

Санкт-Петербургский политехнический университет Петра Великого  
Институт металлургии, машиностроения и транспорта  
Высшая школа автоматизации и робототехники

## **КУРСОВАЯ РАБОТА**

### **Фибоначчиева куча**

по дисциплине «Объектно-ориентированное программирование»

Выполнил: студент гр. 3331506/70401

Елисеев В.В.

Преподаватель:

Ананьевский М.С.

«\_\_\_» \_\_\_\_\_ 2021 г.

Санкт-Петербург

2021

## Содержание

1.	Введение .....	3
2.	Описание .....	4
3.	Реализация фибоначчиевой кучи.....	5
4.	Время работы .....	7
5.	Список литературы .....	8

## 1. Введение

Фибоначчиева куча — структура данных, отвечающая интерфейсу приоритетная очередь. Эта структура данных имеет меньшую амортизированную сложность, чем такие приоритетные очереди как биномиальная куча и двоичная куча.

Кучи являются основной структурой данных во многих приложениях.

Они применяются: при сортировке элементов; в алгоритмах выбора, для поиска минимума или максимума, медианы; в алгоритмах на графах, в частности, при построении минимального остовного дерева алгоритмом Крускала, при нахождении кратчайшего пути алгоритмом Дейкстры.

### Недостатки:

- Большое потребление памяти на узел
- Большая константа времени работы, что делает ее малоприменимой для реальных задач
- Некоторые операции в худшем случае могут работать за  $O(n)$  времени

### Достоинства:

- Одно из лучших асимптотических времен работы для всех операций

## 2. Описание

Фибоначчиева куча является структурой данных для хранения данных позволяющей быстро производить следующие операции: добавление элемента, получение минимального элемента, удаление минимального элемента, уменьшение ключа по ссылке и удаление по ссылке. Данная структура организована следующим образом:

- 1) Существует явная ссылка на минимальный элемент.
- 2) У каждой вершины есть ссылка на правый и левый элемент в двусвязном списке содержащим эту вершину.
- 3) У каждой вершины есть ссылка *child* указывающая на одну из вершин списка ее детей.
- 4) У каждой вершины есть ссылка *parent* указывающая на родителя.
- 5) У каждой вершины есть булевское поле *marked* используемая при уменьшении ключа. Оно истинно если вершина потеряла ребенка после того, как сделалась чьим-нибудь ребенком.
- 6) Список содержащей минимальную вершину называется корневым списком, и родители всех его вершин отсутствуют.

### 3. Реализация фибоначчиевой кучи

Для возможности быстрого удаления элемента из произвольного места и объединением с другим списком будем хранить их в циклическом двусвязном списке. Также будем хранить и все уровни поддерева. Исходя из этого структура каждого узла будет выглядеть вот так.

```
struct node {  
    node* parent; // указатель на родительский узел  
    node* child; // указатель на один из дочерних узлов  
    node* left; // указатель на левый узел того же предка  
    node* right; // указатель на правый узел того же предка  
    int key; // ключ  
    int degree; // Степень узла  
    char mark; // Отметка узла  
    char c; // Флаг для помощи в функции поиска узла  
};  
  
struct node* mini = NULL; //Создание указателя на мин значение  
int size = 0; //Объявить целое число для количества узлов в куче
```

Рисунок 4.1 – Создание структуры

Чтобы не дублировать код, объясню на словах, как действуют определенные функции.

#### ***Insertion()***

Вставка элемента. Данная операция вставляет новый элемент в список корней правее минимума и при необходимости меняет указатель на минимум кучи.

#### ***Fibonnaci\_link()***

Для сливания двух Фибоначчиевых куч необходимо просто объединить их корневые списки, а также обновить минимум новой кучи, если понадобится. Вынесем в вспомогательную функцию *Fibonnaci\_link* логику, объединяющую два списка вершины, которых подаются ей в качестве аргументов.

#### ***Consolidate()***

Сливаем два корневых списка в один и обновляем минимум, если нужно.

#### ***Extract\_min()***

Удаление минимального элемента. Первая рассматриваемая операция, в ходе которой значительно меняется структура кучи. Здесь используется вспомогательная процедура *consolidate*, благодаря которой, собственно, и достигается желанная амортизированная оценка.

### ***Cut()***

Вырезание. При вырезании вершины мы удаляем ее из списка детей своего родителя, уменьшаем степень ее родителя и снимаем пометку с текущей вершины.

### ***Cascade\_cut()***

Каскадное вырезание. Перед вызовом каскадного вырезания нам известно, удаляли ли ребенка у этой вершины. Если у вершины до этого не удаляли дочерний узел, то мы помечаем эту вершину и прекращаем выполнение операции. В противном случае применяем операцию *cut* для текущей вершины и запускаем каскадное вырезание от родителя.

### ***Decrease\_key()***

Функция для уменьшения значения узла в куче.

### ***Find()***

Как бы странно это не звучало, но это функция необходима для поиска определенного узла, с целью его последующего удаления или доступа.

### ***Deletion()***

Удаление вершины реализуется через уменьшение ее ключа до 0 и последующим извлечением минимума.

### ***display()***

Показывает все элементы в куче в выстроенном порядке.

### ***find\_min()***

Показывает минимум.

### ***interview()***

Функция, которая опрашивает пользователя. Некоторые функции изначально недоступны пользователю, с целью снижения загруженности интерфейса.

## 4. Время работы

Для анализа производительности операций введем потенциал для фибоначчиевой кучи как  $\Phi = trees + 2 \cdot marked$ , где  $trees$  — количество элементов в корневом списке кучи, а  $marked$  — количество вершин, у которых удален один ребенок (то есть вершин с пометкой  $x.mark = true$ ). Договоримся, что единицы потенциала достаточно для оплаты константного количества работы.

### Вставка элемента

Для оценки амортизированной стоимости операции рассмотрим исходную кучу  $H$  и получившуюся в результате вставки нового элемента кучу  $H'$ .  $trees(H') = trees(H) + 1$  и  $marked(H') = marked(H)$ . Следовательно, увеличение потенциала составляет  $(trees(H) + 1 + 2 \cdot marked(H)) - (trees(H) + 2 \cdot marked(H)) = 1$ . Так как реальное время работы составляет  $O(1)$ , то амортизированная стоимость данной операции также равна  $O(1)$ .

### Получение минимального элемента

Истинное время работы —  $O(1)$ .

### Соединение двух куч

Реальное время работы —  $O(1)$ . Амортизированное время работы также  $O(1)$ , поскольку, при объединении двух куч в одну, потенциалы обеих куч суммируются, итоговая сумма потенциалов не изменяется,  $\Phi_{n+1} - \Phi_n = 0$ .

Таблица 1 – Время выполнения различных операций

Операция	Амортизированная сложность
makeHeap	$O(1)$
getMin	$O(1)$
merge	$O(1)$
extractMin	$O(\log(n))$
decreaseKey	$O(1)$
delete	$O(\log(n))$
insert	$O(1)$

## 5. Список литературы

- Томас Кормен, Чарльз Лейзерсон, Рональд Ривест, Клиффорд Штайн — Алгоритмы: построение и анализ. — М.: Издательский дом «Вильямс», 2005. — С. 1296. — ISBN 5-8459-0857-4
- Mehlhorn, Kurt, Sanders, Peter. 6.2.2 Fibonacci Heaps // Algorithms and Data Structures: The Basic Toolbox. — Springer, 2008. — 300 с. — ISBN 978-3-540-77978-0.
- Fibonacci Heaps — Duke University
- Fibonacci Heaps — Princeton University



