COMP2012H
Spring 2014
Dekai Wu

Hashing

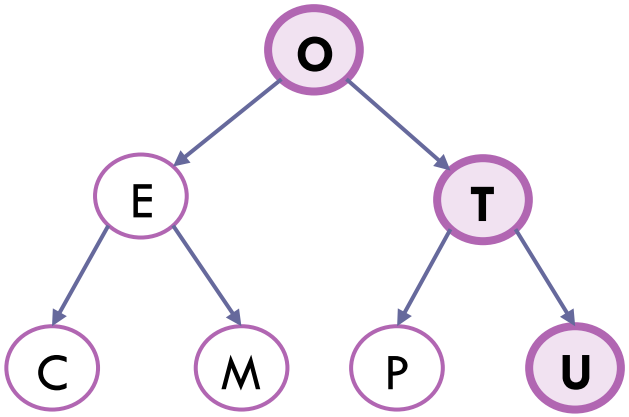
(data structures for the dictionary ADT)

Outline

- Motivation
- Hashing Algorithms and Improving the Hash Functions
- Collisions Strategies
  - Open addressing and linear probing
  - Separate chaining

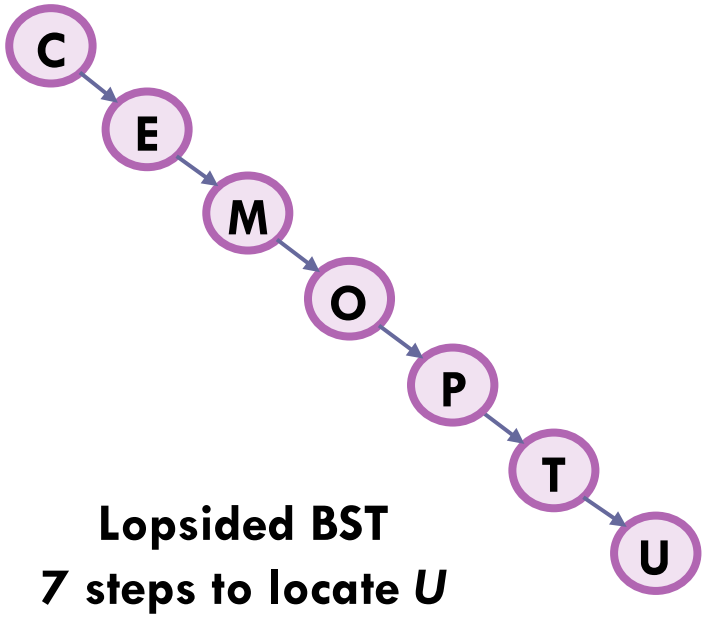
Searching

- BST trees
  - Balanced BST trees (such as an AVL tree) have a search time of O(logN)
  - This is optimal when we consider a comparison-based searching mechanism
- Can we make searching even faster?
  - O(1)?
  - Yes, with Hashing
  - Has some limitations, but for some applications where insertions and deletions are not frequent it is very suitable



Balanced BST

3 steps to locate *U*



Lopsided BST

7* steps to locate *U

Re-thinking Keys Again

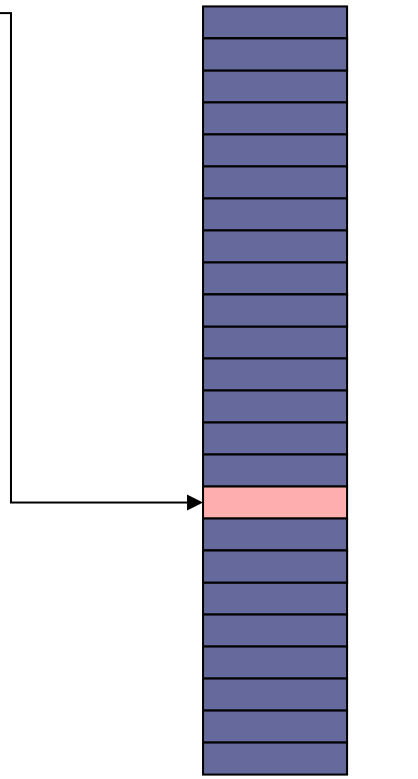
- Tree structures discussed so far assume that we can only use the keys for comparisons, no other operations on the keys are considered

- In practice, however, the key can be decomposed into smaller units, for example:
  1. Integers consists of digits: can be used as array index
  2. Strings consist of letters: We may even perform arithmetic operations on the letters

The idea behind Hashing

- Given a key in the key universe, compute an index into a hash table
- Index is computed using a hash function

Index = hash_function(Key)



Hash Table

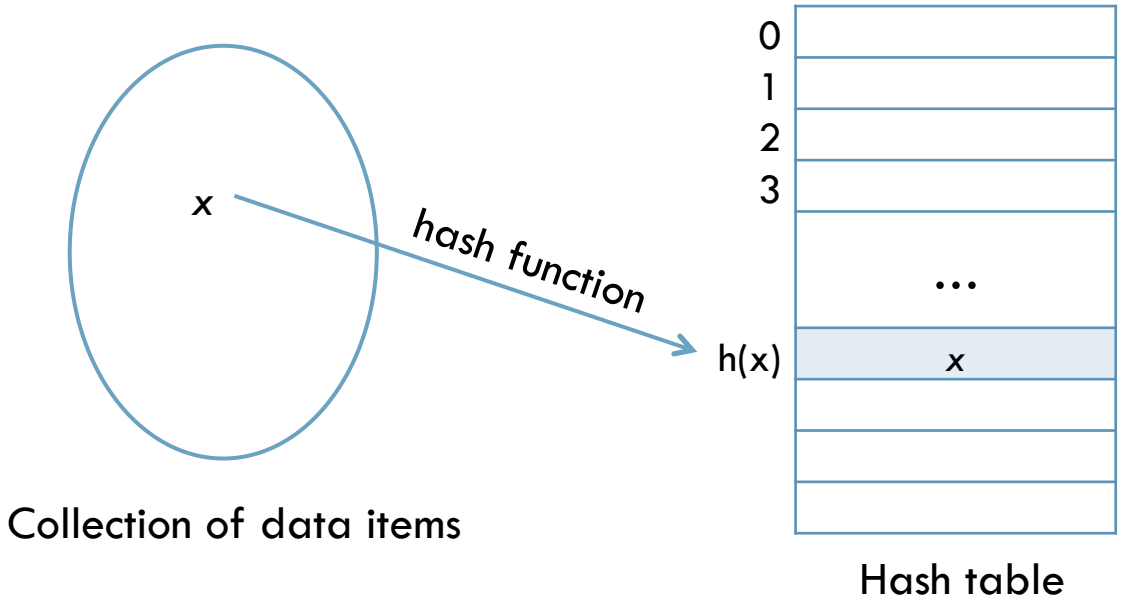
This is an ultra-fast way to search!
Time complexity O(1).

What are the things we need to consider?

1. What is a good hash function for the key.

2. Table size is limited, some keys may map to the same location! (We call this a collision)
Need to resolve collisions.

Hash Tables


x
hash function
0
1
2
3
...
h(x)
x
Collection of data items
Hash table

General Terms and Conditions

- Universe U = $\{u_0, u_1, \ldots, u_{n-1}\}$
- It is relatively easy or efficient to compute some index given a key
- Hash support operations:
  - Find()
  - Insert()
  - Delete()
    - Deletions may be unnecessary in some apps

Hash Tables vs. Trees

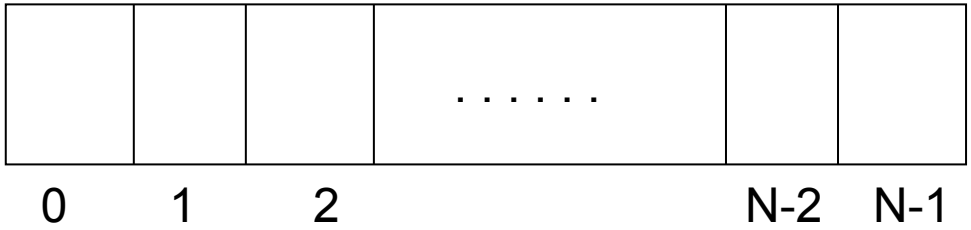
- Hash tables are only for problems that require fast *search*
  - Unlike trees
    - No notion of order
      - remember a tree is sorted if we visit the nodes in-order
    - No notion of successor or predecessor in the data structure
    - Not efficient to find the range
      - min() and max() elements
- Hash tables are generally implemented using an array structure with fixed size.

Example Applications

- Compilers use hash tables to keep track of declared variables

- On-line spell checkers
  - We can “hash” an entire dictionary (or the most common words)
  - Allows us to quickly check if words are spelled correctly in constant time

Bit Vector Representation for O(1) Operations



- entry = 0 if key u_i is absent; otherwise entry = 1
- Find: test entry
- Insert: Set entry to 1
- Delete: Set entry to 0
- Constant time each, independent of the number of keys!

Strengths and Weaknesses

- Advantages
  - Simple implementation
  - Operation can be translated into machine instruction
- Disadvantages
  - Need a bit vector as large as the key space
  - Wastes too much space in general, especially when the number of actual keys is much smaller than the universe
  - E.g., in a dictionary with words of at most 10 characters, we need a bit vector size of $26^{10} \sim 1.4 \times 10^{14}$ bits = 17.6 Tbytes --- This is very large, too large to be held in memory for efficient indexing! In reality, the number of words (keys) is much much smaller than that, i.e., about 200,000 words, and hence we really do not need so much memory to index all the words.

Hashing

- Let $\{K_0, K_1, \ldots K_N\}$ 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 most important part of hashing is the hash function:

  h: Key universe -> [0..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)***]

Hashing

- There are two aspects to the hashing
  - 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
  - Need a solution when the keys do collide

Example of a Hash Function

- Suppose a dictionary of words, and that our keys are a string of letters.
  - Let's consider a letter equals its ASCII value

    Key = $c_{n-1}\ c_{n-2} \ldots c_o$
  - For example:

    'A' = 65, 'Z' = 90, 'a' = 97, 'z' = 122

- Our hash function:

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

(This simply adds the string's characters values up and takes the modulus by m, where m is the size of the hash table.)

Why is This a Bad Hash Function?

- Hash function:

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

- This hash function yields the same results for any permutation of the string
  - *h("CAT") = h("ACT") = h("TAC")*
  - English words have many examples of valid permutations similar to the above
- Need a way to consider the position of the characters in the string

Improving the Hash Function

- We can improve the hash function so that the letters contribute differently according to their positions.

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

Weight by the i^{th} position

- r is the radix
  - Integers: $r = 10$
  - Bit strings: $r = 2$
  - Strings: $r = 128$

Computing the Hash value

- Need to be careful about overflows in summing up the terms, since we may add up to a large number
- We can do all computations in modulo arithmetic by taking modulus at each step
- For example:

```
sum = 0;
for (int j=0; j < n-1; i++)
       sum = (sum + (c_j * r^j) % m ) % m;
```

Take modulus at each step.

Also for the values r^j, you can pre-compute a table to store these values.

"Rule of Thumbs" on Hash Functions

- Hash table size is generally prime
  - Reduces collision due to the modulus operator (% m)

Example: m = 10 (not prime)

100 % 10 -> 0
200 % 10 -> 0
300 % 10 -> 0

m=13 (prime)

100 % 13 -> 9
200 % 13 -> 5
300 % 13 -> 1

- *m* should not divide $r^k \pm a$, for some small integral values of *k* or *a*.
  - As a counter example, if one makes *m=r-1* (i.e., m divides r -1), then all permutations of the same character string have the same hash value
- When we compute hash functions of strings
  - For speed, we often only use a fixed number of elements
  - Such as the first 5 characters of the string
    - example

"Brown" -> h("Brown")
"Schwarzenegger" -> h("Schwa")
"Fong" -> h("FongX") ← For short words, we could use dummy character (like X), or 0,

Linear Open Addressing

- When the key range is too large to use the ideal method, we use a hash table whose size is smaller than the range and a hash function that maps several keys into the same position of the hash table
- The hash function has the form $h(k) = x(k) \% D$
  - where $x(k)$ is the computed value based on key k
  - D is the size (i.e., number of positions) of the hash table
- Each position is called a bucket
- $h(k)$ is the home bucket

Linear Open Addressing

- In case of collision, search for the next available bucket
  - The search for the next available bucket is made by regarding the table as circular
- The choice of *D* has a significant effect on the performance of hashing
- Best results are obtained when *D* is either a prime number or has no prime factors less than 20

Open Addressing via Linear Probing: Insertion

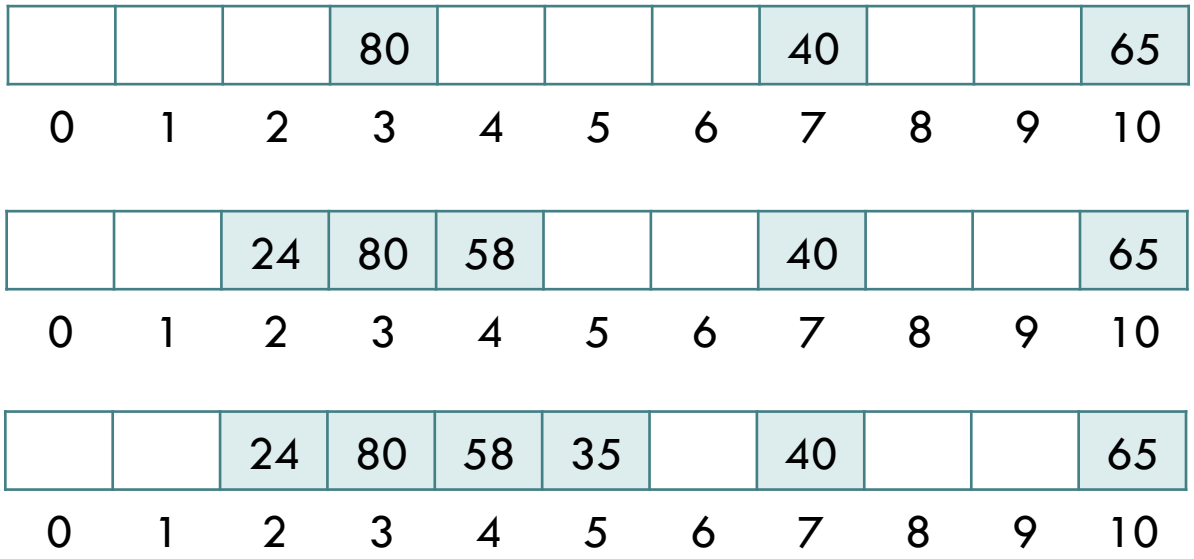
- Compute the home bucket $L = h(K)$
- if $T[L]$ is not empty, consider the hash table as circular:

```
for( i = 0; i < m; i++ )
    compute L = ( h(K) + i ) % m;
    if T[L] is empty, put K there and stop
```

- If we can't find a position (the table is completely full), return an error message

An Example

- $D = 11$
- Add 58: Collision with 80
- Add 24
- Add 35: Collision with 24
- Insertion of 13 will join two clusters



Searching

- Search begins at the home bucket $h(k)$ for the key k
- Continues by examining successive buckets in the table by regarding the table as circular until one of the following happens
  1. A bucket containing the element with key k is reached; in this case, we found the element;
  2. An empty bucket is reached; in which case, the element is not found
  3. We return to the home bucket; in which case, the element is not found

Hash Functions

Strategies for improved performance:

1. Increase table capacity (less collisions)
2. Use a different collision resolution technique (e.g., hierarchical hashing where hashing the second time on the home bucket)
3. Use a different hash function

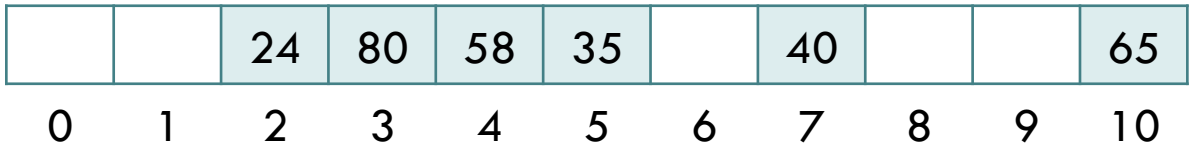
- Hash table capacity
  - Size of table is better to be at least1.5 to 2 times the size of the number of items to be stored
  - Otherwise probability of collisions is too high

Clustering in Linear Probing

- We call a block of contiguously occupied table entries a cluster
- Linear probing becomes slow when large clusters begin to form (primary clustering)
- For example:
  - Once $h(K)$ falls into a cluster, the cluster will definitely grow in size by 1
  - Larger clusters are easier targets for collision in the future
  - If two clusters are only separated by one entry, then inserting one key into a cluster can merge the two clusters

Deletion

- May require several movement: we cannot simply make the position empty. E.g., consider deleting 58



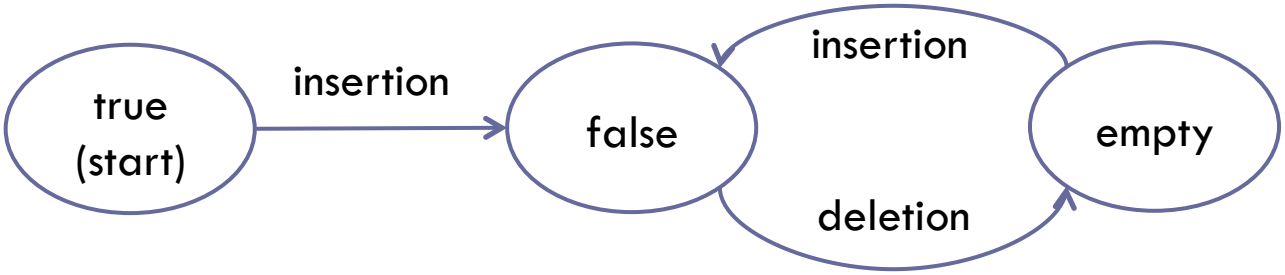
- Move begins just after the bucket vacated by the deleted element. We need to *rehash* buckets one by one in the remainder of the cluster.
- Clearly, it may lead to a lot of movements involving at worst the whole table (of O(m), where m is the size of the table)
- To reduce rehashing overhead, we can simply mark the entry "empty," which means treating it belonging to the cluster but skipping it in a cluster inspection
  - We need to distinguish it from the never-used bucket at the cluster boundary

Introduction of a NeverUsed Field

- A more efficient alternative to reduce move
- Introduce the field *NeverUsed* in each bucket
  - Lazy moving or lazy rehashing
- *NeverUsed* can have only 3 values: {true, false, empty}
- If the entry has never been occupied, *NeverUsed* is *true*
  - *NeverUsed* is set to *true* initially
- When an element is inserted/placed in the bucket, *NeverUsed* is set to *false*
  - Meaning that the bucket has a valid element
- When an element is deleted
  - We simply mark the deleted space as *empty*
  - An empty slot belongs to a cluster but search can simply skip those *empty* buckets in a cluster
  - An element can be inserted into the *empty* slot later, turning the *NeverUsed* to false
- Therefore, deletion may be accomplished by simply setting the position *empty without any data rehashing*
- *NeverUsed* field is never reset to *true,* and hence over time NeverUsed field of the buckets will be either *false* or *empty*

Introduction of a NeverUsed Field

- A new element may be inserted into the first *empty* or *true* bucket encountered after the home bucket



- For an unsuccessful search, the condition for search to terminate is when the *NeverUsed* field equals to *true* (not *empty* as it belongs to the cluster)
  - Which is a bucket never been used before
- After a while, almost all buckets have this *NeverUsed* field equal to *false* or *empty*, and unsuccessful searches examine all buckets
- To improve performance, we must reorganize the table by, for example, rehashing into a new fresh table

Hash ADT (No NeverUsed and Deletion) (1/6)

```
// file hash.h
#ifndef HashTable_
#define HashTable_

#include <iostream.h>
#include <stdlib.h>
#include "xcept.h"

// E is the record
// K is the key
template<class E, class K>
class HashTable {
public:
     HashTable(int divisor = 11);  // constructor
     ~HashTable() {delete [] ht; delete [] empty;}
     bool Search(const K& k, E& e) const;
     HashTable<E,K>& Insert(const E& e);
     void Output();// output the hash table
     // rehashing needs to be done everytime a record is deleted
private:
     int hSearch(const K& k) const;  // helper function for Search and Insert
     int D;         // hash function divisor; hash table size
     E *ht;         // hash table array
     bool *empty;  // 1D array on whether the entry is empty or not
};
```

Hash ADT (No Deletion Method) (2/6)

```
// constructor
template<class E, class K>
HashTable<E,K>::HashTable(int divisor)
{// Constructor.
    D = divisor;
    // allocate hash table arrays
    ht = new E [D];  // holding records
    empty = new bool [D];
    // set all buckets to state true
    for (int i = 0; i < D; i++)
      empty[i] = true;  // no keys/records there
    }
```

Hash ADT (No Deletion Method) (3/6)

```
    // Intermediate function for Search and Insert
    // Search an open addressed table.
    // Return ONE of the following 3 cases:
    // Case 1. If no match, return the next empty bucket from home bucket of k
    // Case 2. If matched, return the matched bucket of k
    // Case 3. If no match and a full hash table, its non-empty home bucket

template<class E, class K>
int HashTable<E,K>::hSearch(const K& k) const
{
    int i = k % D; // home bucket
    int j = i;     // start at home bucket
    do {
        // if end of a cluster or a match, return j
        // Should overload key casting for ht[j] == k
        if (empty[j] || ht[j] == k)
          return j;       // Case 1 or 2
        j = (j + 1) % D; // linearly probe the next bucket
    } while (j != i);    // returned to home?

    return j; // Case 3: wrapped around, table is full, i.e., j == i
}
```

Hash ADT (No Deletion Method) (4/6)

```
template<class E, class K>
bool HashTable<E,K>::Search(const K& k, E& e) const
{
    // Put element that matches k in e.
    // Return false if no match.
    int b = hSearch(k);

    if (empty[b] || ht[b] != k) // implicit cast on k
      return false;  // not found
    e = ht[b];  // need assignment operator
    return true;
}
```

Hash ADT (No Deletion Method) (5/6)

```
template<class E, class K>
HashTable<E,K>& HashTable<E,K>::Insert(const E& e)
{
    // Hash table insert.
    K k = e; // extract key by casting
    int b = hSearch(k);

    // check if insert is to be done
    if (empty[b]) {  // Case 1: not found
        empty[b] = false;
        ht[b] = e;
        return *this;}

    // no insert, check if duplicate or full
    if (ht[b] == k) throw BadInput(); // Case 2: duplicate
    throw NoMem(); // Case 3: table full, cannot insert
}
```

Hash ADT (No Deletion Method) (6/6)

```
template<class E, class K>
void HashTable<E,K>::Output()
{
    for (int i = 0; i< D; i++) {
        if (empty[i]) cout << "empty" << endl;
        else cout << ht[i] << endl; // need << overloading
        }
}

#endif
```

Open Addressing: Linear Probing

- Vulnerable to clustering
- Contiguous occupied positions are easy targets for collisions
- Cluster grows as collision occurs
- Clusters may merge with each other
- This results in increased expected search time
- → Need to spread out the clusters

Performance

- Let b be the number of buckets in the hash table
- n elements are present in the table
- Worst case search and insert time is $\Theta(n)$ (all n keys have the same home bucket)
- Average performance: U_n and S_n be the average number of buckets examined during an unsuccessful and successful search, respectively, and $\alpha = n/b$ be the load factor:

$$U_n \approx \frac{1}{2}\left[1 + \frac{1}{(1-\alpha)^2}\right] \qquad S_n \approx \frac{1}{2}\left[1 + \frac{1}{1-\alpha}\right]$$

An Application Example

- A hash table is to store up to 1,000 elements. Need to find its hash table size.
- Successful searches should require no more than 4 bucket examination on average and unsuccessful searches should examine no more than 50.5 buckets on average
  - i.e., $S_n <= 4$ ➔ $\alpha <= 6/7$
  - i.e., $U_n <= 50.5$ ➔ $\alpha <= 0.9$
- We hence require $\alpha = min(6/7, 0.9) = 6/7$ and therefore b >= 1167
- We choose D to be $37 \times 37 = 1369$ (no prime factors less than 20)

Quadratic Probing

- Insertion
- Compute $L = h(K)$
- Quadric jump away from its home bucket
- If $T[L]$ is not empty:

```
for( i = 0; i < m; i++ )
    compute L = ( h(K) + i*i ) % m;
    if T[L] is empty, put K there and stop
```

- Helps to eliminate primary clustering
- However, if the table gets too full, this approach is not guaranteed of finding an empty slot!
- It may also never visit the home bucket again.

Double Hashing

- To alleviate the problem of primary clustering
- Use a second hash function h_2 when collision occurs
- Resolve collision by choosing the subsequent positions with a constant offset independent of the primary position
- Incrementally jump away from its home bucket in constant step size depending on the key
  - $H(K_i, 0) = h(\text{Ki})$
  - $H(K_i, 1) = (H(K_i, 0) + h_2(K_i)) \bmod m$
  - $H(K_i, 2) = (H(K_i, 1) + h_2(K_i)) \bmod m$
  - . . .
  - $H(K_i, m) = (H(K_i, m-1) + h_2(K_i)) \bmod m$

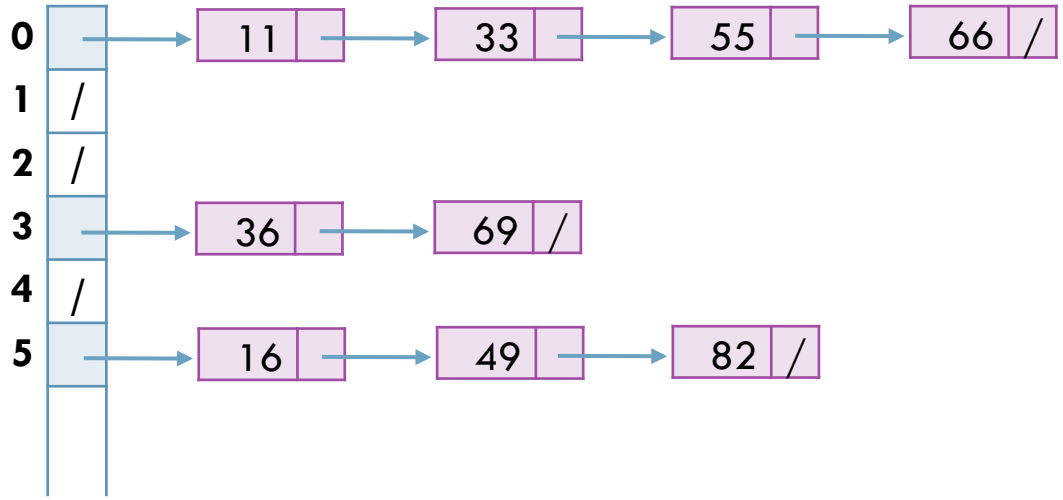
Choice of h_2

- 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 $h_2(K) = m/2$ (not relatively prime to m), then for $h(K) = 0$ we can only examine the entries $T[0]$ and $T[m/2]$ and nothing else!
- The only solution is to make m prime, and choose r to be a prime smaller than m, and set $h_2(K) = r - (K \bmod r)$
  - E.g., if m = 37, we may pick r = 23 and hence $h_2(K) = 23 - K \bmod 23$
  - We may as well use r = 11, and hence $h_2(K) = 11 - K \bmod 11$

Hashing With Chains: Separate Chaining

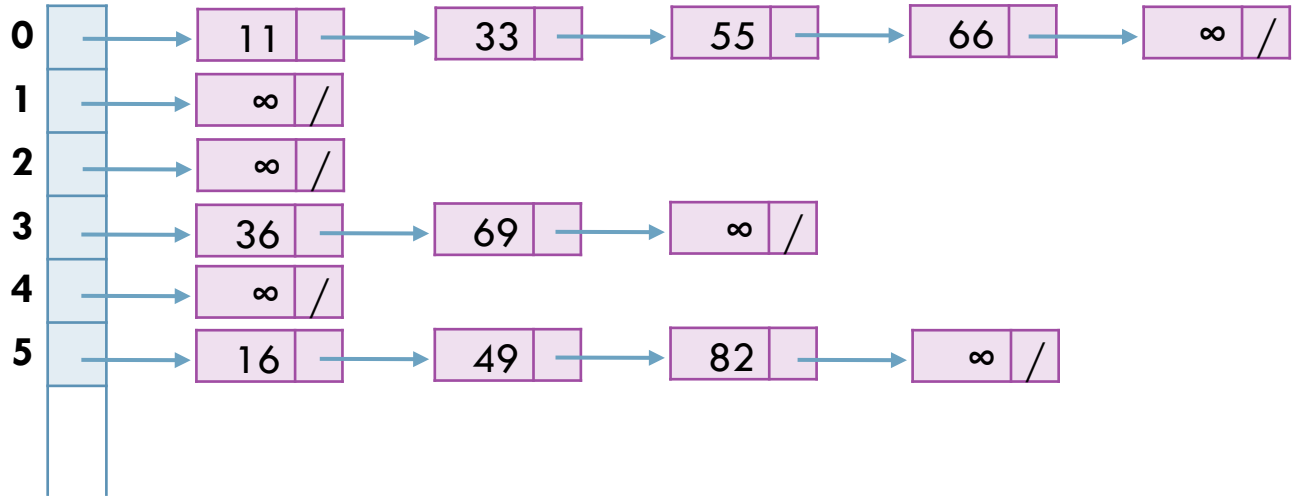
- Maintaining chains of elements that have the same home bucket
- As each insert is preceded by a search, it is more efficient to maintain the chains in ascending order of the key values
- One needs to check two conditions:

```
while ( (node != NULL) && (node -> key < search_key))…
```



Improved Implementation

- Adding a tail node to the end of each chain with key at least as large as any element to be inserted into the table
- Make comparisons more efficient as no NULL comparison is needed (`while (node->key < search_key)`...)



Linear Probing vs. Separate Chaining

- n records of size s bytes; pointer of size p bytes; int takes q bytes
- Hash table has $b >= n$ buckets
- Space requirement
  - Chaining: $bp + n(s+p)$
  - Open linear addressing: $b(s + q)$ (*empty* is an integer array)
  - Chaining takes less space than linear addressing whenever

    $n < b(s+q-p)/(s+p)$
  - The expected performance is superior to that of linear open addressing

Remarks on Hashing

- Ultra-fast searching method
- Best case O(1) (constant time) to find a key
  - Very useful algorithm that requires intensive searching
  - Like spell-checker
  - Some database applications will build temporary hash tables for quick-access to data
- Limited operations as compared to trees
  - Only provides insert, search, and possibly delete
  - No notion of order, min, max, etc.