Министерство образования и науки РФ

Новосибирский государственный технический университет

Кафедра вычислительной техники

Пояснительная записка к курсовому проекту по дисциплине: «Программирование»

Факультет: АВТ

Группа: АП-518

Студент: Курагин А.В.

Проверил:

Романов Е.Л.

Новосибирск 2007

1.Задание:

Шаблон иерархической структуры данных в памяти (начальный балл 4,0)

Для заданной двухуровневой структуры данных, содержащей указатели на объекты (или сами объекты) - параметры шаблона, разработать полный набор операций (добавление, включение и извлечение по логическому номеру, сортировка, включение с сохранением порядка, загрузка и сохранение строк в текстовом файле, балансировка выравнивание размерностей структур данных нижнего уровня) (см. bk57. rtf). Предполагается, что операции сравнения хранимых объектов переопределены стандартным образом (в виде операций <,> и т.д..). Программа должна использовать шаблонный класс с объектами- строками и реализовывать указанные выше действия над текстом любого объема, загружаемого из файла.

Шаблон структуры данных двоичное дерево. Вершина дерева содержит статический массив указателей на объекты (ограниченный NULL-указателем) и два указателя на правое и левое поддерево. Значения в дереве упорядочены. (массив в каждом элементе упорядочен, дерево в целом также упорядочено). Если при включении указателя в найденный массив последний переполняется, то самый правый указатель переносится в правое поддерево.

2.Структурное описание разработки:

В данной работе была реализована структура данных - двоичное дерево, вершина которого содержит статический массив указателей на объекты.

В самом общем смысле деревом называется набор элементов одного типа с нециклическими однонаправленными непересекающимися связями. 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 терминальными. Каждый потомок образует свое поддерево, являясь в нем корнем.

Типичный пример терминальной вершины представлен в песне Владимира Высоцкого «Камень в степи»:

Лежит камень во степи,

А под него вода течет,

А на камне написано слово:

«*Кто направо пойдет – ничего не найдет,*

А кто прямо пойдет – никуда не придет,

Кто на лево пойдет – ничего не поймет,

И не за грош пропадет»

В данной песне поэт также отразил рекурсивный характер дерева:

Был один из них зол – он направо пошел,

В одиночку пошел – ничего не нашел,

Ни деревни, ни сел и обратно пришел.

Дерево является золотой серединой между списком и массивом. С одной стороны, дерево дает логарифмическую трудоемкость при доступе к элементу, в отличие от списка с его последовательным доступом. С другой стороны, при вставке элемента в середину последовательности дереву, в отличие от массива, нет необходимости сдвигать элементы, освобождая место для нового.

Частный случай дерева – бинарное дерево. Вершина бинарного дерева содержит двух потомков, которые называются левый и правый. Значение левого потомка меньше значения текущей вершины, значение правого потомка больше значения текущей вершины. Для индексирования вершин бинарного дерева применяют следующий принцип: индексация левого поддерева – индексация текущей вершины – индексация правого поддерева. Индексация дает возможность линеаризовать дерево.

Бинарное дерево принципиально отличается от простого дерева с двумя потомками, тем, что бинарное дерево сохраняет свою упорядоченность при добавлении нового элемента, следовательно, отпадает необходимость сортировки структуры данных. Расплатой за это удовольствие является увеличение трудоемкости добавления новой вершины при росте дерева.

Специфичной чертой деревьев является понятие сбалансированности. Задача балансировки обеспечить равенство длин ветвей дерева. Балансировка дерева приводит к росту эффективности операций поиска и добавления новой вершины.

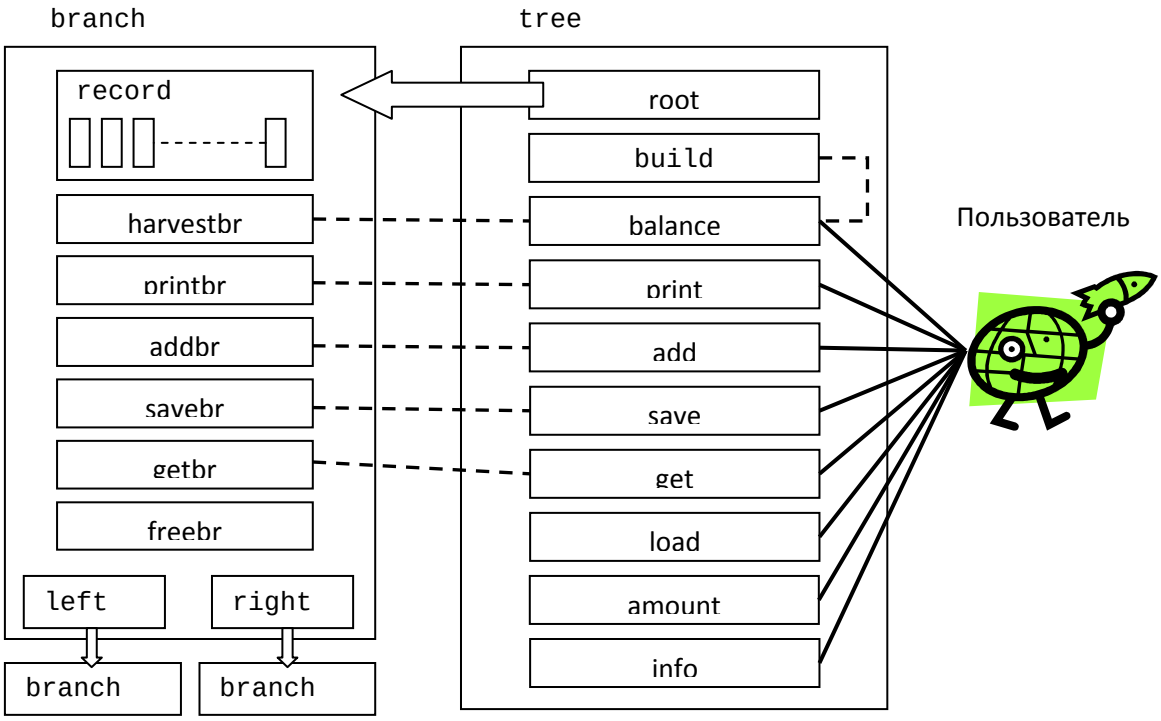
Еще одна неприятная черта двоичных деревьев – возможность выродится в список при вставке упорядоченной последовательности.

Итак, основными претензиями к дереву являются: 1. Трудоемкий доступ к терминальной вершине (в лучшем случае логарифмический). 2. Неэффективность при работе с упорядоченной последовательностью. Частично сгладить эти минусы поможет следующая хитрость: хранить в вершине не одно значение, а массив значений.

3. Функциональное описание:

В данном работе использовалось две структуры: дерево (class tree) и вершина (struct branch). Вершина хранит массив указателей на переменные, количество элементов в поддереве, набор рекурсивных методов и указатели на левого и правого потомка. Дерево хранит указатель на корневую вершину и набор интерфейсных методов.

Схематически организацию модуля можно отобразить следующим образом:



root – указатель на корневую вершину.

balance() – интерфейсный метод балансировки дерева, нуждается в рекурсивном методе сбора элементов дерева harvestbr() и рабочем методе build().

print() – интерфейсный метод вывода дерева на экран, вызывает рекурсивный метод printbr() – вывод вершины на экран.

add() – интерфейсный метод добавления нового элемента в дерево, вызывает соответствующий рекурсивный метод addbr().

get() – интерфейсный метод получения элемента по логическому номеру, вызывает соответствующий рекурсивный метод getbr().

save()/savef() – интерфейсные методы записи элементов дерева в файловый поток/файл вызывают рекурсивный метод savebr – сохранение элементов вершины в файловый поток.

load()/loadf() – чтение значений из файлового потока/файла.

amount() – функция, возвращает число значений в дереве.

info() - побочный метод, выводит число значений в дереве и размерность массивов в вершинах.

record – массив значений в вершине

freebr() – рекурсивный метод, удаляет массивы указателей не удаляя значения на которые они ссылаются (деструктор вершины рекурсивный, но в отличие от freebr() удаляет значения).

left, right – указатели на левею, правую вершины.

Описание алгоритма добавления нового элемента

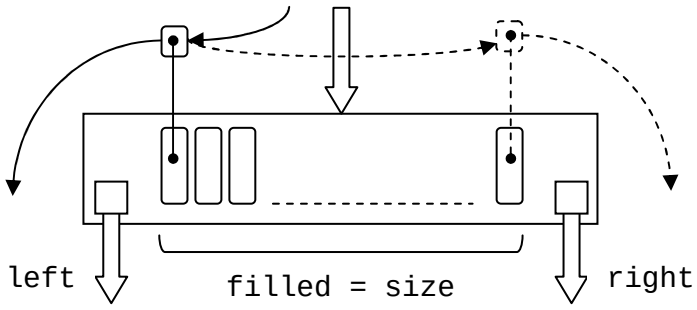
В данном описании значащими будут считаться операции сравнения по значению элементов. Иными операциями, например перестановкой элементов массива указателей, будем пренебрегать, так как их трудоемкость не зависит от определяющего шаблон типа данных. Следует отметить, что структура данных работает с указателями на переменные заданного типа. Память под значение выделяется в интерфейсной функции, после чего в рабочую функцию передается указатель на выделенную память. Данная организация интерфейса класса освобождает пользователя от работы с памятью и дает классу исключительные права на доступ к данным, минусом является необходимость передачи данных по значению и возможное их дублирование.

Интерфейсная функция: **void add(TYPE value);**

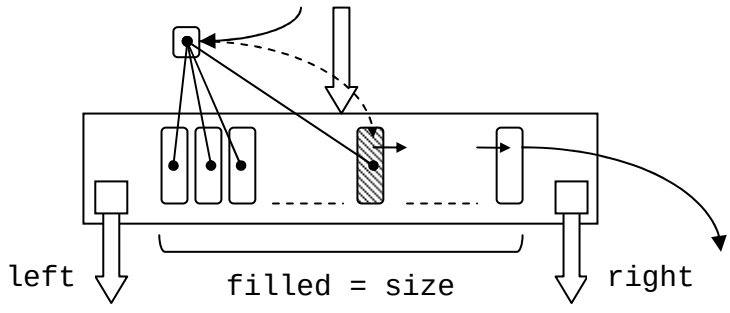
Рабочая функция: **void addbr(TYPE *value, int size);**

Принцип работы рабочей функции:

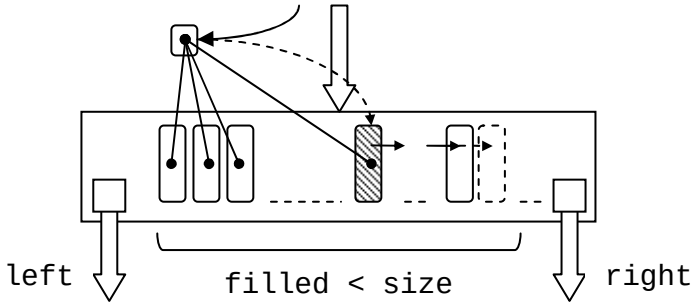
a) Если вершина заполнена полностью и если новый элемент меньше (больше либо равен) первого (последнего). Одна значащая операция (две значащих операций).



b) Если вершина заполнена полностью и новое значение находится между крайних. Значения сравниваются последовательно, начиная с первого, пока не найдется значение больше нового (на схеме операции сравнения обозначены как линии с точками на концах). После того как найден элемент, значение которого больше значения нового элемента (закрашенный прямоугольник), все элементы массива, начиная с найденного, сдвигаются на одну позицию вправо, выталкивая крайний. Новый элемент встает на освободившееся место. Значащих операций: 2 + № позиции найденного элемента + 1.



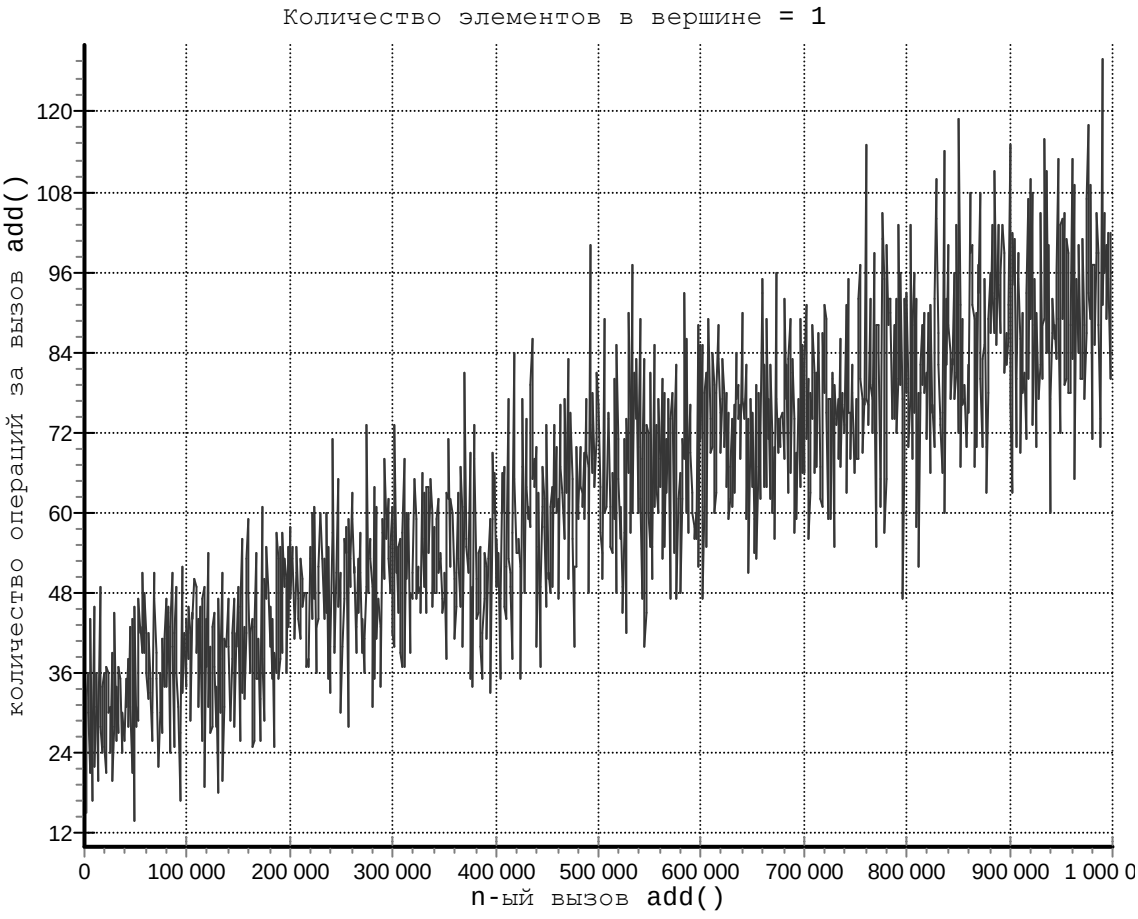
c) Если вершина заполнена не полностью и новое значение находится между крайних. Добавление элемента в массив происходит аналогично вставке с вытеснением крайнего, однако крайний элемент не вытесняется, а остается в массиве передвигаясь на первую свободную позицию (помечено пунктирным прямоугольником). Значащих операций: позиция найденного элемента + 1.

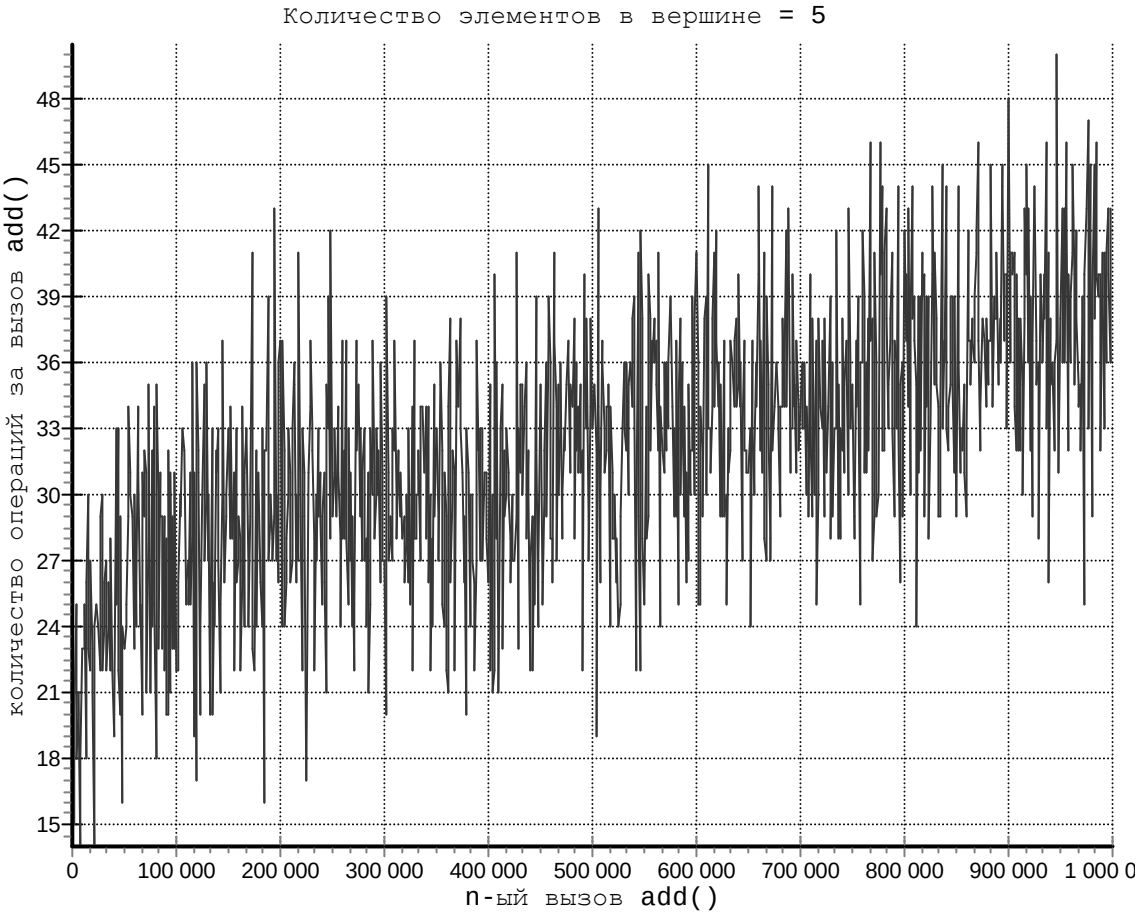


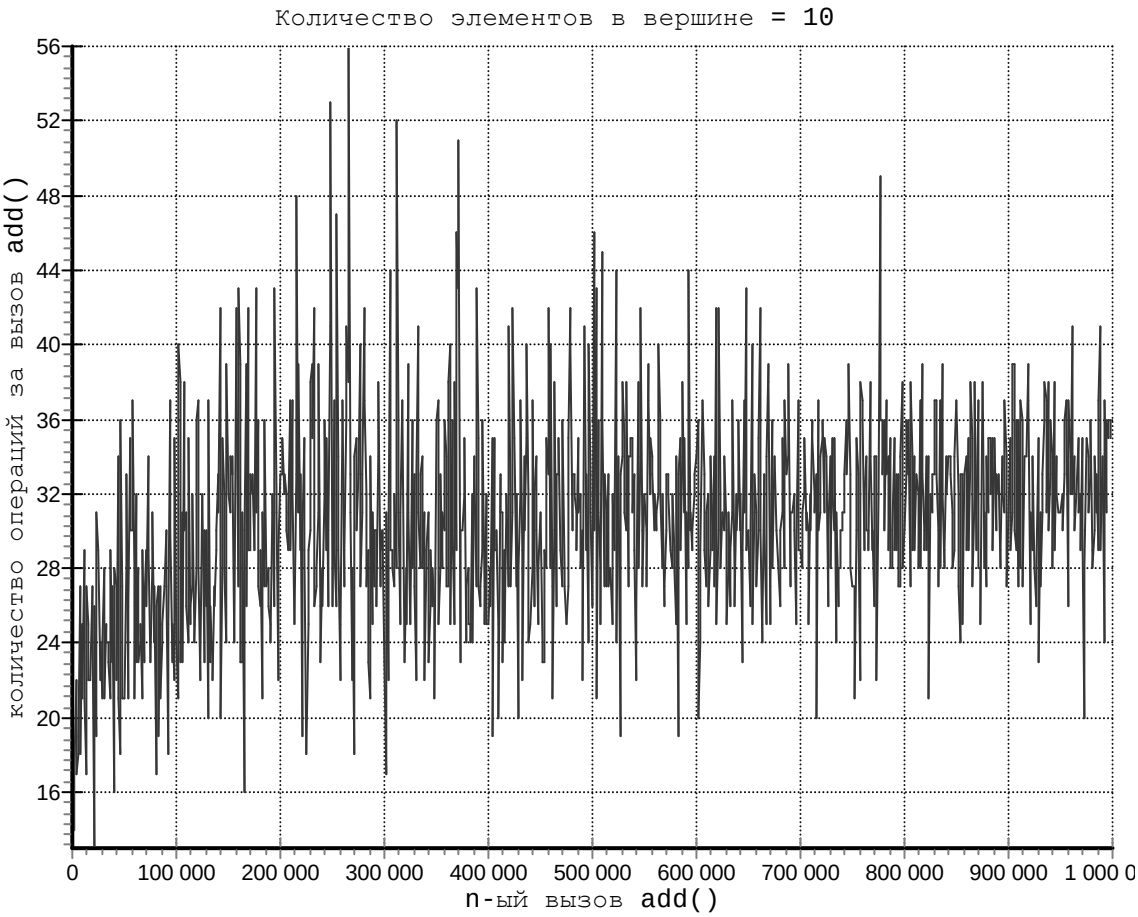
Эксперименты:

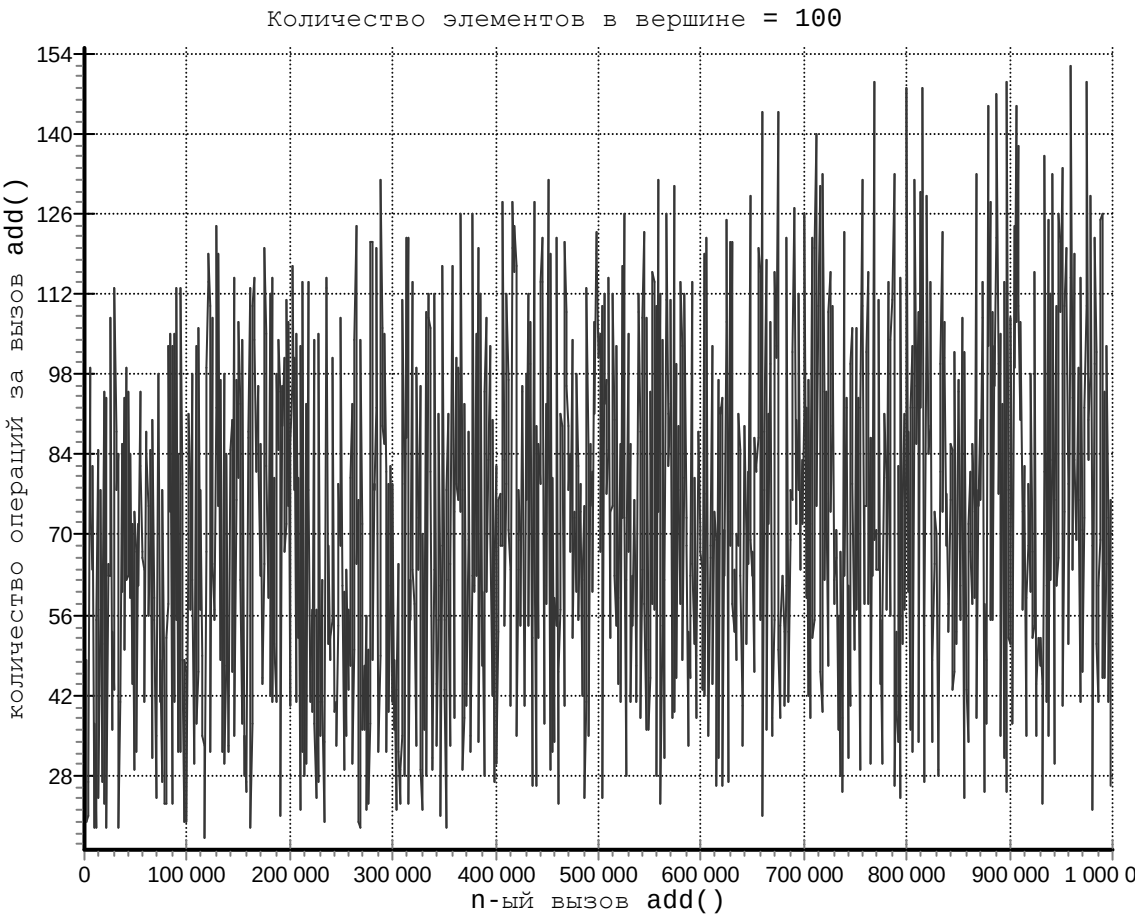
Добавление n-ого элемента

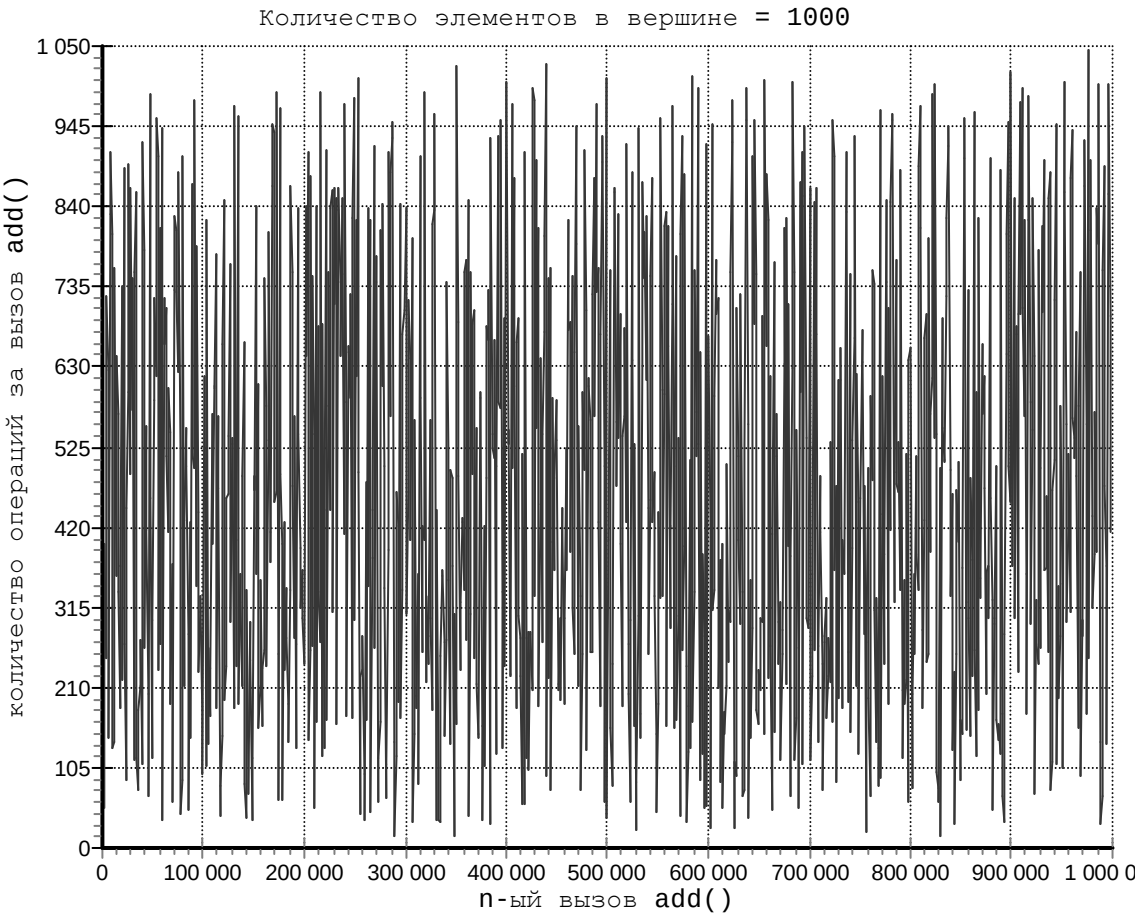
Определим количество значащих операций при добавлении нового элемента. Создадим дерево и заполним его миллионом случайных значений. По оси абсцисс отложим номер вызова метода add(), по оси ординат – количество значащих операций за метод. Будем менять параметр size – размер массива в вершине. (На графике отмечается каждое тысячное значение)




Количество элементов в вершине = 5
количество операций за вызов add()
48
45
42
39
36
33
30
27
24
21
18
15
0
100 000
200 000
300 000
400 000
500 000
600 000
700 000
800 000
900 000
1 000 0
n-ый вызов add()


Количество элементов в вершине = 10
количество операций за вызов add()
56
52
48
44
40
36
32
28
24
20
16
0
100 000
200 000
300 000
400 000
500 000
600 000
700 000
800 000
900 000
1 000 0
n-ый вызов add()

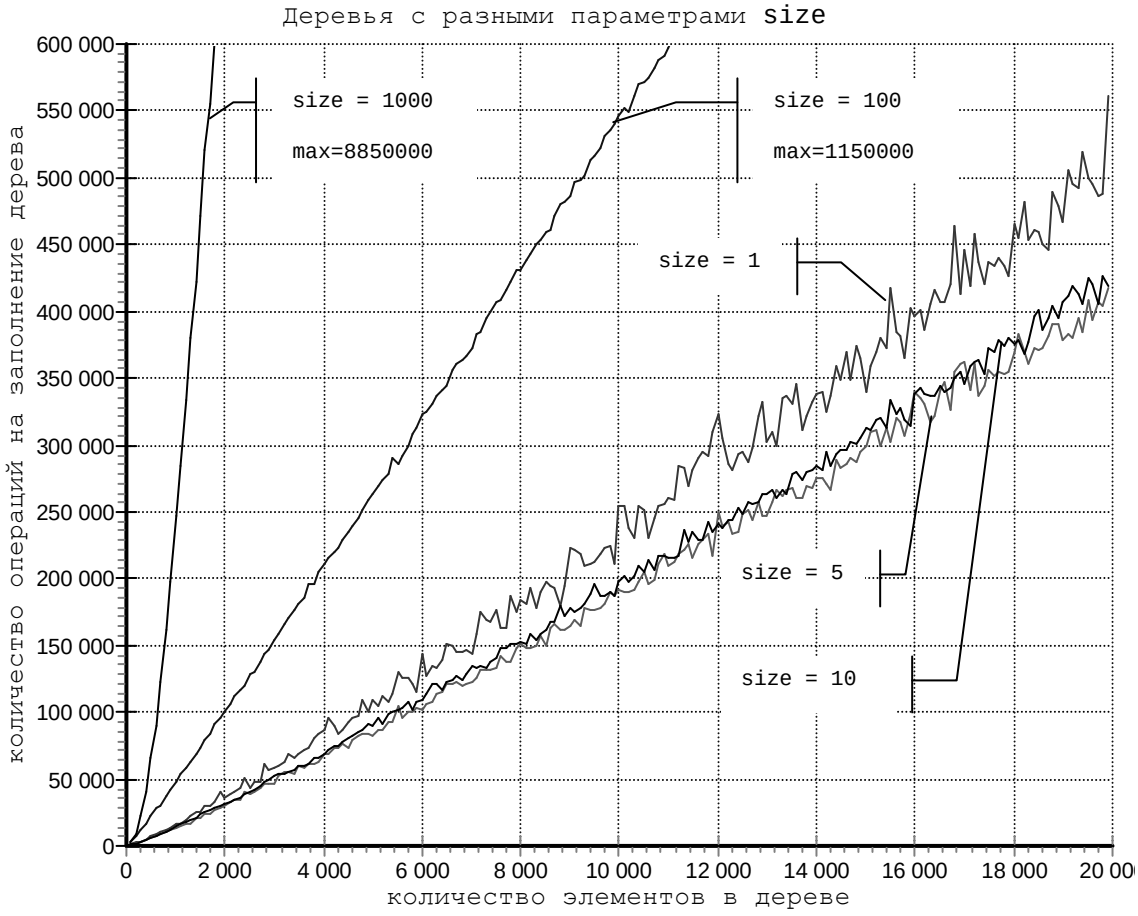

Количество элементов в вершине = 100
количество операций за вызов add()
154
140
126
112
98
84
70
56
42
28
0
100 000
200 000
300 000
400 000
500 000
600 000
700 000
800 000
900 000
1 000 0
n-ый вызов add()


Количество элементов в вершине = 1000
количество операций за вызов add()
1 050
945
840
735
630
525
420
315
210
105
0
0
100 000
200 000
300 000
400 000
500 000
600 000
700 000
800 000
900 000
1 000 0
n-ый вызов add()

В принципе, трудоемкость добавления нового элемента будет расти вместе с ростом дерева, это неизбежно, но вот скорость роста трудоемкости будет различен. Лучше всего линейный характер роста трудоемкости наблюдается при size = 1, size = 5, size = 10. Нельзя не отметить, что при size = 5 количество значащих операций в 2-3 раза меньше чем при size = 1.

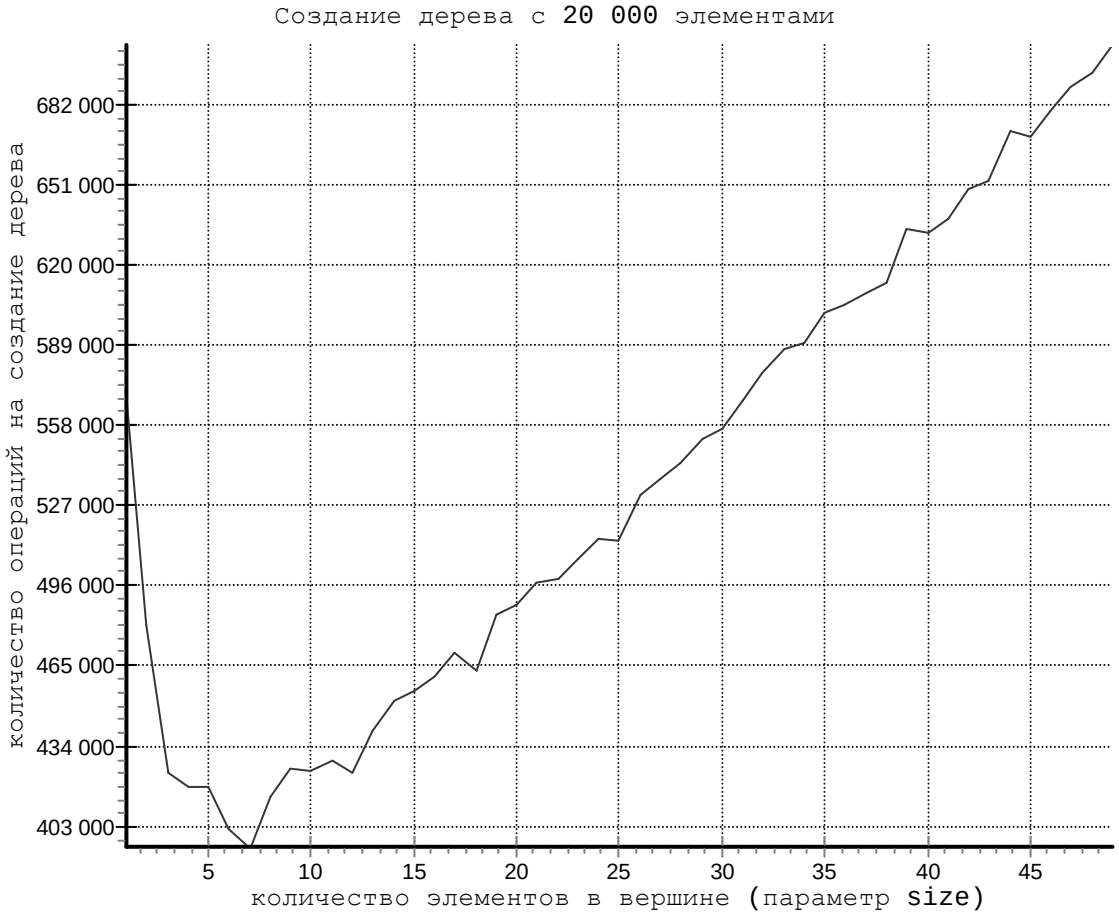
Добавление n элементов

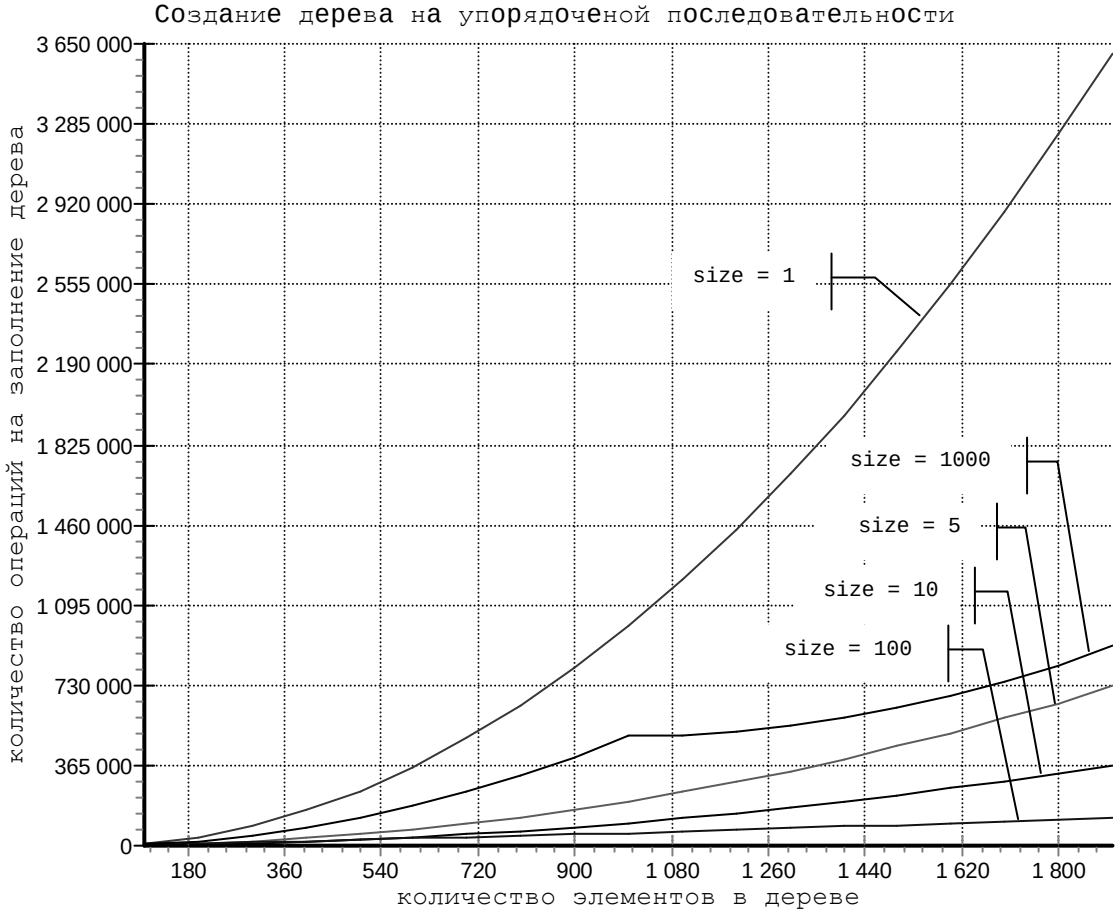
Определим количество значащих операций при создании дерева с n элементами. Создадим дерево и заполним его n=[0: 20000] случайными значениями. По оси абсцисс отложим количество элементов в дереве, по оси ординат - количество значащих операций при создании дерева. Будем менять параметр size - размер массива в вершине. (На графике отмечается каждое сотое значение)



Как и в прошлом эксперименте налицо выгода от использования массивов. Однако, тут как с супом - главное не «пересолить». Уже при использовании массива в 100 элементов трудоемкость возрастет в два раза, а при 1000 элементах - в 16 раз!!!

Для проявления картины происходящего проведем еще два эксперимента: 1. Изменяя параметр size заполним дерево 20 000 элементами, измерим количество операций на создание дерева. 2. Повторим эксперимент « Добавление n элементов» с упорядоченной последовательностью.

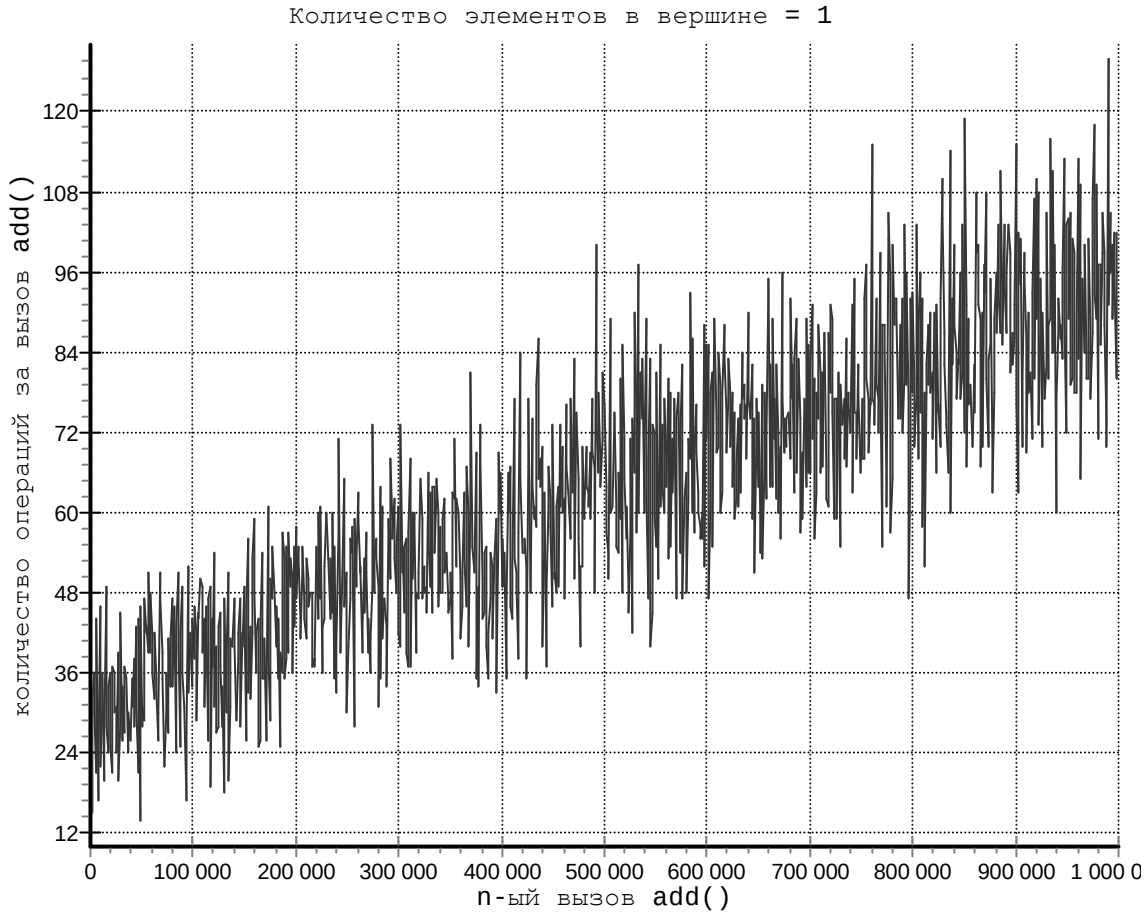

Создание дерева с 20 000 элементами
количество операций на создание дерева
682 000
651 000
620 000
589 000
558 000
527 000
496 000
465 000
434 000
403 000
5
10
15
20
25
30
35
40
45
количество элементов в вершине (параметр size)

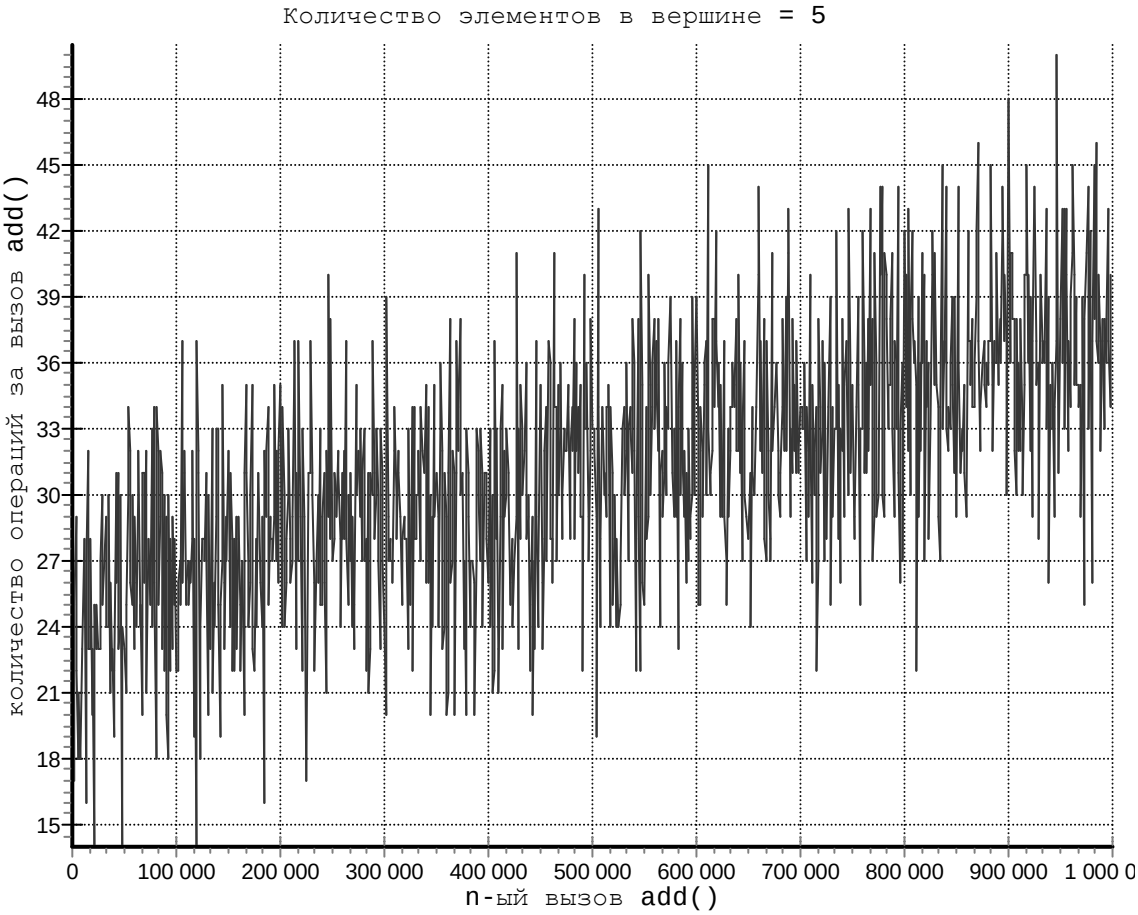

Создание дерева на упорядоченой последовательности
количество операций на заполнение дерева
3 650 000
3 285 000
2 920 000
2 555 000
2 190 000
1 825 000
1 460 000
1 095 000
730 000
365 000
0
size = 1
size = 1000
size = 5
size = 10
size = 100
180
360
540
720
900
1 080
1 260
1 440
1 620
1 800
количество элементов в дереве

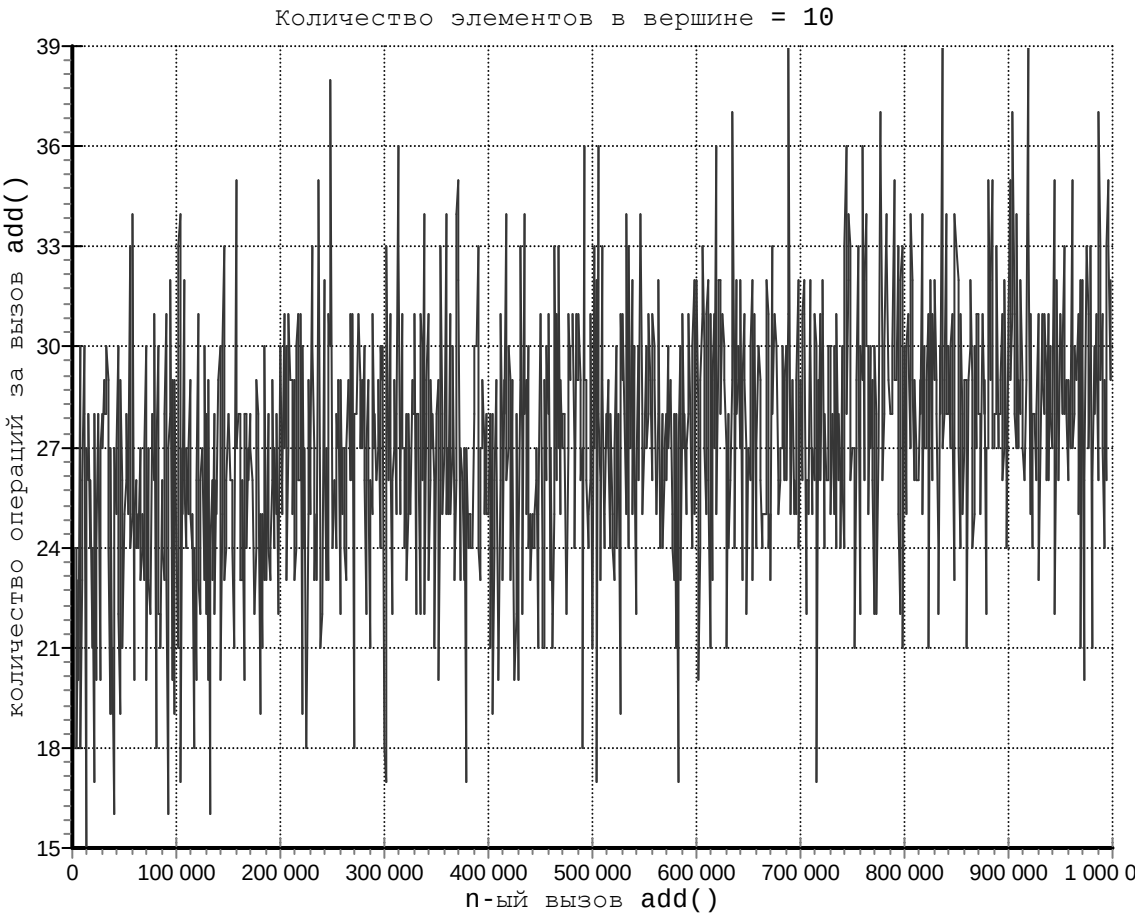
Модернизация алгоритма добавления нового элемента в двоичное дерево.

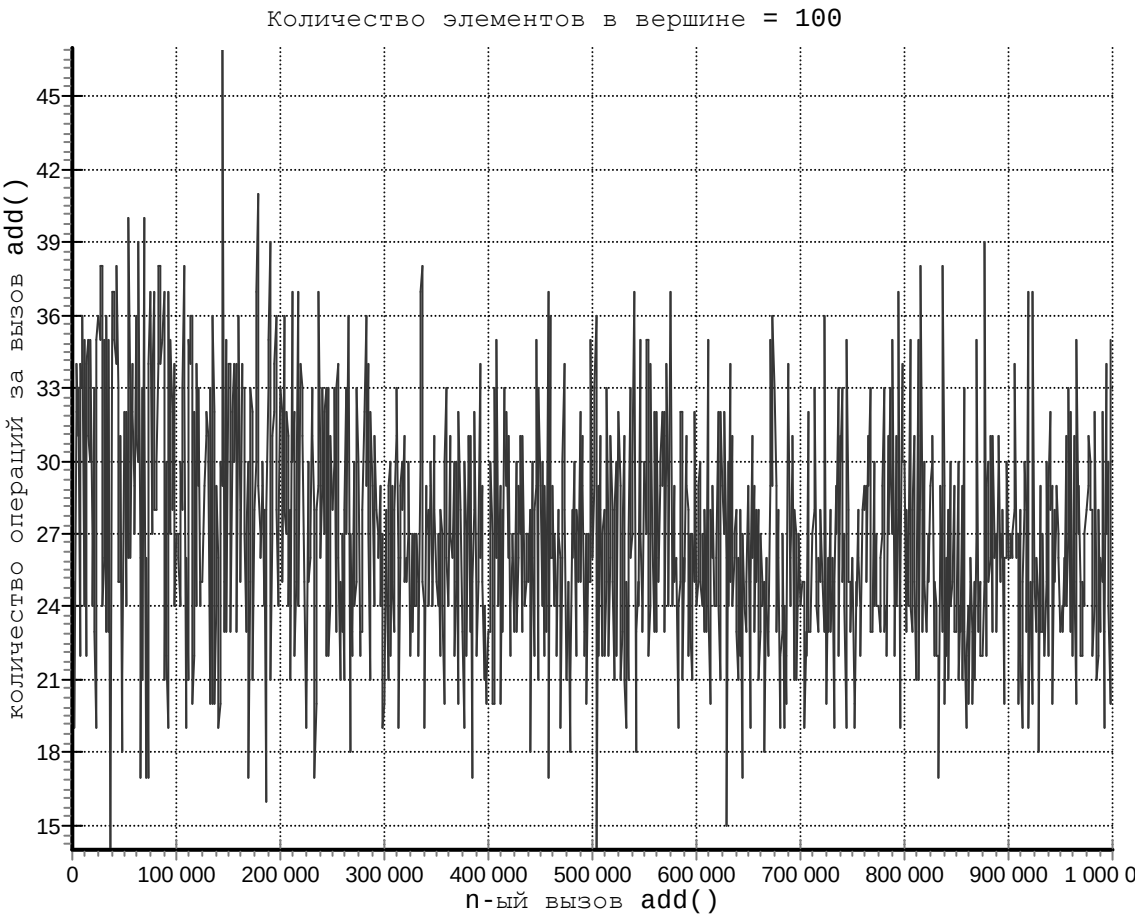
Как видно из прошлых опытов, использование дерева с большой размерностью массива нижнего уровня приводит к снижению эффективности алгоритма. Предложим для поиска значения в массиве нижнего уровня применять не линейный, а бинарный поиск (модернизированный код приведен в приложении). Повторим эксперименты, используя модернизированный метод add().

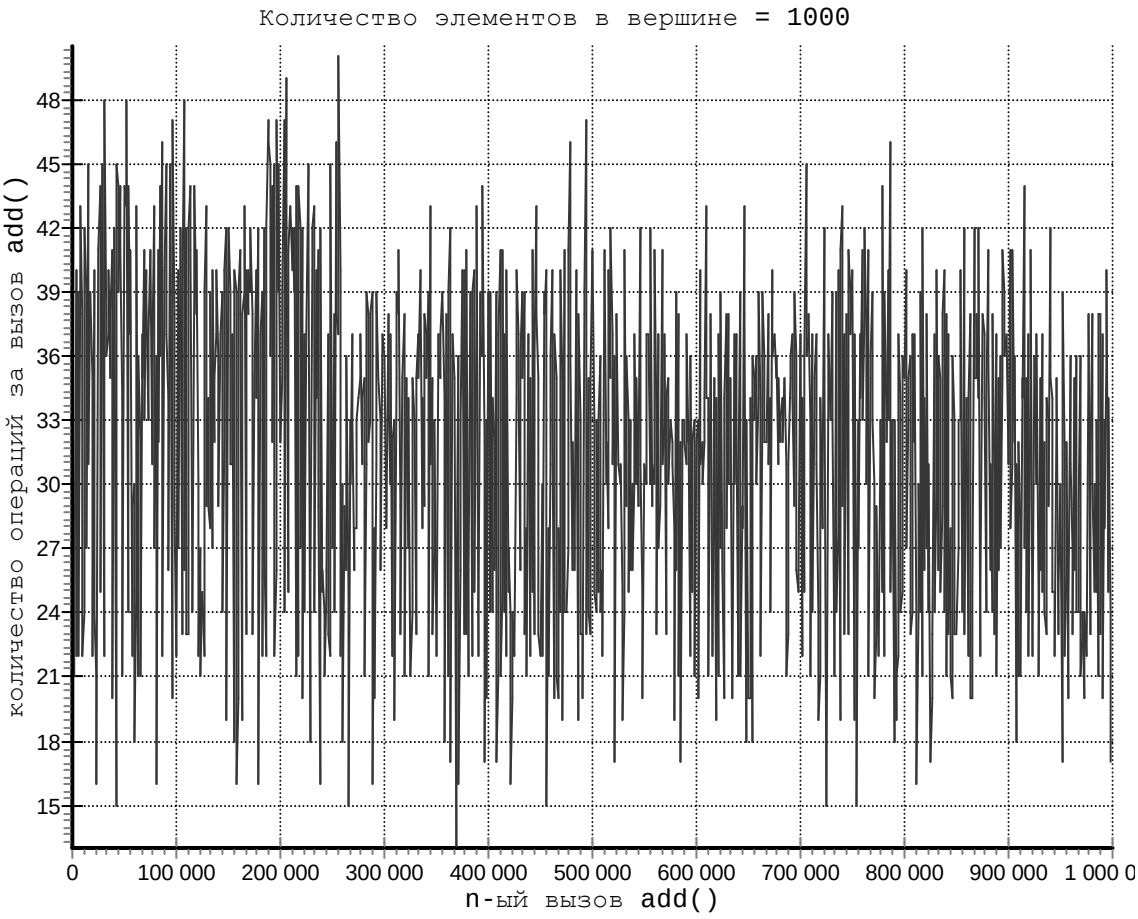
Добавление n-ого элемента



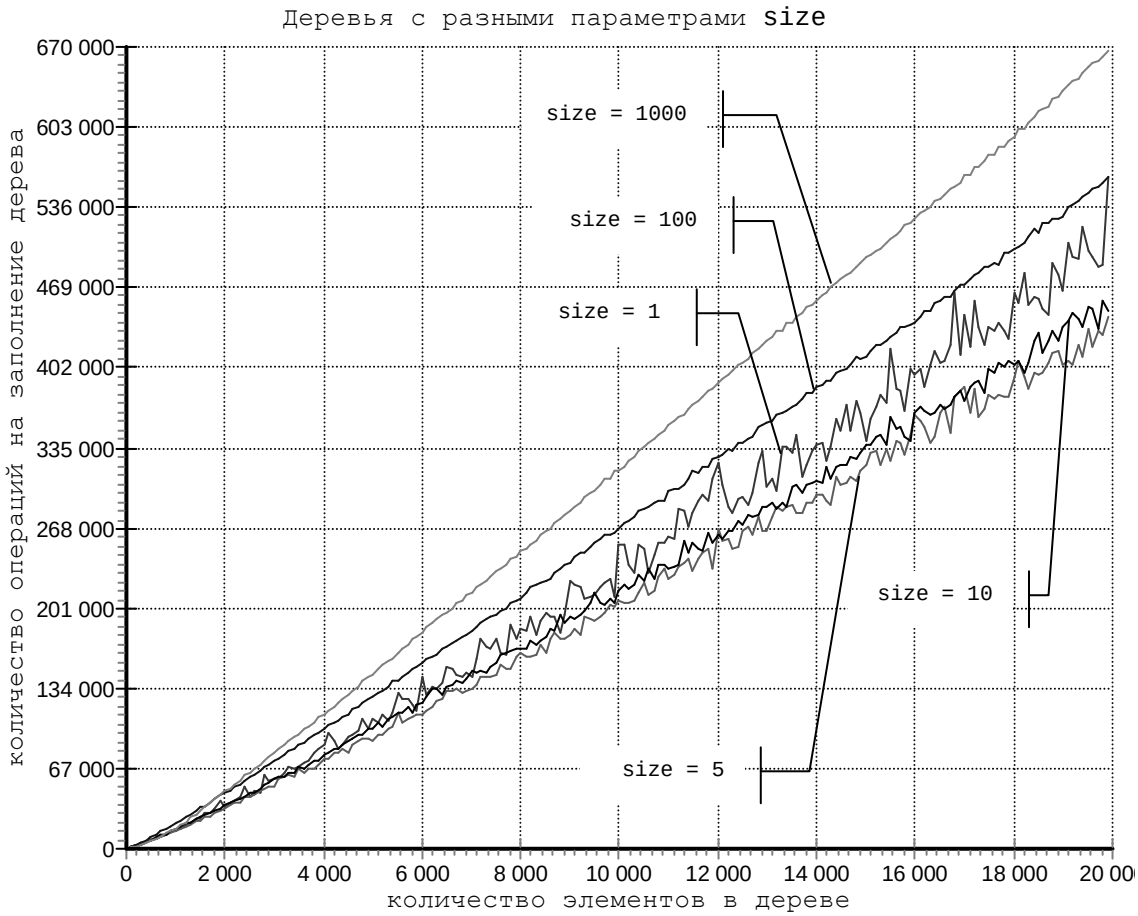

Количество элементов в вершине = 5
количество операций за вызов add()
48
45
42
39
36
33
30
27
24
21
18
15
0
100 000
200 000
300 000
400 000
500 000
600 000
700 000
800 000
900 000
1 000 0
n-ый вызов add()

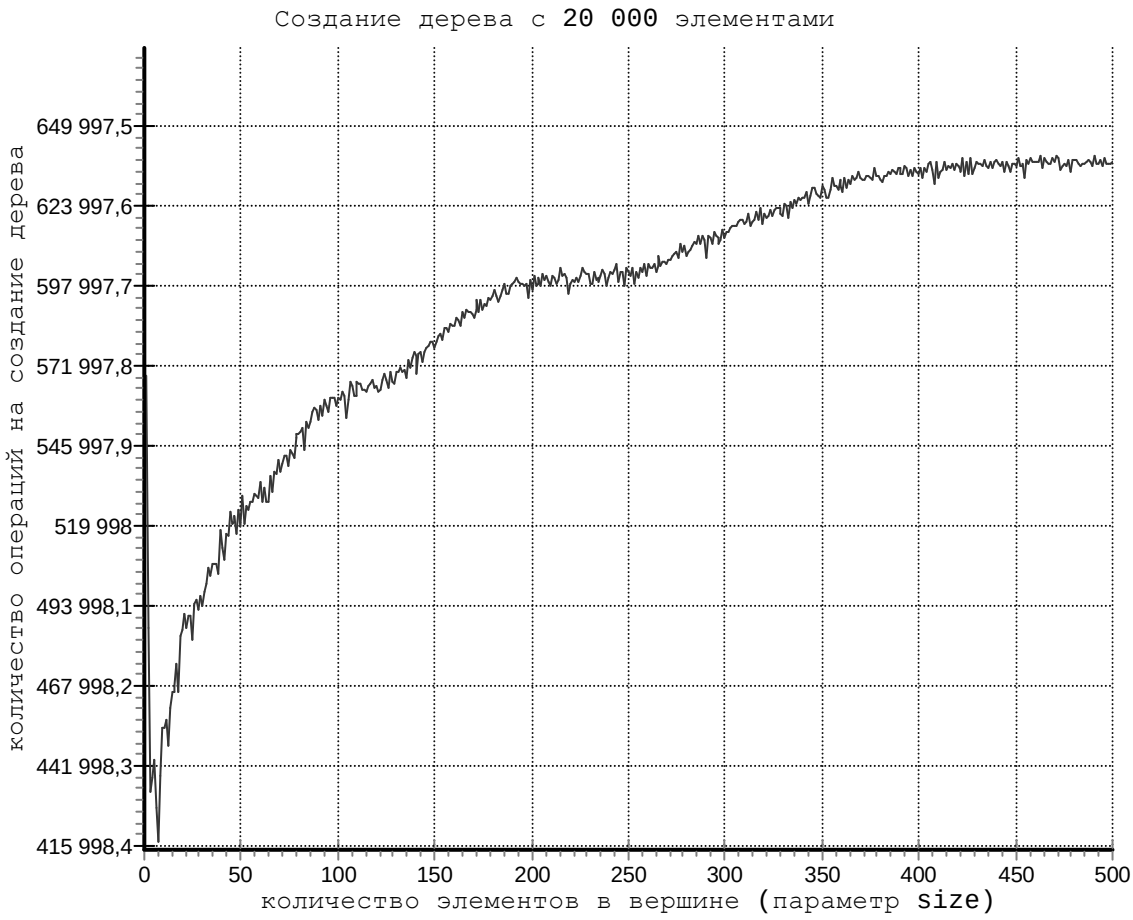

Количество элементов в вершине = 10
количество операций за вызов add()
39
36
33
30
27
24
21
18
15
0
100 000
200 000
300 000
400 000
500 000
600 000
700 000
800 000
900 000
1 000 0
n-ый вызов add()

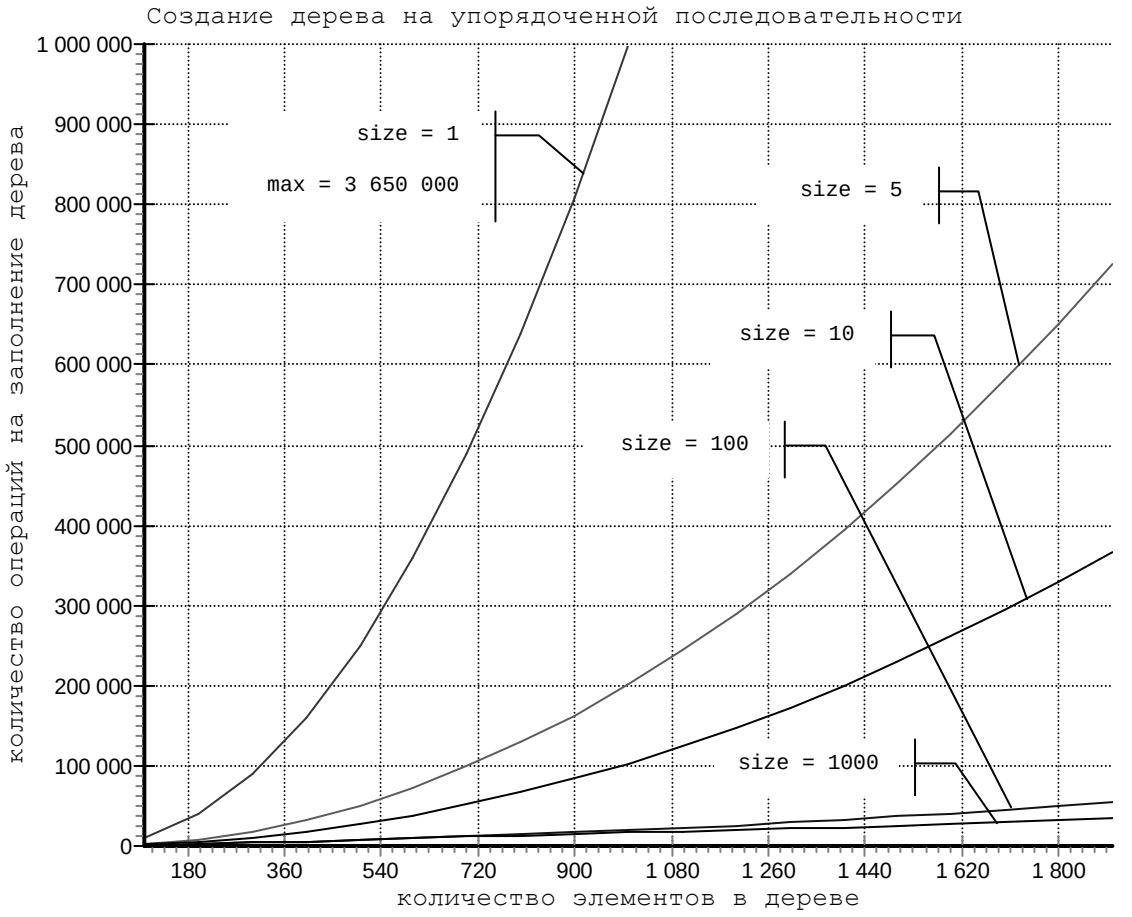

Количество элементов в вершине = 100
количество операций за вызов add()
45
42
39
36
33
30
27
24
21
18
15
0
100 000
200 000
300 000
400 000
500 000
600 000
700 000
800 000
900 000
1 000 0
n-ый вызов add()


Количество элементов в вершине = 1000
количество операций за вызов add()
48
45
42
39
36
33
30
27
24
21
18
15
0
100 000
200 000
300 000
400 000
500 000
600 000
700 000
800 000
900 000
1 000 0
n-ый вызов add()

Добавление n элементов




Создание дерева с 20 000 элементами
количество операций на создание дерева
649 997,5
623 997,6
597 997,7
571 997,8
545 997,9
519 998
493 998,1
467 998,2
441 998,3
415 998,4
0
50
100
150
200
250
300
350
400
450
500
количество элементов в вершине (параметр size)


Создание дерева на упорядоченной последовательности
количество операций на заполнение дерева
1 000 000
900 000
800 000
700 000
600 000
500 000
400 000
300 000
200 000
100 000
0
size = 1
max = 3 650 000
size = 5
size = 10
size = 100
size = 1000
180
360
540
720
900
1 080
1 260
1 440
1 620
1 800
количество элементов в дереве

Выводы:

Проведенные эксперименты подтвердили народную мудрость: «выше головы не прыгнешь». Выгода от использования двухуровневой структуры данных на операции добавления составила в лучшем случае 25%. Оказалось, что размерность массива нижнего уровня желательно устанавливать от 5 до 10, при выходе за границы этого интервала мы, в лучшем случае, получим либо малую выгоду, либо никакой, в худшем случае – значительное увеличение трудоемкости.

С другой стороны, на упорядоченной последовательности двухуровневая структура данных дает колоссальное преимущество перед простым деревом, сказываются особенности массива. Двухуровневой структура данных вырождается в обыкновенный статический массив, при числе элементов в дереве <= размерности массива нижнего уровня. Отсюда вытекает следующее свойство: чем больше размер массива нижнего уровня, тем ниже трудоемкость операции получения по логическому номеру. Например, при размере массива нижнего уровня size = 1 000 и при заполнения дерева 2 000 элементов, на получение элемента по логическому номеру уйдет 1-2 операции. Но тут возникает резонный вопрос: зачем организовывать дерево, если проще использовать динамический массив?

Итак, возможности данной структуры данных сильно ограничены. Оптимальная размерность нижнего уровня: 5-10. При выходе за этот интервал вправо, структура будет вырождаться в массив, при выходе влево – в обычное дерево. Однако, имеет смысл использовать такую организацию дерева, при размере массива нижнего уровня 5-10 мы получаем уменьшение трудоемкости как добавления нового элемента, так и позиционирования по логическому номеру.

Приложение.

Код рабочей функции:

```
template <class TYPE> void branch<TYPE>::addbr(TYPE *value,int size){
	count++;
	if (filled==size){
		if(*value<*record[0]){
			if(left==NULL)
				left=new branch(size);
			left->addbr(value, size);
		}
		else if(*value>=*record[size-1]){
			if(right==NULL)
				right=new branch(size);
			right->addbr(value, size);
		}
		else{
			TYPE *temp;
			for(int i=0;i<filled;i++)
				if(*value<*record[i]){
					temp=record[filled-1];
					for(int j=filled-1;j>i;j--)
						record[j]=record[j-1];
					record[i]=value;
					value=temp;
					break;
				}
			if(right==NULL)
				right=new branch(size);
			right->addbr(value, size);
		}
	}
	else{
		int i;
		for(i=0;i<filled;i++)
			if(*value<*record[i]){
				for(int j=filled;j>i;j--)
					record[j]=record[j-1];
				break;
			}
		record[i]=value;
		filled++;
	}
	return;
}
```

Код рабочей функции для снятия статистики (stat – количество значащих операций):

```
template <class TYPE> long branch<TYPE>::addbr(TYPE *value,int size){
	long stat=0;
	count++;
	if (filled==size){
		if(*value<*record[0]){
			stat++;
			if(left==NULL)
				left=new branch(size);
			stat+=left->addbr(value, size);
		}
		else if(*value>=*record[size-1]){
			stat+=2;
			if(right==NULL)
				right=new branch(size);
			stat+=right->addbr(value, size);
		}
		else{
			stat+=2;
			TYPE *temp;
			for(int i=0;i<filled;i++){
				stat++;
				if(*value<*record[i]){
					temp=record[filled-1];
					for(int j=filled-1;j>i;j--)
						record[j]=record[j-1];
					record[i]=value;
					value=temp;
					break;
				}
			}
			if(right==NULL)
				right=new branch(size);
			stat+=right->addbr(value, size);
		}
	}
	else{
		int i;
		for(i=0;i<filled;i++){
			stat++;
			if(*value<*record[i]){
				for(int j=filled;j>i;j--)
					record[j]=record[j-1];
				break;
			}
		}
		record[i]=value;
		filled++;
	}
	return stat;
}
```

Код модернизированной рабочей функции для снятия статистики:

```
template <class TYPE> long branch<TYPE>::addbr(TYPE *value,int size){
      long stat=0;
      count++;
      if (filled==size){
            if(*value<*record[0]){
                  stat++;
                  if(left==NULL) left=new branch(size);
                  stat+=left->addbr(value, size);
            }
            else if(*value>=*record[size-1]){
                  stat+=2;
                  if(right==NULL) right=new branch(size);
                  stat+=right->addbr(value, size);
            }
            else{
                  stat+=2;
                  TYPE *temp;
                  int a,b,m,k=0;
                  for(a=0,b=filled-1; a < b;){
                        m = (a + b)/2;
                        stat++;
                        if (*record[m] == *value){
                              k=m; break;}
                        stat++;
                        if (*record[m] > *value) b = m-1;
                        else a = m+1;
                  }
                  if (k==0){
                        k=a;
                        stat++;
                        if (*value > *record[a]) k++;
                  }
                  temp=record[filled-1];
                  for(int j=filled-1;j>k;j--)
                        record[j]=record[j-1];
                  record[k]=value;
                  value=temp;
                  if(right==NULL) right=new branch(size);
                  stat+=right->addbr(value, size);
            }
      }
      else{
            int a,b,m,k=0;
            for(a=0,b=filled-1; a < b;){
                  m = (a + b)/2;
                  stat++;
                  if (*record[m] == *value){k=m; break;}
                  stat++;
                  if (*record[m] > *value) b = m-1;
                  else a = m+1;
            }
            if (k==0){k=a;
                  if (filled!=0){ stat++; if (*value > *record[a]) k++;}
            }
            for(int j=filled;j>k;j--)
                  record[j]=record[j-1];
            record[k]=value;
            filled++;
      }
      return stat;
}
```

Алгоритмы сбора статистики:

Для эксперимента «Добавление n-ого элемента»:

```
void main(void){
	tree<int> T(1000);
	long i,j,l;

	cout<<"wait...\n";
	for(i=0;i<1000;i++) rand();

	fstream fileout;
	fileout.open("D:/stat1_1000.txt",ios::out);
	fileout<<"elem\toper\n";
	for(i=0,j=0;i<1000000;i++,j++){
		l=T.add(rand());
		cout<<i<<"\n";
		if(j==1000){
			j=0;
			fileout<<i<<'\t'<<l<<'\n';
		}
	}
	fileout.close();
	cout<<"ready!";
	getch();
}
```

Для эксперимента «Добавление n элементов»:

```
long stat_add(int SZ, int NUM){
	tree<int> TT(SZ);
	long l=0;
	for(int j=0;j<NUM;j++)
		l+=TT.add(rand());
	return l;
}

void main(void){
	long i,j,l,l1,l5,l10,l100,l1000;
	cout<<"wait...\n";
	for(i=0;i<1000;i++) rand();

	fstream fileout;
	fileout.open("D:/stat2.txt",ios::out);
	fileout<<"elem\tl1\tl5\tl10\tl100\tl1000\n";
	for(i=100;i<20000;i+=100){
		l1= stat_add(1,i);
		l5= stat_add(5,i);
		l10= stat_add(10,i);
		l100= stat_add(100,i);
		l1000= stat_add(1000,i);
		fileout<<i<<'\t'<<l1<<'\t'<<l5<<'\t';
		fileout<<l10<<'\t'<<l100<<'\t'<<l1000<<'\n';
		cout<<i<<'\n';
	}
	fileout.close();
	cout<<"ready!";

	getch();
}
```