Hash Table Notes

**Hashing** is a technique used to perform insertions, removes, and finds in constant average time. Unlike the BST, the average-case running time of hash table operations is based on statistical properties rather than the expectation of random looking input.

This improvement is obtained at the expense of a loss of ordering information among the elements. Operations such as finding a minimum or maximum or printing the entire table in sorted order in linear time are not supported. Thus the hash table has a very different use than a BST would have.

The basic plan of hashing is to save data items in a key-indexed table. The Index is a function of the key. A **hash function** is a method for computing a table index from a key.

**Example 1:** We have data with keys 0, 1, 2. …1000. If this were the case, then we could use the hash function: hash(key) = key (in other words, the key is the index into the table). Data with the key 0 would be stored at index 0 in the table and so forth.

In example 1, we have what is called a perfect hash function. A **perfect hash function** is one that has the characteristic that if key1 is not equal to key2, then hash(key1) is not equal to hash(key2).

**Example 2:** The key for the data is a phone number. Suppose we decide to use as our hash function the last four digits of the phone number. For example if key = 898-2392, then hash(898-2392) = 2392. Then we would need a table size of 10,000 to store all of the entries since the last four digits could range from 0000 to 9999. We would hash the following keys as follows:

- Hash(898-2392) = 2392 -> meaning data with phone# 898-2392 is stored at index 2392 in the table.
- Hash(898-2392) = 2397
- Hash(890-1234) = 1234
- Hash(893-2392) = 2392 -> whoops! Now we have two data items trying to fit in the same spot in the table. This is a problem called a collision (just like two cars trying to park in the same parking spot).

A **collision** occurs when two unique keys are hashed to the same index by the hash function.

Thus we must discuss two things:

1. What makes a good hash function.
2. Collision resolution

**Choosing a “good” hash function:**

The goal would be to “scramble the key” so that each table position is equally likely to be chosen.

**Example 3:** If the keys are social security numbers:

- A **Bad** choice of hash function would be selecting the first three digits. This is bad because social security numbers are assigned in chronological order within a given geographical region. For example, 573 = California; 574 = Alaska.

- A **Better** choice of hash function would be selecting the last three digits.
Example 4: If the keys were a person’s date of birth (1/25/1988).

A Bad choice of hash functions would be the first three digits of the birth year. This would be 198 for the majority of students in class.

A Better choice would be the day of the year on which you were born (1 – 366). This would involve fewer collisions even with only 366 possible values.

Thus what makes a “good” hash function? The following characteristics are all desirable:

- Computation should be fast and easy.
- Collisions should be minimized.
- The hash function should be uniform. In other words, each position in the table is equally likely to be chosen by the function or the probability that index \( k \) in the hash table is selected is \( 1/\text{TABLE\_SIZE} \) for all indices \( k \).

If the data in a table is stable (no inserts or deletes), then a perfect hash function can be developed based on that data.

In general, there will be some inserts and deletes into a hash table so a perfect hash function is not possible. Some common hash functions are:

1. Division method: Let \( \text{TABLE\_SIZE} \) be the size of the table, then \( \text{hash}(\text{key}) = \text{key}\%\text{TABLE\_SIZE} \). The table’s size is critical. If the table size were chosen to be 100, then only the last two digits of the key would be used in the hash function. If the table size were chosen to be an even number, then all keys that were even numbers would be mapped to even numbered indices. If for some reason even keys were predominant in the data set, then the hash function would not be uniform.

Thus if little is known about the keys, the best choice of \( \text{TABLE\_SIZE} \) is a prime number close to the size desired for the table. If a table size of 100 were desired, then the actual table size could be 101 which is a prime number.

2. Middle Square method: If \( K \) is a key, convert it to an unsigned integer, square it’s value and select middle digits from the number.

Example: If \( K = 4839 \), then \( K^2 = 23|4159|21 \) and the index might be selected to be the middle digits 4159.

In reality, this technique actually returns a succession of bits that are extracted from the middle of the square. Assuming that the int type specifies a 32-bit number, the function below extracts the middle bits from the square of the integer key:

```c
unsigned  int hash(int key)
{
    unsigned int value = (unsigned int)item;
    value *= value;  //square the item
    value /= 256;    //discard the low order 8 bits
    return value % 65536; //return item in range 0 to 65535
}
```

The table size is usually a power of 2.
3. **Extraction method:** The hash function selects certain digits from the key. We looked at selecting or extracting the last four digits of a phone number in Example 2. The digits that are extracted do not have to be successive. For example, if your search key is the social security number, then you could select the fourth digit and the last digit so Hash(413568907) = 57. Extraction methods are fast and simple but they may not evenly distribute the items in a hash table. A hash function really should utilize the entire search key.

4. **Folding:**
   a. **Shift folding:** Group the digits in the search key and add the groups. For example, the hash function could form three groups from the key 413568907 as 413 | 568 | 907 then to determine the index to which the key is mapped, the hash function “shifts” each set of digits under the other and adds them:
      
      413  
      568  
      907  
      1888 (so the key maps to index 1888)
   
b. **Folding on the boundaries:** Group the digits in the search key as in shift folding but the middle numbers are folded on the boundary between the first group and the middle group and they are thus reversed. Using the same key as in a. the key 413568907 would form three groups 413 | 568 | 907 as before. Then to determine the index to which the key would be mapped, the second set of numbers would be folded on the first boundary and become 865 so the hash function would determine the key as:
      
      413  
      865  
      907  
      2185 (so the key maps to index 2185)

If a table of size 1000 (for example) were being used for either shift folding or folding on the boundaries, it is common to combine folding with division to obtain an index in the desired range. If folding and division were combined, then one extra step would be added – namely division by 1000 to yield an index of 888 in a or an index of 185 in b.

**Converting a character string to an integer:** If the key is a string, such as a name, the string would need to be converted to an integer before applying the hash function. To do so, you could first assign each character in the string an integer value. For example, for the word “NOTE” you could assign the ASCII values 78, 79, 84, and 69 to the letters ‘N’, ‘O’, ‘T’, and ‘E’ respectively. If you now simply add these numbers, you will get an integer. It will not give a unique integer because any string with the same characters such as “TONE” would yield the same integer. Code to perform this conversion could be given as:

```c
unsigned int hash(string str)
{
    int index = 0;
    for (int j = 0; j < str.length(); j++)
        index += (unsigned int) str[j];
    return index;
}
```

A second way to think of a string is to mimic the way we think of a number like 1234. This number is just $1 \times 10^3 + 2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0$. A character can be represented as an ASCII number in the range
from 0 to 127. Since a character is just a small integer, we can interpret a string as an integer in a manner similar to the way we visualized a number. The string “junk” could be represented as \( j^1 \cdot 128^3 + u^0 \cdot 128^2 + n^1 \cdot 128^1 + k^0 \cdot 128^0 \). The only problem with this strategy is that the integer representation yields a very large number. For “junk”, the hash function would yield 224229227, and longer strings would yield larger representations. Thus we might combine this method of changing a string to an integer with another method such as division to yield a manageable index into a table. A hash function that might be used to perform this can be seen below:

```c
unsigned int hash(string str, int tableSize) {
    int index = 0;
    for (int j = 0; j < str.length(); j++)
        index = (index * 128 + (unsigned int) str[j] % tableSize;
    return index;
}
```

Once we have chosen a good hash function, unless it is a perfect hash function, we must decide what to do about collisions. The method that we use to resolve collisions is called collision resolution.

Collision resolution schemes include:

1. **Open addressing** – when a key collides with another key, the collision is resolved by finding an available table entry other than the address to which the colliding key is originally hashed. Thus if hash(key) results in an index that is already occupied, then the positions in the probe sequence hash(key) + p(1), hash(key) + p(2), hash(key) + p(3), ... hash(key) + p(i), ... are tried until either an available cell is found, the same positions are tried, or the table is full. Here p is the probing function. It may be necessary to normalize hash(key) + p(i) by division modulo the size of the table.

   **Linear probing** is an example of open addressing. In this method, p(i) = i and for the ith probe, the index to be tried is (hash(key)+i) % tableSize.

Example: If we mapped the keys 13, 35, 26, and 46 to a table of size 11 using the division method, we would obtain the following

<table>
<thead>
<tr>
<th>Index</th>
<th>Table Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>________</td>
</tr>
<tr>
<td>1</td>
<td>________</td>
</tr>
<tr>
<td>2</td>
<td><em><strong>13</strong></em></td>
</tr>
<tr>
<td>3</td>
<td><em><strong>35</strong></em></td>
</tr>
<tr>
<td>4</td>
<td><em><strong>26</strong></em></td>
</tr>
<tr>
<td>5</td>
<td><em><strong>46</strong></em></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>________</td>
</tr>
</tbody>
</table>

Hash function hash(key) = key % 11 so:

- hash(13) = 2 so 13 is placed at position 2
- hash(35) = 2 which is a collision so we probe hash(35) + 1 = 3
- hash(26) = 4 so 26 is placed at position 4
- hash(46) = 2 so linear probing says probe 3, 4, 5

How do we know whether an array slot is empty? We must initialize all array slots with a special emptyItem value. Example: if all employees have nonnegative integer keys, use emptyItem = -1.

An example of a retrieve method in a HashTable class that uses linear probing might appear as below:
template<class T>
void HashTable<T>::retrieve(T& item, bool& found)
{
    int location;            //current location in table
    int startLoc;           //starting location in table
    bool moreToSearch = true;  //are we done?
    startLoc = hash(item);  //get the starting location
    location = startLoc;    //record the starting location

    //loop until done
    do{
        //if the item is found or an empty location is found we are done
        if (items[location] == item || items[location] == emptyItem)
            moreToSearch = false;
        else    //we aren’t done so continue to probe
            location = (location + 1) % MAX_SIZE;
    }while (location != startLoc && moreToSearch);

    //did we find the item?
    found = items[location] == item;
    if(found)
        item = items[location];
}

Deleting an item presents a problem. In the example above, suppose we deleted 26. We might think it would be OK to just set position 4 to emptyItem. But then a search for the item 46 would terminate at location 4 because emptyItem is there. To avoid this problem, another special constant called deletedItem would be needed to occupy slots that were occupied by deleted records. Code for the insert and retrieve would need to be modified to accommodate this change.

A problem with linear probing is that primary clustering occurs which means that the table contains groups of consecutively occupied locations.

A second open addressing method is quadratic probing. With quadratic probing, the probe sequence is (hash(key) + i^2) mod table size. If the table size were 11 as above, then the keys 13, 35, 26, and 46 would be mapped as follows:

hash(13) = 2 so place 13 in slot 2
hash(35) = 2 but 2 is occupied so hash(35) + 1^2 = 3 is considered and 35 is placed there
hash(26) = 4 so place 26 in slot 4
hash(46) = 2 so consider hash(35) + 1^2 = 3, and then hash(35) + 2^2 = 6 so place 46 there

Quadratic probing is only guaranteed to visit every position in the table if the table size is a prime number of the form 4K+3 where K is an integer. Note that for the table size 11, K = 2.

A problem with quadratic probing is secondary clustering which means that two items hashed to the same location use the same probe sequence.

A third type of open addressing scheme is double hashing. With double hashing, there are two hash functions. The first hash function is called the primary hash and yields the index at which
the key should be placed. The second hash function is called the secondary hash and it yields the step size to be used in case collision occurs.

An example of double hashing might be if the primary hash were hash1(key) = key mod 11 and the secondary hash were hash2(key) = 7 − (key mod 7). Then the probe sequence for the key 58 would be

\[
\begin{align*}
\text{hash1}(58) &= 58 \mod 11 = 3, \\
3 + \text{hash2}(58) &= 3 + 5 = 8, \\
8 + \text{hash2}(58) &= 8 + 5 = 13 \text{ so mod with 11 to wraparound} = 2 \\
2 + \text{hash2}(58) &= 2 + 5 = 7 \text{ and so on}
\end{align*}
\]

The probe sequence for the key 14 would be 3, 10, 6, 2, 9, 5, 1, 8, 4, 0

2. A second method of collision resolution would be to use buckets. With this method, each location items[i] would be an array called a bucket. With this method, items that hash into items[i] would be placed into the next location in the bucket. With this scheme, the choice of bucket size would have to be made very carefully. If the bucket size is too small, then when the bucket is full, collisions are again a problem. If the bucket size is too large, then a lot of storage is wasted.

3. Separate chaining is the last method that we will consider for collision resolution. With this method, items[i] is a pointer to a linked list – called a chain and consists of items that map to location i. The picture below shows what would happen if separate chaining were used on the example above and the keys 13, 35, 26, and 46 were mapped to the hash table.

```
0:  
1:  
2: 13 46 35  
3: 26  
4:  
. . .
```