

ИВАНОВ А.Б.

АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

ХЕШ-ТАБЛИЦЫ. ГРАФЫ.

TRIE (ПРЕФИКСНОЕ ДЕРЕВО)

Структура придумана Rene de la Briandais в 1959.

TRIE = центральные буквы слово reTRIEval. Термин придуман Edward Fredkin в 1960, обыгрывая схожесть со словом tree. Автор термина произносил [tri:], но общепринято произношение [traɪ], чтобы как раз не путать на слух.

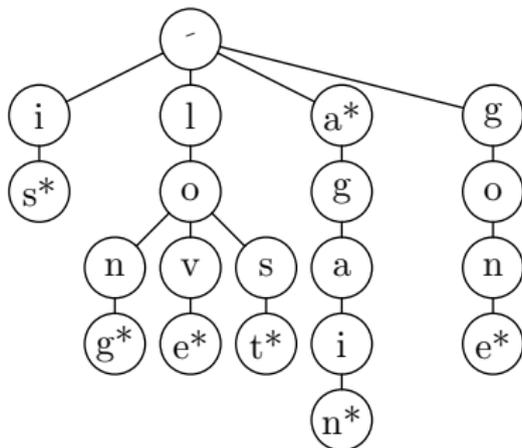
В русских переводах встречаются варианты

- БОР = выБОРка (Кнут, 3й том, 1 издание)
- ЛУЧ = поЛУЧение (Кнут, 3й том, 2 издание)
- нагруженное дерево (Ахо Ульман)

TRIE (ПРЕФИКСНОЕ ДЕРЕВО)

Структура похожа на дерево поиска с парой отличий:

- в узлах храним буквы, а не строки;
- количество детей может быть до R (количество букв в алфавите).



ОТОБРАЖЕННОЕ ДЕРЕВО ХЕШ-МАССИВА (НАМТ)

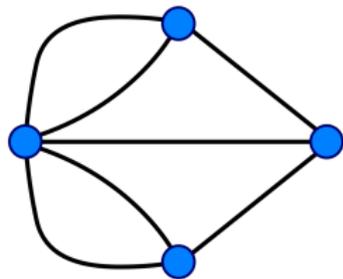
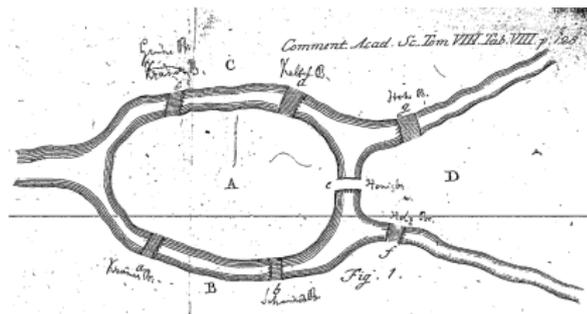
Структура придумана Phil Bagwell в 2000.

В определенном смысле гибрид хеш-таблицы и префиксного дерева:

- Считаем 64-битную хеш-функцию
- Рассматриваем значение хеш-функции как слово из 6-битных букв
- Строим префиксное дерево

Задача о Кёнигсбергских мостах

Эйлер (1736)



ОСНОВНЫЕ ОПРЕДЕЛЕНИЯ

Граф (V, E)

V – множество вершин

E – множество ребер

Ориентированный:

$$E = \{(u, v) \mid u, v \in V\}$$

Неориентированный:

$$E = \{(u, v) \mid u, v \in V\} / \sim \quad (u, v) \sim (v, u)$$

Степень вершины

Ориентированный:

$$d_{out}(v), d_{in}(v)$$

$$\sum_{v \in V} d_{out}(v) = \sum_{v \in V} d_{in}(v) = |E|$$

Неориентированный:

$$d(v)$$

$$\sum_{v \in V} d(v) = 2|E|$$

ПРИЛОЖЕНИЯ: ГРАФЫ ВЕЗДЕ

- Веб-граф
- Графы социальных сетей
- Компьютерные и электросети
- Дорожная сеть
- Управление проектами
- ...

ПУТИ И СВЯЗНОСТЬ

Путь от вершины u к вершине v : $e_1, e_2, \dots, e_k \in E$

$$e_i = (u_{i-1}, u_i), \quad u_0 = u, \quad u_k = v$$

Цикл: путь, в котором начало и конец совпадает

Связный граф: для любых $u, v \in V$ \exists путь от u к v

Компонента связности: максимальное $C \subset V$ такое, что для любых $u, v \in C$ \exists путь от u к v

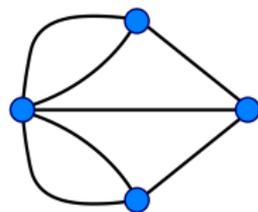
Дерево: связный неориентированный граф без циклов

ЗАДАЧА О КЁНИГСБЕРГСКИХ МОСТАХ

Эйлеров путь: путь без повторяющихся ребер

Существует ли эйлеров путь?

Эйлеров путь существует \Leftrightarrow
в графе не более 2 вершин нечетной степени



Гамильтонов путь: путь без повторяющихся вершин

Существует ли гамильтонов путь?

NP-полная задача

КОГДА ГРАФ МОЖНО УЛОЖИТЬ НА ПЛОСКОСТЬ?

Теорема Куратовского, 1930:

Граф является планарным тогда и только тогда, когда в нем нет топологических миноров, изоморфных K_5 или $K_{3,3}$.

Граф H является топологическим минором G если существует гомеоморфное вложение H в G , т.е. существует подграф H' в G такой, что H' получается из H подразделением ребер.

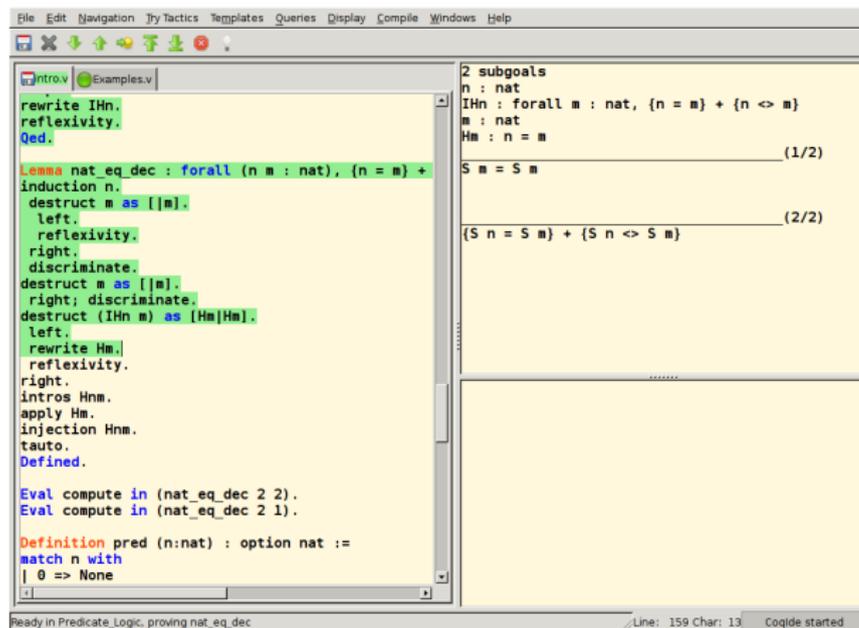
Алгоритм: гамма-алгоритм

ПРОБЛЕМА 4 КРАСОК

Сколько красок необходимо, чтобы раскрасить плоскую карту так, чтобы граничащие страны имели разный цвет?

- 5 красок достаточно: Heawood, 1890
- 4 краски: Appel, Haken, 1976
 - любой контрпример содержит минимальный
 - 1936 карт, которые могли бы быть минимальными контрпримерами
 - сотни страниц доказательства + программа
- более простое доказательство (также с использованием компьютера): Robertson, Sanders, Seymour, Thomas, 1997
- формализованное доказательство в системе Coq: Gonthier, 2005

СИСТЕМА COQ: INTERACTIVE THEOREM PROVER



The screenshot shows the Coq IDE interface. The left pane contains a proof script for the lemma `nat_eq_dec`. The script uses `induction`, `destruct`, `rewrite`, and `apply` tactics. The right pane shows the current goals, which are `S m = S m` and `(S n = S m) + (S n <> S m)`. The status bar at the bottom indicates the current line and character position.

```
File Edit Navigation Try Tactics Templates Queries Display Compile Windows Help
Examples.v
rewrite IHn.
reflexivity.
Qed.

Lemma nat_eq_dec : forall (n m : nat), (n = m) +
induction n.
destruct m as [|m].
left.
reflexivity.
right.
discriminate.
destruct m as [|m].
right; discriminate.
destruct (IHn m) as [Hm|Hm].
left.
rewrite Hm.
reflexivity.
right.
intros Hnm.
apply Hm.
injection Hnm.
tauto.
Defined.

Eval compute in (nat_eq_dec 2 2).
Eval compute in (nat_eq_dec 2 1).

Definition pred (n:nat) : option nat :=
match n with
| 0 => None
```

2 subgoals
n : nat
IHn : forall m : nat, {n = m} + {n <> m}
m : nat
Hm : n = m
S m = S m (1/2)
(S n = S m) + (S n <> S m) (2/2)

Ready in Predicate_Logics.proving_nat_eq_dec Line: 159 Char: 13 Coqide started

Gonthier + team, 2012: теорема Фейта-Томпсона (1962)

Не существует простых абелевых групп нечетного порядка

РАССТОЯНИЕ МЕЖДУ ВЕРШИНАМИ (BFS)

```
procedure BFS( G, s ):
  dist = array of size G.V
  for x in G.V:
    dist[x] = ∞

  Q = queue
  dist[s] = 0
  Q.push(s)

  while not Q.isEmpty:
    x = Q.pop()
    for y in G.neighbours(y):
      if dist[y] = ∞:
        dist[y] = dist[x] + 1
        Q.push(y)
      end
    end
  end
end
```

АЛГОРИТМ ДЕЙКСТРЫ

(Edsger Wybe Dijkstra)

Пусть на ребрах графа заданы длины: $G = (V, E)$, $l : E \rightarrow \mathbb{R}_+$

Задача: найти кратчайшие расстояния до всех вершин из данной вершины s

АЛГОРИТМ ДЕЙКСТРЫ

Очередь с приоритетами

```
insert(element, key)
deleteMin()
decreaseKey(element, key)
makeQueue(elements, keys)
```

```
procedure Dijkstra( G, s ):
    dist = array of size G.V
    prev = array of size G.V

    for x in G.V:
        dist[x] =  $\infty$ 
        prev[x] = -1

    dist[s] = 0
    Q = priority queue
    Q.makeQueue(V, dist)

    while not Q.isEmpty:
        x = Q.deleteMin()
        for y in G.neighbours(x):
            key = dist[x] + l(x, y)
            if key < dist[y]:
                dist[y] = key
                prev[y] = x
                Q.decreaseKey(y, key)
            end
        end
    end
end
```

ОЧЕРЕДЬ С ПРИОРИТЕТАМИ

Операции (интерфейс):

- Добавить элемент (insert)
- Исключить минимальный элемент (deleteMin)

Дополнительные операции (интерфейс):

- Построить очередь (makeQueue / buildHeap)
- Уменьшить значение ключа (decreaseKey)

РЕАЛИЗАЦИИ ОЧЕРЕДИ С ПРИОРИТЕТАМИ

1. Неупорядоченный массив.

РЕАЛИЗАЦИИ ОЧЕРЕДИ С ПРИОРИТЕТАМИ

1. Неупорядоченный массив.

`insert`: $O(1)$ - просто добавляем в конец

`deleteMin`: $O(n)$ - надо перебрать все элементы

РЕАЛИЗАЦИИ ОЧЕРЕДИ С ПРИОРИТЕТАМИ

1. Неупорядоченный массив.

`insert`: $O(1)$ - просто добавляем в конец

`deleteMin`: $O(n)$ - надо перебрать все элементы

2. Упорядоченный массив

РЕАЛИЗАЦИИ ОЧЕРЕДИ С ПРИОРИТЕТАМИ

1. Неупорядоченный массив.

`insert`: $O(1)$ - просто добавляем в конец

`deleteMin`: $O(n)$ - надо перебрать все элементы

2. Упорядоченный массив (по убыванию).

`insert`: $O(n)$ - приходится сдвигать кусок массива

`deleteMin`: $O(1)$ - просто берем последний элемент

РЕАЛИЗАЦИИ ОЧЕРЕДИ С ПРИОРИТЕТАМИ

1. Неупорядоченный массив.

insert: $O(1)$ - просто добавляем в конец

deleteMin: $O(n)$ - надо перебрать все элементы

2. Упорядоченный массив (по убыванию).

insert: $O(n)$ - приходится сдвигать кусок массива

deleteMin: $O(1)$ - просто берем последний элемент

3. Двоичная куча (Heap, Binary heap).

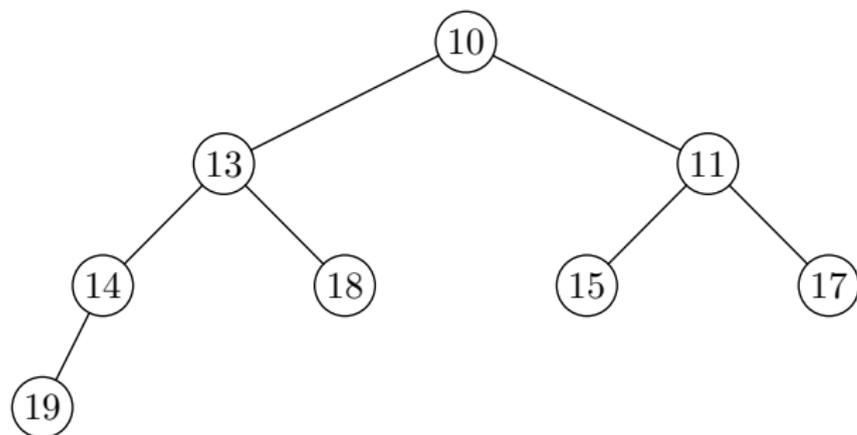
insert: $O(\log n)$

deleteMin: $O(\log n)$

decreaseKey: $O(\log n)$

buildHeap: $O(n)$

ПРЕДСТАВЛЕНИЕ

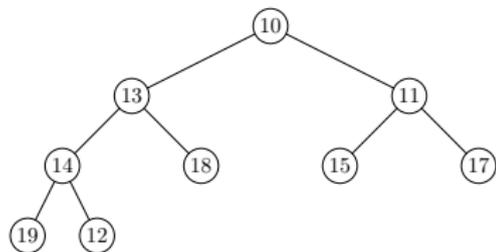


×	10	13	11	14	18	15	17	19
---	----	----	----	----	----	----	----	----

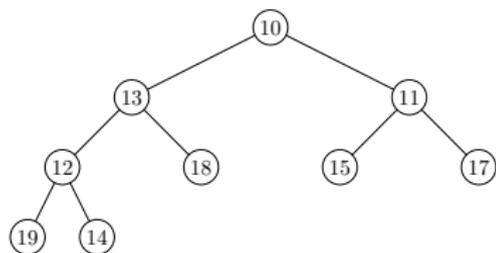
для элемента с индексом i :

- индексы “детей” = $\{2i, 2i + 1\}$
- индекс “родителя” = $\lfloor \frac{i}{2} \rfloor$

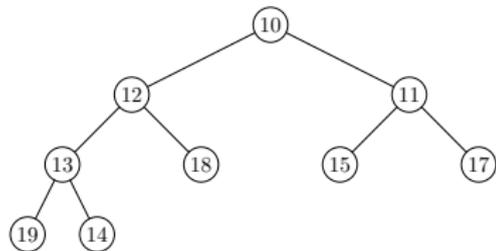
Всплытие (АКА SWIM, BUBBLE UP, SHIFT UP, ETC)



×	10	13	11	14	18	15	17	19	12
---	----	----	----	----	----	----	----	----	----

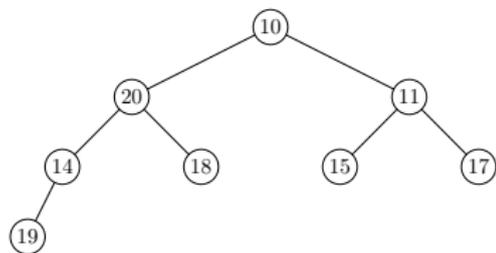


×	10	13	11	12	18	15	17	19	14
---	----	----	----	----	----	----	----	----	----

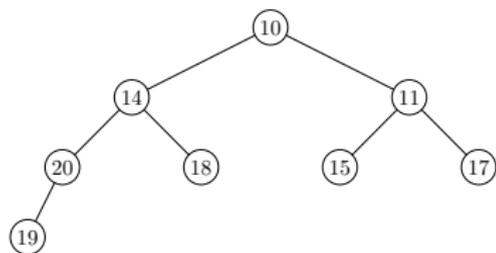


×	10	12	11	13	18	15	17	19	14
---	----	----	----	----	----	----	----	----	----

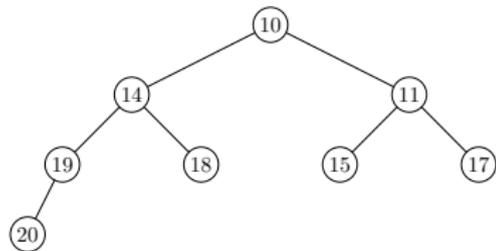
ПОГРУЖЕНИЕ (АКА SINK, BUBBLE DOWN, ETC)



×	10	20	11	14	18	15	17	19
---	----	----	----	----	----	----	----	----



×	10	14	11	20	18	15	17	19
---	----	----	----	----	----	----	----	----



×	10	14	11	19	18	15	17	20
---	----	----	----	----	----	----	----	----

ВСПЛЫТИЕ & ПОГРУЖЕНИЕ

`h = array of keys of length N`

```
procedure swim(i):  
  while i > 1 and h[i/2] < h[i]:  
    h[i/2], h[i] = h[i], h[i/2]  
    i = i / 2  
  end  
end
```

```
procedure sink(i):  
  while 2*i <= N:  
    int k = argmin(h[2*i], h[2*i + 1])  
    if h[k] < h[i]:  
      h[k], h[i] = h[i], h[k]  
    else:  
      break  
  end  
end
```

ПОСТРОЕНИЕ КУЧИ

`N = array length`
`h = array of keys of length N`

```
procedure buildHeap():  
  for i = N to 1:  
    sink(i)  
end
```

$$T(n) = \sum_0^d i2^{d-i} = 2^d \sum_0^d \frac{i}{2^i} = O(n)$$

ДРУГИЕ РЕАЛИЗАЦИИ

4. Биномиальная куча

insert: $O(1)$

deleteMin: $O(\log n)$

decreaseKey: $O(\log n)$

5. Фибоначчиева куча

insert: $O(1)$

deleteMin: $O(\log n)$

decreaseKey: $O(1)$

Хорошая теоретическая оценка, но на практике медленно.

СЛОЖНОСТЬ АЛГОРИТМА ДЕЙКСТРЫ

- в очереди всегда не более V элементов
- `deleteMin` вызываем V раз
- `decreaseKey` вызываем не более E раз

Результат:

- $O(V \log V + E \log V) = O(E \log V)$
при использовании двоичной кучи
(считаем, что граф связан, тогда $E \geq V - 1$)
- $O(V \log V + E)$
при использовании фибоначчиевой кучи

НУЖЕН ЛИ МЕТОД decreaseKey?

Можно обойтись и без него (см. вариант алгоритма Дейкстры на следующем слайде).

Сложность версии без decreaseKey:

- в очереди всегда не более E элементов
- deleteMin вызываем не более E раз
- insert вызываем не более E раз

Результат:

- $O(E \log E) = O(E \log V)$
при использовании двоичной кучи
(считаем, что нет кратных ребер, тогда $E \leq V(V - 1)/2$)
- $O(E \log E + E) = O(E \log V)$
при использовании фибоначчиевой кучи

АЛГОРИТМ ДЕЙКСТРЫ

Вариант без decreaseKey

```
procedure Dijkstra( G, s ):
  dist = array of size G.V
  prev = array of size G.V
  visited = array of size G.V
  Q = priority queue

  for x in G.V:
    dist[x] =  $\infty$ 
    prev[x] = -1
  dist[s] = 0
  Q.insert(s, dist[s])

  while not Q.isEmpty:
    x = Q.deleteMin()
    if not visited[x]:
      visited[x] = true
      for y in G.neighbours(x):
        key = dist[x] + l(x, y)
        if key < dist[y]:
          dist[y] = key
          prev[y] = x
          Q.insert(y, key)
        end
      end
    end
  end
end
end
```

РЕАЛИЗАЦИЯ МЕТОДА decreaseKey

Проблема: необходимо знать позицию элемента в бинарной куче.

Вариант 1: дополнительно хранить элементы в ассоциативном массиве (хеш-таблице). Для каждого элемента храним его индекс в массиве, изменяя при перестановках.

Вариант 2: если наши объекты пронумерованы / хранятся в массиве, то

- вместо объектов в очереди будем хранить их индексы;
- вместо хеш-таблицы ведем вспомогательный массив: по индексу объекту определяем его место в бинарной куче.

См. реализацию `IndexMinPQ` далее. Хорошо подходит для реализации алгоритма Дейкстры: индекс = номер вершины.

PRIORITYQUEUE В JAVA

```
public class PriorityQueue<E> {
    public PriorityQueue() { ... }
    public PriorityQueue(Comparator<? super E> comparator) { ... }

    public boolean add(E e) { return offer(e); }
    public boolean offer(E e) { ... }

    public E peek() { ... }
    public E poll() { ... }
    public E remove() { ... }

    public boolean contains(Object o) { ... } // O(n)
    public boolean remove(Object o) { ... } // O(n)
}
```

Метод `decreaseKey` отсутствует.

INDEXMINPQ

(Р. Седжвик, К. Уэйн "Алгоритмы на Java")

```
class IndexMinPQ<Key extends Comparable<Key>>
    IndexMinPQ(int maxN)
    void insert(int k, Key item) // log N
    void change(int k, Key item) // log N
    boolean contains(int k) // 1
    void delete(int k) // log N
    Key min() // 1
    int minIndex() // 1
    int delMin() // 1
    boolean isEmpty()
    int size()
```

```
private int N;           // number of elements on PQ
private int[] pq;       // binary heap using 1-based indexing
private int[] qp;       // inverse of pq - qp[pq[i]] = pq[qp[i]] = i
private Key[] keys;     // keys[i] = priority of i

public IndexMinPQ(int nMax) {
    keys = (Key[]) new Comparable[maxN + 1];
    pq   = new int[maxN + 1];
    qp   = new int[maxN + 1];
    for (int i = 0; i <= maxN; i++) qp[i] = -1;
}

public boolean contains(int i) {
    return qp[i] != -1;
}

// General helper functions
private boolean greater(int i, int j) {
    return keys[pq[i]].compareTo(keys[pq[j]]) > 0;
}

private void exch(int i, int j) {
    int swap = pq[i]; pq[i] = pq[j]; pq[j] = swap;
    qp[pq[i]] = i; qp[pq[j]] = j;
}
```

```
// Heap helper functions
private void swim(int k) {
    while (k > 1 && greater(k/2, k)) {
        exch(k, k/2);
        k = k/2;
    }
}

private void sink(int k) {
    while (2*k <= N) {
        int j = 2*k;
        if (j < N && greater(j, j+1)) j++;
        if (!greater(k, j)) break;
        exch(k, j);
        k = j;
    }
}
```

```
public void insert(int i, Key key) {
    if (contains(i)) throw new IllegalArgumentException("index is
        already in the priority queue");
    N++;
    qp[i] = N;
    pq[N] = i;
    keys[i] = key;
    swim(N);
}

public void change(int i, Key key) {
    if (!contains(i)) throw new NoSuchElementException("index is not
        in the priority queue");
    keys[i] = key;
    swim(qp[i]);
    sink(qp[i]);
}
```

```
public Key minKey() {
    return keys[pq[1]];
}

public int minIndex() {
    return pq[1];
}

public int delMin() {
    if (N == 0) throw new NoSuchElementException("Priority queue
        underflow");
    int min = pq[1];
    exch(1, N--);
    sink(1);
    qp[min] = -1;           // delete
    keys[pq[N+1]] = null; // to help with garbage collection
    pq[N+1] = -1;         // not needed
    return min;
}
```

ПРИМЕР ИСПОЛЬЗОВАНИЯ: СЛИЯНИЕ ПОТОКОВ

Дано: несколько текстовых файлов, строки в них отсортированы.

Требуется: слить их в один файл, в котором строки тоже будут отсортированы.

Используем `IndexMinPQ`, в которой индексом является номер файла, а ключом – а ключом следующая необработанная строка в этом файле (см. код на следующем слайде).

```
public class Multiway {

    private static void merge(In[] streams) {
        int N = streams.length;
        IndexMinPQ<String> pq = new IndexMinPQ<String>(N);
        for (int i = 0; i < N; i++)
            if (!streams[i].isEmpty())
                pq.insert(i, streams[i].readString());

        while (!pq.isEmpty()) {
            StdOut.println(pq.min());
            int i = pq.delMin();
            if (!streams[i].isEmpty())
                pq.insert(i, streams[i].readString());
        }
    }

    public static void main(String[] args) {
        int N = args.length;
        In[] streams = new In[N];
        for (int i = 0; i < N; i++)
            streams[i] = new In(args[i]);
        merge(streams);
    }
}
```

ПРИМЕНЕНИЕ ОЧЕРЕДИ С ПРИОРИТЕТАМИ

Применение:

- Пирамидальная сортировка (Heapsort)
- Графы: алгоритмы Дейкстры, Прима
- Оптимальное кодирование (код Хаффмана)
- Событийная симуляция (сталкивающиеся частицы)
- Искусственный интеллект (алгоритм A*)

АЛГОРИТМ A^* : ПОИСК ПУТИ

Задача: поиск пути в лабиринте или на игровой карте.

Используем поиск в ширину, но с использованием очереди с приоритетами: в первую очередь рассматриваем те клетки, которые ближе к цели.

АЛГОРИТМ A*: ИГРА В 15

Граф: вершины - все возможные позиции в игре. Две позиции соединены ребром, если можно перейти из одной в другую за 1 ход.

Граф очень большой, не пытаемся создать его целиком.

Начинаем поиск от заданной позиции, перебирая возможные ходы (соседние вершины). Все позиции помещаем в очередь с приоритетами (приоритет = эвристика, например, кол-во ходов уже потраченных + манхэттенское расстояние до целевой позиции).

Подробнее:

<http://introcs.cs.princeton.edu/java/assignments/8puzzle.html>

<http://introcs.cs.princeton.edu/java/assignments/checklist/8puzzle.html>