Московский государственный университет им. М. В. Ломоносова
Факультет вычислительной математики и кибернетики

Алгоритмы и алгоритмические языки
Лекция 20

20 ноября 2019 г.

Быстрая сортировка

```
static void QuickSort (int *a, int left, int right) {
  /* comp -- компаранд, i, j -- значения индексов */
  int comp, tmp, i, j;
  i = left; j = right;
  comp = a[(left + right)/2];
  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);
```

Быстрая сортировка

```
static void QuickSort (int *a, int left, int right) {
  ...
  if (left < j)
    QuickSort (a, left, j);
  if (i < right)
    QuickSort (a, i, right);
}
```

Программа быстрой сортировки.

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

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.

Быстрая сортировка

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

- В процессе работы цикла индексы `i` и `j` не выходят за пределы отрезка `[left, right]`, так как в циклах `while` выполняются соответствующие проверки.
- В момент окончания работы цикла `do-while` $j \leqslant right$, так как части разбиения не могут быть пустыми: хотя бы один элемент массива `a[]` (в крайнем случае `a[right]`) содержится в правой части разбиения.
- Аналогично, в момент окончания работы цикла `do-while` $i \geqslant left$.
- В момент окончания работы цикла `do-while` любой элемент подмассива `a[left..j]` не больше любого элемента подмассива `a[i..right]`, что очевидно.

Быстрая сортировка. Пример разделения массива

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: `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`.
- Третий проход лишь увеличивает i.

Теперь массив a состоит из двух подмассивов `3 3 2 1 4` и `6 5 7`, причём $i = 5$, $j = 4$. Нужно рекурсивно применить метод к этим подмассивам.

Быстрая сортировка. Выбор компаранда

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` и `[6] 5 7`.

Оценка времени работы быстрой сортировки (Θ-нотация).
Если $f(n)$ и $g(n)$ — некоторые функции, то запись $g(n) = \Theta(f(n))$ означает, что найдутся такие константы $c_1, c_2 > 0$ и такое n_0, что для всех $n \geqslant n_0$ выполняются соотношения

$$0 \leqslant c_1 f(n) \leqslant g(n) \leqslant c_2 f(n),$$

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

Быстрая сортировка. Оценка времени работы

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

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 $n - 1$.

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

$$T_{max}(n) = T_{max}(n-1) + \Theta(n) = n(n-1)/2 = \Theta(n^2).$$

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 работы

Минимальное и среднее время выполнения алгоритма QuickSort $T_{mean}(n) = \Theta(n \log n)$ с разными константами: чем ближе разбиение на подмассивы к сбалансированному, тем константы меньше.

Доказательство использует теорему о рекуррентных оценках из Кормен, Ч. Лейзерсон, Р. Ривест. Алгоритмы: построение и анализ. М.: МЦНМО, 1999. ISBN 5-900916-37-5, с. 66-73.

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

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

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 $T_{min}(n) = \Theta(n \log n)$.

Быстрая сортировка. Оценка времени работы

Рекуррентное соотношение для $T(n)$ в общем случае, когда на каждом шаге массив делится в отношении $q : (n - q)$, причем q равновероятно распределено между 1 и n, также можно решить и установить, что $T(n) = \Theta(n \log n)$ (та же книга, с. 160-164).

Двоичное дерево

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*.

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

- пустое дерево является двоичным деревом, но не является обычным деревом;
- двоичные деревья `(A(B,NULL))` и `(A(NULL,B))` различны, а обычные деревья — одинаковы.

Термины: узлы, ветви, корень, листья, высота.

Описание узла двоичного дерева на Си

```
typedef struct bin_tree {
  char info;
  struct bin_tree *left;
  struct bin_tree *right;
} node;
```

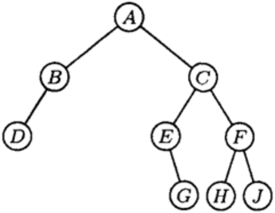


Рис. 1. Двоичное дерево

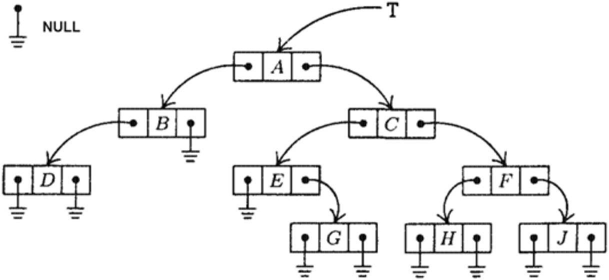


Рис. 2 Представление дерева с рис.1 в компьютере.

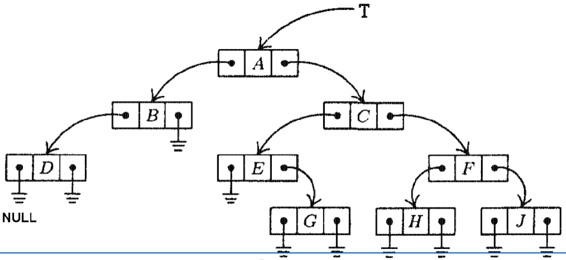
Способы обхода двоичного дерева

Обход в глубину в *прямом порядке*:

- обработать корень,
- обойти левое поддерево,
- обойти правое поддерево.

Порядок обработки узлов дерева: `A B D C E G F H J`.

Линейная последовательность узлов, полученная при прямом обходе, отражает «спуск» информации от корня дерева к листьям.



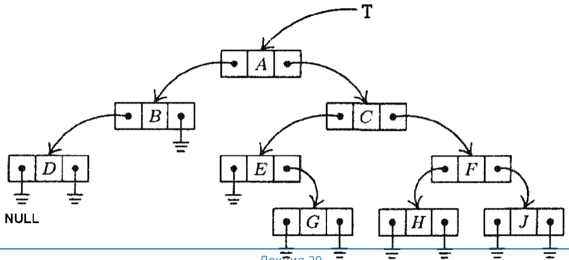
Способы обхода двоичного дерева

Обход в глубину в *обратном порядке*:

- обойти левое поддерево,
- обойти правое поддерево,
- обработать корень.

Порядок обработки узлов дерева: `D B G E H J F C A`.

Линейная последовательность узлов, полученная при обратном обходе, отражает «подъём» информации от листьев к корню дерева.

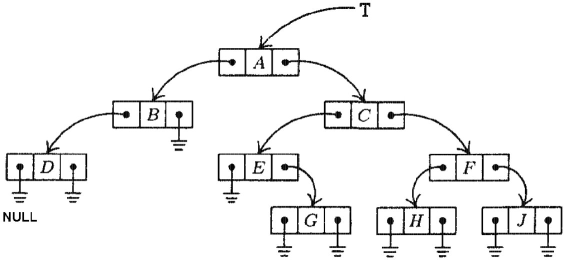


Способы обхода двоичного дерева

Симметричный обход в глубину (обход в *симметричном порядке*):

- обойти левое поддерево,
- обработать корень,
- обойти правое поддерево.

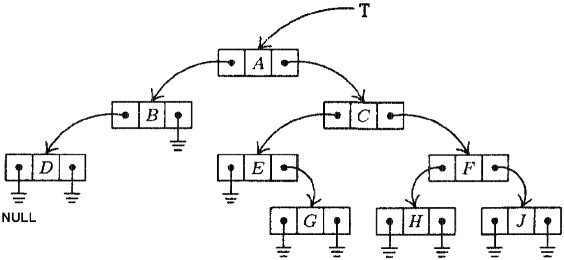
Порядок обработки узлов дерева: `D B A E G C H F J`.



Способы обхода двоичного дерева

Обход двоичного дерева в *ширину*: узлы дерева обрабатываются «по уровням» (уровень составляют все узлы, находящиеся на одинаковом расстоянии от корня).

Порядок обработки узлов дерева: `A B C D E F G H J`.



Рекурсивные реализации соответствующих функций на Си

```
void preorder (node * r) {
  if (r == NULL)
    return;
  if (r->info)
    printf ("%c", r->info);
  preorder (r->left);
  preorder (r->right);
}
```

Рекурсивные реализации соответствующих функций на Си

```
void postorder (node *r) {
  if (r == NULL)
    return;
  postorder (r->left);
  postorder (r->right);
  if (r->info)
    printf ("%c", r->info);
}
void inorder (node *r) {
  if (r == NULL)
    return;
  inorder (r->left);
  if (r->info)
    printf ("%c", r->info);
  inorder (r->right);
}
```

Нерекурсивная функция симметричного обхода

`r` — указатель на корень дерева;

`t` — указатель на корень обрабатываемого (текущего) поддерева;

`stack` — массив,на котором моделируется стек;

`depth` — глубина стека;

`top` — указатель вершины стека.

Стек требуется для ручного сохранения параметров функции, локальных переменных и точки возврата (если рекурсивных вызовов функции несколько).

В функции `inorder` нет локальных переменных, а второй из двух рекурсивных вызовов хвостовой, что позволяет не сохранять его параметры в стеке.
Поэтому сохраняется только параметр функции.

Нерекурсивная функция симметричного обхода. Алгоритм

1. Инициализация. Сделать стек пустым, т.е. затолкнуть `NULL` на дно стека: `stack[0] = NULL;` установить указатель стека на дно стека: `top = 0;` установить указатель `t` на корень дерева: `t = r`.
2. Конец ветви. Если `t == NULL`, перейти к 4.
3. Продолжение ветви. Затолкнуть `t` в стек: `stack[++top] = t;` установить `t = t->left` и вернуться к шагу 2.
4. К обработке правой ветви. Вытолкнуть верхний элемент стека в `t`: `t = stack[top]; top--;` Если `t == NULL`, выполнение алгоритма прекращается, иначе обработать данные узла, на который указывает `t`, и перейти к шагу 5.
5. Начало обработки правой ветви. Установить `t = t->right` и вернуться к шагу 2.

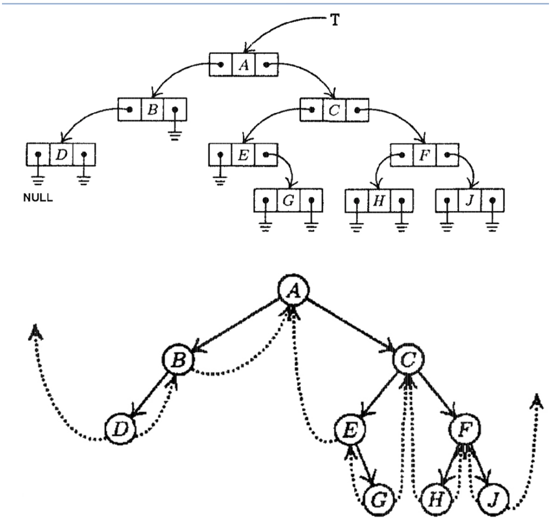
Нерекурсивная функция симметричного обхода. Код

```
int inorder (node *r, char *order) {
  node *t = r, *stack[depth];   // depth = ?
  int top = 0, i = 0;
  if (!t)
    return 0;
  stack[0] = NULL;                      // 1
  while (1) {
    while (t) {                         // 2
      stack[++top] = t;                 // 3
      t = t->left;
    }
<...>
```

Нерекурсивная функция симметричного обхода. Код

```
<...>
    t = stack[top--];          // 4
    if (t) {
      order[i++] = t->info;    // обработка
      t = t->right;            // 5
    } else                     // t == NULL
      break;                   // 4
  }
  return i;
}
```

Прошитое двоичное дерево



Рассмотрим двоичное дерево на верхнем рисунке. У этого дерева нулевых указателей, больше, чем ненулевых: 10 против 8. Это — типичный случай.

Будем записывать вместо нулевых указателей указатели на родителей (или более далеких предков) соответствующих узлов (такие указатели называются нитями). Это позволит при обходе дерева не использовать стек.

Прошитое двоичное дерево. Описание узла

```c
typedef struct bin_tree {
  char info;
  struct bin_tree *left;
  struct bin_tree *right;
  char left_tag;
  char right_tag;
} threaded_node;
```

Нити устанавливаются таким образом, чтобы указывать на предшественников (левые нити) или последователей (правые нити) текущего узла при соответствующем обходе дерева.

Обычное дерево	Прошитое дерево
`P->left == NULL`	`P->left_tag == 1, P->left == P_pred_in`
`P->left == Q`	`P->left_tag == 0, P->left == Q`
`P->right == NULL`	`P->right_tag == 1, P->right == P_next_in`
`P->right == Q`	`P->right_tag == 0, P->right == Q`

Прошитое двоичное дерево. Симметричный обход

```
threaded_node * next_in (threaded_node *p) {
  threaded_node *q = p->right;
  if (p->right_tag == 1)
    return q;
  while (q->left_tag == 0) //q != NULL
    q = q->left;           //q->left != NULL
  return q;
}
```

Функция **next_in** фактически реализует симметричный обход дерева, так как позволяет для произвольного узла дерева **P** найти следующий элемент **P_next_in**. Многократно применяя эту функцию, можно вычислить топологический порядок узлов двоичного дерева, соответствующий симметричному обходу.

Аналогичным образом можно построить функции, вычисляющие следующий узел дерева в прямом или обратном порядке обхода.

Прошитое двоичное дерево. Симметричный обход

```
threaded_node * next_in (threaded_node *p) {
  threaded_node *q = p->right;
  if (p->right_tag == 1)
    return q;
  while (q->left_tag == 0) //q != NULL
    q = q->left;           //q->left != NULL
  return q;
}
```

С помощью обычного представления невозможно для произвольного узла P вычислить `P_next_in`, не вычисляя всей последовательности узлов.

Функции `next_in` не требуется стек ни в явной, ни в неявной (рекурсия) форме.

Прошитое двоичное дерево. Симметричный обход

Если p — произвольно выбранный узел дерева,то следующий фрагмент функции `next_in`:

```
q = p->right;
if (p->right_tag == 1)
  return q;
```

выполняется только один раз.

Обход прошитого дерева выполняется быстрее, так как для него не нужны операции со стеком.

Для `inorder` требуется больше памяти, чем для `next_in`, из-за массива `stack[depth]` (пропорционально высоте дерева).

Нельзя допускать переполнение стека деревьев (массив выделяется с запасом либо используется реализация стека с динамическим перевыделением памяти).

Прошитое двоичное дерево. Заголовок

В функции **inorder** используется указатель **r** на корень двоичного дерева. Желательно, применив функцию **next_in** к корню **r**, получить указатель на самый первый узел дерева для выбранного порядка обхода. Для этого к дереву добавляется еще один узел — заголовок дерева (header).

```
header->left_tag = 0;
header->right_tag = 0;
header->left = r;
header->right = header;
```