Оглавление

Глава 1. Теория.

Условие задачи.

Реализовать класс «Фибоначчиева куча» (в корне – максимум), а также реализовать необходимые для класса функции: конструкторы, деструкторы, нужные методы: добавление элемента, удаление максимума, поиск максимума, изменения приоритета некоторого элемента, слияния куч, поиска квантиля (можно ли это сделать?). Продемонстрировать работу операций добавления нового элемента, удаления корня, поиска максимума, изменения приоритета некоторого элемента, слияния куч, поиска квантиля, вывод элементов по убыванию приоритета.

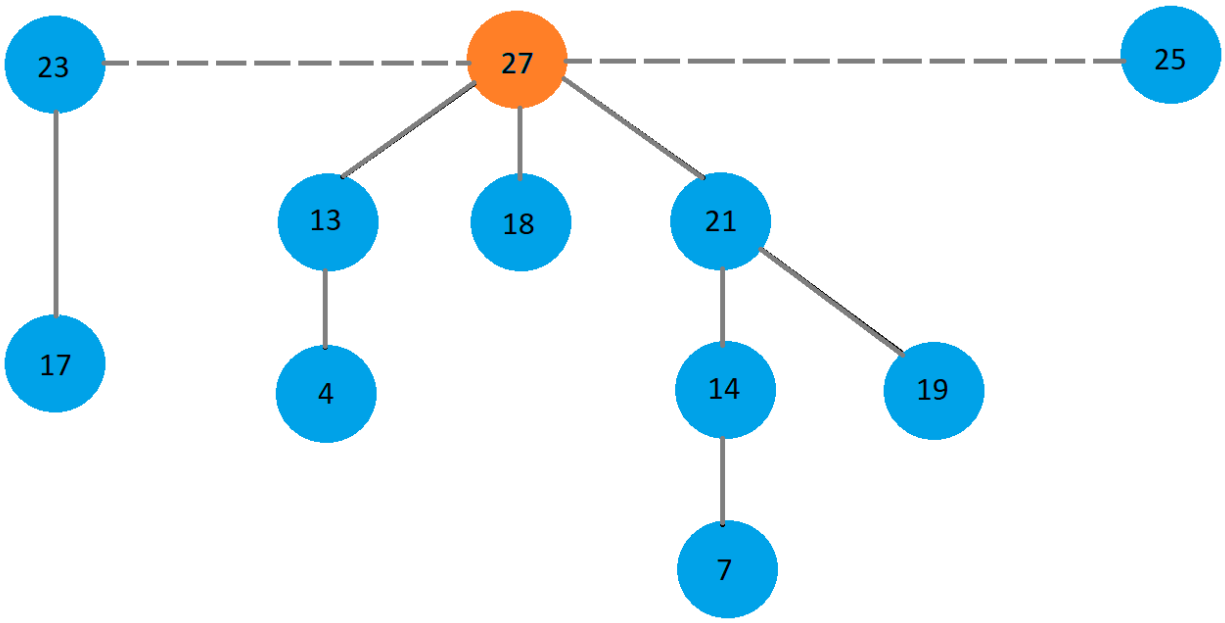
Что такое «Фибоначчиева куча»?

Фибоначчиева куча (англ. *Fibonacci heap*) — структура данных, отвечающая интерфейсу приоритетная очередь. Эта структура данных имеет меньшую амортизированную сложность, чем такие приоритетные очереди как биномиальная и двоичная куча. Изначально эта структура данных была разработана Майклом Фридманом и Робертом Тарьяном при работе по улучшению асимптотической сложности алгоритма Дейкстры. Свое название «Фибоначчиева куча» получила из-за использования некоторых свойств чисел Фибоначчи в потенциальном анализе этой реализации.

Структура «Фибоначчиевой кучи».

«Фибоначчиева куча» — набор из подвешенных деревьев удовлетворяющих свойству: каждый предок не меньше своих детей (если дерево на максимум). Это означает, что максимум всей кучи это один из корней этих деревьев. Одно из главных преимуществ *Фибоначчиевой кучи* — гибкость её структуры из-за того, что на деревья не наложены никакие ограничения по форме. Например, *Фибоначчиева куча* может состоять хоть из деревьев в каждом из которых по одному элементу. Такая гибкость позволяет выполнять некоторые операции лениво, оставляя работу более поздним операциям.

Рисунок 1.1. Пример *«Фибоначчиевой кучи»* с максимумом в корне.



Сравнение «Фибоначчиевой кучи» с биномиальной, двоичной, бинарной кучей, массивом, связным списком, сбалансированным деревом поиска.

Таблица 1.1. Сравнительная таблица «Фибоначчиевой кучи» с биномиальной, двоичной, бинарной кучей, массивом, связным списком и сбалансированным деревом поиска.

Критерии сравнения структур	Биномиальная куча	Двоичная куча	Массив	Связный список	Сбалансированное дерево поиска	Фибоначчиева куча
Потребление памяти	Требуется память для ссылок или указателей на корень и его детей и братские деревья. Также требуется выделение памяти под степень вершины (количество дочерних узлов данного узла).	Требуется память для ссылок или указателей на узлы, содержащие значения и ссылки на левого и правого потомков.	Требуется память под непрерывный блок, в котором расположены элементы массива.	Требуется память под отдельные объекты узлов, которые связываются между собой с помощью указателей или ссылок.	Требуется память под отдельные объекты узлов, которые связываются между собой с помощью указателей или ссылок на правого и левого потомков. Для поддержания баланса также используется память под высоту дерева и указатель на корень дерева.	Требуется память для каждого узла дерева, а также дополнительные ссылки и указатели на один из дочерних классов узла, на левый и правый узел того же предка. Также необходимо выделить память под степень каждой вершины и ее метку. В «Фибоначчиевой куче» также должна выделяться память под указатель на корень дерева и размер кучи.
Добавление элемента (Push)	O(log n), где n - количество элементов в куче.	O(log n), где n - количество элементов в куче.	Время добавления элемента в конец массива	*O*(1)	O(log n) времени, где n - количество элементов в дереве.	*O*(1)

			составляет O(1). Однако, если массив требуется расширить, то это может занять O(n) времени, где n - текущий размер массива.			
Удаление максимума/минимума (ExtractMax)	O(log n), где n - количество элементов в куче.	O(log n), где n - количество элементов в куче.	O(n), где n - количество элементов в массиве.	O(n), где n - количество элементов в связном списке.	O(log n) времени, где n - количество элементов в дереве.	O(log n), где n - количество элементов в куче.
Поиск максимума (FindMax)	O(log n), где n - количество элементов в куче.	*O*(1)	O(n), где n - количество элементов в массиве.	O(n), где n - количество элементов в связном списке.	O(log n) времени, где n - количество элементов в дереве.	*O*(1)
Изменение приоритета некоторого элемента (EncreaseData)	O(log n) времени, где n - количество элементов в куче. Необходимо найти элемен	O(log n) времени, где n - количество элементов в куче.	Изменение приоритета некоторого элемента в массиве зависит от того, есть ли информа	Изменение приоритета некоторого элемента в связном списке зависит от того, есть ли информа	O(log n) времени, где n - количество элементов в дереве.	Изменение приоритета некоторого элемента в «Фибоначчиевой» куче занимает *O*(1) амортизированно. Можно просто обновить значение приоритета в

	т в куче и обновить его приоритет, а затем выполнить операции перебалансировки для восстановления свойств кучи.		информация о его позиции в массиве. Если позиция известна, то изменение приоритета может занять O(1) времени. Если позиция неизвестна, то необходимо выполнить поиск элемента в массиве, что может занимать O(n) времени в худшем случае.	ция о его позиции в списке. Если позиция известна, то изменение приоритета может занять O(1) времени, поскольку можно просто обновить значение приоритета в узле. Если позиция неизвестна, то необходимо выполнить поиск элемента в списке, что может занимать O(n) времени в худшем случае.		соответствующем узле, не требуя дополнительных операций перебалансировки.
Слияние куч (Join)	O(log n), где n - общее количество элемен	O(log n), где n - общее количество элементов в	Слияние двух массивов требует выделе	Слияние двух связных списков занимает O(1) времени,	O(mlog n), где m и n - количество элементов в каждом дереве. Необходим	O(1) Можно просто объединить две кучи, обновив указатели на

	тов в обеих кучах. Необходимо выполнить операции объединения поддеревьев с одинаковыми рангами и перебалансировки для восстановления свойств кучи.	обеих кучах. Необходимо выполнить операции сравнения и перестроения дерева для объединения двух куч.	ния нового массива и копирования элементов из обоих массивов в новый массив. Занимает O(m + n), где m и n - размеры двух массивов.	поскольку можно просто обновить указатели на начало и конец списков.	о перебалансировать дерево после вставки каждого элемента из одного дерева в другое.	минимальный элемент.
Удаление элемента (Remove)	O(log n), где n - количество элементов в куче.	O(log n) времени, где n - количество элементов в куче.	Если важен порядок элементов, удаление может занимать O(n) времени, где n - размер массива. Однако, если порядок элементов в массиве не важен,	В связном списке удаление элемента происходит в среднем за O(1) время, если имеется ссылка на удаляемый элемент. Однако, если для удаления элемента требуетс	Удаление элемента в сбалансированном дереве поиска занимает O(log n) времени, где n - количество элементов в дереве. При удалении элемента необходимо выполнить ребалансировку дерева, чтобы сохранить его сбалансиро	Удаление элемента в фибоначчиевой куче имеет амортизированную сложность O(log n), где n - количество элементов в куче. Фибоначчиева куча обеспечивает эффективное удаление элемента, включая обновление структуры кучи и объединение дочерних

			можно выполнить удаление за константное время, просто заменив удаленный элемент последним элементом и уменьшив размер массива.	я поиск его позиции в списке, то время удаления может составить O(n), где n - количество элементов в списке, так как потребуется пройти по всему списку до места удаления.	ванное состояние.	куч для поддержания свойств кучи.

Исходя из результатов сравнения таблицы 1.1 можно сделать некоторые заключения, которые помогут при выборе структуры для хранения данных:

- Используйте биномиальную кучу, когда вам нужно эффективно выполнять операции вставки, удаления и извлечения минимального (максимального) элемента. Она также может быть полезна, когда вам нужно выполнять слияние двух куч.
- Используйте двоичную кучу, когда вам нужно эффективно выполнять операции вставки, удаления и извлечения минимального (максимального) элемента. Она обычно проста в реализации и может быть хорошим выбором для основных операций кучи.
- Используйте массив, когда вам требуется простая структура данных для хранения элементов и быстрый доступ к элементам по индексу. Массивы обеспечивают постоянное время доступа к элементам по индексу, но могут быть неэффективными при вставке и удалении элементов в середине или начале массива.
- Используйте связный список, когда вам требуется динамическая структура данных с эффективной вставкой и удалением элементов в середине или начале списка. Связные списки обеспечивают гибкость в изменении размера и отсутствие необходимости в непрерывной памяти, но их эффективность в поиске и доступе к элементам хуже, чем у массивов или куч.
- Используйте сбалансированное дерево поиска, когда вам нужно эффективно выполнять операции поиска, вставки и удаления элементов, а также поддерживать отсортированный порядок элементов. Сбалансированные деревья поиска обеспечивают логарифмическое время выполнения этих операций и могут быть полезными в случаях, когда требуется быстрый доступ к отсортированным данным.

- Используйте фибоначчиеву кучу, когда вам нужно эффективно выполнять операции вставки, удаления и извлечения минимального (максимального) элемента, а также изменение приоритета элементов. Фибоначчиевы кучи обычно имеют лучшую амортизированную производительность при некоторых операциях, но требуют более сложной реализации.

Сравнение особенностей поиска квантиля в структурах данных: «Фибоначчиева куча», биномиальная, двоичная куча, сбалансированное дерево поиска, связный список, массив.

Сбалансированное дерево поиска:

Сбалансированное дерево поиска поддерживает отсортированный порядок элементов, вследствие чего все элементы, меньшие (или большие) определенного значения, находятся в одной части дерева, что упрощает поиск элементов по квантилю. При поиске квантиля можно определить, в какой части дерева следует продолжить поиск и, таким образом, уменьшить размер пространства поиска и ускорить операцию поиска.

Массив:

В массиве элементы хранятся в упорядоченном порядке, и поиск квантиля может быть выполнен путем бинарного поиска. Это позволяет находить элементы в массиве по значению и определять их положение в упорядоченном массиве.

Связный список:

Связный список не является оптимальной структурой данных для поиска квантиля, так как требуется последовательный проход по элементам списка. Однако, если список отсортирован, можно использовать поиск квантиля, выполнив последовательное смещение по элементам списка до достижения нужного квантиля.

Амортизированное время поиска квантиля в «Фибоначчиевой куче», биномиальной, бинарной, двоичной куче равно $O(\log n)$, где n - общее количество элементов в куче. Однако в разных структурах данных существуют особенности для его поиска, поэтому стоит их рассмотреть.

Фибоначчиева куча:

Поиск квантиля в Фибоначчиевой куче может быть несколько более сложным и менее эффективным, чем в других кучах, таких как биномиальная и двоичная. Это связано с тем, что Фибоначчиева куча оптимизирована для других операций, таких как вставка, удаление и объединение. Для поиска квантиля в Фибоначчиевой куче требуется выполнить обход кучи, проверяя размер поддеревьев и переходя к соответствующим вершинам.

Биномиальная куча:

Биномиальная куча может быть лучшим выбором для поиска квантиля по сравнению с Фибоначчиевой кучей. Это связано с тем, что в биномиальной куче имеется прямой доступ к корневым элементам деревьев, что упрощает поиск. Для поиска квантиля в биномиальной куче может использоваться алгоритм двоичного поиска, позволяющий эффективно находить искомый квантиль.

Двоичная куча:

Двоичная куча также может быть неплохим выбором для поиска квантиля, особенно если она является сбалансированной. Для поиска квантиля в двоичной куче также может использоваться алгоритм двоичного поиска, который позволяет эффективно находить искомый квантиль.

Исходя из особенностей поиска квантилей в разных структурах данных, можно сделать вывод, что сбалансированное дерево поиска, связный список и массив подходят для поиска квантиля лучше всего, однако в фибоначчиевой, биномиальной и двоичной куче можно реализовать алгоритм поиска квантиля, но он будет менее эффективным и сложным по сравнению с сбалансированным деревом поиска и массивом.

Достоинства и недостатки «Фибоначчиевой кучи».

Недостатки:

- Большое потребление памяти на узел (минимум 21 байт)
- 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

Таблица 1.2. Итоговая таблица амортизированной сложности операций «Фибоначчиевой кучи» с максимумом в корне.

Добавление элемента (Push)	O(1)
Удаление элемента (Remove)	O(log n)
Слияние куч (Join)	O(1)
Извлечение максимума (ExtractMax)	O(log n)
Поиск максимума (FindMax)	O(1)
Изменение приоритета некоторого элемента (EncreaseData)	O(1)
Поиск квантиля (FindQuantile)	O(log n)

Глава 2. Практика.

Для решения задачи использовались следующие заголовочные файлы из стандартной библиотеки C++:

- iostream – необходима для ввода и вывода.
- string – инструментарий для работы со строками.
- cmath – набор математических функций и констант для работы с числами.

Весь код был написан в пространстве имен std.

Для решения задачи были реализованы несколько классов:

- Класс Schoolboy (школьник) - потребуется для демонстрирования работы «Фибоначчиевой кучи» со сложным классом.
- Класс HeapException – потребуется для выброса стандартного исключения при работе с «Фибоначчиевой кучей».
- Класс InvalidArgument – производный класс от HeapException, который потребуется для выброса исключения при передаче неправильного аргумента в методы «Фибоначчиевой кучи».
- Класс FibHeapNode – потребуется для содержания отдельного узла «Фибоначчиевой кучи».
- Класс FibMaxHeap – реализация «Фибоначчиевой кучи», в корне которой находится максимум.

Рассмотрим каждый класс детальнее:

Класс Schoolboy (школьник).

Private переменные (поля данного класса):

1. Переменные типа string (last_name – фамилия школьника, first_name – имя школьника, address – адрес школьника).
2. Переменные типа int (group – группа (класс) школьника, date_of_birth – год рождения школьника, gender – 1 (если школьник является представителем мужского пола), 0 (если школьник является представителем женского пола)).

Public методы:

1. Конструктор по умолчанию.
2. Конструктор, принимающий параметры (last_name, first_name, address, group, date_of_birth, gender).
3. Конструктор копий.
4. Деструктор.
5. Переопределение операторов больше, меньше или равно для сравнения элементов класса по приоритетам для дальнейшего их применения в «Фибоначчиевой куче» (приоритеты для сравнения элементов в порядке убывания приоритета: group, date_of_birth, last_name, first_name).

Необходимо объявить дружественными переопределенные операторы ввода и вывода для класса Schoolboy, реализованные вне нашего класса.

Класс HeapException.

Private переменная (поле данного класса):

1. Указатель на символ (char*) для представления строк в виде последовательностей символов в памяти – str (сообщение об ошибке).

Public методы:

1. Конструктор, принимающий сообщение об ошибке.
2. Конструктор копий.
3. Деструктор.
4. Вывод сообщения об ошибке.

Класс InvalidArgument.

Private переменная (поле данного класса):

1. Указатель на символ (char*) для представления строк в виде последовательностей символов в памяти – str (сообщение об ошибке).

Public методы:

1. Конструктор, принимающий сообщение об ошибке (унаследован от HeapException).
2. Конструктор копий (унаследован от HeapException).
3. Деструктор.
4. Вывод сообщения об ошибке.

Шаблонный класс FibHeapNode.

Для него необходимо объявить дружественным класс FibMaxHeap для удобства обращению к полям (так как до этого класс FibMaxHeap не был определен необходимо заранее определить, что в дальнейшем данный класс будет реализован!).

Для возможности быстрого удаления элемента из произвольного места и объединением с другим списком будем хранить элементы в **циклическом двусвязном списке**. Также стоит упомянуть, что нам нужен указатель только на одного ребенка, поскольку остальные хранятся в двусвязном списке с ним!

Private переменные (поля данного класса):

1. Переменная произвольного типа - _data, которая будет содержать данные об узле.
2. Указатели типа FibHeapNode<T> (_parent – указатель на родительский узел, _child – указатель на один из дочерних узлов, _left – указатель на левый узел того же предка, _right – указатель на правый узел того же предка).
3. Переменная типа int - _degree (степень вершины), то есть сколько у данной вершины детей.
4. Переменная типа bool - _marked, обозначающая был ли удален ребенок в процессе изменения ключа ребенок данной вершины.

Public методы:

1. Конструктор по умолчанию.
2. Конструктор, принимающий в виде аргумента переменную произвольного типа _data, которая будет содержать информацию об узле.
3. Деструктор.

4. Геттеры и сеттеры для полей класса FibHeapNode.

Также необходимо объявить дружественным в public переопределенный оператор вывода для класса FibHeapNode, реализованный вне нашего класса.

Шаблонный класс FibMaxHeap.

Итак, для доступа ко всей куче нам тоже нужен всего один элемент, поэтому разумно хранить именно указатель на максимум кучи (он обязательно один из корней), а для получения размера за константное время будем хранить размер кучи отдельно.

Protected переменные (поля данного класса):

1. Указатель типа FibHeapNode – указатель на корень кучи _root (всегда максимум).
2. Переменная типа int - _size, обозначающая размер кучи.

Приступим к рассмотрению реализации методов «Фибоначчиевой кучи», представленных в задании.

Добавление элемента в «Фибоначчиеву кучу».

Для добавления элемента мной был реализовать метод Push и связанные с ними методы Add, Insert и Remove в классе FibMaxHeap.

Метод Push служит для добавления нового элемента в кучу. Он создает новый узел FibHeapNode<T> с данными data и вызывает метод Add, передавая ему созданный узел.

Метод Add добавляет узел в кучу, учитывая правильное размещение в списке корней и обновляет указатель на максимальный элемент (_root). Сначала он проверяет, есть ли уже элементы в куче. Если куча пуста, добавляемый узел становится новым корнем. В противном случае вызывается метод Insert, который вставляет узел в список корней. Затем проверяется, является ли добавленный узел новым максимальным элементом, и если это так, то указатель на максимальный элемент обновляется.

Реализация метода Add представлена на рисунке 2.1.

```
template<class T>
FibHeapNode<T>* FibMaxHeap<T>::Add(FibHeapNode<T>* node)
{
    if (_root == nullptr) // Если в куче нет элементов, то только что добавленный максимальный.
        _root = node;
    else // Иначе аккуратно меняем указатели в списке, чтобы не перепутать указатели
    {
        Insert(node);

        if (node->_data > _root->_data) // Передвигаем указатель на новый корень
            _root = node;
    }

    _size++;

    return node;
}
```

Рисунок 2.1. Реализация добавления узла в кучу, учитывая правильное размещение в списке корней с помощью метода Add.

Метод Insert выполняет аккуратную вставку узла слева от корня. Сначала он удаляет узел из списка, если он уже там присутствует, вызывая метод Remove. Затем происходит перестановка указателей так, чтобы вставляемый узел оказался между предыдущим левым соседом корня и самим корнем.

Реализация метода Insert представлена на рисунке 2.2.

```
template<class T>
void FibMaxHeap<T>::Insert(FibHeapNode<T>* node)
{
    Remove(node); // Удаляем узел из списка, если он там уже присутствует.

    FibHeapNode<T>* rootLeft = _root->_left;

    node->_right = _root;
    _root->_left = node;

    if (rootLeft != nullptr)
    {
        node->_left = rootLeft;
        rootLeft->_right = node;
    }
}
```

Рисунок 2.2. Реализация аккуратной вставки узла слева от корня с помощью метода Insert.

Метод Remove выполняет удаление узла из списка корней кучи. Он обновляет ссылки на левую и правую связи у соседних узлов, чтобы они указывали друг на друга, пропуская удаляемый узел. Таким образом, узел извлекается из списка корней.

Реализация метода Remove представлена на рисунке 2.3.

```
template<class T>
void FibMaxHeap<T>::Remove(FibHeapNode<T>* node)
{
    FibHeapNode<T>* left = node->_left;
    FibHeapNode<T>* right = node->_right;

    left->_right = right;
    right->_left = left;

    // Обнуляем ссылки на левую и правую связи, чтобы node не указывал ни на какие другие узлы.
    node->_left = node;
    node->_right = node;
}
```

Рисунок 2.3. Реализация удаление узла из списка корней кучи с помощью метода Remove.

Поиск максимума в «Фибоначчиевой куче».

Так как в корне «Фибоначчиевой кучи» лежит максимум, то он и будет максимальным элементом в нашей структуре данных.

Слияние «Фибоначчиевых куч».

Для слияния двух Фибоначчиевых куч необходимо просто объединить их корневые списки, а также обновить максимум новой кучи, если понадобится.

Реализация слияния двух «Фибоначчиевых куч» было разделено мной на 4 метода.

Первая версия Join нужна для удобства использования (чтобы можно было передавать целую кучу для объединения). Она принимает указатель на другую Фибоначчиеву кучу и вызывает вторую версию Join, передавая ей корневой узел другой кучи. В данном методе также выполняется корректировка размера кучи.

Вторая версия Join принимает корневой узел и выполняет процесс объединения двух корневых списков куч. Она проверяет на пустоту нашу кучу и, если наша куча пуста, просто устанавливает корневой узел переданной кучи в нашу кучу. В противном случае, она вызывает метод Union, который объединяет корневой список переданного узла с корневым списком нашей кучи. После объединения корневых списков, проверяется, значение данных корневого узла и обновляется указатель на новый максимальный узел, если переданный корневой узел больше заданного.

Реализация методов Join представлена на рисунке 2.3.

```
template<class T>
void FibMaxHeap<T>::Join(FibMaxHeap<T>* heap)
{
    if (heap->_root != nullptr)
        Join(heap->_root);

    _size += heap->_size;
}

template<class T>
void FibMaxHeap<T>::Join(FibHeapNode<T>* node)
{
    if (node == nullptr)
        throw HeapException("We can't connect to an empty heap");

    if (_root == nullptr) // если наша куча пуста, то результатом будет вторая куча
        _root = node;
    else // иначе объединяем два корневых списка
        Union(node);

    if (node->_data > _root->_data) // если максимум кучи изменился, то надо обновить указатель
        _root = node;
}
```

Рисунок 2.4. Реализация двух версий методов Join, для объединения корневых списков куч.

Метод Union принимает два узла first и second и выполняет операцию объединения, чтобы связать их вместе в куче (для удобства восприятия узел first будет располагаться справа от узла second). Для выполнения объединения, сначала сохраняются указатели на левого и правого соседа узла, чтобы не потерять связи при изменении указателей. Затем мы

связываем узлы first и second так, чтобы узел first стал правым соседом узла second, а узел second стал левым соседом узла first. Далее, восстанавливаются связи с соседними узлами.

Мной также была реализована вторая версия метода Union, когда мы передаем только один параметр. Данный метод объединяет вершину, передаваемую нами с корнем справа. Данный метод позволяет удобно переносить узлы к корню.

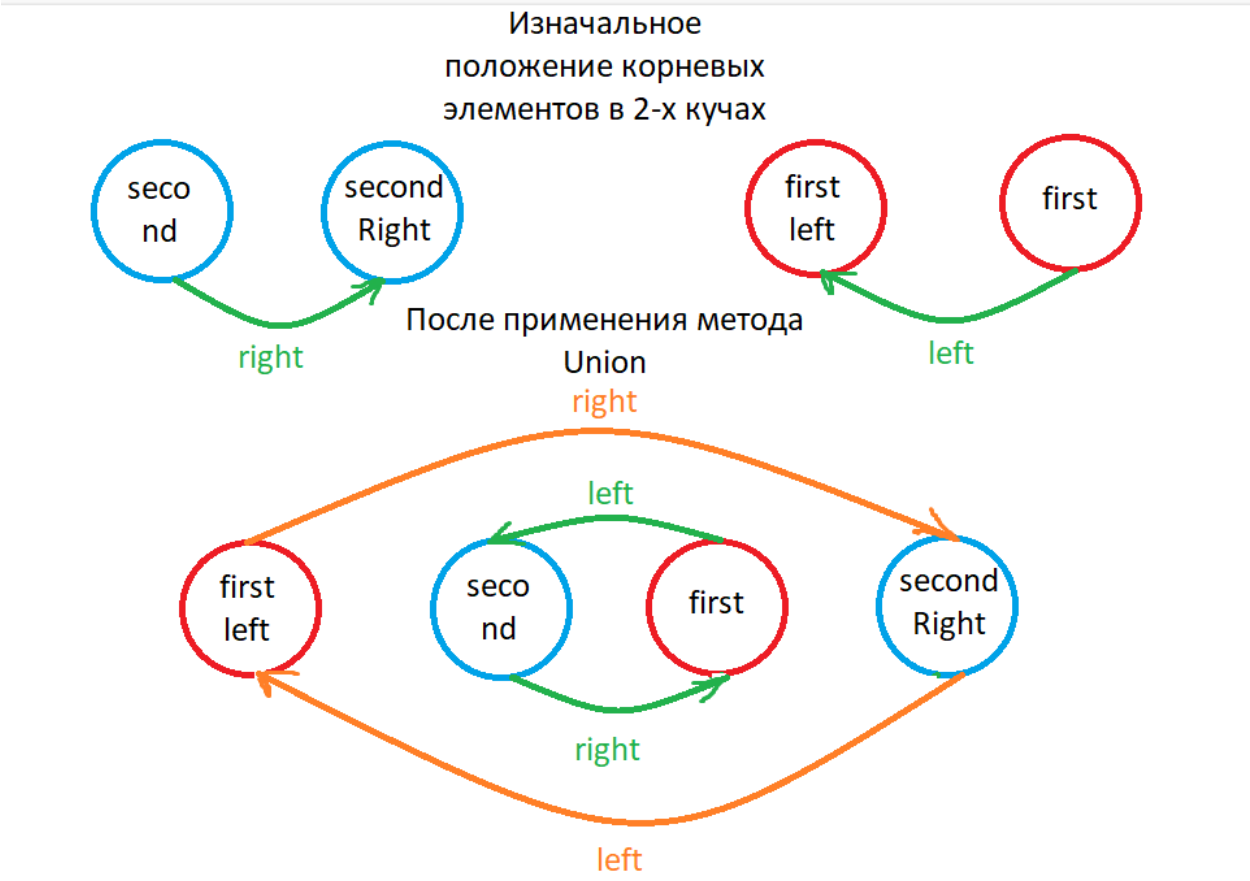


Рисунок 2.5. Объединение двух узлов, чтобы связать их вместе в куче с помощью метода Union.

Удаление максимума в «Фибоначчиевой куче».

Для извлечения и возврата максимального элемента из Фибоначчиевой кучи мной был реализован метод ExtractMax.

Поэтапные шаги работы метода ExtractMax:

1. Выполняем проверку на пустоту кучи. Если куча не пуста, переходим ко второму этапу.
2. Если корневой узел имеет дочерние узлы, вызывается метод Union, который объединяет корневой список с дочерним списком.
3. Затем корневой узел отсоединяется от списка корней. Это достигается обновлением ссылок на левую и правую связи у соседних узлов, чтобы они указывали друг на друга, пропуская удаляемый корневой узел.
4. Уменьшаем размер кучи на 1.

5. Проверяем, остался ли только один узел в куче. Если да, то обнуляем корень, если нет, устанавливается новый корневой узел как следующий узел после полученного максимального узла. Затем вызывается процедура консолидации кучи (Consolidate), которая выполняет слияние узлов с одинаковыми степенями и восстанавливает свойства «Фибоначчиевой кучи».
6. Возвращаем извлеченный максимальный узел.

Код реализации извлечения и возврата максимального элемента из «Фибоначчиевой кучи» представлен на рисунке 2.6.

```
template<class T>
FibHeapNode<T>* FibMaxHeap<T>::ExtractMax()
{
    if (_root == nullptr) // Если нечего удалять
        return nullptr;

    FibHeapNode<T>* prev = _root;

    if (_root->_child != nullptr)
        Union(_root->_child); // Объединяем корневой список с дочерним списком, если у корневого узла есть дочерние узлы

    // Отсоединяем корневой узел от списка корней

    FibHeapNode<T>* left = _root->_left;
    FibHeapNode<T>* right = _root->_right;

    left->_right = right;
    right->_left = left;

    _size--;

    if (prev == prev->_right) // Если в куче остался только один узел
        _root = nullptr;
    else // В противном случае, если в куче остаются еще узлы, устанавливем новый корневой узел и выполняем процедуру консолидации кучи.
    {
        _root = prev->_right;

        Consolidate();
    }

    return prev;
}
```

Рисунок 2.6. Реализация метода ExtractMax по извлечению и возврату максимального элемента из «Фибоначчиевой кучи».

В ходе поэтапного рассмотрения работы ExtractMax, изложенного выше, был упомянут метод прорежения деревьев.

Процесс консолидации (прорежение деревьев) выполняется после удаления максимального элемента из кучи. Он сканирует список корневых узлов и объединяет узлы с одинаковой степенью, путем создания нового узла с более высокой степенью и сделки между узлами с наименьшим и наибольшим значением данных. Это позволяет поддерживать уникальные степени у корневых узлов и сохранять свойство «Фибоначчиевой кучи».

Для реализации процесса консолидации понадобится вспомогательный метод Link, который используется для связывания узлов. Метод Link играет важную роль в процессе консолидации и объединения узлов в «Фибоначчиевой куче». Он обеспечивает правильное установление связей между узлами, сохраняя структурную целостность кучи.

Работа данного метода заключается в пяти этапах:

1. Убираем связи у дочерней вершины путем удаления узла из списка, в котором он находится.

2. Подвешиваем дочерний узел к родительскому узлу и сбрасываем флаг у дочернего узла.
3. Если у родительского узла нет дочернего узла, то child становится его единственным дочерним узлом. Устанавливаются связи таким образом, чтобы child указывал на себя в качестве левого и правого соседа.
4. Если у родительского узла уже есть другие дочерние узлы, то child добавляется в список дочерних узлов справа от текущего самого левого дочернего узла. Устанавливаются соответствующие связи, чтобы child встал между текущим самым левым дочерним узлом и его левым соседом.
5. Увеличиваем степень родительского узла на единицу, чтобы отразить добавление нового дочернего узла.

Реализация связывания узлов представлена на рисунке 2.7.

```
template<class T>
void FibMaxHeap<T>::Link(FibHeapNode<T>* child, FibHeapNode<T>* parent)
{
    // Убираем связи у дочерней вершины
    Remove(child);

    // Подвешиваем вершину
    child->_parent = parent;
    child->_marked = false;

    // Даём родителю указатель на дочерний узел, если его нет
    if (parent->_child == nullptr)
    {
        parent->_child = child;
        child->_left = child;
        child->_right = child;
    }
    else
    {
        // Добавляем новых соседей (слева от ребенка)
        FibHeapNode<T>* parentChild = parent->_child;

        child->_left = parentChild->_left;
        child->_right = parentChild;

        parentChild->_left->_right = child;
        parentChild->_left = child;
    }

    parent->_degree++;
}
```

Рисунок 2.7. Реализация связывания узлов с помощью метода Link.

Теперь посмотрим на визуализацию процесса консолидации на рисунке 2.8.

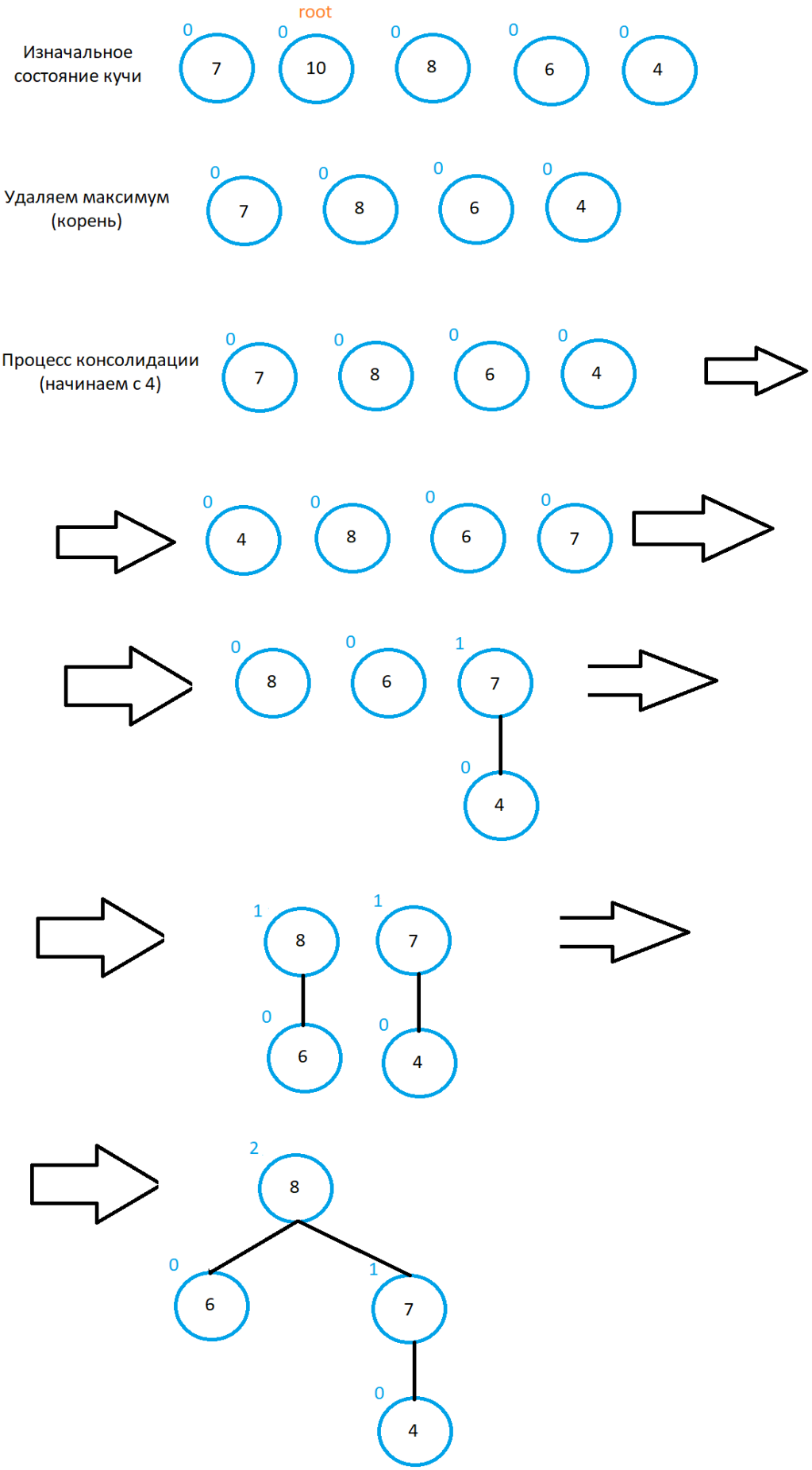

Изначальное состояние кучи
root
0 7
0 10
0 8
0 6
0 4
Удаляем максимум (корень)
0 7
0 8
0 6
0 4
Процесс консолидации (начинаем с 4)
0 7
0 8
0 6
0 4
0 4
0 8
0 6
0 7
0 8
0 6
1 7
0 4
1 8
0 6
1 7
0 4
2 8
0 6
1 7
0 4

Рисунок 2.8. Визуализация процесса консолидации.

Опишем процесс консолидации по рисунку 2.8.

Консолидация – процесс, аналогичный слиянию биномиальных куч: добавляем поочередно каждый корень, смотря на его степень. Пусть она равна d. Если в соответствующей ячейке array еще нет вершины, записываем текущую вершину туда. Иначе подвешиваем одно дерево к другому, и пытаемся также добавить дерево, степень корня которого уже равна d+1. Продолжаем, пока не найдем свободную ячейку. Подвешиваем мы его следующим образом: в корневой список добавляем корень максимальный из тех двух, а корень другого добавляем в список детей корневой вершины.

Изменение приоритета некоторого элемента в «Фибоначчиевой куче».

Для того чтобы реализовать изменение приоритета некоторого элемента в «Фибоначчиевой куче» понадобятся 2 вспомогательных метода: операция отрезания (cut) и каскадного отрезания узлов (cascading cut).

Операция отрезания (Cut):

Когда узел удаляется из кучи, его необходимо отсоединить от его родительского узла и добавить в корневой список кучи. Метод Cut выполняет эти действия. Он удаляет связи между родителем и ребенком, уменьшает степень родительского узла и обновляет указатель на дочерний узел, если отрезанный узел был первым дочерним узлом родителя. Затем отрезанный узел добавляется в корневой список кучи с помощью объединения (подвешивания).

Каскадное отрезание (CascadingCut):

Когда происходит отрезание узла, возникает возможность, что его родительский узел также был отрезан ранее, что может привести к нарушению структурной целостности кучи. Метод CascadingCut решает эту проблему, применяя каскадное отрезание. Если узел был отрезан от своего родителя, то рекурсивно вызывается CascadingCut для родительского узла. Это гарантирует, что все родительские узлы, которые были отрезаны ранее, также будут отрезаны и добавлены в корневой список.

Функции Cut и CascadingCut важны для поддержания баланса и оптимальной структуры «Фибоначчиевой кучи». Они предотвращают увеличение степени узлов и обеспечивают быстрое выполнение операций, таких как удаление максимального элемента и изменение ключей.

Код реализации данных вспомогательных методов представлен на рисунке 2.9.

```cpp
template<class T>
void FibMaxHeap<T>::Cut(FibHeapNode<T>* child, FibHeapNode<T>* parent)
{
    if (child->_parent != parent) // Родитель ребёнка не соотвествует переданным данным
        throw HeapException("The child's parent does not match the transmitted data");

    Remove(child); // Убираем связи child с parent

    parent->_degree--;

    if (parent->_child == child) // Проверка на то, является ли узел child первым ребенком узла parent
    {
        parent->_child = child->_left ? child->_left : child->_right;
    }

    Union(child); // Подвешиваем узел child к корневому списку кучи

    child->_parent = nullptr;
    child->_marked = false; // Так как у него не может быть дополнительных отрезаний
}

template<class T>
void FibMaxHeap<T>::CascadingCut(FibHeapNode<T>* node)
{
    FibHeapNode<T>* parent = node->_parent;

    if (parent != nullptr)
    {
        if (node->_marked) // Если узел не был отрезан от его родительского узла
        {
            Cut(node, parent);

            CascadingCut(parent);
        }
        else
        {
            node->_marked = true;
        }
    }
}
```

Рисунок 2.9. Код реализации вспомогательных методов Cut и CascadingCut.

Теперь перейдем к поэтапному описанию работы метода EncreaseData, который используется для изменения приоритета определенного узла в «Фибоначчиевой куче».

1. Выполняется проверка на то, что новое значение данных больше текущего значения данных узла, так как значения данных в узлах «Фибоначчиевой кучи» должны быть монотонно возрастающими.
2. Обновляется значение данных в узле на новое значение.
3. Проверяет, есть ли у узла родитель. Если есть и новое значение данных больше значения данных родителя, выполняется срезание (Cut) узла от его родителя и перемещение отрезанного узла в корневой список кучи.
4. Если после срезания узла его родитель стал отрезанным узлом, выполняется каскадное отрезание (CascadingCut) родителя.
5. Выполняется проверка на то, чтобы новое значение данных было больше значения данных текущего корневого узла, обновляется указатель _root на текущий узел.

Функция EncreaseData позволяет изменить приоритет элемента в «Фибоначчиевой куче» и обеспечивает поддержание корректности и баланса кучи. После изменения приоритета узла, функция проверяет и, при необходимости, выполняет срезание и каскадное отрезание

для обновления структуры кучи. Это гарантирует, что узел с наибольшим приоритетом всегда будет находиться в корневом списке кучи.

Поиск квантиля в «Фибоначчиевой куче».

Квантиль — это значение, которое разделяет упорядоченное множество данных на равные или пропорциональные части.

Опишем работу метода FindQuantile в «Фибоначчиевой куче».

1. Входной параметр quantile представляет собой значение квантиля, которое должно быть в диапазоне (0, 1].
2. Функция начинает обход кучи, начиная с корневого узла. Внутри цикла происходит проверка текущего поддерева, чтобы определить, может ли оно быть полностью включено в квантиль или если искомый квантиль должен находиться внутри этого поддерева.
3. Если размер текущего поддерева позволяет полностью включить его в квантиль, то увеличивается currentSize на размер поддерева, и если currentSize становится равным n, то это означает, что искомый квантиль найден и возвращается текущий узел current.
4. Если размер текущего поддерева превышает n, это означает, что искомый квантиль должен находиться внутри текущего поддерева, и переходим к его дочерней вершине. Продолжается обход кучи до тех пор, пока не будет достигнут корневой узел _root.

Таким образом, метод FindQuantile позволяет найти узел в «Фибоначчиевой куче», соответствующий заданной доле данных, определенной квантилем.

Код реализации поиска квантиля в «Фибоначчиевой куче» представлен на рисунке 2.10.

```
template<class T>
FibHeapNode<T>* FibMaxHeap<T>::FindQuantile(double quantile)
{
    if (quantile ≤ 0.0 || quantile > 1.0) // Значение квантиля должно находится в диапазоне(0, 1]!
        throw InvalidArgument("Quantile value should be in range (0, 1]");

    if (_root == nullptr) // Если куча пустая
        return nullptr;

    int n = static_cast<int>(quantile * _size);  // Количество элементов, которое должно быть в максимальном квантиле
    int currentSize = 0;

    FibHeapNode<T>* current = _root;

    do
    {
        int subtreeSize = current->GetDegree() + 1;  // Размер поддерева с корнем в текущей вершине

        if (currentSize + subtreeSize ≤ n) // Текущее поддерево может быть полностью включено в квантиль
        {
            currentSize += subtreeSize;
            if (currentSize == n)
                return current;

            current = current->GetRight();
        }
        else // Если размер текущего поддерева превышает n, это означает, что искомый квантиль должен находиться внутри текущего поддерева.
        {
            current = current->GetChild();  // Переходим к дочерней вершине
        }

    } while (current ≠ _root); // Обход кучи, начиная с корневого узла.

    return nullptr;
}
```

Рисунок 2.10. Поиск квантиля в «Фибоначчиевой куче».

Демонстрация работы методов «Фибоначчиевой кучи» со сложным классом и вывод ее элементов в порядке убывания приоритета.

```cpp
int main()
{

    FibMaxHeap<Schoolboy> heap_sc;

    Schoolboy S1("Lykov", "Danya", 11, 1, 2004, "Bryansk");
    Schoolboy S2("Lazarev", "Sasha", 11, 1, 2004, "Moscow");
    Schoolboy S3("Malyash", "Yarik", 17, 1, 2004, "Balashikha");
    Schoolboy S4("Pak", "Nastya", 110, 0, 2004, "Korea");
    Schoolboy S5("Kuslieva", "Vika", 21, 0, 2004, "Moscow");

    heap_sc.Push(S1); heap_sc.Push(S2); heap_sc.Push(S3); heap_sc.Push(S5); FibHeapNode<Schoolboy>* node_sc = heap_sc.Push(S4);

    FibMaxHeap<Schoolboy> heap_sc_2;

    Schoolboy S1_("Kitkat", "Danya", 4, 1, 2004, "Bryansk");
    Schoolboy S2_("Twix", "Yarik", 17, 1, 2004, "Bryansk");
    Schoolboy S3_("Nuts", "Sanya", 23, 1, 2004, "Bryansk");
    Schoolboy S4_("MilkyWay", "Senya", 32, 1, 2004, "Penza");

    heap_sc_2.Push(S1_); heap_sc_2.Push(S2_); FibHeapNode<Schoolboy>* node_sc_2 = heap_sc_2.Push(S3_);

    cout << "\nRealization FindMaximum: " << *heap_sc.FindMaximum() << endl;

    cout << "\nRealization FindQuantile: " << *heap_sc.FindQuantile(0.25) << endl;

    cout << "\nRealization EncreaseData: " << endl; heap_sc.EncreaseData(node_sc_2, S4_);

    heap_sc.Join(&heap_sc_2);


    // Вывод по убыванию приоритета-----------------------------------------------------------------------------------------------------------------

    while (!heap_sc.IsEmpty())
        cout << heap_sc.ExtractMax()->GetData() << endl;

    return 0;
}
```

Рисунок 2.11. Создание «Фибоначчиевых куч» из элементов сложного класса Schoolboy, демонстрирование работы всех методов и вывод элементов в порядке убывания ее приоритета.

```
Last Name: Pak
First Name: Nastya
Group: 110
Gender: 0
Date of Birth: 2004
Address: Korea

Last Name: Kuslieva
First Name: Vika
Group: 21
Gender: 0
Date of Birth: 2004
Address: Moscow

Last Name: Malyash
First Name: Yarik
Group: 17
Gender: 1
Date of Birth: 2004
Address: Balashikha

Last Name: Lykov
First Name: Danya
Group: 11
Gender: 1
Date of Birth: 2004
Address: Bryansk

Last Name: Lazarev
First Name: Sasha
Group: 11
Gender: 1
Date of Birth: 2004
Address: Moscow
```

Рисунок 2.12. Вывод элементов по убыванию приоритета с помощью извлечения максимума из корня в «Фибоначчиевой куче» без применения других методов.

```
Realization FindMaximum:
Last Name: Pak
First Name: Nastya
Group: 110
Gender: 0
Date of Birth: 2004
Address: Korea
```

Рисунок 2.13. Результат поиска максимума в «Фибоначчиевой куче» без применения других методов.

```
Realization FindQuantile:
Last Name: Pak
First Name: Nastya
Group: 110
Gender: 0
Date of Birth: 2004
Address: Korea
```

Рисунок 2.14. Результат поиска квантиля (25 %) в «Фибоначчиевой куче».

Рисунок 2.15. Результат объединения двух «Фибоначчиевых куч».

Рисунок 2.16. Результат изменения приоритета некоторого элемента (Lazarev на MilkyWay) у «Фибоначчиевой кучи» до объединения куч.

Заключение

Мной была выполнена реализация такой структуры данных, как «Фибоначчиева куча».

В ходе проделанной мной работы, я сделал вывод, что «Фибоначчиева куча» — это мощная структура данных, которая сочетает в себе преимущества различных типов куч и предлагает эффективные операции для работы с приоритетами и промежуточными значениями.

Одно из главных преимуществ «Фибоначчиевой кучи» заключается в быстроте операций вставки и объединения. Вставка нового элемента в кучу выполняется за амортизированное константное время $O(1)$, что делает ее отличным выбором для динамического добавления элементов. Объединение двух куч также выполняется за константное время $O(1)$, что делает операцию эффективной и удобной.

Кроме того, «Фибоначчиева куча» обеспечивает эффективность операции извлечения максимального элемента. Время выполнения этой операции составляет $O(\log n)$, где n - количество элементов в куче. Это делает ее хорошим выбором для задач, требующих поиска и удаления элементов с наивысшим приоритетом.

Однако следует отметить, что «Фибоначчиева куча» может потреблять больше памяти по сравнению с другими структурами данных. Это связано с использованием дополнительных указателей и хранением деревьев определенной структуры. Несмотря на это, общая производительность операций обычно оправдывает некоторое увеличение использования памяти.

Приложение

```
#include <iostream>
#include <cmath>
#include <string>

using namespace std;

class Schoolboy
{
private:
	string last_name, first_name, address;
	int group, date_of_birth, gender;
public:
	Schoolboy()
	{
		//cout << "\nSchoolboy default constructor";
		group = 0;
		gender = 0;
	}

	Schoolboy(const char* l_n, const char* f_n, int g, int sex, int d_o_b, const char* add)
	{
		//cout << "\nSchoolboy constructor";
		last_name = l_n;
		first_name = f_n;
		group = g;
		gender = sex;
		date_of_birth = d_o_b;
		address = add;
	}

	Schoolboy(const Schoolboy& S)
	{
		//cout << "\nSchoolboy copy constructor";
		last_name = S.last_name;
		first_name = S.first_name;
		group = S.group;
		gender = S.gender;
		date_of_birth = S.date_of_birth;
		address = S.address;
	}

	// переопределение оператора "больше"
	bool operator>(const Schoolboy& other)
	{
		if (group > other.group)
			return true;
		else if (group < other.group)
			return false;
```

```
                if (date_of_birth > other.date_of_birth)
                        return true;
                else if (date_of_birth < other.date_of_birth)
                        return false;

                if (last_name > other.last_name)
                        return true;
                else if (last_name < other.last_name)
                        return false;

                if (first_name > other.first_name)
                        return true;
                else if (first_name < other.first_name)
                        return false;
        }

        // переопределение оператора "меньше"
        bool operator<(const Schoolboy& other)
        {
                if (group < other.group)
                        return true;
                else if (group > other.group)
                        return false;

                if (date_of_birth < other.date_of_birth)
                        return true;
                else if (date_of_birth > other.date_of_birth)
                        return false;

                if (last_name < other.last_name)
                        return true;
                else if (last_name > other.last_name)
                        return false;

                if (first_name < other.first_name)
                        return true;
                else if (first_name > other.first_name)
                        return false;
        }

        // переопределение оператора "равно"
        bool operator==(const Schoolboy& other)
        {
                return last_name == other.last_name && first_name == other.first_name &&
group == other.group && date_of_birth == other.date_of_birth;
        }

        ~Schoolboy() { }

        friend ostream& operator<< (ostream& stream, const Schoolboy& S);
        friend istream& operator>> (istream& stream, Schoolboy& S);
};
```

```
ostream& operator<< (ostream& stream, const Schoolboy& S)
{
	return stream << "\nLast Name: " << S.last_name << "\nFirst Name: " << S.first_name
<<
		"\nGroup: " << S.group << "\nGender: " << S.gender << "\nDate of Birth: " <<
S.date_of_birth <<
		"\nAddress: " << S.address;
}

istream& operator>> (istream& stream, Schoolboy& S)
{
	return stream >> S.last_name >> S.first_name >> S.group >> S.gender >>
S.date_of_birth >> S.address;
}

class HeapException : public exception
{
protected:
	//сообщение об ошибке
	char* str;
public:
	HeapException(const char* s)
	{
		str = new char[strlen(s) + 1];
		strcpy_s(str, strlen(s) + 1, s);
	}
	HeapException(const HeapException& e)
	{
		str = new char[strlen(e.str) + 1];
		strcpy_s(str, strlen(e.str) + 1, e.str);
	}
	~HeapException()
	{
		delete[] str;
	}

	//функцию вывода можно будет переопределить в производных классах, когда
будет ясна конкретика
	virtual void print()
	{
		cout << "HeapException: " << str << "; " << what();
	}
};

class InvalidArgument : public HeapException
{
protected:
	//сообщение об ошибке
	char* str;
public:
	InvalidArgument(const char* s) : HeapException(s) { }
```

```
        InvalidArgument(const InvalidArgument& e) : HeapException(e) { }
        ~InvalidArgument()
        {
                delete[] str;
        }
        virtual void print()
        {
                cout << "InvalidArgument: " << str << "; " << what();
        }
};

template<class T>
class FibMaxHeap;

// Узел фибоначчиевой кучи
template<class T>
class FibHeapNode
{
        // Дружественный класс для кучи для удобства обращения к полям
        friend class FibMaxHeap<T>;

private:
        T _data; // данные

        FibHeapNode<T>* _parent,  // указатель на родительский узел
                * _child,                       // указатель на один из дочерних узлов
                * _left,                        // указатель на левый узел того же предка
                * _right;                       // указатель на правый узел того же предка

        int _degree;    // степень вершины
        bool _marked;// был ли удален в процессе изменения ключа ребенок этой вершины
public:

        FibHeapNode<T>()
        {
                _parent = _child = nullptr; // По умолчанию родителей и детей нет

                _left = _right = this; // С самого начала зациклен сам на себе

                _degree = 0;
                _marked = false;
        }

        FibHeapNode<T>(T data) : FibHeapNode<T>() { _data = data; }

        T GetData() { return _data; }
        FibHeapNode<T>* GetChild() { return _child; }
        FibHeapNode<T>* GetLeft() { return _left; }
        FibHeapNode<T>* GetRight() { return _right; }
        FibHeapNode<T>* GetParent() { return _parent; }
        int GetDegree() { return _degree; }
        bool GetMarked() { return _marked; }
```

```
        void SetData(T data) { _data = data; }
        void SetChild(FibHeapNode<T>* child) { _child = child; }
        void SetLeft(FibHeapNode<T>* left) { _left = left; }
        void SetRight(FibHeapNode<T>* right) { _right = right; }
        void SetParent(FibHeapNode<T>* parent) { _parent = parent; }
        void SetDegree(int deg) { _degree = deg; }
        void SetMarked(bool mark) { _marked = mark; }

        ~FibHeapNode() { };

        template <class T>
        friend ostream& operator<< (ostream& stream, const FibHeapNode<T>& N);
};

template<class T>
ostream& operator<< (ostream& stream, FibHeapNode<T>& N)
{
        stream << N.GetData();
        return stream;
}

template<class T>
class FibMaxHeap
{
protected:
        // Корень кучи. Всегда максимум
        FibHeapNode<T>* _root;

        // Кол-во узлов в куче
        int _size;

        // объединение вершин (Справа: first, слева: second)
        virtual void Union(FibHeapNode<T>* first, FibHeapNode<T>* second);

        // объединение вершин, когда справа корень кучи (Удобно переносить узлы к
корню)
        virtual void Union(FibHeapNode<T>* node) { if (_root != nullptr) { Union(_root, node);
} };

        // Вставить вершину слева от корня
        virtual void Insert(FibHeapNode<T>* node);

        // Удаление всех связей у узла
        virtual void Remove(FibHeapNode<T>* node);

        // Присоеденение дочернего элемента к родительскому
        virtual void Link(FibHeapNode<T>* child, FibHeapNode<T>* parent);

        // Прорежение деревьев
        virtual void Consolidate();
```

```
        // Слияние двух куч
        void Join(FibHeapNode<T>* node);

        // Вырезание вершины (отнимаем у родителей >:P )
        virtual void Cut(FibHeapNode<T>* child, FibHeapNode<T>* parent);

        // Каскадное вырезание (С учётом метки)
        virtual void CascadingCut(FibHeapNode<T>* node);
public:

        FibMaxHeap<T>()
        {
                cout << "\nConstructor in FibMaxHeap is working!\n";
                _root = nullptr;
                _size = 0;
        }

        // Можно и _size == 0
        virtual bool IsEmpty() { return _root == nullptr; }

        // Добавление узла в кучу
        virtual FibHeapNode<T>* Push(T data) { return Add(new FibHeapNode<T>(data)); }

        // Добавление узла в кучу
        virtual FibHeapNode<T>* Add(FibHeapNode<T>* node);

        // Слияние куч
        virtual void Join(FibMaxHeap<T>* heap);

        virtual int GetSize() { return _size; }

        virtual FibHeapNode<T>* GetRoot() { return _root; }

        // В куче максимум всегда в корне
        virtual FibHeapNode<T>* GetMaximum() { return GetRoot(); }

        virtual FibHeapNode<T>* FindMaximum() { return GetMaximum(); }

        // Извлечение максимума
        virtual FibHeapNode<T>* ExtractMax();

        // Увеличение приоритета
        virtual FibHeapNode<T>* EncreaseData(FibHeapNode<T>* node, T data);

        // Поиск узла, который соответствует заданному квантилю
        FibHeapNode<T>* FindQuantile(double quantile);

        ~FibMaxHeap<T>() { cout << "\nDestructor in FibMaxHeap is working!"; }
};

// Изменение приоритета некоторого элемента
template<class T>
```

```
FibHeapNode<T>* FibMaxHeap<T>::EncreaseData(FibHeapNode<T>* node, T data)
{
	if (data < node->_data)
		throw InvalidArgument("The data values in the nodes of the Fibonacci heap are monotonically increasing");

	node->_data = data; // Обновление значения данных в узле

	FibHeapNode<T>* parent = node->_parent;

	if (parent != nullptr && node->_data > parent->_data)
	{
		// Рекурсивно выполняем срезание родительского узла и его перемещение в корневой список кучи.

		Cut(node, parent);

		CascadingCut(parent);
	}

	if (node->_data > _root->_data)
	{
		_root = node;
	}

	return node;
}

template<class T>
FibHeapNode<T>* FibMaxHeap<T>::Add(FibHeapNode<T>* node)
{
	if (_root == nullptr) // Если в куче нет элементов, то только что добавленный максимальный.
		_root = node;
	else // Иначе аккуратно меняем указатели в списке, чтобы не перепутать указатели
	{
		Insert(node);

		if (node->_data > _root->_data) // Передвигаем указатель на новый корень
			_root = node;
	}

	_size++;

	return node;
}

template<class T>
void FibMaxHeap<T>::Insert(FibHeapNode<T>* node)
{
	Remove(node); // Удаляем узел из списка, если он там уже присутствует.
```

```
	FibHeapNode<T>* rootLeft = _root->_left;

	node->_right = _root;
	_root->_left = node;

	if (rootLeft != nullptr)
	{
		node->_left = rootLeft;
		rootLeft->_right = node;
	}
}

template<class T>
void FibMaxHeap<T>::Remove(FibHeapNode<T>* node)
{
	FibHeapNode<T>* left = node->_left;
	FibHeapNode<T>* right = node->_right;

	left->_right = right;
	right->_left = left;

	// Обнуляем ссылки на левую и правую связи, чтобы node не указывал ни на какие
другие узлы.
	node->_left = node;
	node->_right = node;
}

// Связать две вершины

template<class T>
void FibMaxHeap<T>::Union(FibHeapNode<T>* first, FibHeapNode<T>* second) // Справа:
first! , слева: second!
{
	// Аккуратно меняем указатели местами
	FibHeapNode<T>* firstLeft = first->_left, * secondRight = second->_right;

	second->_right = first;
	first->_left = second;

	firstLeft->_right = secondRight;
	secondRight->_left = firstLeft;
}

template<class T>
void FibMaxHeap<T>::Link(FibHeapNode<T>* child, FibHeapNode<T>* parent)
{
	// Убираем связи у дочерней вершины
	Remove(child);

	// Подвешиваем вершину
	child->_parent = parent;
	child->_marked = false;
```

```
	// Даём родителю указатель на дочерний узел, если его нет
	if (parent->_child == nullptr)
	{
		parent->_child = child;
		child->_left = child;
		child->_right = child;
	}
	else
	{
		// Добавляем новых соседей
		FibHeapNode<T>* parentChild = parent->_child;

		child->_left = parentChild->_left;
		child->_right = parentChild;

		parentChild->_left->_right = child;
		parentChild->_left = child;
	}

	parent->_degree++;
}

template<class T>
void FibMaxHeap<T>::Join(FibMaxHeap<T>* heap)
{
	if (heap->_root != nullptr)
		Join(heap->_root);

	_size += heap->_size;
}

template<class T>
void FibMaxHeap<T>::Join(FibHeapNode<T>* node)
{
	if (node == nullptr)
		throw HeapException("We can't connect to an empty heap");

	if (_root == nullptr) // если наша куча пуста, то результатом будет вторая куча
		_root = node;
	else // иначе объединяем два корневых списка
		Union(node);

	if (node->_data > _root->_data) // если максимум кучи изменился, то надо обновить
указатель
		_root = node;
}

template<class T>
FibHeapNode<T>* FibMaxHeap<T>::ExtractMax()
{
	if (_root == nullptr) // Если нечего удалять
```

```
		return nullptr;

	FibHeapNode<T>* prev = _root;

	if (_root->_child != nullptr)
		Union(_root->_child); // Объединяем корневой список с дочерним списком, если у корневого узла есть дочерние узлы

	// Отсоединяем корневой узел от списка корней

	FibHeapNode<T>* left = _root->_left;
	FibHeapNode<T>* right = _root->_right;

	left->_right = right;
	right->_left = left;

	_size--;

	if (prev == prev->_right) // Если в куче остался только один узел
		_root = nullptr;
	else // В противном случае, если в куче остаются еще узлы, устанавливем новый корневой узел и выполняем процедуру консолидации кучи.
	{
		_root = prev->_right;

		Consolidate();
	}

	return prev;
}

// Процесс консолидации, который гарантирует, что в куче не будет двух корневых узлов с одинаковыми степенями!
template<class T>
void FibMaxHeap<T>::Consolidate()
{
	int max_degree = _size; // Память для хранения всех возможных степеней корневых узлов.

	FibHeapNode<T>** array = new FibHeapNode<T>*[max_degree]; // Создаем массив и инициализируем его.

	for (int i = 0; i < max_degree; i++)
		array[i] = nullptr;

	array[_root->_degree] = _root;
	FibHeapNode<T>* current = _root->_right;

	while (array[current->_degree] != current)
	{
		if (array[current->_degree] == nullptr)
		{
```

```
                array[current->_degree] = current;
                current = current->_right;
            }
            else
            {
                FibHeapNode<T>* conflict = array[current->_degree], *addTo, *adding;

                if (conflict->_data > current->_data)
                {
                    addTo = conflict;
                    adding = current;
                }
                else
                {
                    addTo = current;
                    adding = conflict;
                }

                Link(adding, addTo);

                current = addTo;
            }

            if (_root->_data < current->_data)
                _root = current;
        }
    }

    template<class T>
    void FibMaxHeap<T>::Cut(FibHeapNode<T>* child, FibHeapNode<T>* parent)
    {
        if (child->_parent != parent) // Родитель ребёнка не соотвествует переданным данным
            throw HeapException("The child's parent does not match the transmitted data");

        Remove(child); // Убираем связи child с parent

        parent->_degree--;

        if (parent->_child == child) // Проверка на то, является ли узел child первым
    ребенком узла parent
        {
            parent->_child = child->_left ? child->_left : child->_right;
        }

        Union(child); // Подвешиваем узел child к корневому списку кучи

        child->_parent = nullptr;
        child->_marked = false; // Так как у него не может быть дополнительных отрезаний
    }

    template<class T>
    void FibMaxHeap<T>::CascadingCut(FibHeapNode<T>* node)
```

```
{
	FibHeapNode<T>* parent = node->_parent;

	if (parent != nullptr)
	{
		if (node->_marked) // Если узел не был отрезан от его родительского узла
		{
			Cut(node, parent);

			CascadingCut(parent);
		}
		else
		{
			node->_marked = true;
		}
	}
}

template<class T>
FibHeapNode<T>* FibMaxHeap<T>::FindQuantile(double quantile)
{
	if (quantile <= 0.0 || quantile > 1.0) // Значение квантиля должно находится в диапазоне(0, 1]!
		throw InvalidArgument("Quantile value should be in range (0, 1]");

	if (_root == nullptr) // Если куча пустая
		return nullptr;

	int n = static_cast<int>(quantile * _size);  // Количество элементов, которое должно быть в максимальном квантиле
	int currentSize = 0;

	FibHeapNode<T>* current = _root;

	do
	{
		int subtreeSize = current->GetDegree() + 1;  // Размер поддерева с корнем в текущей вершине

		if (currentSize + subtreeSize <= n) // Текущее поддерево может быть полностью включено в квантиль
		{
			currentSize += subtreeSize;
			if (currentSize == n)
				return current;

			current = current->GetRight();
		}
		else // Если размер текущего поддерева превышает n, это означает, что искомый квантиль должен находиться внутри текущего поддерева.
		{
			current = current->GetChild();  // Переходим к дочерней вершине
```

```
                }

        } while (current != _root); // Обход кучи, начиная с корневого узла.

        return nullptr;
}

int main()
{

        FibMaxHeap<Schoolboy> heap_sc;

        Schoolboy S1("Lykov", "Danya", 11, 1, 2004, "Bryansk");
        Schoolboy S2("Lazarev", "Sasha", 11, 1, 2004, "Moscow");
        Schoolboy S3("Malyash", "Yarik", 17, 1, 2004, "Balashikha");
        Schoolboy S4("Pak", "Nastya", 110, 0, 2004, "Korea");
        Schoolboy S5("Kuslieva", "Vika", 21, 0, 2004, "Moscow");

        heap_sc.Push(S1); heap_sc.Push(S4); heap_sc.Push(S3); heap_sc.Push(S5);
FibHeapNode<Schoolboy>* node_sc = heap_sc.Push(S3);

        FibMaxHeap<Schoolboy> heap_sc_2;

        Schoolboy S1_("Kitkat", "Danya", 4, 1, 2004, "Bryansk");
        Schoolboy S2_("Twix", "Yarik", 17, 1, 2004, "Bryansk");
        Schoolboy S3_("Nuts", "Sanya", 23, 1, 2004, "Bryansk");
        Schoolboy S4_("MilkyWay", "Senya", 32, 1, 2004, "Penza");

        heap_sc_2.Push(S1_); heap_sc_2.Push(S2_); FibHeapNode<Schoolboy>* node_sc_2 =
heap_sc_2.Push(S3_);

        cout << "\nRealization FindMaximum: " << *heap_sc.FindMaximum() << endl;

        cout << "\nRealization FindQuantile: " << *heap_sc.FindQuantile(0.25) << endl;

        cout << "\nRealization EncreaseData: " << endl; heap_sc.EncreaseData(node_sc, S4_);

        heap_sc.Join(&heap_sc_2);

        // Вывод по убыванию приоритета----------------------------------------------------------------
----------------------------------------------------------

        while (!heap_sc.IsEmpty())
                cout << heap_sc.ExtractMax()->GetData() << endl;

        return 0;
}
```