МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение высшего образования
«Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского» (ННГУ)

Институт информационных технологий, математики и механики

Кафедра: Кафедра алгебры, геометрии и дискретной математики

Направление подготовки: «Фундаментальная информатика и информационные технологии»
Профиль подготовки: «Инженерия программного обеспечения»

ОТЧЕТ

по учебной практике (научно-исследовательская работа)
на тему:

«Реализация и сравнительный анализ алгоритма Дейкстры на основе различных приоритетных очередей»

Выполнил: студент группы
382006-1
_____ Пикин И. С.
Подпись

Научный руководитель:
д.ф.-м.н., проф. каф. АГДМ ИИТММ

_____ Малышев Д. С.
Подпись

Нижний Новгород
2023

Оглавление

Введение

Задача нахождения кратчайшего пути довольно естественным образом возникает на практике или в реальной жизни человека. Какой путь от дома до работы самый короткий? Каким маршрутом доставить груз из одного города в другой с наименьшими транспортными расходами? Как настроить маршрутизацию пакетов данных в компьютерной сети, чтобы временная задержка была минимальна? Все эти, а также другие вопросы часто можно выразить в терминах задачи о кратчайшем пути.

В теории графов одна из возможных формулировок задачи о кратчайшем пути может звучать следующим образом: во взвешенном графе необходимо найти последовательность рёбер (путь) от одной конкретной вершины к другой или ко всем остальным так, чтобы сумма весов этих рёбер была минимальна. На практике под весами может подразумеваться расстояние, временные интервалы, стоимости, штрафы, убытки или любая другая величина, которая линейно накапливается по мере продвижения вдоль ребер графа и которую нужно свести к минимуму.

Для решения задачи о кратчайшем пути в графе были разработаны различные алгоритмы, имеющие разную временную и пространственную оценку. Некоторые из них перечислены ниже:

- Алгоритм Беллмана-Форда[5][6];
- Алгоритм Дейкстры[7];
- Алгоритм Торупа[8].

В отличие от алгоритма Беллмана-Форда, алгоритм Дейкстры не может работать на графах с отрицательными рёбрами, но на графах с неотрицательными рёбрами во многих случаях он работает быстрее. Алгоритм Торупа имеет хорошую оценку трудоёмкости, однако может работать только на неориентированных графах с целочисленными весами.

Алгоритм Дейкстры был изобретён нидерландским учёным Эдсгером Дейкстрой в 1959 году. Изначально в описании алгоритма не содержалось никаких упоминаний по поводу очереди с приоритетами. В настоящее время различают две возможные реализации алгоритма - на метках и на приоритетной очереди.

В данной работе будет рассмотрен алгоритм Дейкстры и его реализация с использованием различных приоритетных очередей.

Определения

Пусть задан взвешенный ориентированный граф $G = (V, E)$ с весовой функцией $w : E \to \boldsymbol{R}$, отображающей ребра на их веса, значения которых выражаются неотрицательными действительными числами. *Вес пути* $p = <v_0, v_1, \ldots, v_k>$ равен суммарному весу входящих в него рёбер:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Вес кратчайшего пути из вершины u в вершину v определяется соотношением:

$$\delta(u, v) = \begin{cases} min\left\{w(p): u \xrightarrow{p} v\right\}, & \text{если имеется путь от } u \text{ к } v \\ \infty, & \text{в противном случае} \end{cases}$$

Тогда по определению *кратчайший путь* из вершины u в вершину v — это любой путь, вес которого удовлетворяет соотношению $w\ (p) = \delta\ (u, v)$.

Оценка кратчайшего пути $d(v)$ вершины $v, v \in V$ представляет собой верхнюю границу веса, которым обладает кратчайший путь из исходной вершины в вершину v.

Предшественником или *отцом* $\pi(v)$ вершины $v, v \in p$ называется вершина, которая предшествует v в последовательности вершин, составляющих путь p. Для первой вершины в последовательности p предшественника нет.

Под процедурой *релаксации* (ослабления) ребра (u, v) понимается сравнение и выбор минимума среди текущей оценки кратчайшего пути $d(v)$ до вершины v и веса пути до вершины v при прохождении пути через вершину u и ребро (u, v), то есть суммы $d(u) + w(u, v)$. Если значение суммы $d(u) + w(u, v)$ меньше текущего значения $d(v)$, то $d(v)$ обновляется. В противном случае обновление $d(v)$ не происходит. Также при обновлении $d(v)$ происходит обновление предшественника $\pi[v]$ на вершину u.

Постановка задачи

Для взвешенного ориентированного графа $G = (V, E)$ с весовой функцией $w : E \to \boldsymbol{R}$, такой, что для всех ребер $(x, y) \in E$ выполняется неравенство $w(x, y) \geq 0$, требуется найти кратчайшие пути, которые начинаются в определенной исходной вершине $s \in V$ (s - исток) и заканчиваются в каждой из вершин $v \in V$.

Обзор алгоритмов и структур данных

Алгоритм Дейкстры

Описание алгоритма

На начальном этапе происходит инициализация значений $d(v)$ и $\pi(v)$ для всех вершин $v \in V$. По умолчанию для всех всех вершин, кроме истока s, $d(v)$ принимает условное значение ∞, $d(s)$ считается равным нулю. Для всех всех вершин предшественник $\pi(v)$ принимает значение NIL, обозначающее отсутствие предшественника.

Для запоминания вершин, для которых окончательный вес кратчайшего пути уже вычислен, в алгоритме поддерживается множество вершин S. Изначально множество S пустое.

Для хранения вершин, кратчайший путь до которых ещё не найден, используется неубывающая приоритетная очередь Q. Ключём вершины в очереди Q выступает значение $d(v)$. Заметим, что на протяжении всей работы алгоритма верно равенство Q = V – S. Изначально в Q содержатся все вершины.

Основной процесс работы алгоритма происходит в цикле. На каждой итерации цикла из очереди Q извлекается вершина u с минимальным значением $d(v)$ и включается в множество S. Кратчайший путь до вершины u и его вес считаются найденными. Таким образом, на первой итерации цикла из Q извлекается вершина $u = s$ со значениями $d(s) = 0, \pi(s) = NIL$. Кратчайший путь до s $p =$ <s>, $w(p) = 0$. Далее происходит релаксация для всех рёбер (u, v), исходящих из вершины u. Если текущий кратчайший путь к вершине v может быть улучшен в результате прохождения через вершину u, выполняется ослабление и соответствующее обновление оценки величины $d(v)$ и предшественника $\pi(v)$.

Данный цикл заканчивается в тот момент, когда очередь Q становится пустой. Число итераций по завершении цикла равно $|V|$. Для всех вершин найден кратчайший путь и его вес. Алгоритм закончил свою работу.

Псевдокод

```
DIJKSTRA(G, w, s)
1  INITIALIZE_SINGLE_SOURCE(G, s)
2  S ← ∅
3  Q ← V[G]
4  while Q ≠ ∅
5        do u ← EXTRACT_MIN(Q)
6           S ← S ∪ {u}
7           for (для) каждой вершины v ∈ Adj[u]
8                 do RELAX(u, v, w)
```

Процедура DIJKSTRA(G, w, s) реализует работу алгоритма Дейкстры и принимает на вход три параметра: G – сам граф, w - массив весов для рёбер графа G, s – номер исходной вершины.

Вспомогательная процедура INITIALIZE_SINGLE_SOURCE(G,s) инициализирует первоначальные значения $d(v)$ и $\pi(v)$ всех вершин. Время работы - Θ(|V|).

```
INITIALIZE_SINGLE_SOURCE(G, s)
1  for (Для) каждой вершины v ∈ V[G]
2        do d[v] ← ∞
3           π[v] ← NIL
4  d[s] ← 0
```

Множество S изначально пустое, приоритетная очередь Q строится из множества всех вершин графа G по ключу $d(v)$. Внешний цикл while работает, пока очередь Q не пуста, и, таким образом, совершает $|V|$ итераций. После извлечение вершины u с минимальной $d(v)$ из Q и добавления её в множество S во внутреннем цикле for происходит обход окрестности данной верщины. Вспомогательная процедура RELAX(u,v,w), принимающая на вход вершину u, соседнюю ей вершину v и w - вес ребра между ними, выполняет релаксацию ребра (u, v).

```
RELAX(u, v, w)
1  if d[v] > d[u] + w(u, v)
2     then d[v] ← d[u] + w(u, v)
3          π[v] ← u
```

Сложность

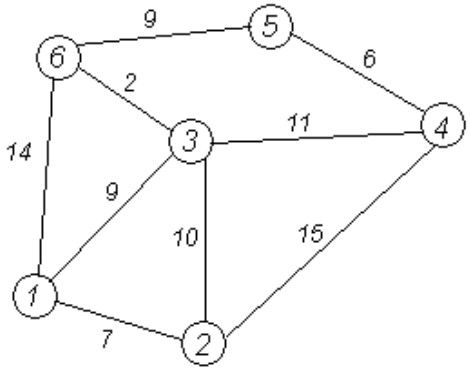
Время выполнения алгоритма Дейкстры зависит от реализации неубывающей очереди с приоритетами. Основными операциями при работе алгоритма с приоритетной очередью являются вставка, извлечение минимума и уменьшение ключа. Вставка и извлечение минимума вызываются по одному разу для каждой вершины. Поскольку каждая вершина $v \in V$ добавляется в множество S ровно один раз, каждое ребро в списке смежных врешин обрабатывается в цикле for ровно один раз за всё время работы алгоритма. Так как полное количество ребер во всех списках смежных вершин равно

$|E|$, всего выполняется $|E|$ итераций этого цикла for, а следовательно, не более $|E|$ операций уменьшения ключа (уменьшение ключа неявно присутствует в процедуре RELAX).

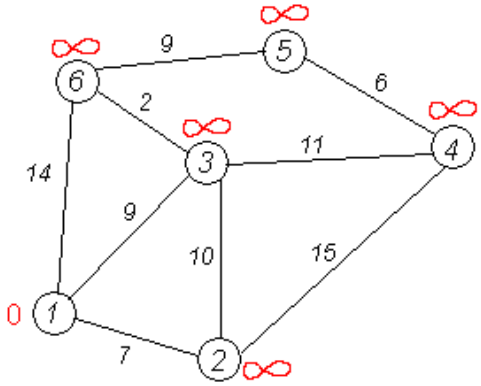
Сравнение итоговой сложности алгоритма при использовании d-кучи и биномиальной кучи будет рассмотрено далее.

Пример

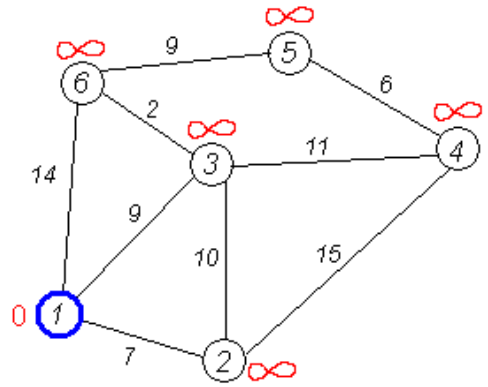
Рассмотрим пример работы алгоритма Дейкстры на графе, изображенном на рисунке ниже. Кружки здесь – вершины с обозначением их номера, линии – рёбра между вершинами с обозначением их веса. Будем полагать, что все рёбра данного графа двунаправлены, а истоком s является вершина с номером 1. Найдём кратчайшие пути от s к остальным вершинам графа.



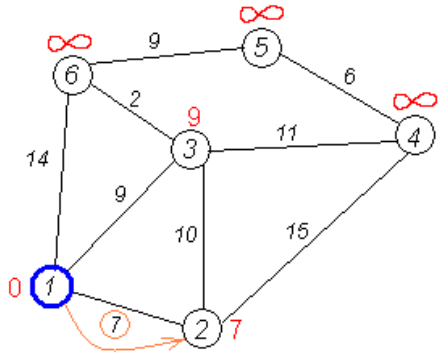
Изначально для всех вершин, кроме истока, длина пути из s условно равна бесконечности. Предшественников нет для всех вершин.

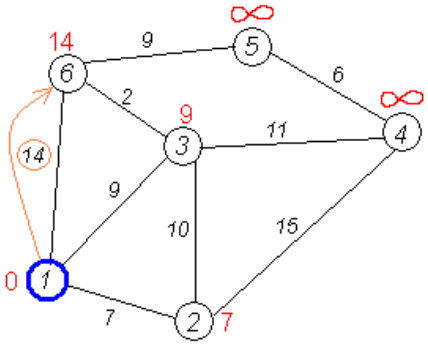


Первый шаг. Минимальное текущее расстояние до вершины 1 имеет сама вершина 1 (расстояние равно нулю). В окрестность вершины 1 входят вершины с номерами 2, 3, 6.

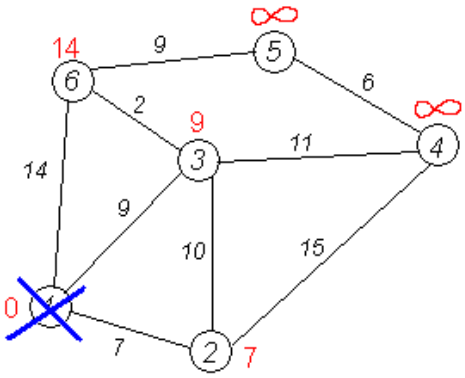


Сделаем обход по вершинам окрестности 1 и проведём релаксацию рёбер. Начнём с вершины 2. Длина пути в вершину 2 через вершину 1 равна сумме значения длины кратчашего пути вершины 1 и длины ребра, идущего из 1-й в 2-ю, то есть 0 + 7 = 7. Текущее значение пути из 1 до вершины 2 равно бесконечности. 7 меньше бесконечности, следовательно, длина пути в вершину 2 теперь будет равна 7. Предшественником вершины 2 становится вершина 1. Таким же образом обновляем значение расстояний и предшественников для вершин 3 и 6.

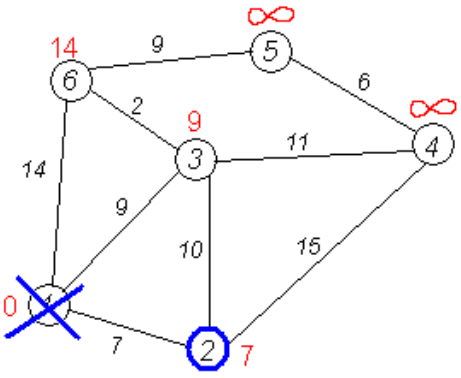




Все вершины окрестности 1 открыты. Расстояние до вершины 1 считается окончательным и равным нулю. Предшественника вершины 1 нет, поэтому кратчайший путь до неё состоит из самой этой вершины. Далее вершина 1 рассматриваться не будет.

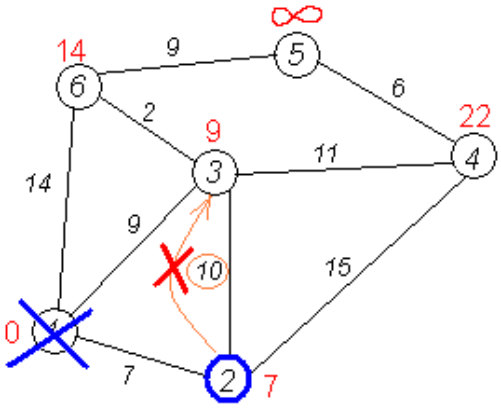


Второй шаг. На данном этапе минимальное текущее расстояние до вершины 1 имеет вершина 2. Это расстояние равно 7. В окрестность вершины 2 входят вершины с номерами 1, 3, 4. Произведём релаксацию рёбер до этих вершин, исходящих из вершины 2.

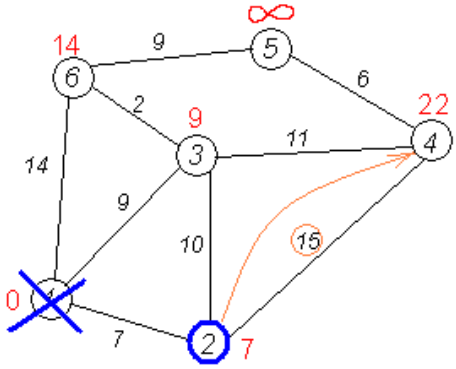


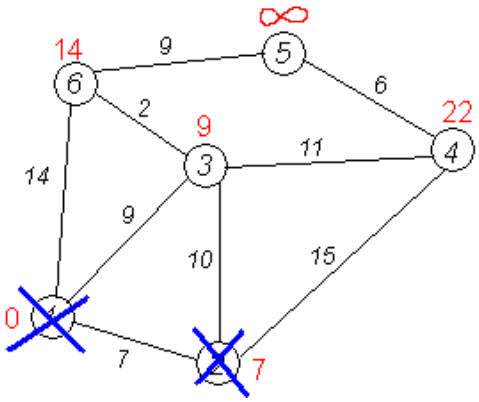
Первая на очереди – вершина 1. Вершина 1 уже посещена, перерасчёт её кратчайшего пути проводиться не будет. Далее идёт вершина 3. Если идти в неё через вершину 2, то длина такого пути

будет равна 7 + 10 = 17. Но текущая оценка кратчайшего пути третьей вершины равна 9, а это меньше 17, поэтому оценка не изменяется, предшественник тоже.

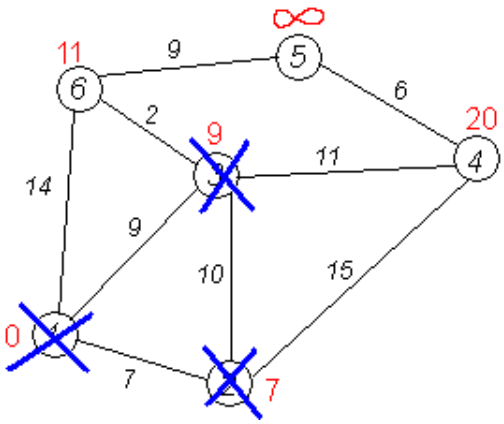


Теперь откроем вершину 4. Длина пути до неё при прохождении через вершину 2 будет равна $d(2) + w(2,4) = 7 + 15 = 22$. Бесконечность меньше, чем 22, следовательно, обновляем оценку кратчайшего пути для вершины 4, её предшественником становится вершина 2. Обход соседей вершины 2 закончен, расстояние до неё её считаем окончательным, далее она рассматриваться не будет.

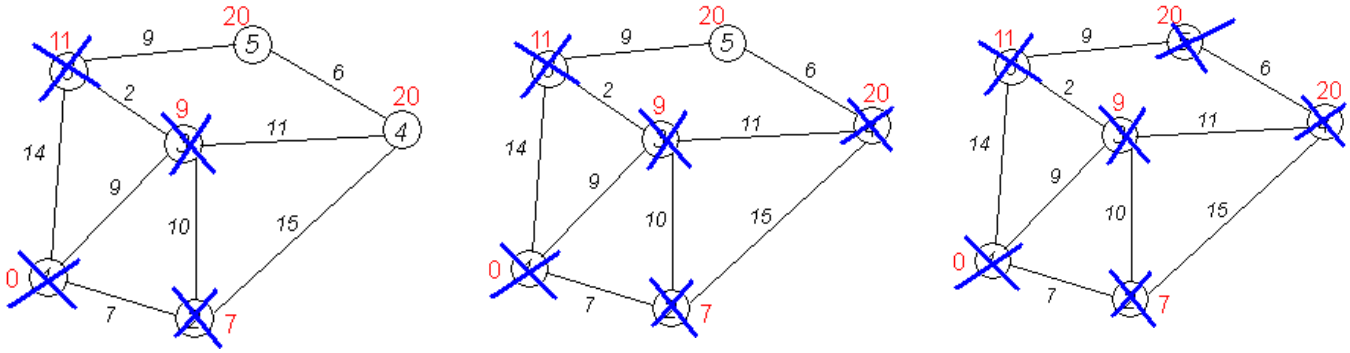




Третий шаг. Проводим те же операции для вершины 3 и её окрестности.



Дальнейшие шаги. Повторяем те же операции для оставшихся вершин.



Завершение работы алгоритма. Когда все вершины посещены, алгоритм завершается. Кратчайшие пути из верщины 1 во все вершины можно выстроить из последовательности предшественников. Веса кратчаших путей получились следующие: для вершины 1 вес пути 0, для 2-ой – 7, для 3-ей – 9, для 4-ой – 20, для 5-ой – 20, для 6-ой – 11. Если граф был несвязный, то для вершин из других компонент связности оценка кратчашего пути до вершины 1 была ба равна бесконечности.

Приоритетные очереди

Приоритетной очередью называется абстрактная структура данных, предназначенная для представления множеств, элементы которых упорядочены в соответствии с их приоритетом (ключом). Различают невозрастающие и неубывающие очереди с приоритетом. Далее будут рассматриваться только неубывающие приоритетные очереди.

Чаще всего приоритетная очередь представляется с помощью корневого дерева или набора корневых деревьев с определенными свойствами. При этом узлам дерева ставятся во взаимно однозначное соответствие элементы рассматриваемого множества. Обычно приоритетные очереди реализуются с помощью *куч*.

Кучей называется представленная деревом приоритетная очередь со свойством кучеобразности.

Под *свойством кучеобразности* понимается следующее: значение ключа любого узла дерева не превосходит значения ключа любого его потомка. Куча с таким свойством в корне будет иметь узел с минимальным ключом (при обратной формулировке свойства в корне будет максимум).

Для того, чтобы эффективно поддерживать операции при работе с кучей, необходимо выбирать деревья с хорошими комбинаторными свойствами.

d-куча

Описание d-кучи

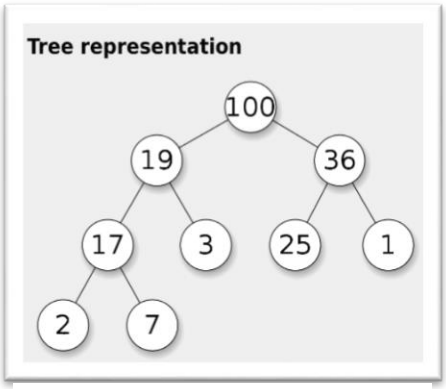
d-кучей называется *d*-дерево со свойством кучеобразности.

d-деревьями ($d \geq 2$) называются деревья со следующими свойствами:

1. Каждый внутренний узел (то есть узел, не являющийся листом дерева), за исключением, быть может, только одного, имеет ровно d потомков. Один узел-исключение может иметь от 1 до d-1 потомков.

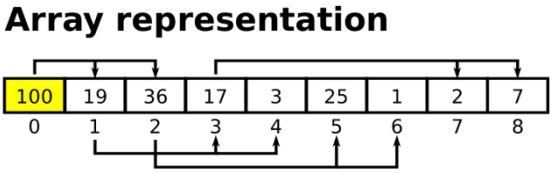
2. Если k − глубина дерева, то для любого i = 1, ..., k-1 такое дерево имеет ровно d^i узлов глубины i.

3. Количество узлов глубины k в дереве глубины k может варьироваться от 1 до d^k. Это свойство является следствием первых двух.



Пример d-кучи с максимумом в корне

При нумерации узлов d-кучи так называемой *змеевидной* нумерацией (то есть в порядке слева-направо и сверху-вниз, начиная от корня) d-кучи удобно представлять массивом.



Вспомогательные процедуры

Процедура нахождения предка узла i. Возвращает -1, если i-ый узел является листом.

```
PARENT(i)
1  if i == 0
2        return −1
3  else return ⌊(i − 1)/d⌋
```

Процедура нахождение левого потомка узла i.

```
LEFT_CHILD(i)
1  j = i * d + 1
2  if j ≥ n
3        return −1
4  else return j
```

Процедура нахождение правого потомка узла i.

```
RIGHT_CHILD(i)
1  j = LEFT_CHILD(i)
2  if j == −1
3        return −1
4  else
5        return MIN(j + d − 1, n − 1)
```

Зная левого и правого потомка узла i процедура MIN_CHILD находит его потомка с минимальным ключом по массиву кучи на его отрезке. Процедуры LEFT_CHILD, RIGHT_CHILD и PARENT вычисляются за $O(1)$, а MIN_CHILD за $O(d)$.

Процедура DIVING выполняет “погружение” узла и используется для восстановления свойств кучеобразности. Если в некотором узле нарушено свойство кучи, то это эквивалентно тому, что ключ этого узла больше, чем минимальный ключ среди всех ключей его потомков.

```
DIVING(i)
1  j1 = i
2  j2 = MIN_CHILD(j1)
3  while j2 ≠ −1 и key[j1] > key[j2]
4        SWAP(j1, j2)
5        j1 = j2
6        j2 = MIN_CHILD(j1)
```

Процедура SWAP меняет местами свои аргументы. Сложность процедуры DIVING – $O(log(n))$, где n – число узлов в d-куче.

Процедура EMERSION (всплытие) симметрична процедуре DIVING.

Операции при работе с d-кучей

1. Поиск узла с минимальным ключом – взятие корневого элемента, сложность $O(1)$.

2. Процедура уменьшения ключа выполняется за $O(\log(n))$.

```
DECREASE_KEY(i, delta)
1  key[i] = key[i] − delta
2  EMERSION(i)
```

3. Вставка нового узла с ключом k, сложность $O(\log(n))$.

```
INSERT(k)
1  n = n + 1
2  key[n − 1] = k
3  EMERSION(n − 1)
```

4. Удаление узла с минимальным ключом, сложность $O(\log(n))$.

```
EXTRACT_MIN()
1  min = key[0]
2  SWAP(0, n − 1)
3  физическое удаление узла n-1
4  n = n − 1
5  DIVING(0)
6  return min
```

5. Удаление заданного элемента, сложность $O(\log(n))$.

```
DELETE(i)
1  DECREASE_KEY(i, +∞)
2  EXTRACT_MIN()
```

6. Процедура “окучивания” приоритетной очереди, сложность $O(n)$.

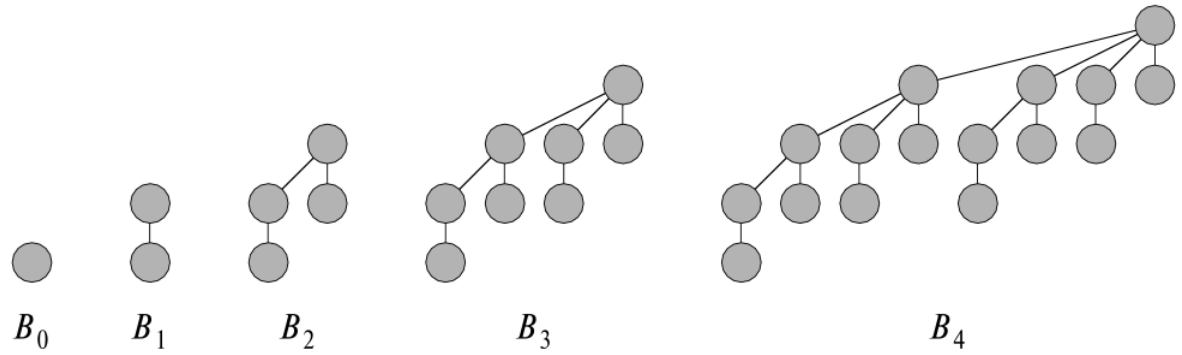
```
MAKE_HEAP(key[])
1  for i = n − 1, i ≥ 0, i = i − 1
2        DIVING(i)
```

Недостатком *d*-куч является невозможность их эффективного слияния. Не существует эффективного алгоритма объединения, использующего структуру уже созданных *d*-куч.

Биномиальная куча

Описание биномиальной кучи

Биномиальным деревом B_k представляет собой рекурсивно определенное упорядоченное дерево, которое для каждого k = 0, 1, 2, … строится следующим образом: B_0 – дерево, состоящее из одного узла высоты 0; далее при k = 1, 2,… дерево B_k высоты k формируется из двух деревьев B_{k-1}, при этом корень одного из них становится потомком корня другого. На рисунке ниже изображены биномиальные деревья B_0, B_1, B_2, B_3, B_4.



Биномиальный лес - это набор биномиальных деревьев, в котором любые два дерева имеют разные высоты.

Свойства биномиального дерева B_k:

1. имеет 2^k узлов;

2. имеет высоту k;

3. имеет ровно $\binom{k}{i}$ узлов на глубине i = 0, 1, ... , k;

4. имеет корень степени k; степень всех остальных вершин меньше степени корня биномиального дерева. Кроме того, если дочерние узлы корня пронумеровать слева направо числами $k-1$, $k-2$, ... , 0 , то i-ый дочерний узел корня является корнем биномиального дерева B_i.

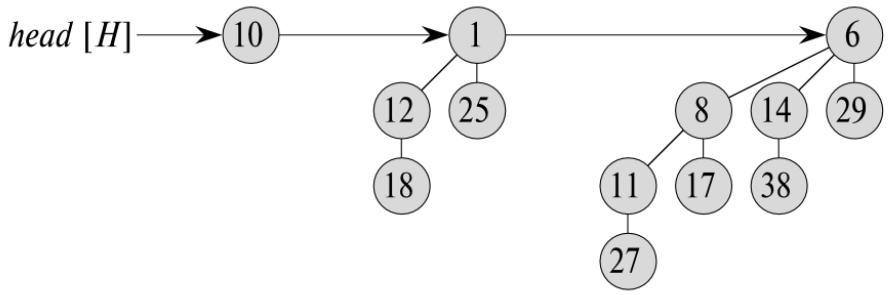
Биномиальная куча H представляет собой множество биномиальных деревьев, которые удовлетворяют следующим *свойствам биномиальных куч*.

1. Каждое биномиальное дерево в H подчиняется свойству неубывающей кучи: ключ узла не меньше ключа его родительского узла.

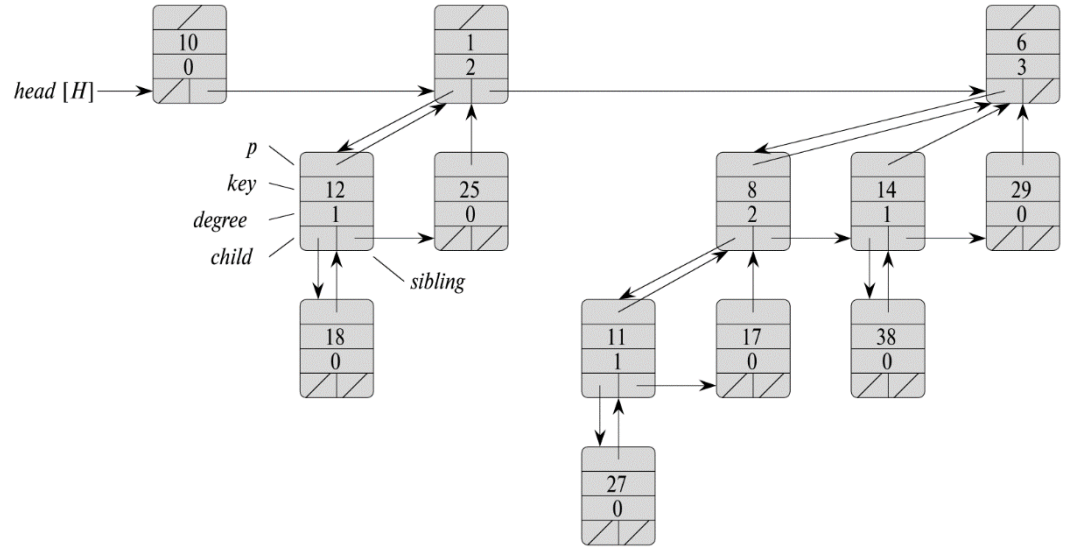
2. Для любого неотрицательного целого k имеется не более одного биномиального дерева H, чей корень имеет степень k.

Представление биномиальной кучи

Корни деревьев в биномиальной куче связываются в односвязный *братский корневой список*.



Каждое биномиальное дерево в биномиальной куче хранится в представлении с левым дочерним и правым братским узлами. Каждый узел имеет поле *key* и содержит сопутствующую информацию, необходимую для работы программы. Кроме того, каждый узел содержит указатель $p[x]$ на родительский узел, указатель $child[x]$ на крайний левый дочерний узел и указатель $sibling[x]$ на правый братский по отношению к x узел. Если x — корневой узел, то $p[x] = NIL$. Если узел x не имеет дочерних узлов, то $child[x] = NIL$, а если x — самый правый дочерний узел своего родителя, то $sibling[x] = NIL$. Каждый узел x, кроме того, содержит поле $degree[x]$, в котором хранится количество дочерних узлов x.



Поле *sibling* имеет различный смысл для корней и для прочих узлов. Если x — корень, то $sibling[x]$ указывает на следующий корень в списке (как обычно, если x — последний корень в списке, то $sibling[x] = NIL$).

Обратиться к данной биномиальной пирамиде H можно при помощи поля $head[H]$, которое представляет собой указатель на первый корень в списке корней H. Если биномиальная пирамида не содержит элементов, то $head[H] = NIL$.

Операции при работе с биномиальной кучей

1. Поиск минимального ключа, сложность $O(log\ n)$, где n – число узлов в куче.

Процедура BINOMIAL_HEAP_MINIMUM проходит по всем корням деревьев братского списка биномиальной кучи H (в каждом корне находится минимум одного дерева), и возвращает указатель на корень с минимальным ключом.

```
BINOMIAL_HEAP_MINIMUM(H)
1  y ← NIL
2  x ← head[H]
3  min ← ∞
4  while x ≠ NIL
5        do if key[x] < min
6              then min ← key[x]
7                   y ← x
8           x ← sibling[x]
9  return y
```

2. Объединение двух биномиальных куч. Сложность процедуры объединения BINOMIAL_HEAP_UNION составляет $O(log\ n)$, где n – общее число узлов в биномиальных кучах H_1 и H_2.

```
BINOMIAL_HEAP_UNION(H1, H2)
1  H ← MAKE_BINOMIAL_HEAP()
2  head[H] ← BINOMIAL_HEAP_MERGE(H1, H2)
3  Освобождение объектов H1 и H2, но не
   списков, на которые они указывают
4  if head[H] = NIL
5    then return H
6  prev-x ← NIL
7  x ← head[H]
8  next-x ← sibling[x]
9  while next-x ≠ NIL
10       do if (degree[x] ≠ degree[next-x]) или
                    (sibling[next-x] ≠ NIL и degree[sibling[next-x]] = degree[x])
11             then prev-x ← x
12                  x ← next-x
13             else if key[x] ⩽ key[next-x]
14                    then sibling[x] ← sibling[next-x]
15                         BINOMIAL_LINK(next-x, x)
16                    else if prev-x = NIL
17                         then head[H] ← next-x
18                         else sibling[prev-x] ← next-x
19                         BINOMIAL_LINK(x, next-x)
20                         x ← next-x
21       next-x ← sibling[x]
22 return H
```

Во время работы основной процедуры BINOMIAL_HEAP_UNION используется три дополнительные процедуры, которые перечислены ниже.

Вспомогательная процедура MAKE_BINOMIAL_ HEAP просто выделяет память и возвращает объект H, где $head[H] = NIL$. Время работы этой процедуры составляет $\Theta(1)$.

Вспомогательная процедура BINOMIAL_HEAP_MERGE объединяет списки корней куч H_1 и H_2 в единый связанный список, отсортированный по степеням в монотонно возрастающем порядке. Сложность BINOMIAL_HEAP_MERGE составляет $O(m)$.

Вспомогательная процедура BINOMIAL_LINK за $O(1)$ связывает дерево B_{k-1} с корнем y с деревом B_{k-1} с корнем z, делая z родительским узлом для y, после чего узел z становится корнем дерева B_k.

```
BINOMIAL_LINK(y, z)
1  p[y] ← z
2  sibling[y] ← child[z]
3  child[z] ← y
4  degree[z] ← degree[z] + 1
```

Принцип работы основной процедуры объединения BINOMIAL_HEAP_UNION состоит в следующем. На первом этапе осуществляется вызовом процедуры BINOMIAL_HEAP_MERGE, которая объединяет списки корней биномиальных пирамид H_1 и H_2 в единый связанный список H, который отсортирован по степеням корней в монотонно возрастающем порядке. В этом списке может оказаться по два (но не более) корня каждой степени, поэтому на втором этапе происходит связывание корней одинаковой степени до тех пор, пока все корни не будут иметь разную степень. Поскольку связанный список H отсортирован по степеням, все операции по связыванию выполняются достаточно быстро.

3. Вставка нового узла x.

Процедура просто создает биномиальную пирамиду H' с одним узлом за время $O(1)$ и объединяет её с биномиальной пирамидой H, содержащей n узлов, за время $O(log\ n)$.

```
BINOMIAL_HEAP_INSERT(H, x)
1  H' ← MAKE_BINOMIAL_HEAP()
2  p[x] ← NIL
3  child[x] ← NIL
4  sibling[x] ← NIL
5  degree[x] ← 0
6  head[H'] ← x
7  H ← BINOMIAL_HEAP_UNION(H, H')
```

4. Извлечение узла с минимальным ключом, сложность $O(log\ n)$.

```
BINOMIAL_HEAP_EXTRACT_MIN(H)
1  Поиск корня x с минимальным значением ключа в списке корней H,
       и удаление x из списка корней H
2  H' ← MAKE_BINOMIAL_HEAP()
3  Обращение порядка связанного списка дочерних узлов x,
       установка поля p каждого дочернего узла равным NIL
       и присвоение указателю head[H'] адреса заголовка
       получающегося списка
4  H ← BINOMIAL_HEAP_UNION(H, H')
5  return x
```

5. Уменьшение ключа, сложность $O(log\ n)$.

   Процедура BINOMIAL_HEAP_DECREASE_KEY изменяет ключ узла *x* на значение *k*, которое меньше исходного.

```
BINOMIAL_HEAP_DECREASE_KEY(H, x, k)
1  if k > key[x]
2     then error "Новый ключ больше текущего"
3  key[x] ← k
4  y ← x
5  z ← p[y]
6  while z ≠ NIL и key[y] < key[z]
7        do Обменять key[y] ↔ key[z]
```

6. Удаление заданного элемента, сложность $O(log\ n)$.

```
BINOMIAL_HEAP_DELETE(H, x)
1  BINOMIAL_HEAP_DECREASE_KEY(H, x, −∞)
2  BINOMIAL_HEAP_EXTRACT_MIN(H)
```

Фибоначчиева куча

Описание фибоначчиевой кучи

Фибоначчиева куча представляет собой набор из подвешенных деревьев удовлетворяющих свойству кучи: ключ каждого предка не больше каждого из ключей своих дочерних узлов (если дерево на минимум). Это означает, что узлом с минимальным ключом во всей куче является один из корней этих деревьев.

Фибоначчиева куча по структуре и поддерживаемым операциям близка к биномиальной куче. Если над фибоначчиевой кучей не выполняются операции уменьшения ключа и удаление, то каждое дерево в куче выглядит как биномиальное. Однако в отличие от биномиальной фибоначчиева куча имеет более гибкую структуру, так как на деревья не наложены никакие ограничения по форме. Например, фибоначчиева куча может состоять из одних деревьев, в каждом из которых по одному элементу. Эта особенность позволяет выполнять некоторые операции лениво, оставляя работу более поздним операциям. Таким образом достигается хорошее амортизированное время работы операций над кучей.

Представление фибоначчиевой кучи

Корни всех деревьев в фибоначчиевой куче связаны при помощи указателей *left* и *right* в циклический дважды связанный ***список корней*** (root list) фибоначчиевой кучи. Порядок деревьев в списке корней неупорядоченный.

Обращение к данной фибоначчиевой куче H выполняется посредством указателя $min[H]$ на корень дерева с минимальным ключом. Этот узел называется ***минимальным узлом*** (minimum node) фибоначчиевой кучи. Если фибоначчиева куча H пуста, то $min\ [H] = NIL$.

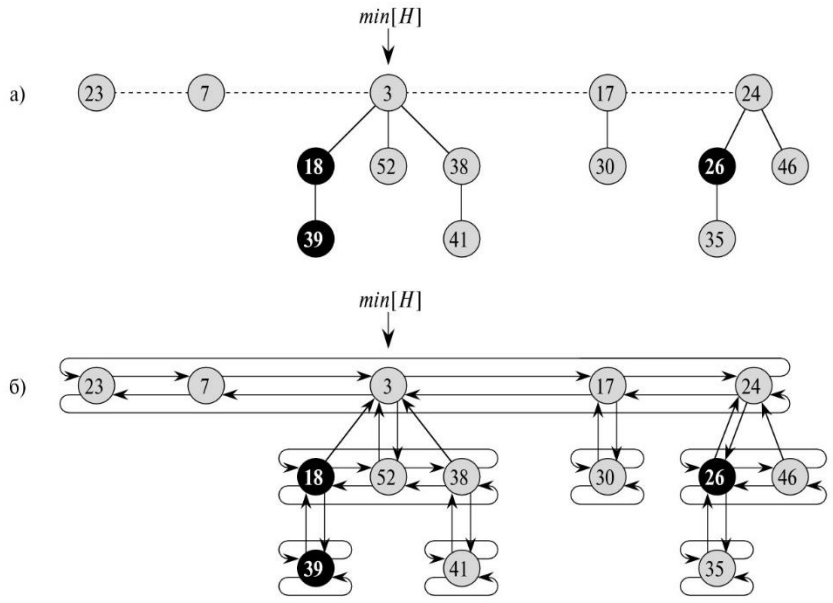


Рисунок 1Пример фибоначчиевой кучи с пятью деревьями и 14 узлами

Каждый узел x в куче содержит указатель $p[x]$ на родительский узел и указатель $child[x]$ на один из дочерних узлов. Дочерние узлы x объединены в один циклический дважды связанный список, называемый ***списком дочерних узлов*** (child list) x. Каждый дочерний узел y в списке дочерних узлов имеет указатели $left[y]$ и $right[y]$, которые указывают на его левый и правый сестринские узлы соответственно. Если узел y является единственным дочерним узлом, то $left[y] = right[y] = y$. Порядок размещения узлов в списке дочерних узлов произволен.

Циклический дважды связанный список обладает двумя преимуществами для использования в фибоначчиевых кучах. Во-первых, удаление элемента из такого списка выполняется за время $O(1)$. Во-вторых, если имеется два таких списка, их легко объединить в один за то же время $O(1)$.

Помимо этого, будут использоваться два других поля каждого узла. Количество дочерних узлов x хранится в поле $degree[x]$, а логическое значение $mark[x]$ указывает, были ли потери узлом x дочерних узлов начиная с момента, когда x стал дочерним узлом какого-то другого узла. Вновь создаваемые узлы не помечены, а имеющаяся пометка снимается, если узел становится дочерним узлом какого-то другого узла.

Фибоначчиева куча, кроме того, имеет еще один атрибут — текущее количество узлов в фибоначчиевой куче H хранится в $n[H]$.

Потенциальная функция

Для оценки времени работы операций над фибоначчиевой кучей применяется один из методов амортизационного анализа — метод потенциалов (подробнее о методе см. в разделе 17.3 из [1]).

Для данной фибоначчиевой кучи *H* обозначим через *t(H)* количество деревьев в списке корней *H*, а через *m(H)* — количество помеченных узлов в *H*. Тогда потенциал фибоначчиевой кучи *H* определяется как

$$\Phi(H) = t(H) + 2m(H).$$

Потенциал множества фибоначчиевых куч представляет собой сумму потенциалов составляющих его куч. Будем считать, что единицы потенциала достаточно для оплаты константного количества работы, где константа достаточно велика для покрытия стоимости любой операции со временем работы *O(1)*.

Также предполагается, что алгоритм начинает свою работу, не имея ни одной фибоначчиевой кучи, так что начальный потенциал равен 0 и, в соответствии с формулой, во все последующие моменты времени потенциал неотрицателен. Верхняя граница общей амортизированной стоимости является верхней границей общей фактической стоимости последовательности операций.

Операции при работе с фибоначчиевой кучей

1. Создание новой фибоначчиевой кучи, амортизированная стоимость - *O(1)*.

Для создания пустой фибоначчиевой кучи процедура MAKE_FIB_HEAP выделяет память и возвращает объект фибоначчиевой кучи *H*, причем и *min[H]=NIL*. Деревьев в *H* нет. Поскольку и *m(H)=0*, потенциал пустой фибоначчиевой кучи *Ф(H)=0*. Таким образом, амортизированная стоимость процедуры MAKE_FIB_HEAP равна ее фактической стоимости *O(1)*.

2. Вставка узла в кучу, амортизированная стоимость - *O(1)*.

Приведенная далее процедура вставляет узел *x* в фибоначчиеву кучу *H* в предположении, что узлу уже выделена память и поле узла *key[x]* уже заполнено.

```
FIB_HEAP_INSERT(H, x)
 1  degree[x] ← 0
 2  p[x] ← NIL
 3  child[x] ← NIL
 4  left[x] ← x
 5  right[x] ← x
 6  mark[x] ← FALSE
 7  Присоединение списка корней, содержащего x, к списку корней H
 8  if min[H] = NIL или key[x] < key[min[H]]
 9      then min[H] ← x
10  n[H] ← n[H] + 1
```

3. Поиск минимального узла, стоимость - *O(1)*.

На минимальный узел фибоначчиевой кучи *H* указывает указатель *min[H]*, так что поиск минимального узла занимает время *O(1)*.

4. Объединение двух фибоначчиевых пирамид, стоимость – *O(1).*

Приведенная далее процедура объединяет фибоначчиевы кучи H_1 и H_2, попросту соединяя списки корней H_1 и H_2 и находя затем новый минимальный узел:

```
FIB_HEAP_UNION(H1, H2)
1  H ← MAKE_FIB_HEAP()
2  min[H] ← min[H1]
3  Добавление списка корней H2 к списку корней H
4  if (min[H1] = NIL) или (min[H2] ≠ NIL и key[min[H2]] < key[min[H1]])
5     then min[H] ← min[H2]
6  n[H] ← n[H1] + n[H2]
7  Освобождение объектов H1 и H2
8  return H
```

5. Извлечение минимального узла, амортизированная стоимость – *O(log(n)).*

Процесс извлечения минимального узла наиболее сложный из всех операций, рассматриваемых в данном разделе. Это также то место, где выполняются отложенные действия по объединению деревьев. Псевдокод процедуры извлечения минимального узла приведен ниже.

```
FIB_HEAP_EXTRACT_MIN(H)
 1  z ← min[H]
 2  if z ≠ NIL
 3     then for (для) каждого дочернего по отношению к z узла x
 4             do Добавить x в список корней H
 5                p[x] ← NIL
 6          Удалить z из списка корней H
 7          if z = right[z]
 8            then min[H] ← NIL
 9            else min[H] ← right[z]
10                 CONSOLIDATE(H)
11          n[H] ← n[H] − 1
12  return z
```

Отдельный этап этой процедуры, на котором будет уменьшено количество деревьев в фибоначчиевой куче, — ***уплотнение*** (consolidating) списка корней *H*, которое выполняется вспомогательной процедурой CONSOLIDATE(H). Уплотнение списка корней состоит в многократном выполнении следующих шагов, до тех пор, пока все корни в списке корней не будут иметь различные значения поля *degree*.

Процедура CONSOLIDATE использует вспомогательный массив *A[0..D(n[H])]*, где *D(n)* – это верхняя граница максимальной степени узла в фибоначчиевой куче из *n* узлов. Если *A[i]=y*, то *y* в настоящий момент является корнем со степенью *degree[y]= i*.

```
CONSOLIDATE(H)
 1  for i ← 0 to D(n[H])
 2       do A[i] ← NIL
 3  for (для) каждого узла w в списке корней H
 4       do x ← w
 5          d ← degree[x]
 6          while A[d] ≠ NIL
 7               do y ← A[d]          ▷ Узел с той же степенью, что и у x.
 8                  if key[x] > key[y]
 9                    then обменять x ↔ y
10                  FIB_HEAP_LINK(H, y, x)
11                  A[d] ← NIL
12                  d ← d + 1
13          A[d] ← x
14  min[H] ← NIL
15  for i ← 0 to D(n[H])
16       do if A[i] ≠ NIL
17             then Добавить A[i] в список корней H
18                  if min[H] = NIL или key[A[i]] < key[min[H]]
19                    then min[H] ← A[i]
```

Вспомогательная процедура FIB_HEAP_LINK выполняет ***привязывание*** (link) *y* к *x*: удаляет *y* из списка корней и делает его дочерним узлом *x*.

6. Уменьшение ключа, амортизированное время - *O(1)*.

Псевдокод процедуры, производящей уменьшение ключа узла, FIB_HEAP_DECREASE_KEY приведён ниже:

```
FIB_HEAP_DECREASE_KEY(H, x, k)
1  if k > key[x]
2     then error "Новый ключ больше текущего"
3  key[x] ← k
4  y ← p[x]
5  if y ≠ NIL и key[x] < key[y]
6     then CUT(H, x, y)
7          CASCADING-CUT(H, y)
8  if key[x] < key[min[H]]
9     then min[H] ← x
```

В том случае, когда свойство неубывающих куч оказывается нарушенным, требуется внести множество изменений. Первое – это операция ***вырезания*** (cutting) *x* в строке 6. Процедура CUT "вырезает" связь между *x* и его родительским узлом *y*, делая *x* корнем.

```
CUT(H, x, y)
1  Удаление x из списка дочерних узлов y, уменьшение degree[y]
2  Добавление x в список корней H
3  p[x] ← NIL
4  mark[x] ← FALSE
```

Поле *mark* используется для получения желаемого времени работы. В нем хранится маленькая часть истории каждого узла. Предположим, что с узлом *x* произошли следующие события.

1) В некоторый момент времени *x* был корнем,
2) затем *x* был привязан к другому узлу,
3) после чего два дочерних узла *x* были вырезаны.

Как только *x* теряет второй дочерний узел, происходит вырезание *x* у его родителя и *x* становится новым корнем. Поле *mark[x]* равно TRUE, если произошли события 1 и 2 и у *x* вырезан только один дочерний узел. Процедура CUT, следовательно, в строке 4 должна очистить поле *mark[x]*, поскольку произошло событие 1. (Поэтому в строке 3 процедуры FIB_HEAP_LINK выполняется сброс поля *mark[y]*: узел *y* оказывается связан с другим узлом, т.е. выполняется событие 2. Затем, когда будет вырезаться дочерний узел у узла *y*, полю *mark[y]* будет присвоено значение TRUE.) Но это ещё не всё, поскольку *x* может быть вторым дочерним узлом, вырезанным у его родительского узла *y* с того момента, когда *y* был привязан к другому узлу. Поэтому в строке 7 процедуры FIB_HEAP_DECREASE_KEY делается попытка выполнить операцию ***каскадного вырезания*** (cascading-cut) над *y*. Если *y* — корень, то проверка в строке 2 процедуры CASCADING_CUT заставляет ее прекратить работу. Если *y* не помечен, процедура помечает его в строке 4, поскольку его первый дочерний узел был только что вырезан, после чего также прекращает работу. Однако если узел уже был помечен, значит, только что он потерял второй дочерний узел. Тогда *y* вырезается в строке 5, и процедура CASCADING_CUT в строке 6 рекурсивно вызывает себя для узла *z*, родительского по отношению к узлу *y*. Процедура CASCADING_CUT рекурсивно поднимается вверх по дереву до тех пор, пока не достигает корня или непомеченного узла.

7. Удаление узла, амортизированная стоимость - $O(D(n))=O(\log(n))$, где $D(n)$ – это верхняя граница максимальной степени узла в фибоначчиевой куче из *n* узлов.

FIB_HEAP_DELETE(H, x)
1 FIB_HEAP_DECREASE_KEY$(H, x, -\infty)$
2 FIB_HEAP_EXTRACT_MIN(H)

Анализ и сравнение сложности реализаций алгоритма Дейкстры

Рассмотрев описание и основные операции при работе с биномиальными, фибоначчиевыми и d-кучами, теперь перейдём к сравнению итоговой сложности алгоритма Дейкстры при использовании этих вариантов приоритетной очереди.

В частности, при использовании d-кучи оценка сложности алгоритма Дейкстры складывается следующим образом. Операция извлечения минимума оценивается $O(log\ |V|\ )$, всего таких операций $|V|$. Сложность окучивания массива вершин имеет оценку $O(|V|)$. Операция уменьшения ключа оценивается $O(log\ |V|)$ и всего выполняется не более $|E|$ раз. Отсюда следует, что полная сложность алгоритма равна $O\ \big((|V|+\ |E|)log\ |V|\big)$.

В случае с биномиальной кучей для создания кучи из множества элементов с приоритетом потребуется $|V|$ вставок в пустую кучу со сложностью $O(log\ |V|)$. Так же, как и с d-куче, всего будет $|V|$ операций извлечения минимума, каждая из которых оценивается $O(log\ |V|\ )$. Уменьшение ключа будет выполнено не более $|E|$ раз с оценкой сложности $O(log\ |V|)$. Таким образом, итоговая сложность алгоритма $O(2|V|log|V|+|E|log|V|) =\ O\ \big((|V|+\ |E|)log\ |V|\big)$.

Аммортизированная оценка сложности для алгоритма на основе фибоначиевой кучи: требуется $|V|$ вставок в пустую кучу со сложностью $O(1)$; производится $|V|$ операций извлечения минимума с оценкой времени $O(log\ |V|\ )$; уменьшение ключа происходит не более $|E|$ раз с оценкой сложности $O(1)$. Итоговая аммортизированная оценка сложности алгоритма равна $O(|V|+|V|log|V|+|E|) =\ O\ (|V|log\ |V|+|E|)$.

Операция	**d-куча (фактическая сложность)**	**Биномиальная куча (фактическая сложность)**	**Фибоначчиева куча (амортизированная сложность)**
Создание пустой кучи	*O(1)*	*O(1)*	*O(1)*
Вставка	*O(log(n))*	*O(log(n))*	*O(1)*
Взятие минимума	*O(1)*	*O(log(n))*	*O(1)*
Слияние	—	*O(log(n))*	*O(1)*
«Окучивание»	*O(n)*	—	—
Извлечение минимума	*O(log(n))*	*O(log(n))*	*O(log(n))*
Уменьшение ключа	*O(log(n))*	*O(log(n))*	*O(1)*
Удаление	*O(log(n))*	*O(log(n))*	*O(log(n))*

С теоретической точки зрения, фибоначчиева куча имеет лучшую оценку времени работы, так как все операции работают за время O(1), кроме удаления и извлечения минимума. Благодаря этому

можно облегчить релаксацию рёбер в алгоритме Дейкстры до O(1) за счет увеличения времени извлечения минимума до O(log(n)). Однако с практической точки зрения программная сложность реализации и высокие значения постоянных множителей в формулах времени работы существенно снижают эффективность применения фибоначчиевых пирамид, делая их для большинства приложений менее привлекательными, чем обычные d-кучи.

Заключение

В данной работе был рассмотрен алгоритм Дейкстры, решающий задачу нахождения кратчайших путей в графе. Были описаны и реализованы варианты данного алгоритма на различных приоритетных очередях, а также даны оценки времени работы и проведён сравнительный анализ.

Список литературы

1) Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн Алгоритмы: построение и анализ — 2-е изд. — М.: «Вильямс», 2007. — с. 459. — ISBN 5-8489-0857-4.
2) Алексеев В.Е., Таланов В.А. Графы. Модели вычислений. Структуры данных: Учебник. – Нижний Новгород: Изд-во ННГУ, 2005. 307 с. ISBN 5–85747–810–8.
3) [Электронный ресурс]: Википедия. Свободная энциклопедия, статья “Алгоритм Дейкстры” - URL: https://ru.wikipedia.org/wiki/Алгоритм_Дейкстры (дата обращения: 12.11.2022).
4) [Электронный ресурс]: Википедия. Свободная энциклопедия, статья “Куча (структура данных)” - URL: https://ru.wikipedia.org/wiki/Куча_(структура_данных) (дата обращения: 12.11.2022).
5) Richard Bellman. On a Routing Problem. Quarterly of Applied Mathematics, 16(1):87–90, 1958.
6) Lestor R. Ford, Jr., and D. R. Fulkerson. Flows in Networks. Princeton University Press, 1962.
7) E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. Numerische Mathematik, 1:269–271, 1959.
8) Mikkel Thorup. Undirected Single-Source Shortest Paths with Positive Integer Weights in Linear Time. Journal of the ACM, 46(3):362–394, 1999.
9) [Электронный ресурс]: Викиконспекты университета ИТМО, статья “Фибоначчиева куча” - URL: https://neerc.ifmo.ru/wiki/index.php?title=Фибоначчиева_куча (дата обращения: 23.12.2023).
10) [Электронный ресурс]: Алгоритмика, статья “Алгоритм Дейкстры” - URL: https://ru.algorithmica.org/cs/shortest-paths/dijkstra/ (дата обращения: 23.12.2023).