Министерство образования и науки РФ

ФГБОУ ВПО «Пермский государственный национальный исследовательский университет»

Кафедра прикладной математики и информатики

**Индивидуальное задание №2**

**«Динамические структуры данных»**

***по дисциплине «Алгоритмизация и программирование - II»***

|  |  |
| --- | --- |
|  | **Выполнил:** |
|  | Студент 1-го курса механико-математического ф-та  Пьяных Андрей,  группа ПМИ-1,2. |
|  | **Проверил:** |
|  | Доцент кафедры прикладной математики и информатики ПГУ, к.т.н.  Перескокова О.И. |
|  | «28» апреля 2021 г. |

Пермь 2021

# Постановка задачи.

Имеется список студентов некоторого ВУЗа. Необходимо реализовать хранение этих данных с помощью Б-дерева порядка n. Программа должна выдавать полный отчет о построении дерева и изменении дерева, т.е. распечатывать дерево так, чтобы видна была структура дерева после каждого включения и исключения. Можно обойтись только динамической памятью и файлы не использовать.

# Определение идеи алгоритма, выбор методов решения и структур данных.

Выберем систему программирования Microsoft Visual Studio.

Описание использованных структур

int const N – степень Б-дерева

struct BTree:(структура узла Б-дерева)

* bool full – переменная показывающая заполнен ли полностью узел или нет
* string Data\* – массив хранящий в себе элементы(ключи) узла
* BTree\* Prev – ссылка на предыдущий узел
* BTree\* Next\* – массив хранящий указатели на сыновей-узлов
* int countData – переменная хранящая кол-во элементов в узле
* int countNext -переменная хранящая кол-во сыновей узла

Описание работы функций:

void simple\_insert(string x, BTree\* p) – вставляет элемент x в ячейку, если она не заполненная полностью, то-есть содержит менее 2 \* N – 1 элементов

* вставляет элемент х в первую пустую ячейку в узле
* сортирует его при помощи sort()
* проверяет заполненность узла после вставки

void drob(BTree\* p) – разделяет ячейку, для вставки элемента в заполненную ячейку

* разделяет текущую узел на 2 узла
* центральный элемент переходит в родительский узел
* если узел является корнем дерева, создает 2 узла и оставляет в себе только центральный элемент

void search(string x, BTree\* p) – ищет лист для вставки и производит вставку с помощью simple\_insert() и drob()

* проверяет пуст ли узел, для вставки элемента х в узел
* проверяет больше ли х, последнего элемента в узле
* если узел полон, то вставляет элемент при помощи simple\_insert() и разделяет узел drob()
* если ни одно из условий не выполнилось, проходимся по узлу и его сыновьям в поисках нужного узла, для сохранения свойств Б-дерева и выполняет вставку с помощью тех же функций

void output(string str[], int l, BTree\* p, int& max\_level) – организует вывод по уровням самого Б-дерева

* определяет максимальный уровень дерева
* создает строки для вывода уровней Б-дерева

int main()

* создает корень дерева
* организует ввод данных через цикл, пока не будет введена команда остановки ввода
* считывает команды из консоли
* выводит данные полученные из функции output() в цикле

void simple\_del(string x, BTree\* p, int N) – удаляет элемент из листа, если в нем более N-1 ключей

* ищет элемент в листе
* смещает массив ключей и дочерних элементов
* зануляет последний ключ и дочерний элемент

void leaf\_del(string x, BTree\* p, int N, BTree\* head) – удаляет ключ из листа, если в нем менее N ключей

* ищет элемент в листе
* ищет текущий лист в предке
* если есть сосед с более N-1 ключей переносит из него 1 ключ и удаляет задаваемый ключ
* если в дочерних элементах N-1 ключей или менее, соеденяем их и переносим самый левый ключ на место удаляймого элемента

void knot\_del(string x, BTree\* p, int N) – удаляет ключ из узла

* ищет элемент в листе
* если в левом узле больше N-1 эмлементов, переносим вместо удаляймого ключа самый правый элемент дочернего узла
* если в правом узле больше N-1 эмлементов, переносим вместо удалймого ключа, самый левый элемент дочернего узла
* если в дочерних элементах N-1 ключей или менее, соеденяем их и переносим самый левый ключ на место удаляймого элемента

void Del(string x, BTree\* p, int N, BTree\* head) – основная функция удаления

* ищет удаляемый ключ во всем дереве рекурсивно
* если это лист в котором, более N-1 ключей просто удаляем его (simple\_del)
* если это лист в котором, менее N ключей удаляет его (leaf\_del)
* если текущий элемент узел (knot\_del)

**Входные данные**

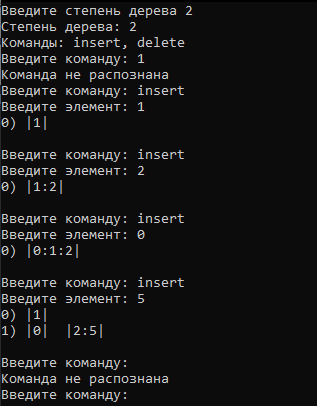
Формат ввода:

На вход подается степень дерева. Далее подаются команды и фамилии учеников для вставки/удаления в Б-дерево

Формат вывода:

Элементы Б-дерева выведенные по уровням

Пример:



**Тестирование программы**

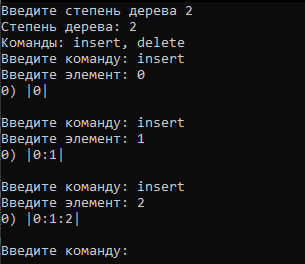
Присваиваем степень Б-дерева 2, для удобства проверок

1. Вводим 3 элемента для в ставки в 1 узел

Ожидаемы результат: Вывод упорядоченного корня |0:1:2|

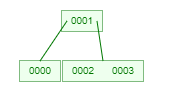


Результат:

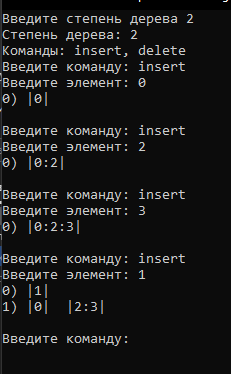


1. Вставляем более 3 элементов, для проверки разбиения корня

Ожидаемы результат: Вывод нескольких узлов

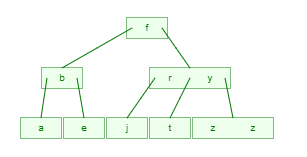


Результат:

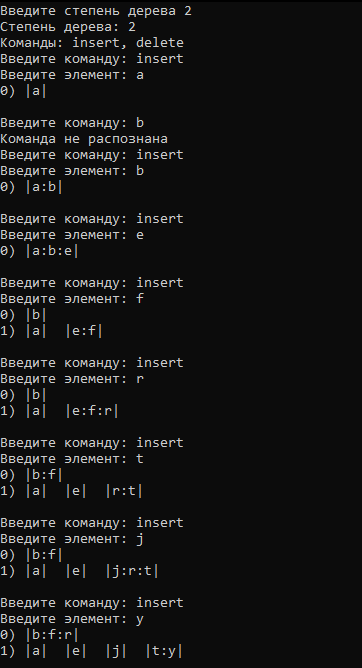


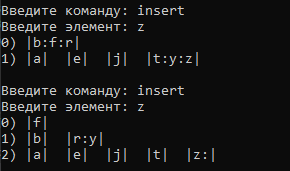
3.Вставляем несколько элементов для проверки разбиения узла

Ожидаемы результат:



Результат:

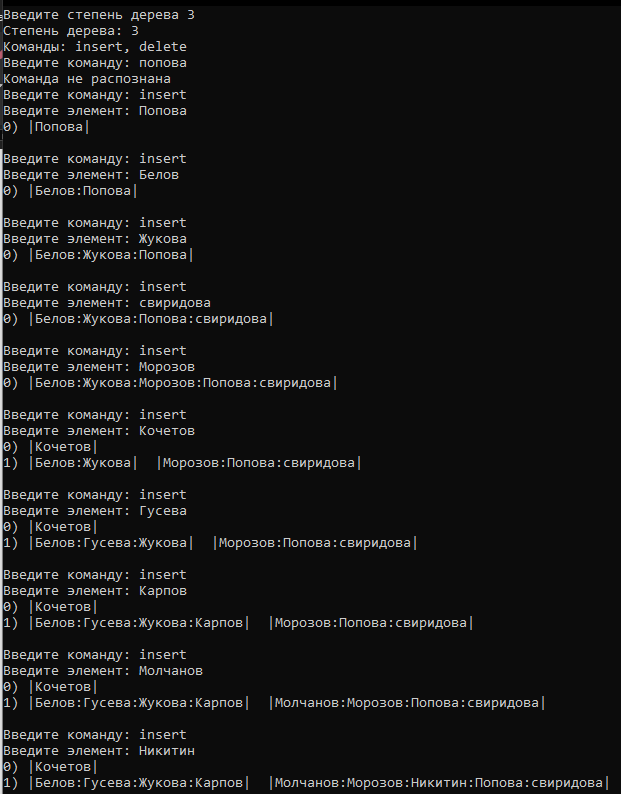




4.Изменим степень дерева на 3 и вставим фамилии

Ожидаемый результат: 

Результат:

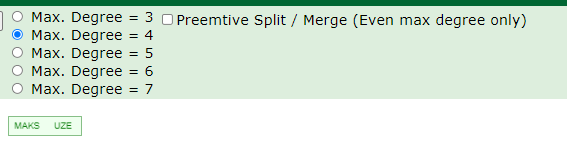


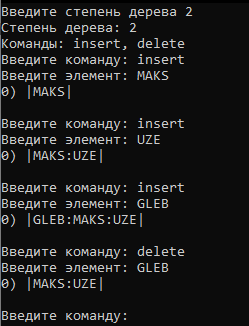
Сайты для проверки:

<https://www.cs.usfca.edu/~galles/visualization/BTree.html>

<https://randomus.ru/name>

5. Удалим элемент из листа





# Текст программы.

#include <iostream>

#include <windows.h>

#include <string>

using namespace std;

struct BTree {

bool full = false;//показывает заполненность узла

string\* Data;//массив ключей

BTree\* Prev = nullptr; //пердыдущий элемент

BTree\*\* Next;//дочерние элементы

int countData = 0;//счетчик ключей

int countNext = 0;//счетчик дочерних элементов

};

void simple\_insert(string x, BTree\* p, int N) {

p->countData++; //увелечиваем счетчик элементов

if (p->countData == 2 \* N - 1) p->full = true; //проверяем заполненность

if (x < p->Data[0]) {//если элемент для вставки меньше 1 элемента

for (int j = (2 \* N) - 1; j > 0; j--) p->Data[j] = p->Data[j - 1];

for (int j = (2 \* N); j > 0; j--) p->Next[j] = p->Next[j - 1];

p->Data[0] = x;

p->Next[0] = nullptr;

}

else

if (p->countData == 1) {

//если контейнер пустой

p->Data[0] = x;

p->Next[0] = nullptr;

}

else

if (x > p->Data[p->countData - 2]) {

//если элемнт для вставки больше крайнего элемента

p->Data[p->countData - 1] = x;

p->Next[p->countData - 1] = nullptr;

}

else

for (int i = 0; i < p->countData - 2; i++) {

if (x > p->Data[i] && x < p->Data[i + 1]) {//поиск места в узле

//смещение элементов после вставки

for (int j = (2 \* N) - 1; j > i + 1; j--) p->Data[j] = p->Data[j - 1];

for (int j = (2 \* N); j > i + 1; j--) p->Next[j] = p->Next[j - 1];

p->Data[i + 1] = x;

p->Next[i + 1] = nullptr;

break;

}

}

}

void drob(BTree\* p, int N) {//разбиение узла

BTree\* ch1 = new BTree;

//создаем новый элемент который будет содеражать первые N-1 элементов

ch1->Data = new string[2 \* N];

ch1->Next = new BTree \* [2 \* N + 1];

for (int i = 0; i < N - 1; i++) {//копируем в него первые N-1 элементов

ch1->countData++;

ch1->Data[i] = p->Data[i];

}

for (int i = 0; i <= 2 \* N; i++) ch1->Next[i] = nullptr;

if (p->countNext > 0) { //перенаправляем связи дочерних элементов

for (int i = 0; i < N; i++) {

ch1->Next[i] = p->Next[i];

if (ch1->Next[i] != nullptr) ch1->Next[i]->Prev = ch1;

}

ch1->countNext = N - 1;

}

if (ch1->countData == 2 \* N - 1) ch1->full = true;

//создаем новый элемент который будет содеражать последние N-1 элементов

BTree\* ch2 = new BTree;

ch2->Data = new string[2 \* N];

ch2->Next = new BTree \* [2 \* N + 1];

for (int i = 0; i < N && (i + N) < p->countData; i++) {//копируем в него последние N-1 элементы

ch2->countData++;

ch2->Data[i] = p->Data[i + N];

}

for (int i = 0; i <= 2 \* N; i++) ch2->Next[i] = nullptr;

if (p->countNext > 0) {//перенаправляем связи дочерних элементов

for (int i = 0; i <= N; i++) {

ch2->Next[i] = p->Next[i + N];

if (ch2->Next[i] != nullptr) ch2->Next[i]->Prev = ch2;

}

ch2->countNext = N;

}

if (p->Prev != nullptr) {//если текущий элемент не голова дерева

simple\_insert(p->Data[N - 1], p->Prev, N);

ch1->Prev = p->Prev; //соеденяем с пред узлом

ch2->Prev = p->Prev;

for (int i = 0; i <= 2 \* N; i++) {

if (p->Prev->Next[i] == p) {

p->Prev->Next[i] = nullptr;

break;

}

}

for (int i = 0; i <= 2 \* N; i++) {//перенаправляем связь от родителя

if (p->Data[N - 1] == p->Prev->Data[i]) {

p->Prev->Next[i] = ch1;

p->Prev->countNext++;

if (p->Prev->Next[i + 1] == nullptr) p->Prev->Next[i + 1] = ch2;

else {

for (int j = 2 \* N; j > i; j--) p->Prev->Next[j] = p->Prev->Next[j - 1];

p->Prev->Next[i + 1] = ch2;

}

break;

}

}

if (p->Prev->countData == 2 \* N) drob(p->Prev, N);

delete p;

}

else {

//если текущий элемент это голова дерева

p->Data[0] = p->Data[N - 1];

p->Next[0] = ch1;

p->Next[1] = ch2;

for (int i = 1; i < p->countData; i++) p->Data[i] = "";

for (int i = 2; i <= 2 \* N; i++) p->Next[i] = nullptr;

p->countData = 1;

p->countNext = 2;

ch1->Prev = p;

ch2->Prev = p;

p->full = false;

}

}

void search(string x, BTree\* p, int N) {

if (p->countData == 0) simple\_insert(x, p, N);//если узел пустой просто вставляем

else

//или ищем позицию для вставки

if (x >= p->Data[p->countData - 1]) {

if (p->full && p->Next[p->countData] == nullptr) {

//вставка если узел полон

simple\_insert(x, p, N);

drob(p, N);

}

else {

if (p->Next[p->countData] == nullptr)simple\_insert(x, p, N);//если есть место в узле

else

search(x, p->Next[p->countData], N);//рекурсивный поиск по ребенку

}

return;

}

else {

for (int i = 0; i < p->countData; i++) {

if (x < p->Data[i]) {

if (p->Next[i] == nullptr) {

if (p->full) {

//вставка если узел полон

simple\_insert(x, p, N);

drob(p, N);//разделяем узел

}

else simple\_insert(x, p, N);//если есть место в узле

}

else {

search(x, p->Next[i], N);//рекурсивный поиск по ребенку

}

return;

}

}

}

}

void output(string str[], int l, BTree\* p, int& max\_level, int N) {

if (!p) {//если пусто то завершаем функц-ю

return;

}

if (max\_level < l) max\_level = l; //увеличиваем максимальный уровень для вывода

str[l] += "|";//создаем границу

for (int i = 0; i < p->countData; i++) {

if (i == p->countData - 1) str[l] = str[l] + p->Data[i] + "| ";

else

str[l] = str[l] + p->Data[i] + ":";//копируем элемент в строку

}

for (int i = 0; i < 2 \* N + 1; i++) {//если есть дочерние элементы, рекурсивно переходим в них

if (p->Next[i] != nullptr) output(str, l + 1, p->Next[i], max\_level, N);

}

}

void simple\_del(string x, BTree\* p, int N) {

for (int i = 0; i < p->countData; i++) {

if (p->Data[i] == x) {//ищем элемент в узле

for (int j = i; j < p->countData; j++) {//смещаем ключи и дочерние элементы

p->Data[j] = p->Data[j + 1];

p->Next[j] = p->Next[j + 1];

}

p->Data[p->countData - 1] = "";//делаем пустой последний ключ

if (p->Next[p->countNext] != nullptr) {//смещаем самый правый дочерний элемент,если он есть

p->Next[p->countNext - 1] = p->Next[p->countNext];

p->Next[p->countNext] = nullptr;

}

p->countData--;//уменьшаем счетчик ключей

break;

}

}

if (p->countData < 2 \* N - 1 && p->full) p->full = false;

}

void leaf\_del(string x, BTree\* p, int N, BTree\* head) {

for (int i = 0; p->countData; i++) {

if (p->Data[i] == x) {//ищем ключ в узле

int j = 0;

for (; j < 2 \* N + 1; j++) if (p->Prev->Next[j] == p) break;//ищем текущий узел в родительском узле

if (p->Prev->Next[j + 1] != nullptr) {//если существует узел справа

if (p->Prev->Next[j + 1]->countData > N - 1) {//если в узле справа больше N-1 ключей

simple\_del(x, p, N);

simple\_insert(p->Prev->Data[j], p, N);//вставляем элемент предшествующий текущему

p->Prev->Data[j] = p->Prev->Next[j + 1]->Data[0];//меняем его же на первый элемент в правом узле

simple\_del(p->Prev->Next[j + 1]->Data[0], p->Prev->Next[j + 1], N);//удаляем предшествующий узлу

break;

}

else {

simple\_del(x, p, N);//удаляем ключ из узла

while (p->countData > 0) { //соеденяем текущий узел с узлом справа

simple\_insert(p->Data[0], p->Prev->Next[j + 1], N);

simple\_del(p->Data[0], p, N);

}

simple\_insert(p->Prev->Data[j], p->Prev->Next[j + 1], N);//вставляем предшествующий элемент в узел справа

simple\_del(p->Prev->Data[j], p->Prev, N);//удаляем его из предшествующего узла

delete p; //удаляем узел,т к мы его перенесли в узел справа

break;

}

}

if (p->Prev->Next[j - 1] != nullptr) {//тоже самое делаем если нет соседа справа,но есть узел слева

if (p->Prev->Next[j - 1]->countData > N - 1) {

simple\_del(x, p, N);

simple\_insert(p->Prev->Data[j], p, N);

p->Prev->Data[j] = p->Prev->Next[j - 1]->Data[p->Prev->Next[j - 1]->countData-1];

simple\_del(p->Prev->Next[j - 1]->Data[p->Prev->Next[j - 1]->countData - 1], p->Prev->Next[j - 1], N);

break;

}

else {

simple\_del(x, p, N);

while (p->countData > 0) {

simple\_insert(p->Data[0], p->Prev->Next[j - 1], N);

simple\_del(x, p, N);

}

simple\_insert(p->Prev->Data[j], p->Prev->Next[j - 1], N);

simple\_del(p->Prev->Data[j], p->Prev, N);

delete p;

break;

}

}

}

}

}

void knot\_del(string x, BTree\* p, int N) {

for (int i = 0; p->countData; i++) {

if (p->Data[i] == x) {//ищем элемент в узле

if (p->Next[i]->countData > N-1) {//если в левом узле больше N-1 эмлементов, переносим вместо удалймого ключа, самый правый элемент дочернего узла

p->Data[i] = p->Next[i]->Data[p->Next[i]->countData - 1];

simple\_del(p->Next[i]->Data[p->Next[i]->countData - 1], p->Next[i], N);

}

else {

if (p->Next[i + 1]->countData > N-1) {//если в правом узле больше N-1 эмлементов, переносим вместо удалймого ключа, самый левый элемент дочернего узла

p->Data[i] = p->Next[i + 1]->Data[0];

simple\_del(p->Next[i + 1]->Data[0], p->Next[i + 1], N);

}

else {

while (p->Next[i + 1]->countData > 0) {//если в дочерних элементах N-1 ключей или менее, соеденяем их и переносим самый левый ключ на место удаляймого элемента

simple\_insert(p->Next[i + 1]->Data[0], p->Next[i], N);

simple\_del(p->Next[i + 1]->Data[0], p->Next[i + 1], N);

}

p->Data[i] = p->Next[i]->Data[0];

simple\_del(p->Next[i]->Data[0], p->Next[i], N);

p->Next[i + 1] = nullptr;

delete p->Next[i + 1];//удаляемый правый дочерний узел

}

}

break;

}

}

}

void Del(string x, BTree\* p, int N, BTree\* head) {

for (int i = 0; i < p->countData; i++) {//ущем подходящий узел, так же как в алгоритме поиска

if (x == p->Data[i]) {

if (p == head && p->Next[i] == nullptr && i == 0|| p == head && p->Next[i] == nullptr) {//если ключ находится в узле-вершине

simple\_del(x, p, N);

break;

}

if (p->countData >= N && p->Next[i] == nullptr) simple\_del(x, p, N);//если у ключа нет дочерних узлов и в узле более N-1 ключей

else

if (p->Next[i] == nullptr) {

leaf\_del(x, p, N, head);//если у ключа нет дочерних узлов

}

else

knot\_del(x, p, N);//если у ключа есть дочерние узлы

break;

}

if (x < p->Data[i]) {//если ключа не существует

if (p->Next[i] == nullptr) cout << "Элемент не найден";

else Del(x, p->Next[i], N, head);

break;

}

if (x > p->Data[i] && p->countData - 1 == i) {

if (p->Next[i + 1] == nullptr) cout << "Элемент не найден";

else Del(x, p->Next[i + 1], N, head);

break;

}

}

}

int main() {

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

string s;

cout << "Введите степень дерева ";

getline(cin, s);//считываем степень дерева

const int N = stoi(s);

cout << "Степень дерева: " << N << endl;

BTree\* head = new BTree;//создаем вершину дерева

head->Data = new string[2 \* N];

head->Next = new BTree \* [2 \* N + 1];

head->Prev = nullptr;

for (int i = 0; i < 2 \* N + 1; i++) {

head->Next[i] = nullptr;

}

int max\_level = 0;//задаем максимальную глубину, для дальнейшего вывода

string comand;//строка для ввода команд

cout << "Команды: insert, delete" << endl;

cout << "Введите команду: ";

for (getline(cin, comand); s != ""&&comand!=""; cout << "Введите команду: ", getline(cin, comand)) {

if (comand == "delete" || comand == "insert") {//считываем команду

cout << "Введите элемент: ";

getline(cin, s);//считываем элемент

if (comand == "delete") Del(s, head, N, head);

else

if (comand == "insert") search(s, head, N);

string str[1000];

output(str, 0, head, max\_level, N);

for (int i = 0; i <= max\_level; i++) {

cout << i << ") " << str[i] << endl;

}//осуществляем вывод дерева по уровням

cout << endl;

}

else

cout << "Команда не распознана" << endl;//если команда не распознана-просим повторить ввод

}

}