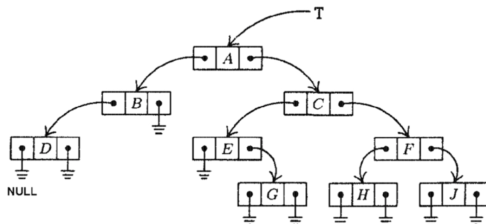
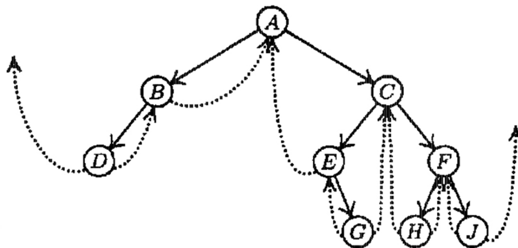
Московский государственный университет им. М. В. Ломоносова
Факультет вычислительной математики и кибернетики

Алгоритмы и алгоритмические языки
Лекция 21

23 ноября 2019 г.

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.

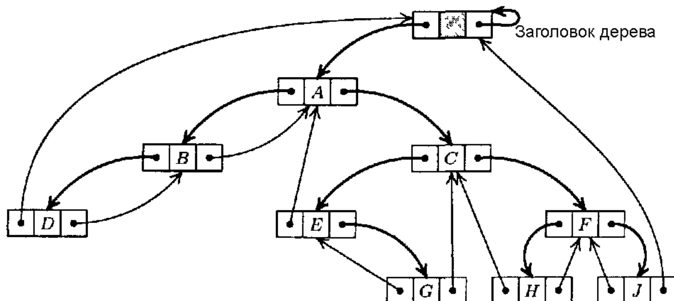
Для `inorder` требуется больше памяти, чем для `next_in`, из-за массива `stack[depth]` (пропорционально высоте дерева).

Нельзя допускать переполнение стека деревьев (массив выделяется с запасом либо используется реализация стека с динамическим выделением 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 узел, у которого есть значение ключа, а все поля с указателями имеют значение NULL.

```
struct BT_node * Btinsert (struct BT_node *root, struct BT_nod
  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 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),
необходимозаменить 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.