**все якщо**

**якщо** поточний вузол – листок **то**

**повернути** null

**все якщо**

**якщо** index == -1 && !IsLeaf **то**

**повернути** Children[lastGreaterThanKey].Search(key)

**все якщо**

**повернути** null

**кінець**

**FindIndex**(requiredKey)

lastGreaterThanKey = 0

index = SharrSearch(requiredKey, ref lastGreaterThanKey)

**якщо** index == -1 **то**

**повернути** lastGreaterThanKey

**інакше**

**повернути** index

**все якщо**

**кінець**

## Часова складність пошуку

В кращому випадку, коли необхідне значення знаходитиметься в ключах кореневого вузла, пошук виконається за O(log k), де k – кількість ключів у даному вузлі.

В гіршому випадку, коли необхідне значення знаходиться в листках, час пошуку становитиме O(log(n)).

## Програмна реалізація

### Вихідний код

namespace ADS\_2  
{  
 static class Program  
 {  
 static void Main(string[] args)  
 {  
 BTree bTree = WorkWithFile.ImportFromFile();  
 UserInput.Listen(bTree);  
 }  
 }  
}

using System.Collections.Generic;  
  
namespace ADS\_2  
{  
 public class BTree  
 {  
 public BNode Root;  
 private const int **T** = 50;  
 private readonly int \_maxKeys = 2 \* **T** - 1;  
 private static int \_maxKeyValue;  
 public string SourceFile;  
 public int MaxKey  
 {  
 get => \_maxKeyValue;  
 set => \_maxKeyValue = value;  
 }  
  
 public BNode Search(int key)  
 {  
 if (Root != null)  
 {  
 return Root.Search(key);  
 }  
 return null;  
 }  
 public void Insert(int requiredKey, string newValue)  
 {  
 if (Root == null)  
 {  
 Root = new BNode(true);  
 Root.Data[0] = new KeyValuePair<int, string>(requiredKey, newValue);  
 Root.CountKeys = 1;  
 }  
 else  
 {  
 if (Root.CountKeys == \_maxKeys)  
 {  
 BNode upperNode = new BNode(false);  
 upperNode.Children[0] = Root;  
 upperNode.SplitChild(0, Root);  
 int insertTo = upperNode.Data[0].Key < requiredKey ? 1 : 0;  
 upperNode.Children[insertTo].InsertionToNotFull(requiredKey, newValue);  
 Root = upperNode;  
 }  
 else  
 Root.InsertionToNotFull(requiredKey, newValue);  
 }  
 \_maxKeyValue += 1;  
 }  
  
 public void RemoveRecord(int requiredKey)  
 {  
 BNode nodeWithRequiredKey = Root.Search(requiredKey);  
 nodeWithRequiredKey.RemoveKey(requiredKey);  
 }  
  
 public void EditRecord(int requiredKey, string newValue)  
 {  
 BNode nodeWithRequiredKey = Root.Search(requiredKey);  
 int index = nodeWithRequiredKey.FindIndex(requiredKey);  
 nodeWithRequiredKey.Data[index] = new KeyValuePair<int, string>(requiredKey, newValue);  
 }  
  
 public string GetRecordByKey(int requiredKey)  
 {  
 BNode nodeWithKey = Root.Search(requiredKey);  
 int index = nodeWithKey.FindIndex(requiredKey);  
 return nodeWithKey.Data[index].Value;  
 }  
 }  
}

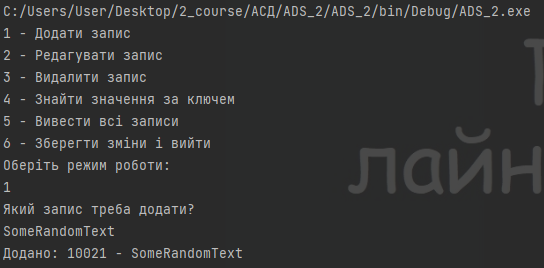
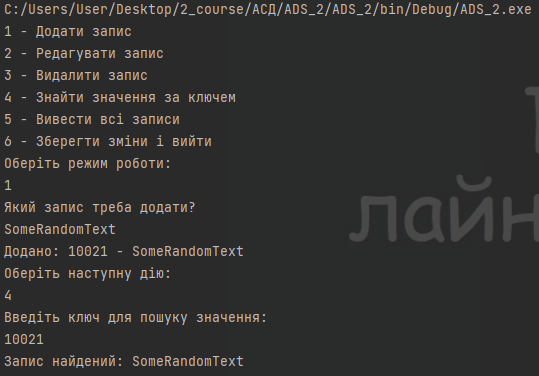
using System;  
using System.Collections.Generic;  
  
namespace ADS\_2  
{  
 public class BNode  
 {  
 private const int **T** = 50;  
 private const int **MinKeys** = **T** - 1;  
 private const int **MaxKeys** = 2 \* **T** - 1;  
  
 public readonly BNode[] Children;  
 public readonly KeyValuePair<int, string>[] Data;  
 public int CountKeys;  
 public readonly bool IsLeaf;  
  
 public BNode(bool leaf)  
 {  
 Data = new KeyValuePair<int, string>[**MaxKeys**];  
 Children = new BNode[2 \* **T**];  
 CountKeys = 0;  
 IsLeaf = leaf;  
 }  
  
 public int FindIndex(int requiredKey)  
 {  
 int lastGreaterThanKey = 0;  
 int index = SharrSearch(requiredKey, ref lastGreaterThanKey);  
 return index == -1 ? lastGreaterThanKey : index;  
 }  
  
 public void RemoveKey(int key)  
 {  
 int index = FindIndex(key);  
  
 if (index < CountKeys && Data[index].Key == key)  
 {  
 if (IsLeaf)  
 RemoveFromLeaf(index);  
 else  
 RemoveFromNonLeaf(index);  
 }  
 else  
 {  
 if (!IsLeaf)  
 {  
 if (Children[index].CountKeys < **MinKeys**) FillMissing(index);  
 Children[index].RemoveKey(key);  
 }  
 else  
 {  
 Console.WriteLine($"Ключа {key} нет в базе?");  
 }  
 }  
 }  
  
 private void RemoveFromLeaf(int index)  
 {  
 for (int i = index + 1; i < CountKeys; ++i)  
 {  
 Data[i - 1] = new KeyValuePair<int, string>(Data[i].Key, Data[i].Value);  
 }  
 CountKeys -= 1;  
 }  
  
 private void RemoveFromNonLeaf(int index)  
 {  
 int k = Data[index].Key;  
 if (Children[index].CountKeys >= **T**)  
 {  
 KeyValuePair<int, string> previous = GetPrevious(index);  
 Data[index] = new KeyValuePair<int, string>(previous.Key, previous.Value);  
 Children[index].RemoveKey(previous.Key);  
 }  
 else if (Children[index + 1].CountKeys >= **T**)  
 {  
 KeyValuePair<int, string> succ = GetNext(index);  
 var tempValue = Data[index].Value;  
 Data[index] = new KeyValuePair<int, string>(succ.Key, tempValue);  
 Children[index + 1].RemoveKey(succ.Key);  
 }  
 else  
 {  
 Unite(index);  
 Children[index].RemoveKey(k);  
 }  
 }  
  
 private KeyValuePair<int, string> GetPrevious(int index)  
 {  
 BNode currentNode = Children[index];  
 while (!currentNode.IsLeaf)  
 currentNode = currentNode.Children[currentNode.CountKeys];  
 return new KeyValuePair<int, string>(currentNode.Data[currentNode.CountKeys - 1].Key, currentNode.Data[currentNode.CountKeys - 1].Value);  
 }  
  
 private KeyValuePair<int, string> GetNext(int index)  
 {  
 BNode currentNode = Children[index + 1];  
 while (!currentNode.IsLeaf)  
 currentNode = currentNode.Children[0];  
 return new KeyValuePair<int, string>(currentNode.Data[0].Key, currentNode.Data[0].Value);  
 }  
  
 private void FillMissing(int index)  
 {  
 if (index != 0 && Children[index - 1].CountKeys >= **MinKeys**)  
 ReplaceFromPrevious(index);  
 else if (index != CountKeys && Children[index + 1].CountKeys >= **MinKeys**)  
 ReplaceFromNext(index);  
 else  
 {  
 if (index == CountKeys)  
 Unite(index - 1);  
 else  
 Unite(index);  
 }  
 }  
  
 private void ReplaceFromPrevious(int index)  
 {  
 BNode replaceFrom = Children[index - 1];  
 BNode replaceTo = Children[index];  
   
 for (int i = replaceTo.CountKeys - 1; i >= 0; --i)  
 {  
 replaceTo.Data[i + 1] = new KeyValuePair<int, string>(replaceTo.Data[i].Key, replaceTo.Data[i].Value);  
 }  
  
 if (!replaceTo.IsLeaf)  
 {  
 for (int i = replaceTo.CountKeys; i >= 0; --i)  
 {  
 replaceTo.Children[i + 1] = replaceTo.Children[i];  
 }  
 }  
  
 replaceTo.Data[0] = new KeyValuePair<int, string>(Data[index - 1].Key, Data[index - 1].Value);  
  
 if (!replaceTo.IsLeaf)  
 replaceTo.Children[0] = replaceFrom.Children[replaceFrom.CountKeys];  
  
 Data[index - 1] = new KeyValuePair<int, string>(replaceFrom.Data[replaceFrom.CountKeys - 1].Key, replaceFrom.Data[replaceFrom.CountKeys - 1].Value);  
  
 replaceTo.CountKeys += 1;  
 replaceFrom.CountKeys -= 1;  
 }  
  
 private void ReplaceFromNext(int index)  
 {  
 BNode replaceFrom = Children[index + 1];  
 BNode replaceTo = Children[index];  
  
 replaceTo.Data[replaceTo.CountKeys] = new KeyValuePair<int, string>(Data[index].Key, Data[index].Value);  
  
 if (!replaceTo.IsLeaf)  
 replaceTo.Children[replaceTo.CountKeys + 1] = replaceFrom.Children[0];  
  
 Data[index] = new KeyValuePair<int, string>(replaceFrom.Data[0].Key, replaceFrom.Data[0].Value);  
  
 for (int i = 1; i < replaceFrom.CountKeys; ++i)  
 {  
 replaceFrom.Data[i - 1] = new KeyValuePair<int, string>(replaceFrom.Data[i].Key, replaceFrom.Data[i].Value);  
 }  
  
 if (!replaceFrom.IsLeaf)  
 {  
 for (int i = 1; i <= replaceFrom.CountKeys; ++i)  
 {  
 replaceFrom.Children[i - 1] = replaceFrom.Children[i];  
 }  
 }  
 replaceTo.CountKeys += 1;  
 replaceFrom.CountKeys -= 1;  
 }  
  
 private void Unite(int index)  
 {  
 BNode uniteIn = Children[index];  
 BNode uniteFrom = Children[index + 1];  
 uniteIn.Data[**MinKeys**] = new KeyValuePair<int, string>(Data[index].Key, Data[index].Value);  
  
 for (int i = 0; i < uniteFrom.CountKeys; ++i)  
 {  
 uniteIn.Data[i + **T**] = new KeyValuePair<int, string>(uniteFrom.Data[i].Key, uniteFrom.Data[i].Value);  
 }  
  
 if (!uniteIn.IsLeaf)  
 {  
 for (int i = 0; i <= uniteFrom.CountKeys; ++i)  
 {  
 uniteIn.Children[i + **T**] = uniteFrom.Children[i];  
 }  
 }  
  
 for (int i = index + 1; i < CountKeys; ++i)  
 {  
 Data[i - 1] = new KeyValuePair<int, string>(Data[i].Key, Data[i].Value);  
 }  
  
 for (int i = index + 2; i <= CountKeys; ++i)  
 {  
 Children[i - 1] = Children[i];  
 }  
  
 uniteIn.CountKeys += uniteFrom.CountKeys + 1;  
 CountKeys -= 1;  
 }  
  
 public void InsertionToNotFull(int key, string value)  
 {  
 int insertTo = CountKeys - 1;  
  
 if (IsLeaf)  
 {  
 while (insertTo >= 0 && Data[insertTo].Key > key)  
 {  
 Data[insertTo + 1] = new KeyValuePair<int, string>(Data[insertTo].Key, Data[insertTo].Value);  
 insertTo--;  
 }  
  
 Data[insertTo + 1] = new KeyValuePair<int, string>(key, value);  
 CountKeys += 1;  
 }  
 else  
 {  
 while (insertTo >= 0 && Data[insertTo].Key > key)  
 insertTo--;  
  
 if (Children[insertTo + 1].CountKeys == **MaxKeys**)  
 {  
 SplitChild(insertTo + 1, Children[insertTo + 1]);  
  
 if (Data[insertTo + 1].Key < key)  
 insertTo++;  
 }  
 Children[insertTo + 1].InsertionToNotFull(key, value);  
 }  
 }  
  
 public void SplitChild(int splitIndex, BNode nodeToSplit)  
 {  
 BNode neighborNode = new BNode(nodeToSplit.IsLeaf)  
 {  
 CountKeys = **MinKeys** };  
  
 for (int j = 0; j < **MinKeys**; j++)  
 {  
 neighborNode.Data[j] = new KeyValuePair<int, string>(nodeToSplit.Data[j + **T**].Key, nodeToSplit.Data[j + **T**].Value);  
 }  
  
 if (!nodeToSplit.IsLeaf)  
 {  
 for (int j = 0; j < **T**; j++)  
 {  
 neighborNode.Children[j] = nodeToSplit.Children[j + **T**];  
 }  
 }  
  
 nodeToSplit.CountKeys = **MinKeys**;  
 for (int j = CountKeys; j >= splitIndex + 1; j--)  
 {  
 Children[j + 1] = Children[j];  
 }  
 Children[splitIndex + 1] = neighborNode;  
  
 for (int j = CountKeys - 1; j >= splitIndex; j--)  
 {  
 Data[j + 1] = new KeyValuePair<int, string>(Data[j].Key, Data[j].Value);  
 }  
 Data[splitIndex] = new KeyValuePair<int, string>(nodeToSplit.Data[**MinKeys**].Key, nodeToSplit.Data[**MinKeys**].Value);  
 CountKeys += 1;  
 }  
  
 public BNode Search(int key)  
 {  
 int lastGreaterThanKey = CountKeys;  
 int index = SharrSearch(key, ref lastGreaterThanKey);  
 if (index != -1 && Data[index].Key == key) return this;  
 if (IsLeaf) return null;  
 if (index == -1 && !IsLeaf) return Children[lastGreaterThanKey].Search(key);  
 return null;  
 }  
  
 private int SharrSearch(int requiredKey, ref int lastGreaterThanKey)  
 {  
 int k = (int)Math.Log(CountKeys, 2);  
 int i = (int)Math.Pow(2, k);  
 if (Data[i].Key == requiredKey) return i;  
 if (requiredKey > Data[i].Key)  
 {  
 k = (int)Math.Log((CountKeys - Math.Pow(2, k) + 1), 2);  
 i = CountKeys + k - (int)Math.Pow(2, k);  
 if (Data[i].Key == requiredKey) return i;  
 }  
 else  
 {  
 lastGreaterThanKey = i;  
 i /= 2;  
 if (Data[i].Key == requiredKey) return i;  
 }  
 int б = (int)Math.Pow(2, k - 1);  
 while (б>0)  
 {  
 if (requiredKey > Data[i].Key)  
 {  
 i+=б;  
 if (i >= CountKeys) i = CountKeys - 1;  
 }  
 else  
 {  
 lastGreaterThanKey = i;  
 i -= б;  
 }  
 if (Data[i].Key == requiredKey) return i;  
 б /= 2;  
 }  
 return -1;  
 }  
 }  
}

using System.IO;  
  
namespace ADS\_2  
{  
 public static class WorkWithFile  
 {  
 public static BTree ImportFromFile(string fileName = "data.csv")  
 {  
 BTree tree = new ()  
 {  
 SourceFile = fileName  
 };  
 using (StreamReader streamReader = new (fileName))  
 {  
 int dataKey;  
 string currentLine, dataValue;  
 while ((currentLine = streamReader.ReadLine()) != null)  
 {  
 if (currentLine == "")  
 continue;  
 dataKey = int.Parse(currentLine.Split(';')[0]);  
 dataValue = currentLine.Split(';')[1];  
 if (tree.Search(dataKey) == null)  
 tree.Insert(dataKey, dataValue);  
 }  
 }  
  
 return tree;  
 }  
   
 public static void TreeToFile(BTree tree, string fileName = null)  
 {  
 if (tree.Root==null) return;  
 fileName ??= tree.SourceFile;  
 using (StreamWriter streamWriter = new StreamWriter(fileName, false)) NodeToFile(tree.Root, streamWriter);  
 }  
  
 private static void NodeToFile(BNode node, StreamWriter sw)  
 {  
 int currentIndex;  
 for (currentIndex = 0; currentIndex < node.CountKeys; currentIndex++)  
 {  
 if (!node.IsLeaf)  
 NodeToFile(node.Children[currentIndex], sw);  
 sw.WriteLine($"{node.Data[currentIndex].Key};{node.Data[currentIndex].Value}");  
 }  
   
 if (!node.IsLeaf) NodeToFile(node.Children[currentIndex], sw);  
 }  
 }  
}

using System;  
using System.Collections.Generic;  
  
namespace ADS\_2  
{  
 public static class UserInput  
 {  
 public static void Listen(BTree tree)  
 {  
 Console.WriteLine("1 - Додати запис");  
 Console.WriteLine("2 - Редагувати запис");  
 Console.WriteLine("3 - Видалити запис");  
 Console.WriteLine("4 - Знайти значення за ключем");  
 Console.WriteLine("5 - Вивести всi записи");  
 Console.WriteLine("6 - Зберегти змiни i вийти");  
 Console.WriteLine("Оберiть режим роботи:");  
 int type, key;  
 while (!int.TryParse(Console.ReadLine(), out type))  
 {  
 Console.WriteLine("Некоректний ввiд!");  
 Console.WriteLine("Оберiть режим роботи:");  
 }  
 string input;  
 while (type != 6)  
 {  
 switch (type)  
 {  
 case 1:  
 key = tree.MaxKey;  
 while (tree.Search(key) != null)  
 {  
 key++;  
 }  
 tree.MaxKey = key;  
 Console.WriteLine("Який запис треба додати?");  
 input = Console.ReadLine();  
 tree.Insert(key, input);  
 Console.WriteLine($"Додано: {key} - {input}");  
 break;  
  
 case 2:  
 Console.WriteLine("Введiть ключ для редагування запису:");  
 key = int.Parse(Console.ReadLine());  
 if (tree.Search(key) != null)  
 {  
 Console.WriteLine("Введiть новий запис:");  
 input = Console.ReadLine();  
 tree.EditRecord(key, input);  
 }  
 else  
 {  
 Console.WriteLine("Запису з таким ключем немає. Додати запис? [y/n]");  
 string YesOrNo = Console.ReadLine();  
 while (!new List<string>{"Y", "y", "Yes", "yes", "Ok", "ok", "Okay", "okay", "+", "Так", "так", "N", "n", "No", "no", "-", "Ні", "ні"}.Contains(YesOrNo))  
 {  
 Console.WriteLine("Некоректний ввід! Оберіть з таких варіантів: Y, N");  
 YesOrNo = Console.ReadLine();  
 }  
  
 if (new List<string> { "Y", "y", "Yes", "yes", "Ok", "ok", "Okay", "okay", "+", "Так", "так" }.Contains(YesOrNo))  
 {  
 Console.WriteLine("Який запис треба додати?");  
 input = Console.ReadLine();  
 tree.Insert(key, input);  
 Console.WriteLine($"Додано: {key} - {input}");  
 }  
 }  
  
 break;  
  
 case 3:  
 Console.WriteLine("Введiть ключ для видалення:");  
 key = int.Parse(Console.ReadLine());  
 if (tree.Search(key) != null)  
 {  
 tree.RemoveRecord(key);  
 }  
 else  
 Console.WriteLine("Запису з таким ключем немає");  
 break;  
  
 case 4:  
 Console.WriteLine("Введiть ключ для пошуку значення: ");  
 key = int.Parse(Console.ReadLine());  
 if (tree.Search(key) != null)  
 {  
 Console.WriteLine($"Запис найдений: {tree.GetRecordByKey(key)}");  
 }  
 else  
 Console.WriteLine("Запису з таким ключем немає");  
 break;  
  
 case 5:  
 if (tree.Root!=null) Print(tree.Root);  
 else Console.WriteLine("Дерево порожнє!");  
 break;  
 default:  
 Console.WriteLine("Невiдома команда!");  
 break;  
 }  
 Console.WriteLine("Оберiть наступну дiю:");  
 while (!int.TryParse(Console.ReadLine(), out type))  
 {  
 Console.WriteLine("Некоректний ввiд!");  
 Console.WriteLine("Оберiть режим роботи:");  
 }  
 }  
 WorkWithFile.TreeToFile(tree);  
 }  
   
 private static void Print(BNode node)  
 {  
 int currentIndex;  
 for (currentIndex = 0; currentIndex < node.CountKeys; currentIndex++)  
 {  
 if (!node.IsLeaf)  
 Print(node.Children[currentIndex]);  
 Console.WriteLine($"{node.Data[currentIndex].Key};{node.Data[currentIndex].Value}");  
 }  
  
 if (!node.IsLeaf) Print(node.Children[currentIndex]);  
 }  
 }  
}

### Приклади роботи

На рисунках 3.1 і 3.2 показані приклади роботи програми для додавання і пошуку запису.

  
Рисунок 3.1 –Додавання запису  
  
Рисунок 3.2 – Пошук запису

## Тестування алгоритму

### Часові характеристики оцінювання

В таблиці 3.1 наведено кількість порівнянь для 15 спроб пошуку запису по ключу.

Таблиця 3.1 – Число порівнянь при спробі пошуку запису по ключу

|  |  |
| --- | --- |
| Номер спроби пошуку | Число порівнянь |
| 1 | 1179 |
| 2 | 34 |
| 3 | 2409 |
| 4 | 3881 |
| 5 | 702 |
| 6 | 1310 |
| 7 | 2592 |
| 8 | 284 |
| 9 | 3615 |
| 10 | 8709 |
| 11 | 2161 |
| 12 | 6305 |
| 13 | 2046 |
| 14 | 1608 |
| 15 | 973 |
| Cереднє | 2520,533 |

Висновок

В рамках лабораторної роботи я реалізував В-дерево, а також функції додавання, видалення, редагування та пошуку у ньому на мові C#. Дану структуру можна ефективно використовувати для зберігання та роботи з великою кількістю даних оскільки воно збалансоване, а використання бінарного пошуку методом Шарра у вузлах дозволяє зменшити кількість порівнянь та ще більше оптимізувати код.

Критерії оцінювання

За умови здачі лабораторної роботи до 15.10.2021 включно максимальний бал дорівнює – 5. Після 15.10.2021 максимальний бал дорівнює – 1.

Критерії оцінювання у відсотках від максимального балу:

* псевдокод алгоритму – 20%;
* аналіз часової складності – 5%;
* програмна реалізація алгоритму – 60%;
* тестування алгоритму – 10%;
* висновок – 5%.

+1 додатковий бал можна отримати за реалізацію графічного інтерфейсу.

+1 додатковий бал можна отримати за реалізацію графічного зображення структури ключів.