Курс «Алгоритмы и алгоритмические языки»

Лекция 22

Цифровой поиск

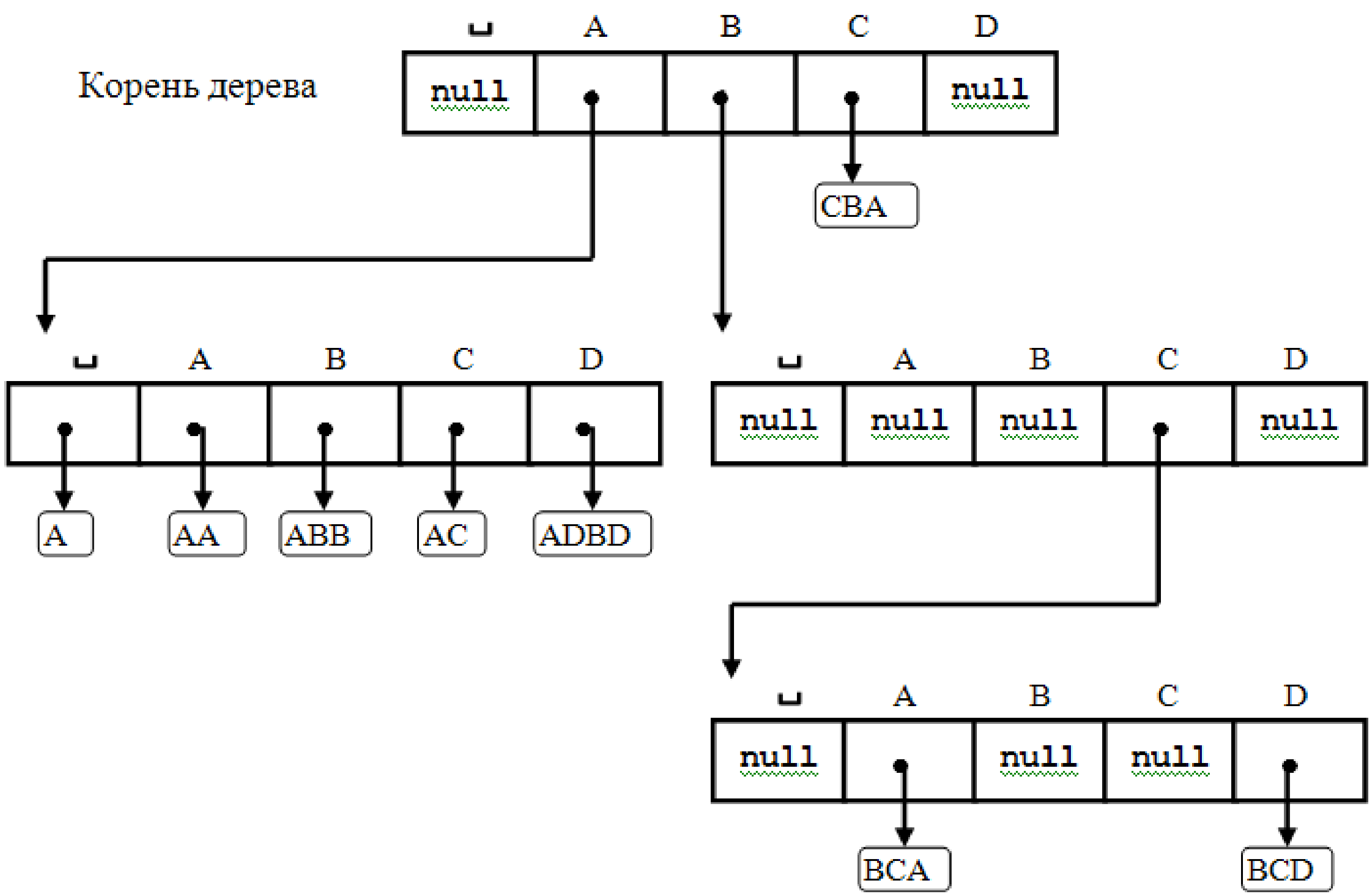
- *Цифровой поиск* – частный случай поиска заданной подстроки (*образца*) в длинной строке (*тексте*).
- *Примеры цифрового поиска*: поиск в словаре, в библиотечном каталоге и т.п., когда делается поиск по образцу в нескольких текстах (названиях книг, фамилиях авторов, текстах на вызванных сайтах и т.п.).
- 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

```
#include <stdlib.h>

#define M 20

typedef char key[M];
typedef enum {ident, node} tag;
struct record {
   key k;
   int value;
};
struct tree {
   tag t;
   union {
     struct record *r;
     struct tree *a[M+1];
   } u;
};
```

Цифровой поиск

```
static int ord (char c) {
   if (c == ' ')
     return 0;
   return c - 'A' + 1;
}

struct record *find (struct tree *p, key k) {
   int i = 0;
   while (p) {
     switch (p->t) {
       case ident:
         for (; i < M; i++)
           if (p->u.r->k[i] != k[i])
             return NULL;
         return p->u.r;
       case node:
         p = p->u.a[ord(k[i++])];
     }
   }
   return NULL;
}
```

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 переключение к последовательным таблицам.
- Обобщения: поиск по неполным ключам, поиск по образцу (лекция 23).

Алгоритмы перебора множеств

- Перестановка некоторого набора элементов – это упорядоченная последовательность из этих элементов. Например, множество {1, 2, 3} имеет 6 различных перестановок (1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1).
- Для любого множества из n элементов существует ровно $n! = 1 \cdot 2 \cdot ... \cdot n$ различных перестановок.
- ***Задача*** состоит в том, чтобы написать программу, которая выводит все перестановки множества $\{1,2,..., n\}$ в лексикографическом порядке (перестановки можно рассматривать как слова в алфавите $B = \{1, 2, ..., n\}$)

Рекурсивный алгоритм генерации перестановок

- Будем перебирать перестановки чисел {1, 2, ..., *n*} в глобальном массиве *a*[*n*], последовательно заполняя его числами 1, ..., *n* в различном порядке.

  Для перебора всех перестановок будем записывать на первое место в массиве *a* по очереди числа 1, ..., *n*, и для каждого числа рекурсивно вызывать функцию генерации перестановок оставшихся *n* – 1 чисел.

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

void PrintPerm (int *a, int n) {
  int i;
  for (i = 0; i < n; i++)
    printf("%3d", a[i]);
  printf("\n");
}
```

Рекурсивный алгоритм генерации перестановок

```
void GenPerm (int *a, int *b, int i, int n) {
  if (i == n)
    PrintPerm (a, n);
  } else {
    int j;
    for (j = 0; j < n; j++)
      if (b[j] == 0) {
        b[j] = 1;
        a[i] = j + 1;
        GenPerm (a, b, i+1, n);
        b[j]  = 0;
      }
  }
}
```

Рекурсивный алгоритм генерации перестановок

```
int main (void) {
  int *a, *b;
  int n;
  scanf ("%d", &n);
  a = (int *) malloc (n * sizeof(int));
  b = (int *) calloc (n * sizeof(int));
  GenPerm (a, b, 0, n); /* первый вызов генератора */
  free (a);
  free (b);
  return 0;
}
```

Нерекурсивный алгоритм генерации перестановок

- Задачу посещения (перебора) всех перестановок заданного множества можно свести к следующей задаче: по данной перестановке сгенерировать следующую за ней перестановку (например, в лексикографическом порядке). Один из первых алгоритмов решения этой задачи придумал индийский математик Пандит Нарайана еще в XIV веке.
- Алгоритм Нарайаны по любой данной перестановке из n элементов a_1 a_2 ... a_n генерирует следующую (в лексикографическом порядке) перестановку.

  Если алгоритм Нарайаны применить в цикле к исходной последовательности n элементов a_1 a_2 ... a_n, отсортированных так, что $a_1 \leq a_2 \leq ... \leq a_n$, то он сгенерирует все перестановки элементов множества $\{a_1\ a_2\ ...\ a_n\}$, в лексикографическом порядке.

Алгоритм Нарайаны

- Шаг 1. [Первая перестановка.] Составить перестановку $a_1\ a_2\ ...\ a_n$ ($a_1 < a_2 < ... < a_n$).

  Шаг 2. [Найти j, для которого $a_j < a_{j+1}$] Установить $j \leftarrow n - 1$. Если $a_j \geq a_{j+1}$, уменьшать j на 1 повторно, пока не выполнится условие $a_j < a_{j+1}$. Если окажется, что $j = 0$, завершить алгоритм.

  - На шаге 2 j является наименьшим индексом, для которого были посещены все перестановки, начиная с $a_1\ ...\ a_j$. Следовательно, лексикографически следующая перестановка увеличит значение a_j.

  Шаг 3. [Увеличить a_j] Установить $l \leftarrow n$. Если $a_j \geq a_l$, уменьшать l на 1, пока не выполняется условие $a_j < a_l$. Затем поменять местами $a_j \leftrightarrow a_l$.

  - Поскольку $a_{j+1} \geq ... \geq a_n$, элемент a_l является наименьшим элементом, который больше a_j, и который может следовать за $a_1\ ...\ a_{j-1}$ в перестановке. Перед заменой выполнялись отношения $a_{j+1} \geq ... \geq a_{l-1} \geq a_l \geq a_j \geq a_{l+1} \geq ... \geq a_n$, а после замены – выполняются $a_{j+1} \geq ... \geq a_{l-1} \geq a_j \geq a_l \geq a_{l+1} \geq ... \geq a_n$

  Шаг 4. [Обратить $a_{j+1}\ ...\ a_n$] Установить $k \leftarrow j + 1$ и $l \leftarrow n$. Затем, если $k < l$, поменять местами $a_k \leftrightarrow a_l$, установить $k \leftarrow k + 1$, $l \leftarrow l - 1$ и повторять, пока не выполнится условие $k \geq l$. Вернуться к шагу 1.

Алгоритм Нарайаны

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

int NextPerm (int *a, int n) {
  int i, k, t, tmp;
  /* находим k такое что: a[k] < a[k+1] > .... > a[n-1]*/
  for (k = n - 2; (a[k] > a[k + 1]) && (k >= 0); k--);
  /* последняя перестановка */
  if (k == -1)
    return 0;
  /* находим t > k такое, что среди a[k+1],..., a[n-1]
     a[t] - минимальное число, большее a[k] */
  for (t = n - 1; (a[k] > a[t]) && (t >= k + 1); t--);
  tmp = a[k], a[k] = a[t], a[t] = tmp;
  /* оборачиваем участок массива a[k+1],..., a[n-1] */
  for (i = k + 1; i <= (n + k)/2; i++) {
    t = n + k - i;
    tmp = a[i], a[i] = a[t], a[t] = tmp;
  }
  return i;
}
```

Алгоритм Нарайаны

```
int main (void) {
  int *a, n, i;
  scanf ("%d", &n);
  a = (int*) malloc (n * sizeof(int));

  for (i = 0; i < n; i++)
    a[i] = i + 1;
  do {
    PrintPerm (a, n);
  } while (NextPerm (a, n));
  return 0;
}
```