# Московский авиационный институт (национальный исследовательский университет)

## Факультет информационных технологий и прикладной математики

## Кафедра вычислительной математики и программирования


**Лабораторная работа №3 по курсу «Дискретный анализ»**


Студент: А. Т. Бахарев
Преподаватель: А. А. Кухтичев
Группа: М8О-206Б

Дата:
Оценка:
Подпись:

**Москва, 2018**

# Лабораторная работа №3

**Задача:** Требуется провести исследование скорости выполнения и потребления оперативной памяти в реализации словаря из 2ой лабораторной работы. В случае выявления ошибок или явных недочётов, требуется их исправить.

# 1 Описание

Для выполнения данной лабораторной работы я решил реализовать класс бинарного дерева. Я не стал использовать AVL-дерево из прошлой лабораторной работы, потому что она не совсем хорошо написана – есть много недочетов при планировании и реализации класса. Все помарки я учел и решил закрепить результат, написав бинарное дерево. Для профилирования моей программы я выбрал 2 инструмента – gprof и valgrind.

**gprof** – Помогает отслеживать время работы каждой функции. Есть 2 способа наблюдения: таблица и граф вызовов. Таблица сообщает, какое количество времени в процентах работала та или иная функция, а так же показывает число её вызовов. Граф вызовов показывает, какая функция из какой вызывалась. Таким образом, мы можем отследить самые долгоработающие участки программы и решить, стоит ли их переписывать для ускорения работы нашего приложения.

**valgrind** – Отслеживание утечек памяти в программе. Часто бывают случаи, когда некоторый участок памяти выделен, но не освобождён. Если мы работаем над небольшим проектом, то утечка памяти не так уж и страшна, ведь операционная система сама почистит память при завершении работы программы. Но если мы пишем только какую-то часть приложения, то тут могут быть различные проблемы с непредвиденным расходом памяти, что негативно сказывается на быстродействии. Так же, при неправильной работе с указателями, возможна ситуация, когда мы обращаемся к невыделенному участку памяти для нашей программы. Скорее всего, программа успешно скомпилируется и даже будет работать на некоторых ЭВМ, но для других возможны критические ошибки. Данная утилита помогает сократить расходы памяти, выявить потери данных и нарушения прав доступа к участкам памяти.

# 2 Исходный код

Ниже приведен отлаженный и рабочий код бинарного дерева.

**btree.h**

```
class TNode
{
public:
    TNode();
    ~TNode();
    long GetVal();
    void SetVal(long val);
 private:
     TNode* Left;
     TNode* Right;
     long Value;

friend class TBtree;
};

class TBtree
{
public:
    TBtree();
    ~TBtree();
    bool Insert(long value);
    bool Remove(long value);
    TNode* Find(long value);
private:
    void ClearTree(TNode*& nd);
    bool Insert_(TNode*& nd, long val);
    bool Remove_(TNode*& nd, long val);
    TNode* Find_(TNode*& nd, long val);
    TNode* Root;
};
```

**btree.cpp**

```
#include <iostream>
#include <cstdlib>
#include "bin_tree.h"
TNode::TNode(){}
TNode::~TNode(){}
void TNode::SetVal(long val)
{
    Value = val;
}
long TNode::GetVal()
{
    return Value;
```

```
13 }
14 TBtree::TBtree()
15 {
16     Root = nullptr;
17 }
18 void TBtree::ClearTree(TNode*& nd)
19 {
20     if(nd == nullptr)
21         return;
22     if(nd->Left != nullptr)
23         ClearTree(nd->Left);
24     else if(nd->Right != nullptr)
25         ClearTree(nd->Right);
26     delete nd;
27     nd = nullptr;
28     return;
29 }
30
31 TBtree::~TBtree()
32 {
33     ClearTree(Root);
34 }
35 bool TBtree::Insert(long val)
36 {
37     if(Insert_(Root, val))
38         return true;
39     else return false;
40 }
41 TNode* TBtree::Find(long val)
42 {
43     return Find_(Root, val);
44 }
45 TNode* TBtree::Find_(TNode*& nd, long val)
46 {
47     if(nd == nullptr)
48         return nullptr;
49     if(nd->Value > val)
50         Find_(nd->Left, val);
51     else if(nd->Value < val)
52         Find_(nd->Right, val);
53     else return nd;
54 }
55 bool TBtree::Insert_(TNode*& nd, long val)
56 {
57     if(nd == nullptr)
58     {
59         nd = new TNode;
60         if(nd == nullptr)
61             return false; // Not enough memory
```

```
62          nd->Value = val;
63          return true;
64      }
65      else if(nd->Value > val)
66          return Insert_(nd->Left, val);
67      else if(nd->Value < val)
68          return Insert_(nd->Right, val);
69      return false; // item already added
70  }
```

### main.cpp

```
1   #include <iostream>
2   #include <cstdlib>
3   #include <random>
4   #include "bin_tree.h"
5   int main()
6   {
7       TBtree tree;
8       for(int i = 0; i < 2000; ++i)
9       {
10          std::cin>>val;
11          tree.Insert(val);
12      }
13      for(int i = 0; i < 2000; ++i)
14      {
15          std::cin>>val;
16          tree.Find(val);
17      }
18      return 0;
19  }
```

Исследуем данную программу на быстродействие и утечки памяти при помощи утилит valgrind и gprof.

**Valgrind**

```
alex$make
g++ -std=c++11 -pg -pedantic -Wall main.cpp bin_tree.cpp
alex$valgrind --leak-check=full ./a.out <01.t
==1832== HEAP SUMMARY:
==1832==     in use at exit: 72,704 bytes in 1 blocks
==1832==   total heap usage: 1,915 allocs,1,914 frees,122,712 bytes allocated
==1832==
==1832== LEAK SUMMARY:
==1832==    definitely lost: 0 bytes in 0 blocks
==1832==    indirectly lost: 0 bytes in 0 blocks
==1832==      possibly lost: 0 bytes in 0 blocks
==1832==    still reachable: 72,704 bytes in 1 blocks
==1832==         suppressed: 0 bytes in 0 blocks
==1832== Reachable blocks (those to which a pointer was found) are not shown.
==1832== To see them,rerun with: --leak-check=full --show-leak-kinds=all
==1832==
==1832== For counts of detected and suppressed errors,rerun with: -v
==1832== Use --track-origins=yes to see where uninitialised values come from
==1832== ERROR SUMMARY: 3826 errors from 22 contexts (suppressed: 0 from 0)
```

Анализ паямяти показывает, что есть много ошибок при обращении к адресам памяти. Valgrind трактует это следующим образом : "Conditional jump or move depends on uninitialised value(s)". Это значит, что мы берем значение по указателю, но инициализация значения не была произведена. Там лежит какой-то мусор. Программа не завершается аварийно, но на некоторых ЭВМ может произойти критическая ошибка. С помощью анализатора, я смог найти место, где указатели не инициализируются. Это конструктор класса *TNode*. Я не обнулил указатели на левое и правое поддерево. Хотел сделать это при создании узла, в функции добавления элемента, но забыл. Поэтому получил много ошибок. Конструктор должен выглядеть так:

```
1 TNode::TNode()
2 {
3     Left = Right = nullptr;
4 }
```

Так же, вывод утилиты Valgrind говорит нам о том, что мы забыли освободить один участок памяти. Выделено 1915 фрагментов памяти, а освобождено только 1914. Но это, к счастью, ошибка в компиляторе gcc. Называется она так: "Emergency buffer

for exception allocation too small". На самом деле, все освобождено и утечек памяти нет. Проверим программу на быстродействие.

**gprof**

```
alex$make
g++ -std=c++11 -pg -pedantic -Wall main.cpp bin_tree.cpp
alex$./a.out <01.t
alex$gprof -D -b -a ./a.out gmon.out
Flat profile:

Each sample counts as 0.01 seconds.
%   cumulative   self              self     total
time   seconds   seconds    calls   s/call   s/call  name
47.27      1.61     1.61 10000000     0.00     0.00  TBtree::Find_(TNode*&,long)
40.77      2.99     1.39 10000001     0.00     0.00  TBtree::Insert_(TNode*&,long)
6.50       3.21     0.22        1     0.22     3.42  main
3.25       3.32     0.11 10000000     0.00     0.00  TBtree::Find(long)
2.36       3.41     0.08 10000000     0.00     0.00  TBtree::Insert(long)
0.30       3.42     0.01        1     0.01     0.01  TBtree::~TBtree()
0.00       3.42     0.00    20001     0.00     0.00  TNode::TNode()
0.00       3.42     0.00        9     0.00     0.00  TNode::~TNode()
0.00       3.42     0.00        1     0.00     0.00  TBtree::ClearTree(TNode*&)
0.00       3.42     0.00        1     0.00     0.00  TBtree::TBtree()
alex$
```

Из таблицы видно, что основное время программы выполняются только приватные функции вставки и поиска, так как они являются внутренними для функций *Insert* и *Find*. В принципе, я считаю, что тут оптимизировать ничего не стоит, ведь большинство времени работают только нужные нам функции. Конечно, всегда можно найти варианты для отимизации, но сейчас не стоит. Но что если мы захотим повысить безопасность нашего класса, добавив функции типа Set-Get для значения в узле дерева? Их листинг приведен ниже:

```
1 void TNode::SetVal(long val)
2 {
3     Value = val;
4 }
5 long TNode::GetVal()
6 {
7     return Value;
8 }
```

Такие функции должны повысить безопасность при работе с экземплярами классов, ведь мы защищены от непроизвольного доступа к приватным полям класса. Но такая защита сильно сказывается на быстродействии программы:

```
alex$make
g++ -std=c++11 -pg -pedantic -Wall main.cpp bin_tree.cpp
alex$./a.out <01.t
alex$gprof -D -b -a ./a.out gmon.out
Flat profile:
Flat profile:

Each sample counts as 0.01 seconds.
%   cumulative   self              self     total
time   seconds   seconds    calls  ms/call  ms/call  name
32.53      1.64     1.64 10000000     0.00     0.00  TBtree::Find_(TNode*&,long)
24.47      2.87     1.23 10000000     0.00     0.00  TBtree::Insert_(TNode*&,long)
18.30      3.79     0.92 498809892     0.00     0.00  TNode::GetVal()
5.97       4.52     0.30                             main
3.18       4.68     0.16    10001     0.02     0.02  TNode::SetVal(long)
2.88       4.82     0.15 10000000     0.00     0.00  TBtree::Find(long)
1.79       4.91     0.09 10000000     0.00     0.00  TBtree::Insert(long)
0.00       5.05     0.00    10001     0.00     0.00  TNode::TNode()
0.00       5.05     0.00        9     0.00     0.00  TNode::~TNode()
0.00       5.05     0.00        1     0.00     0.00  TBtree::ClearTree(TNode*&)
0.00       5.05     0.00        1     0.00     0.00  TBtree::TBtree()
0.00       5.05     0.00        1     0.00     0.00  TBtree::~TBtree()
alex$
```

Видно, что функции Get и Set работают вместе почти 20% от времени работы программы. Сравним теперь время работы программ:

```
real 0m25.670s
user 0m11.152s
sys  0m14.512s

real 0m33.116s
user 0m18.648s
sys  0m14.452s
```

Первый замер производился для "небезопасной"программы, а второй, соответсвенно, для безопасной. Видно, что время работы отличаются довольно прилично. При боль-

ших обьемах данных, эта разница будет весьма заметной. Но единственно правильного и быстрого решения тут нет. Разработчику всегда приходится балансировать между быстродействием и безопасностью.

# 3 Выводы

Выполнив первую лабораторную работу по курсу «Дискретный анализ», я познакомился с утилитами и программами, которые помогают устранять ошибки и проблемы в программах и приложениях. Есть много различных средств, которые облегчат жизнь любому программисту, который следит за качеством своих программ. Какие именно выбрать – это дело вкуса и конкретно решаемой задачи. Самое главное — это не забывать их использовать.

# Список литературы


[1] *StackOverflow - Valgrind error*
URL: `http://https://stackoverflow.com/questions/31775034/valgrind-error-in-use-at-exit-72-704-bytes-c-initialization-list-weirdness-w` (дата обращения: 21.09.2018).

[2] *OpenNET - Профилятор gprof*
URL: `http://www.opennet.ru/docs/RUS/gprof` (дата обращения: 21.09.2018).

[3] *IBM Community - Detect memory leaks with Memcheck tool provided by Valgrind*
URL: `https://www.ibm.com/developerworks/community/blogs/entry/detect-memory-leaks-with-memcheck-tool-provided-by-valgrind-part-i8` (дата обращения: 21.09.2018).