ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ОБРАЗОВАНИЯ

*«*САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ

УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»

Институт машиностроения, материалов и транспорта

Высшая школа автоматизации и робототехники

КУРСОВАЯ РАБОТА

**Алгоритм «B\* – Дерево»**

по дисциплине «Объектно-ориентированное программирование»

|  |  |  |
| --- | --- | --- |
| Выполнил  Студент  гр. 3331506/90401 | *(подпись)* | Назаренко И.И. |
| Работу принял | *(подпись)* | Ананьевский М.С. |

Санкт-Петербург

2022 г.

**Введение**

В деревьях поиска, таких как двоичное дерево поиска, AVL дерево, красно-чёрное дерево и т.п. каждый узел содержит только одно значение (ключ) и максимум двое потомков. Однако есть особый тип дерева поиска, который называется B-дерево. В нем узел содержит более одного значения (ключа) и более двух потомков. B-дерево считается сбалансированным по высоте. Для B-дерева всегда задается порядок m, который определяет его ветвистость. Обычно от 50 до 2000.

B-деревья имеют существенные преимущества по сравнению с альтернативными реализациями, когда время доступа к данным узла значительно превышает время, затрачиваемое на обработку этих данных, потому что тогда стоимость доступа к узлу может быть амортизирована по нескольким операциям внутри узла. Обычно это происходит, когда данные узла находятся во вторичном хранилище, например на дисках. Благодаря максимизации количества ключей в каждом внутреннем узле высота дерева уменьшается, а количество обращений к дорогим узлам уменьшается.

**B\*-дерево** — разновидность [B-дерева](https://ru.wikipedia.org/wiki/B-%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE), в которой каждый узел дерева заполнен не менее чем на ⅔ (в отличие от B-дерева, где этот показатель составляет 1/2). Для выполнения требования «заполненность узла не менее 2/3», приходится отказываться от простой процедуры разделения переполненного узла, как в обычном B-дереве. Вместо этого происходит «переливание» в соседний узел. Если же и соседний узел заполнен, то ключи приблизительно поровну разделяются на 3 новых узла. Самой сложной операцией для B\*-дерева является удаление узлов, поэтому такой вид деревьев предпочтителен, когда основные операции, совершаемые над ним – поиск и добавление новых узлов. Основные свойства B\*-дерева:

*Свойство 1:* Глубина всех листьев одинакова.  
 *Свойство 2:* Каждый узел, за исключением корневого, имеет не более m потомков.  
 *Свойство 3:* Каждый узел, за исключением корневого и листьев, имеет не менее потомков.  
 *Свойство 4:* Корневой узел имеет не менее 2 и не более потомков  
 *Свойство 5:* Все ключи в узле должны располагаться в порядке возрастания их значений.

*Свойство 6*: Узлы, не являющиеся листьями и имеющие k потомков, содержать k-1 ключ.

На рисунке 1 изображено B\*-дерево с порядком, равным 4. При таком порядке корневой узел должен иметь от 1 до 5ти ключей, остальные узлы – от 2х до 3х ключей.

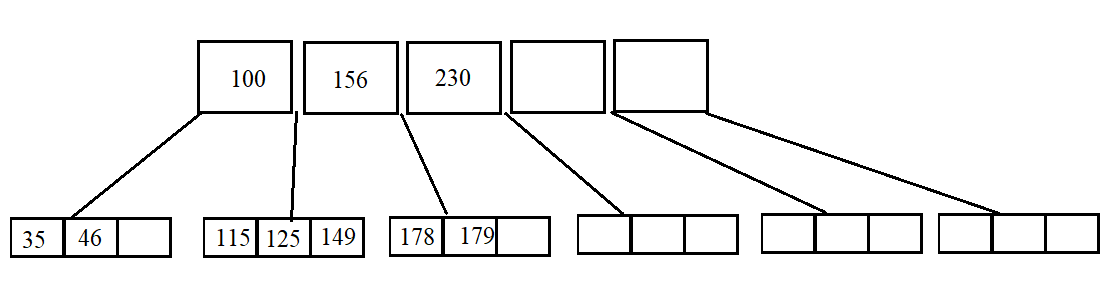


Рисунок 1 – Пример структуры B\*-дерева

**Описание алгоритма**  
Над B\*-деревом можно проводить следующие операции:

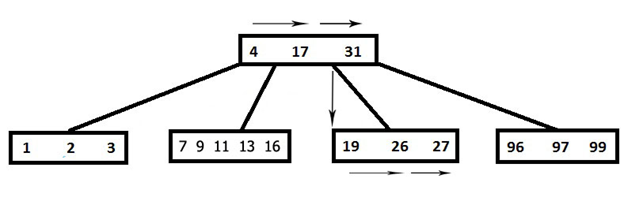
1. Поиск
2. Вставка
3. Удаление

***Алгоритм поиска***

Операция поиска осуществляется довольно быстро благодаря сбалансированности дерева

Поиск в B-дереве очень схож с поиском в бинарном дереве, только здесь мы должны сделать выбор пути к потомку не из 2 вариантов, а из нескольких. В остальном — никаких отличий. На рисунке ниже показан поиск ключа 27. Поясним иллюстрацию (и соответственно стандартный алгоритм поиска):

* Идем по ключам корня, пока меньше необходимого. В данном случае дошли до 31.
* Спускаемся к ребенку, который находится левее этого ключа.
* Идем по ключам нового узла, пока меньше 27. В данном случае – нашли 27 и остановились.

  
  
 Операция поиска выполняется за время O(t logt n), где t – минимальная степень. Важно здесь, что дисковых операций мы совершаем всего лишь O(logt n)!

***Алгоритм добавления***

  В этом алгоритме при переполнении листа обязательно используется переливание на одну из соседних страниц. Когда соседние листы заполнены, выполняется расщепление не одной страницы, а двух соседних страниц. На новый лист переносится треть ключей с каждого из двух заполненных листов, так что каждый лист B\*-дерева будет заполнен по крайней мере на 2/3, а не на 1/2, как для обычного B-дерева. Корневой лист B\*-дерева должен при этом быть большего размера, чем остальные, и он может быть заполнена не более чем на 4/3 от объема обычного листа. При расщеплении корня он дает два обычных листа, заполненных на 2/3, а новый корень, как и для B-деревьев, содержит только один ключ.

Рассмотрим примеры добавления узлов в B\*-дерево. Будем рассматривать пример для порядка m=4. В таком случае в корне должно быть от 1 до 4 ключей, в остальных узлах от 2х до 3х ключей.

**1 сценарий**

У нас имеется узел, который является листом и в нем еще есть свободные места. Тогда просто идем по порядку, сравнивая значения и вставляем наш ключ в подходящее место.



Рисунок 2 – до добавления



Рисунок 3 – после добавления

**2 сценарий**

У нас имеется корневой узел, он является листом и он полностью заполнен. Хотим добавить еще ключ с номером 5.

**

Рисунок 4 – до добавления

Выделяем среднее значение из имеющихся и выносим этот ключ наверх, создавая новый узел. Оставшиеся значения раскидываем в 2 узла, которые являются потомками узла, образованного ранее.

**

Рисунок 5 – после добавления

**3 сценарий**

У нас имеется полностью заполненный узел, в который нам нужно добавить еще ключ 3.

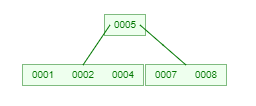


Рисунок 6 – до добавления

Вместо разделения узла, как это делается с обычным B-деревом, мы смотрим, есть ли место в правом соседнем узле, и если место есть, то перераспределяем ключи между этими узлами. Если в правом узле нету места, то проверяем таким же образом левый.

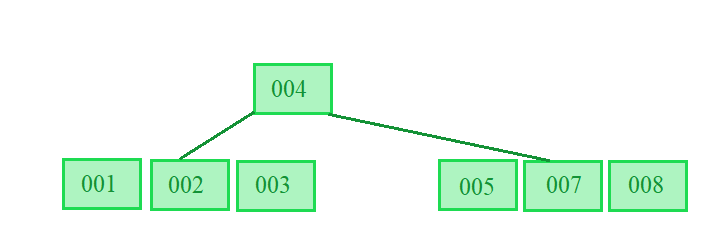


Рисунок 7 – после добавления

Если же нам нужно добавить ключ, а узел и его соседи заполнены, а в родительском узле еще есть свободное место, тогда происходит разделение 2х этих узлов одновременно, с добавлением ключей к родительскому узлу.

Принцип такой: мы выстраиваем в порядке возрастания все имеющиеся ключи в 2х узлах, родительский ключ, и ключ, который мы хотим добавить. Добавляем в первый дочерний узел первые ключей. Следующий ключ будет первым родительским ключом. Во второй дочерний узел заливаем следующие ключей. Следующий ключ будет вторым родительским ключом. Добавляем оставшиеся ключи в третий дочерний узел. Готово, мы разделили 2 заполненных узла на 3 узла, наполненность которых не меньше 2/3. Добавим для примера ключ 6.

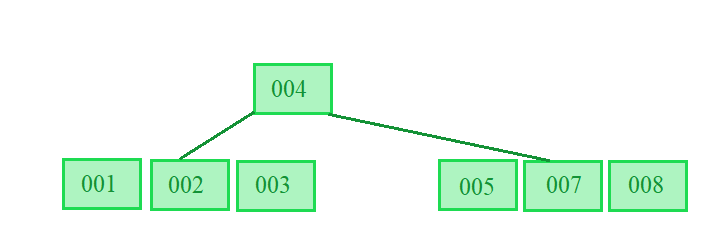


Рисунок 8 – до добавления

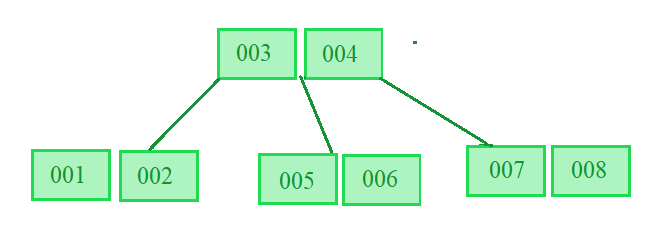


Рисунок 9 - после добавления

**4 сценарий**

Если нам нужно добавить ключ, а узел, его сосед и родительский узел заполнены, то происходит разделение родительского узла. Добавим для примера ключ 7.

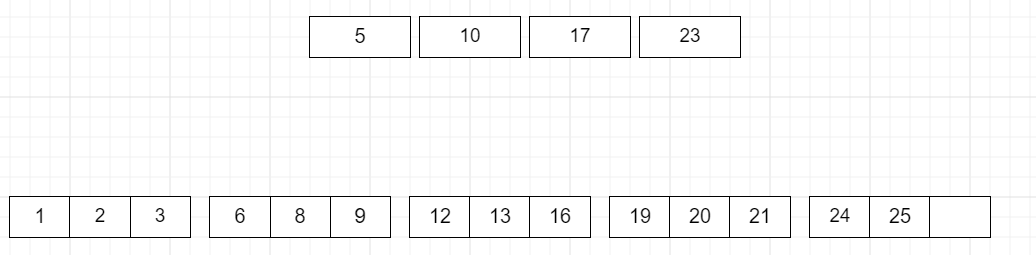


Рисунок 10 – до добавления

Рассмотрим выделенную область. Проделываем условно все операции из предыдущего сценария. В итоге у нас появился лишний ключ в родительском узле.

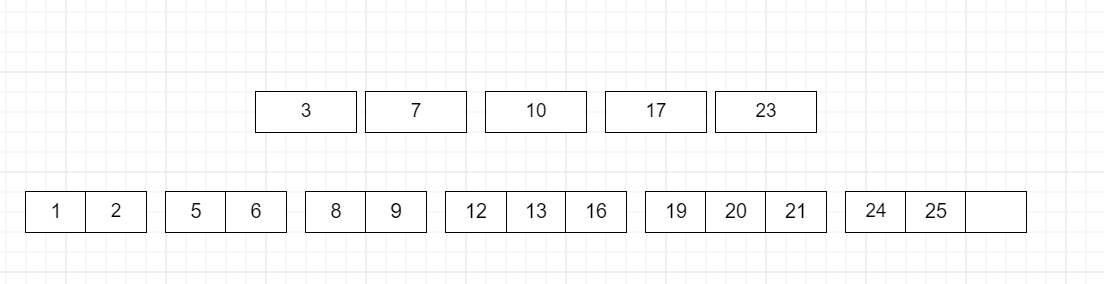


Рисунок 11 - Выделился лишний ключ

После этого перестраиваем дерево, выделяя средний по значению узел наверх. Меняем указатели.

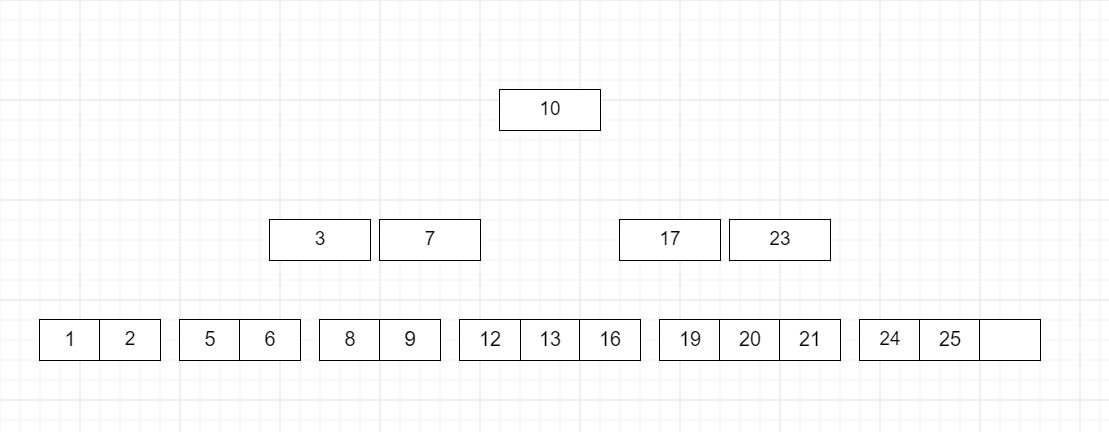


Рисунок 11 – Перестроенное дерево

***Алгоритм удаления***

**1 сценарий**

Ключ, который нам нужно удалить, находится в листе и при удалении ключа лист останется заполненным на 2/3. Тогда мы можем просто удалить ключ. Например, удалим значение 4.

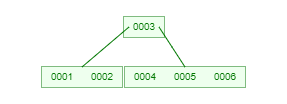


Рисунок 12 – Дерево до удаления

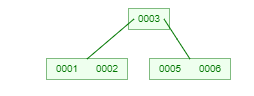


Рисунок 13 – Дерево после удаления

**2 сценарий**

Значение надо удалять из листа, а он останется заполнен меньше, чем на 2/3 после простого удаления. В таком случае смотрим на соседей слева и справа. Если в них заполнение более плотное, тогда можем удалить нужный ключ, а после выполнить переливание между узлами. Допустим, хотим удалить значение 4.

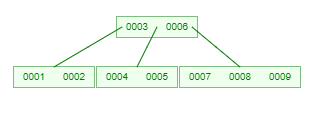


Рисунок 14 – Дерево до удаления

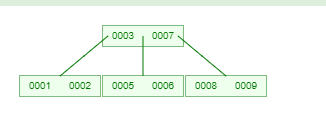


Рисунок 15 – Дерево после удаления

**3 сценарий**

У узла минимальное количество ключей, тогда можем, удаляя, объединить 2 соседних узла, выполняя переливание с родительским узлом. Удалим, например, значение 5.

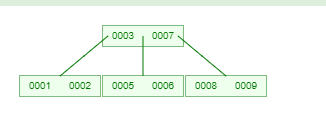


Рисунок 14 – Дерево до удаления

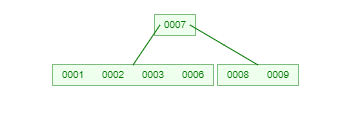


Рисунок 15 – Дерево после удаления

**4 сценарий**

Если удаляем значение из узла, который не является листом, тогда нам нужно найти дочерний узел, в котором допустимое количество ключей, чтобы один можно было удалить. Тогда выполняем переливание.

**5 сценарий**

Если узел не является листом, у его дочерних узлов не имеется лишних мест. Если кол-во дочерних узлов нельзя уменьшить, то обращаемся к родительскому узлу, выполняем переливание между ним и его другими дочерними узлами. Удалим, например, узел 18.

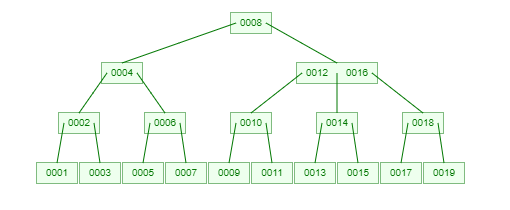


Рисунок 16 – Дерево до удаления

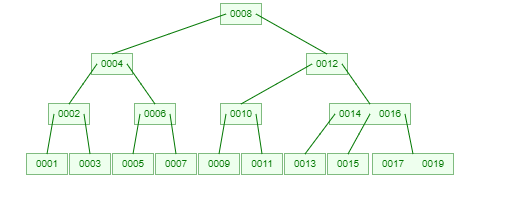


Рисунок 17 – Дерево после удаления

**7 сценарий**

У родительского или дочерних узлов уже нечего занимать, тогда придется уменьшать высоту дерева и полностью его перестраивать. Например, удалим ключ 6.

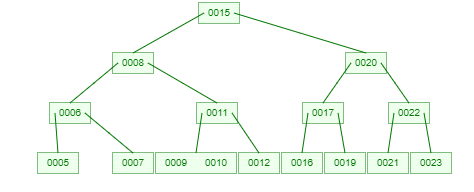


Рисунок 16 – Дерево до удаления

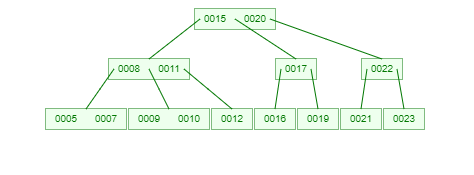


Рисунок 17 – Дерево после удаления

**Временная сложность алгоритма**

Временная сложность заявлена, как , где n – количество узлов, t – степень ветвления.

Построим графики для добавления, удаления элементов и поиска по дереву.

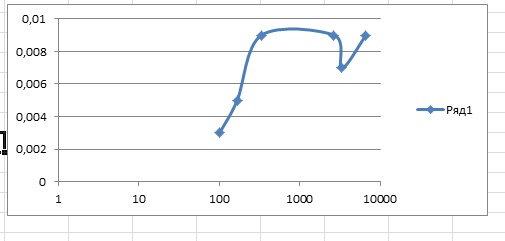


Рисунок 18 – Временная сложность добавления

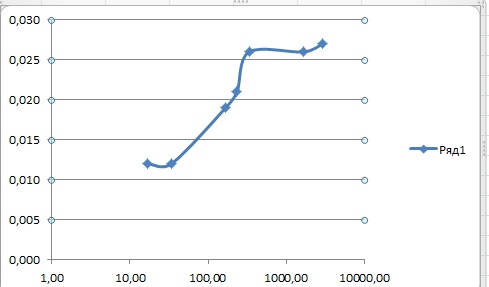


Рисунок 19 – Временная сложность поиска

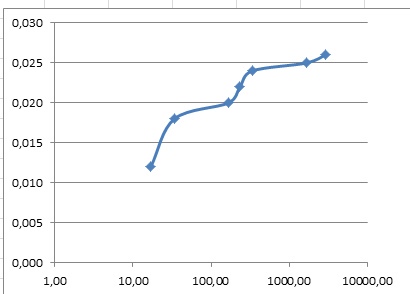


Рисунок 20 – Временная сложность удаления

Графики пока не успеваю выйти на логарифмический уровень, скорее всего для этого нужно большие объемы данных, но в принципе уже прослеживается некая зависимость: графики находятся примерно на одном уровне, потом скачком переходят на следующий, связано это с увеличением высоты дерева.

**Список литературы**

1. Дональд Кнут. 3. Сортировка и поиск // Искусство программирования

2. B-дерево [электронный ресурс] // <https://ru.wikipedia.org/wiki/B-%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE>

3. B-Tree Vizualization [Электронный ресурс] // https://www.cs.usfca.edu/~galles/visualization/BTree.html

4. B\*-Trees implementation in C++ // <https://www.geeksforgeeks.org/b-trees-implementation-in-c/>

5. Сбалансированное дерево поиска B-tree [электронный ресурс] // <https://habr.com/ru/post/337594/>

6. B-Tree [электронный ресурс] // <https://habr.com/ru/post/114154/>

Приложение 1 – заголовочный файл

#include **<iostream>** *//std::cout, std::endl*#include **<fstream>** *//std::ofstream, std::ifstream*#include **<cmath>** *//floor, ceil*#include **<algorithm>** *//std::random\_shuffle*#include **<list>** *//std::list*#include **<queue>** *//std::queue*#include **<vector>** *//std::vector*#include **<array>** *//std::array*#ifndef **UNTITLED23\_BSTAR\_H**#define **UNTITLED23\_BSTAR\_H  
  
class** Node;  
**class** BStarTree  
{  
  
**public**:  
  
 BStarTree (**int** order); *// BStarTree with bfactor* ~BStarTree();  
 **void** empty(); *//to delete all keys* **bool** add(**double** val);  
 **bool** erase(**double** val);  
 **bool** find(**double** val) **const**;  
 **void** print() **const**;  
 **unsigned** maxKeysNNode() **const** { **return** maxKeysNormalNode; }  
 **unsigned** maxKeysRNode() **const** { **return** maxKeysRootNode; }  
 **unsigned** minKeysNNode() **const** { **return** minKeysNormalNode; }  
 **unsigned** minKeysRNode() **const** { **return** minKeysRootNode; }  
**private**:  
 **const int** ORDER; *//order of the tree* **unsigned** id; *//id of the next element of the tree.* Node\* root;  
 **unsigned** maxKeysNormalNode;  
 **unsigned** maxKeysRootNode;  
 **unsigned** minKeysNormalNode;  
 **unsigned** minKeysRootNode;  
 **unsigned** keysSplitChild1;  
 **unsigned** keysSplitChild2;  
 **unsigned** keysSplitChild3;  
 **void** handleOverload(Node\* overloadedNode);  
 **void** handleUnderload(Node\* underloadedNode);  
 **bool** find(**double** val, Node\*& node) **const**;  
 Node\* findPlaceAdd(**double** val) **const**;  
 Node\* findPlaceErase(**double** val) **const**;  
 **bool** searchSpace(Node\* node);  
 **bool** searchSpaceErase(Node\* node);  
 **bool** isLeftmost(Node\* node) **const**;  
 **bool** isRightmost(Node\* node) **const**;  
 **bool** areLeftSiblingsFull(Node\* node) **const**;  
 **bool** areRightSiblingsFull(Node\* node) **const**;  
 **bool** areLeftSiblingsAtMinimum(Node\* node) **const**;  
 **bool** areRightSiblingsAtMinimum(Node\* node) **const**;  
 **auto** rotateLeft(Node\* node);  
 **auto** rotateRight(Node\* node);  
 **void** rotateLeftAdd(Node\* node);  
 **void** rotateRightAdd(Node\* node);  
 **void** rotateLeftErase(Node\* node);  
 **void** rotateRightErase(Node\* node);  
 **void** splitRoot();  
 **void** splitLeft(Node\* node);  
 **void** splitRight(Node\* node);  
 **void** mergeRootChildren(Node\* nodeChildren);  
 **void** merge(Node\* node);  
 Node\* getGreaterMinor(Node\* node, **double** val) **const**;  
 **auto** getIterator(Node\* node) **const**;  
};  
  
  
  
**class** Node  
{  
**public**:  
 *//use the default of the following* Node() = **default**; *//constructor* **virtual** ~Node() = **default**;  
 Node(BStarTree **const** \* tree, Node\* ancestor = **nullptr**):tree(tree), ancestor(ancestor) {}  
  
 *//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Pure virtual function\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\** **virtual bool** isOverloaded() **const** = 0;  
 **virtual bool** isFull() **const** = 0;  
 **virtual bool** isUnderloaded() **const** = 0;  
 **virtual bool** isAtMinimum() **const** = 0;  
 **virtual bool** isRoot() **const** = 0;  
 **void** print() **const**;  
 **void** addItem(**double** item);  
 **void** addChild(Node\* child);  
 **double** popFrontKey();  
 **double** popBackKey();  
 Node\* popFrontChild();  
 Node\* popBackChild();  
 **void** putKeys(std::list<**double**>& takeList, **int** limit);  
 **void** putChildren(std::list<Node\*>& takeList, **int** limit);  
 **unsigned** getId() **const** { **return** id; }  
 std::size\_t getNumKeys() **const** { **return** keysList.size(); }  
 Node\* getAncestor() **const** { **return** ancestor; }  
 std::list<**double**> **const** & keys() **const** { **return** keysList; }  
 std::list<**double**>& keys() { **return** keysList; }  
 std::list<Node\*> **const** & children() **const** { **return** childList; }  
 std::list<Node\*>& children() { **return** childList; }  
 **bool** isLeaf() { **return this**->children().empty(); }  
  
**protected**:  
 **unsigned** id; *//id to identify nodes* BStarTree **const** \* tree; *//pointer to the tree* std::list<**double**> keysList; *// Pointer for allocating dynamic memory store* std::list<Node\*> childList;  
 Node\* ancestor;  
  
};  
  
  
**bool** compareKeyNodes(Node\* nodeA, Node\* nodeB);  
  
  
  
  
  
  
  
  
  
  
**class** NormalNode : **public** Node *// Node is the base class, all public and*{ *// protected members of Node can be accessed***public**: *// by NormalNode  
  
 //use the default of the following* NormalNode() = **default**; *//constructor* **virtual** ~NormalNode() = **default**;  
  
 NormalNode(BStarTree **const** \* tree, Node\* ancestor = **nullptr**, **unsigned** id = 0);  
  
 **void** setAncestor(Node\* newAncestor) { **this**->ancestor = newAncestor; }  
  
 *//Definitions of Node class pure virtual functions* **virtual bool** isOverloaded() **const**;  
 **virtual bool** isFull() **const**;  
 **virtual bool** isUnderloaded() **const**;  
 **virtual bool** isAtMinimum() **const**;  
 **bool** isRoot() **const** { **return false**; }  
};  
  
  
  
  
  
  
  
  
  
  
**class** RootNode : **public** Node *// Node is the base class, all public and  
 // protected members of Node can be accessed  
 // by RootNode*{  
**public**:  
  
 *//use the default of the following* RootNode() = **default**; *//constructor* **virtual** ~RootNode() = **default**;  
 RootNode(BStarTree **const** \* tree, Node\* ancestor = **nullptr**, **unsigned** id = 0);  
  
 **virtual bool** isOverloaded() **const**;  
 **virtual bool** isFull() **const**;  
 **virtual bool** isUnderloaded() **const**;  
 **virtual bool** isAtMinimum() **const**;  
 **bool** isRoot() **const** { **return true**; }  
};  
  
  
  
**void** menu();  
**void** addMenu(BStarTree& tree);  
**void** eraseMenu(BStarTree& tree);  
**void** findMenu(BStarTree& tree);  
  
  
  
  
#endif *//UNTITLED23\_BSTAR\_H*

Приложение 2 – Основной код

#include **<iostream>**#include **"BStar.h"**BStarTree::BStarTree (**int** order): ORDER(order < 4 ? 4 : order), id(1),  
 root(**new** RootNode(**this**, **nullptr**, id))  
{  
 ++id;  
  
 maxKeysNormalNode = ORDER - 1;  
 maxKeysRootNode = 2.0 \* floor( (2.0\*ORDER - 2.0) / 3.0 );  
 minKeysNormalNode = ceil( (2.0\*ORDER-1.0) / 3.0 ) - 1.0;  
 minKeysRootNode = 1;  
 keysSplitChild1 = floor( (2.0\*ORDER - 2.0)/3.0 );  
 keysSplitChild2 = floor( (2.0\*ORDER - 1.0)/3.0 );  
 keysSplitChild3 = floor( 2.0\*ORDER/3.0 );  
}  
  
  
BStarTree::~BStarTree()  
{  
 empty();  
 **delete** root;  
}  
  
  
**void** BStarTree::empty()  
{  
 Node\* currentNode;  
 std::queue<Node\*> nodeQueue;  
  
 **for**(Node\* child : root->children()) nodeQueue.push(child);  
  
 **while** (!nodeQueue.empty()) {  
 currentNode = nodeQueue.front();  
 nodeQueue.pop();  
  
 **for**(Node\* child : currentNode->children()){  
 nodeQueue.push(child);  
 }  
  
 **delete** currentNode;  
 }  
  
 root->keys().clear();  
 root->children().clear();  
 id = 2; *//the next node id will be two*}  
  
  
**bool** BStarTree::add(**double** val)  
{  
 **bool** added;  
 Node\* nodeAdd = **nullptr**; *//Node where it will add the number if the number  
 //doesn't exist in the tree.* Node\* currentNode;  
  
 nodeAdd = findPlaceAdd(val);  
 **if** (nodeAdd != **nullptr**) {  
 nodeAdd->addItem(val);  
 currentNode = nodeAdd;  
 **while** (currentNode != **nullptr** && currentNode->isOverloaded()) {  
 handleOverload(currentNode);  
 currentNode = currentNode->getAncestor();  
 }  
  
 added = **true**;  
 }**else**{  
 added = **false**;  
 }  
  
 **return** added;  
}  
  
  
**void** BStarTree::handleOverload(Node\* overloadedNode)  
{  
 **if** (!overloadedNode->isRoot()) {  
 **if** (!**this**->searchSpace(overloadedNode)) {  
 **if** (!**this**->isLeftmost(overloadedNode)) {  
 splitLeft(overloadedNode); *//This split every node with their left sibling* }**else**{  
 splitRight(overloadedNode); *//This split the leftmost node with his right sibling  
 //because he has no left sibling* }  
 }  
 }**else**{  
 splitRoot();  
 }  
}  
  
  
**bool** BStarTree::erase(**double** val)  
{  
 **bool** erased;  
 Node\* nodeErase = **nullptr**; *//Node where it will add the number if the number  
 //doesn't exist in the tree.* Node\* currentNode;  
  
 nodeErase = findPlaceErase(val);  
 **if** (nodeErase == **nullptr**) {  
 **return false**;  
 }  
  
 currentNode = nodeErase;  
  
 **if** (!nodeErase->isLeaf()) {  
 currentNode = getGreaterMinor(nodeErase, val);  
 nodeErase->addItem(currentNode->popBackKey());  
 }  
 nodeErase->keys().remove(val);  
  
 Node\* ancestor;  
 **while** (currentNode != root && currentNode->isUnderloaded()) {  
 ancestor = currentNode->getAncestor();  
 handleUnderload(currentNode);  
 currentNode = ancestor;  
 }  
  
 erased = **true**;  
  
 **return** erased;  
}  
  
  
**void** BStarTree::handleUnderload(Node\* underloadedNode)  
{  
 **if**(!searchSpaceErase(underloadedNode)){  
 **if** (!underloadedNode->getAncestor()->isRoot()) {  
 merge(underloadedNode); *//inside of this method it checks if the node is leftmost  
 //rightmost or none of those two* }**else**{  
 mergeRootChildren(underloadedNode);  
 }  
 }  
}  
  
  
**bool** BStarTree::find(**double** val) **const**{  
 **return** findPlaceAdd(val) == **nullptr** ? **true** : **false**;  
}  
  
  
**bool** BStarTree::find(**double** val, Node\*& node) **const**{  
 node = root;  
 std::list<Node\*>::iterator child;  
  
 **while**(!node->isLeaf()){  
 child = node->children().begin();  
 **for**(**auto** key = node->keys().begin(); \*key <= val && key != node->keys().end(); ++key){  
 **if**(\*key == val) **return true**;  
 ++child;  
 }  
 **if**(!node->isLeaf()){  
 node = \*child;  
 }  
 }  
  
 *//search if the value to add is in the leaf node about to be returned* **for**(**auto** key = node->keys().begin(); \*key <= val && key != node->keys().end(); ++key){  
 **if**(\*key == val) **return true**;  
 }  
  
 **return false**;  
}  
  
*//this search can be optimized because when searching a node there is no need to keep  
//searching for a value once the values of the node are bigger than the searched value.*Node\* BStarTree::findPlaceAdd(**double** val) **const**{  
 Node\* node = **nullptr**;  
 **if**(!find(val, node)){  
 **return** node;  
 }  
  
 **return nullptr**;  
}  
  
*// can probably be more optimized*Node\* BStarTree::findPlaceErase(**double** val) **const**{  
 Node\* node = **nullptr**;  
 **if**(find(val, node)){  
 **return** node;  
 }  
  
 **return nullptr**;  
}  
  
  
**bool** BStarTree::searchSpace(Node\* node)  
{  
 **bool** foundSpace;  
 foundSpace = **true**;  
  
 **if** (**this**->areLeftSiblingsFull(node)){  
 **if** (**this**->areRightSiblingsFull(node)) {  
 foundSpace = **false**;  
 }**else**{  
 **this**->rotateRightAdd(node);  
 }  
 }**else**{  
 **this**->rotateLeftAdd(node);  
 }  
  
 **return** foundSpace;  
}  
  
  
**bool** BStarTree::searchSpaceErase(Node\* node)  
{  
 **bool** foundSpace;  
 foundSpace = **true**;  
  
 **if** (**this**->areLeftSiblingsAtMinimum(node)){  
 **if** (**this**->areRightSiblingsAtMinimum(node)) {  
 foundSpace = **false**;  
 }**else**{  
 **this**->rotateRightErase(node);  
 }  
 }**else**{  
 **this**->rotateLeftErase(node);  
 }  
  
 **return** foundSpace;  
  
}  
  
  
**bool** BStarTree::areLeftSiblingsFull(Node\* node) **const**{  
 Node\* ancestor = node->getAncestor();  
 **bool** nodeIsFull = **true**;  
  
 *//this checks all the nodes before the received one to see if at least one of them is  
 //not full* **for**(**auto** leftSibling = ancestor->children().begin(); \*leftSibling != node; ++leftSibling){  
 **if** ( !(\*leftSibling)->isFull() ) {  
 nodeIsFull = **false**;  
 **break**;  
 }  
 }  
  
 **return** nodeIsFull;  
}  
  
  
**bool** BStarTree::areRightSiblingsFull(Node\* node) **const**{  
 Node\* ancestor = node->getAncestor();  
 **bool** nodeIsFull = **true**;  
  
 **for**(**auto** rightSibling = ancestor->children().rbegin(); \*rightSibling != node; ++rightSibling){  
 **if** ( !(\*rightSibling)->isFull() ) {  
 nodeIsFull = **false**;  
 **break**;  
 }  
 }  
  
 **return** nodeIsFull;  
}  
  
  
**bool** BStarTree::areLeftSiblingsAtMinimum(Node\* node) **const**{  
 Node\* ancestor = node->getAncestor();  
 **bool** nodeIsAtMinimum = **true**;  
  
 *//this checks all the nodes before the received one to see if at least one of them is  
 //not full* **for**(**auto** leftSibling = ancestor->children().begin(); \*leftSibling != node; ++leftSibling){  
 **if** ( !(\*leftSibling)->isAtMinimum() ) {  
 nodeIsAtMinimum = **false**;  
 **break**;  
 }  
 }  
  
 **return** nodeIsAtMinimum;  
}  
  
  
**bool** BStarTree::areRightSiblingsAtMinimum(Node\* node) **const**{  
 Node\* ancestor = node->getAncestor();  
 **bool** nodeIsAtMinimum = **true**;  
  
 **for**(**auto** rightSibling = ancestor->children().rbegin(); \*rightSibling != node; ++rightSibling){  
 **if** ( !(\*rightSibling)->isAtMinimum() ) {  
 nodeIsAtMinimum = **false**;  
 **break**;  
 }  
 }  
  
 **return** nodeIsAtMinimum;  
}  
  
  
**bool** BStarTree::isLeftmost(Node\* node) **const**{  
 **return** \*node->getAncestor()->children().begin() == node ? **true** : **false**;  
}  
  
  
**bool** BStarTree::isRightmost(Node\* node) **const**{  
 **return** \*node->getAncestor()->children().rbegin() == node ? **true** : **false**;  
}  
  
  
**auto** BStarTree::getIterator(Node\* node) **const**{  
 **auto** it = node->getAncestor()->children().begin();  
 **while**(\*it != node){  
 ++it;  
 }  
  
 **return** it;  
}  
  
  
  
**auto** BStarTree::rotateLeft(Node\* node)  
{  
 Node \*ancestor, \*leftSibling, \*child;  
 std::list<Node\*>::iterator nodeIt;  
  
 ancestor = node->getAncestor();  
 **auto** ancestorKey = ancestor->keys().begin();  
 **for**(nodeIt = next(ancestor->children().begin()); \*nodeIt != node; ++nodeIt){  
 ++ancestorKey;  
 }  
 leftSibling = \*prev(nodeIt);  
  
 *//key rotation* leftSibling->keys().push\_back(\*ancestorKey);  
 \*ancestorKey = node->popFrontKey();  
  
 *//child rotation* **if**(!node->children().empty()){  
 child = node->popFrontChild();  
 **dynamic\_cast**<NormalNode\*>(child)->setAncestor(leftSibling);  
 leftSibling->children().push\_back(child);  
 }  
  
 **return** nodeIt;  
}  
  
  
**auto** BStarTree::rotateRight(Node\* node)  
{  
 Node \*ancestor, \*rightSibling, \*child;  
 std::list<Node\*>::iterator nodeIt;  
  
 ancestor = node->getAncestor();  
 **auto** ancestorKey = ancestor->keys().begin();  
 **for**(nodeIt = ancestor->children().begin(); \*nodeIt != node; ++nodeIt){  
 ++ancestorKey;  
 }  
 rightSibling = \*next(nodeIt);  
  
 *//key rotation* rightSibling->keys().push\_front(\*ancestorKey);  
 \*ancestorKey = node->popBackKey();  
  
 *//child rotation* **if**(!node->children().empty()){  
 child = node->popBackChild();  
 **dynamic\_cast**<NormalNode\*>(child)->setAncestor(rightSibling);  
 rightSibling->children().push\_front(child);  
 }  
  
 **return** nodeIt;  
}  
  
  
**void** BStarTree::rotateLeftAdd(Node\* node)  
{  
 Node \*currentNode;  
  
 currentNode = node;  
 **do** {  
 currentNode = \*prev(rotateLeft(currentNode));  
 } **while**(!isLeftmost(currentNode) && currentNode->isOverloaded());  
}  
  
  
**void** BStarTree::rotateRightAdd(Node\* node)  
{  
 Node \*currentNode;  
 currentNode = node;  
 **do** {  
 currentNode = \*next(rotateRight(currentNode));  
 } **while**(!isRightmost(currentNode) && currentNode->isOverloaded());  
}  
  
  
**void** BStarTree::rotateLeftErase(Node\* node)  
{  
 **auto** currentNode = getIterator(node);  
  
 **do** {  
 currentNode = prev(currentNode);  
 rotateRight(\*currentNode);  
 } **while**(!isLeftmost(\*currentNode) && (\*currentNode)->isUnderloaded());  
}  
  
  
**void** BStarTree::rotateRightErase(Node\* node)  
{  
 **auto** currentNode = getIterator(node);  
  
 **do** {  
 currentNode = next(currentNode);  
 rotateLeft(\*currentNode);  
 } **while**(!isRightmost(\*currentNode) && (\*currentNode)->isUnderloaded());  
}  
  
  
**void** BStarTree::splitRoot()  
{  
 Node \*child1, \*child2;  
  
 child1 = **new** NormalNode(**this**, root, id++);  
 child2 = **new** NormalNode(**this**, root, id++);  
  
 **unsigned** limitRoot = maxKeysRootNode /2;  
 child1->putKeys(root->keys(), limitRoot);  
  
 **double** auxKey = root->popFrontKey();  
  
 child2->putKeys(root->keys(), limitRoot);  
  
 root->keys().push\_front(auxKey);  
  
 **unsigned** limitForChild1 = child1->keys().size() + 1;  
 **unsigned** limitForChild2 = child2->keys().size() + 1;  
  
 **if**(!root->isLeaf()){  
 child1->putChildren(root->children(), limitForChild1);  
 child2->putChildren(root->children(), limitForChild2);  
 }  
  
 root->children().push\_back(child1);  
 root->children().push\_back(child2);  
}  
  
  
**void** BStarTree::splitLeft(Node\* node)  
{  
 Node \*leftSibling, \*ancestor;  
 **double** ancestorKeyCopy;  
 std::list<Node\*>::iterator nodeIt;  
  
 ancestor = node->getAncestor();  
 **auto** ancestorKey = ancestor->keys().begin();  
 **for**(nodeIt = next(ancestor->children().begin()); \*nodeIt != node; ++nodeIt){  
 ++ancestorKey;  
 }  
  
 ancestorKeyCopy = \*ancestorKey;  
 ancestor->keys().erase(ancestorKey);  
  
 leftSibling = \*prev(nodeIt);  
  
 *//moves all the keys of the left sibling to an auxiliar list, leaving the sibling empty* std::list<**double**> auxList(std::move(leftSibling->keys()));  
 auxList.push\_back(ancestorKeyCopy);  
 *//moves all the keys of the node to the auxiliar list, leaving the node empty* auxList.merge(node->keys());  
  
 Node \*newNode; *//new node that goes in the middle of the current node and its left sibling* newNode = **new** NormalNode(**this**, ancestor, id++);  
  
 **auto** putKeyAncestor = [&ancestor, &auxList](){  
 ancestor->addItem( auxList.front() );  
 auxList.pop\_front();  
 };  
  
 *//accommodate keys in the nodes* **unsigned** limitOne = keysSplitChild1;  
 leftSibling->putKeys(auxList, limitOne);  
  
 putKeyAncestor();  
  
 **unsigned** limitTwo = keysSplitChild2;  
 newNode->putKeys(auxList, limitTwo);  
  
 putKeyAncestor();  
  
 **unsigned** limitThree = keysSplitChild3;  
 node->putKeys(auxList, limitThree);  
  
 *//accommodate children in the nodes.* std::list<Node\*> auxListChildren(std::move(leftSibling->children()));  
 auxListChildren.merge(node->children(), compareKeyNodes);  
  
 **if**(!auxListChildren.empty()){  
 leftSibling->putChildren(auxListChildren, limitOne+1);  
 newNode->putChildren(auxListChildren, limitTwo+1);  
 node->putChildren(auxListChildren, limitThree+1);  
 }  
  
 ancestor->addChild(newNode);  
}  
  
  
**void** BStarTree::splitRight(Node\* node)  
{  
 Node \*rightSibling, \*ancestor;  
 **double** ancestorKeyCopy;  
 std::list<Node\*>::iterator nodeIt;  
  
 ancestor = node->getAncestor();  
 **auto** ancestorKey = ancestor->keys().begin();  
 **for**(nodeIt = ancestor->children().begin(); \*nodeIt != node; ++nodeIt){  
 ++ancestorKey;  
 }  
  
 ancestorKeyCopy = \*ancestorKey;  
 ancestor->keys().erase(ancestorKey);  
  
 rightSibling = \*next(nodeIt);  
  
 *//moves all the keys of the node to the auxiliar list, leaving the node empty* std::list<**double**> auxList(std::move(node->keys()));  
 auxList.push\_back(ancestorKeyCopy);  
 *//moves all the keys of the right sibling to an auxiliar list, leaving the sibling empty* auxList.merge(rightSibling->keys());  
  
 Node \*newNode; *//new node that goes in the middle of the current node and its right sibling* newNode = **new** NormalNode(**this**, ancestor, id++);  
  
 **auto** putKeyAncestor = [&ancestor, &auxList](){  
 ancestor->addItem( auxList.front() );  
 auxList.pop\_front();  
 };  
  
 **unsigned** limitOne = keysSplitChild1;  
 node->putKeys(auxList, limitOne);  
 putKeyAncestor();  
  
 **unsigned** limitTwo = keysSplitChild2;  
 newNode->putKeys(auxList, limitTwo);  
 putKeyAncestor();  
  
 **unsigned** limitThree = keysSplitChild3;  
 rightSibling->putKeys(auxList, limitThree);  
  
 *//accommodate children in the nodes.* std::list<Node\*> auxListChildren(std::move(node->children()));  
 auxListChildren.merge(rightSibling->children(), compareKeyNodes);  
  
 **if**(!auxListChildren.empty()){  
 node->putChildren(auxListChildren, limitOne+1);  
 newNode->putChildren(auxListChildren, limitTwo+1);  
 rightSibling->putChildren(auxListChildren, limitThree+1);  
 }  
  
 ancestor->addChild(newNode);  
}  
  
  
**void** BStarTree::mergeRootChildren(Node\* rootChildren)  
{  
 **if**(root->keys().size() > 1){  
 merge(rootChildren);  
 }**else**{  
  
 **auto** deleteRootChildren = [**this**](){  
 root->keys().merge( root->children().front()->keys() );  
  
 **for**(Node\* node : root->children().front()->children()){  
 **dynamic\_cast**<NormalNode\*>(node)->setAncestor(root);  
 }  
 *//adds all the children of the first children of the root to the end of the list of  
 //children of the root.* root->children().splice(root->children().end(), root->children().front()->children(),  
 root->children().front()->children().begin(),  
 root->children().front()->children().end());  
  
 **for**(Node \*child: (\*root->children().begin())->children()){  
 root->children().push\_front(child);  
 }  
 **delete** root->popFrontChild();  
 };  
  
 deleteRootChildren();  
 deleteRootChildren();  
 }  
}  
  
  
**void** BStarTree::merge(Node\* node)  
{  
 Node \*ancestor, \*leftSibling, \*rightSibling;  
 std::list<Node\*>::iterator nodeIt;  
  
 ancestor = node->getAncestor();  
 **auto** ancestorKey = ancestor->keys().begin();  
 **for**(nodeIt = ancestor->children().begin(); \*nodeIt != node; ++nodeIt){  
 ++ancestorKey;  
 }  
  
 *//this assigns the left sibling, node and right sibling depending on the node to  
 //merge being the leftmost, rightmost or not any node in particular* **if**(isLeftmost(node)){  
 leftSibling = \*nodeIt;  
 node = \*next(nodeIt);  
 rightSibling = \*next(next(nodeIt));  
  
 ++ancestorKey;  
 }**else if**(isRightmost(node)){  
 leftSibling = \*prev(prev(nodeIt));  
 node = \*prev(nodeIt);  
 rightSibling = \*nodeIt;  
  
 --ancestorKey;  
 }**else**{  
 leftSibling = \*prev(nodeIt);  
 rightSibling = \*next(nodeIt);  
 }  
  
 std::list<**double**> auxList( std::move(leftSibling->keys()) );  
 auxList.push\_back(\*prev(ancestorKey));  
 ancestor->keys().erase(prev(ancestorKey));  
  
 auxList.merge(node->keys());  
 auxList.push\_back(\*ancestorKey);  
 ancestor->keys().erase(ancestorKey);  
  
 auxList.merge(rightSibling->keys());  
  
 **unsigned** limitOne = auxList.size() / 2;  
 **unsigned** limitTwo = limitOne;  
 **if**(auxList.size() % 2 == 0){  
 limitOne -= 1;  
 }  
  
 leftSibling->putKeys(auxList, limitOne);  
 ancestor->addItem( auxList.front());  
 auxList.pop\_front();  
 node->putKeys(auxList, limitTwo);  
  
 *//move all childrens before removing the right sibling* std::list<Node\*> auxListChildren( std::move(leftSibling->children()) );  
 auxListChildren.merge(node->children(), compareKeyNodes);  
 auxListChildren.merge(rightSibling->children(), compareKeyNodes);  
  
 **if**(!auxListChildren.empty()){  
 leftSibling->putChildren(auxListChildren, limitOne+1);  
 node->putChildren(auxListChildren, limitTwo+1);  
 }  
  
 **delete** rightSibling; *//erases the memory used by this node* ancestor->children().remove(rightSibling);  
}  
  
  
Node\* BStarTree::getGreaterMinor(Node \*node, **double** val) **const**{  
 **if** (node->isLeaf()) {  
 **return nullptr**;  
 }  
  
 std::list<**double**>::iterator ancestorKey;  
  
 **auto** childIt = node->children().begin();  
  
 **for** (ancestorKey = node->keys().begin(); \*ancestorKey < val; ++ancestorKey) {  
 ++childIt;  
 }  
  
 Node\* greaterMinor;  
 greaterMinor = \*childIt;  
 **while** (!greaterMinor->isLeaf()) {  
 greaterMinor = greaterMinor->children().back();  
 }  
  
 **return** greaterMinor;  
}  
  
  
**void** BStarTree::print() **const**{  
 Node\* currentNode;  
 std::queue<Node\*> nodeQueue;  
 **unsigned** height = 0;  
 Node\* lastNode = root;  
 Node\* prevNode = root;  
  
 **auto** printSeparator = [](**unsigned const** & height){  
 std::cout << **"----------------------------"**;  
 **for**(**int** i = height; i / 10 > 0; ++i) std::cout << **'-'**;  
 std::cout << std::endl;  
 };  
  
 std::cout << **"------------------------"** << std::endl;  
 std::cout << **"--------- Root ---------"** << std::endl;  
 std::cout << **"------------------------"** << std::endl;  
 **this**->root->print();  
  
 **for**(Node\* child : **this**->root->children()) nodeQueue.push(child);  
 **int** i =0;  
 **while** (!nodeQueue.empty()) {  
 currentNode = nodeQueue.front();  
 nodeQueue.pop();  
  
 **if**(prevNode == lastNode){  
 printSeparator(height);  
 std::cout << **"--------- Level "** << ++height << **" ----------"** << std::endl;  
 printSeparator(height);  
 **if**(!lastNode->isLeaf()) lastNode = lastNode->children().back();  
 **else** lastNode = **nullptr**;  
 }  
  
 currentNode->print();  
 i++;  
 **for**(Node\* child : currentNode->children()){  
 nodeQueue.push(child);  
 }  
 prevNode = currentNode;  
 }  
 std::cout<<i<<std::endl;  
}  
  
  
**void** Node::addItem(**double** elem)  
{  
 **auto** key = keys().begin();  
 **while**(\*key < elem && key != keys().end()){  
 ++key;  
 }  
  
 keys().insert(key, elem);  
}  
  
  
**void** Node::addChild(Node\* child)  
{  
 **auto** childIt = children().begin();  
 **while**(compareKeyNodes(\*childIt, child) && childIt != children().end()){  
 ++childIt;  
 }  
  
 children().insert(childIt, child);  
}  
  
  
**double** Node::popFrontKey()  
{  
 **double** copy = keys().front();  
 keys().pop\_front();  
  
 **return** copy;  
}  
  
  
**double** Node::popBackKey()  
{  
 **double** copy = keys().back();  
 keys().pop\_back();  
  
 **return** copy;  
}  
  
  
Node\* Node::popFrontChild()  
{  
 Node\* copy = children().front();  
 children().pop\_front();  
  
 **return** copy;  
}  
  
  
Node\* Node::popBackChild()  
{  
 Node\* copy = children().back();  
 children().pop\_back();  
  
 **return** copy;  
}  
  
  
**void** Node::putKeys(std::list<**double**>& takeList, **int** limit)  
{  
 **for**(**int** i = 0; i < limit; i++){  
 keys().push\_back( takeList.front() );  
 takeList.pop\_front();  
 }  
}  
  
  
**void** Node::putChildren(std::list<Node\*>& takeList, **int** limit)  
{  
 **for** (**int** i = 0; i < limit; i++) {  
 children().push\_back( takeList.front() );  
 takeList.pop\_front();  
 **dynamic\_cast**<NormalNode\*>(children().back())->setAncestor(**this**);  
 }  
}  
  
  
**void** Node::print() **const**{  
  
 std::cout << **"Id: "** << **this**->id << **" | "**;  
 **this**->ancestor != **nullptr** ? std::cout << **this**->ancestor->id : std::cout << **' '**;  
 std::cout << std::endl;  
  
 std::cout << **"keys: "**;  
 **for**(**auto** key : keysList){  
 std::cout << key << **" "**;  
 }  
  
 std::cout << **"\n\n"**;  
}  
  
  
**bool** compareKeyNodes(Node\* nodeA, Node\* nodeB)  
{  
 **return** \*nodeA->keys().begin() < \*nodeB->keys().begin();  
}  
  
  
  
  
  
  
*////NormalNode*NormalNode::NormalNode(BStarTree **const** \* tree, Node\* ancestor, **unsigned** id): Node(tree, ancestor)  
{  
 **this**->id = id;  
}  
  
**bool** NormalNode::isOverloaded() **const**{  
 **return this**->keysList.size() > **this**->tree->maxKeysNNode() ? **true** : **false**;  
}  
  
**bool** NormalNode::isFull() **const**{  
 **return this**->keysList.size() == **this**->tree->maxKeysNNode() ? **true** : **false**;  
}  
  
**bool** NormalNode::isUnderloaded() **const**{  
 **return this**->keysList.size() < **this**->tree->minKeysNNode() ? **true** : **false**;  
}  
  
  
**bool** NormalNode::isAtMinimum() **const**{  
 **return this**->keysList.size() == **this**->tree->minKeysNNode() ? **true** : **false**;  
}  
  
  
  
  
  
  
*/////rootNode*RootNode::RootNode(BStarTree **const** \* tree, Node\* ancestor, **unsigned** id): Node(tree, ancestor)  
{  
 **this**->id = id;  
}  
  
  
**bool** RootNode::isOverloaded() **const**{  
 **return this**->keysList.size() > **this**->tree->maxKeysRNode() ? **true** : **false**;  
}  
  
  
**bool** RootNode::isFull() **const**{  
 **return this**->keysList.size() == **this**->tree->maxKeysRNode() ? **true** : **false**;  
}  
  
  
**bool** RootNode::isUnderloaded() **const**{  
 **return this**->keysList.size() < **this**->tree->minKeysRNode() ? **true** : **false**;  
}  
  
  
**bool** RootNode::isAtMinimum() **const**{  
 **return this**->keysList.size() == **this**->tree->minKeysRNode() ? **true** : **false**;  
}  
  
  
  
  
*/////menu***void** menu()  
{  
 std::cout << **"This programs lets you add, delete find and print in a B\* star tree"** << std::endl;  
 **int** order;  
 std::cout << **"Order of the tree: "**;  
 std::cin >> order;  
  
 BStarTree tree(order);  
  
 **char** option;  
 **do**{  
 std::cout << std::endl << **"--------------------"** << std::endl;  
 std::cout << **"You can do the following: "** << std::endl;  
 std::cout << **"[1] Add to the tree"** << std::endl;  
 std::cout << **"[2] Erase from the tree"** << std::endl;  
 std::cout << **"[3] Look for an element in the tree"** << std::endl;  
 std::cout << **"[4] Print the tree by level"** << std::endl;  
 std::cout << **"[0] Exit"** << std::endl;  
 std::cin >> option;  
 std::cout << **"--------------------"** << std::endl << std::endl;  
  
 **switch**(option){  
 **case '1'**:  
 addMenu(tree);  
 **break**;  
 **case '2'**:  
 eraseMenu(tree);  
 **break**;  
 **case '3'**:  
 findMenu(tree);  
 **break**;  
 **case '4'**:  
 tree.print();  
 **break**;  
 }  
 }**while**(option != **'0'**);  
}  
  
  
**void** addMenu(BStarTree& tree)  
{  
 **char** option;  
 std::cout << **"You selected to add an element to the tree"** << std::endl;  
 **do**{  
 std::cout << std::endl << **"--------------------"** << std::endl;  
 std::cout << **"Select one of the following: "** << std::endl;  
 std::cout << **"[1] To capture and add an element from the keyboard"** << std::endl;  
 std::cout << **"[2] To add n elements by random"** << std::endl;  
 std::cout << **"[0] To return to the main menu"** << std::endl;  
 std::cin >> option;  
 std::cout << **"--------------------"** << std::endl << std::endl;  
 clock\_t time\_start= clock();  
 clock\_t time\_end;  
 **int** i=0;  
  
 **switch**(option){  
 **case '1'**:  
 **double** element;  
 std::cout << **"Capture and add element from keyboard"** << std::endl;  
 std::cout << **"Write the element you want to add to the tree: "**;  
 std::cin >> element;  
 time\_start= clock();  
 **while** (i<1000){  
 tree.add(element+i);  
 i++;  
 }  
 time\_end = clock() - time\_start;  
 std::cout <<**"Adding was made by "**<< (**double**)time\_end / **CLOCKS\_PER\_SEC** << std::endl;  
 **break**;  
 **case '2'**:  
 **double** number;  
 std::cout << **"How many elements do you want to add?"** << std::endl;  
 std::cin >> number;  
 i=0;  
 time\_start= clock();  
 **while**(i<number) {  
 i++;  
 tree.add(i);  
 }  
 time\_end = clock() - time\_start;  
 std::cout <<**"Adding was made by "**<< (**double**)time\_end / **CLOCKS\_PER\_SEC** << std::endl;  
 }  
 }**while**(option != **'0'**);  
}  
  
  
**void** eraseMenu(BStarTree& tree)  
{  
 **char** option;  
 std::cout << **"You selected to erase an element from the tree"** << std::endl;  
 **do**{  
 std::cout << std::endl << **"--------------------"** << std::endl;  
 std::cout << **"Select one of the following: "** << std::endl;  
 std::cout << **"[1] To erase all the elements from the tree"** << std::endl;  
 std::cout << **"[2] To capture and erase an element from the keyboard"** << std::endl;  
 std::cout << **"[0] To return to the main menu"** << std::endl;  
 std::cin >> option;  
 std::cout << **"--------------------"** << std::endl << std::endl;  
  
 **switch**(option){  
  
 **case '1'**:  
 std::cout << **"Erasing elements..."** << std::endl;  
 tree.empty();  
 std::cout << **"Erased all elements from the tree"** << std::endl;  
 **break**;  
  
 **case '2'**:  
 **double** element;  
 std::cout << **"Capture and erase element from keyboard"** << std::endl;  
 std::cout << **"Write the element you want to erase to the tree: "**;  
 std::cin >> element;  
 **if**(tree.erase(element)){  
 std::cout << **"Element successfully erased."** << std::endl;  
 }**else**{  
 std::cout << **"Element couldn't be erased. Maybe is already in the tree?"** << std::endl;  
 }  
 **break**;  
  
 }  
 }**while**(option != **'0'**);  
}  
  
  
  
  
**void** findMenu(BStarTree& tree)  
{  
 **char** option;  
 std::cout << **"You selected to find an element in the tree"** << std::endl;  
 **do**{  
 std::cout << std::endl << **"--------------------"** << std::endl;  
 std::cout << **"Select one of the following: "** << std::endl;  
 std::cout << **"[1] To find an element in the tree from the keyboard"** << std::endl;  
 std::cout << **"[0] To return to the main menu"** << std::endl;  
 std::cin >> option;  
 std::cout << **"--------------------"** << std::endl << std::endl;  
 clock\_t time\_start= clock();  
 clock\_t time\_end;  
 **switch**(option){  
 **case '1'**:  
 **double** element;  
 std::cout << **"Capture and find element from keyboard"** << std::endl;  
 std::cout << **"Write the element you want to find in the tree: "**;  
 std::cin >> element;  
 time\_start= clock();  
  
 **for**(**int** i=element; i<=element+1000; i++){  
 tree.find(i);  
 }  
 time\_end = clock() - time\_start;  
 std::cout <<**"Adding was made by "**<< (**double**)time\_end / **CLOCKS\_PER\_SEC** << std::endl;  
 **break**;  
 }  
 }**while**(option != **'0'**);  
}  
  
**int** main() {  
 menu();  
  
 **return** 0;  
}