**Министерство образования и науки Российской Федерации**

*Федеральное государственное бюджетное образовательное учреждение   
высшего профессионального образования*   
**«АЛТАЙСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ИМ. И.И. ПОЛЗУНОВА» (АлтГТУ)**

Факультет информационных технологий

Кафедра прикладной математики

Отчёт по дисциплине «Алгоритмы и структуры данных»

Лабораторная работа №7

«Б-деревья порядка n»

Выполнил:

Студент гр. ПИ-32

Глушков Г.Г.

Проверил:

ст. преподаватель Н.Д. Бубнова

Барнаул 2015

* Формулировка задачи:

Удалить из Б-дерева ключи принадлежащие ,заданному диапазону.

* Алгоритм решения:

Для начала мы определяем расположение узла, в котором находится ключ. Возможны три варианта расположения: узел является листом, внутренним и корнем. Если удаление происходит из листа, то необходимо проверить, сколько ключей находится в нем. Если больше t-1, то просто удаляем и больше ничего делать не нужно. Иначе, если существует соседний лист (находящийся рядом с ним и имеющий такого же родителя), который содержит больше t-1 ключа, то выберем ключ из этого соседа, который является разделителем между оставшимися ключами узла-соседа и исходного узла (то есть не больше всех из одной группы и не меньше всех из другой). Пусть это ключ k1. Выберем ключ k2 из узла-родителя, который является разделителем исходного узла и его соседа, который мы выбрали ранее. Удалим из исходного узла нужный ключ (который необходимо было удалить), спустим k2 в этот узел, а вместо k2 в узле-родителе поставим k1. Если же все соседи нашего узла имеют по t-1 ключу. То мы объединяем его с каким-либо соседом, удаляем нужный ключ. И тот ключ из узла-родителя, который был разделителем для этих двух «бывших» соседей, переместим в наш новообразовавшийся узел.

Если удаление из внутреннего узла x ключа k. Если дочерний узел, предшествующий ключу k содержит больше t-1 ключа, то находим k1 – предшественника k в поддереве этого узла. Удаляем его (рекурсивно запускаем наш алгоритм). Заменяем k в исходном узле на k1. Проделываем аналогичную работу, если дочерний узел, следующий за ключом k, имеет больше t-1 ключа. Если оба (следующий и предшествующий дочерние узлы) имеют по t-1 ключу, то объединяем этих детей, переносим в них k, а далее удаляем k из нового узла (рекурсивно запускаем наш алгоритм). Если сливаются 2 последних потомка корня – то они становятся корнем, а предыдущий корень освобождается.

* Текст программы с комментариями:

#include "stdafx.h"

#include<iostream>

#include<fstream>

using namespace std;

// BTree node

class BTreeNode

{

int \*keys; // Ключи

int t; // Минимальная степень (Опеределяет диапазон для числа ключей)

BTreeNode \*\*C; // Массив указателей на детей

int n; // Текущий номер ключа

bool leaf; // true если узел лист. Иначе ложь

public:

BTreeNode(int \_t, bool \_leaf); // Конструктор

//Функция вывода всех узлов в дереве

void traverse(bool no\_childs);

// Функция поиска ключа в дереве

BTreeNode \*search(int k); // Возвращает 0 если K не найдено

//Функция возвращает номер первого ключа, который больше или равен K

int findKey(int k);

//Функция вставки узла в дерево, предполагается, что узел не полный

void insertNonFull(int k);

//Функция для разделения ребенка у узла. Ребенок должен быть полным

void splitChild(int i, BTreeNode \*y);

// Функция удаления ключа из дерева

void remove(int k);

//Функция удаление ключа , присутствующего в idx позиции в узле, который представляет лист

void removeFromLeaf(int idx);

//Функция удаление ключа , присутствующего в idx позиции в узле, который не представляет лист

void removeFromNonLeaf(int idx);

//Функция получение предка у idx позиции

int getPred(int idx);

//Функция получение приемника у idx позиции

int getSucc(int idx);

// A function to fill up the child node present in the idx-th

// position in the C[] array if that child has less than t-1 keys

void fill(int idx);

// A function to borrow a key from the C[idx-1]-th node and place

// it in C[idx]th node

void borrowFromPrev(int idx);

// A function to borrow a key from the C[idx+1]-th node and place it

// in C[idx]th node

void borrowFromNext(int idx);

// A function to merge idx-th child of the node with (idx+1)th child of

// the node

void merge(int idx);

// Make BTree friend of this so that we can access private members of

// this class in BTree functions

friend class BTree;

};

class BTree

{

BTreeNode \*root; // Pointer to root node

int t; // Minimum degree

public:

// Constructor (Initializes tree as empty)

BTree(int \_t)

{

root = NULL;

t = \_t;

}

void traverse(bool no\_childs)

{

if (root != NULL) root->traverse(false);

}

// function to search a key in this tree

BTreeNode\* search(int k)

{

return (root == NULL)? NULL : root->search(k);

}

// The main function that inserts a new key in this B-Tree

void insert(int k);

// The main function that removes a new key in thie B-Tree

void remove(int k);

};

BTreeNode::BTreeNode(int t1, bool leaf1)

{

// Copy the given minimum degree and leaf property

t = t1;

leaf = leaf1;

// Allocate memory for maximum number of possible keys

// and child pointers

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

C = new BTreeNode \*[2\*t];

// Initialize the number of keys as 0

n = 0;

}

// A utility function that returns the index of the first key that is

// greater than or equal to k

int BTreeNode::findKey(int k)

{

int idx=0;

while (idx<n && keys[idx] < k)

++idx;

return idx;

}

// A function to remove the key k from the sub-tree rooted with this node

void BTreeNode::remove(int k)

{

int idx = findKey(k);

// The key to be removed is present in this node

if (idx < n && keys[idx] == k)

{

// If the node is a leaf node - removeFromLeaf is called

// Otherwise, removeFromNonLeaf function is called

if (leaf)

removeFromLeaf(idx);

else

removeFromNonLeaf(idx);

}

else

{

// If this node is a leaf node, then the key is not present in tree

if (leaf)

{

cout << "Ключ "<< k <<" не существует в дереве\n";

return;

}

// The key to be removed is present in the sub-tree rooted with this node

// The flag indicates whether the key is present in the sub-tree rooted

// with the last child of this node

bool flag = ( (idx==n)? true : false );

// If the child where the key is supposed to exist has less that t keys,

// we fill that child

if (C[idx]->n < t)

fill(idx);

// If the last child has been merged, it must have merged with the previous

// child and so we recurse on the (idx-1)th child. Else, we recurse on the

// (idx)th child which now has atleast t keys

if (flag && idx > n)

C[idx-1]->remove(k);

else

C[idx]->remove(k);

}

return;

}

// A function to remove the idx-th key from this node - which is a leaf node

void BTreeNode::removeFromLeaf (int idx)

{

// Move all the keys after the idx-th pos one place backward

for (int i=idx+1; i<n; ++i)

keys[i-1] = keys[i];

// Reduce the count of keys

n--;

return;

}

// A function to remove the idx-th key from this node - which is a non-leaf node

void BTreeNode::removeFromNonLeaf(int idx)

{

int k = keys[idx];

// If the child that precedes k (C[idx]) has atleast t keys,

// find the predecessor 'pred' of k in the subtree rooted at

// C[idx]. Replace k by pred. Recursively delete pred

// in C[idx]

if (C[idx]->n >= t)

{

int pred = getPred(idx);

keys[idx] = pred;

C[idx]->remove(pred);

}

// If the child C[idx] has less that t keys, examine C[idx+1].

// If C[idx+1] has atleast t keys, find the successor 'succ' of k in

// the subtree rooted at C[idx+1]

// Replace k by succ

// Recursively delete succ in C[idx+1]

else if (C[idx+1]->n >= t)

{

int succ = getSucc(idx);

keys[idx] = succ;

C[idx+1]->remove(succ);

}

// If both C[idx] and C[idx+1] has less that t keys,merge k and all of C[idx+1]

// into C[idx]

// Now C[idx] contains 2t-1 keys

// Free C[idx+1] and recursively delete k from C[idx]

else

{

merge(idx);

C[idx]->remove(k);

}

return;

}

// A function to get predecessor of keys[idx]

int BTreeNode::getPred(int idx)

{

// Keep moving to the right most node until we reach a leaf

BTreeNode \*cur=C[idx];

while (!cur->leaf)

cur = cur->C[cur->n];

// Return the last key of the leaf

return cur->keys[cur->n-1];

}

int BTreeNode::getSucc(int idx)

{

// Keep moving the left most node starting from C[idx+1] until we reach a leaf

BTreeNode \*cur = C[idx+1];

while (!cur->leaf)

cur = cur->C[0];

// Return the first key of the leaf

return cur->keys[0];

}

// A function to fill child C[idx] which has less than t-1 keys

void BTreeNode::fill(int idx)

{

// If the previous child(C[idx-1]) has more than t-1 keys, borrow a key

// from that child

if (idx!=0 && C[idx-1]->n>=t)

borrowFromPrev(idx);

// If the next child(C[idx+1]) has more than t-1 keys, borrow a key

// from that child

else if (idx!=n && C[idx+1]->n>=t)

borrowFromNext(idx);

// Merge C[idx] with its sibling

// If C[idx] is the last child, merge it with with its previous sibling

// Otherwise merge it with its next sibling

else

{

if (idx != n)

merge(idx);

else

merge(idx-1);

}

return;

}

// A function to borrow a key from C[idx-1] and insert it

// into C[idx]

void BTreeNode::borrowFromPrev(int idx)

{

BTreeNode \*child=C[idx];

BTreeNode \*sibling=C[idx-1];

// The last key from C[idx-1] goes up to the parent and key[idx-1]

// from parent is inserted as the first key in C[idx]. Thus, the loses

// sibling one key and child gains one key

// Moving all key in C[idx] one step ahead

for (int i=child->n-1; i>=0; --i)

child->keys[i+1] = child->keys[i];

// If C[idx] is not a leaf, move all its child pointers one step ahead

if (!child->leaf)

{

for(int i=child->n; i>=0; --i)

child->C[i+1] = child->C[i];

}

// Setting child's first key equal to keys[idx-1] from the current node

child->keys[0] = keys[idx-1];

// Moving sibling's last child as C[idx]'s first child

if (!leaf)

child->C[0] = sibling->C[sibling->n];

// Moving the key from the sibling to the parent

// This reduces the number of keys in the sibling

keys[idx-1] = sibling->keys[sibling->n-1];

child->n += 1;

sibling->n -= 1;

return;

}

// A function to borrow a key from the C[idx+1] and place

// it in C[idx]

void BTreeNode::borrowFromNext(int idx)

{

BTreeNode \*child=C[idx];

BTreeNode \*sibling=C[idx+1];

// keys[idx] is inserted as the last key in C[idx]

child->keys[(child->n)] = keys[idx];

// Sibling's first child is inserted as the last child

// into C[idx]

if (!(child->leaf))

child->C[(child->n)+1] = sibling->C[0];

//The first key from sibling is inserted into keys[idx]

keys[idx] = sibling->keys[0];

// Moving all keys in sibling one step behind

for (int i=1; i<sibling->n; ++i)

sibling->keys[i-1] = sibling->keys[i];

// Moving the child pointers one step behind

if (!sibling->leaf)

{

for(int i=1; i<=sibling->n; ++i)

sibling->C[i-1] = sibling->C[i];

}

// Increasing and decreasing the key count of C[idx] and C[idx+1]

// respectively

child->n += 1;

sibling->n -= 1;

return;

}

// A function to merge C[idx] with C[idx+1]

// C[idx+1] is freed after merging

void BTreeNode::merge(int idx)

{

BTreeNode \*child = C[idx];

BTreeNode \*sibling = C[idx+1];

// Pulling a key from the current node and inserting it into (t-1)th

// position of C[idx]

child->keys[t-1] = keys[idx];

// Copying the keys from C[idx+1] to C[idx] at the end

for (int i=0; i<sibling->n; ++i)

child->keys[i+t] = sibling->keys[i];

// Copying the child pointers from C[idx+1] to C[idx]

if (!child->leaf)

{

for(int i=0; i<=sibling->n; ++i)

child->C[i+t] = sibling->C[i];

}

// Moving all keys after idx in the current node one step before -

// to fill the gap created by moving keys[idx] to C[idx]

for (int i=idx+1; i<n; ++i)

keys[i-1] = keys[i];

// Moving the child pointers after (idx+1) in the current node one

// step before

for (int i=idx+2; i<=n; ++i)

C[i-1] = C[i];

// Updating the key count of child and the current node

child->n += sibling->n+1;

n--;

// Freeing the memory occupied by sibling

delete(sibling);

return;

}

// The main function that inserts a new key in this B-Tree

void BTree::insert(int k)

{

// If tree is empty

if (root == NULL)

{

// Allocate memory for root

root = new BTreeNode(t, true);

root->keys[0] = k; // Insert key

root->n = 1; // Update number of keys in root

}

else // If tree is not empty

{

// If root is full, then tree grows in height

if (root->n == 2\*t-1)

{

// Allocate memory for new root

BTreeNode \*s = new BTreeNode(t, false);

// Make old root as child of new root

s->C[0] = root;

// Split the old root and move 1 key to the new root

s->splitChild(0, root);

// New root has two children now. Decide which of the

// two children is going to have new key

int i = 0;

if (s->keys[0] < k)

i++;

s->C[i]->insertNonFull(k);

// Change root

root = s;

}

else // If root is not full, call insertNonFull for root

root->insertNonFull(k);

}

}

// A utility function to insert a new key in this node

// The assumption is, the node must be non-full when this

// function is called

void BTreeNode::insertNonFull(int k)

{

// Initialize index as index of rightmost element

int i = n-1;

// If this is a leaf node

if (leaf == true)

{

// The following loop does two things

// a) Finds the location of new key to be inserted

// b) Moves all greater keys to one place ahead

while (i >= 0 && keys[i] > k)

{

keys[i+1] = keys[i];

i--;

}

// Insert the new key at found location

keys[i+1] = k;

n = n+1;

}

else // If this node is not leaf

{

// Find the child which is going to have the new key

while (i >= 0 && keys[i] > k)

i--;

// See if the found child is full

if (C[i+1]->n == 2\*t-1)

{

// If the child is full, then split it

splitChild(i+1, C[i+1]);

// After split, the middle key of C[i] goes up and

// C[i] is splitted into two. See which of the two

// is going to have the new key

if (keys[i+1] < k)

i++;

}

C[i+1]->insertNonFull(k);

}

}

// A utility function to split the child y of this node

// Note that y must be full when this function is called

void BTreeNode::splitChild(int i, BTreeNode \*y)

{

// Create a new node which is going to store (t-1) keys

// of y

BTreeNode \*z = new BTreeNode(y->t, y->leaf);

z->n = t - 1;

// Copy the last (t-1) keys of y to z

for (int j = 0; j < t-1; j++)

z->keys[j] = y->keys[j+t];

// Copy the last t children of y to z

if (y->leaf == false)

{

for (int j = 0; j < t; j++)

z->C[j] = y->C[j+t];

}

// Reduce the number of keys in y

y->n = t - 1;

// Since this node is going to have a new child,

// create space of new child

for (int j = n; j >= i+1; j--)

C[j+1] = C[j];

// Link the new child to this node

C[i+1] = z;

// A key of y will move to this node. Find location of

// new key and move all greater keys one space ahead

for (int j = n-1; j >= i; j--)

keys[j+1] = keys[j];

// Copy the middle key of y to this node

keys[i] = y->keys[t-1];

// Increment count of keys in this node

n = n + 1;

}

// Function to traverse all nodes in a subtree rooted with this node

void BTreeNode::traverse(bool no\_childs)

{

int i;

if (no\_childs)

{

cout << "[";

for (i = 0; i < n; i++)

{

cout << keys[i];

if (i < n-1) cout << " ";

}

cout << "]\n";

}

else

{

cout << "Родитель:\n";

cout << "[";

for (i = 0; i < n; i++)

{

cout << keys[i];

if (i < n-1) cout << " ";

}

cout << "]\n";

cout << "Дети:\n";

if (!leaf)

{

for (i = 0; i <= n; i++)

{

C[i]->traverse(true);

}

}

cout << "\n";

if (!leaf)

{

for (i = 0; i < n; i++)

{

// If this is not leaf, then before printing key[i],

// traverse the subtree rooted with child C[i].

if (C[i]->leaf == false)

C[i]->traverse(false);

}

// Print the subtree rooted with last child

if (C[i]->leaf == false)

C[i]->traverse(false);

}

}

}

// Function to search key k in subtree rooted with this node

BTreeNode \*BTreeNode::search(int k)

{

// Find the first key greater than or equal to k

int i = 0;

while (i < n && k > keys[i])

i++;

// If the found key is equal to k, return this node

if (keys[i] == k)

return this;

// If key is not found here and this is a leaf node

if (leaf == true)

return NULL;

// Go to the appropriate child

return C[i]->search(k);

}

void BTree::remove(int k)

{

if (!root)

{

cout << "Дерево пустое\n";

return;

}

// Call the remove function for root

root->remove(k);

// If the root node has 0 keys, make its first child as the new root

// if it has a child, otherwise set root as NULL

if (root->n==0)

{

BTreeNode \*tmp = root;

if (root->leaf)

root = NULL;

else

root = root->C[0];

// Free the old root

delete tmp;

}

return;

}

// Driver program to test above functions

int main()

{

int keys;

BTree t(3); // Степень дерева, минимум три

setlocale(LC\_CTYPE,"rus");

ifstream a("input.txt");

while(!a.eof())

{

a>>keys;

t.insert(keys);

}

cout << "Вывод дерева:\n";

t.traverse(true);

cout << endl;

while(keys!=999){

cin>>keys;

if(keys==999)

break;

if(keys==0)

cout<<"Вы ввели 0\n";

else

{

t.remove(keys);

cout << "Вывод после удаление ключа\n";

t.traverse(true);

cout << endl;

}

}

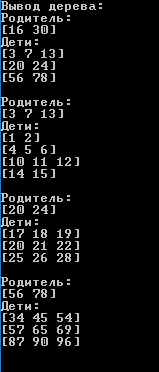
return 0;

}

Содержимое исходного файла:

1 3 7 10 11 13 14 15 18 16 19 24 25 26 21 4 5 20 22 2 17 12 6 20 30 34 57 28 90 78 87 65 56 45 54 69 96

Вывод дерева



Последовательно удаляем ключи 78,2,15

