Lab5 - Hash Report

2021201709 李俊霖

1. 测试数据的构造

1.1.构造方法

- 编写 `generator.cpp` 程序，分别读取 `poj.txt` 和 `hdu.txt` 中的数据，并生成相应数据特征的数据。
- **为实现不同的插入/查询操作比例，一些重要的参数如下：**

```
// 输入输出文件
#define INPUT_POJ "./data/poj.txt" // ASCII
#define OUTPUT_POJ "./data/test_a2.in"
// #define OUTPUT_POJ "./data/1-1.in"
#define INPUT_HDU "./data/hdu.txt" // utf-8
#define OUTPUT_HDU "./data/test_u1.in"
// 数据特征参数
#define DATA_NUM 10000                      // 插入组数
#define ASK_NUM 9000                        // 查询组数
#define ASK_NOT_IN_NUM 1000                 // 不在插入中的组数
#define ASK_IN_NUM ASK_NUM - ASK_NOT_IN_NUM // 在插入中的组数
```

- 参数解释：
  - 插入数据：生成的该数据规模的数据中，插入操作的总数。
  - 查询数据：生成的该数据规模的数据中，查询操作的总数。
  - 不在插入中的组数：该数据中，查询的数据不在插入的数据范围内的数据组数。
- **不同插入/查询的分布方式实现思路：**
  - 先插入后查询：分别生成插入数据和查询数据，再拼接即可。
  - 插入查询随机分布：用产生的随机数mod2的方式选择查询/插入操作。为防止在数据集的开始便做太多无意义的查询操作（此时未插入，比无法查询），选择在整体规模的前1/2、1/3等阶段只做插入操作。
  - 简要代码实现如下：

```
    if (k < (data_num2 + ask_num2) / 3)
      {
        outfile2 << "0 " << out_name[i] << " " << out_number[i] << endl;
        i++;
        continue;
      }
    if (line2[k] % 2 == 1 && i <= data_num2)
      {
        outfile2 << "0 " << out_name[i] << " " << out_number[i] << endl;
        i++;
      }
    else if (line2[k] % 2 == 1 && i > data_num2)
      {
        outfile2 << "1 " << prepare_name[j] << endl;
        j++;
      }
```

- 其他代码细节不再一一赘述，具体代码保存在 `generator` 文件夹的 `generator1.cpp` 、 `generator2.cpp` 中。

1.2.数据特征

ascii编码数据特征：

组别	数据规模	插入/ 查询操作比例	插入/查询的分布方式
1	600000	1:1	前30万组插入，后30万组查询
2	1260000	20:1	每50000组插入，接2500组查询
3	912000	10:9	每19000组数据内，前2/3组为插入数据， 此后插入和查询数据随机分布

- 数据分别保存在 `1-1.in` 、 `1-2.in` 、 `1-3.in` 中。

utf-8编码数据特征如下：

（注：数据特征同上）

组别	数据规模	插入/ 查询操作比例	插入/查询的分布方式
1	600000	1:1	前30万组插入，后30万组查询
2	1260000	20:1	每50000组插入，接2500组查询

组别	数据规模	插入/ 查询操作比例	插入/查询的分布方式
3	912000	10:9	每19000组数据内，前2/3组为插入数据， 此后插入和查询随机分布

- 数据分别保存在 `2-1.in` 、 `2-2.in` 、 `2-3.in` 中。

2. 哈希函数的实现

2.1.哈希函数实现思路

针对ascii编码的哈希函数

- 将散列码每次循环左移5位，再累加当前字符，得到字符串的循环移位散列码，实现字符串到整型的哈希函数。
- 代码如下：
- ```
  int hash_ASCII(string str)
  {
      int hash = 0;
      for (int i = 0; i < str.length(); i++)
      {
          hash = (hash << 5) | (hash >> 27);
          hash += (int)str[i];
      }
      hash = hash & 0x7FFFFFFF;
      return hash;
  }
  ```

针对utf-8编码的哈希函数

- 先通过不同长度的utf-8编码开头的标识识别该编码的长度。（`int get_byte(unsigned char c)` 函数实现）
- 再根据识别出来的长度从字符串中把一个一个的utf-8字符提取出来，并转换成整型数据。（`int byte_to_int(char *s, int byte)` 函数实现）
- 然后类似ascii的哈希函数，将散列码每次循环左移5位，再累加当前字符，得到字符串的循环移位散列码。
- 最后 `hash = hash & 0x7FFFFFFF` ，得到整型数据。

2.2.冲突处理策略

策略1：独立链法

- 插入：
  ```

    - 先根据查找链找到最末尾的节点。
    - 如果要插入的节点未冲突，直接插入即可。如果待插入节点冲突，将now接到pre后面，形成查找链。
    - 代码如下：

```
// 冲突处理策略1:独立链法（插入）
void insert_model1(int id, int v)
{
    int hashCode = abs(id % M);
    struct hashNode1 *now = Hash_ID1[hashCode];
    struct hashNode1 *pre = 0;
    while (now) // 找到最末尾的节点
    {
        pre = now;
        now = now->next;
    }

    now = new hashNode1;
    if (pre) // 待插入节点冲突
    {
        now->key = id;
        now->value = v;
        pre->next = now; // 将now接到pre后面，形成查找链
    }
    else // 要插入的节点未冲突
    {
        now->key = id;
        now->value = v;
        Hash_ID1[hashCode] = now;
    }
}
```

- 查找算法：
    - 根据查找链在哈希表中找到符合id的值。
    - 其中，M为桶的容量。
    - 代码如下：

```
// 冲突处理策略1:独立链法（查找）
int find_model1(int id)
{
    int hashCode = abs(id % M);
    struct hashNode1 *now = Hash_ID1[hashCode];
    while (now)
    {
        if (now->key == id)
        {
            return now->value;
        }
        else
            now = now->next;
    }
    return -1;
}
```

策略2：线性试探法

- **插入算法**:
  - 调用 `probe4free2(key)` 函数，为新词条找到空桶，然后插入新元素。
  - 代码:

```
// 冲突处理策略2:线性试探法（插入）
void insert_model2(int key, int val)
{
    if (Hash_ID2[probe4hit2(key)].key == key) // 已经存在，不必再插入
        return;
    int r = probe4free2(key);
    Hash_ID2[r].key = key;
    Hash_ID2[r].value = val;
    return;
}
```

- **寻找空桶函数 `probe4free2` 。**
  - 如果该桶内已经有存在值，跳过，桶的下标加1，依次往后继续找空桶。
  - 代码:

```
int probe4free2(int key)
{
    int r = abs(key % M);
    while (Hash_ID2[r].key != 0) // 已经有存在值，跳过，继续找空桶
        r = (r + 1) % M;
    return r;
}
```

- **查找算法**:
  - 调用 `probe4hit2(key)` 函数，沿着关键码key的查找链顺序查找。
  - 代码:

```
// 冲突处理策略2:线性试探法（查找）
int find_model2(int key)
{
    int r = probe4hit2(key);
    if (Hash_ID2[r].value == 0)
        return -1;
    return Hash_ID2[r].value;
}
```

- **寻找命中函数 `probe4hit2` 。**
  - 当桶不为空且里面存的关键码不是我们要寻找的关键码时，桶的下标加1，依次往后继续寻找。
  - 代码：

```
int probe4hit2(int key)
{
    int r = abs(key % M);
    // 带删除算法
    // while ((Hash_ID2[r].key != 0 && (key != Hash_ID2[r].key) || (Hash_ID2[r].key==0 && lazy
    while (Hash_ID2[r].key != 0 && (key != Hash_ID2[r].key))
        r = (r + 1) % M;
    return r;
}
```

策略3：双向平方试探法

- 查找和插入的基本算法和策略二线性试探法一致，唯有两个命中函数有不同，因此以下只介绍两个函数的实现细节。
- **寻找空桶函数 `probe4free2` 。**
  - 如果该桶内已经有存在值，跳过，桶的下标以加/减递增值的平方往两边试探桶单元。
  - 代码：

```
int probe4free3(int key)
{
    int r = abs(key % M);
    // 双向平方探测
    int i = 1, times = 1, temp_r = r;
    while (Hash_ID2[r].key != 0) // 已经有存在值，跳过，继续找空桶
    {
        if (times % 2 == 1) // 奇数次，加
        {
            r = temp_r;
            r = (r + i * i) % M;
            i++;
            times++;
        }
        else if (times % 2 == 0) // 偶数次，减
        {
            r = temp_r;
            r = (r - i * i) % M;
            i++;
            times++;
        }
    }
    return r;
}
```

- **寻找命中函数 `probe4hit2` 。**
  - 当桶不为空且里面存的关键码不是我们要寻找的关键码时，桶的下标桶的下标以加/减递增值的平方往两边试探桶单元，依次往后继续寻找。
  - 代码：

```
int probe4hit3(int key)
{
    int r = abs(key % M);
    // 双向平方探测
    int i = 1, times = 1, temp_r = r;
    while (Hash_ID2[r].key != 0 && (key != Hash_ID2[r].key))
    {
        if (times % 2 == 1) // 奇数次，加
        {
            r = temp_r;
            r = (r + i * i) % M;
            i++;
            times++;
        }
        else if (times % 2 == 0) // 偶数次，减
        {
            r = temp_r;
            r = (r - i * i) % M;
            i++;
            times++;
        }
    }
    return r;
}
```

3.测试结果

注:

- 表格内时间数据单位为秒。
- 冲突策略1、2、3分别为独立链法、线性试探法、双向平方试探法。
- 时间数据测试结果为连续测量三次取平均值。

3.1.ascii编码哈希函数测试结果:

输入数据	独立链法	线性试探法	双向平方试探法
1-1	4.098	2.330	2.157
1-2	39.356	4.457	4.351
1-3	6.008	3.485	3.345
2-1	4.850	1.539	1.486
2-2	85.604	2.544	2.667
2-3	11.234	1.838	1.856

3.2.utf-8编码哈希函数测试结果：

输入数据	独立链法	线性试探法	双向平方试探法
1-1	3.566	2.208	2.213
1-2	38.671	4.335	4.220
1-3	6.041	3.380	3.388
2-1	3.220	1.577	1.521
2-2	32.795	2.784	2.714
2-3	6.183	1.967	1.951

4.问题回答

4.1.问题1

- **问：将utf-8字符串“当作”ascii字符串进行处理，使用针对ascii字符串的哈希函数，实际效果如何（相比“针对utf-8字符串设计的哈希函数）？可能的原因是什么?**
- 答：
  - 整体上而言，前者的效果要差于针对utf-8字符串设计的哈希函数。特别是对于数据规模比较大的那组数据而言，使用针对性的哈希函数效果要明显更好，但是对于开放定址法和解决一些数据规模比较小的数据集，其优异效果不明显。
  - 可能的原因：使用针对utf-8字符串设计的哈希函数来处理utf-8字符，可以更针对性地找到字符和哈希值的映射关系，减少出错的可能性，减少不同字符串直接映射到相同哈希值的可能性，提高查找的效率和准确性。
  - 但是另一方面，由于本时间数据的统计包括了哈希函数之前的utf-8字符预处理时间，因此对于预处理更复杂一些的utf-8编码而言，在小数据范围内优势并不明显。

4.2.问题2

- **问：在你的测试中，不同的冲突处理策略性能如何？可能的原因是什么?**
- 答：
  - 线性试探法和双向平方试探法的性能要明显优于独立链法；线性试探法和双向平方试探法性能相近，双向平方试探法略占优。
  - 可能的原因：
    - 链地址的存储采用链表的结构，存储是动态的，查询时跳转需要更多的时间，因此查询时效率较低。
    - 线性试探法和双向平方试探法等开放定址法更容易进行序列化操作，查询的效率更高。

4.3.问题3

- **问：设计哈希函数时，我们往往假定字符串每个位置上出现字符集内每个字符的概率都是相等的，但实际的数据集往往并不满足这一点。这可能造成什么影响?**
- 答：大量的哈希冲突之后会导致数据聚集现象。
  - 对于链地址法，这会导致某一个桶对应的空间的查找链变得非常长，查询时效率显著较低。
  - 对于线性试探法，由于各查找链均由物理地址连续的桶单元组成，会加剧数据聚集现象，对后续关键码的查询需要做更多次的试探，因此时间成本也上升。
  - 对于双向平方试探法，顺着查找链，试探位置的间距将以线性的速度增长（1、3、5、7……），因此一旦发生冲突可较快地调离聚集区域，有效缓解聚集现象。

4.4.问题4

- **问：对于“字符串到数字映射”问题(给定一组字符串以及它们各自对应的数字，然后多次查询某个字符串对应的数字，可能会在中途更新某个字符串对应的数字)，哈希表并不总是最优方案。请描述一种输入数据，再举出一种哈希表之外的数据结构，对这种数据，这种数据结构能比哈希表更高效地解决“字符串到数字映射”问题。**
- 答：
  - 输入数据：一个数据库中不同同学的姓名和对应的各科学习成绩。
  - 数据结构：BitMap（位图），使用每个位表示某种状态。即给定一个整型数据，将该整数映射到对应的位上，并将该位由0改为1。该方法可以大大节省在海量数据集下的内存空间，适合对整型数据进行查询统计等操作。

5.总结与收获

- 通过这次lab，笔者对哈希函数实现和几种哈希冲突策略有了更深的理解，尤其是对开放定址法的查找空位和查找命中函数，愈发理解其精妙之处。
- 除此以外，笔者还补充了有关utf-8编码的基础知识，对编码解码有了初步的了解。

6.参考资料

1. 《数据结构(c++语言版)》邓俊辉编著-清华大学出版社
2. 哈希表针对冲突的两种方式优缺点是什么?
3. 解决哈希冲突