Курс «Алгоритмы и алгоритмические языки»

Лекция 15

Сортировка

- ***Оценка сложности алгоритмов сортировки***
- Скорость сортировки определяется количеством сравнений и количеством обменов (обмены занимают больше времени). Эти показатели интересны для худшего и лучшего случаев, а также интересно их среднее значение.
- Кроме скорости оценивается «естественность» алгоритма сортировки:
  естественным считается алгоритм, который на уже отсортированном массиве работает минимальное время, а на не отсортированном работает тем дольше, чем больше степень его неупорядоченности.
- Важным показателем является и объем дополнительной памяти для хранения промежуточных данных алгоритма.
  Для рекурсивных алгоритмов расход памяти связан с необходимостью дублировать стек, в котором расположены некоторые промежуточные данные.

Сортировка

- ***Оценка сложности алгоритмов сортировки***.
- ***Теорема***. *Для любого алгоритма S внутренней сортировки сравнением массива из n элементов количество сравнений*

$$C_S \geq \mathrm{O}(n \cdot \log_2(n))$$

- Доказательство.

(1) *Для любого алгоритма S внутренней сортировки сравнением массива из n элементов количество сравнений*

$$C_S \geq \log_2(n!).$$

(a) Алгоритм S можно представить в виде двоичного дерева сравнений.
Так как любая перестановка индексов рассматриваемого массива может быть ответом в алгоритме, она должна быть приписана хотя бы одному листу дерева сравнений.
Таким образом, дерево сравнений будет иметь не менее n! листьев.

Сортировка

- ***Оценка сложности алгоритмов сортировки***.
- ***Теорема***. *Для любого алгоритма S внутренней сортировки сравнением массива из n элементов количество сравнений*

$$C_S \geq \mathrm{O}(n \cdot \log_2(n))$$

- Доказательство.

(1) *Для любого алгоритма S внутренней сортировки сравнением массива из n элементов количество сравнений*

$$C_S \geq \log_2(n!). \qquad (*)$$

(б) Для высоты h_m двоичного дерева с m листьями имеет место оценка:

$$h_m \geq \log_2 m.$$

Любое двоичное дерево высоты h можно достроить до полного двоичного дерева высоты h, а у полного двоичного дерева высоты h 2^h листьев.

Применив полученную оценку к дереву сравнений, получим оценку (*)

Сортировка

- ***Оценка сложности алгоритмов сортировки***.
- ***Теорема***. *Для любого алгоритма S внутренней сортировки сравнением массива из n элементов количество сравнений*

$$C_S \geq O(n \cdot \log_2(n))$$

- Доказательство.

(2) К $\log_2(n!)$ применим формулу Стирлинга

$$n! = \sqrt{2\pi n} \cdot n^n e^{-n} e^{\vartheta(n)} \qquad (**)$$

$$|\vartheta(n)| \leq \frac{1}{12n}$$

Логарифмируя (**), получаем

$$\log(n!) = \frac{1}{2}\log(2\pi n) + n \cdot \log(n) - n + \vartheta(n)$$

$$\log(n!) \geq O(n \cdot \log(n))$$

Быстрая сортировка

- ***QuickSort* – рекурсивная Си-функция следующего вида**:

```
/* Быстрая сортировка. Предполагается, что left<right */
static void QuickSort (int *a, int left, int right) {
  /* comp – компаранд, i, j – значения индексов элементов массива a */

  int comp, tmp, i, j;
  i = left; j = right;
  comp = a[(left + right)/2]; //можно и a[left] или a[right]
  /* построение Partition – цикл do-while */
  do {
    while (a[i] < comp && i < right)
      i++;
    while (comp < a[j] && j > left)
      j--;
    if (i <= j) {
      tmp = a[i];
      a[i] = a[j];
      a[j] = tmp;
      i++; j--;
    }
  } while (i < j);
  /* продолжение сортировки, если не все отсортировано */
  if (left < j)
    QuickSort (a, left, i - 1);
  if (i < right)
    QuickSort (a, j + 1, right);
}
```

Быстрая сортировка

- ***QuickSort* – рекурсивная Си-функция следующего вида**:

- Нужно, чтобы значение компаранда было таким, чтобы он попал в середину результирующей последовательности. Мы пытаемся угадать, какой из элементов массива имеет такое значение. Чем лучше мы угадаем, тем быстрее выполнится алгоритм
- Программа быстрой сортировки.

```
void qsort (int *a, int n) {
      QuickSort (a, 0, n-1);
}
```

Быстрая сортировка

- Покажем, что цикл **do-while** действительно строит нужное нам разбиение массива **a[]**.

  (1) В процессе работы цикла индексы **i** и **j** не выходят за пределы отрезка [**left, right**], так как в циклах **while** выполняются соответствующие проверки.

  (2) В момент окончания работы цикла
  do-while j ≤ right,
  так как части разбиения не могут быть пустыми: хотя бы один элемент массива **a[]** (в крайнем случае **a[right]**) содержится в правой части разбиения.

  (3) Аналогично, в момент окончания работы цикла
  do-while i ≥ left.

  (4) В момент окончания работы цикла **do-while** любой элемент подмассива **a[left..k-1]** не больше любого элемента подмассива **a[k+1.. right]** , где **k** – индекс компаранда, что очевидно.

Быстрая сортировка

- Работа цикла **do-while** на примере: **5 3 2 6 4 1 3 7**.
  - Пусть в качестве первого компаранда выбран первый элемент массива – **5** (**a[left]**).
  - Во время первого прохода цикла **do-while** после выполнения обоих циклов **while** получим:

    (5) 3 2 6 4 1 {3} 7;

    (в круглых скобках элемент с индексом **i**,
    в фигурных – элемент с индексом **j**).
  - Поскольку **i < j**, элементы, выделенные скобками, нужно поменять местами (оператор **if**):

    3 (3) 2 6 4 {1} 5 7;
  - В результате второго прохода цикла **do-while** получим:
    до обмена **3 3 2 (6) 4 {1} 5 7;**
    после обмена **3 3 2 1 ({4}) 6 5 7;**
  - Теперь массив **a** состоит из двух подмассивов
    3 3 2 1 4 и **6 5 7**
    причем **i = 5**, **j = 5**.
    и нужно рекурсивно применить метод к этим подмассивам.

Быстрая сортировка

- При выборе компаранда можно брать первый элемент, значение которого больше значения следующего элемента. Для результирующих подмассивов из примера компаранды заключены в квадратные скобки:

  3 [3] 2 1 4; **[6] 5 7.**

- Если $f(n)$ и $g(n)$ – некоторые функции, то запись $g(n) = \Theta(f(n))$ означает, что найдутся такие константы $c1, c2 > 0$ и такое n_0, что для всех $n \geq n_0$ выполняются соотношения

  $$0 \leq c_1 f(n) \leq g(n) \leq c_2 f(n).$$

  т.е. при больших n
  $f(n)$ хорошо описывает поведение $g(n)$.

Быстрая сортировка

- Оценка времени выполнения алгоритма **`QuickSort`**.

(1) Время выполнения цикла **`do-while`**
$\Theta(n)$, где $n = right - left + 1$.

(2) для алгоритма **`QuickSort`** максимальное (наихудшее) время выполнения $T_{max}(n) = \Theta(n^2)$.

Наихудшее время: при каждом *Partition* массив длины n разбивается на подмассивы длины 1 и $n - 1$.

(2Д) Для $T_{max}(n)$ имеет место соотношение
$T_{max}(n) = T_{max}(n - 1) + \Theta(n)$.
Очевидно, что $T_{max}(1) = \Theta(1)$.
Следовательно,

$$T_{max}(n) = T_{max}(n - 1) + \Theta(n) =$$

$$\sum_{k=1}^{n} \Theta(k) = \Theta(\sum_{k=1}^{n} k) = \quad n \cdot (n - 1)/2 = \Theta(n^2).$$

(3) Если исходный массив **`a`** отсортирован в порядке убывания, время его сортировки в порядке возрастания с помощью алгоритма **`QuickSort`** будет $\Theta(n^2)$.

Быстрая сортировка

- Оценка времени выполнения алгоритма **`QuickSort`**.

(4) Минимальное и среднее время выполнения алгоритма *QuickSort*

$$T_{mean}(n) = \Theta(n \cdot \log n)$$

с разными константами: чем ближе разбиение на подмассивы к сбалансированному, тем константы меньше.

(4Д) Доказательство использует теорему о рекуррентных оценках [1]

(5) Рекуррентное соотношение для минимального (наилучшего) времени сортировки $T_{min}(n)$ имеет вид

$$T_{min}(n) = 2 \cdot T_{min}(n/2) + \Theta(n),$$

так как минимальное время получается тогда, когда на каждом шаге удается выбрать компаранд, который делит массив на два подмассива одинаковой длины $\lceil n/2 \rceil$.

Применяя ту же теорему, получаем $T_{min}(n) = \Theta(n \cdot \log n)$.

[1] Т. Кормен, Ч. Лейзерсон, Р. Ривест. Алгоритмы: построение и анализ. М.: МЦНМО, 1999. ISBN 5-900916-37-5, с. 66 – 73.

Быстрая сортировка

- Оценка времени выполнения алгоритма **`QuickSort`**.

(6) Рекуррентное соотношение для (n) в общем случае, когда на каждом шаге массив делится в отношении q:(n – q), причем q с вероятностью 2/n принимает значение 1 и с вероятностями 1/n значения 2, …, n – 1, имеет вид

$$T(n) = \frac{1}{n}\left(T(1) + T(n-1) + \sum_{q=1}^{n-1} (T(q) + T(n-q)) \right) + \Theta(n)$$

Достаточно сложные рассуждения позволяют решить это соотношение и установить, что $T(n) = \Theta(n \cdot \log n)$
(та же книга, с.160 – 164).

Двоичные деревья поиска

- Проблема: организовать хранилище данных, которое позволяет хранить большие объемы данных и предоставляет возможность быстро находить и модифицировать данные.
- Хранилище данных обеспечивает пользователю *интерфейс*, в котором определены *словарные операции*: *search* (найти, иногда называется *fetch*), *insert* (вставить) и *delete* (удалить).
- Варианты решения – деревья поиска, хеширование
- *Двоичное дерево* – набор узлов, который:
  - либо пуст (пустое дерево),
  - либо разбит на три непересекающиеся части:
    узел, называемый *корнем*,
    двоичное дерево, называемое *левым поддеревом*, и
    двоичное дерево, называемое *правым поддеревом*.

Двоичные деревья поиска

- ***Замечание.*** Двоичное дерево не является частным случаем обычного дерева, хотя у этих структур много общего.
  Основные отличия:

  (1) Пустое дерево является двоичным деревом, но не является обычным деревом.

  (2) Двоичные деревья **(*A*(*B*,*NULL*))** и **(*A*(*NULL*,*B*))** различны.
  Деревья **(*A*(*B*,*NULL*))** и **(*A*(*NULL*,*B*))**одинаковы.

Двоичные деревья поиска

- Структура для представления узла двоичного дерева:

```
struct BT_node {
  int key;
  struct BT_node *left;
  struct BT_node *right;
  struct BT_node *parent;
}
```

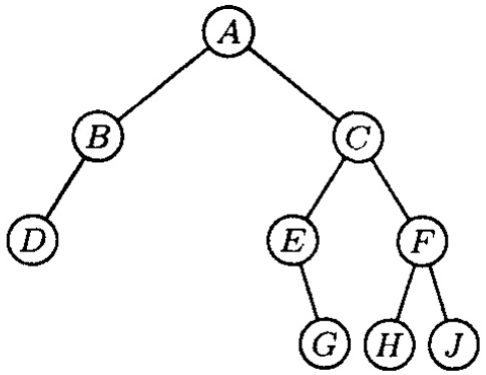

Рис. 1. Двоичное дерево

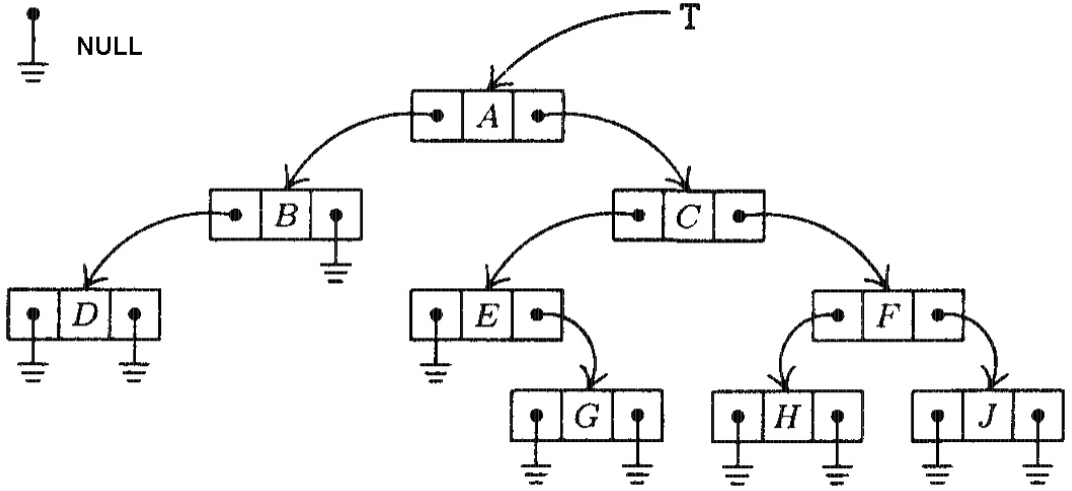

Рис. 2 Представление дерева с рис.1 в компьютере.

Двоичные деревья поиска

- Структура для представления узла двоичного дерева:

```
struct BT_node {
  int key;
  struct BT_node *left;
  struct BT_node *right;
  struct BT_node *parent;
}
```

- Ключи в двоичном дереве поиска хранятся с соблюдением свойства упорядоченности:

  Пусть *x* – произвольный узел двоичного дерева поиска. Если узел *y* принадлежит левому поддереву, то

  $$key[y] \le key[x],$$

  если *y* находится в правом поддереве узла *x*, то

  $$key[y] \ge key[x].$$

Двоичные деревья поиска: *поиск узла*

- **На входе**: искомый ключ *k* и указатель *root* на корень поддерева, в котором производится поиск.

  На выходе: указатель на узел с ключом *key* (если такой узел есть), либо пустой указатель NULL.

```
struct BT_node *Btsearch (struct BT_node *root, int k)
{
  if (! root || root->key == k)
    return root;
  if (root->key > k)
    return Btsearch (root->left, k);
  else
    return Btsearch (root->right, k);
}
```

Двоичные деревья поиска: *поиск узла*

- Итеративная версия поиска.

```
struct BT_node *Btsearch (struct BT_node *root, int k)
{
  struct BT_node *p = root;

  while (p && p->key != k)
    if (k < p->key)
      p = p->left;
    else
      p = p->right;
  return p;
}
```

- Среднее время поиска $O(h)$, где h – высота дерева.

Двоичные деревья поиска: *минимум и максимум*

- **На входе**: указатель *root* на корень поддерева.
  На выходе: указатель на узел с минимальным ключом *k*.

```
struct BT_node *Btmin (struct BT_node *root)
{
  struct BT_node *p = root;
  while (p->left)
    p = p->left;
  return p;
}
```

- Среднее время выполнения $O(h)$, где h – высота дерева.

Двоичные деревья поиска: *следующий элемент*

- **На входе**: указатель *node* на узел дерева.
  На выходе: указатель на следующий за *node* узел дерева.

```
struct BT_node *Btsucc (struct BT_node *node) {
  struct BT_node *p = node, *q;
  /* I случай: правое поддерево узла не пусто */
  if (p->right)
    return Btmin (p->right);
  /* II случай: правое поддерево узла пусто,
     идем по родителям до тех пор, пока не найдем
     родителя, для которого наше поддерево левое */
  q = p->parent;
  while (q && p == q->right) {
    p = q;
    q = q->parent;
  }
  return q;
}
```

- Среднее время выполнения $O(h)$, где h – высота дерева.