БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Лабораторная работа № 15

по дисциплине «Структуры и алгоритмы обработки данных»

Вариант № 4

Выполнил студент: Ефименко Павел Викторович,

группа 981063

Минск 2020

**Условие:**

Ввести 10-15 целых чисел и построить из них В-дерево порядка 5.  Реализовать операции поиска, вставки и удаления элементов В-дерева.

**Листинг:**

using System;

using System.Linq;

using System.Diagnostics;

using System.Collections.Generic;

namespace Lr15

{

public class Unit<TK, TP>

{

// Степень

private int power;

public Unit(int power)

{

this.power = power;

this.Child = new List<Unit<TK, TP>>(power);

this.Recordings = new List<Entry<TK, TP>>(power);

}

public List<Unit<TK, TP>> Child { get; set; }

// Входящие значения

public List<Entry<TK, TP>> Recordings { get; set; }

// Если это узел

public bool IsLeaf => this.Child.Count == 0;

// Проверкам максимального значения

public bool MaxRecordings => this.Recordings.Count == (2 \* this.power) - 1;

// Проверка, достижения минимального количества входных значений

public bool MinRecordings => this.Recordings.Count == this.power - 1;

}

// Блок листа

public class Entry<TK, TP> : IEquatable<Entry<TK, TP>>

{

public TK Key { get; set; }

public TP Pointer { get; set; }

// Проверка эквивалентности значений

public bool Equals(Entry<TK, TP> other)

{

return this.Key.Equals(other.Key) && this.Pointer.Equals(other.Pointer);

}

}

// реализация B-дерева

public class BTree<TK, TP> where TK : IComparable<TK>

{

public BTree(int power)

{

if (power < 2)

{

throw new ArgumentException("Степень не менее 2-ух", "power");

}

this.Root = new Unit<TK, TP>(power);

this.Power = power;

this.Height = 1;

}

// Корень

public Unit<TK, TP> Root { get; private set; }

// Степень

public int Power { get; private set; }

// Высота

public int Height { get; private set; }

public Entry<TK, TP> Search(TK key)

{

// Возвращает узел

return this.SearchInternal(this.Root, key);

}

// Для вывода

public void PrintTree(Unit<TK, TP> root, string indent = "")

{

if (!root.IsLeaf)

{

Console.Write($"{indent}[Root] - ");

// Выводим все значения внутри

foreach (var value in root.Recordings)

{

if (root.Recordings.Last().Key.Equals(value.Key))

{

Console.Write(value.Key + ".\n");

}

else

{

Console.Write(value.Key.ToString() + ',');

}

}

foreach (var leaf in root.Child)

{

// Для каждого потомка запускаем вывод на экран

PrintTree(leaf, indent + " ");

}

}

else

{

if (Root.Equals(root))

{

Console.Write($"{indent}[Root] - ");

foreach (var value in root.Recordings)

{

if (root.Recordings.Last().Key.Equals(value.Key))

{

Console.Write(value.Key + ".\n");

}

else

{

Console.Write(value.Key.ToString() + ',');

}

}

}

else

{

// Если это лист

Console.Write($"{indent}[Leaf] - ");

foreach (var value in root.Recordings)

{

if (root.Recordings.Last().Key.Equals(value.Key))

{

Console.Write(value.Key + ".\n");

}

else

{

Console.Write(value.Key.ToString() + ',');

}

}

}

}

}

// Добавление в дерево

public void Insert(TK newKey, TP newPointer)

{

if (!this.Root.MaxRecordings)

{

// Вставка

this.InsertNonFull(this.Root, newKey, newPointer);

return;

}

/\*

\* Если родительский узел был заполнен, то нужно разбивать.

\* Дальеш до корня, если разбивается корень, то появляется новый корень и глубина дерева увеличивается

\* Вставить ключ в уже заполненный лист нельзя, нужно разбить узел на 2 части

\*/

var oldRoot = this.Root;

// Узел Power

this.Root = new Unit<TK, TP>(this.Power);

// Перемещаем старый корень

this.Root.Child.Add(oldRoot);

// Разбиваем корень

this.SplitChild(this.Root, 0, oldRoot);

// Добалвяем ключ

this.InsertNonFull(this.Root, newKey, newPointer);

this.Height++;

}

// Удаление по внешнему ключу

public void Delete(TK keyToDelete)

{

this.DeleteInternal(this.Root, keyToDelete);

// Если последняя запись root была перемещена в дочерний модуль, то удаляем ее

if (this.Root.Recordings.Count == 0 && !this.Root.IsLeaf)

{

this.Root = this.Root.Child.Single();

this.Height--;

}

}

// Метод удаления

private void DeleteInternal(Unit<TK, TP> unit, TK keyToDelete)

{

var count = unit.Recordings.TakeWhile(entry => keyToDelete.CompareTo(entry.Key) > 0).Count();

// Если нашли ключ в узле, то удаляем из него

if (count < unit.Recordings.Count && unit.Recordings[count].Key.CompareTo(keyToDelete) == 0)

{

this.DeleteKeyFromUnit(unit, keyToDelete, count);

return;

}

// Если не в узле, то удаляем из поддерева

if (!unit.IsLeaf)

{

this.DeleteKey(unit, keyToDelete, count);

}

}

private void DeleteKey(Unit<TK, TP> parentUnit, TK keyToDelete, int subtreeIndexInUnit)

{

var childUnit = parentUnit.Child[subtreeIndexInUnit];

/\*

\* Если удаление происходит из листа, то необходимо проверить, сколько ключей находится в нем

\* Если существует соседний лист, находящийся рядом с ним и имеющий такого же родителя, который содержит больше одного ключа,

\* то выберем ключ из этого соседа, который является разделителем между оставшимися ключами узла соседа и исходного узла

\*/

if (childUnit.MinRecordings)

{

var leftIndex = subtreeIndexInUnit - 1;

// Сосед слева

var leftSibling = subtreeIndexInUnit > 0 ? parentUnit.Child[leftIndex] : null;

// Сосед справа

var rightIndex = subtreeIndexInUnit + 1;

var rightSibling = (subtreeIndexInUnit < parentUnit.Child.Count - 1) ? parentUnit.Child[rightIndex] : null;

// Если сосед слева не пуст и содержит больше t-1 ключа

if (leftSibling != null && leftSibling.Recordings.Count > this.Power - 1)

{

// Перемещение разделителя

childUnit.Recordings.Insert(0, parentUnit.Recordings[subtreeIndexInUnit]);

// Вставка соседа в родительский

parentUnit.Recordings[subtreeIndexInUnit] = leftSibling.Recordings.Last();

// Удаление из соседа

leftSibling.Recordings.RemoveAt(leftSibling.Recordings.Count - 1);

if (!leftSibling.IsLeaf)

{

childUnit.Child.Insert(0, leftSibling.Child.Last());

leftSibling.Child.RemoveAt(leftSibling.Child.Count - 1);

}

}

else if (rightSibling != null && rightSibling.Recordings.Count > this.Power - 1)

{

// Перемещение разделителя

childUnit.Recordings.Add(parentUnit.Recordings[subtreeIndexInUnit]);

// Вставка соседа в родительский

parentUnit.Recordings[subtreeIndexInUnit] = rightSibling.Recordings.First();

// Удаление из соседа

rightSibling.Recordings.RemoveAt(0);

if (!rightSibling.IsLeaf)

{

// Если это не корень, выполняем аналогичную процедуру

childUnit.Child.Add(rightSibling.Child.First());

rightSibling.Child.RemoveAt(0);

}

}

else

{

/\*

\* Если все соседи узла имеют по одному ключу, то объединяем его с соседом, удаляя нужный ключ

\* тот ключ из узла родителя, который был разделителем для этих двух соседей, переместим в новый узел

\*/

if (leftSibling != null)

{

childUnit.Recordings.Insert(0, parentUnit.Recordings[subtreeIndexInUnit]);

var oldRecordings = childUnit.Recordings;

childUnit.Recordings = leftSibling.Recordings;

childUnit.Recordings.AddRange(oldRecordings);

if (!leftSibling.IsLeaf)

{

var oldChildren = childUnit.Child;

childUnit.Child = leftSibling.Child;

childUnit.Child.AddRange(oldChildren);

}

parentUnit.Child.RemoveAt(leftIndex);

parentUnit.Recordings.RemoveAt(subtreeIndexInUnit);

}

else

{

Debug.Assert(rightSibling != null, "У объекта должен быть хотя бы один сосед");

childUnit.Recordings.Add(parentUnit.Recordings[subtreeIndexInUnit]);

childUnit.Recordings.AddRange(rightSibling.Recordings);

if (!rightSibling.IsLeaf)

{

childUnit.Child.AddRange(rightSibling.Child);

}

parentUnit.Child.RemoveAt(rightIndex);

parentUnit.Recordings.RemoveAt(subtreeIndexInUnit);

}

}

}

//Если больше одного ключа, то просто удаляем

this.DeleteInternal(childUnit, keyToDelete);

}

private void DeleteKeyFromUnit(Unit<TK, TP> unit, TK keyToDelete, int keyIndexInUnit)

{

// Если корень одновременно является листом, то есть в дереве всего один узел, просто удаляем ключ из этого узла.

if (unit.IsLeaf)

{

unit.Recordings.RemoveAt(keyIndexInUnit);

return;

}

// Прошлый потомок

var predecessorChild = unit.Child[keyIndexInUnit];

if (predecessorChild.Recordings.Count >= this.Power)

{

// Если в нем количество вхождений больше или равно минимальной степени

// Удаляем предшественника

var predecessor = this.DeletePredecessor(predecessorChild);

// Перемешаем его в исходный узел

unit.Recordings[keyIndexInUnit] = predecessor;

}

else

{

var successorChild = unit.Child[keyIndexInUnit + 1];

if (successorChild.Recordings.Count >= this.Power)

{

//если в нем количество вхождений больше или равно мин. степени

// Удаляем преемника

var successor = this.DeleteSuccessor(predecessorChild);

// Перемешаем его в исходный узел

unit.Recordings[keyIndexInUnit] = successor;

}

else

{

// Добавляем в предшественника все значение прошлого узла

predecessorChild.Recordings.Add(unit.Recordings[keyIndexInUnit]);

predecessorChild.Recordings.AddRange(successorChild.Recordings);

predecessorChild.Child.AddRange(successorChild.Child);

unit.Recordings.RemoveAt(keyIndexInUnit);

unit.Child.RemoveAt(keyIndexInUnit + 1);

// Удаление предшественника

this.DeleteInternal(predecessorChild, keyToDelete);

}

}

}

// Удаление предшественника

private Entry<TK, TP> DeletePredecessor(Unit<TK, TP> unit)

{

if (unit.IsLeaf)

{

var result = unit.Recordings[unit.Recordings.Count - 1];

// Удаляем последнее входное значение и возвращаем его

unit.Recordings.RemoveAt(unit.Recordings.Count - 1);

return result;

}

// Или рекурсивно переходим к потомку

return this.DeletePredecessor(unit.Child.Last());

}

// Удаление преемника

private Entry<TK, TP> DeleteSuccessor(Unit<TK, TP> unit)

{

if (unit.IsLeaf)

{

var result = unit.Recordings[0];

// Удаляем первое входное узначение и возвращаем его

unit.Recordings.RemoveAt(0);

return result;

}

// Или рекурсивно переходим к потомку

return this.DeletePredecessor(unit.Child.First());

}

// Поиск по ключу

private Entry<TK, TP> SearchInternal(Unit<TK, TP> unit, TK key)

{

var i = unit.Recordings.TakeWhile(entry => key.CompareTo(entry.Key) > 0).Count();

if (i < unit.Recordings.Count && unit.Recordings[i].Key.CompareTo(key) == 0)

{

//если нашли нужное знаение, то возврвщаем

return unit.Recordings[i];

}

/\*

\* Если это был лист без потомков, ничего не нашли

\* Если еще не дошли, то продолжаем поиск в потомке

\*/

return unit.IsLeaf ? null : this.SearchInternal(unit.Child[i], key);

}

private void SplitChild(Unit<TK, TP> parentUnit, int unitToBeSplitIndex, Unit<TK, TP> unitToBeSplit)

{

/\*

\* Разбиваем на 2 по 1 ключу, а средний элемент, для которого один из первых ключей меньше его,

\* а ключ последних больше перемещается в родительский узел.

\*/

var newUnit = new Unit<TK, TP>(this.Power);

parentUnit.Recordings.Insert(unitToBeSplitIndex, unitToBeSplit.Recordings[this.Power - 1]);

parentUnit.Child.Insert(unitToBeSplitIndex + 1, newUnit);

newUnit.Recordings.AddRange(unitToBeSplit.Recordings.GetRange(this.Power, this.Power - 1));

unitToBeSplit.Recordings.RemoveRange(this.Power - 1, this.Power);

if (!unitToBeSplit.IsLeaf)

{

newUnit.Child.AddRange(unitToBeSplit.Child.GetRange(this.Power, this.Power));

unitToBeSplit.Child.RemoveRange(this.Power, this.Power);

}

}

// Вставка

private void InsertNonFull(Unit<TK, TP> unit, TK newKey, TP newPointer)

{

// Позиция для вставки

var positionToInsert = unit.Recordings.TakeWhile(entry => newKey.CompareTo(entry.Key) >= 0).Count();

// Если это вершина

if (unit.IsLeaf)

{

// Вствка элемента в лист

unit.Recordings.Insert(positionToInsert, new Entry<TK, TP>() { Key = newKey, Pointer = newPointer });

return;

}

var child = unit.Child[positionToInsert];

if (child.MaxRecordings)

{

// Если лист заполнен

this.SplitChild(unit, positionToInsert, child);

if (newKey.CompareTo(unit.Recordings[positionToInsert].Key) > 0)

{

// Вставляем по возрастанию, если значение больше, то позиция для вставки увеличивется

positionToInsert++;

}

}

// От родителя к листам дальше

this.InsertNonFull(unit.Child[positionToInsert], newKey, newPointer);

}

}

public class Program

{

// Вывод дерева

static void PrintTree(BTree<int, int> tree, string title = null)

{

if (title != null)

{

Console.WriteLine(title);

}

tree.PrintTree(tree.Root);

}

static void Main()

{

// Ключи

var testKeyData = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

// Указатели

var testPointerData = new int[] { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };

// Создание дерева

var BT = new BTree<int, int>(2);

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

{

// Добавление всех значений из массивов.

BT.Insert(testKeyData[i], testPointerData[i]);

}

PrintTree(BT, "Начальное дерево: ");

BT.Delete(10);

PrintTree(BT, "\nДерево после удаления 10 элемента: ");

}

}

}

**Выполнение:**

