Министерство Образования и Науки РФ

Новосибирский Государственный Технический Университет

Кафедра вычислительной техники

Курсовая работа

по дисциплине «Информатика»

на тему «Древовидные структуры»

Факультет: АВТФ

Группа: АВТ-010

Студент: Кокинос А.К.

Преподаватель: Романов Е.Л.

Новосибирск

2011 г.

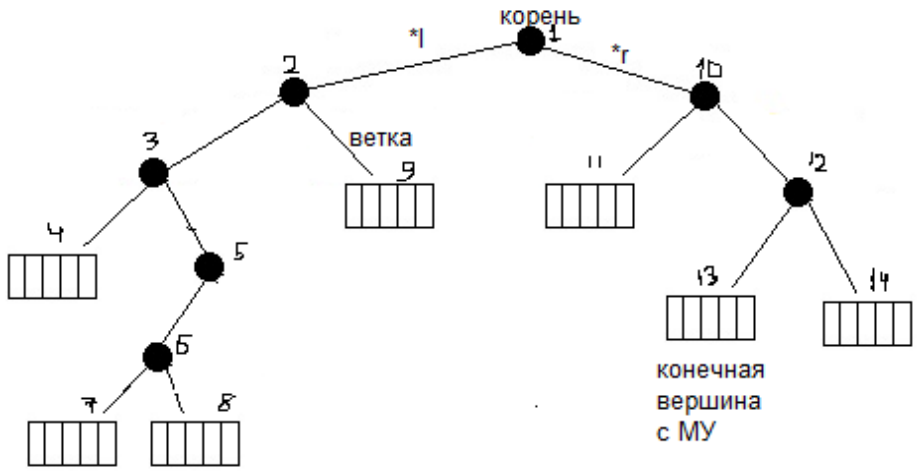
Задание

Написать алгоритм создания дерева, концевая вершина которого содержит внешний (динамический) массив указателей постоянной размерности, а промежуточная - 2 указателя на потомков и счетчик вершин в поддереве. При переполнении концевой вершины она становится промежуточной и порождает два концевых потомка. Предусмотреть вывод характеристик сбалансированности дерева (средняя длина ветви) и процедуру выравнивания (балансировки). При выполнении работы произвести измерение зависимости «грязного» времени работы программы и ее трудоемкости (количества базовых операций). Оценить вид полученной зависимости (линейно-логарифмическая, квадратичная).

Структурное описание разработки

В данной работе была реализована структура данных - двоичное дерево, концевая вершина которого содержит динамический массив указателей на строки.

В самом общем смысле деревом называется набор элементов одного типа с нециклическими однонаправленными непересекающимися связями. В дереве реализована иерархическая структура представления данных: чтобы попасть на какой-то нужный уровень, нужно пройти все предыдущие. Элемент дерева называется вершиной. Связь между двумя вершинами называется ветвью. Вершина, которая ссылается на текущую вершину, называется предком. Вершины, на которые ссылается текущая вершина, называются потомками. Вершина, у которой нет предка, называется корнем. Вершина, у которой нет потомков, называются конечной. Каждый потомок образует свое поддерево, являясь в нем корнем. Каждая вершина двоичного дерева имеет только 2 потомка левый и правый.



Функциональное описание.

В данной работе использовалась структура `struct tree{}`, которая представляет собой вершину дерева. В вершине дерева содержатся: счетчик вершин в поддереве, МУ, и указатели на потомков. Так же в структуру «вшиты» функции добавления новых вершин.

```
struct tree{

int n;//Количество строк-значений в поддереве
char **s; //МУ
tree *l,*r;//Указатели: левый потомок и правый
void creat()//создание пустой вершины

{
r=l=NULL;
n=NULL; s=new char*[N];
for(int i=0;i<N;i++) s[i]="null";
}
void create(char** s0,int n0) //создание вершины со значением
{
r=l=NULL;
n=n0;
s=s0;
}};
```

Дерево будет строиться путем создания связей (ветвей) между вершинами. Указатели на левый и правый потомков будут хранить в себе адрес на этих потомков и, если нам нужно будет перейти к какой-то вершине, мы на неё ссылаемся через указатель.

Движение по дереву носит в себе рекурсивный характер. Каждый раз, когда мы делаем рекурсивный вызов, мы идем вглубь дерева, а когда происходит `return` – возвращаемся.

`void MENU()` – функция вызова меню для удобного пользования

`void exteud()` – функция создания новых вершин при переполнении МУ

`void clearl()` – функция удаления пустой левой вершины

`void clearr()` – функция удаления пустой правой вершины

`void insert()` – функция вставки строки по логическому номеру

`int delete0m()` – функция удаления строки по логическому номеру

`void show()` – функция вывода дерева с указанием глубины

`void show2()` – функция вывода всех значений дерева с указанием их логического номера

`void add()` – функция добавления строки в самый конец дерева

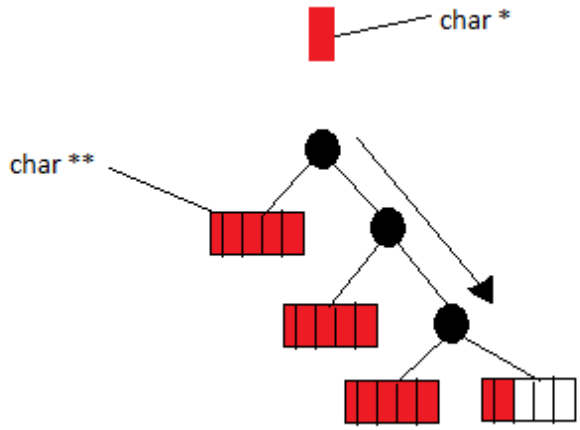
`void scan()` – функция обхода дерева и перепись значений всех вершин в МУ

`void destroy()` – функция удаления дерева

`tree *create()` – функция создания сбалансированного дерева из значений, хранящихся в МУ

Описание работы программы.

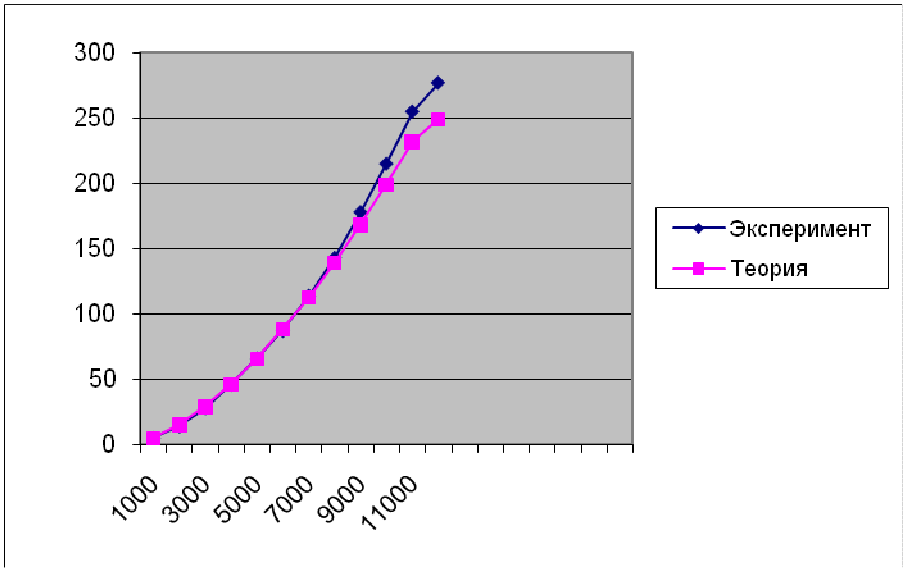
Дерево заполняется строками из файла. Из текстового файла читается слово и передается в функцию `void add(tree *p,char *c)` которая «запихивает» слово в самый конец (в самую правую вершину на самое последнее место). Таким образом получается, что мы создаем дерево с одной длинной веткой, что делает более трудоемким процесс обхода дерева.



Теоретическая трудоемкость данного процесса $T(N)=N^2$. Эксперимент для нахождения трудоемкости показал, что:

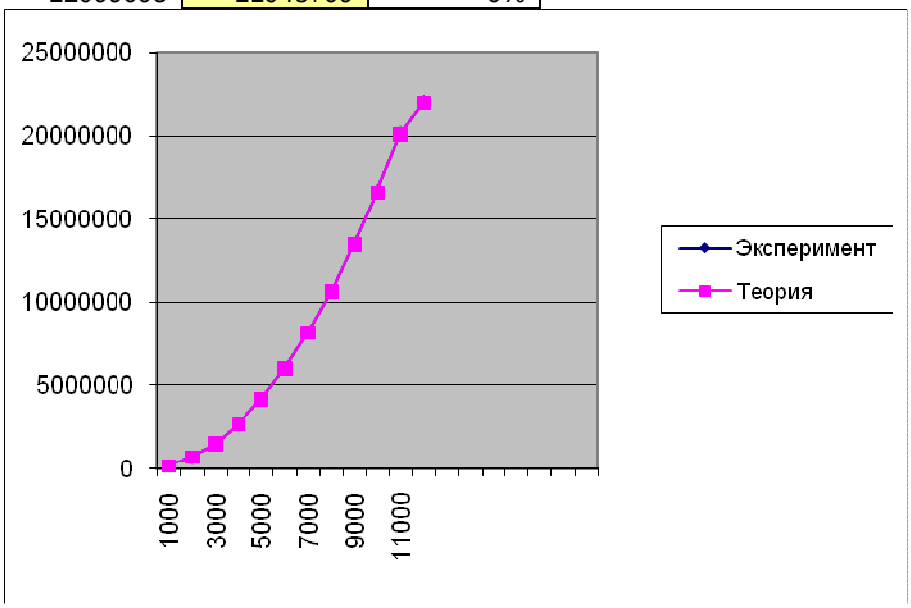
1)Грязное» время работы программы подчиняется степенной зависимости с показателем P=1.6, т.е. лучше чем квадратичная. Отклонение сначала достигает 8% в лучшую сторону, а затем постепенно ухудшается с увеличением длины файла.

N	T1(эскп)	T1*(матем)	отклонение	коэф по N1	Степень
1000	5	5,0	0%	0,0000792447	1,6
2000	14	15,2	-8%		
3000	28	29,0	-3%		
4000	46	45,9	0%		
5000	66	65,7	1%		
6000	87	87,9	-1%		
7000	114	112,5	1%		
8000	143	139,3	3%		
9000	178	168,2	6%		
10000	215	199,1	8%		
11000	255	231,8	10%		
11493	277	248,7	11%		

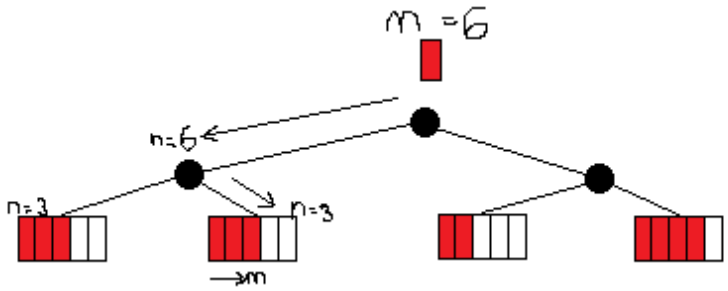


2) Количество операций сравнения практически точно имеет зависимость $T(N)=N^2/6$ (K=0.16), что соответствует нашим предварительным теоретическим оценкам;

N	cmp(эскп)	cmp*(матем)	отклонение	коэф по N1	Степень
1000	166166	166166	0%	0,166166	2
2000	665665	664664	0%		
3000	1498500	1495494	0%		
4000	2664666	2658656	0%		
5000	4164165	4154150	0%		
6000	5997000	5981976	0%		
7000	8163166	8142134	0%		
8000	10662665	10634624	0%		
9000	13495500	13459446	0%		
10000	16661666	16616600	0%		
11000	20161165	20106086	0%		
11493	22009095	21948709	0%		

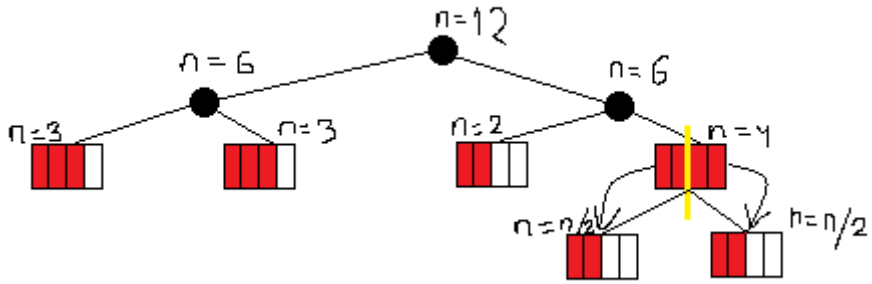


Далее дерево можно «пополнить» дополнительной строкой, указав перед этим номер для вставки. Пользователь вводит строку с клавиатуры и номер места, куда она должна попасть. Данные передаются функции `void insert(tree *p,char *c,int m)`. Функция, цепью рекурсивных вызовов, находит концевую вершину, в которой содержится ячейка с нужным номером, после чего туда помещается введенная строка.



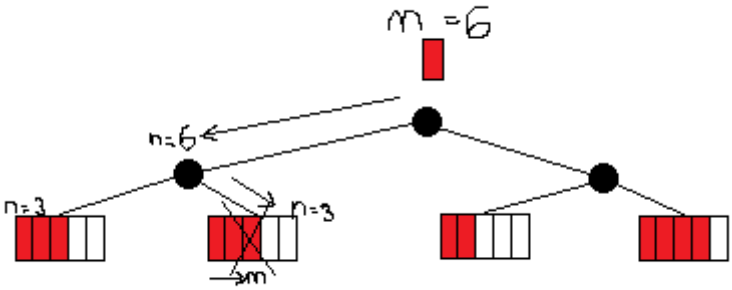
Алгоритм отыскания нужной вершины из двух потомков выбирает одного, поэтому он «жадный». Его трудоемкость будет равна количеству пройденных вершин (длине выбранной ветви). Для сбалансированного дерева зависимость будет логарифмической $T = L < \log_m N$.

Так как массивы указателей имеют ограниченную размерность, то может случиться переполнение. Что бы этого не произошло, была создана функция `void exteud(tree *p)` При заполнении МУ, создается 2 новые вершины с пустыми МУ. Заполненый МУ «делится» на 2 части и эти части записываются в новые МУ.



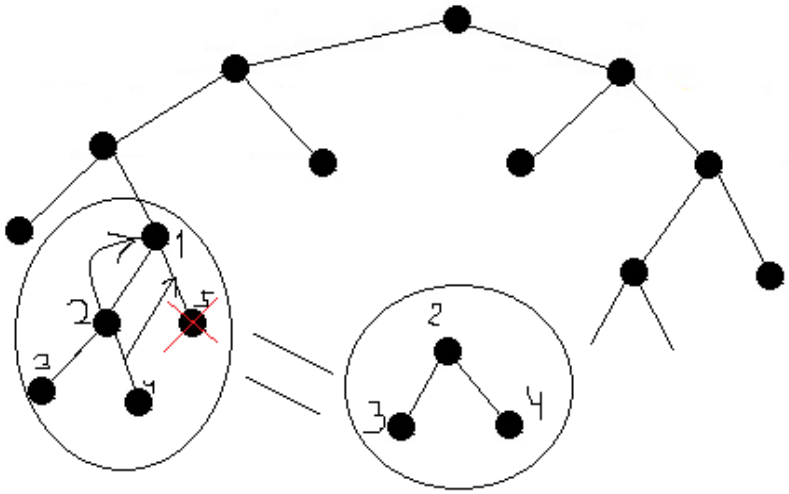
Функция удаления по логическому номеру схожа с функцией вставки. Функции `int delete0m(tree *p, int m)` передается логический номер строки, которую нужно удалить. Цепью рекурсивных вызовов находится вершина, в которой присутствует

нужное значение, после чего производится удаление.



Алгоритм отыскания нужной вершины так же из двух потомков выбирает одного, поэтому он тоже «жадный» и его трудоемкость будет зависеть от количества пройденных вершин. Для сбалансированного дерева зависимость будет логарифмической $T = L<\log_m N$.

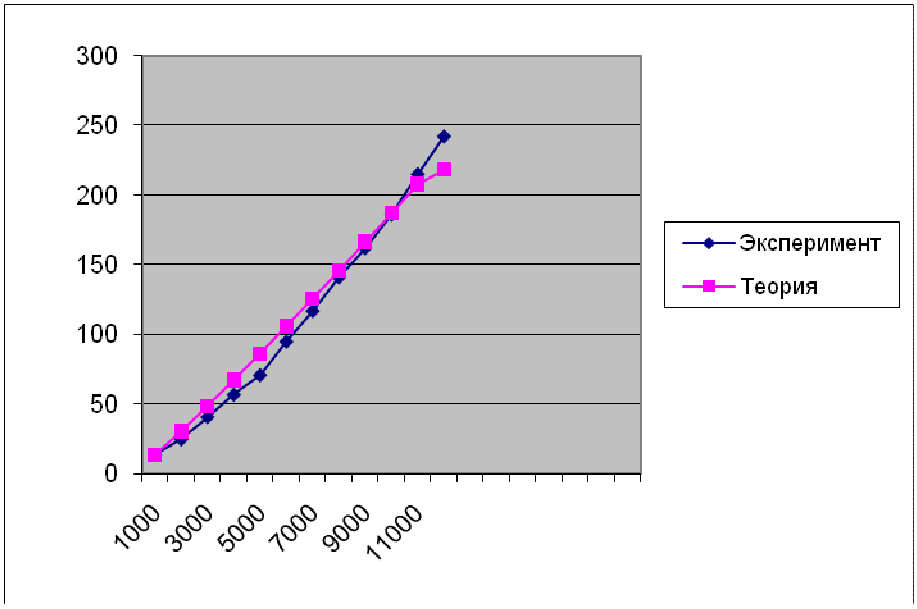
После очереди удалений может образоваться пустая вершина. Эту вершину нужно удалить, но при этом должна остаться прежняя структура в виде двоичного дерева, для этого мы сдвигаем на 1 уровень вверх «брата» удаляемой вершины.



Функция балансировки дерева, созданная для уменьшения трудоемкости обхода дерева. Дерево называется сбалансированным, если длина максимальной и минимальной ветви отличается не более чем на 1. После балансировки все алгоритмы будут иметь меньшую трудоемкость, нежели чем в несбалансированном дереве. Балансировка дерева делится на 3 этапа: 1) Сбор данных с вершин дерева и их запись в специально созданный МУ; 2) Удаление дерева; 3)Создание нового сбалансированного дерева. Для этого были созданы функции: `void scan(tree *p, char **dd, int&m), void destroy(tree *p), tree *create(char **dd, int a, int nn)` . Трудоемкость балансировки найдем экспериментальным путем. Из файла загружаем строки в программу, после каждой 1000-ой строки делаем балансировку. Замеряем время и кол-во операций(рекурсивных вызовов):

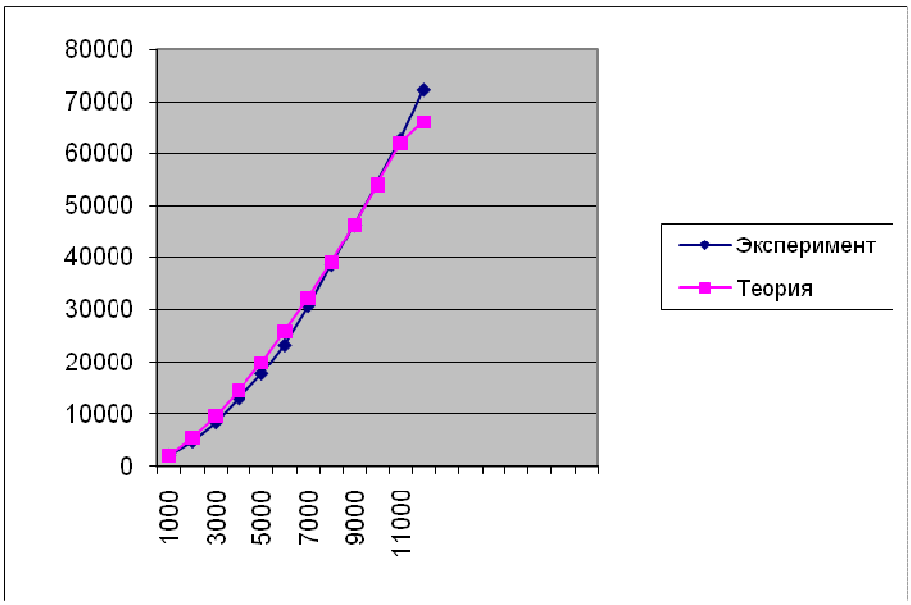
1)«Грязное» время работы программы подчиняется логарифмической зависимости . Отклонение сначала достигает 19% в лучшую сторону, а затем постепенно ухудшается с увеличением длины файла и в конце достигает 11%.

N	T1(эскп)	T1*(матем)	отклонение	коэф по N1
1000	14	14,0	0%	0,0014048066
2000	25	30,8	-19%	
3000	41	48,7	-16%	
4000	57	67,2	-15%	
5000	71	86,3	-18%	
6000	95	105,8	-10%	
7000	117	125,6	-7%	
8000	141	145,7	-3%	
9000	162	166,1	-2%	
10000	186	186,7	0%	
11000	215	207,5	4%	
11493	242	217,8	11%	



2) Количество операций (рекурсивных вызовов) подчиняется зависимости $T(N)=N^{3/2}/12$ (K=0.08) .

N	cmp(эскп)	cmp*(матем)	отклонение	коэф по N1	Степень
1000	1918	1918	0%	0,085674	1,45
2000	4607	5240	-12%		
3000	8248	9434	-13%		
4000	12841	14316	-10%		
5000	17578	19786	-11%		
6000	23200	25773	-10%		
7000	30721	32229	-5%		
8000	38530	39114	-1%		
9000	46339	46398	0%		
10000	54153	54057	0%		
11000	62727	62068	1%		
11493	72464	66142	10%		



Выводы: Деревья – удобная структура представления данных в памяти. Трудоемкость алгоритмов в деревьях могут быть линейными, степенными и логарифмическими.

Приложение.

```
#include <stdio.h>
#include <string.h>
#include <iostream>
#include <conio.h>
#include <time.h>                    // Для функции clock
#include <windows.h>                 // Для функции CharToOemA
#include <dos.h>
using namespace std;
int N=100;
int cnt=0;// Счетчик сравнений
int nn;
long T0=0;
void MENU()
                    // Меню
{
      printf("\nAdd file-1\n");
      printf("Include-2\n");
      printf("Delete-3\n");
      printf("Balans-4\n");
      printf("Exit-0\n");
}

struct tree
{
      int n;            // Значений в поддереве
      char **s; //МУ
      tree *l,*r;//Левый потомок и правый
      void creat(){r=l=NULL; n=NULL; s=new char*[N]; for(int i=0;i<N;i++)
s[i]="null";
      } //создание пустой вершины
void create(char** s0,int n0){ //создание вершины со значением
      r=l=NULL;
      n=n0;
      s=s0;
}};
void exteud(tree *p){
```