Эффективная организация алгоритмов и данных

Любой вариант курсовой роботы по программированию можно реализовать так, что он будет работать сколь угодно медленно или использовать сколь угодно много памяти
«Опыт приема КР во втором семестре»

Предпосылки:

- Ресурсов много не бывает
- При увеличении доступных ресурсов размерности обрабатываемых данных растут настолько, чтобы использовать их полностью.
- На каждом уровне программирования разработчик обычно не учитывает, что делается на нижележащих
- «Именно поэтому наш оркестр так звучит». М.Жванецкий

Содержание:

- Лайфхаки здравого смысла
- Трудоемкость алгоритмов
- Организация данных, размещение и поиск

Лайфхаки здравого смысла

Не вычерпывай бочку чайной ложкой: группировка запросов, буферизация данных в потоках

Шей да пори, не будет простой поры (Не делай много раз одну и ту же работу). Динамическое программирование: кэш решений для повторного их использования (при комбинаторном переборе вариантов)

Порядок имеет значение: при селекции по нескольким параметрам начинать отсеивать нужно с того, который дает максимальное отсечение

Тупой алгоритм надежнее, но умный – быстрее.

Трудоемкость алгоритмов. Организация размещения и поиска данных

- **Трудоемкость –** зависимость количества операций алгоритма он размерности структуры данных
- **Диапазон трудоемкости –** вид функции для лучшего, худшего и среднего, чувствительность к данным
- Оценочный характер трудоемкости, вид функции трудоемкости, O-нотация
- **Масштабирование** – оценка времени выполнения программы при размерности данных N1, если известно время выполнения T0 при размерности данных N0

  T0 = K*O(N0), T1 = K*O(N1) – K – коэффициент пропорциональности

  T1 = T0*O(N1)/O(N0)

- **Линейный поиск** – неупорядоченные данные
- **Двоичный поиск, двоичное дерево**, B-дерево – упорядоченные данные (ключ, индексирование)
- **Размещение данных** для последующего эффективного поиска

Распределение по разрядам ключа

- Ключ–строка – разряд-символ
- Ключ-число – разряд-цифра

Лексикографическая сортировка:

- «карманы» в соответствии с кодами символа/цифры/двоичного разряда
- Распределение по карманам, начиная с **младшего разряда**
- Соединение карманов
- Повторение по всем разрядам

Линейная трудоемкость: T(N*K), K-количество разрядов
Дополнительная память под «карманы» N*K (для списков – не нужна)

Распределяющая сортировка

125 354 876 388 945 543 567 234 765

543

354 234

125 945 765

876

567

388

543 354 234 125 945 765 876 567 388

125

234

543 945

354

765 567

876

388

125 234 543 945 354 765 567 876 388

125 234 543 945 354 765 567 876 388

125

234

364 388

543 567

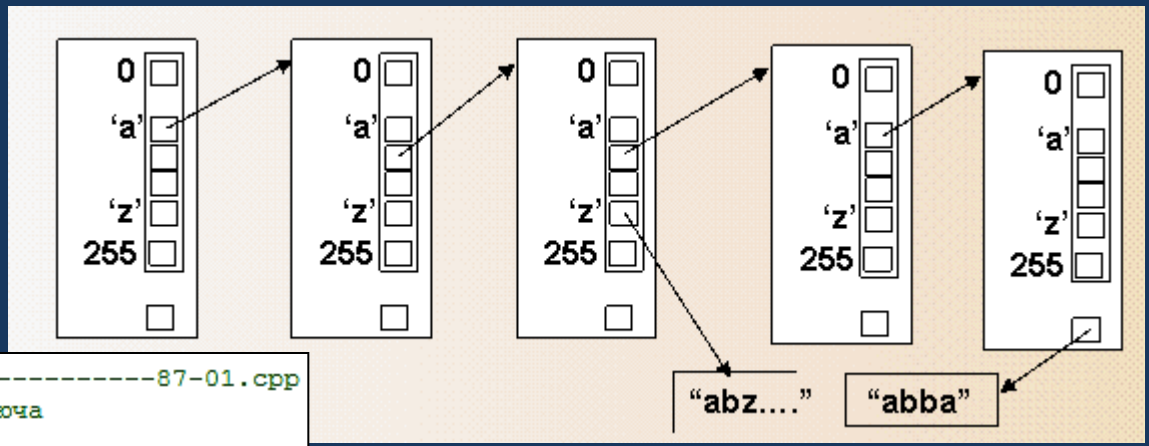
765

876

945

125 234 364 388 543 567 765 876 945

Дерево с распределением по разрядам ключа

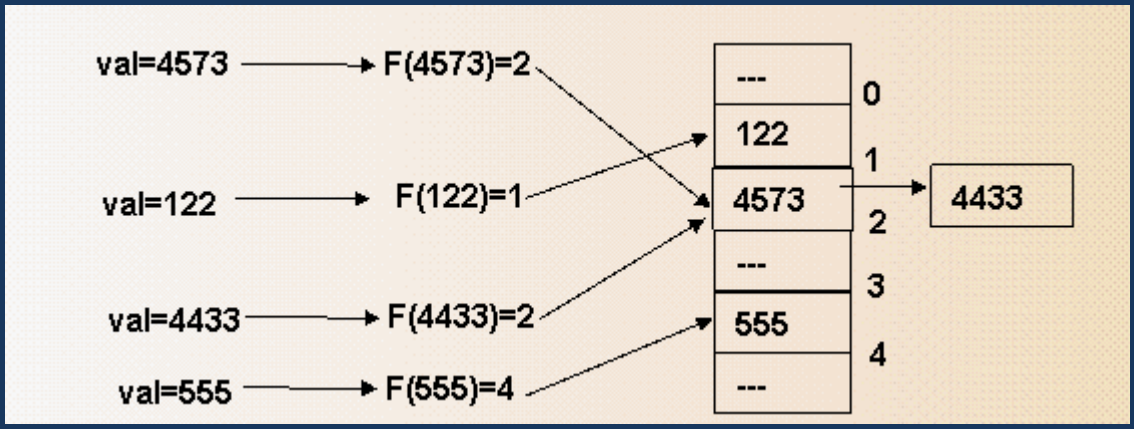



```
//-----------------------------------------------87-01.cpp
// Дерево с распределением по разрядам ключа
struct node{
    char *data;      // Связанные данные для текущего ключа
    int cnt;         // Счетчик повторений
    node *pr[256];   // Вершины - потомки по разрядам ключа
void create(){
    for (int i=0;i<256;i++) pr[i]=NULL;
    cnt=0;
    data=NULL;
    }
void insert(unsigned char *wd, int lv){
    if (wd[lv]==0 || wd[lv]==' ' || wd[lv]=='\t'){
        cnt++;
        if (cnt==1) data=strdup((char*)wd);
        }
    else{
        unsigned k=wd[lv];
        if (pr[k]==NULL) {
            pr[k]=new node();
            pr[k]->create(); }
        pr[k]->insert(wd,lv+1);
        }
    }
```

```
void scan(int lv){
    if (cnt!=0) {
        char c2[80];
        CharToOemA(data,c2);
        printf("n=%d:%s\n",cnt,c2);
        }
    for (int i=0;i<256;i++)
        if (pr[i]!=NULL) pr[i]->scan(lv+1);
    }
```

Хеширование

- Hash – рубить, крошить (англ.)
- Решение «от обратного»: разместить там, где будем искать
- Вычисление адреса: позиция размещения (индекс) определяется формальным преобразованием адреса (*номер квартиры, где будете жить, определяется функцией преобразования номера паспорта, F(5009 138467) -> 35*)
- Хеш-функция выглядит как «псевдослучайная», должна равномерно распределять значения ключа по пространству адресов (индексов)



Хеширование

- для преобразования значения в адрес используется функция (в обычном, математическом смысле этого слова, преобразующая размещаемое значение в адрес – индекс в таблице). Она должна обеспечивать псевдо-случайное, хаотическое разбрасывание значений по ячейкам. Для этого используются всяческие способы «перемешивания» разрядов размещаемых данных (отсюда **hash** – кромсать, перемешивать);
- при размещении данных возможны *столкновения (коллизии*), когда два разных значения получают от функции один и тот же адрес размещения;
- коллизии разрешаются либо за счет размещения «лишних» значений в соседних свободных ячейках, либо путем создания из них линейных цепочек, например, списков;
- частота коллизий *зависит от степени заполнения таблицы*, чтобы метод работал эффективно, она должна содержать достаточно свободного места;
- существует вероятность, что все значения попадут в одну и ту же ячейку и тогда структура выродится в неупорядоченную последовательность с линейным поиском
- Трудоемкость (хеширование) Tмин(N)=1 Tср(N) >= 1, **Tmax(N)=N**
- Трудоемкость (дв.поиск) Tмин(N)=1 Tср(N) <= $\log_2(N)$, **Tmax(N)=$\log_2(N)$**

Хеширование

```
//-------------------------------------------------------------87-02.cpp
// Хеширование - списки конфликтующих ключей
struct cell{              // Ячейка хэш-таблицы - элемент списка
    int ckey;             // Значение ключа
    cell *next;
};
struct HASH{              // Структура со встроенными функциями
    cell **hash;          // Хеш-таблица - массив указателей на списки
    int POW,sz;           // Размерность таблицы sz=10^POW
    int KSIZE,w;          // Размерность ключа (кол-во цифр) w=10^KSIZE
    int s1;               // Простое число, следующее за sz
void reset(){             // Разрушение хеш-таблицы
    cell *p,*q;
    for (int i=0;i<sz;i++)
        for (p=hash[i];p!=NULL;){
            q=p; p=p->next; delete q;
            }
    delete []hash;
    }
void init(int p0, int ksz0){
    int i;
    srand(time(NULL));
    POW=p0;
    sz=pow(10,(double)POW);
    KSIZE=ksz0;
    w=pow(10.,KSIZE);
    for(s1=sz+1;1;s1++){// Простое число, следующее за sz
        for (i=2;i<s1 && s1%i!=0;i++);
        if (i==s1) break;
        }
    hash=new cell*[sz];
    for (int i=0;i<sz;i++) hash[i]=NULL;
    }
```

Хеширование

```
// хэш-функция - остаток от деления на s1
int hash_fun(int key){ return (key%s1) %sz; }
void insert(int val){
    int v=hash_fun(val);
    cell *q;
    for (q=hash[v];q!=NULL;q=q->next)
        if (q->ckey==val) break;
    if (q==NULL)              // Значения в таблице нет
        {
        q=new cell;           // Вставка в начало списка
        q->ckey=val;
        q->next=hash[v];
        hash[v]=q;
        }
    }
void show(){
    int cnt=0,cnt1=0;
    for (int i=0;i<sz;i++){
        printf("\n[%d]=",i);
        if (hash[i]!=NULL){
            cnt++;
            for (cell *q=hash[i]; q!=NULL; q=q->next,cnt1++)
                printf("%d ",q->ckey);
        }}
    printf("\nT=%lf\n",(double)cnt1/cnt);
    getch();
    }
void load_rand(double proc){ // Процент загрузки таблицы
    for (int i=0;i<sz*proc;i++)
        insert(rand()%w);
    }
};
```

```
int main(){
    HASH TB;
    TB.init(1,3);
    TB.load_rand(0.50); TB.show();
    TB.load_rand(0.25); TB.show();
    TB.load_rand(0.20); TB.show();
    TB.load_rand(0.20); TB.show();
    TB.load_rand(0.20); TB.show();
    TB.load_rand(0.20); TB.show();
}
```

```
[0]=418
[1]=
[2]=
[3]=201
[4]=
[5]=
[6]=435
[7]=
[8]=492
[9]=537
T=1.000000

[0]=296 418
[1]=551
[2]=
[3]=234 201
[4]=
[5]=
[6]=435
[7]=
[8]=492
[9]=537
T=1.333333
```

```
[0]=582 296 418
[1]=881 551
[2]=893
[3]=289 234 201
[4]=
[5]=
[6]=655 435
[7]=403
[8]=492
[9]=53 658 262 537
T=2.125000

[0]=253 582 296 418
[1]=881 551
[2]=893
[3]=289 234 201
[4]=
[5]=
[6]=655 435
[7]=403
[8]=492
[9]=405 196 53 658 262 537
T=2.500000
```

```
[0]=582 296 41
[1]=881 551
[2]=
[3]=234 201
[4]=
[5]=
[6]=435
[7]=403
[8]=492
[9]=537
T=1.571429

[0]=582 296 418
[1]=881 551
[2]=893
[3]=234 201
[4]=
[5]=
[6]=435
[7]=403
[8]=492
[9]=658 262 537
T=1.750000
```