Hash Table and Hashing

The tree structures discussed so far assume that we can only work with the input keys by comparing them. No other operation is considered. In practice, it is often true that an input key can be decomposed into smaller units.

1. Integers consist of digits.
2. Strings consist of letters.

Moreover, we can perform operations on these smaller units. For example, we can perform arithmetic operations on the digits. Even for strings, we can map each letter to its ASCII code and then perform arithmetic operations on the numeric codes. Also, we can use them as array indices.

In general,

- Universe $U = \{u_0, u_1, \ldots, u_{N-1}\}$.
- Relatively easy to compute the index of an element.
- Support operations:
  - Find
  - Insert
  - Delete. Deletions may be unnecessary in some applications.
- Unlike binary search tree, AVL tree and B^+- tree, the following functions cannot be done:
  - minimum and maximum.
  - no "approximate" searching like the completion of a URL as in programming assignment 2.
  - successor and predecessor.
  - report data within a given range.
  - list out the data in order.

Example Applications

- Compilers use hash tables (symbol table) to keep track of declared variables.
- Useful when the input keys come in sorted order. This is a bad case for binary search tree. AVL tree and B^+-tree could be difficult to implement and they are not necessarily more efficient.
- On-line spell checkers. After prehashing the entire dictionary, one can check each word in constant time and print out the misspelled word in order of their appearance in the document.

Hashing

- Let $\{K_0, K_1, \cdots, K_N\}$ be the universe of keys.
- Let $T[0..m-1]$ be an array representing the hash table, where m is much smaller than N.
- The soul of the hashing scheme is the hash function

  $$h : \text{Key universe} \longrightarrow [0..m-1]$$

  that maps each key in the universe to an integer between 0 and $m-1$.
- For each key K, $h(K)$ is called the *hash value* of K and K is supposed to be stored at $T[h(K)]$.
- What do we do if two keys have the same hash values?
- There are two aspects to this. We should design the hash function such that it spreads the keys uniformly among the entries of T. This will decrease the likelihood that two keys have the same hash values. Nevertheless, we still need a solution when this unlikely event happens.

Modular Hash Function

- Suppose that we are dealing with strings of letters. A naive attempt is to add up the ASCII codes of the letters in the string, divide the sum by m, and take the remainder. This hash function is easy to implement and runs fast. However, it is a very bad hash function in general because any permutation of the string has the same hash value.
- It is better to improve the hash function so that all the letters are involved and they contribute differently according to their positions. Let $c_{n-1} \cdots c_0$ be a string of letters. We also use c_i to denote the ASCII code of c_i. Note that an ASCII code has value between 0 and 127. Then

$$h(c_{n-1} \cdots c_0) = (\sum_{i=0}^{n-1} c_i * 128^i) \ \% \ m.$$

  Basically, we treat $c_{n-1} \cdots c_0$ as a number represented using base 128, we convert it to the equivalent decimal number, and then take the remainder of the division by the hash table size. This is called a *modular hash function*.
- The same idea can be used for keys that are integers (base = 10) and bit strings (base = 2).

Modular Hash function

- When computing the hash function, one has to be careful about overflow since we may raise the base to a large power. One solution is to do all computation in modulo arithmetic. That is, instead of computing

$$(\sum_{i=0}^{n-1} c_i * 128^i) \ \% \ m,$$

  we compute

```
sum = 0;
for (int j = 0; j < i; j++)
    sum = (sum + (c_j * 128^i) % m ) % m;
```

  Instead of computing $(c_i * 128^i) \ \% \ m$, we compute

$$(c_i * (128^i \ \% \ m)) \ \% \ m.$$

  Instead of computing 128^i, we do:

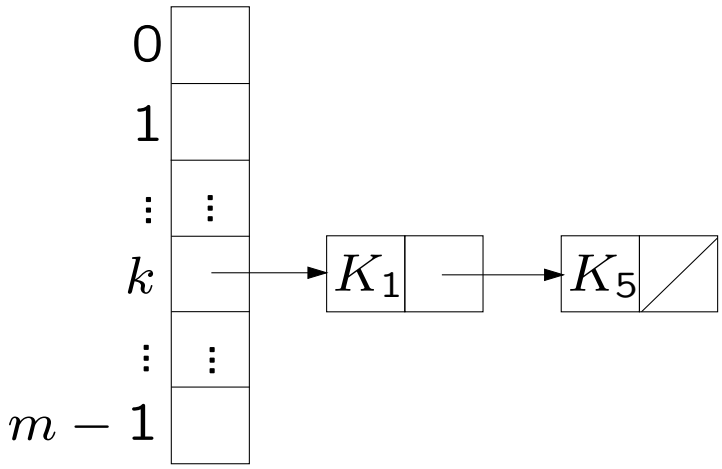
```
product = 1;
for (int j = 0; j < i; j++)
    product = (product * 128) % m;
```

- If $m = 128^k$, then the hash value of a key K is the same as the hash value of the last k letters of K. This is not good since the hash function should take all the letters into consideration. If we are dealing with integers instead of strings of letters, then $m = 10^k$ would have a similar effect.
- The simplest way to avoid such effects is to make m prime.

Collision Handling

- When the hash values of two keys to be stored in the hash table are the same, we have a *collision*.
- Collisions could happen even if we design our hash function carefully since N is much larger than m.
- There are two approaches to *resolve collisions*. We can convert the hash table to a table of linked lists. This is called *separate chaining*. Alternatively, we can relocate a key to a different entry in case of collision. This is called *open addressing*.

Separate Chaining



- Table of linked lists.
- To insert a key K, we compute $h(K)$. If $T[h(K)]$ contains a null pointer, we initialize this entry to point to a linked list that contains K alone. If $T[h(K)]$ is a non-empty list, we add K at the beginning of this list.
- To delete a key K, we compute $h(K)$. Then search for K within the list at $T[h(K)]$. Delete K if it is found.
- Assume that we will be storing n keys. Then we should make m the next larger prime number. If the hash function works well, the number of keys in each linked list will be a small constant. Therefore, we expect that each search, insertion, and deletion can be done in constant time.

Open Addressing

Separate chaining has the disadvantage of using linked lists. Memory allocation in linked list manipulation will slow down the program. An alternative method is to relocate the key K to be inserted if it collides with an existing key. That is, we store K at an entry different from $T[h(K)]$. Two issues arise. First, what is the relocation scheme? Second, how to search for K later?

There are two common methods for resolving a collision in open addressing:

- Linear probing.
- Double hashing.

Linear Probing

- Insertion:
  - Let K be the new key to be inserted. We compute $h(K)$.
  - For $i = 0$ to $m - 1$,
    1. compute $L = (h(K) + i)$ % m,
    2. if $T[L]$ is empty, then we put K there and stop.
  - If we cannot find an empty entry to put K, it means that the table is full and we should report an error.
- Searching:
  - Let K be the new key to be inserted. We compute $h(K)$.
  - For $i = 0$ to $m - 1$,
    1. Compute $L = (h(K) + i)$ % m,
    2. If $T[L]$ contains K, then we are done and we can stop,
    3. If $T[L]$ is empty, then K is not in the table and we can stop too. (If K were in the table, it would have been placed in $T[L]$ by our insertion strategy.)
  - If we cannot find K at the end of the for-loop, we have scanned the entire table and so we can report that K is not in the table.

	Empty table	After 21	After 42	After 31	After 12	After 54
0				31	31	31
1					12	12
2						54
3						
4						
5						
6						
7						
8						
9			42	42	42	42
10		21	21	21	21	21

h(K) = K % 11.

If we search for 31 at the end, we will access the entries T[9], T[10], and T[0]. 31 is found at T[0].

If we search for 20 at the end, then we will access the entries T[9], T[10], T[0], T[1], T[2], and T[3] in this sequence. Since T[3] is empty, we know that 20 is not in the table.

Primary Clustering

We call a block of contiguously occupied table entries a *cluster*. On the average, when we insert a new key K, we may hit the middle of a cluster. Therefore, the time to insert K would be proportional to half the size of a cluster. That is, the larger the cluster, the slower the performance.

Linear probing has the following disadvantages:

- Once $h(K)$ falls into a cluster, this cluster will definitely grow in size by one. Thus, this may worsen the performance of insertion in the future.
- If two cluster are only separated by one entry, then inserting one key into a cluster can merge the two clusters together. Thus, the cluster size can increase drastically by a single insertion. This means that the performance of insertion can deteriorate drastically after a single insertion.
- Large clusters are easy targets for collisions.

Double Hashing

To alleviate the problem of primary clustering, when resolving a collision, we should examine alternative positions in a more "random" fashion. To this end, we work with two hash functions h and h_2.

- Insertion:
  - Let K be the new key to be inserted. We compute $h(K)$.
  - For $i = 0$ to $m - 1$,
    1. compute $L = (h(K) + i * h_2(K))$ % m,
    2. if $T[L]$ is empty, then we put K there and stop.
  - If we cannot find an empty entry to put K, it means that the table is full and we should report an error.
- Searching:
  - Let K be the new key to be inserted. We compute $h(K)$.
  - For $i = 0$ to $m - 1$,
    1. Compute $L = (h(K) + i * h_2(K))$ % m,
    2. If $T[L]$ contains K, then we are done and we can stop,
    3. If $T[L]$ is empty, then K is not in the table and we can stop too. (If K were in the table, it would have been placed in $T[L]$ by our insertion strategy.)
  - If we cannot find K at the end of the for-loop, we have scanned the entire table and so we can report that K is not in the table.

Choice of h_2

- For any key K, $h_2(K)$ must be relatively prime to the table size m. Otherwise, we will only be able to examine a fraction of the table entries. For example, if $h(K) = 0$ and $h_2(K) = m/2$, then we can only examine the entries $T[0]$, $T[m/2]$, and nothing else!
- One solution is to make m prime, choose r to be a prime smaller than m, and set

$$h_2(K) = r - (K \ \% \ r).$$

	Empty table	After 21	After 42	After 31	After 12	After 54
0						
1					12	12
2						
3						
4						54
5						
6						
7						
8				31	31	31
9			42	42	42	42
10		21	21	21	21	21

$h(K) = K \% 10$
$h_2(K) = 7 - (K \% 7)$

Deletion in Open Addressing

Actual deletion cannot be performed in open addressing strategies.

Otherwise, suppose that the table stores three keys K_1, K_2 and K_3 that have identical probe sequences. Suppose that K_1 is stored at $h(K_1)$, K_2 is stored at the second probe location and K_3 is stored at the third probe location. If K_2 is to be deleted and we make the slot containing K_2 empty, then when we search for K_3, we will find an empty slot before finding K_3. So we will report that K_3 does not exist in the table.

Instead, we add an extra bit to each entry to indicate whether it has been deleted or not.