Адрианов Н.М.
Иванов А.Б.

# Алгоритмы и структуры данных

## Алгоритм Дейкстры. Очередь с приоритетами

# 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: $G = (V, E),\ l : E \to \mathbb{R}_+$

Задача: найти кратчайшие расстояния до всех вершин из данной вершины $\mathtt{s}$

# 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] = ∞
    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

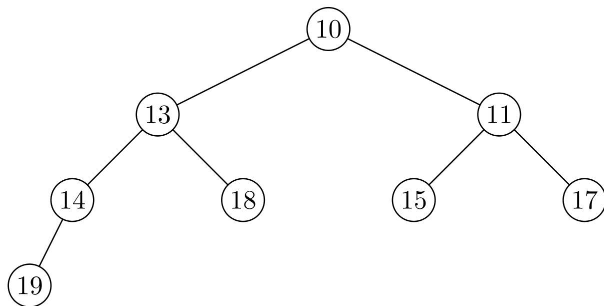
# Реализации очереди с приоритетами

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

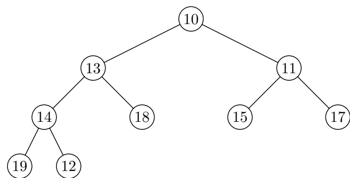



| × | 10 | 13 | 11 | 14 | 18 | 15 | 17 | 19 |
|---|---|---|---|---|---|---|---|---|

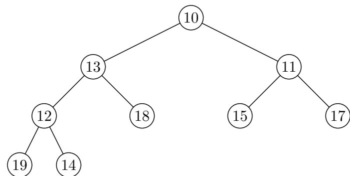
для элемента с индексом $i$:

- индексы "детей" $= \{2i, 2i+1\}$
- индекс "родителя" $= \left[\frac{i}{2}\right]$

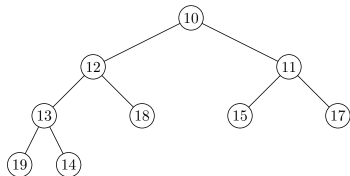
# Всплытие (aka swim, bubble up, shift up, etc)




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

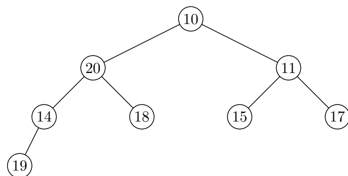



| × | 10 | 13 | 11 | 12 | 18 | 15 | 17 | 19 | 14 |
|---|---|---|---|---|---|---|---|---|---|

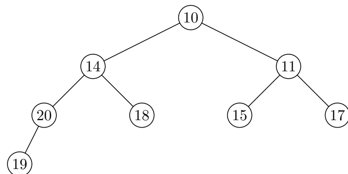



| × | 10 | 12 | 11 | 13 | 18 | 15 | 17 | 19 | 14 |
|---|---|---|---|---|---|---|---|---|---|

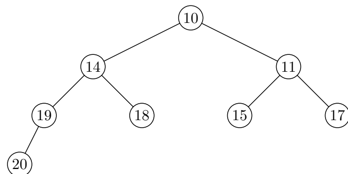
# Погружение (aka sink, bubble down, etc)




| × | 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 связен, 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] = ∞
    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
```

# 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()
```

```java
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;
}
```

```java
// 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;
    }
}
```

```java
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]);
}
```

```java
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