Санкт-Петербургский политехнический университет Петра Великого
Институт машиностроения, материалов и транспорта
Высшая школа автоматизации и робототехники

Курсовая работа

Дисциплина: Программирование на языках высшего уровня

Тема: B – дерево, добавление узлов

Выполнил студент гр. 3331506/80401 А. В. Леонов

Руководитель Е. М. Кузнецова

«___»_____ 2021 г.

Санкт-Петербург

2021

Оглавление

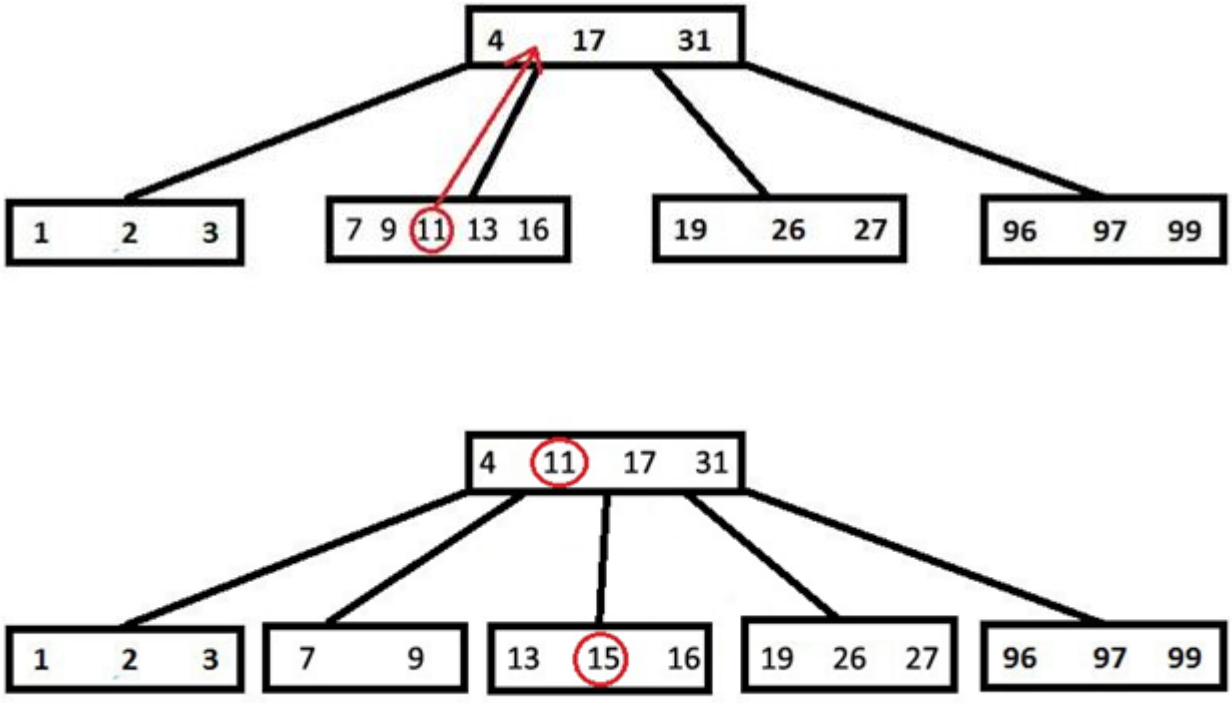
Введение

B-дерево – структура данных, дерево поиска. С точки зрения внешнего логического представления, сбалансированное, сильно ветвистое дерево. Часто используется для хранения данных во внешней памяти. Сбалансированность означает, что длинна любых двух путей от корня до листьев различается не более, чем на единицу. Ветвистость дерева – это свойство каждого узла дерева ссылаться на большее число узлов-потомков. С точки зрения физической организации B-дерево представляется как мультисписочная структура страниц памяти, то есть каждому узлу дерева соответствует блок памяти (страница). Внутренние и листовые страницы обычно имеют разную структуру. Применяется структура B-дерева для организации индексов современных СУБД. Также применяться для структурирования информации на жёстком диске. Время доступа к произвольному блоку на жёстком диске очень велико, поскольку оно определяется скоростью вращения диска и перемещения головок. Поэтому важно уменьшить количество узлов, просматриваемых при каждой операции. Использование поиска по списку каждый раз для нахождения случайного блока могло бы привести к чрезмерному количеству обращений к диску вследствие необходимости последовательного прохода по всем его элементам, предшествующим заданному, тогда как поиск в B-дереве, благодаря свойствам сбалансированности и высокой ветвистости, позволяет значительно сократить количество таких операций. Относительно простая реализация алгоритмов и существование готовых библиотек для работы со структурами B-дерева обеспечивают популярность применения такой организации памяти в самых разнообразных программах, работающих с большими объёмами памяти.

Описание алгоритма добавления ключа

Для добавления ключа в B-дереве, просто создать новый лист и вставить туда ключ нельзя, поскольку будут нарушаться свойства этого дерева. Также вставить ключ в уже заполненный лист невозможно => необходима операция разбиения узла на 2. Если лист был заполнен, то в нем находилось 2t-1 ключей => разбиваем на 2 по t-1, а средний элемент перемещается в родительский узел. Соответственно, если родительский узел также был заполнен – то нам опять приходится разбивать. И так далее до корня. Вставка осуществляется за один проход от корня к листу. На каждой итерации мы разбиваем все заполненные узлы, через которые проходим. Таким образом, если в результате для вставки потребуется разбить какой-то узел – мы уверены в том, что его родитель не заполнен!

Чтобы лучше понять, как работает добавление узла лучше всего показать на рисунке. На рисунке ниже показан процесс добавления узла.



На рисунке показано дерево с t = 3. Мы хотим добавить ключ «15» В поисках позиции для нового ключа мы натыкаемся на заполненный узел (7, 9, 11, 13, 16). Следуя алгоритму, разбиваем его – при этом «11» переходит в родительский узел, а исходный разбивается на 2. Далее ключ «15» вставляется во второй «отколовшийся» узел. Все свойства B-дерева сохраняются.

Сложность алгоритма

Сложность алгоритма добавления узла из B-дерева равна высоте дерева. Поскольку B-дерево является сбалансированным, его высота может быть вычислена от количества элементов и от фактора t - $O(t * \log_t n)$.

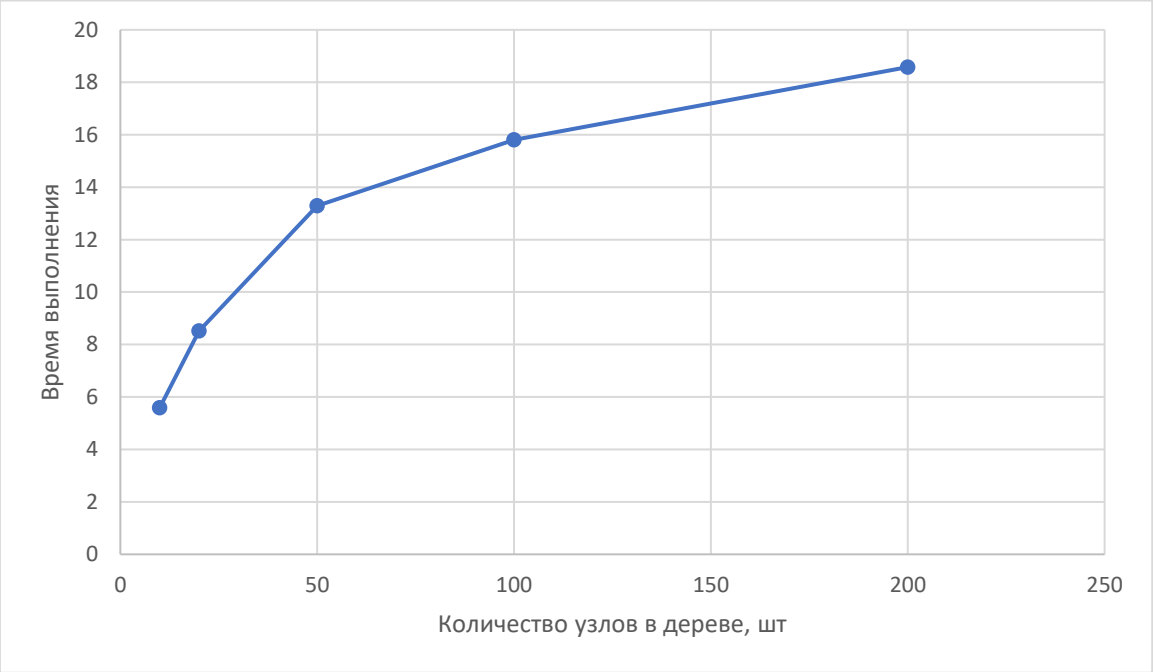
Исследование алгоритма

Результаты измерения зависимости скорости работы алгоритма от количества элементов и t представлены в таблице 1. Для примера возьмём t = 2.

Таблица 1 – Время выполнения алгоритма

Кол-во эл-тов, шт.	10	20	50	100	200
Время выполнения, мс	5.59	5.71	13.29	15.81	28.58

Ниже приведён график полученных значений и из него можно сделать вывод о том, что график соответствует логарифмической зависимости



Список используемой литературы

1. Левитин А. В. Глава 7. Пространственно-временной компромисс: B-Деревья // алгоритмы. Введение в разработку и анализ – М.: Вильямс, 2006.

2. https://habr.com/ru/post/114154/

Приложение. Код

```
#include <iostream>

constexpr auto t = 2;

class BNode {
public:
    bool leaf;
    int numberOfKeys;
    int keys[2 * t - 1];
    BNode* children[2 * t];

    void splitChild(int indexInChildrenArray, BNode* nodeForSplit);
    void insertNonFull(int key);
    int findKey(int key);

    BNode(bool leaf_)
    {
        leaf = leaf_;
        numberOfKeys = 0;
        for (int i = 0; i < (2 * t - 1); i++)
        {
            keys[i] = 0;
        }
        for (int i = 0; i < 2 * t; i++)
        {
            children[i] = nullptr;
        }
    }
};


class BTree
{
public:
    BNode* root;

    BTree()
    {
        root = nullptr;
    }

    void addData(int key);
    BNode* search(int key, BNode* root);
    void traverse(BNode* root);
    void deleteData(int key);
};

void BTree::addData(int key)
{
    if (root == nullptr)
    {
        root = new BNode(true);
        root->keys[0] = key;
        root->numberOfKeys = 1;
        return;
    }

    if (root->numberOfKeys == (2 * t - 1))
    {
        BNode* tempRoot = new BNode(false);
        tempRoot->children[0] = root;
        tempRoot->splitChild(0, root);
```

```
        if (key > tempRoot->keys[0]) tempRoot->children[1]->insertNonFull(key);
        else tempRoot->children[0]->insertNonFull(key);

        root = tempRoot;
    }
    else root->insertNonFull(key);
}

BNode* BTree::search(int key, BNode* root)
{

    int counter = 0;

    while (counter<root->numberOfKeys && key>root->keys[counter]) counter++;

    if (root->keys[counter] == key)
        return root;
    if (root->leaf)
        return nullptr;

    return search(key, root->children[counter]);
}

void BTree::traverse(BNode* root) // Проход от наименьшего ключа к наибольшему
{
    int counter = 0;
    for (counter; counter < root->numberOfKeys; counter++)
    {
        if (root->leaf == false) traverse(root->children[counter]);
        std::cout << root->keys[counter] << " ";
    }
    if (root->leaf == false) traverse(root->children[counter]);
}

int BNode::findKey(int key) // Находит позицию искомого ключа среди ключей узла
{
    int index_of_key = 0;
    while (keys[index_of_key] < key && index_of_key < numberOfKeys)
    {
        index_of_key++;
    }
    return 0;
}

void BTree::deleteData(int key)
{
    BNode* node_that_lost_key = search(key, root);
    //Самый простой случай удаления: лист с достаточным количеством ключей
    if (node_that_lost_key->leaf && node_that_lost_key->numberOfKeys >= t)
    {
        int index_of_key = node_that_lost_key->findKey(key);
        for (int i = index_of_key + 1; i < node_that_lost_key->numberOfKeys; i++)
        {
            node_that_lost_key->keys[i - 1] = node_that_lost_key->keys[i];
        }
        node_that_lost_key->numberOfKeys--;
    }
}

void BNode::splitChild(int indexInChildrenArray, BNode* nodeForSplit)
{
    BNode* newChild = new BNode(nodeForSplit->leaf);
    newChild->numberOfKeys = t - 1;
```

```
    for (int i = 0; i < t - 1; i++) newChild->keys[i] = nodeForSplit->keys[i + t];
    if (nodeForSplit->leaf == false)
    {
        for (int i = 0; i < t; i++) newChild->children[i] = nodeForSplit->children[i +
t];
    }

    nodeForSplit->numberOfKeys = t - 1;

    for (int i = numberOfKeys; i >= indexInChildrenArray + 1; i--) children[i + 1] =
children[i]; //Смещение детей вправо для выделения места под нового ребенка
    children[indexInChildrenArray + 1] = newChild;

    for (int i = numberOfKeys - 1; i >= indexInChildrenArray; i--) keys[i + 1] = keys[i];
    keys[indexInChildrenArray] = nodeForSplit->keys[t - 1];

    numberOfKeys++;
}

void BNode::insertNonFull(int key)
{
    int currentIndex = numberOfKeys - 1;//индекс самого правого элемента

    if (leaf)
    {
        while (currentIndex >= 0 && keys[currentIndex] > key)
        {
            keys[currentIndex + 1] = keys[currentIndex];
            currentIndex--;
        }

        keys[currentIndex + 1] = key;
        numberOfKeys++;
    }
    else
    {
        while (currentIndex >= 0 && keys[currentIndex] > key) currentIndex--;

        if (children[currentIndex + 1]->numberOfKeys == 2 * t - 1)
        {
            splitChild(currentIndex + 1, children[currentIndex + 1]);
            if (keys[currentIndex + 1] < key) currentIndex++;
        }
        children[currentIndex + 1]->insertNonFull(key);
    }
}


int main()
{

    BTree Tree;
    Tree.addData(50);
    Tree.addData(35);
    Tree.addData(24);
    Tree.addData(11);
    Tree.addData(43);
    Tree.addData(15);
    Tree.addData(34);
    Tree.traverse(Tree.root);
    std::cout << std::endl;
```

```
Tree.addData(44);
Tree.addData(8);
Tree.traverse(Tree.root);
      }
```