Hashing

Fall 2020

School of Software Engineering

South China University of Technology

Content

- Linear Searching
- Hashing

Searching

- Suppose we have a collection L of n records of the form $(k_1, I_1), (k_2, I_2), \ldots, (k_n, I_n)$ where I_j is the information associated with key k_j from record j

- Given a particular key value K, the **search problem** is to locate a record (k_j, I_j) in L such that $k_j = K$ (if one exists)

- **Searching** is the systematic method for locating the record(s) with k_j=K.

Searching

- A successful search is one in which a record with key $k_j = K$ is found.
- An unsuccessful search is one in which no record with $k_j = K$ is found (and no such record exists)

- An exact-match query is a search for the record(s) whose key value matches a given key value.
- A range query is a search for all records whose key values fall within a given range of key values.

Searching

- How to perform searching?
  - Sequential and list methods
    - Appropriate for searching data stored in RAM
  - Direct access by key value (hashing)
    - For searching data stored either in RAM or on disk
  - Tree indexing methods
    - Mainly for searching data on disk

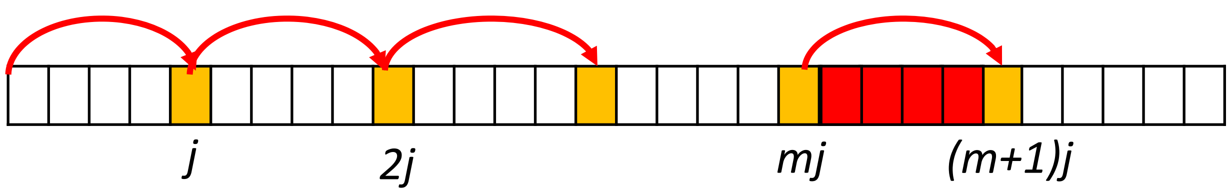
Searching Unsorted Arrays

- Sequential search on unsorted lists requires Θ(n) time in the worst case

- The cost of linear search on average
  - p_i is the probability that K is in position i of list L with $i \in [0, n-1]$;
  - p_n is the probability that K is not in L.
  - When K is in position i, (i+1) comparisons are needed.
  - When K is not in L, n comparisons are needed.
    - The average cost $T(n) = np_n + \sum_{i=0}^{n-1}(i+1)p_i$
    - If all the p_i's are equal (except **p_n**), i.e., $n*p+p_n=1$
    - $T(n) = np_n + \sum_{i=0}^{n-1}(i+1)p = p_n n + p\frac{n(n+1)}{2}$
    $$= \frac{n+1+p_n(n-1)}{2}$$
    - Depending on the value of p_n, $\frac{n+1}{2} \le T(n) \le n$

Searching Sorted Arrays

- When the array elements are sorted
  - One comparison between element i and K may rule out elements from position 0 to i-1 (or elements from position i+1 to n);
- Jump search
  - For some value j, we check every j'th element in L, i.e., L[j], L[2j], and so on。
  - So long as K is greater than the values being checked, we continue
  - If L[mj]<K<L[(m+1)j], we search j-1 elements in the range (L[mj], L[(m+1)j])



  - The total cost (number of comparisons) is

$$\mathbf{T(n, j) = m + j - 1 = \lfloor n/j \rfloor + j - 1}$$

  - When $j = \sqrt{n}$, T(n, j) is minimum.

Searching Sorted Arrays

- Basic principle: divide and conquer
  - selecting a sublist
  - searching a sublist
  - Find a strategy to balance the ‘selecting’ with the ‘searching’

- If we know nothing about the distribution of key values, binary search is the best for searching a sorted array.

- If something about the expected key distribution is known, “computed” binary search (or called dictionary search) is usually called.
  - Ex. if look up for a word starting with ‘S”, jump 19/26 ≈ ¾ of the dictionary

Self-Organizing Lists

- Order records by expected frequency of access, instead of key values.
- Assume p_i is the probability that the record with key k_i will be requested.
  - The most frequently requested record is ordered first in the list; the next most frequently requested record is followed, and so on.
- Sequential search is performed beginning with the first position.

Self-Organizing Lists

- The expected number of comparisons required for one search is
- $$\overline{C}_n = 1\mathrm{p}_0 + 2p_1 + \ldots + np_{n-1}$$
  - $1p_0$ – the number of comparisons to access L[0] is 1, the probability that k_o is requested is p_0.

- Zipf Distributions
  - The distribution of data follows the 80/20 rule.
  - 80% of the records accesses are to 20% of the records
  - If the Zipf frequency for item i in the distribution for n records is $1/(iH_n)$
  - The expected cost will be

$$\overline{C}_n = \sum_{i=1}^{n} i / i\,\mathrm{H}_n = n/\mathrm{H}_n \approx n/\log_e n.$$

  - The average search looks at about 10-15% of the records in a list ordered by frequency.

Self-Organizing Lists

- In most applications, we have no means of knowing in advance the frequencies of access for the data records.
- The probability of access for records might change over time.
- Self-organizing lists uses heuristic strategies for deciding how to reorder the list.
  - Count: store a count of accesses to each record and always maintain records in this order
  - Move-to-front: bring a record to the front of the list when it is found.
  - Transpose: swap any record found with the record immediately preceding it in the list.

Readings

- Reading
  - Chapter 5 Hashing

The Need for Speed

- Data structures we have looked at so far
  - Use comparison operations to find items
  - Need O(log N) time for Find and Insert

- In real world applications, N is typically between 100 and 100,000 (or more)
  - log N is between 6.6 and 16.6

- Hash tables are an abstract data type designed for **O(1)** Find and Inserts

Fewer Functions Faster

- compare lists and stacks
  - by reducing the flexibility of what we are allowed to do, we can increase the performance of the remaining operations
  - insert(L,X) into a list versus push(S,X) onto a stack

- compare trees and hash tables b
  - trees provide for known ordering of all elements
  - hash tables just let you (quickly) find an element

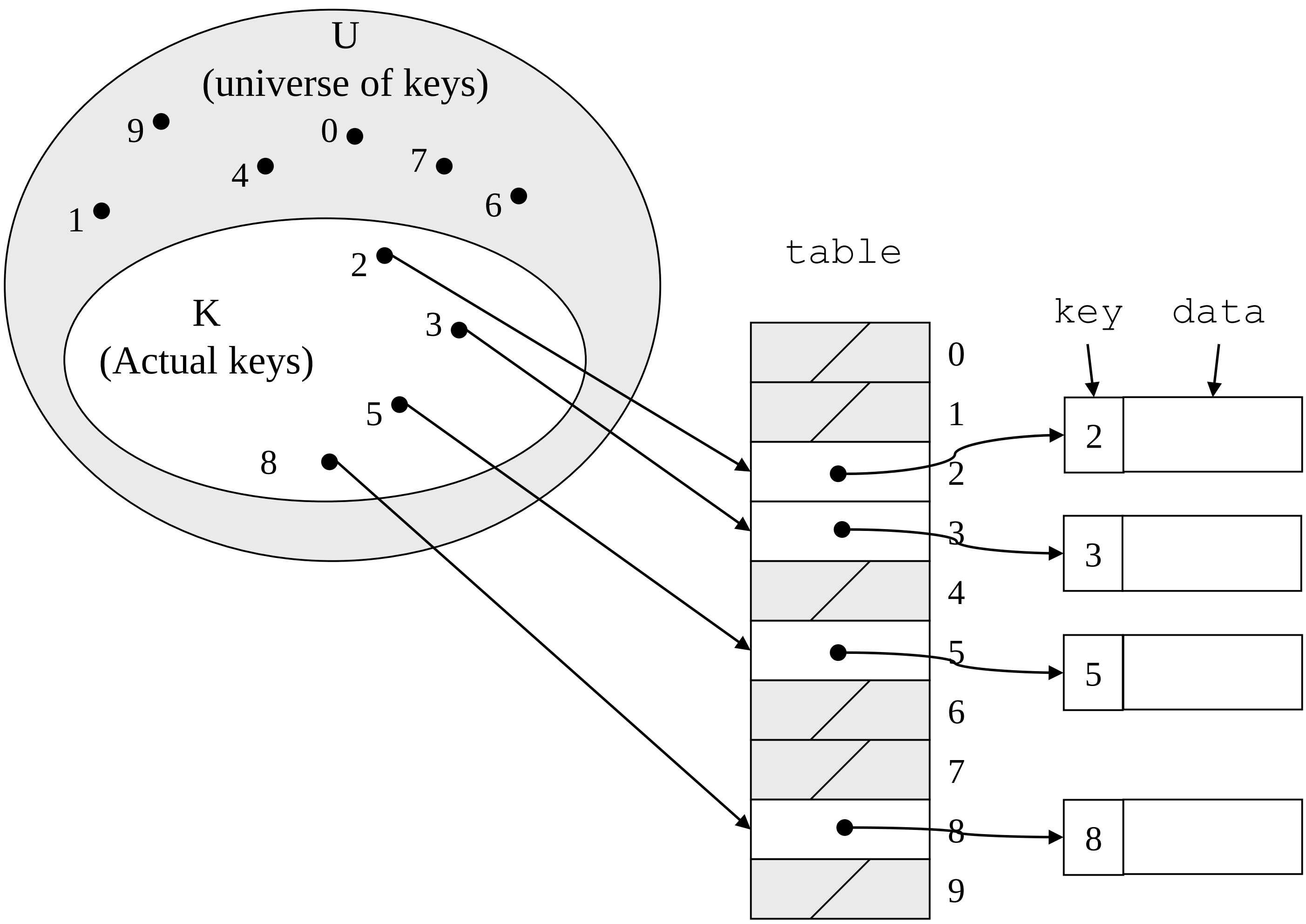
Limited Set of Hash Operations

- For many applications, a limited set of operations is all that is needed
  - Insert, Find, and Delete
  - Note that no ordering of elements is implied
- For example, a compiler needs to maintain information about the symbols in a program
  - user defined
  - language keywords

Direct Address Tables

- Direct addressing using an array is very fast
- Assume
  - keys are integers in the set U={0,1,...m-1}, m is small
  - no two elements have the same key
- Then just store each element at the array location array[key]
  - search, insert, and delete are trivial

Direct Access Table



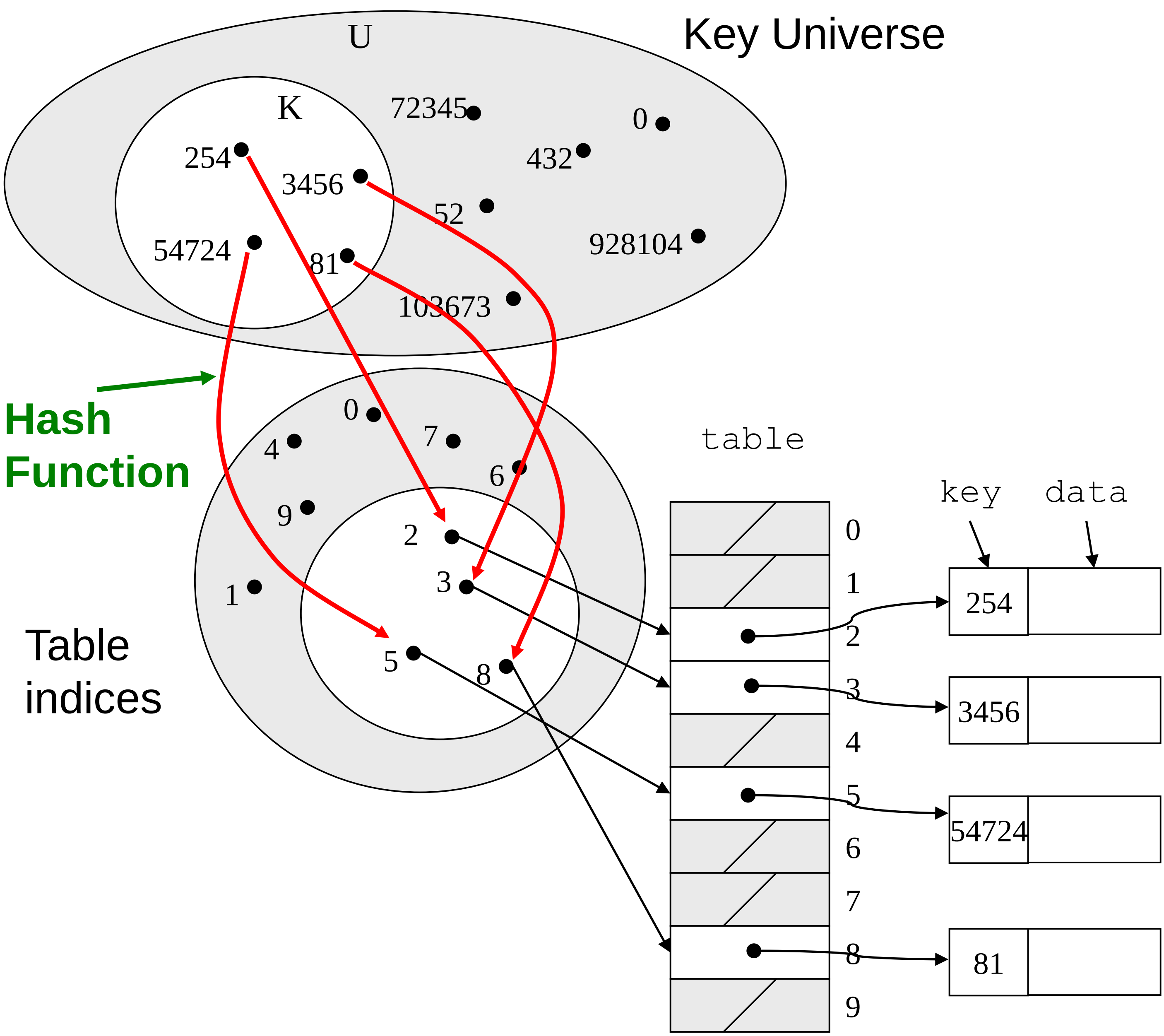
Direct Address Implementation

```
Delete(Table T, ElementType x)
  T[key[x]] = NULL  //key[x] is an integer

Insert(Table T, ElementType x)
  T[key[x]] = x

Find(Table T, Key k)
  return T[k]
```

An Issue

- If most keys in U are used
  - direct addressing can work very well (m small)
- The largest possible key in U , say m, may be much larger than the number of elements actually stored (|U| much greater than |K|)
  - the table is very sparse and wastes space
  - in worst case, table too large to have in memory
- If most keys in U are not used
  - need to map U to a smaller set closer in size to K

Mapping the Keys



Hashing Schemes

- We want to store N items in a table of size M, at a location computed from the key K (which may not be numeric!)

- **Hash function**
  - Method for computing table index from key

- Need of a collision resolution strategy
  - How to handle two keys that hash to the same index

"Find" an Element in an Array

- Data records can be stored in arrays.
  - A[0] = {"CHEM 110", Size 89}
  - A[3] = {"CSE 142", Size 251}
  - A[17] = {"CSE 373", Size 85}
- Class size for CSE 373?
  - Linear search the array – O(N) worst case time
  - Binary search - O(log N) worst case

Go Directly to the Element

- What if we could directly index into the array using the key?
  - A["CSE 373"] = {Size 85}

- Main idea behind hash tables
  - Use a key based on some aspect of the data to index directly into an array
  - O(1) time to access records

Indexing into Hash Table

- Need a fast *hash function* to convert the element key (string or number) to an integer (the *hash value*) (i.e, map from U to index)
  - Then use this value to index into an array
  - Hash("CSE 373") = 157, Hash("CSE 143") = 101
- Output of the hash function
  - must always be less than size of array
  - should be as evenly distributed as possible

Choosing the Hash Function

- What properties do we want from a hash function?
  - Want universe of hash values to be distributed randomly to minimize collisions
  - Don’t want systematic nonrandom pattern in selection of keys to lead to systematic collisions
  - Want hash value to depend on all values in entire key and their positions

The Key Values are Important

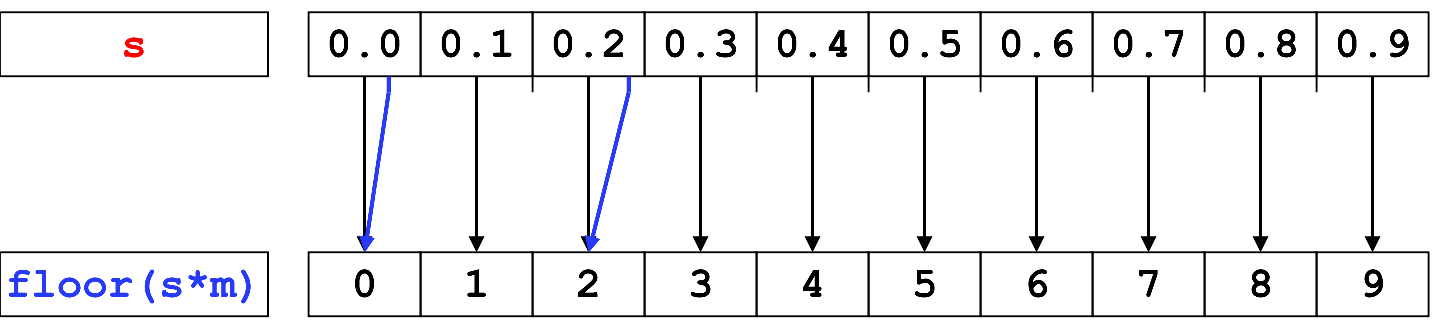
- Notice that one issue with all the hash functions is that the actual content of the key set matters
- The elements in K (the keys that are used) are quite possibly a restricted subset of U, not just a random collection
  - variable names, words in the English language, reserved keywords, telephone numbers, etc, etc

Simple Hashes

- It's possible to have very simple hash functions if you are certain of your keys
- For example,
  - suppose we know that the keys s will be real numbers uniformly distributed over $0 \le s < 1$
  - Then a very fast, very good hash function is
    - hash(s) = floor($s \cdot m$)
    - where m is the size of the table

Example of a Very Simple Mapping

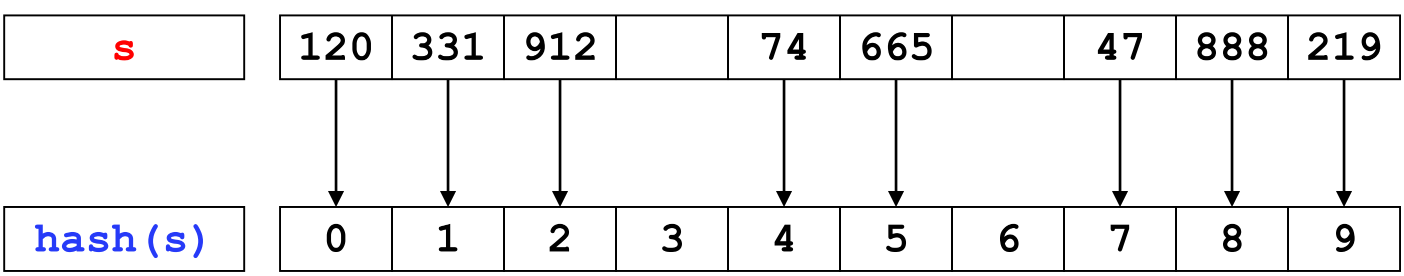
- hash(s) = floor($s \cdot m$) maps from $0 \le s < 1$ to 0..m-1
  - m = 10



Note the even distribution. There are collisions, but we will deal with them later.

Perfect Hashing

- In some cases it's possible to map a known set of keys uniquely to a set of index values
- You must know every single key beforehand and be able to derive a function that works *one-to-one*



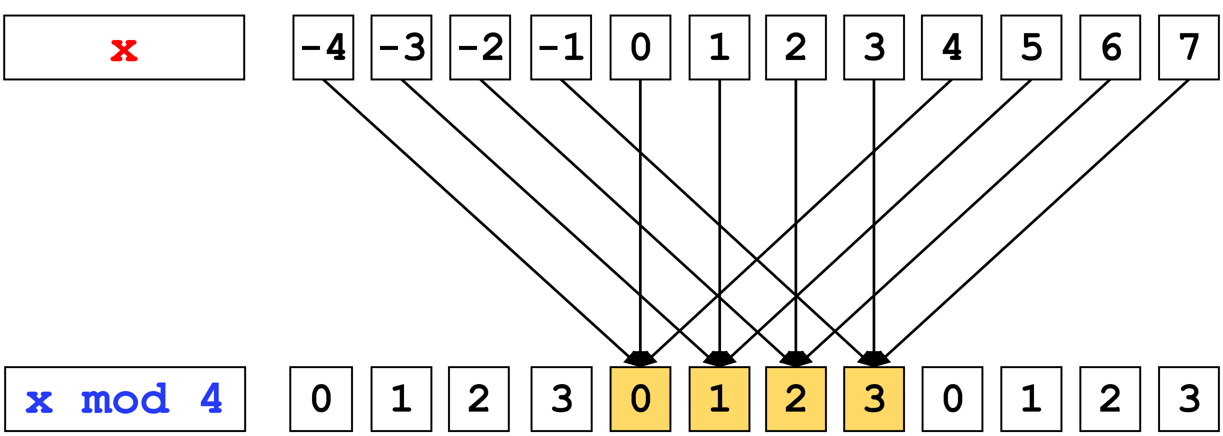
Mod Hash Function

- One solution for a less constrained key set
  - modular arithmetic

- `a` **`mod`** `size`
  - remainder when "a" is divided by "size"
  - in C or Java this is written as **`r = a % size;`**
  - If TableSize = 251
    - 408 mod 251 = 157
    - 352 mod 251 = 101

Modulo Mapping

- *a* mod *m* maps from integers to *0..m-1*
  - one to one? no
  - onto? yes



Hashing Integers

- If keys are integers, we can use the hash function:
  - Hash(key) = key mod TableSize

- Problem 1: What if TableSize is 11 and all keys are 2 repeated digits? (eg, 22, 33, ...)
  - all keys map to the same index
  - Need to pick TableSize carefully: often, a prime number

Hash Functions (III)

- Example 1

```
int h(int x) {
  return(x % 16);
}
```

  - Depends on the least significant four bits of the key, which are likely to be poorly distributed
- Example 2: mid-square method
  - Square the key value, and then take the middle r bits of the results
  - Hash values fall in the range 0 to 2^r-1 bits
  - Most or all bits contribute to the result
  - r=2, K=4567, 4567^2=208**57**489, the result is 57

```
    4567
    4567
--------
   31969
  27402
 22835
18268
--------
20857489
  4567
```

Nonnumerical Keys

- Many hash functions assume that the universe of keys is the natural numbers **N**={0,1,…}
- Need to find a function to convert the actual key to a natural number quickly and effectively before or during the hash calculation
- Generally work with the ASCII character codes when converting strings to numbers

Characters to Integers

- If keys are strings can get an integer by adding up ASCII values of characters in *key*
- We are converting a very large string $c_0c_1c_2 \ldots c_n$ to a relatively small number $(c_0+c_1+c_2+\ldots+c_n)$ mod size.

character →	C	S	E		3	7	3	<0>
ASCII value →	67	83	69	32	51	55	51	0

Hash Must be Onto Table

- Problem 2: What if *TableSize* is 10,000 and all keys are 8 or less characters long?
  - chars have values between 0 and 127
  - Keys will hash only to positions 0 through 8*127 = 1016
- Need to distribute keys over the entire table or the extra space is wasted

Problems with Adding Characters

- Problems with adding up character values for string keys
  - If string keys are short, will not hash evenly to all of the hash table
  - Different character combinations hash to same value
    - "abc", "bca", and "cab" all add up to the same value (recall this was Problem 1)

Characters as Integers

- A character string can be thought of as a base 256 number. The string $c_1c_2 \ldots c_n$ can be thought of as the number
  $c_n + 256c_{n-1} + 256^2c_{n-2} + \ldots + 256^{n-1} c_1$
-
- Use Horner's Rule to Hash! (see Ex. 2.14)

```
r= 0;
for i = 1 to n do
r := (c[i] + 256*r) mod TableSize
```

$$P_n(x) = a_nx^n + a_{n-1}x^{n-1} + \ldots + a_1x + a_o$$
$$= ((\ldots(((a_nx + a_{n-1})x + a_{n-2})x + a_{n-3})\ldots)x + a_1)x + a_o$$

Collisions

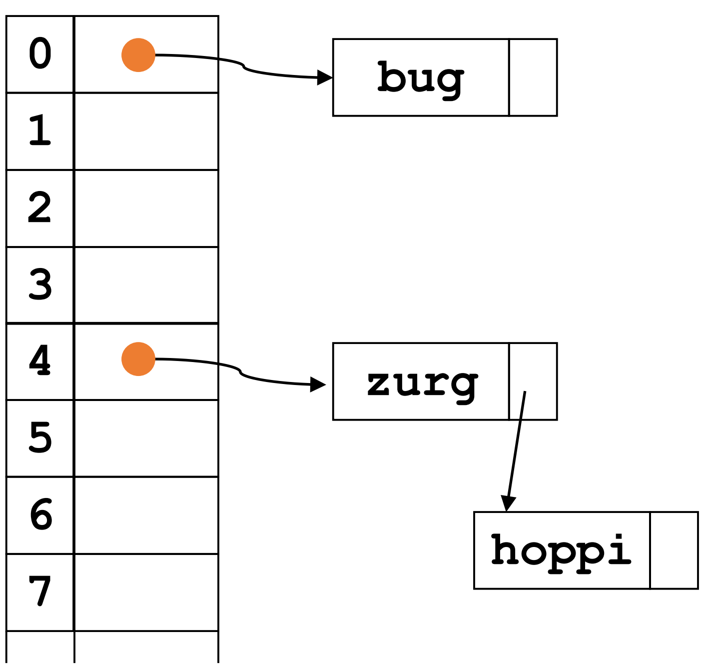
- A collision occurs when two different keys hash to the same value
  - E.g. For *TableSize* = 17, the keys 18 and 35 hash to the same value for the mod17 hash function
  - 18 mod 17 = 1 and 35 mod 17 = 1
- Cannot store both data records in the same slot in array!

Collision Resolution

- Separate Chaining
  - Use data structure (such as a linked list) to store multiple items that hash to the same slot
- Open addressing (or probing)
  - search for empty slots using a second function and store item in first empty slot that is found

Resolution by Chaining

- Each hash table cell holds pointer to linked list of records with same hash value
- Collision: Insert item into linked list
- **To Find an item**: compute hash value, then do Find on linked list
- Note that there are potentially as many as TableSize lists



Why Lists?

- Can use List ADT for Find/Insert/Delete in linked list
  - O(N) runtime where N is the number of elements in the particular chain
- Can also use Binary Search Trees
  - O(log N) time instead of O(N)
  - But the number of elements to search through should be small (otherwise the hashing function is bad or the table is too small)
  - generally not worth the overhead of BSTs

Load Factor of a Hash Table

- Let N = number of items to be stored
- Load factor λ = N/TableSize
  - TableSize = 101 and N =505, then $\lambda = 5$
  - TableSize = 101 and N = 10, then $\lambda = 0.1$

- Average length of chained list = λ and so average time for accessing an item = $O(1) + O(\lambda)$
  - Want λ to be smaller than 1 but close to 1 if good hashing function (i.e. TableSize $\approx$ N)
  - With chaining hashing continues to work for $\lambda > 1$

Resolution by Open Addressing

- No links, all keys are in the table
  - reduced overhead saves space
- When searching for **X**, check locations $h_1(X)$, $h_2(X)$, $h_3(X)$, … until either
  - **X** is found; or
  - we find an empty location (**X** not present)
- Various flavors of open addressing differ in which probe sequence they use
  - When inserting a record and its home position is occupied, the collision resolution method searches a sequence of slots and tries to find a free one for the record
  - Searching in a hash table should follow the same probe sequence used for inserting records

Cell Full? Keep Looking.

- **$h_i(X)=(Hash(X)+p(X,i))$ mod TableSize**
  - Define $p(X,0) = 0$

- F is the collision resolution function. Some possibilities:
  - **Linear:** $p(X,i) = i$
  - **Pseudo-random probing**
  - **Quadratic:** $p(X,i) = i^2$
  - **Double Hashing:** $p(X,i) = i \cdot Hash_2(X)$

Linear Probing

- $p(X,i) = i$
- When searching for `K`, check locations

  `h(K), h(K)+1, h(K)+2,` … mod TableSize
- until either
  - `K` is found; or
  - we find an empty location (`K` not present)
- If table is very sparse, almost like separate chaining.
- When table starts filling, we get clustering but still constant average search time.
- Full table ⇒ infinite loop.

Primary Clustering Problem

- Once a block of a few contiguous occupied positions emerges in table, it becomes a “target” for subsequent collisions

- As clusters grow, they also merge to form larger clusters.

- Primary clustering: elements that hash to different slots probe same alternative slots

Primary Clustering Problem

- Hash table size M=10
- Hash function h(K) = K mod 10
- Probe function p(K,i)= i;

- When the next key value whose home position is 7, 8, 9, 0, 1, and 2, it will end up in slot 2.
- The probability that the next record inserted will end up in slot 2 is 6/10
- The probability that the next record inserted will end up in slot 3, 4, 5, or, 6 is 1/10

0	9050
1	1001
2	
3	
4	
5	
6	
7	9877
8	2037
9	1059

Improved Collision Resolution

- Use linear probing but to skip slots by a constant *c* other than 1
  - Probe function: p(K, i) = *ci*
  - The *i*th slot in the probe sequence (h(K)+c*i*))%M
- How to choose a better *c* value?
  - A good probe sequence is that it will cycle through all slots in the hash table before returning to the home position.
- Better to make c be relatively prime to M.
  - A linear probing sequence may visit all slots in the table
  - If M=10, c can be 1, 3, 7, or 9
  - If M=11, c can be any value between 1 and 10

Improved Collision Resolution

- Does the linear probing with a value of c>1 solve the problem of primary clustering?
  - Sadly, no!
  - With c=2, the probe sequence with $h(k_1)=3$ is 3, 5, 7, 9,… and the probe sequence with $h(k_2)=5$ is 5, 7, 9, …

Improved Collision Resolution

- The ideal probe function selects the next position on the probe sequence at random from among the unvisited slots.
- The probe sequence be a random permutation of the hash table positions
  - Pure randomness? No, the same probe sequence cannot be duplicated when searching for the key
- Pseudo-random probing
  - p(K, i) = Perm[i-1]
  - The *i*th slot in the probe sequence is **(h(K)+ Perm[i-1]) % M**.
  - All insertions and search operations use the same random permutation.

Improved Collision Resolution

- Pseudo-random probing example
  - A hash table with size M=101
  - Perm[0] = 5, perm[1] = 2, perm[2] = 32
  - Assume $h(k_1) = 30$ and $h(k_2) = 35$

  - The probe sequence for k_1 is 30, **35**, 32, and 62
  - The probe sequence for k_2 is **35**, 40, 37, and 67

Quadratic Probing

- When searching for **X**, check locations **$h_1(X)$, $h_1(X)+1^2$, $h_1(X)+2^2$,… mod TableSize** until either
  - **X** is found; or
  - we find an empty location (**X** not present)

- The probe function is some quadratic function

  $$p(K, i) = c_1 i^2 + c_2 i + c_3$$

  for some constants c_1, c_2, and c_3.

  - Example: $p(K, i)=i^2$, the *i*th probing is $(h(K)+i^2)\%M$.
  - The probe sequence for $h(k_1)=30$ is 30, 31, 34, 39, ...
  - The probe sequence for $h(k_2)=29$ is 29, 30, 33, 38,...

  - Two keys with different home positions will have diverging probe sequences.

Quadratic Probing

- Quadratic probing can eliminate primary clustering
- If quadratic probing is used, and the table size is prime, then a new element can always be inserted if the table is at least half empty(textbook page204)

Quadratic Probing

- No primary clustering but secondary clustering possible

- secondary clustering: If two keys hash to the same home position, they will always follow the same probe sequence

- Secondary clustering remains under pseudo-random and quadratic probing

Double Hashing

- When searching for **X**, check locations
$h_1(X)$, $h_1(X)+ h_2(X)$,$h_1(X)+2*h_2(X)$,... mod Tablesize
- until either
  - **X** is found; or
  - we find an empty location (**X** not present)

- $p(K, i)= i*h_2(K)$

- Must be careful about **$h_2(X)$**
  - Not 0 and not a divisor of **M**

Double Hashing

- Example: a hash table with size M=101
  - $h(k_1)=30$, $h(k_2)=28$, $h(k_3)=30$
  - $h_2(k_1)=2$, $h_2(k_2)=5$, $h_2(k_3)=5$

  - The probe sequence for k_1: 30, 32, 34, 36, …;
  - The probe sequence for k_2: 28, 33, 38, 43, …;
  - The probe sequence for k_3: 30, 35, 40, 45, …

Rules of Thumb

- Separate chaining is simple but wastes space…
- Linear probing uses space better, is fast when tables are sparse
- Double hashing is space efficient, fast (get initial hash and increment at the same time), needs careful implementation

Rehashing – Rebuild the Table

- Need to use lazy deletion
  - Need to mark array slots as deleted after Delete
  - consequently, deleting doesn't make the table any less full than it was before the delete
- If table gets too full ($\lambda \approx 1$) or if many deletions have occurred, running time gets too long and Inserts may fail

Rehashing

- Build a bigger hash table of approximately twice the size when λ exceeds a particular value
  - Go through old hash table, ignoring items marked deleted
  - Recompute hash value for each non-deleted key and put the item in new position in new table
  - Cannot just copy data from old table because the bigger table has a new hash function
- Running time is O(N) but happens very infrequently
  - Not good for real-time safety critical applications

Rehashing Example

- Open hashing – $h_1(x) = x \bmod 5$ rehashes to $h_2(x) = x \bmod 11$.

$\lambda = 1$

0	1	2	3	4
25		37 52	83 98	

$\lambda = 5/11$

0	1	2	3	4	5	6	7	8	9	10
			25	37		83		52		98

Rehashing

- Strategies of Rehashing
  - build new table that is about twice as big
  - When?
    - rehash as soon as the table is half full
    - rehash only when an insertion fails
    - middle-of-the-road strategy (best)
      - rehash when the table reaches a certain load factor

Analysis of Open addressing

- Measurements
  - The number of record accesses when performing an operation

  - Operations of concern: insertion, deletion, and **search**
    - Insertion: an unsuccessful search for the record to be inserted (two records with the same key are not allowed)

    - Deletion: a successful search for the record to be deleted

Analysis of Open addressing

- When the hash table is almost empty,
  - The records are very likely to be stored in their home positions
  - Both insertion, deletion, and search require only one record access to find a free slot.
- When the table is getting full,
  - More and more records are likely to be located further from their home position
- The expected cost of hashing is a function of how full the table is, i.e. $f(\lambda)$

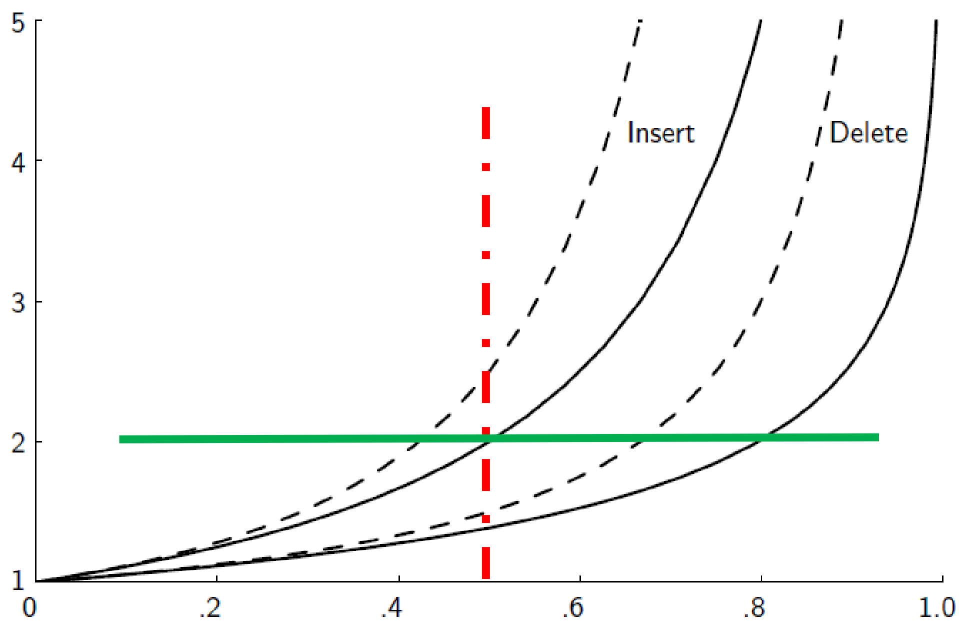
Analysis of Open addressing

- The expected number of probes for an insertion
  - Assume the probe sequence follows a random permutation of the slots, and every slot has equal probability of being the home slot for next record
  - The probability of i collisions is
    $$\frac{N(N-1)\cdots(N-i+1)}{M(M-1)\cdots(M-i+1)} \approx \left(\frac{N}{M}\right)^i$$
  - The expected number of probes is
    $$1 + \sum_{i=1}^{\infty} \left(\frac{N}{M}\right)^i = 1/(1-\lambda)$$

  - The expected cost of has the same cost as originally inserting that record.
    $$\frac{1}{\lambda}\int_0^{\lambda} \frac{1}{1-x}dx = \frac{1}{\lambda} log_e \frac{1}{1-\lambda} = \frac{1}{\lambda} ln \frac{1}{1-\lambda}$$

Analysis of Open addressing

- The true average cost under linear probing is $\frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$ for insertions and $\frac{1}{2}\left(1+\frac{1}{(1-\lambda)}\right)$ for deletions



Analysis of Open addressing

- Rule of thumb: design a hashing system with the hash table never getting above half full.

- Reducing the expect cost of access in the face of collision
  - If two records hash to the same home position, the record with higher frequency of access should be placed in the home position
  - Order records along a probe sequence by their frequency of access

Caveats

- Hash functions are very often the cause of performance bugs.
- Hash functions often make the code not portable.
- If a particular hash function behaves badly on your data, then pick another.
- Always check where the time goes

Exercise 1

- A 7-slot hash table (numbered 0 through 6)
- The hash function h(k) = k mod 7
- Linear probing is used to resolve collision
- Show the table after inserting 3, 12, 9, 2, 10

0	
1	
2	9
3	3
4	2
5	12
6	10

h(3) = 3
h(12) = 5
h(9) = 2
h(2) = 2 – collision
pos = (h(2) + 1) mod 7
= 3 – collision
pos = (h(2) + 2) mod 7 = 4
h(10) = 3 – collision
pos = (h(10) + 1) mod 7
= 4 – collision
pos = (h(10) + 2) mod 7
= 5 – collision
pos = (h(10) + 3) mod 7
= 6

Exercise 2

- A hash table with 13 slots (numbered 0 through 12), use open adrressing hashing with double hashing to resolve collision
- The hash functions are
  H1(k) = k mod 13
  H2(k) = (k+1) mod 11
- Show the table after inserting 2, 8, 31, 20, 19, 18, 53, 26

0	18
1	53
2	2
3	
4	
5	31
6	19
7	20
8	8
9	
10	26
11	
12	

H1(2) = 2
H1(8) = 8
H1(31) = 5
H1(20) = 7
H1(19) = 6
H1(18) = 5 - collision
Pos = (home + 1*H2(18))%13
= (5+8) % 13 = 0
H1(53) = 1
H1(26) = 0 – collision
Pos = (home + 1*H2(26)) % 13
= (0 + 5) % 13 = 5 – collision
Pos = (home + 2*H2(26)) % 13
= (0 + 10) % 13 = 10

Homework

- Self-study:textbook 5.6~5.9
- Exercise 5.1, 5.8,
- Deadline: to be confirmed.