Алгоритмы и алгоритмические языки

Лекция 25

11 декабря 2019 г.

Хеширование с открытой адресацией

- Все записи хранятся в самой хеш-таблице: каждая ячейка таблицы (массива длины m) содержит либо хранимый элемент, либо **`NULL`**. Указатели вообще не используются, что приводит к сохранению места и ускорению поиска.
- Таким образом, *коэффициент заполнения* $\alpha = n/m$ не больше 1.
- ***Поиск*** (**search**): мы *определенным образом* просматриваем элементы таблицы, пока не найдем искомый или не убедимся, что искомый элемент отсутствует.
- Просматриваются не все элементы (иначе это был бы последовательный поиск), а только некоторые согласно значению хеш-функции, которая в этом случае имеет два аргумента – ключ и «номер попытки»:

$$hash: U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}.$$

- Функцию *hash* нужно выбрать такой, чтобы в последовательности проб $\langle hash(k, 0), hash(k, 1), \ldots, hash(k, m-1)\rangle$ каждый номер ячейки 0, 1, …, m – 1 встретился только один раз.
- Если при поиске мы добираемся до ячейки, содержащей **`NULL`**, можно быть уверенным, что элемент с данным ключом отсутствует (иначе он попал бы в эту ячейку).

Хеширование с открытой адресацией: программы

```
#define m 1999
struct htype {
  int key;          /* ключ */
  int val;          /* значение элемента данных */
} *index[m];

/* Поиск элемента */
struct htype *search (int k) {
  int i = 0, j;

  do {
    j = hash (k, i);
    if (index[j] && index[j]->key == k)
      return index[j];
  } while (index[j] && ++i < m);
  return NULL;
}
```

Хеширование с открытой адресацией: программы

```
/* Добавление элемента */
int insert (int k, int v) {
  int i = 0, j;

  do {
    j = hash (k, i);
    if (index[j] && index[j]->key == k) {
      index[j]->val = v;
      return j;
    }
  } while (index[j] && ++i < m);
  /* Таблица может оказаться заполненной */
  if (i == m)
    return -1; /* Или расширим index */
  index[j] = new ();
  index[j]->key = k, index[j]->val = v;
  return j;
}
```

Хеширование с открытой адресацией: программы

```
/* Внутренний поиск: вернем индекс массива */
static int search_internal (int k) {
 int i = 0, j;

  do {
    j = hash (k, i);
    if (index[j] && index[j]->key == k)
      return j;
  } while (index[j] && ++i < m);
  return -1;
}
/* Внешний поиск легко реализуется через внутренний */
struct htype *search (int k) {
  int j = search_internal (k);
  return j >= 0 ? index[j] : NULL;
}
```

Хеширование с открытой адресацией: программы

```
/* Удаление элемента */
void delete (int k) {
  int j;

  j = search_internal (k);
  if (j < 0)
    return;
  /* Нельзя писать index[j] = NULL!
     Будут потеряны ключи, возможно, находящиеся
     за удаляемым ключом (с тем же хешем). */
  ???
}
```

Хеширование с открытой адресацией: программы

```
#define SHADOW ((void *) (intptr_t) 1)

/* Удаление элемента */
void delete (int k) {
  int j;

  j = search_internal (k);
  if (j < 0)
    return;
  /* Нельзя писать index[j] = NULL! */
  free (index[j]);
  index[j] = SHADOW;
}
```

Хеширование с открытой адресацией: программы

```
#define SHADOW ((void *) (intptr_t) 1)
#define ISEMPTY(el) ((!el) || (el) == SHADOW)

static int search_internal (int k) {
 int i = 0, j;

  do {
    j = hash (k, i);
    if (!ISEMPTY (index[j]) && index[j]->key == k)
      return j;
   } while (index[j] && ++i < m);
  return -1;
}
```

Хеширование с открытой адресацией: программы

```
#define SHADOW ((void *) (intptr_t) 1)
#define ISEMPTY(el) ((!el) || (el) == SHADOW)

/* Добавление элемента */
int insert (int k, int v) {
  int i = 0, j;

  do {
    j = hash (k, i);
     if (! ISEMPTY (index[j]) && index[j]->key == k) {
      index[j]->val = v;
      return j;
    }
} while (! ISEMPTY (index[j]) && ++i < m);
/* Таблица может оказаться заполненной (много вставок/удалений) */
  if (i == m)
    return -1; /* Или расширим index */
  index[j] = new ();
  index[j]->key = k, index[j]->val = v;
  return j;
}
```

Хеш-функции для открытой адресации

- *Линейная последовательность* проб.
  Пусть $hash'$: $U \to \{0, 1, \ldots, m - 1\}$ – обычная хеш-функция.
  Функция $hash(k, i) = (hash'(k) + i) \bmod m$
  определяет *линейную последовательность проб*.
- При линейной последовательности проб начинают с ячейки *index*[*h′*(*k*)], а потом перебирают ячейки таблицы подряд: *index*[*h′*(*k*) + 1], *index*[*h′*(*k*) + 2], … (после *index*[*m* – 1] переходят к *index*[*0*]).
- Существует лишь *m* различных последовательностей проб, т.к. каждая последовательность однозначно определяется своим первым элементом.

Хеш-функции для открытой адресации

- Серьезный недостаток – тенденция к образованию *кластеров* (длинных последовательностей занятых ячеек, идущих подряд), что удлиняет поиск:
  - Если в таблице все четные ячейки заняты, а нечетные ячейки свободны, то среднее число проб при поиске отсутствующего элемента равно 1,5.
  - Если же те же $m/2$ занятых ячеек идут подряд, то среднее число проб равно $(m/2)/2 = m/4$.
- Причины образования кластеров: если k заполненных ячеек идут подряд, то:
  - вероятность того, что при очередной вставке в таблицу будет использована ячейка, непосредственно следующая за ними, есть $(k+1)/m$ (пропорционально «толщине слоя»),
  - вероятность использования конкретной ячейки, предшественница которой тоже свободна, всего лишь $1/m$.
- Таким образом, хеширование с использованием линейной последовательности проб далеко не равномерное.
- Возможное улучшение: добавляем не 1, а константу c, взаимно простую с m (для полного обхода таблицы).

Хеш-функции для открытой адресации

- *Квадратичная последовательность* проб:
  $hash(k, i) = (hash'(k) + c_1 \cdot i + c_2 \cdot i^2) \bmod m$,
  c_1 и $c_2 \neq 0$ – константы.
- Пробы начинаются с ячейки *index*[$h'(k)$], а потом ячейки просматриваются не подряд, а по более сложному закону. Метод работает значительно лучше, чем линейный.
- Чтобы при просмотре таблицы *index* использовались все ее ячейки, значения m, c_1 и c_2 следует брать не произвольными, а подбирать специально. Если обе константы равны единице:
  - находим $i \leftarrow hash'(k)$; полагаем $j \leftarrow 0$;
  - проверяем *index*[i]:

    если она свободна, заносим в нее запись и выходим из алгоритма,

    если нет – полагаем $j \leftarrow (j + 1) \bmod m$, $i \leftarrow (i + j) \bmod m$ и повторяем текущий шаг.

Хеш-функции для открытой адресации

- *Двойное хеширование* – один из лучших методов открытой адресации.

  $hash(k, i) = (h_1(k) + i\, h_2(k)) \bmod m,$

  где $h_1(k)$ и $h_2(k)$ – обычные хеш-функции.
- Дополнительная хеш-функция $h_2(k)$ генерирует хеши, взаимно простые с m.
- Если основная и дополнительная функция существенно независимы (т.е. вероятность совпадения их хешей обратно пропорциональна квадрату m), то скучивания не происходит, а распределение ключей по таблице близко к случайному.
- *Оценки*. Среднее число проб для равномерного хеширования оценивается при успешном поиске как $\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$.

  При коэффициенте заполнения 50% среднее число проб для успешного поиска $\leq$ 1,387, а при 90% – $\leq$ 2,559.
- При поиске отсутствующего элемента и при добавлении нового элемента оценка среднего числа проб $\frac{1}{1-\alpha}$.

Хеширование других данных

- Хеширование идентификаторов в компиляторе

```
hashval_t
htab_hash_string (const PTR p)
{
  const unsigned char *str = (const unsigned char *) p;
  hashval_t r = 0;
  unsigned char c;

  while ((c = *str++) != 0)
    r = r * 67 + c - 113;

  return r;
}
```

- Хеширование ключа переменной длины: в GCC используется http://burtleburtle.net/bob/hash/evahash.html (если не отвечает, смотрите в web.archive.org)

Цифровой поиск

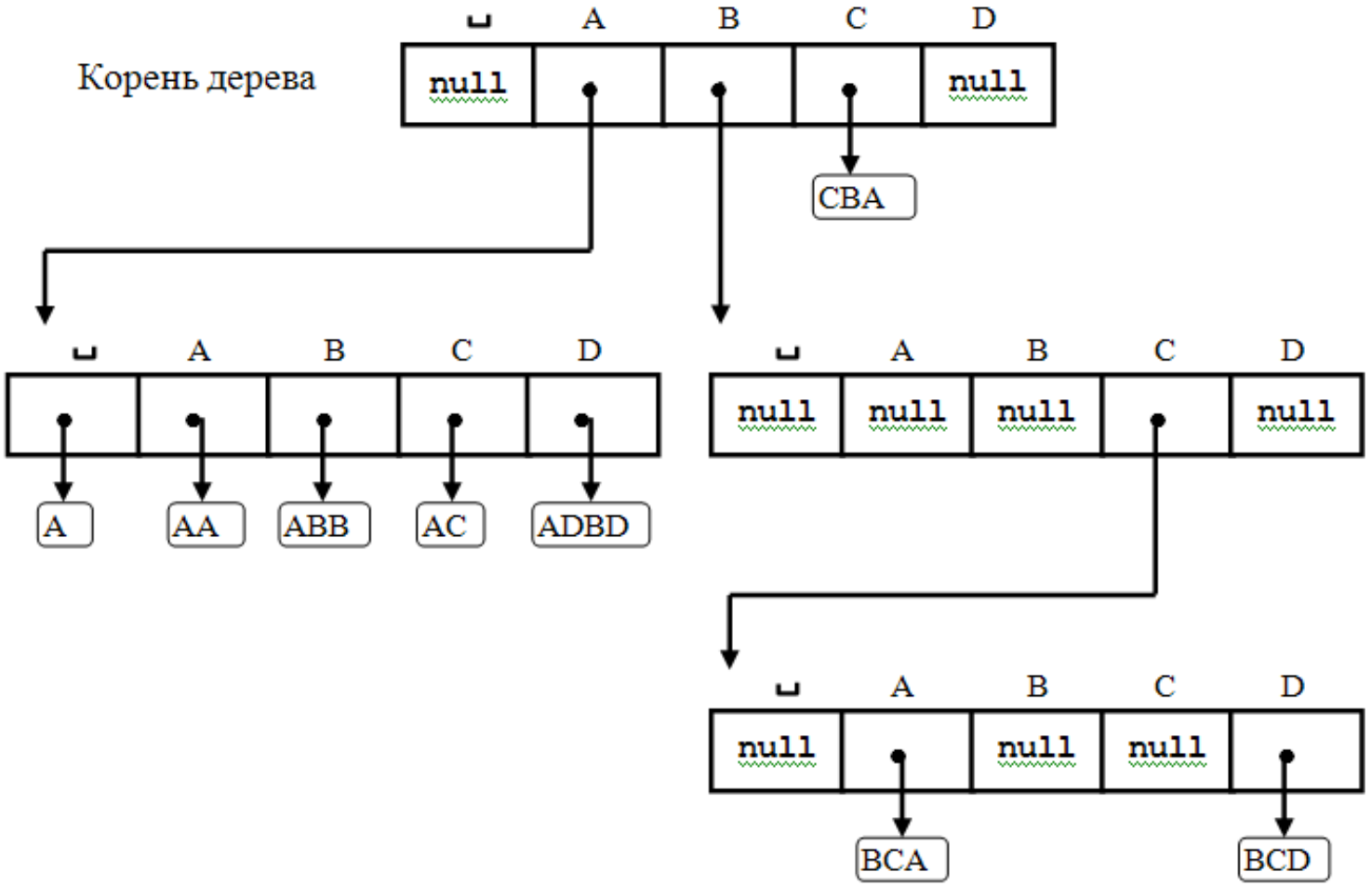
- *Цифровой поиск* – частный случай поиска заданной подстроки (*образца*) в длинной строке (*тексте*).
- *Примеры цифрового поиска*: поиск в словаре, в библиотечном каталоге и т.п., когда делается поиск по образцу в нескольких текстах (названиях книг, фамилиях авторов, текстах на вызванных сайтах и т.п.).
- Хороший пример – *словарь с высечками*, т.е. словарь, в котором обеспечен быстрый доступ к некоторым страницам (например, начальным страницам списков слов, начинающихся на очередную букву алфавита). Иногда используются *многоуровневые высечки*.
- При цифровом поиске ключи рассматриваются как последовательности символов рассматриваемого алфавита (в частности, цифр или букв). Ожидаемое число сравнений порядка $O(\log_m N)$, где m - число различных букв, используемых в словаре, N – мощность словаря. В худшем случае дерево содержит k уровней, где k – длина максимального слова.

Цифровой поиск

- *Пример*. Пусть множество используемых букв (алфавит) {*A*, *B*, *C*, *D*}. Мы добавим к алфавиту еще одну букву␣ (пробел). По определению слова *AA*, *AA*␣, *AA*␣␣ совпадают.
  Пусть {*A*, *AA*, *ABB*, *AC*, *ADBD*, *BCA*, *BCD*, *CBA*} – словарь (множество ключей).
- Построим *m*-ичное дерево, где *m* = 5 = |␣, *A*, *B*, *C*, *D* |. Следующая небольшая хитрость позволит иногда сократить поиск: если в словаре есть слово $a_1a_2a_3...a_k$ и первые *i* его букв (*i* < *k*) задают уникальное значение: комбинация $a_1...a_i$ встречается в словаре только один раз, то не нужно строить дерево для *j* > *i*, так как слово можно идентифицировать по первым *i* буквам.
- Очень важное обобщение цифрового поиска: таким же образом можно обрабатывать любые ключи, не привязываясь к байту (8 битам), который обычно используется для кодирования символов алфавита. Мы можем отсекать от ключа первые *m* бит, использовать 2^m-ичное разветвление, т.е. строить 2^m-ичное дерево поиска (на двоичных деревьях для разветвления берется один бит: *m* = 1).

Цифровой поиск

- Прямоугольниками изображены вершины дерева, в овалах – значения слов (ключей) и связанная с ним информация.
  Тем самым любая вершина дерева – массив из *m* элементов.
  Каждый элемент вершины содержит либо ссылку на другую вершину *m*-ичного дерева, либо на овал (ключ).



Цифровой поиск

- Иногда используют комбинации нескольких методов: цифровой поиск вначале, а затем переключение на поиск в последовательных таблицах.
  - Именно так мы и работаем со словарем с высечками: вначале на высечку, а затем либо последовательный поиск, либо дихотомический.
- Обычно предлагается пользоваться цифровым поиском, пока количество различных слов не меньше некоторого k, а затем переключаться на последовательные таблицы.
- Обобщения: поиск по неполным ключам, поиск по образцу.
- Варианты:
  - Не строить промежуточных узлов из одного разветвления, вместо этого хранить индекс следующего символа с нетривиальным разветвлением
  - Писать символы ключа на ребрах (“бор/сжатый бор”)

Цифровой поиск

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define M 5

typedef enum {word, node} tag_t;
struct record {
  char *key;
  int value;
};

struct tree {
  tag_t tag;
  union {
    struct record *r;
    struct tree *nodes[M+1];
  };  /* анонимное объединение */
};
```

Цифровой поиск: поиск элемента

```
static inline int ord (char c) {
  return c ? c - 'A' + 1 : 0; // ASCII-only
}

struct record *find (struct tree *t, char *key) {
  int i = 0;

  while (t) {
    switch (t->tag) {
    case word:
      for (; key[i]; i++)
        if (key[i] != t->r->key[i])
          return NULL;
      return t->r->key[i] ? NULL : t->r;
    case node:
      t = t->nodes[ord(key[i])];
      if (key[i])
        i++;
    }
  }
  return NULL;
}
```

Цифровой поиск: вставка – вспомогательные функции

```
struct record *make_record (char *key, int value) {
  struct record *r = malloc (sizeof (struct record));
  r->key = strdup (key);
  r->value = value;
  return r;
}

struct tree *make_from_record (struct record *r) {
  struct tree *t = malloc (sizeof (struct tree));
  t->tag = word;
  t->r = r;
  return t;
}

struct tree *make_word (char *key, int value) {
  return make_from_record (make_record (key, value));
}

struct tree *make_node (void) {
  struct tree *t = calloc (1, sizeof (struct tree));
  t->tag = node;
  return t;
}
```

Цифровой поиск: вставка элемента

```
struct tree *insert (struct tree *t, char *key, int value) {
  if (!t)
    return make_word (key, value);

  int i = 0;
  struct tree *root = t;

  /* skip all nodes */
  while (t->tag == node) {
    struct tree **p = &t->nodes[ord(key[i++])];
    if (!*p) {
      *p = make_word (key, value);
      return root;
    }
    t = *p;
  }

  /* all word skipped -- key exists, update value */
  if (i && !key[i - 1]) {
    t->r->value = value;
    return root;
  }
```

Цифровой поиск: вставка элемента

```
/* compare the remaining part */
int j = i;
for (; key[i]; i++)
  if (key[i] != t->r->key[i])
    break;

/* key already exists -- update value */
if (!key[i] && !t->r->key[i]) {
  t->r->value = value;
  return root;
}

/* turn t into a node */
struct record *other = t->r;
t->tag = node;
memset (t->nodes, 0, sizeof (t->nodes));
```

Цифровой поиск: *вставка элемента*

```
 /* make new nodes for remaining common prefix */
  for (; j < i; j++) {
    struct tree *p = make_node ();
    t->nodes[ord(key[j])] = p;
    t = p;
  }

  /* accommodate both other and new record */
  t->nodes[ord(other->key[i])]
    = make_from_record (other);
  t->nodes[ord(key[i])] = make_word (key, value);
  return root;
}
```

Цифровой поиск: печать элементов

```
void print (struct tree *t, char c) {
  static int level = 0;
  if (!t) {
    printf ("empty\n"); // also maybe if level == 0
    return;
  }
  for (int i = 0; i < level; i++)
    putchar (' ');
  if (level)
    printf ("%c: ", chr (c));
  if (t->tag == word) {
    printf ("word: %s %d\n", t->r->key, t->r->value);
  } else {
    printf ("node: ");
    for (int i = 0; i < M + 1; i++)
      if (t->nodes[i])
        printf ("%c ", chr(i));
    putchar ('\n');
    level++;
    for (int i = 0; i < M + 1; i++)
      if (t->nodes[i])
        print (t->nodes[i], i);
    level--;
  }
}
```

Резюме курса. Введение в теорию алгоритмов.

- Формализация алгоритмов: информация (кодирование), исполнители. Связь с задачей обработки информации (частично вычислимыми функциями).
- Возможность построения универсального вычислителя.
- Алгоритмическая неразрешимость.
- Эквивалентность формальных систем описания алгоритмов.

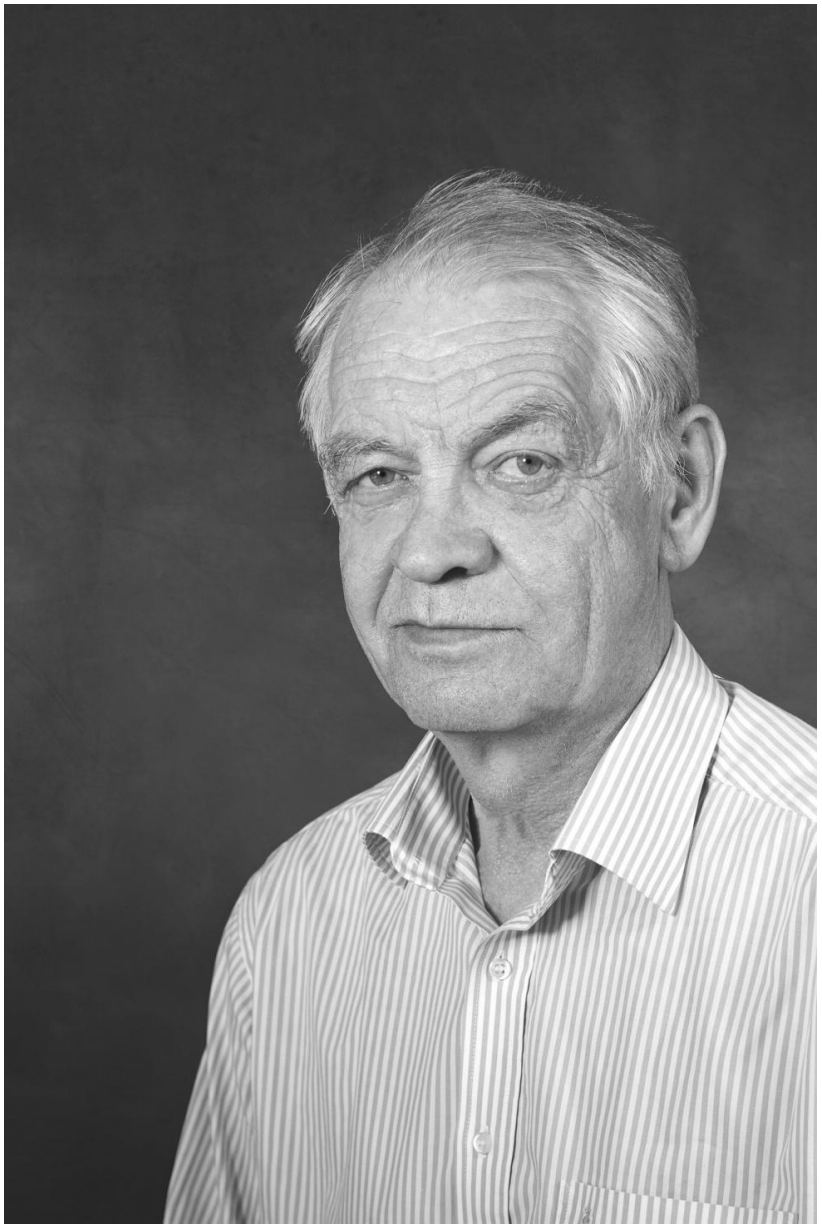
Резюме курса. Язык программирования Си.

- Си-машина. Устройство памяти.
- Особенности, требующие понимания
  - Приведение типов, в том числе integer promotion
  - Точки следования и побочные эффекты
  - «Ленивая» логика
  - Битовые операции
  - Оператор выбора
  - Индексация массивов
  - Строки
  - Адресная арифметика
  - Выравнивание структур
  - Рекурсия (в том числе хвостовая), inline
  - Вызовы по указателю
  - VLA-массивы
  - Динамическая память
  - Программы из нескольких файлов, заголовочные файлы, внешние переменные, компоновка…

Резюме курса. Алгоритмы и структуры данных.

- Списки (варианты «возвратить новый указатель» или «передать двойной указатель»)
- Стеки, очереди
- Сортировка (простые алгоритмы, быстрая сортировка, их сложность). Минимально возможная сложность сортировки.
- Двоичные деревья и их обходы. Замена рекурсии итерацией.

  Прошитые двоичные деревья («посчитать и сохранить» или «пересчитать каждый раз, не хранить»)
- Двоичные деревья поиска. Основные операции и высота дерева.
- Сбалансированные двоичные деревья поиска. Повороты как средство восстановления балансировки. Splay-деревья и чем они отличаются от классических сбалансированных.
  Возможное обобщение сбалансированных деревьев (ранги).
- Хеширование: как разрешать коллизии и как делать хорошие хеш-функции.
- Пирамида и пирамидальная сортировка.
- Дерево цифрового поиска.



Виктор Петрович Иванников
(1940-2016)