ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ОБРАЗОВАНИЯ

*«*САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ

УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»

Институт металлургии, машиностроения и транспорта

Кафедра «Мехатроника и роботостроение»

**Курсовой проект**

по дисциплине «Программирование»

**«В-дерево»**

Пояснительная записка

|  |  |  |
| --- | --- | --- |
| Выполнил  студент гр. 33335/2 | *(подпись)* | Ненахов И.Д. |
| Работу принял | *(подпись)* | Ананьевский М. С. |

Санкт-Петербург

2018 г.

Формулировка задачи, которую решает алгоритм.

В-дерево – структура данных, предназначенная для эффективного доступа к информации. При работе алгоритма реализуются добавление, удаление и поиск ключа.

Словесное описание алгоритма.

В-дерево представляет из себя совокупность связанных узлов, удовлетворяющих следующим правилам:

* Ключи в каждом узле отсортированы;
* В корне содержится от 1 до 2t-1 ключей, где t – параметр дерева;
* Во всех остальных узлах содержится от t-1 до 2t-1 ключей;
* Листья – узлы, у которых нет потомков;
* Глубина всех листьев одинакова;
* Другие узлы, содержащие n ключей, содержат n+1 потомков. При этом:
  + - Первый потомок содержит ключи из интервала (-∞,К1);
    - Каждый следующий потомок содержит ключи в интервале (Ki-2,Ki);
    - Последний потомок содержит ключи в интервале (Kn,∞).

Реализация алгоритма.

Алгоритм был реализован при помощи языка программирования C++.

Для описания узла была создана структура node:

**struct** node{  
 node \*parent;  
 node \*children[2\*t];  
 **bool** leaf;  
 **double** keys[2\*t-1];  
 **int** keyscount;  
 **int** childrencount;  
 **int** childorder;  
};

Для всего дерева был создан класс tree:

**class** tree{  
  
**public**:  
 node \*root;  
 **void** add(**double** var);  
 **void** sort(node \*curnode);  
  
 **double** \*search(**double** var);  
 **void** deletekey(**double** var);  
 tree()  
 {  
 root=**new** node;  
 root->leaf=**true**;  
 root->keyscount=0;  
 root->childrencount=0;  
 root->parent=0;  
 **for**(**int** i=0;i<2\*t;i++)  
 {  
 root->children[i]=0;  
 }  
  
  
 };  
**private**:  
 **void** addtonode(**double** var,node \*curnode);  
 node \*create(node \*parent);  
 node\* searchnode(node \*curnode,**double** var);  
 **double** \*searchinnode(node \*curnode, **double** var);  
 **void** deleteinnode(**double** var, node \*curnode);  
 **void** sortchildren(node \* curnode);  
 node \*createuppernode();  
  
  
};

Также были реализованы внутренние (private) методы для этого класса:

1. Сортировка ключей в узле:

**void** tree::sort(node \*curnode)  
{  
 **int** i=1;  
 **int** j=2;  
 **double** k;  
 **int** t;  
 node \*z;  
 **while**(i<curnode->keyscount)  
 {  
 **if**(curnode->keys[i]>curnode->keys[i-1])  
 {  
 i=j;  
 j++;  
 }  
 **else** {  
 k=curnode->keys[i];  
 z=curnode->children[i];  
  
 curnode->keys[i]=curnode->keys[i-1];  
 curnode->children[i]=curnode->children[i-1];  
  
 curnode->keys[i-1]=k;  
 curnode->children[i-1]=z;  
  
 i--;  
 **if** (i==0)  
 {  
 i=j;  
 j++;  
 }  
 }  
 }  
}

1. Создание узла:

node \*tree::create(node \*parent)  
{  
 node \*curnode=**new** node;  
 curnode->childrencount=0;  
 curnode->keyscount=0;  
 curnode->leaf=**true**;  
 *//curnode->childorder=CO;* parent->childrencount++;  
 curnode->parent=parent;  
 **for**(**int** i=0;i<2\*t;i++)  
 {  
 curnode->children[i]=0;  
 }  
 **for**(**int** i=0;i<2\*t-1;i++)  
 {  
 curnode->keys[i]=0;  
 }  
 **return** curnode;  
}

1. Создание нового корня:

node \*tree::createuppernode()  
{  
 node \*curnode=**new** node;  
 root->parent=curnode;  
 curnode->childrencount=1;  
 root->childorder=0;  
 curnode->children[0]=root;  
 curnode->keyscount=0;  
 curnode->leaf=**false**;  
 root=curnode;  
 curnode->parent=0;  
 **for**(**int** i=1;i<2\*t;i++)  
 {  
 curnode->children[i]=0;  
 }  
 **for**(**int** i=1;i<2\*t-1;i++)  
 {  
 curnode->keys[i]=0;  
 }  
 **return** curnode;  
}

1. Сортировка указателей на потомков:

**void** tree::sortchildren(node \*curnode)  
{  
 **int** i=1;  
 **int** j=2;  
 **int** t;  
 node\* z;  
 **while** (i<curnode->keyscount+1) {  
 **if** (curnode->children[i]->keys[0]>curnode->children[i-1]->keys[0]) {  
 i = j;  
 j++;  
 } **else** {  
 z = curnode->children[i];  
 curnode->children[i] = curnode->children[i-1];  
 curnode->children[i-1] = z;  
 i--;  
 **if** (i==0) {  
 i = j;  
 j++;  
 }  
 }  
 }  
}

1. Поиск узла, в котором содержится ключ:

node \*tree::searchnode(node \*curnode, **double** var)  
{  
 **if**(curnode->keyscount==0) **return** curnode;  
 **if**(var>=curnode->keys[curnode->keyscount-1])  
 {  
 **if**(curnode->children[0]==0) **return** curnode;  
 **else return** searchnode(curnode->children[curnode->keyscount],var);  
 }  
 **for**(**int** i=0;i<curnode->keyscount;i++)  
 {  
 **if**(var<curnode->keys[i])  
 {  
 **if**(curnode->children[i]==0) **return** curnode;  
 **else return** searchnode(curnode->children[i],var);  
 }  
 **if**((curnode->keyscount==1)&&(var==curnode->keys[0])) **return** curnode;  
 }  
 **return** 0;  
}

1. Поиск положения ключа в узле:

**double** \*tree::searchinnode(node \*curnode, **double** var)  
{  
 **double** \*ret=0;  
 **for**(**int** i=0;i<curnode->keyscount;i++)  
 {  
 **if**(var==curnode->keys[i]) ret=&curnode->keys[i];  
 }  
 **return** ret;  
}

1. Добавление ключа к узлу:

**void** tree::addtonode(**double** var, node \*curnode) {  
 **if** ((curnode->keyscount < 2 \* t - 1)) {  
 curnode->keys[curnode->keyscount] = var;  
 curnode->keyscount++;  
 sort(curnode);  
 } **else** {  
  
 **if** (!curnode->parent) createuppernode();  
 addtonode(curnode->keys[t-1],curnode->parent);  
 **double** key=curnode->keys[t-1];  
 node \*par = curnode->parent;  
 par->children[par->keyscount] = create(par);  
 node \*child2 = par->children[par->keyscount];  
 **int** z = 0;  
 **for** (**int** i = t; i < 2 \* t - 1; i++) {  
 child2->keys[z] = curnode->keys[i];  
 child2->keyscount++;  
 curnode->keys[i]=0;  
 curnode->keyscount--;  
 **if** (curnode->children[i]) {  
 child2->childrencount++;  
 child2->children[z] = curnode->children[i];  
 curnode->children[i]->parent=child2;  
 curnode->children[i] = 0;  
 }  
 z++;  
 }  
 **if** (curnode->children[2 \* t - 1]) {  
 child2->childrencount++;  
 child2->children[z] = curnode->children[2 \* t - 1];  
 curnode->children[2 \* t - 1]->parent=child2;  
 curnode->children[2 \* t - 1] = 0;  
 }  
  
 curnode->keys[t-1]=0;  
 curnode->keyscount--;  
 **if**(var<key) {  
 curnode->keys[t-1]=var;  
 curnode->keyscount++;  
 sort(curnode);  
 }  
 **else** {  
 child2->keys[child2->keyscount]=var;  
 child2->keyscount++;  
 sort(child2);  
 }  
 sort(par);  
 sortchildren(par);}  
}

Публичные (public) методы:

1. Поиск ключа в дереве:

**double** \*tree::search(**double** var)  
{  
 **using namespace** std;  
 node \*targetnode=searchnode(root,var);  
 **double** \*ret=searchinnode(targetnode,var);  
  
 **return** ret;  
}

1. Добавление ключа к дереву:

**void** tree::add(**double** var)  
{  
 **using namespace** std;  
 node \*node=searchnode(root,var);  
 **double** \*position=searchinnode(node,var);  
 **if**(position!=0) {  
 cout<<var<<**" has already been added to address "**<<search(var)<<endl;  
 **return**;  
 }  
 addtonode(var,node);  
}

1. Удаление ключа из дерева:

**void** tree::deletekey(**double** var)  
{  
 deleteinnode(var,searchnode(root,var));  
  
}

Анализ алгоритма

1. Время работы алгоритма

Каждое действие при работе с В-деревом (поиск, удаление, добавление ключа) происходит за время О(, n – количество узлов.

Применение алгоритма

Структура B-дерева применяется для организации [индексов](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D0%B4%D0%B5%D0%BA%D1%81_(%D0%B1%D0%B0%D0%B7%D1%8B_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85)) во многих современных [системах](https://ru.wikipedia.org/wiki/%D0%A1%D0%A3%D0%91%D0%94) управления базами данных.

B-дерево может применяться для структурирования информации на жёстком диске. Время доступа к произвольному блоку на жёстком диске очень велико (порядка миллисекунд), поскольку оно определяется скоростью вращения диска и перемещения головок. Поэтому важно уменьшить количество узлов, просматриваемых при каждой операции. Использование поиска по [списку](https://ru.wikipedia.org/wiki/%D0%A1%D0%BF%D0%B8%D1%81%D0%BE%D0%BA_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) 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.