Санкт-Петербургский политехнический университет
Высшая школа прикладной математики и вычислительной физики, ФизМех

Направление подготовки
«01.03.02 Прикладная математика и информатика»

Отчет по летней практике
**тема "Фибоначева куча"**
Дисциплина "Технологическая (проектно-технологическая) практика"

Выполнил студент гр. 5030102/00001 Золин И.М.
Преподаватель: Беляев.С.Ю.

Санкт-Петербург

**2022**

## Содержание



# 1 Постановка задачи

- Необходимо реализовать Фибоначеву кучу.
- А также её главные функции:
  - Забоать минимум
  - Добавить новый элемент
  - Сделать слияние
- Написать систему тестов, проверяющую раотоспособность программы.

# 2 Описание алгоритма

```
node_t - узел Фибоначевой кучи, содержащий:
key - ключ,
degree - кол-во потомков,
parent - ук-ль на родительский узел,
child - ук-ль на потомка,
left - ук-ль на левого соседа,
right - ук-ль на правого соседа ,
marked - логическая переменная, отвечающая за то, были ли потери узлом потомков,
начиная с момента, когда x стал дочерним узлом какого-то другого узла.
visited - логическая переменная, отвечающая за посещение этого узла
(нужно для функции Find)
```

```
typedef struct node_t{
  int key;
  int degree;// vertex degree
  struct node_t* parent;
  struct node_t* child;
  struct node_t* left;
  struct node_t* right;
  bool marked;// was the child of this vertex deleted in the process of changing the key
  bool visited;
}node_t;
```

```
fibonacci_heap_t - структура Фибоначевой кучи, содержащая:
heap - ук-ль на минимум, яв-ся корнем одного из дерева,
maxNodes - кол-во узлов
```

```
typedef struct fibonacci_heap_t {
  struct node_t* heap;
  int maxNodes;
}fibonacci_heap_t;
```

Далее для Фибоначевой кучи опишем основные функции:

**Главные функции:**

Инициализация фибоначевой кучи.

```
fibonacci_heap_t* Init();
```

Отрисовка Фибоначевой кучи в консоли.

```
void Display(fibonacci_heap_t* f_h);
```

Поиск минимального узла.

```
node_t* findMin(fibonacci_heap_t* f_h);//O(1)
```

Вставка узла.

```
void Insert(fibonacci_heap_t* f_h, int value);//O(1)
```

Извлечение минимального узла.

```
node_t* extractMin(fibonacci_heap_t* f_h);//O(logn)
```

Объеденение двух фибоначевых куч.

```
fibonacci_heap_t* Merge(fibonacci_heap_t* f_h1, fibonacci_heap_t* f_h2);//O(1)
```

Уменьшение ключа.

```
bool decreaseKey(fibonacci_heap_t* f_h, int key, int new_key);//O(1)
```

Удаление узла.

```
void Delete(fibonacci_heap_t* f_h, int value);//O(logn)
```

**Вспомогательные функции:**

Функция, создающая новый узел. Используется в fibonacci_heap_t* Init();

```
node_t* newNode(int value);
```

Функция, соеденяющая два узла.
Используется в void Consolidate(fibonacci_heap_t* f_h);

```
void fibHeapLink(fibonacci_heap_t* f_h, node_t* y, node_t* x);
```

Функция, уплотняющая список корней.
Используется в node_t* extractMin(fibonacci_heap_t* f_h);

```
void Consolidate(fibonacci_heap_t* f_h);
```

Функция, удалющая узел x из тек. позиции и доб-щая в корневой список.
Используется в void decreaseKey(fibonacci_heap_t* f_h, node_t* node, int new_key);

```
void Cut(fibonacci_heap_t* f_h, node_t* node_decreased, node_t* parent_node);
```

Функция, удалющая узлы с тек позиции, пока не дойдём до помеченного узла - каскадная вырезка.
Используется в void decreaseKey(fibonacci_heap_t* f_h, node_t* node, int new_key);

```
void cascadingCut(fibonacci_heap_t* f_h, node_t* parent_node);
```

Функция, находящая узел с заданным ключом.
Используется в void decreaseKey(fibonacci_heap_t* f_h, node_t* node, int new_key);

```
node_t* Find(node_t* heap, int value);
```

## 3 Текст программы

```
#pragma once
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <math.h>
#pragma warning(disable : 4996)
typedef struct node_t{
  int key;
  int degree;// vertex degree
  struct node_t* parent;
  struct node_t* child;
  struct node_t* left;
  struct node_t* right;
  bool marked;// was the child of this vertex deleted in the process of changing the key
  bool visited;
}node_t;

typedef struct fibonacci_heap_t {
  struct node_t* heap;
  int maxNodes;
}fibonacci_heap_t;

//Main functions
fibonacci_heap_t* Init();
void Display(fibonacci_heap_t* f_h);
node_t* findMin(fibonacci_heap_t* f_h);//O(1)
void Insert(fibonacci_heap_t* f_h, int value);//O(1)
node_t* extractMin(fibonacci_heap_t* f_h);//O(logn)
fibonacci_heap_t* Merge(fibonacci_heap_t* f_h1, fibonacci_heap_t* f_h2);//O(1)
bool decreaseKey(fibonacci_heap_t* f_h, int key, int new_key);//O(1)
void Delete(fibonacci_heap_t* f_h, int value);//O(logn)
//Support functions
node_t* newNode(int value);
void fibHeapLink(fibonacci_heap_t* f_h, node_t* y, node_t* x);
void Consolidate(fibonacci_heap_t* f_h);
void Cut(fibonacci_heap_t* f_h, node_t* node_decreased, node_t* parent_node);
void cascadingCut(fibonacci_heap_t* f_h, node_t* parent_node);
node_t* Find(node_t* heap, int value);
```

Листинг 1: heap.h

```c
#include "heap.h"
fibonacci_heap_t* Init()
{
  fibonacci_heap_t* f_h = (fibonacci_heap_t*)malloc(sizeof(fibonacci_heap_t));
  f_h->heap = NULL;
  f_h->maxNodes = 0;
  return f_h;
}

node_t* newNode(int value)
{
  node_t* node = (node_t*)malloc(sizeof(node_t));
  node->key = value;
  node->degree = 0;
  node->marked = false;
  node->visited = false;
  node->left = node;
  node->right = node;
  node->parent = NULL;
  node->child = NULL;
}

void Insert(fibonacci_heap_t* f_h, int value)
{
  node_t* new_node = newNode(value);
  if (f_h->heap == NULL)
  {
    f_h->heap = new_node;
  }
  else {
    f_h->heap->left->right = new_node;
    new_node->right = f_h->heap;
    new_node->left = f_h->heap->left;
    f_h->heap->left = new_node;
    if (new_node->key < f_h->heap->key)
      f_h->heap = new_node;
  }
  (f_h->maxNodes)++;
}

void Display(fibonacci_heap_t* f_h)
{
  node_t* node = f_h->heap;
  node_t* x;
  x = node;
  if (node == NULL)
    printf("The heap is empty\n");
  else {
    printf("The nodes of heap: \n");

    do {

      printf("(%d)", x->key);
      x = x->right;
      if (x != node)
        printf("-->");
    } while (x != node && x->right != NULL);
    printf("\n The Fibonacci heap has %d nodes\n", f_h->maxNodes);
  }
}

node_t* findMin(fibonacci_heap_t* f_h)
{
  if (f_h == NULL)
  {
    printf("\nThe Fibonach heap has not been created yet\n");
    return NULL;
  }
  else
  {
    return f_h->heap;
  }
}
```

```
74
75 fibonacci_heap_t* Merge(fibonacci_heap_t* f_h1, fibonacci_heap_t* f_h2)
76 {
77   fibonacci_heap_t* f_h3 = Init();
78   f_h3->heap = f_h1 ->heap;
79   node_t *tmp1, *tmp2;
80   tmp1 = f_h3->heap->right;
81   tmp2 = f_h2->heap->left;
82   f_h3->heap->right->left = f_h2->heap->left;
83   f_h3->heap->right = f_h2->heap;
84   f_h2->heap->left = f_h3->heap;
85   tmp2->right = tmp1;
86   if ((f_h1->heap == NULL) || ((f_h2->heap != NULL) && (f_h2->heap->key < f_h1->heap->key)))
87     f_h3->heap = f_h2->heap;
88   f_h3->maxNodes = f_h1->maxNodes + f_h2->maxNodes;
89   return f_h3;
90 }
91
92 int calcDegree(int n)
93 {
94   int count = 0;
95   while (n > 0)
96   {
97     n = n / 2;
98     count++;
99   }
100  return count;
101 }
102
103 void fibHeapLink(fibonacci_heap_t* f_h, node_t* y, node_t* x)
104 {
105   y->right->left = y->left;
106   y->left->right = y->right;
107   if (x->right == x)
108     f_h->heap = x;
109   y->left = y;
110   y->right = y;
111   y->parent = x;
112   if (x->child == NULL)
113     x->child = y;
114   y->right = x->child;
115   y->left = x->child->left;
116   x->child->left->right = y;
117   x->child->left = y;
118   if ((y->key) < (x->child->key))
119     x->child = y;
120   (x->degree)++;
121 }
122
123 void Consolidate(fibonacci_heap_t* f_h)
124 {
125   int degree, i, d;
126   degree = calcDegree(f_h->maxNodes);
127
128   node_t** A = (node_t**)malloc(sizeof(node_t*)*degree);
129   node_t* x, * y, * z;
130   for (int i = 0; i <= degree; i++)
131   {
132     A[i] = NULL;
133   }
134   x = f_h->heap;
135   do {
136     d = x->degree;
137     while (A[d] != NULL)
138     {
139       y = A[d];
140       if (x->key > y->key)
141       {
142         node_t* ex_help;//exchange help
143         ex_help = x;
144         x = y;
145         y = ex_help;
146       }
```

```
      if (y == f_h->heap)
        f_h->heap = x;
      fibHeapLink(f_h, y, x);
      if (y->right == x)
        f_h->heap = x;
      A[d] = NULL;
      d++;
    }
    A[d] = x;
    x = x->right;
  } while (x != f_h->heap);
  f_h->heap = NULL;
  for (int i = 0; i < degree; i++)
  {
    if (A[i] != NULL)
    {
      A[i]->left = A[i];
      A[i]->right = A[i];
      if (f_h->heap == NULL)
        f_h->heap = A[i];
      else {
        f_h->heap->left->right = A[i];
        A[i]->right = f_h->heap;
        A[i]->left = f_h->heap ->left;
        f_h->heap->left = A[i];
        if (A[i]->key < f_h->heap->key)
          f_h->heap = A[i];
      }
      if (f_h->heap == NULL)
        f_h->heap = A[i];
      else if (A[i]->key < f_h->heap->key)
        f_h->heap = A[i];
    }

  }
}

node_t* extractMin(fibonacci_heap_t* f_h)
{
  if (f_h->heap == NULL)
    printf("\n The heap is empty\n");
  else {
    node_t* tmp = f_h->heap;
    node_t* pntr;
    pntr = tmp;
    node_t* x = NULL;
    if (tmp->child != NULL)
    {
      x = tmp->child;
      do {
        pntr = x->right;
        (f_h->heap->left)->right = x;
        x->right = f_h->heap;
        x->left = f_h->heap->left;
        f_h->heap->left = x;
        if (x->key < f_h->heap->key)
          f_h->heap = x;
        x->parent = NULL;
        x = pntr;
      } while (pntr != tmp->child);
    }

    (tmp->left)->right = tmp->right;
    (tmp->right)->left = tmp->left;
    f_h->heap = tmp->right;
    if (tmp = tmp->right && tmp->child == NULL)
      f_h->heap = NULL;
    else {
      f_h->heap = tmp->right;
      Consolidate(f_h);
    }
    (f_h->maxNodes)--;
    return tmp;
```

```
  }
  return f_h->heap;
}
void Cut(fibonacci_heap_t* f_h, node_t* node_decreased, node_t* parent_node)
{
  node_t* tmp_parent_check;
  if (node_decreased == node_decreased->right)
    parent_node->child = NULL;
  node_decreased->left->right = node_decreased->right;
  node_decreased->right->left = node_decreased->left;
  if (node_decreased == parent_node->child)
    parent_node->child = node_decreased->right;
  (parent_node->degree)--;
  node_decreased->left = node_decreased;
  node_decreased->right = node_decreased;
  f_h->heap->left = node_decreased;
  node_decreased->parent = NULL;
  node_decreased->marked = false;
}
void cascadingCut(fibonacci_heap_t* f_h, node_t* parent_node)
{
  node_t* aux;
  aux = parent_node->parent;
  if (aux != NULL)
  {
    if (parent_node->marked == false)
      parent_node->marked = true;
    else {
      Cut(f_h, parent_node, aux);
      cascadingCut(f_h, aux);
    }
  }
}

node_t* Find(node_t* heap, int value)
{
  node_t* x = heap;
  node_t* p = NULL;
  x->visited = true;
  if (x->key == value)
  {
    p = x;
    x->visited = false;
    return p;
  }
  if (p == NULL)
  {
    if (x->child != NULL)
      p = Find(x->child, value);
    if ((x->right)->visited != true)
      p = Find(x->right, value);
  }
  x->visited = false;
  return p;
}

bool decreaseKey(fibonacci_heap_t* f_h, int key, int new_key)
{
  node_t* y;
  if (f_h->heap == NULL)
  {
    printf("The Heap is Empty\n");
    return 0;
  }
  node_t* ptr = Find(f_h->heap, key);
  if (ptr == NULL)
  {
    printf("Node not found in the Heap\n");
    return 1;
  }

  if (ptr->key < new_key)
  {
```

```
293     printf("Entered key greater than current key\n");
294     return 0;
295   }
296   ptr->key = new_key;
297   y = ptr->parent;
298   if (y != NULL && ptr->key < y->key)
299   {
300     Cut(f_h, ptr, y);
301     cascadingCut(f_h, y);
302   }
303
304   if (ptr->key < f_h->heap->key)
305     f_h->heap = ptr;
306
307   return 0;
308 }
309
310 void Delete(fibonacci_heap_t* f_h, int value)
311 {
312   node_t* m = NULL;
313   bool t = decreaseKey(f_h, value, -5000);
314   if (!t)
315     m = extractMin(f_h);
316   if (m != NULL)
317     printf("Node was deleted\n");
318   else
319     printf("Node wasn't deleted\n");
320   return 0;
321 }
```

Листинг 2: heap.c

# 4 Описание тестирования программы

## 4.1 Описание тестов

Проверка инициализация кучи : проверка функции void Init()

```
1 TEST(TestInit, CreateHeap)
```

Проверка создания новго узла : проверка функции node_t* newNode(int key)

```
1 TEST(TestNewNode, CreateNewNode)
```

Проверка добавления узла : проверка функции void Insert(fibonacci_heap_t* f_h)

```
1 TEST(TestInsert, AddNewNode)
```

Проверка добавления 2 узлов : проверка функции void Insert(fibonacci_heap_t* f_h)

```
1 TEST(TestInsert, Add2Nodes)
```

Проверка поиска минимального узла(проверка на ошибку, если нет узлов):
проверка функции node_t* findMin(fibonacci_heap_t* f_h)

```
1 TEST(TestFindMin, TestError)
```

Проверка поиска минимального узла:
проверка функции node_t* findMin(fibonacci_heap_t* f_h)

```
TEST(TestFindMin, FindMin)
```

Проверка слияния 2 куч : проверка функции
fibonacci_heap_t* Merge(fibonacci_heap_t* f_h1, fibonacci_heap_t* f_h2)

```
TEST(TestMerge, Merge2heaps)
```

Проверка извлечения минимального узла(проверка на ошибку, если нет узлов):
проверка функции node_t* extractMin(fibonacci_heap_t* f_h)

```
TEST(TestExtractMin, TestError)
```

Проверка извлечения минимального узла:
проверка функции node_t* extractMin(fibonacci_heap_t* f_h)

```
TEST(TestExtractMin, ExtractMin)
```

Проверка уменьшения ключа:
проверка функции bool decreaseKey(fibonacci_heap_t* f_h, int key, int new_key)

```
TEST(TestDecreaseKey, DecreaseKey)
```

Проверка уменьшения ключа(проверка на ошибку, если значение на замену
больше изначального ключа):
проверка функции bool decreaseKey(fibonacci_heap_t* f_h, int key, int new_key)

```
TEST(TestDecreaseKey, TestError1)
```

Проверка уменьшения ключа(проверка на ошибку, если нет такого ключа в куче):
проверка функции bool decreaseKey(fibonacci_heap_t* f_h, int key, int new_key)

```
TEST(TestDecreaseKey, TestError2)
```

Проверка удаления узла : проверка функции
void Delete(fibonacci_heap_t* f_h, int value)

```
TEST(TestDelete, DeleteElement)
```

Проверка удаления узла(проверка на ошибку, если нет узлов в
куче):
проверка функции void Delete(fibonacci_heap_t* f_h, int value)

```
TEST(TestDelete, TestError1)
```

Проверка удаления узла(проверка на ошибку, если нет такого ключа в
куче):
проверка функции void Delete(fibonacci_heap_t* f_h, int value)

```
TEST(TestDelete, TestError2)
```

Запускает все тесты

```
int main(int argc, char* argv[]) {
```

## 4.2 Код тестов

```
#pragma warning(disable : 4996)
#include "gtest/gtest.h"
extern "C" {
#include "heap.h"
}

TEST(TestInit, CreateHeap)
{
  fibonacci_heap_t* f_h = Init();
  EXPECT_TRUE(f_h != NULL);
  EXPECT_TRUE(f_h->heap == NULL);
  free(f_h);
}

TEST(TestNewNode, CreateNewNode)
{
  double value = 1;
  node_t* node  = newNode(value);
  EXPECT_TRUE(node != NULL);
  EXPECT_TRUE(node->key = 1);
  free(node);
}

TEST(TestInsert, AddNewNode)
{
  fibonacci_heap_t* f_h = (fibonacci_heap_t*)malloc(sizeof(fibonacci_heap_t));
  f_h->heap = NULL;
  f_h->maxNodes = 0;
  Insert(f_h, 1);
  EXPECT_TRUE(f_h != NULL);
  EXPECT_TRUE(f_h->heap != NULL);
  EXPECT_TRUE(f_h->heap->key = 1);
  free(f_h);
}

TEST(TestInsert, Add2Nodes)
{
  fibonacci_heap_t* f_h = (fibonacci_heap_t*)malloc(sizeof(fibonacci_heap_t));
  f_h->heap = NULL;
  f_h->maxNodes = 0;
  Insert(f_h, 1);
  Insert(f_h, 2);
  EXPECT_TRUE(f_h != NULL);
  EXPECT_TRUE(f_h->heap != NULL);
  EXPECT_TRUE(f_h->heap->key = 1);
  EXPECT_TRUE(f_h->heap->left->key = 2);
  EXPECT_TRUE(f_h->heap->right->key = 2);
  free(f_h);
}

TEST(TestFindMin, TestError)
{
  fibonacci_heap_t* f_h = (fibonacci_heap_t*)malloc(sizeof(fibonacci_heap_t));
  f_h->heap = NULL;
  f_h->maxNodes = 0;
  node_t* node = findMin(f_h);
  EXPECT_TRUE(node == NULL);
  free(f_h);
}

TEST(TestFindMin, FindMin)
{
  fibonacci_heap_t* f_h = (fibonacci_heap_t*)malloc(sizeof(fibonacci_heap_t));
  f_h->heap = NULL;
  f_h->maxNodes = 0;
  node_t* node;
  int value = 1;
  node_t* new_node = (node_t*)malloc(sizeof(node_t));
  new_node->key = value;
  new_node->degree = 0;
  new_node->marked = false;
```

```
72    new_node->visited = false;
73    new_node->left = new_node;
74    new_node->right = new_node;
75    new_node->parent = NULL;
76    new_node->child = NULL;
77    f_h->heap = new_node;
78    (f_h->maxNodes)++;
79    node = findMin(f_h);
80    EXPECT_TRUE(node != NULL);
81    free(f_h);
82  }
83
84  TEST(TestMerge, Merge2heaps)
85  {
86    fibonacci_heap_t* f_h1 = (fibonacci_heap_t*)malloc(sizeof(fibonacci_heap_t));
87    f_h1->heap = NULL;
88    f_h1->maxNodes = 0;
89    fibonacci_heap_t* f_h2 = (fibonacci_heap_t*)malloc(sizeof(fibonacci_heap_t));
90    f_h2->heap = NULL;
91    f_h2->maxNodes = 0;
92    fibonacci_heap_t* f_h3 = (fibonacci_heap_t*)malloc(sizeof(fibonacci_heap_t));
93    f_h3->heap = NULL;
94    f_h3->maxNodes = 0;
95    int value = 1;
96    node_t* new_node = (node_t*)malloc(sizeof(node_t));
97    new_node->key = value;
98    new_node->degree = 0;
99    new_node->marked = false;
100   new_node->visited = false;
101   new_node->left = new_node;
102   new_node->right = new_node;
103   new_node->parent = NULL;
104   new_node->child = NULL;
105   f_h1->heap = new_node;
106   (f_h1->maxNodes)++;
107   value = 2;
108   node_t* new_node2 = (node_t*)malloc(sizeof(node_t));
109   new_node2->key = value;
110   new_node2->degree = 0;
111   new_node2->marked = false;
112   new_node2->visited = false;
113   new_node2->left = new_node2;
114   new_node2->right = new_node2;
115   new_node2->parent = NULL;
116   new_node2->child = NULL;
117   f_h2->heap = new_node2;
118   (f_h2->maxNodes)++;
119   f_h3 = Merge(f_h1, f_h2);
120   EXPECT_TRUE(f_h3 != NULL);
121   EXPECT_TRUE(f_h3->heap->key = 1);
122   EXPECT_TRUE(f_h3->heap->right->key = 2);
123   EXPECT_TRUE(f_h3->maxNodes = 2);
124   free(f_h1);
125   free(f_h2);
126   free(f_h3);
127 }
128
129 TEST(TestExtractMin, TestError)
130 {
131   fibonacci_heap_t* f_h = (fibonacci_heap_t*)malloc(sizeof(fibonacci_heap_t));
132   f_h->heap = NULL;
133   f_h->maxNodes = 0;
134   node_t* min = extractMin(f_h);
135   EXPECT_TRUE(min == NULL);
136   free(f_h);
137 }
138
139 TEST(TestExtractMin, ExtractMin)
140 {
141   fibonacci_heap_t* f_h = (fibonacci_heap_t*)malloc(sizeof(fibonacci_heap_t));
142   f_h->heap = NULL;
143   f_h->maxNodes = 0;
144   int value = 1;
```

```
  node_t* new_node = (node_t*)malloc(sizeof(node_t));
  new_node->key = value;
  new_node->degree = 0;
  new_node->marked = false;
  new_node->visited = false;
  new_node->left = new_node;
  new_node->right = new_node;
  new_node->parent = NULL;
  new_node->child = NULL;
  f_h->heap = new_node;
  (f_h->maxNodes)++;
  value = 2;
  node_t* new_node2 = (node_t*)malloc(sizeof(node_t));
  new_node2->key = value;
  new_node2->degree = 0;
  new_node2->marked = false;
  new_node2->visited = false;
  new_node2->left = new_node2;
  new_node2->right = new_node2;
  new_node2->parent = NULL;
  new_node2->child = NULL;
  f_h->heap->left->right = new_node2;
  new_node2->right = f_h->heap;
  new_node2->left = f_h->heap->left;
  f_h->heap->left = new_node2;
  (f_h->maxNodes)++;
  node_t* min = extractMin(f_h);
  EXPECT_TRUE(min != NULL);
  EXPECT_TRUE(min = new_node);
  EXPECT_TRUE(min->key = 1);
  free(f_h);
}

TEST(TestDecreaseKey, DecreaseKey)
{
  fibonacci_heap_t* f_h = (fibonacci_heap_t*)malloc(sizeof(fibonacci_heap_t));
  f_h->heap = NULL;
  f_h->maxNodes = 0;
  int value = 3;
  node_t* node_decreased = (node_t*)malloc(sizeof(node_t));
  node_decreased->key = value;
  node_decreased->degree = 0;
  node_decreased->marked = false;
  node_decreased->visited = false;
  node_decreased->left = node_decreased;
  node_decreased->right = node_decreased;
  node_decreased->parent = NULL;
  node_decreased->child = NULL;
  f_h->heap = node_decreased;
  (f_h->maxNodes)++;
  int new_key = 1;
  decreaseKey(f_h, value, new_key);
  EXPECT_TRUE(f_h->heap->key  = 1);
  free(f_h);
}

TEST(TestDecreaseKey, TestError1)
{
  fibonacci_heap_t* f_h = (fibonacci_heap_t*)malloc(sizeof(fibonacci_heap_t));
  f_h->heap = NULL;
  f_h->maxNodes = 0;
  int value = 3;
  int new_key = 10;
  node_t* node_decreased = (node_t*)malloc(sizeof(node_t));
  node_decreased->key = value;
  node_decreased->degree = 0;
  node_decreased->marked = false;
  node_decreased->visited = false;
  node_decreased->left = node_decreased;
  node_decreased->right = node_decreased;
  node_decreased->parent = NULL;
  node_decreased->child = NULL;
  f_h->heap = node_decreased;
```

```
218   (f_h->maxNodes)++;
219   decreaseKey(f_h, value, new_key);
220   EXPECT_TRUE(f_h->heap->key = 3);
221   free(f_h);
222 }
223
224 TEST(TestDecreaseKey, TestError2)
225 {
226   fibonacci_heap_t* f_h = (fibonacci_heap_t*)malloc(sizeof(fibonacci_heap_t));
227   f_h->heap = NULL;
228   f_h->maxNodes = 0;
229   int value = 3;
230   int new_key = 10;
231   node_t* node_decreased = (node_t*)malloc(sizeof(node_t));
232   node_decreased->key = value;
233   decreaseKey(f_h, value, new_key);
234   EXPECT_TRUE(f_h->heap == NULL);
235   EXPECT_TRUE(node_decreased->key = 3);
236   free(f_h);
237 }
238
239 TEST(TestDelete, TestError1)
240 {
241   fibonacci_heap_t* f_h = (fibonacci_heap_t*)malloc(sizeof(fibonacci_heap_t));
242   f_h->heap = NULL;
243   f_h->maxNodes = 0;
244   int k1 = f_h->maxNodes;
245   int value = 1;
246   Delete(f_h, value);
247   int k2 = f_h->maxNodes;
248   ASSERT_EQ(k1, k2);
249   free(f_h);
250 }
251 TEST(TestDelete, DeleteElement)
252 {
253   fibonacci_heap_t* f_h = (fibonacci_heap_t*)malloc(sizeof(fibonacci_heap_t));
254   f_h->heap = NULL;
255   f_h->maxNodes = 0;
256   int value = 1;
257   node_t* new_node = (node_t*)malloc(sizeof(node_t));
258   new_node->key = value;
259   new_node->degree = 0;
260   new_node->marked = false;
261   new_node->visited = false;
262   new_node->left = new_node;
263   new_node->right = new_node;
264   new_node->parent = NULL;
265   new_node->child = NULL;
266   f_h->heap = new_node;
267   (f_h->maxNodes)++;
268   Delete(f_h, value);
269   EXPECT_TRUE(f_h->heap == NULL);
270   free(f_h);
271 }
272 TEST(TestDelete, TestError2)
273 {
274   fibonacci_heap_t* f_h = (fibonacci_heap_t*)malloc(sizeof(fibonacci_heap_t));
275   f_h->heap = NULL;
276   f_h->maxNodes = 0;
277   int value = 1;
278   node_t* new_node = (node_t*)malloc(sizeof(node_t));
279   new_node->key = value;
280   new_node->degree = 0;
281   new_node->marked = false;
282   new_node->visited = false;
283   new_node->left = new_node;
284   new_node->right = new_node;
285   new_node->parent = NULL;
286   new_node->child = NULL;
287   f_h->heap = new_node;
288   (f_h->maxNodes)++;
289   int val_del = 2;
290   node_t* node = f_h->heap;
```

```
291   Delete(f_h, val_del);
292   EXPECT_TRUE(f_h->heap != NULL);
293   free(f_h);
294 }
295
296 int main(int argc, char* argv[]) {
297   testing::InitGoogleTest(&argc, argv);
298   return RUN_ALL_TESTS();
299 }
```

Листинг 3: tests.cpp

# 5 Выводы

В ходе работы были созданы структуры Фибоначевой кучи и её узла, а также операции над ней: добавить элемент,найти миниум, сделать слияние, извлечь минимум, уменьшить ключ, удалить узел. Они соотвественно имеют свои алгоритмическое сложности: O(1), O(1), O(1), O(logn), O(1), O(logn). Также были разработаны unit тесты с помощью библиотеки для проверки написанных функций.

Из этого можно сделать вывод: Фибоначева куча быстрее(или как минимум не медленее) выполняет базовые операции по сравнению с биномиальной и двоичной кучами.