Тема 6. Множества и хеширование

Хеширование

Быстрые деревья предоставляют средства для реализации множеств и отображений.

Множество здесь понимается как контейнерная структура данных, которая поддерживает следующие операции:

первичные:

— добавить элемент

— удалить элемент

— искать элемент (принадлежность)

— итератор по элементам

вторичные:

— объединение, пересечение, дополнение и т.д. как для математических множеств

Таким образом, мы можем релизовать множества на быстрых деревьях с трудоемкостью первичных операций $O(logN)$, правда, при условии, что на множестве элементов определена операция сравнения "меньше".

Два вопроса:

— а что, если сравнения на "меньше" нет ?

— а нельзя ли работать быстрее, чем $O(\log N)$?

Ответы: при некоторых допущениях, такие реализации множеств возможны.

Простой предварительный пример. Битовое множество.

Постановка задачи. Нужно работать с подмножествами неотрицательных целых чисел. Известно ограничение на максимальное значение (диапазон) таких чисел.

Идея реализации. Возьмем массив целых чисел и каждый бит элементов этого массива сопоставим с некоторым числом из полного диапазона. Если некоторые число содержится в подмножестве, то бит устанавливаем в 1, иначе в 0.

```
  class BitSet
  {
      unsigned int *bitElems;  // массив для хранения битов
      size_t maxVal;    // ограничение на максимальное значение

      bool Test(int k) { return (0 <= k && k <= maxVal); }

#define bit_shift  5         // для 32-разрядного int
#define bit_count  0x1F
#define indElem(k)  ((k) >> bit_shift)
#define indBit(k)   ((k) & bit_count)

   public:
      BitSet(int mVal);
      // другие конструкторы, если надо

     ~BitSet() { delete[] bitElems; }

      // надо еще проверять попадание в заданный диапазон !!!
      // добавить
```

```
    void Set(unsigned k) { bitElems[indElem(k)] |= 1 << indBit(k); }

    // удалить
    void Clear(unsigned k) { bitElems[indElem(k)] &= ~(1 << indBit(k)); }

    // принадлежит?
    bool InSet(unsigned k) { return (bitElems[intElem(k)]) & (1 << indBit(k)); }
};

BitSet::BitSet(size_t mVal)
{
    maxVal = mVal;
    size_t size = indElem(maxVal) + 1;
    bitElems  = new unsigned int[size];
    for (size_t i=0 i<size; i++) bitElems[i] = 0;    // memset
    // обработка отказа по памяти ....
}
```

А вот с итератором тут все не так радужно ... Придется посмотреть на каждый элемент массива.

Ограничение про unsigned легко устранить предварительным сдвигом в положительную область.

Несмотря на примитивность, такое множество может оказаться очень эффективным.

Пример 1. Решето Эратосфена для простых чисел.

Максимальный unsigned $\sim 2^{32} - 1$.

количество элементов в bitElems — 2^{27} целых или 2^{29} байтов т.е. 512 М.

Пример 2. Система бронирования и продажи авиабилетов для Москвы дата-рейс-место — занято/свободно

Сколько таких записей для Москвы?

период — несколько месяцев (256 дней) 2^8

сколько рейсов

аэропортов 2^2

вылетов в день (1 в минуту) 2^{10}

мест в рейсе 2^7

итого 2^{27} бит, 2^{24} байтов — 16 М.

Хеширование

Множество M всех возможных значений.

Множество m, с которым мы работаем $|m| << |M|$.

Hash function $h(x) : M \to \{0, \ldots, p-1\}$.

Классы эквивалентности $M_k = \{x \in M : h(x) = k\}$.

Работу с $x \in m$ сводим к работе с $M_{h(x)}$.

Все зависит от того, насколько хорошо хеш-функция рассеивает наше множество m по разным M_k.

Выбор хорошей хеш-функции для конкретной задачи — это отдельная наука.

Примеры хеш-фукций и их выбор чуть позже.

Пример: метод списков (подмножеств).

Хеш-таблица указателей на подмножества (например, списки, деревья и т.п.) Эффективность напрямую зависит от качества хеш функции.

Пример: метод проб.

Хеш-таблица элементов (tab[]). Размер таблицы с заметным запасом по количеству.

Идеология работы с при поиске или добавлении.

$k = h(x)$

проверка $tab[k]$

при неудаче проверка $tab[k+1]$

при неудаче проверка $tab[k+2]$

и т.д. циклически, пока не найдем

Идеология работы с при удалении.

$k = h(x)$

далее поиск

если нашли на месте $tab[s]$ и цепочка продолжается, то перенос в $tab[s]$ первого подходящего элемента $tab[r]$ из цепочки ($h(tab[r]) \le s$) и т.д. с $s = r$, либо прерываем цепочку в позиции s, если подходящего элемента нет.

Теорема. Пусть в методе проб коэффициент заполнения хеш таблицы есть $s = m/n$, где n — размер таблицы, m — количество занятых ячеек (мощность рабочего множества). Пусть хеш функция обеспечивает равномерное распределение элементов рабочего множества по позициям в таблице. Тогда среднее число проб при работе с множеством (при поиске, добавлении, удалении) составляет $1/(1-s)$.

Следствие. Если хеш таблица заполнена наполовину, то среднее число проб есть 2 независимо от фактического количества элементов в рабочем множестве.

Доказательство. Вероятность попасть на занятую ячейку есть $s_1 = s$, на незанятую — $(1 - s_1)$.

При второй пробе эти вероятности есть $s_2 = (m-1)/(n-1)$ и $(1 - s_2)$.

Для третьей пробы $s_3 = (m-2)/(n-2)$ и $(1 - s_3)$.

Вероятность сделать ровно k проб есть

$q_k = s_1 s_2 \dots s_{k-1}(1 - s_k)$ ($k-1$ неудачных и последняя удачная)

Среднее число проб — это математическое ожидание $\sum_{k=1}^{m} k q_k$.

вычислим его

$$1(1 - s_1) + 2s_1(1 - s_2) + 3s_1 s_2(1 - s_3) \dots =$$

$$= 1 + s_1 + s_1 s_2 + s_1 s_2 s_3 + \dots < 1 + s_1 + s_1^2 \dots - (\text{ геом. прогрессия }) \le \frac{1}{1-s}.$$

Примеры хеш функций

```
int additive_hash (char *key, int len, int p)
{   int k, hash = len;
    for (k=0;k<len;k++)  hash += key[k];
    return hash % p;
}
int XOR_hash (char *key, int len, int p)
{   int k, hash = 0;
    for (k=0;k<len;k++)  hash ^= key[k];
```

```
    return hash % p;
}
```

Это две очень плохих функции. Более-менее сносно они работают только при достаточно длинных последовательностях байтов (несколько сотен).

```
int rotate_hash (char *key, int len, int p)
{   int k, hash = len;
    for (k=0;k<len;k++)
        hash = (hash<<5)^(hash>>27)^key[k];
    return hash % p;
}
int elf_hash (char *key, int len, int p)
{   int hash = 0;
    unsigned int k, g;
    for (k=0;k<len;k++)  {
        hash = (hash<<4) + key[k];
        g = hash & 0xF0000000;
        if ( g ) hash ^= g>>24;
        hash &= ~g;
    }
    return hash % p;
}
```

Это представители перемешивающих функций. Перемешивание осуществляется путем сдвигов и наложений с помощью битовых операций. Константы подбираются обычно опытным путем.

Иногда возникает необходимость получить семейство различных хеш-функций со схожими характеристики. В таких случаях используются функции со случайными таблицами. Идея достаточно проста — взять случайную таблицу с требуемыми значениями хеш-функции, а индекс в этой таблице вычислять также как некоторое хеш-значение. Вот два примера подобных функций.

```
char tab[256];  // заполняется случайными числами

char pearson_hash (char *key, int len, char *tab)
{   int k, hash = 0;
    for (k=0;k<len;k++)  hash = tab[hash^key[k]];
    return hash;
}
char CRC_hash (char *key, int len, int mask, char *tab)
{   int k, hash = len;
    for (k=0;k<len;k++)
        hash = (hash<<8)^tab[(hash>>24)^key[k]];
    return hash & mask;
}
```

Еще одно семейство составляют полиномиальные функции. Значение вычисляется как некоторый полином, коэффициенты которого определяются входной последовательностью байтов. В следующих примерах легко увидеть схему Горнера для вычисления многочлена от аргумента, указанного в комментарии.

```
char djb2_hash (char *key, int len, int p)
{   int k, hash = 5381;    // аргумент = 33
    for (k=0;k<len;k++)
        hash = ((hash<<5) + hash) + key[k];
    return hash % p;
}
char java_hash (char *key, int len, int p)
{   int k, hash = 0;    // аргумент =  31
    for (k=0;k<len;k++)
        hash = ((hash<<5) - hash) + key[k];
    return hash % p;
}
char sdbm_hash (char *key, int len, int p)
{   int k, hash = 0;      // аргумент = 65599
    for (k=0;k<len;k++)
        hash = ((hash<<16) + (hash<<6) - hash) + key[k];
    return hash % p;
}
```

Совершенные хеш-функции

Задача: построить оптимальную хеш-функцию для заданного набора входных значений, чтобы она не имела коллизий на данном наборе. В англоязычной литературе таки функции называются Perfect Hash Functions, мы будем использовать термин "совершенные".

Пусть M — множество всех возможных элементов (ключей), m — подмножество заданных фиксированных ключей, с которым мы работаем.

Определение: Хеш-функция $h(x)$ называется совершенной, если для любых $x_1, x_2 \in M$, $x_1 \neq x_2 \Longrightarrow h(x_1) \neq h(x_2)$.

Определение: Совершенная хеш-функция $h(x)$ называется минимальной, если $h : M \to \{0, \ldots, |m| - 1\}$.

Определение: Пусть на множестве ключей введено отношение порядка "$<$". Совершенная хеш-функция $h(x)$ сохраняет порядок, если для любых $x_1, x_2 \in M$, $x_1 < x_2 \Longrightarrow h(x_1) < h(x_2)$.

Гарантированное построение минимальной совершенной хеш-функции представляет собой непростую задачу. Поэтому на практике обычно применяются итерационные или вероятностные подходы, которые позволяют получить результат после некоторого количества проб. Достаточно хорошо известен пакет gperf, разработанный в рамках проекта GNU. Но в нашем курсе мы разберем другой способ, основанный на вероятностных соображениях и позволяющий получить минимальную совершенную хеш-функцию, сохраняющую произвольный заранее заданный порядок.

Ищем функцию в виде $h(x) = g(f_1(x)) + g(f_2(x))$, где f_1 и f_2 — хеш-функции на основе случайных таблиц, а g — некоторая функция, которую предстоит построить.

Выберем число $n = c|m|$, где $c > 2$ — некоторая константа. Сгенерируем две случайные хеш-функции f_1 и f_2 с диапазонами значений $\{0, \ldots, n-1\}$. Каждому ключу x поставим в соответствие пару $(f_1(x), f_2(x))$, и из этих пар, как из ребер, построим граф, склеивая вершины с одинаковыми значениями.

Нам хочется, чтобы в этом графе не было циклов. Используя вероятностные соображения, можно доказать, что вероятность отсутствия циклов в таком графе асимптотически равна $p = \sqrt{\frac{c-2}{c}}$ при $n = cm, n \to \infty$.

Таким образом, если построенный граф содержит циклы, то мы его отбрасываем, генерируем новые случайные таблицы для функций f_1 и f_2, и строим новый граф. Так как вероятность получить ациклический граф достаточно велика, то ожидаемое число попыток оценивается математическим ожиданием

$$E = \sum_{k=1}^{\infty} kp(1-p)^{k-1} = \frac{1}{p} = \sqrt{\frac{c}{c-2}}$$

Если взять $c = 3$, то эта формула дает весьма оптимистичную оценку — в среднем не более двух попыток.

Иллюстрация: построение графа для конкретного набора значений

Из описанного построения становится ясно, зачем в графе не должно быть циклов. Действительно, если граф содержит цикл, то в общем случае возникающая цепочка значений не будет совместна. То есть вернувшись в начальную вершину цикла, мы можем получить для другое значение в этой вершине.

Отметим плюсы и минусы этого алгоритма. Трудоемкость построения хеш-функции невелика, причем самая сложная часть — проверка графа на цикличность. Однако при реализации требуется память для хранения таблиц хеш-функций. Эта память составляет $c|m|$ для задания функции g и две таблицы, определяющие функции f_1 и f_2 (обычно для каждой используется таблица на 256 значений — по количеству различных байтов). Кроме этого, требуется хранить и сами заданные ключи, так как если для некоторого x имеем $h(x) \in \{0, \ldots, |m|-1\}$, то это еще не означает, что $x \in M$.

Замечание о криптографических хеш-функциях. Но это отдельная наука.