Hash Tables

1. Recall ADT Table

- Some Table operations:
  
  `tableInsert(NewItem);`
  `tableDelete(SearchKey);`
  `tableRetrieve(SearchKey, TableItem);`
  `traverseTable(visit);`

- The worst case performance on each operation:

<table>
<thead>
<tr>
<th></th>
<th>Insertion</th>
<th>Deletion</th>
<th>Retrieval</th>
<th>Traversal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted array based</td>
<td>O(1)</td>
<td>O(N)</td>
<td>O(N)</td>
<td>O(N)</td>
</tr>
<tr>
<td>Unsorted pointer based</td>
<td>O(1)</td>
<td>O(N)</td>
<td>O(N)</td>
<td>O(N)</td>
</tr>
<tr>
<td>Sorted array based</td>
<td>O(N)</td>
<td>O(N)</td>
<td>O(log N)</td>
<td>O(N)</td>
</tr>
<tr>
<td>Sorted pointer based</td>
<td>O(N)</td>
<td>O(N)</td>
<td>O(N)</td>
<td>O(N)</td>
</tr>
<tr>
<td>Binary Search Tree (balanced)</td>
<td>O(log N)</td>
<td>O(log N)</td>
<td>O(log N)</td>
<td>O(N)</td>
</tr>
<tr>
<td>Maxheap</td>
<td>O(log N)</td>
<td>O(N)</td>
<td>O(N)</td>
<td>O(N)</td>
</tr>
</tbody>
</table>

Note:

- Maxheap supports max. element deletion/retrieval in O(log n)/O(1)
- For unbalanced binary search tree Insertion/Deletion/Retrieval $\Rightarrow$ O(N)

- Next few chapters introduce popular methods, hashing, balanced trees and skip lists that support ADT Table operations.
2. Hashing

- Support ADT Table operations without searching (or with minimum searching)

- Not good for sorted order traversal

- Given a table T with m slots : 0, 1, …, m-1 and n records each with a unique search key k

- Direct Address

  If we have a small set of records, i.e. n <= m with each search key < m, then store record with key k into slot k of table, i.e. T[k]

Example: Table T

<table>
<thead>
<tr>
<th>Index</th>
<th>Record or PointerToRecord</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>…</td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>Record with key = i</td>
</tr>
<tr>
<td>…</td>
<td></td>
</tr>
<tr>
<td>m-1</td>
<td></td>
</tr>
</tbody>
</table>

All operations : O(1)

Insertion() : Insert record with key k into Table[k]
Deletion()  : Delete record R of key k from Table[k]
Retrieval() : Return record R of key k from Table[k]
Hash Table

If the range of keys is big (can be bigger than m-1). It is not practical to build a table with \( m \) = maximum key value.

Solution: Use hash function \( h() \) to map keys into numbers in the range of 0, 1, ..., m-1. i.e. given any key \( k \), \( h(k) \in \{0,1,2,\ldots, m-1\} \)

Example: \( h(k) = k \mod 5 \)

\[
\begin{align*}
h(7) &= 7 \mod 5 = 2 \quad \text{/* put this record into slot 2 */} \\
h(14) &= 14 \mod 5 = 4 \quad \text{/* put this record into slot 4 */}
\end{align*}
\]

<table>
<thead>
<tr>
<th>Index</th>
<th>Record or PointerToRecord</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>( \rightarrow ) record with key = 7</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>( \rightarrow ) record with key = 14</td>
</tr>
</tbody>
</table>

Note: two keys may map into same table index \( \rightarrow \) collision. e.g. key 7 and 12

We would like to study two major issues in hashing

Collision resolution

Designing a good hash function
3. Collision resolution

(i) Open addressing

Each slot in the table can only hold 1 record only, i.e. all records are stored in the table

In this scheme, n <= m, but the key numbers may be (>= m)

Idea:

- Each table entry has additional field to indicate “empty”, “occupied” and “deleted”

- To insert – if the slot is occupied, try another slot until an empty or deleted slot is found (need to define the probing strategy), set indicator to “occupied”

- To delete – follow the defined probe sequence. If the table entry is “deleted”, continue to search. If the table entry is “empty”, return “unsuccessful”. If the record is found, delete the record from the table, set indicator to “deleted”.

- To retrieve – follow the defined probe sequence. If the table entry is “deleted”, continue to search. If the table entry is “empty”, return “unsuccessful”. If the record is found, return the record information.
Probing strategies

- Linear probing: search sequentially until the desired entry is found
  i.e. $T(k) = u$, probe sequence is $u, (u+1) \mod m, (u+2) \mod m. \ldots$ etc

  Problem: primary clustering – table contains groups of consecutively occupied entries

- Quadratic probing: use probe sequence $u, (u+1^2), (u+2^2), (u+3^2), \ldots$ etc

  Problem: secondary clustering – same probe sequence for $T(k') = T(k'') = u$

- Double hashing: use two different hash functions
  $h1(k) = a, h2(k) = b$.

  The probe sequence is $a, a+b, a+2b, a+3b, \ldots$ etc
  This method avoids both primary and secondary clusterings

(ii) Separate chaining

Each slot (bucket) in the table hold a head pointer to a linked list, i.e.
Elements in the same slot are linked together

In this scheme, $n$ can be bigger than $m$ and the key numbers may be $\geq m$

Idea: assume $h(k) = u$, get the head pointer of $T[u]$

- To insert – insert record into the front of the list
- To delete – search and delete the record (if it is found)
- To retrieve – search and retrieve the record (if it is found)
(iii) Analysis:

Assume each key is equally likely to be hashed into any slot

Let $n =$ total number of keys and $m =$ number of slots in the table

load factor $\alpha = \frac{n}{m}$
i.e. how full the table is.

Note: it may be more than 1 (using separate chaining)

From the book “The Art of Computer Programming, Vol. 3” by D. Knuth, the following analysis is provided:

- **Linear probing**:

  $0.5* \left[ 1 + \frac{1}{1-\alpha} \right]$ for a successful search
  $0.5* \left[ 1 + \frac{1}{(1-\alpha)^2} \right]$ for an unsuccessful search

  For example: for $\alpha = 2/3$,
  2 comparisons for successful search
  5 comparisons for unsuccessful search

- **Quadratic probing and double hashing**:

  $\left[ -\log_e(1-\alpha) \right]/\alpha$ for a successful search
  $1/(1-\alpha)$ for an unsuccessful search

  For example: for $\alpha = 2/3$,
  2 comparisons for successful search
  3 comparisons for unsuccessful search
11. Hash Tables

- Separate chaining:
  
  \[ 1 + \frac{\alpha}{2} \] \quad \text{for a successful search}
  
  \[ 1 + \alpha \] \quad \text{for an unsuccessful search}

  note: 1 for getting the table entry

  For example: for a \( \alpha = 2 \)
  
  2 comparisons for successful search
  
  3 comparisons for unsuccessful search

4. Designing a good hash function

What is a good hash function?

- Should be easy and fast to compute

- Should distribute keys uniformly into slots

- Regularity in key distribution should not affect uniformity
  
  i.e. map number 1-100 into slot 1, 101-200 into slot 2 \rightarrow \text{bad function}

Consider integer search keys only. Note: if a key is a string, it can be converted to integer by using the ASCII numbers, then apply the hash function

Two general methods, division and multiplication. May combine them.
11. Hash Tables

Division method, i.e. using mod function

- Example: \( h(k) = h \mod m \)

- Note: don’t pick \( m = 2^p \) → only take last \( p \) bits of \( k \), 
  example \( k = 10110011 \) if \( p = 4 \). Should choose \( m = \text{prime} \)

Multiplication method

- Example function:

  \[ h(k) = m \cdot (k \times A \mod 1) \text{ where } A \text{ is a constant in } (0,1) \]

  choose \( m = 2^p \) and assume \( k \) and \( A \) each has \( w \) bits

  \[
  \begin{array}{ccc}
  k & \times & A \\
  \text{w bits} & = & k \times A \\
  \text{w bits} & \times & 2w \text{ bits}
  \end{array}
  \]

  \[ k \times A \mod 1 = w \text{ bits} \leftarrow \text{a number smaller than 1} \]

  \[ m \cdot (k \times A \mod 1) \rightarrow \text{if } m = 2^p, \text{ shift } p \text{ bits of } (k \times A \mod 1) \text{ to left, and take the integral part} \]

- Example usage:

  \[ w = 7, k = 1101011 (107), A = 0.1011001 \] // each has 7 bits

  \[ m = 2^4 \] // hash slots 0 to 15

  \[ k \times A = 1001010.0110011 \]

  \[ k \times A \mod 1 = .0110011 \]

  \[ m \cdot (k \times A \mod 1) = 0110 = 6 \]

  \[ h(k) = 6 \]
• Example of Hash Function for strings

assume that letters (01XXXXXX) and digits (00XXXXXX) are the most common characters in strings, all the information is in the least significant six bits (ignore 1st and 2nd bits).

//---------------------------------------------------------
// Hash function class for strings
//---------------------------------------------------------

class string_hf
{
    public:
        unsigned int operator()(const string& item) const
        {
            unsigned int n = 0, i, c;
            unsigned int const shift = 6;
            unsigned int const mask = ~0U << (sizeof(unsigned int)*8 - shift);

            // convert a string to a unsigned integer number
            // 1st 6 bits XOR last 24 bits XOR ascii number of char in item[i]
            for (i = 0; i < item.length(); i++)
                n = (n & mask) ^ (n << shift) ^ item[i];
        }
};

More about hash tables in sets and maps……..