АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

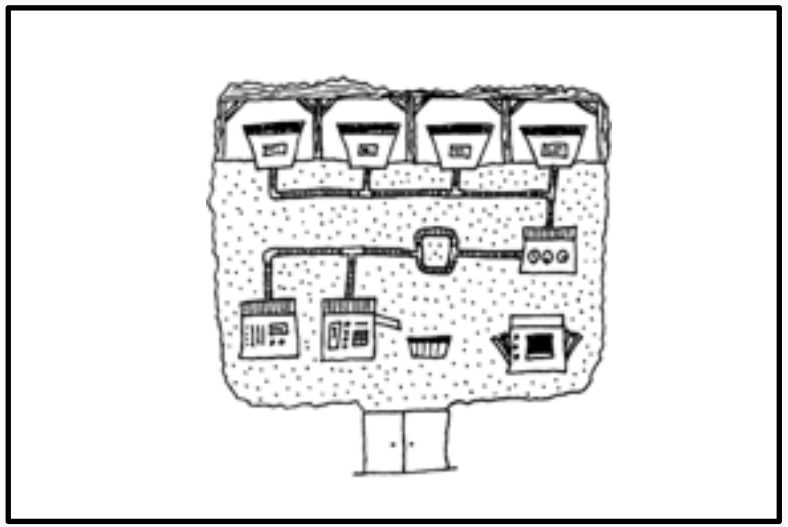
Лекция 3

Иванов Г.В.

Алгоритмы сортировки

- ***Сортировка*** – процесс упорядочивания элементов массива
- Многие программы используют алгоритмы сортировки. Часто время их работы определяется временем сортировки
- Данные часто упорядочены каким либо
- Многие задачи существенно быстрее решаются на предварительно упорядоченных данных

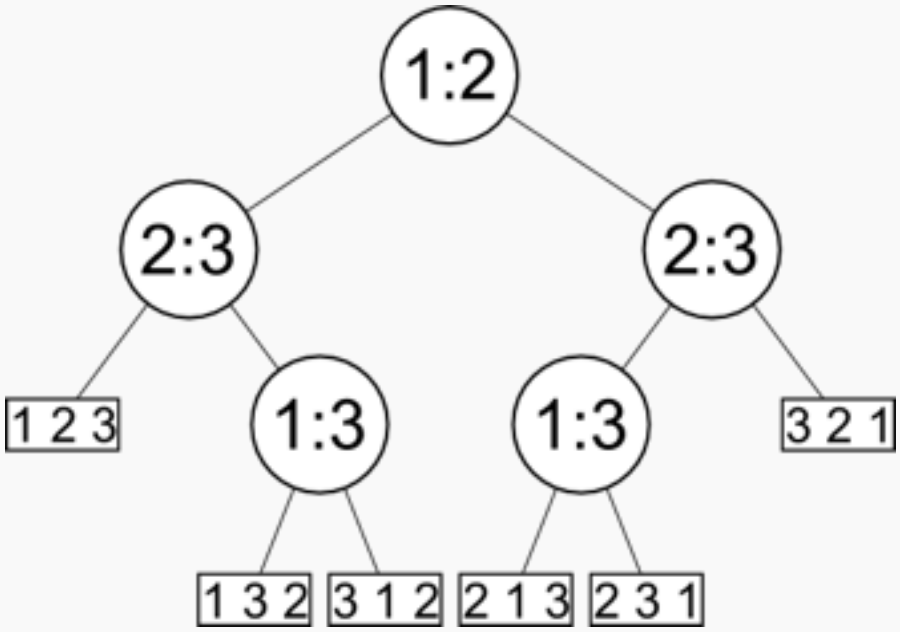
1. **Простые сортировки**
2. **Предел скорости**
3. **Ускорение**
4. **Хорошие сортировки**
5. **Особые свойства**
6. **Поразрядная сортировка**
7. **Сравнение**



Сортировка 1 и 2х элементов

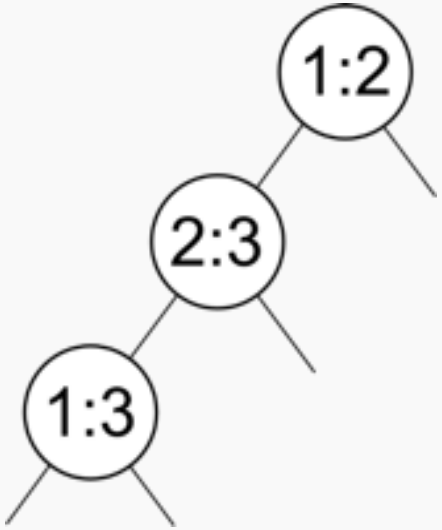

```
void sort_1(int *a) {
    return;
}

void sort_2(int *a) {
    if (a[1] < a[0]) {
      swap(a[1], a[0]);
    }
}
```

Сортировка 3х элементов




```
void sort_3(int *a) {
  if (a[1] < a[0]) {
    if (a[2] < a[1]) {
      // 2 1 0
    } else {
      if (a[2] < a[0])  {
        // 1 2 0
      } else {
        // 1 0 2
      }
    }
  } else {
    if (a[2] < a[0]) {
      // 2 0 1
    } else {
      if (a[2] < a[1])  {
        // 0 2 1
      } else {
        // 0 1 2
      }
    }
  }
}
```

Избыточное сравнение



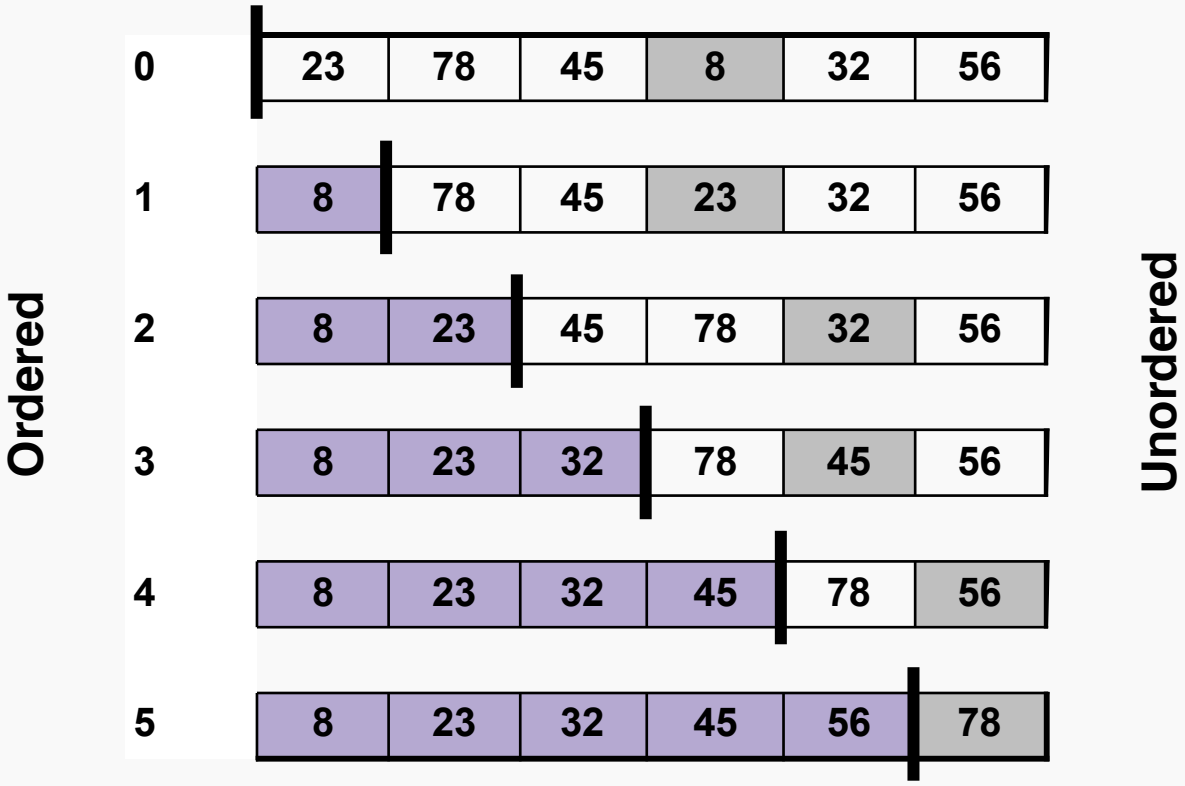
Простые сортировки

- **Существует множество алгоритмов сортировки**
  - **Сортировка выбором - Selection Sort**
  - **Сортировка вставками - Insertion Sort**
  - **Пузырьковая сортировка - Bubble Sort**

Сортировка выбором

- Разделим массив на две части: левую - упорядоченную и правую – неупорядоченную
- Будем гарантировать, что элементы в правой части больше чем в левой
- Выберем наименьший элемент в правой части и переставим его в её начало
- После каждого выбора и перестановки, будем смещать границу между частями массива на ***1*** вправо
- Выбор каждого нового элемента требует прохода по правой части
- Для сортировки массива из ***N*** элементов требуется ***N-1*** проход

Сортировка выбором



Сортировка выбором


```
void selection_sort(int *a, int n) {
  for (int i = 0; i < n - 1; ++i) {
    int min_index = i;
    for (int j = i + 1; j < n; ++j) {
       if (a[j] < a[min_index]) min_index = j;
    }
    swap(a[i], a[min_index]);
  }
}

void swap(int &a, int &b ) {
  int tmp = a;
  a = b;
  b = tmp;
}
```

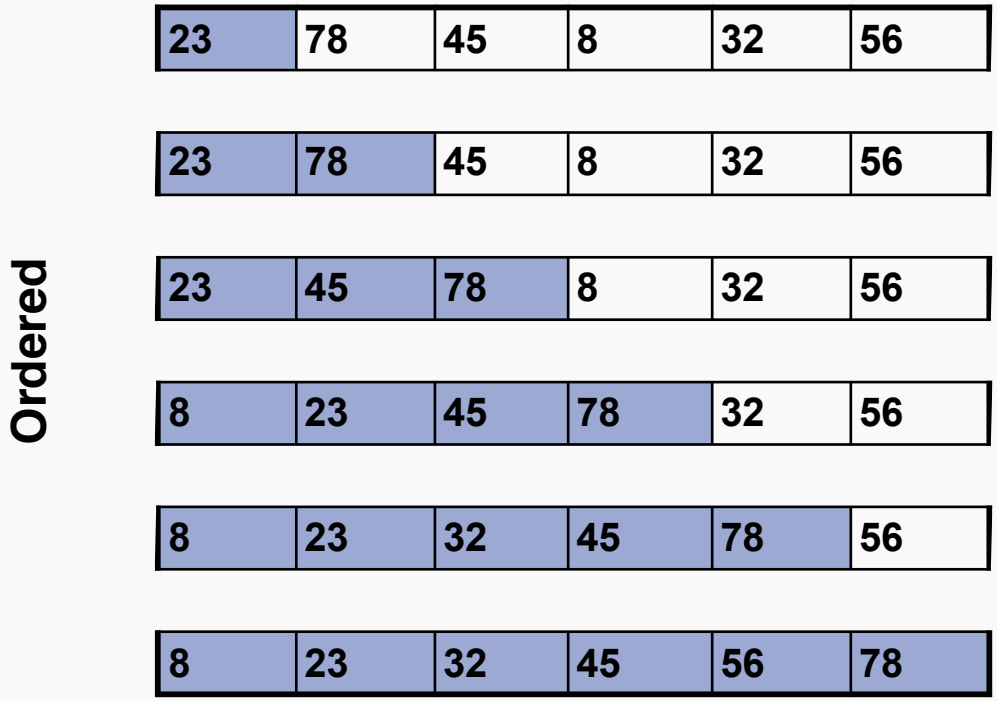
Сортировка выбором: Анализ

- В общем случае алгоритм сортировки состоит из сравнения ключей и перестановки элементов
- Время работы алгоритма пропорционально количеству сравнений и количеству перестановок
- Внешний цикл совершает ***n-1*** итерацию
- В каждой итерации ***1*** перестановка
- В 1й итерации ***n-1*** сравнение, во 2й – ***n-2***, ... в n-1й – ***1***
- Ровно $n(n-1)/2$ сравнений
- Ровно $n-1$ перемещений

Сортировка вставками

- Сортировка вставками – простой алгоритм часто применяемый на малых объёмах данных
- Самый популярный метод сортировки у игроков в покер
- Массив делится на две части, упорядоченную - левую и неупорядоченную - правую
- На каждой итерации выбираем элемент из правой части и вставляем его на подходящее место в левой части
- Массив из ***n*** элементов требует ***n-1*** итерацию

Сортировка вставками



Сортировка вставками


```
void insertion_sort(int *a, int n) {
  for (int i = 1; i < n; ++i) {
    int tmp = a[i];
    for (int j = i; j > 0 && tmp < a[j-1]; --j) {
      a[j] = a[j-1];
    }
    a[j] = tmp;
   }
}
```

Сортировка вставками: Анализ

Время работы алгоритма зависит не только от размера массива, но и от порядка элементов

Лучший случай: **➔ O(n)**

- Массив упорядочен по возрастанию
- Внутренний цикл сделает 0 итераций
- Количество копирований: 2*(n-1) ➔ O(n)
- Количество сравнений: (n-1) ➔ O(n)

Худший случай: **➔ $O(n^2)$**

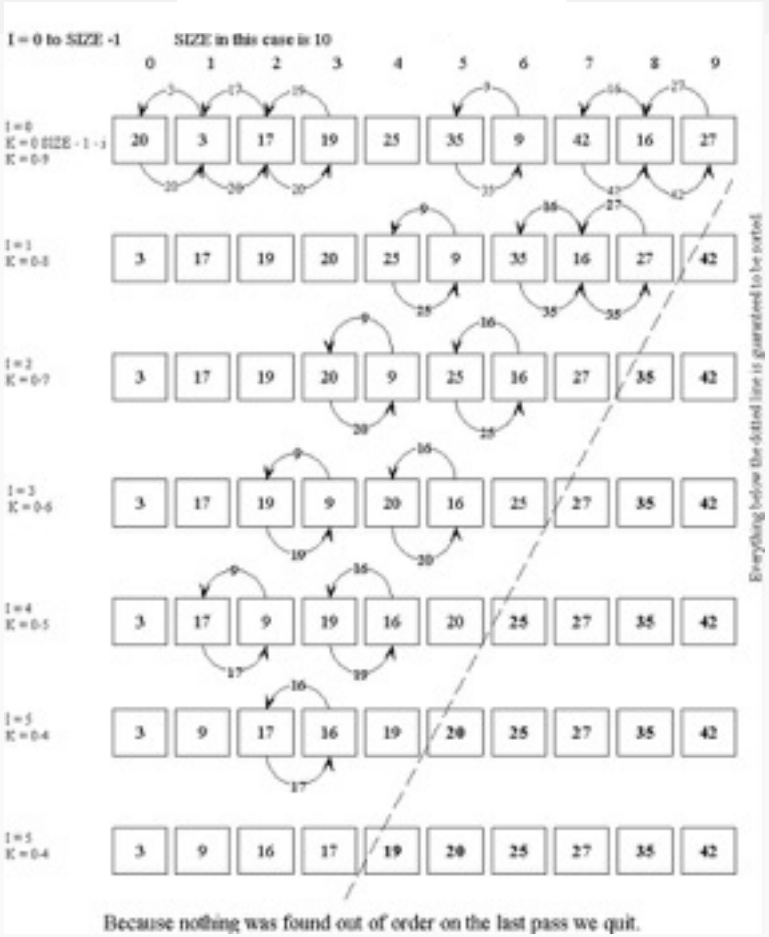
- Массив упорядочен в порядке убывания:
- Внутренний цикл работает i-1 итерацию, для i = 2,3, …, n
- Количество копирований: 2*(n-1)+(1+2+...+n-1)= 2*(n-1)+ n*(n-1)/2 ➔ $O(n^2)$
- Количество сравнений: (1+2+...+n-1)= n*(n-1)/2 ➔ $O(n^2)$

В среднем: **➔ $O(n^2)$**

Сортировка пузырьком

- Массив делится на две части, упорядоченную - левую и неупорядоченную - правую
- На каждой итерации проходим правую часть сравнивая текущий элемент с соседом слева
  - Меняем элементы местами если сосед больше
  - Иначе, уменьшаем индекс текущего на ***1***
- Наименьший элемент всплывает к границе левой части
- Останавливаемся если не было ни одного обмена
- Массив из *n* элементов требует максимум ***n-1*** итерацию

Сортировка пузырьком



Сортировка пузырьком: Анализ

- **Лучший случай**
  - *1* проход, *N-1* сравнение, *0* обменов => $O(N)$
- **Худший случай**
  - *N-1* проход, *N(N-1)/2* сравнений, *N-1* обменов => $O(N^2)$

Теоретико-числовая оценка сложности

- Количество перестановок ***N*** элементов: ***N!***
- ***1*** сравнение = ***1*** бит информации
- Для записи номера перестановки нужно

$$log_2(N!) \cong N\ log(N)$$ бит

Сортировка вставками. *n(log(n)) ?*

- Будем искать позицию вставки в упорядоченную часть массива бинарным поиском
- O(n log(n)) - операций сравнения
- $O(n^2)$ – операций перемещения
- Используем memmove для уменьшения константы C_2
- $T_n \leq C_1 n \log(n) + C_2 n^2$

Сортировка вставками. *n(log(n)) ?*


```
void memmove(char *dst, char *src, int size);

void insertionSortFast(int *a, int n) {
  for (int i = 1; i < n; ++i) {
    int new_pos = binary_search(a, i, a[i]);

    if (new_pos < i) {
      int tmp = a[i];
      memmove(&a[new_pos + 1], &a[new_pos],(i - new_pos)*sizeof(int));

      a[new_pos] = tmp;
    }
   }
}
```

Сортировка вставками. *n(log(n)) ?*

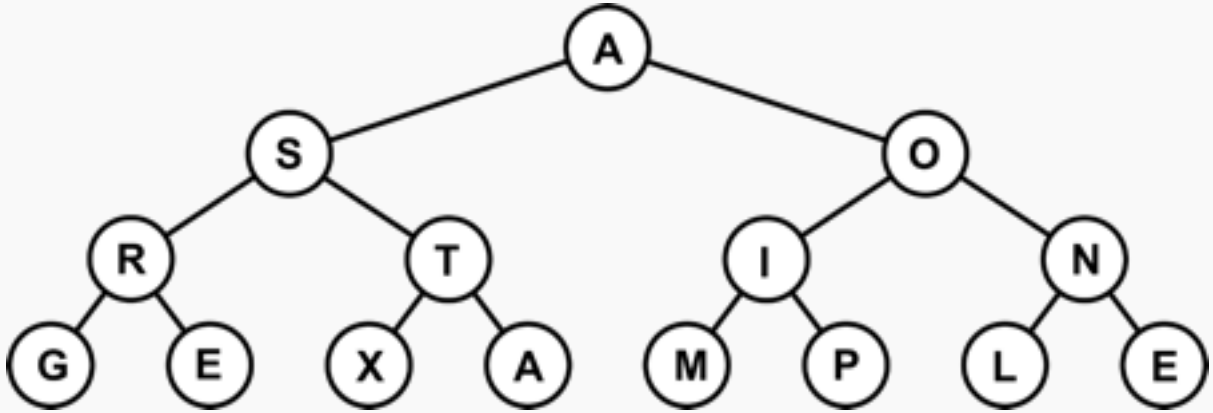
- Будем искать позицию вставки за время O(log(i – k))
- O(n log(n)) , θ(n) - операций сравнения
- $O(n^2)$, θ(1) - операций перемещения

Хорошие сортировки

- Пирамидальная сортировка - Heap Sort
- Сортировка слиянием - Merge Sort
- Быстрая сортировка - Quick Sort

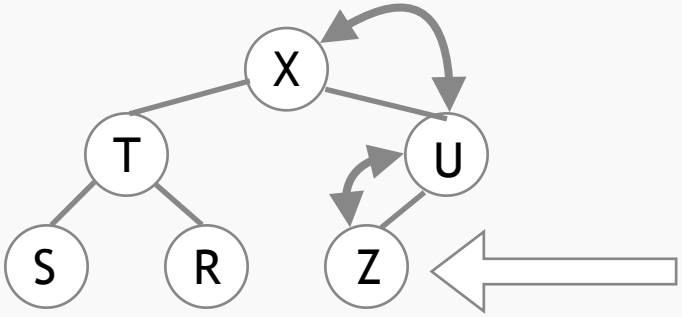
Пирамидальная сортировка

N вставок в кучу: ***N · O(log(N))***
N Извлечение минимума из кучи: ***N · O(log(n))***
Построение кучи из ***N*** элементов: ***O(N·log(N))***



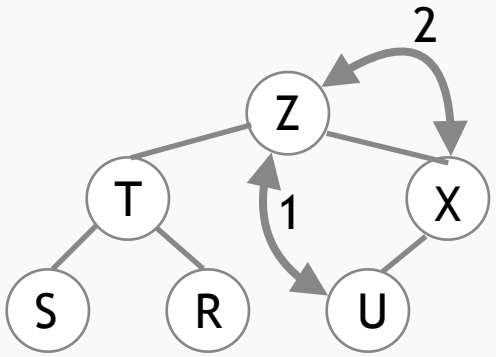
Пирамидальная сортировка .


```
void heap_insert(int *a, int n, int x)
{
  a[n+1] = x;
  for (int i = n+1; i > 1;) {
    if (a[i] > a[i/2]) {
      swap(a[i], a[i/2]);
      i = i/2;
    } else {
      break;
    }
  }
}
```



Пирамидальная сортировка ..


```
void heap_pop(int *a, int n) {
  swap(a[n],a[1]);

  for (int i = 1; 2*i < n;) {
    i *= 2;
    if (i+1 < n && a[i] < a[i+1]) {
      i += 1;
    }
    if (a[i/2] < a[i]) {
      swap(a[i/2], a[i]);
    }
  }
}
```



Пирамидальная сортировка ..!


```
void heap_sort(int *data, int n) {
  int *buff = new int[n+1];
  for (int i = 0; i < n; ++i) {
    heap_insert(buff, i, data[i]);
  }
  for (int i = 0; i < n; ++i) {
    data[n-1-i] = buff[1];
    heap_pop(buff, n - i);
  }
  delete [] buff;
}
```

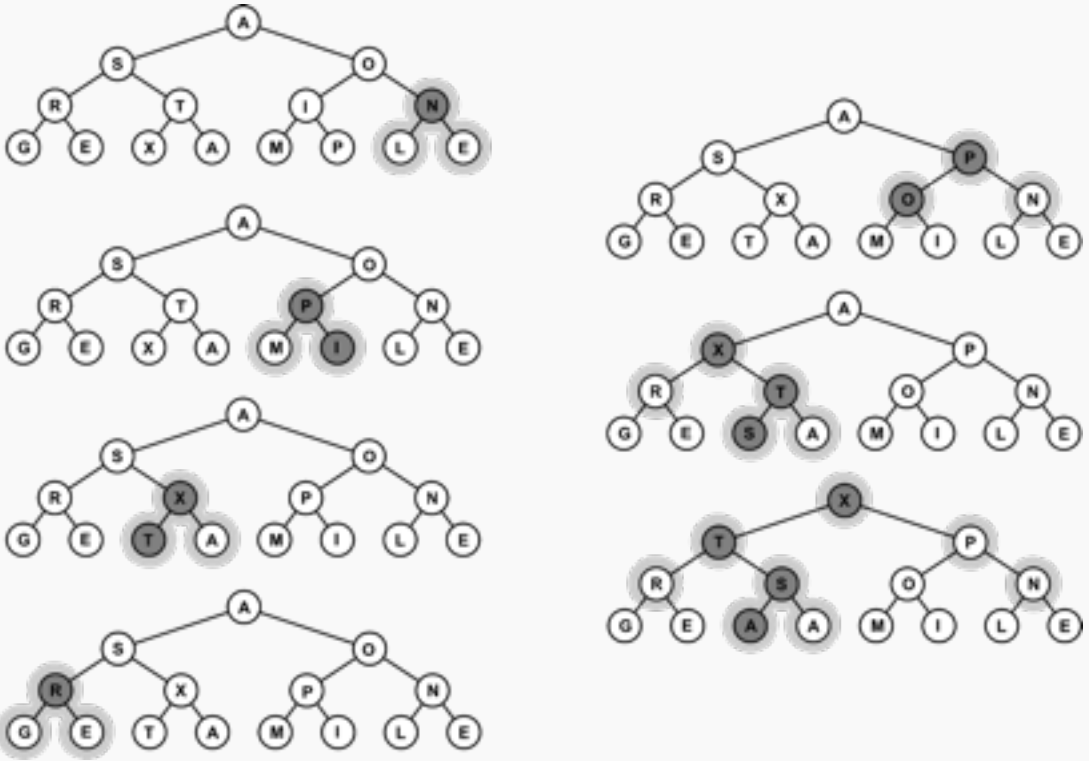
Пирамидальная сортировка

- Построить пирамиду за линейное время: ***O(N)***
- ***N*** раз достать максимальный элемент: ***O(N·log(N))***
- Не использовать дополнительную память

Пирамида за линейное время

- Для внутренних элемента восстановить порядок
- По 2 сравнения на уровень дерева N/2 • 2

Пирамида за линейное время



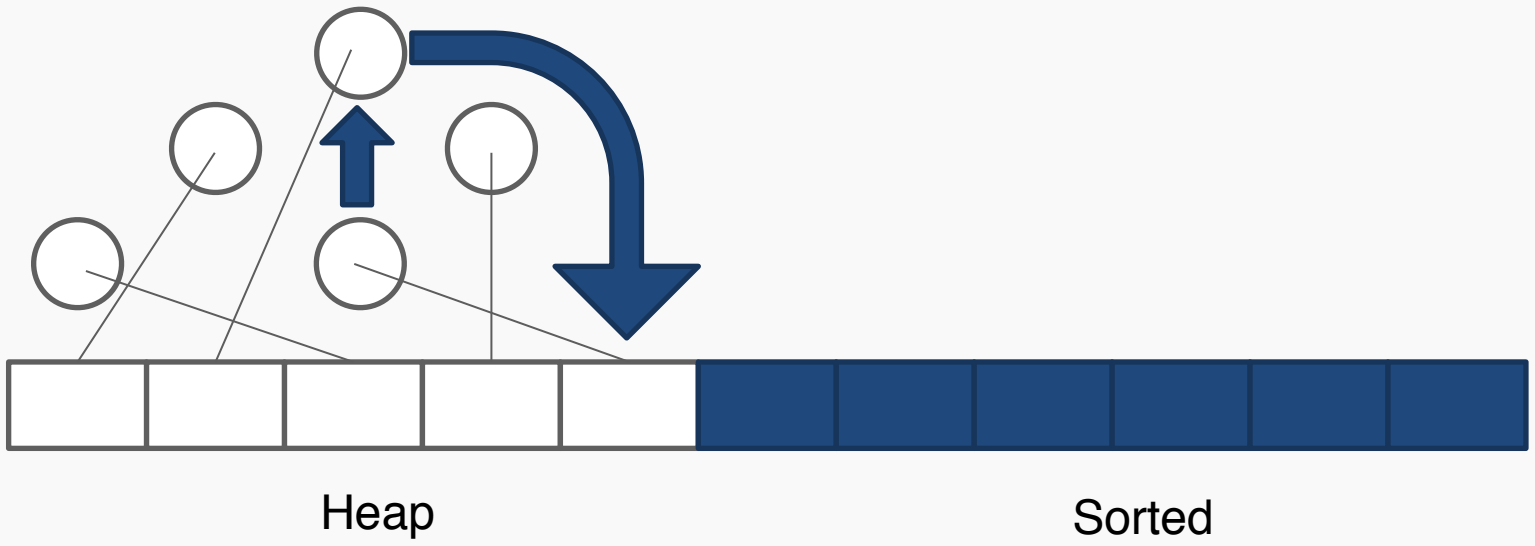
Пирамида за линейное время


```
void heap_make(int *a, int n) {
  for (int i = n/2; i >= 1; --i) {
    for (int j = i; j <= n/2;) {
      int k = j*2;
      if (k+1 <= n and a[k] < a[k+1]) {
        ++k;
      }
      if (a[j] < a[k]) {
        swap(a[k],a[j]);
        j = k;
      } else {
        break;
      }
    }
  }
}
```

Пирамида за линейное время


```
void heap_sort_fast(int *data, int n) {
  heap_make(data - 1, n);
  for (int i = 0; i < n; ++i) {
    heap_pop(data - 1, n - i);
  }
}
```

Пирамидальная сортировка как эволюция сортировки выбором



?

? ***Вопросы*** ?