Лекция 19 Цифровой поиск

19.1. ***Что такое цифровой поиск (поиск подстроки по образцу)***?

19.1.1. *Цифровой поиск* – частный случай поиска заданной подстроки (*образца*) в длинной строке (*тексте*). Поиск подстроки по образцу используется в текстовых редакторах, в Интернетных поисковиках и т.п. Например, биологи любят искать образцы (заданные цепочки нуклеотидов) в молекулах ДНК.

19.1.2. ***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*.

Из толкового словаря: «Высечка – *отделочный процесс* полиграфического производства для образования с помощью вырубки, высечки на высекательном прессе площадок на внешнем поле страниц справочных изданий, чтобы можно было быстро находить начало разделов».

Еще один пример – словарь с побуквенными метками для первого и последнего слова на каждой страницы (обычно помещаются в верхнем колонтитуле).

19.1.3. При цифровом поиске ключи рассматриваются как последовательности символов рассматриваемого алфавита (в частности, цифр или букв). Развивая идею побуквенных меток до ее логического завершения, получаем схему поиска. Цифровой поиск – реализация этой идеи. 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.

19.2. ***Пример***.

Пусть множество используемых букв (алфавит) {*A*, *B*, *C*, *D*}. Мы добавим к алфавиту еще одну букву ␣ (пробел). По определении. 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.

На рисунке (внизу) изображено дерево поиска (5-ичное). Прямоугольниками изображены вершины дерева, в овалах – значения слов (ключей) и связанная с ним информация (комментарий к слову или любая другая информация, которую мы будем обрабатывать). Тем самым любая вершина дерева – массив из m элементов (в данном примере $m = 5$).

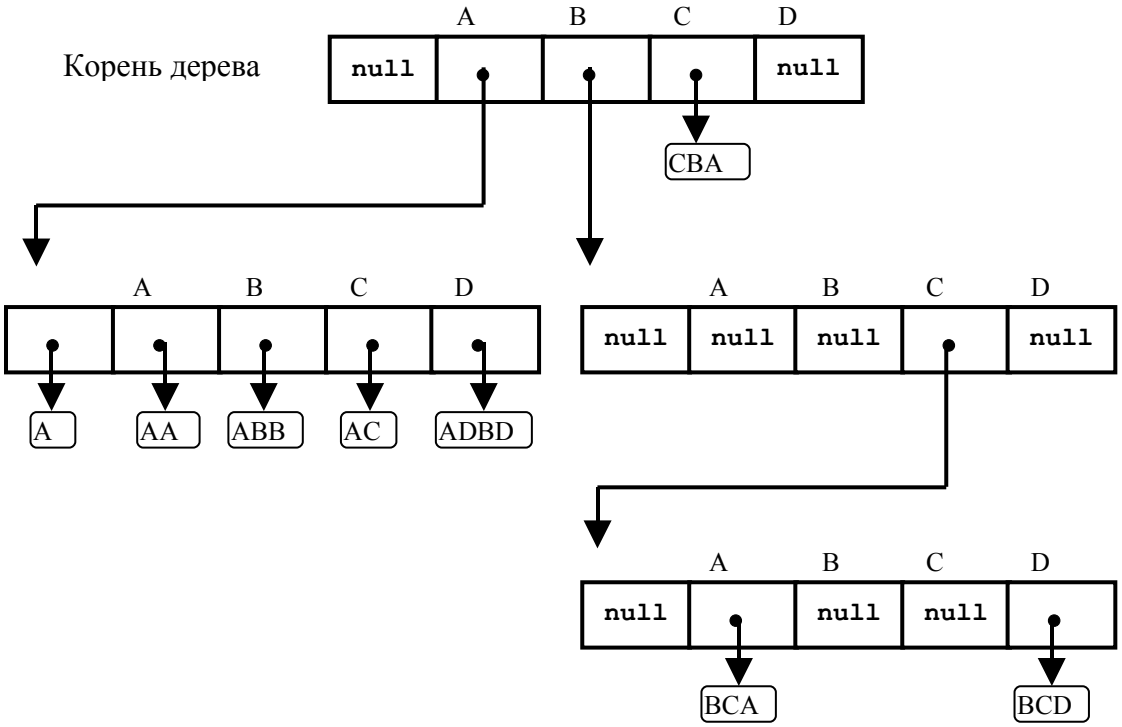
Каждый элемент вершины содержит либо ссылку на другую вершину m-ичного дерева, либо на содержащий информацию овал (ключ).

Рассмотрим **корень дерева**: как видно из словаря все слова начинаются с *A*, *B* или *C*, причем с *C* начинается только одно слово – *CBA*. Поэтому ссылки на узел для слов, начинающихся с *A*, на узел для слов, начинающихся с *B*, и на слово *CBA* (оно единственно).

Рассмотрим **узел для слов, начинающихся с *A***: у него пять ссылок на слова *A*␣, *AA*, *ABB*, *AC*, *ADBD*. Все перечисленные слова уникальны, так что дополнительных узлов не требуется.

Рассмотрим **узел для слов, начинающихся с *B***: оба слова начинаются с комбинации *BC*. Это приводит к необходимости завести еще один **узел для слов, начинающихся с *BC***. Это обстоятельство ухудшает эффективность, так как рассматриваемое дерево содержит много нулевых указателей (7), что вызывает большой перерасход памяти. Можно, конечно, исправить положение, модифицировав поиск.

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 (когда цифровой поиск проводится на двоичных деревьях, для разветвления берется один бит: $m = 1$). В реальной жизни рассмотренные методы используются, например, при поиске внутренних имен, которые строит компилятор. Чтобы сделать оценку по памяти и по эффективности, необходимы машинные эксперименты, чтобы проанализировать, как будет работать метод для конкретного словаря..

Занесение и исключение рассмотреть самостоятельно.

19.3. ***Программа поиска***.

19.3.1. Тип **`union`** (*объединение*). Синтаксически это структура, все поля которой наложены одно на другое. В Паскале этому соответствует вариантная запись. Семантически – это область памяти, в которой хранятся данные разных типов: один и тот же набор битов интерпретируется несколькими (двумя и более) разными способами.

19.3.2. Объявление объединения.

```
union тег {
    тип имя-члена;
    тип имя-члена;
    ……
    тип имя-члена;
```

} *переменные этого объединения;*

Пример:

```
union tree_node {
   struct record *r;
   struct tree * a[M+1];
} u;
```

19.3.3. Текст программы.

```
#include <stdlib.h>

#define M 20 //максимальное число символов в ключе
#define N 30 //мощность словаря

typedef char key[M]; //ключ – массив из M символов
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;
};

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 != NULL) {
     switch (p->t)
     {
       case ident:
         for (i = 0; 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;
}
```

19.4. 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 (используя факт упорядоченности слов в словаре, факт отсортированности). Именно так или похожим образом мы и работаем со словарями. Обычно предлагается пользоваться цифровым поиском пока количество различных слов не меньше некоторого k, а затем переключение к последовательным таблицам.

19.5. Обобщения: поиск по неполным ключам, поиск по образцу.

19.5.1. ***Формальная постановка задачи поиска по образцу***. Даны *текст* – массив $T[N]$ длины N и *образец* – массив $P[m]$ длины $m \le N$, где значениями элементов массивов T и P являются символы некоторого *алфавита* A. Говорят, что образец P входит в текст T со сдвигом s (или, что то же самое, с позиции $s + 1$), если $0 \le s \le N - m$ и для всех $i = 0, 1, \ldots, m - 1$ $T[s + i] = P[i]$. Сдвиг $s(T, P)$ называется *допустимым*, если P входит в T со сдвигом $s = s(T, P)$ и *недопустимым* в противном случае.
Задача поиска подстрок состоит в нахождении множества допустимых сдвигов $s(T, P)$ для заданного текста T и образца P.