**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)

*ІТ-04 Стрільчук Михайло Васильович*

**Перевірив**

(прізвище, ім'я, по батькові)

*Головченко М.Н.*

Київ 2021

Зміст

[1 Мета лабораторної роботи 3](#_Toc81070524)

[2 Завдання 4](#_Toc81070525)

[3 Виконання 7](#_Toc81070526)

[3.1 Псевдокод алгоритмів 7](#_Toc81070527)

[3.2 Часова складність пошуку 7](#_Toc81070528)

[3.3 Програмна реалізація 7](#_Toc81070529)

[3.3.1 Вихідний код 7](#_Toc81070530)

[3.3.2 Приклади роботи 7](#_Toc81070531)

[3.4 Тестування алгоритму 8](#_Toc81070532)

[3.4.1 Часові характеристики оцінювання 8](#_Toc81070533)

[Висновок 9](#_Toc81070534)

[Критерії оцінювання 10](#_Toc81070535)

# 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** |
| 1 | Файли з щільним індексом з перебудовою індексної області, бінарний пошук |
| 2 | Файли з щільним індексом з областю переповнення, бінарний пошук |
| 3 | Файли з не щільним індексом з перебудовою індексної області, бінарний пошук |
| 4 | Файли з не щільним індексом з областю переповнення, бінарний пошук |
| 5 | АВЛ-дерево |
| 6 | Червоно-чорне дерево |
| 7 | B-дерево t=10, бінарний пошук |
| 8 | B-дерево t=25, бінарний пошук |
| 9 | B-дерево t=50, бінарний пошук |
| 10 | B-дерево t=100, бінарний пошук |
| 11 | Файли з щільним індексом з перебудовою індексної області, однорідний бінарний пошук |
| 12 | Файли з щільним індексом з областю переповнення, однорідний бінарний пошук |
| 13 | Файли з не щільним індексом з перебудовою індексної області, однорідний бінарний пошук |
| 14 | Файли з не щільним індексом з областю переповнення, однорідний бінарний пошук |
| 15 | АВЛ-дерево |
| 16 | Червоно-чорне дерево |
| 17 | B-дерево t=10, бінарний пошук |
| 18 | B-дерево t=25, бінарний пошук |
| 19 | B-дерево t=50, бінарний пошук |
| 20 | B-дерево t=100, бінарний пошук |
| 21 | Файли з щільним індексом з перебудовою індексної області, метод Шарра |
| 22 | Файли з щільним індексом з областю переповнення, метод Шарра |
| 23 | Файли з не щільним індексом з перебудовою індексної області, метод Шарра |
| 24 | Файли з не щільним індексом з областю переповнення, метод Шарра |
| 25 | АВЛ-дерево |
| 26 | Червоно-чорне дерево |
| 27 | B-дерево t=10, метод Шарра |
| 28 | B-дерево t=25, метод Шарра |
| 29 | B-дерево t=50, метод Шарра |
| 30 | B-дерево t=100, метод Шарра |

# Виконання

## Псевдокод алгоритмів

…

## Часова складність пошуку

…

## Програмна реалізація

### Вихідний код

class Program

{

static unsafe void Main(string[] args)

{

var tree = new Btree(3);

var insertArray = new[]

{ 5, 7, 15, 40, 2, 1, 3, 10, 12, 80, 200, 500, 150, 90, 20, 50, 16, 14, 13, 9, 38, 89, 101, 102, 201, 202, 91, 88, 30, 18 };

foreach (var elem in insertArray)

tree.Insert(elem);

Console.WriteLine("Search(5): {0}", tree.Search(5));

Console.WriteLine("Search(999): {0}", tree.Search(999));

Console.WriteLine("FindMaxKey(): {0}", tree.FindMaxKey());

Console.WriteLine("Delete(5): {0}", tree.Delete(5));

Console.WriteLine("Search(5): {0}", tree.Search(5));

var deleteArray = new[]

{ 15, 1, 80, 50, 14, 12, 500, 89, 90, 30, 150, 7, 13, 10, 16, 201, 102 };

foreach (var elem in deleteArray)

tree.Delete(elem);

Console.WriteLine("FindMaxKey(): {0}", tree.FindMaxKey());

tree.Print();

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Lab2

{

public class BtreeNode

{

public BtreeNode Parent;

private int T { get; }

public int Length;

public readonly int?[] Keys;

public readonly BtreeNode[] Childrens;

public readonly bool IsLeaf;

public BtreeNode(int t)

{

T = t;

Length = 0;

Keys = new int?[2 \* t - 1];

Childrens = new BtreeNode[2 \* t];

IsLeaf = false;

}

public BtreeNode(int t, bool leaf)

{

T = t;

Length = 0;

Keys = new int?[2 \* t - 1];

Childrens = new BtreeNode[2 \* t];

IsLeaf = leaf;

}

public int? GetFirstKey() => Length == 0 ? null : Keys[0];

public int? GetFirstKeyIndex() => Length == 0 ? null : 0;

public int? GetLastKey() => Length == 0 ? null : Keys[Length - 1];

public int? GetLastKeyIndex() => Length == 0 ? null : Length - 1;

public int? GetLastChildrenIndex()

{

for (var i = Childrens.Length - 1; i >= 0; i--)

if (Childrens[i] != null)

return i;

return null;

}

public BtreeNode GetLastChildren()

{

for (var i = Childrens.Length - 1; i >= 0; i--)

if (Childrens[i] != null)

return Childrens[i];

return null;

}

public int?[] GetHashCodeOfChildrens()

{

var result = new int?[2 \* T];

for (var i = 0; i < 2 \* T; i++)

if (Childrens[i] != null)

result[i] = Childrens[i].GetHashCode();

return result;

}

public (int? key, int? index) FindMaxKey()

{

if (IsLeaf)

return (GetLastKey(), GetLastKeyIndex());

else

{

return GetLastChildren().FindMaxKey();

}

}

public void InsertInOrder(int key)

{

if (Length == 0)

Keys[0] = key;

else

{

var i = Length - 1;

while (i >= 0 && key < Keys[i])

{

Keys[i + 1] = Keys[i];

i--;

}

Keys[i + 1] = key;

}

Length++;

}

public void InsertNonFull(int key)

{

if (IsLeaf)

InsertInOrder(key);

else

{

var i = Length - 1;

while (i >= 0 && Keys[i] > key)

i--;

i++;

if (Childrens[i].Length == 2 \* T - 1)

{

SplitChild(i);

if (key > Keys[i])

i++;

}

Childrens[i].InsertNonFull(key);

}

}

public void SplitChild(int index)

{

var y = Childrens[index];

var z = new BtreeNode(T, y.IsLeaf)

{

Parent = y.Parent,

Length = T - 1

};

y.Length = T - 1;

for (var j = 0; j < T - 1; j++)

{

z.Keys[j] = y.Keys[j + T];

y.Keys[j + T] = null;

}

if (!y.IsLeaf)

for (var j = 0; j < T; j++)

{

z.Childrens[j] = y.Childrens[j + T];

y.Childrens[j + T] = null;

z.Childrens[j].Parent = z;

}

for (var j = Length - 1; j >= index; j--)

Keys[j + 1] = Keys[j];

for (var j = Length; j >= index + 1; j--)

Childrens[j + 1] = Childrens[j];

Childrens[index + 1] = z;

Keys[index] = y.Keys[T - 1];

y.Keys[T - 1] = null;

Length++;

}

public (BtreeNode node, int index) Search(int key)

{

var i = 0;

while (i < Length && key > Keys[i])

i++;

if (i < Length && key == Keys[i])

return (this, i);

if (IsLeaf)

return (null, 0);

return Childrens[i].Search(key);

}

private void MargeTwoNodes(bool leftSibling, int thisNodeIndex)

{

var mergedNode = new BtreeNode(T, IsLeaf) { Parent = this.Parent };

BtreeNode siblingNode;

if (!leftSibling)

{

InsertThisNode();

InsertParentKey();

InsertSibling();

Parent.Childrens[thisNodeIndex] = mergedNode;

MoveParentChildrens(thisNodeIndex + 1);

}

else

{

InsertSibling();

InsertParentKey();

InsertThisNode();

Parent.Childrens[thisNodeIndex - 1] = mergedNode;

MoveParentChildrens(thisNodeIndex);

}

void InsertThisNode()

{

var length = mergedNode.Length;

for (var i = 0; i < Length; i++)

{

mergedNode.Keys[length + i] = Keys[i];

Keys[i] = null;

mergedNode.Length++;

}

for (var i = 0; i < GetLastChildrenIndex(); i++)

{

mergedNode.Childrens[i] = Childrens[i];

Childrens[i] = null;

}

}

void InsertSibling()

{

siblingNode = !leftSibling ? Parent.Childrens[thisNodeIndex + 1] : Parent.Childrens[thisNodeIndex - 1];

var length = mergedNode.Length;

for (var i = 0; i < siblingNode.Length; i++)

{

mergedNode.Keys[length + i] = siblingNode.Keys[i];

siblingNode.Keys[i] = null;

mergedNode.Length++;

}

for (var i = 0; i < siblingNode.GetLastChildrenIndex(); i++)

{

mergedNode.Childrens[length + i] = siblingNode.Childrens[i];

siblingNode.Childrens[i] = null;

}

}

void InsertParentKey()

{

int index;

if (!leftSibling)

index = thisNodeIndex;

else

index = thisNodeIndex - 1;

mergedNode.Keys[mergedNode.Length] = Parent.Keys[index];

for (var i = index; i < 2 \* T - 1; i++)

{

if (Parent.Keys[i] == null)

break;

Parent.Keys[i] = Parent.Keys[i + 1];

}

mergedNode.Length++;

}

void MoveParentChildrens(int startIndex)

{

for (var i = startIndex; i < 2 \* T - 1; i++)

Parent.Childrens[i] = Parent.Childrens[i + 1];

}

}

public void Delete(int index)

{

DeletePhaseOne(index);

DeletePhaseTwo();

}

private int? DeletePhaseOne(int index)

{

return IsLeaf ? DeleteFromLeaf(index) : DeleteFromNoLeaf(index);

}

private int? DeleteFromLeaf(int index)

{

var temp = Keys[index];

for (var i = index; i < Length - 1; i++)

Keys[i] = Keys[i + 1];

Length--;

for (var i = Length; i < 2 \* T - 1; i++)

{

if (Keys[i] == null)

break;

Keys[i] = null;

}

return temp;

}

private int? DeleteFromNoLeaf(int index)

{

var temp = Keys[index];

var (key, \_) = Childrens[index].FindMaxKey();

var (node, i) = Search(key ?? default(int));

Keys[index] = key;

node.Delete(i);

return temp;

}

private void DeletePhaseTwo()

{

if (Parent == null

|| Length >= T - 1)

return;

var thisNodeIndex = Array.IndexOf(Parent.Childrens, this);

if (thisNodeIndex > 0

&& Parent.Childrens[thisNodeIndex - 1].Length > T - 1)

{

MoveKeyOverParent(true);

return;

}

else if (thisNodeIndex < 2 \* T

&& Parent.Childrens[thisNodeIndex + 1] != null

&& Parent.Childrens[thisNodeIndex + 1].Length > T - 1)

{

MoveKeyOverParent(false);

return;

}

if (thisNodeIndex >= 0 && thisNodeIndex < 2 \* T

&& Parent.Childrens[thisNodeIndex + 1] != null

&& Parent.Childrens[thisNodeIndex + 1].Length >= T - 1)

{

MargeTwoNodes(false, thisNodeIndex);

Parent.DeletePhaseTwo();

}

else if (thisNodeIndex > 0 && thisNodeIndex <= 2 \* T

&& Parent.Childrens[thisNodeIndex - 1].Length >= T - 1)

{

MargeTwoNodes(true, thisNodeIndex);

Parent.DeletePhaseTwo();

}

void MoveKeyOverParent(bool leftSibling)

{

int siblingIndex;

int siblingKeyIndex;

int keyIndex;

if (leftSibling)

{

siblingIndex = thisNodeIndex - 1;

siblingKeyIndex = Parent.Childrens[siblingIndex].GetLastKeyIndex() ?? default(int);

keyIndex = thisNodeIndex - 1;

}

else

{

siblingIndex = thisNodeIndex + 1;

siblingKeyIndex = Parent.Childrens[siblingIndex].GetFirstKeyIndex() ?? default(int);

keyIndex = thisNodeIndex;

}

InsertInOrder(Parent.Keys[keyIndex] ?? default(int));

for (var i = GetLastChildrenIndex(); i < 2 \* T - 1; i++)

Childrens[(int)i] = Childrens[(int)i];

Childrens[0] = Parent.Childrens[siblingIndex].GetLastChildren();

Parent.Childrens[siblingIndex].Childrens[GetLastChildrenIndex() ?? default(int)] = null;

Parent.Keys[keyIndex] =

Parent.Childrens[siblingIndex].DeletePhaseOne(siblingKeyIndex);

}

}

public void Print()

{

Console.WriteLine("this: {0}", GetHashCode());

if (Parent != null)

Console.WriteLine("parent: {0}", Parent.GetHashCode());

Console.WriteLine("keys: [{0}]", String.Join(", ", Keys));

Console.WriteLine("childrens: [{0}]", String.Join(", ", GetHashCodeOfChildrens()));

Console.WriteLine("leaf: {0}", IsLeaf);

Console.WriteLine();

foreach (var children in Childrens)

children?.Print();

}

public void PrintRoot()

{

if (Parent == null)

Print();

else

Parent.PrintRoot();

}

}

}

public class Btree

{

public int T { get; }

public BtreeNode Root;

public Btree(int t)

{

if (t <= 2)

throw new ArgumentException("BTree");

T = t;

Root = new BtreeNode(t, true);

}

public void Insert(int key)

{

var root = Root;

if (root.Length == 0)

root.InsertInOrder(key);

else if (root.Length == 2 \* T - 1)

{

var newNode = new BtreeNode(T);

newNode.Childrens[0] = root;

Root = newNode;

root.Parent = newNode;

newNode.SplitChild(0);

newNode.InsertNonFull(key);

}

else

root.InsertNonFull(key);

}

public (BtreeNode node, int index) Search(int key) => Root.Search(key);

public (int? key, int? index) FindMaxKey() => Root.FindMaxKey();

public int? Delete(int key)

{

// Tree is empty

if (Root.Length == 0)

return null;

var (node, index) = Search(key);

// Node not found

if (node == null)

return null;

node.Delete(index);

return key;

}

public void Print()

{

Console.WriteLine("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

Console.WriteLine("Btree");

Console.WriteLine("t: {0}", T);

if (Root == null) return;

Console.WriteLine("\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

Console.WriteLine("R O O T:");

Root.Print();

}

}

### 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 |  |
| 2 |  |
| 3 |  |
| 4 |  |
| … |  |

Висновок

В рамках лабораторної роботи …

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.