Высшая школа экономики
Факультет компьютерных наук
Департамент программной инженерии

Алгоритмы и алгоритмические языки
Лекция 10

2 декабря 2025 г.

Топологическая сортировка на Си

```
#include <stdio.h>
#include <stdlib.h>
typedef struct ldr { /* дескриптор ведущего узла */
  char key;
  int count;
  struct ldr *next;
  struct trl *trail;
} leader;
typedef struct trl { /* дескриптор ведомого узла */
  struct ldr *id;
  struct trl *next;
} trailer;

leader *head, *tail; /* два вспомогательных узла */
int lnum; /* счётчик ведущих узлов */
```

Топологическая сортировка на Си: поиск по ключу

```
leader *find (char w) {
  leader *h = head;
  /* барьер на случай отсутствия w */
  tail->key = w;
  while (h->key != w)
    h = h->next;
  if (h == tail) {
    /* генерация нового ведущего узла */
    tail = malloc (sizeof (leader));
    /* старый tail становится новым элементом списка */
    lnum++;
    h->count = 0;
    h->trail = NULL;
    h->next = tail;
  }
  return h;
}
```

Топологическая сортировка на Си: формирование списка

```
void init_list() {
  leader *p, *q;
  trailer *t;
  char x, y;

  head = (leader *) malloc (sizeof (leader));
  tail = head;
  lnum = 0;                  /* начальная установка */
  while (1) {
    if (scanf ("%c␣%c", &x, &y) != 2)
      break;
    /* включение пары в список */
    p = find (x);
    q = find (y);
    <...>
```

Топологическая сортировка на Си: формирование списка

```
      <...>
      /* коррекция списка */
      t = malloc (sizeof (trailer));
      t->id = q;
      t->next = p->trail;
      p->trail = t;
      q->count += 1;
    }
}
```

Топологическая сортировка на Си: новый список

```
void sort_list() {
  leader *p, *q;
  trailer *t;
  /*  В выходной список включаются все узлы с count == 0 */
  p = head;
  head = NULL; /* голова выходного списка */
  while (p != tail) {
    q = p;
    p = q->next;
    if (q->count == 0) {
      /* включение q в выходной список */
      q->next = head;
      head = q;
    }
  }
<...>
```

Топологическая сортировка на Си: новый список

```
q = head; /* есть ведущий узел -> head != NULL */
while (q != NULL) {
  printf ("%c\n", q->key);
  lnum--;
  t = q->trail;
  q = q->next;
  while (t != NULL) {
    p = t->id;
    p->count -= 1;
    if (p->count == 0) {
      p->next = q;  // достаточно для
      q = p;        // правильной сортировки
    }
    t = t->next;
  }
}
/* lnum == 0 */
```

Топологическая сортировка на Си: основная функция

```
int main (void) {
  init_list ();
  sort_list ();
  return 0;
}
```

Дома. Что поменяется, если узлы идентифицируются не одним символом, а именем (строкой)? Сделайте нужные изменения в коде. Добавьте определение циклов в исходных данных.

Сортировка: постановка задачи

Сортировка — это упорядочение наборов однотипных данных, для которых определено отношение линейного порядка (например, $<$, «меньше») по возрастанию или по убыванию. Здесь будут рассматриваться целочисленные данные и отношение порядка $<$.

Различают *внешнюю* и *внутреннюю* сортировку. Рассматривается только внутренняя сортировка: сортируемый массив находится в основной памяти компьютера. Внешняя сортировка применяется к записям на внешних файлах.

Сортировка: что есть в Си

```
#include <stdlib.h>
void qsort (void *buf, size_t num, size_t size,
      int(*compare)(const void *, const void *));
```

Функция **qsort** сортирует (по возрастанию) массив с указателем **buf**, используя алгоритм быстрой сортировки Ч.Э.Р. Хоара, который считается одним из лучших алгоритмов сортировки общего назначения.

Параметр **num** задает количество элементов массива **buf**, параметр **size** — размер (в байтах) элемента массива **buf**. Параметр **int(*compare)(const void *,const void *)** задаёт правило сравнения элементов массива **num**. Функция сравнивает аргументы и возвращает:

- целое < 0, если $arg1 < arg2$,
- целое = 0, если $arg1 = arg2$,
- целое > 0, если $arg1 > arg2$.

Простейший алгоритм сортировки

Сведение сортировки к задаче нахождения максимального (минимального) из n чисел. Нахождение максимума n чисел (n сравнений). Числа содержатся в массиве **`int a[n];`**

```
max = a[0];
for (i = 1; i < n; i++)
  if (a[i] > max)
    max = a[i];
```

Алгоритм сортировки: находим максимальное из n чисел, получаем последний элемент отсортированного массива (n сравнений); находим максимальное из $n-1$ оставшихся чисел, получаем предпоследний элемент отсортированного массива (еще $n-1$ сравнений); и так далее.

Общее количество сравнений: $1+2+\ldots+n-1+n = n(n-1)/2$. Сложность алгоритма $O(n^2)$.

Три общих метода внутренней сортировки

- сортировка *обменами*: рассматриваются соседние элементы сортируемого массива и при необходимости меняются местами;
- сортировка *выборкой*: идея описана на предыдущем слайде;
- сортировка *вставками*: сначала сортируются два элемента массива, потом выбирается третий элемент и вставляется в нужную позицию относительно первых двух и т.д.

Сортировка обменами (пузырьком)

Общее количество сравнений (действий): $n(n-1)/2$, так как внешний цикл выполняется $(n-1)$ раз, а внутренний — в среднем $n/2$ раза.

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

Сортировка вставками

Количество сравнений зависит от степени перемешанности массива a. Если массив a уже отсортирован, количество сравнений равно $n - 1$. Если массив a отсортирован в обратном порядке (наихудший случай), количество сравнений имеет порядок n^2.

```
void insert_sort (int *a, int n) {
  int i, j, tmp;

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

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 сохранять в автоматической памяти (стеке) локальные переменные и параметры.

Оценка сложности алгоритмов сортировки

Теорема. Для любого алгоритма S внутренней сортировки сравнением массива из n элементов количество сравнений $C_S \geqslant O(n \log_2 n)$.

Доказательство. Сначала покажем, что

$$C_S \geqslant \log_2(n!) \tag{1}$$

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.

Оценка сложности алгоритмов сортировки

Теорема. Для любого алгоритма S внутренней сортировки сравнением массива из n элементов количество сравнений $C_S \geqslant O(n \log_2 n)$.

Доказательство. Сначала покажем, что

$$C_S \geqslant \log_2(n!) \tag{1}$$

Для высоты h_m двоичного дерева с m листьями имеет место оценка: $h_m \geqslant \log_2 m$.

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 искомую оценку (1).

Оценка сложности алгоритмов сортировки

Далее, применим к $\log_2 n!$ формулу Стирлинга

$$n! = \sqrt{2\pi n} n^n e^{-n} e^{\theta(n)},$$

где $|\theta(n)| \leqslant \frac{1}{12n}$. Подставляя и логарифмируя, имеем

$$\log_2 n! = \frac{1}{2} \log_2 2\pi n + n \log_2 n - n + \theta(n),$$

$$\log_2 n! \geqslant O(n \log_2 n).$$

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,

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

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;
```

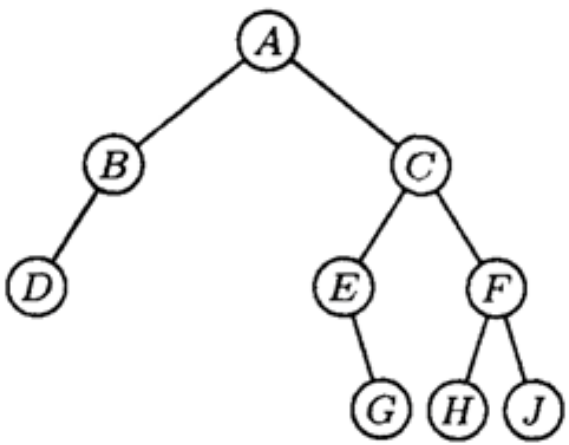


Рис. 1. Двоичное дерево

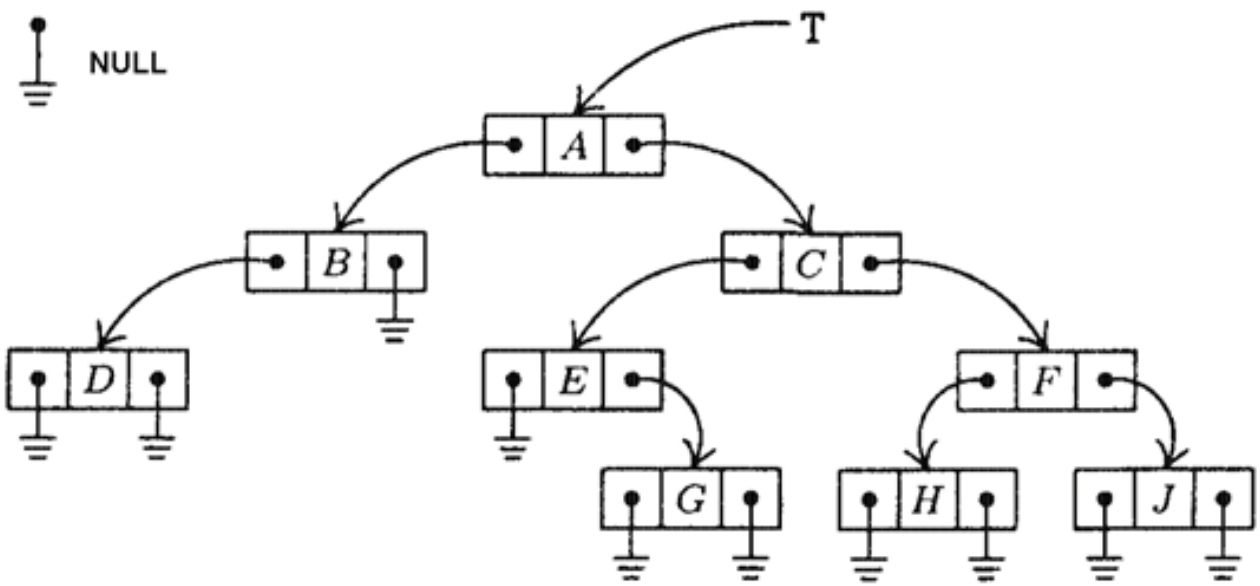


Рис. 2 Представление дерева с рис.1 в компьютере.

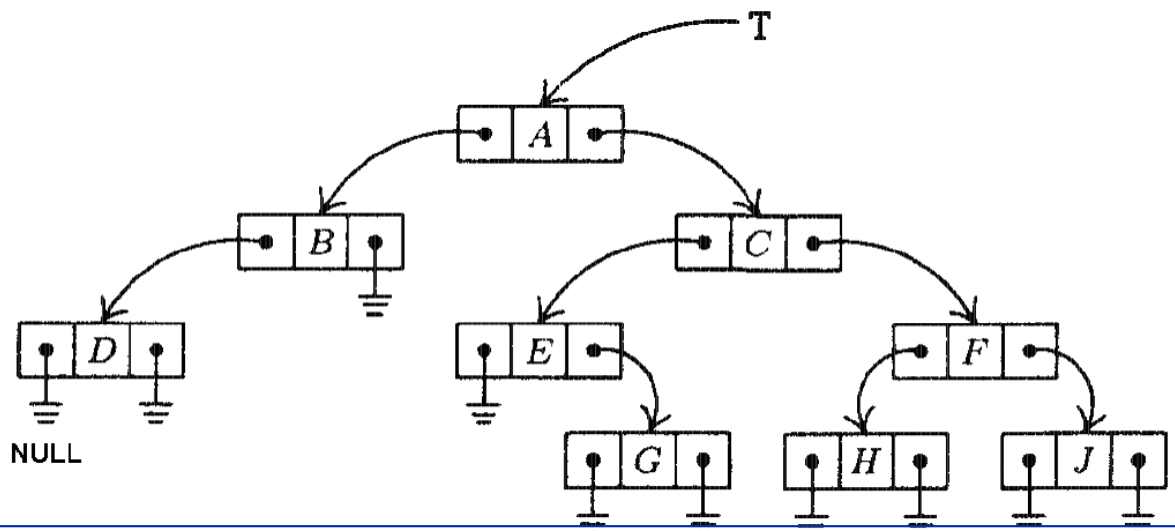
Способы обхода двоичного дерева

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.



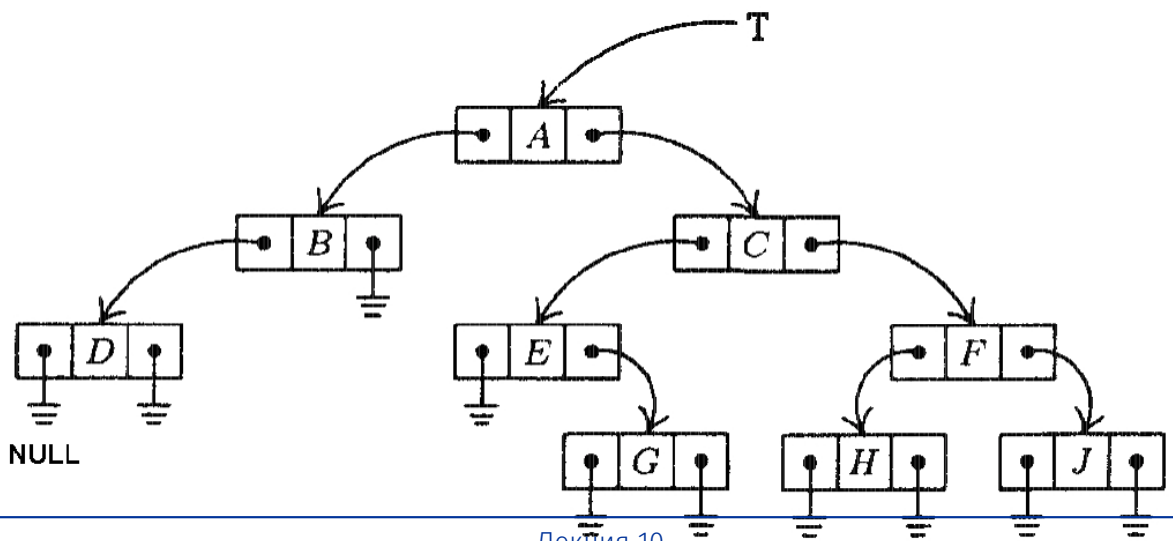
Способы обхода двоичного дерева

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.

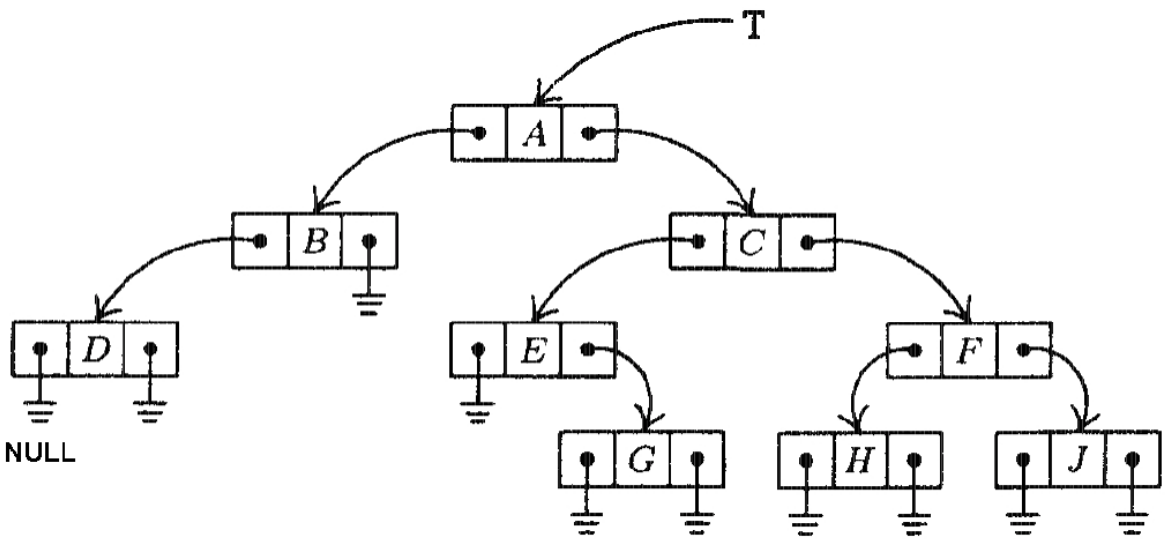


Способы обхода двоичного дерева

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

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

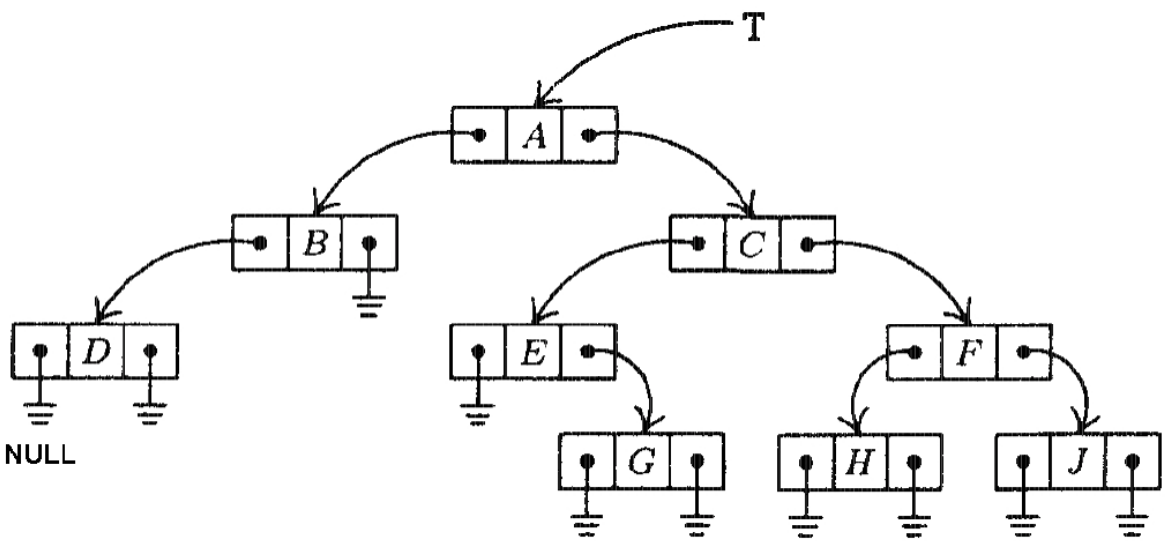
Порядок обработки узлов дерева: D B A E G C H F J.



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.

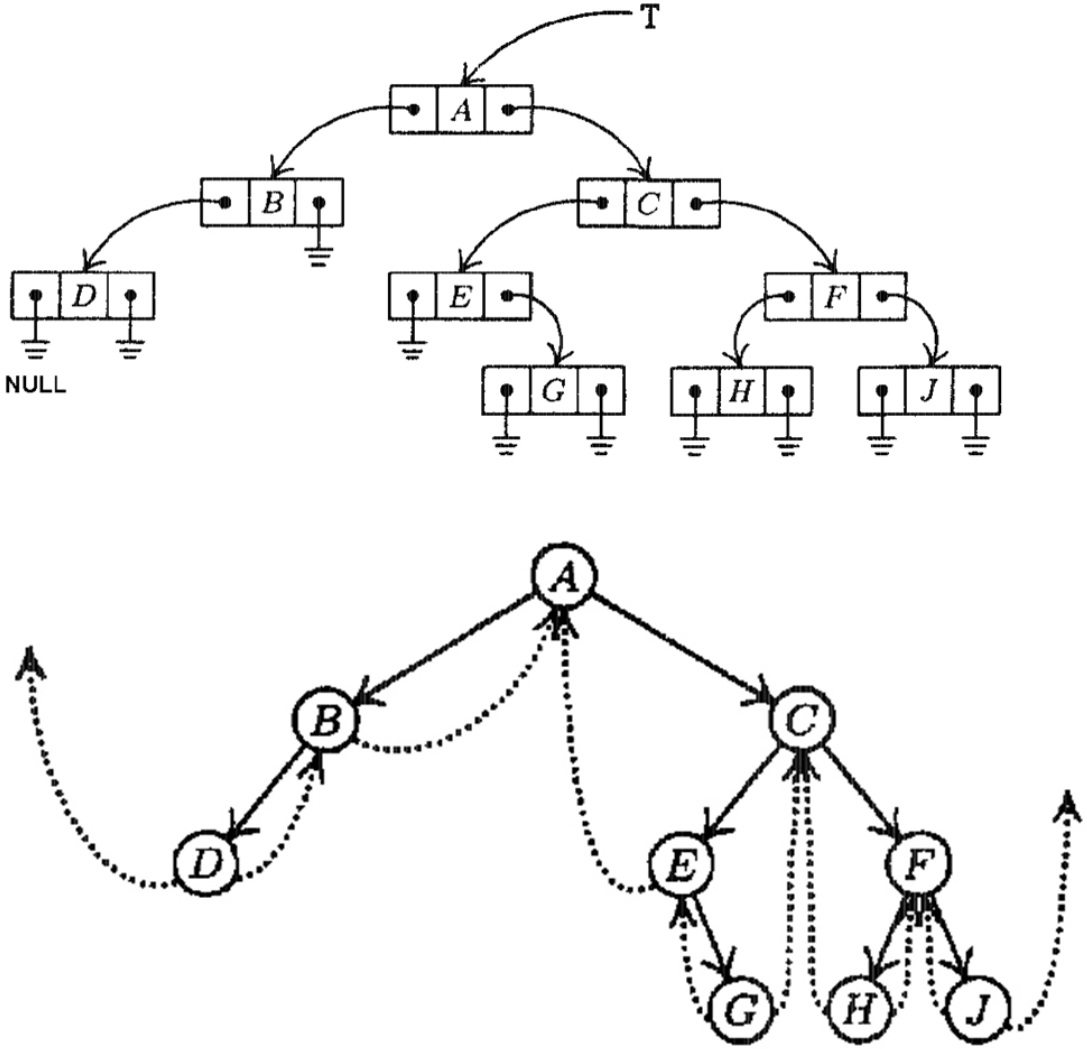
Нерекурсивная функция симметричного обхода. Код

```
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

```
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.

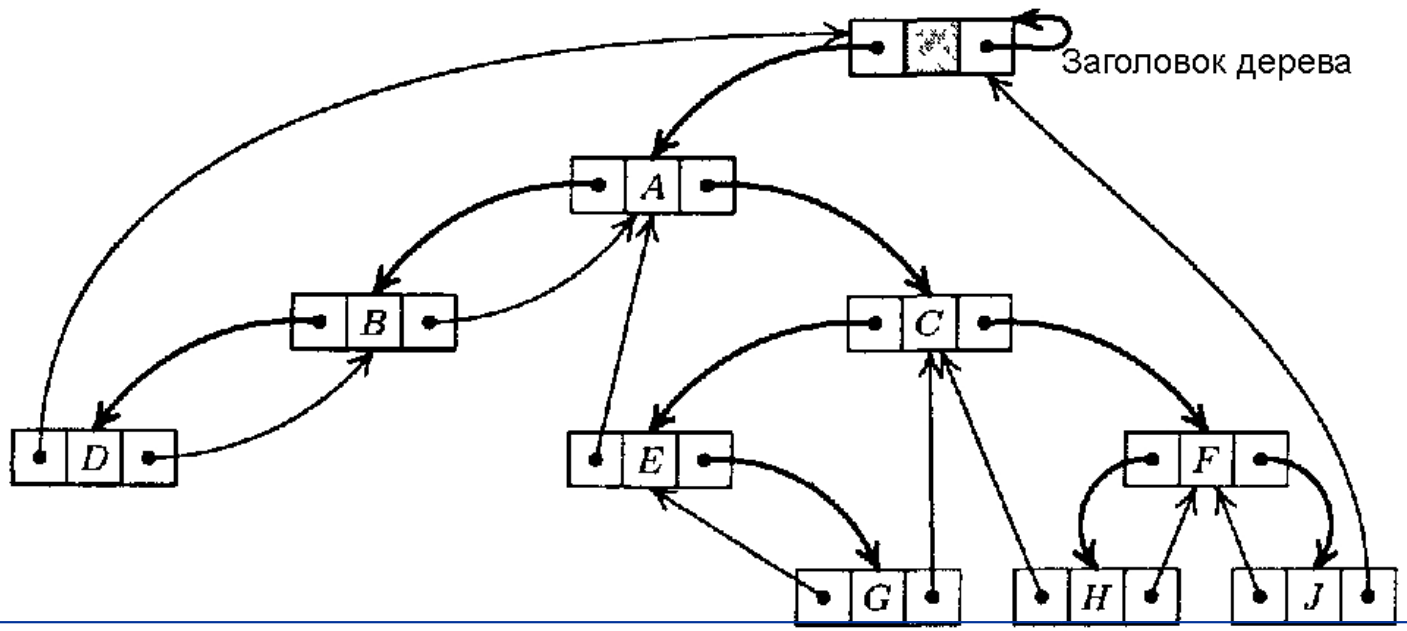
Для `inorder` требуется больше памяти, чем для `next_in`, из-за массива `stack[depth]` (пропорционально высоте дерева).

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;
```



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). Также предоставляется один или несколько вариантов *обхода* хранилища (посещения всех данных).

Варианты решения — деревья поиска, хеширование.

Двоичные деревья поиска

```
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] < key[x]`,

если y находится в правом поддереве узла x, то

`key[y] > key[x]`.

Возможно хранение дублирующихся ключей (нестрогие неравенства), не рассматривающееся в данном курсе.

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==k` (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 (k < root->key)
    return Btsearch (root->left, k);
  else
    return Btsearch (root->right, k);
}
```

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==k` (если такой узел есть), либо пустой указатель `NULL`.

```
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;
}
```

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 **key**.

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

Время выполнения $O(h)$, где h — высота дерева.

Двоичные деревья поиска: следующий элемент

На входе: указатель **node** на узел поддерева.

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

```
struct BT_node *Btsucc (struct BT_node *node) {
  struct BT_node *p = node, *q;
  /*  1 случай: правое поддерево узла не пусто. */
  if (p->right)
    return Btmin (p->right);
  /* 2 случай: правое поддерево узла пусто, идём по родителям до тех пор, пока
не найдём родителя, для которого наше поддерево левое.  */
  q = p->parent;
  while (q && p == q->right) {
    p = q;
    q = q->parent;
  }
  return q;
}
```

Время выполнения $O(h)$, где h — высота дерева.

Связь с симметричным порядком обхода и прошитыми деревьями.

Двоичные деревья поиска: вставка

На входе: указатель **root** на корень дерева и указатель **node** на новый уже инициализированный узел.

```
struct BT_node * Btinsert (struct BT_node *root,
                           struct BT_node *node) {
  struct BT_node *p, *q;
  p = root, q = NULL;
  while (p) {
    q = p;
    p = (node->key < p->key) ? p->left : p->right;
  }
  node->parent = q;
  if (q == NULL)
    root = node;
  else if (node->key < q->key)
    q->left = node;
  else
    q->right = node;
  return root;
}
```

Двоичные деревья поиска: удаление

На входе: указатель на корень `root` дерева T и указатель на узел n дерева T.
На выходе: двоичное дерево T с удаленным узлом n (ключи нового дерева по-прежнему упорядочены).

Необходимо рассмотреть три случая: (1) у узла n нет детей (листовой узел); (2) у узла n только один ребенок; (3) у узла n два ребенка.

1. просто удаляем узел n;
2. вырезаем узел n, соединив единственного ребенка узла n с родителем узла n;
3. находим `succ(n)` и удаляем его, поместив ключ `succ(n)` в узел n.

Двоичные деревья поиска: удаление

Шаг 1: если у n меньше двух детей, удаляем n, иначе удаляем $succ(n)$; устанавливаем указатель y на удаляемый узел.

Шаг 2: находим ребенка удаляемого узла (ребенка либо нет, либо он единственный) и устанавливаем на него указатель $\widehat{x}$.

Шаг 3: подвешиваем ребенка y (указатель x) к родителю y;
если у y нет родителя, новым корнем дерева становится x;
устанавливаем в соответствующем поле родителя указатель на x, полностью исключая y из дерева.

Шаг 4: если удаляемый узел $succ(n)$, заменяем данные узла n на данные узла $succ(n)$.

Двоичные деревья поиска: удаление

```
struct BT_node * BTdelete (struct BT_node **root,
                           struct BT_node *n) {
  struct BT_node *x, *y;
  /* Шаг 1: y -- указатель на удаляемый узел n */
  y = (! n->left || ! n->right) ? n : BT_succ (n);
  /* Шаг 2: x -- указатель на ребенка y либо NULL */
  x = y->left ? y->left : y->right;
  /* Шаг 3: если x есть, вырезаем y из родителей */
  if (x)
    x->parent = y->parent;
  /* Шаг 3: если у y нет родителя, x -- новый корень */
  if (! y->parent)
    *root = x;
  else {
    /* Шаг 3: x присоединяется к y->parent */
    if (y == y->parent->left)
      y->parent->left = x;
    else
      y->parent->right = x;
```

Двоичные деревья поиска: удаление

```
struct BT_node * BTdelete (struct BT_node **root,
                           struct BT_node *n) {
  struct BT_node *x, *y;
        <...>
  /* Шаг 4: если удалялся не узел n, а succ(n),
  необходимо заменить данные узла n на данные узла
succ(n) */
  if (y != n)
    n->key = y->key;

  /* функция возвращает указатель удаленного узла, что
даёт возможность использовать этот узел в других
структурах либо очистить занимаемую им память */
  return y;
}
```

Время выполнения $O(h)$, где h — высота дерева.