UIT2201 Programming and Data Structures

Hashing Techniques

Chandrabose Aravindan
<AravindanC@ssn.edu.in>

Professor of Information Technology
SSN College of Engineering

July 27, 2022

Introduction

- There are several applications that require a “look-up” table — (key, value) pairs to be stored
- Insertion, search, and deletion based on keys

Introduction

- There are several applications that require a "look-up" table — (key, value) pairs to be stored
- Insertion, search, and deletion based on keys
- FindMin, FindMax, sorting, traversal may not be needed

Introduction

- There are several applications that require a “look-up” table — (key, value) pairs to be stored
- Insertion, search, and deletion based on keys
- FindMin, FindMax, sorting, traversal may not be needed
- Examples: dictionary lookup, symbol table, database indexing

Introduction

- There are several applications that require a “look-up” table — (key, value) pairs to be stored
- Insertion, search, and deletion based on keys
- FindMin, FindMax, sorting, traversal may not be needed
- Examples: dictionary lookup, symbol table, database indexing
- We may use Binary Search Tree with balancing

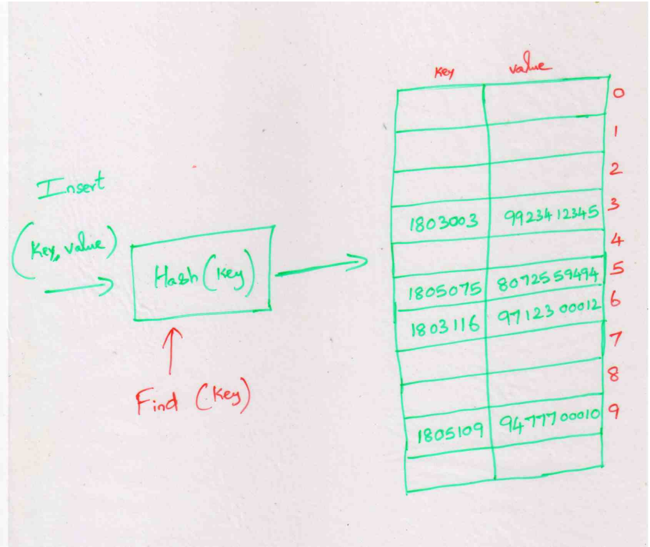
Introduction

- There are several applications that require a "look-up" table — (key, value) pairs to be stored
- Insertion, search, and deletion based on keys
- FindMin, FindMax, sorting, traversal may not be needed
- Examples: dictionary lookup, symbol table, database indexing
- We may use Binary Search Tree with balancing
- Complexity: $O(\log n)$

Introduction

- There are several applications that require a "look-up" table — (key, value) pairs to be stored
- Insertion, search, and deletion based on keys
- FindMin, FindMax, sorting, traversal may not be needed
- Examples: dictionary lookup, symbol table, database indexing
- We may use Binary Search Tree with balancing
- Complexity: $O(\log n)$
- Is it possible to do this in constant time (on the average)!?

Basic Idea



Insert
(Key, value)
Hash (Key)
Find (Key)
Key
value
0
1
2
1803003
9923412345
3
4
1805075
8072559494
5
1803116
9712300012
6
7
8
1805109
9477700010
9

Issues

- Selection of hashing function is very crucial to achieve constant time on the average
- Hashing function should ideally handle any type of key (non-mutable)
- It should be easy to compute!
- Hashing function should ideally generate distinct indexes for distinct keys
- Generally, the "key density" is very high compared to the number of pairs we wish to store
- How do we choose appropriate TableSize?
- What to do when the hashing function maps more than one key to an index (collision handling)?

Hash Functions

- The purpose of a hash function is to map a key to an index

Hash Functions

- The purpose of a hash function is to map a key to an index
- Generally, it can be viewed as two distinct parts — hash code generation and a compression function

Hash Functions

- The purpose of a hash function is to map a key to an index
- Generally, it can be viewed as two distinct parts — hash code generation and a compression function
- Hash code is to map a key to an integer (not necessarily in the range of the Table)

Hash Functions

- The purpose of a hash function is to map a key to an index
- Generally, it can be viewed as two distinct parts — hash code generation and a compression function
- Hash code is to map a key to an integer (not necessarily in the range of the Table)
- Hash code may typically be a 32-bit integer

Hash Functions

- The purpose of a hash function is to map a key to an index
- Generally, it can be viewed as two distinct parts — hash code generation and a compression function
- Hash code is to map a key to an integer (not necessarily in the range of the Table)
- Hash code may typically be a 32-bit integer
- Compression function should map a hash code to a legal index of the Table

Simple Compression Function

- If the key is an integer, then a simple hash function (key mod TableSize) may work
- If the key is not an integer, then it should first be converted into a hash code before performing the modulus operation

Simple Compression Function

- If the key is an integer, then a simple hash function (key mod TableSize) may work
- If the key is not an integer, then it should first be converted into a hash code before performing the modulus operation
- A simple analysis reveals that the keys are well distributed across the index space when TableSize is a prime number
- For example, consider the hash codes $[200, 205, 210, 220, \cdots, 600]$ with TableSize of 100 Vs TableSize of 101

Multiply-Add-and-Divide (MAD) function

- An alternate compression function to avoid patterns is the Multiply-Add-and-Divide (MAD) method

Multiply-Add-and-Divide (MAD) function

- An alternate compression function to avoid patterns is the Multiply-Add-and-Divide (MAD) method
- A hash code i is compressed to an index by

$$[(ai + b) \bmod p] \bmod \mathrm{TableSize}$$

- Here, p is a prime number larger than TableSize, and a and b are random numbers chosen from the interval $[0, p-1]$, with $a > 0$

Multiply-Add-and-Divide (MAD) function

- An alternate compression function to avoid patterns is the Multiply-Add-and-Divide (MAD) method
- A hash code i is compressed to an index by

$$[(ai + b) \bmod p] \bmod \mathrm{TableSize}$$

- Here, p is a prime number larger than TableSize, and a and b are random numbers chosen from the interval $[0, p-1]$, with $a > 0$
- Probability of collision is approximately $1/\mathit{TableSize}$

Hash Codes

- How do we generate a hash code for some arbitrary key?

Hash Codes

- How do we generate a hash code for some arbitrary key?
- Internal bit representation of the key (object) may be used (that is to interpret the bit representation as an integer)

Hash Codes

- How do we generate a hash code for some arbitrary key?
- Internal bit representation of the key (object) may be used (that is to interpret the bit representation as an integer)
- But the bit sequence may be long and we may need only a 32-bit hash code

Hash Codes

- How do we generate a hash code for some arbitrary key?
- Internal bit representation of the key (object) may be used (that is to interpret the bit representation as an integer)
- But the bit sequence may be long and we may need only a 32-bit hash code
- We can break the bit sequence into chunks of 32 bits, and either add them or take exclusive-or of those chunks

Hash Codes

- For strings, Unicode / ASCII representation of each character may be added up

Hash Codes

- For strings, Unicode / ASCII representation of each character may be added up
- This idea may not work because the sequence information is lost
- For example, "stop", "tops", "pots", and "spot" will have same hash code!

Hash Codes

- For strings, Unicode / ASCII representation of each character may be added up
- This idea may not work because the sequence information is lost
- For example, “stop”, “tops”, “pots”, and “spot” will have same hash code!
- Another bad case is when TableSize is quite large and strings are small
- Example: TableSize $= 10007$ and strings are 8 characters at most
- Only indexes up to ($127 * 8 = 1016$) are used! (we have assumed ASCII representation here)

Hash Codes

- We may do the following to generate a larger integer

$$key[0] + 27 * key[1] + 27^2 * key[2]$$

Hash Codes

- We may do the following to generate a larger integer

$$key[0] + 27 * key[1] + 27^2 * key[2]$$

- Why only 3 characters? $26^3 = 17576$ combinations!

Hash Codes

- We may do the following to generate a larger integer

$$key[0] + 27 * key[1] + 27^2 * key[2]$$

- Why only 3 characters? $26^3 = 17576$ combinations!
- But, in reality there may be less than 3000 combinations!!!

Hash Codes

- We may do the following to generate a larger integer

$$key[0] + 27 * key[1] + 27^2 * key[2]$$

- Why only 3 characters? $26^3 = 17576$ combinations!
- But, in reality there may be less than 3000 combinations!!!
- And the computation gets complex when the number of characters are increased

Polynomial Hash Codes with Horner's rule

- The above computation may be simplified as follows (referred to as Horner's rule)

$$k_0 + 27k_1 + 27^2k_2 = ((27 * k_2) + k_1) * 27 + k_0$$

Polynomial Hash Codes with Horner's rule

- The above computation may be simplified as follows (referred to as Horner's rule)

$$k_0 + 27k_1 + 27^2k_2 = ((27 * k_2) + k_1) * 27 + k_0$$

- This idea may be extended to more characters

$$\sum_{i=0}^{KeySize-1} Key[KeySize - i - 1] * 32^i$$

Polynomial Hash Codes with Horner's rule

- The above computation may be simplified as follows (referred to as Horner's rule)

$$k_0 + 27k_1 + 27^2 k_2 = ((27 * k_2) + k_1) * 27 + k_0$$

- This idea may be extended to more characters

$$\sum_{i=0}^{KeySize-1} Key[KeySize - i - 1] * 32^i$$

- The magic number 32 is used here, since multiplication can then be achieved by left shift!

Polynomial Hash Codes with Horner's rule

- The above computation may be simplified as follows (referred to as Horner's rule)

$$k_0 + 27k_1 + 27^2k_2 = ((27 * k_2) + k_1) * 27 + k_0$$

- This idea may be extended to more characters

$$\sum_{i=0}^{KeySize-1} Key[KeySize - i - 1] * 32^i$$

- The magic number 32 is used here, since multiplication can then be achieved by left shift!
- Text book suggests $33, 37, 39, 41$ as good options for English character strings

Cyclic-Shift Hash Codes

- This is a variant of polynomial hash codes
- The multiplication is replaced by cyclic shift by a certain number of bits
- For example, $10110\cdots$ is replaced by $\cdots 10110$
- This operation has no mathematical meaning, but seems to generate better hash codes!

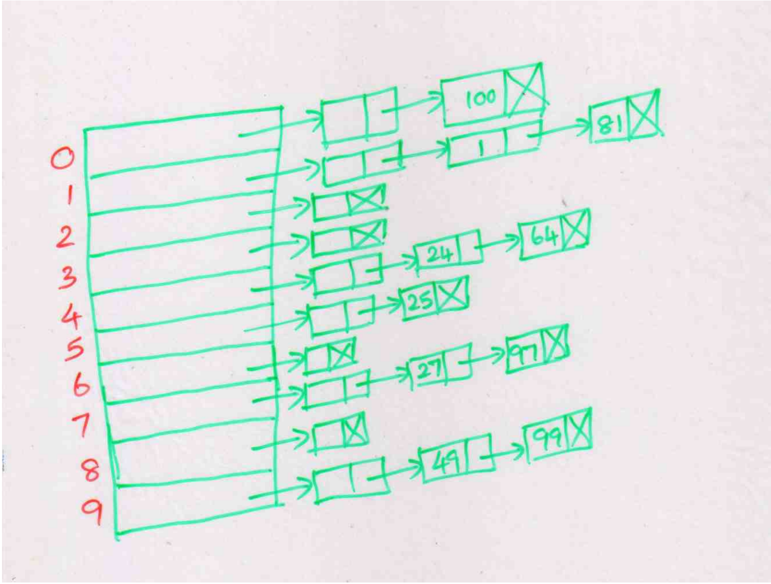
Collision Handling

- Separate Chaining — use additional data structures to store the pairs having the same home index
- Open Addressing — use only the array and systematically look for an alternate index
    - Linear Probing
    - Quadratic Probing
    - Double Hashing
- Rehashing to increase or reduce the TableSize
- Extendible Hashing

Separate Chaining

- Idea here is to use another collection data structure to keep all the pairs that hash to the same index
- A simple linked list will be an ideal choice!

Separate Chaining



Complexity

- Is it possible to achieve constant average time with this design?

Complexity

- Is it possible to achieve constant average time with this design?
- What will be the average length of a linked list?

Complexity

- Is it possible to achieve constant average time with this design?
- What will be the average length of a linked list?
- Load factor λ: Ratio of number of pairs to the TableSize

Complexity

- Is it possible to achieve constant average time with this design?
- What will be the average length of a linked list?
- Load factor λ: Ratio of number of pairs to the TableSize
- Average length of a linked list will be λ

Complexity

- Is it possible to achieve constant average time with this design?
- What will be the average length of a linked list?
- Load factor λ: Ratio of number of pairs to the TableSize
- Average length of a linked list will be λ
- We may achieve constant average time, if we can keep the load factor λ close to 1 and the hash function distributes the keys well

Complexity

- Is it possible to achieve constant average time with this design?
- What will be the average length of a linked list?
- Load factor λ: Ratio of number of pairs to the TableSize
- Average length of a linked list will be λ
- We may achieve constant average time, if we can keep the load factor λ close to 1 and the hash function distributes the keys well
- Some other data structure, such as BST or another hash table, may be tried instead of linked lists, but may not be worth the effort

Closed hashing

- Separate chaining has some disadvantages — use of references with several levels of indirection, need to implement an additional data structure, etc.

Closed hashing

- Separate chaining has some disadvantages — use of references with several levels of indirection, need to implement an additional data structure, etc.
- Alternate solution: Find, in a systematic way, an alternate cell in the array

Closed hashing

- Separate chaining has some disadvantages — use of references with several levels of indirection, need to implement an additional data structure, etc.
- Alternate solution: Find, in a systematic way, an alternate cell in the array
- Try the indexes $h_0(Key), h_1(Key), h_2(Key), \ldots$, where

$$h_i(Key) = (Hash(Key) + F(i)) \ \% \ TableSize$$

Closed hashing

- Separate chaining has some disadvantages — use of references with several levels of indirection, need to implement an additional data structure, etc.
- Alternate solution: Find, in a systematic way, an alternate cell in the array
- Try the indexes $h_0(Key), h_1(Key), h_2(Key), \ldots$, where

$$h_i(Key) = (Hash(Key) + F(i))\ \%\ TableSize$$

- Note that $F(0)$ must be 0!

Closed hashing

- Separate chaining has some disadvantages — use of references with several levels of indirection, need to implement an additional data structure, etc.
- Alternate solution: Find, in a systematic way, an alternate cell in the array
- Try the indexes $h_0(Key), h_1(Key), h_2(Key), \ldots$, where

$$h_i(Key) = (Hash(Key) + F(i))\ \%\ TableSize$$

- Note that $F(0)$ must be 0!
- The function F is referred to as a collision resolution probing strategy

Issues in closed hashing

- How do we know that the array cell corresponding to the index returned by Hash(Key) is free or not?

Issues in closed hashing

- How do we know that the array cell corresponding to the index returned by Hash(Key) is free or not?
- Let's say, key k_1 is stored at some location, key k_2 which collides with k_1 is stored at an alternate location
- What happens when k_1 is deleted!?

Issues in closed hashing

- How do we know that the array cell corresponding to the index returned by Hash(Key) is free or not?
- Let's say, key k_1 is stored at some location, key k_2 which collides with k_1 is stored at an alternate location
- What happens when k_1 is deleted!?
- We are forced to use lazy deletion

Issues in closed hashing

- How do we know that the array cell corresponding to the index returned by Hash(Key) is free or not?
- Let's say, key k_1 is stored at some location, key k_2 which collides with k_1 is stored at an alternate location
- What happens when k_1 is deleted!?
- We are forced to use lazy deletion
- And, we should add status field (Valid, Empty, Deleted) to each entry in the table

Linear Probing

- When F is a linear function of i, probing becomes linear

Linear Probing

- When F is a linear function of i, probing becomes linear
- The simplest linear probing function is $F(i) = i$
- That means, we will first try the home index Hash(key). If it is occupied, we will sequentially try the subsequent cells until an empty cell is found

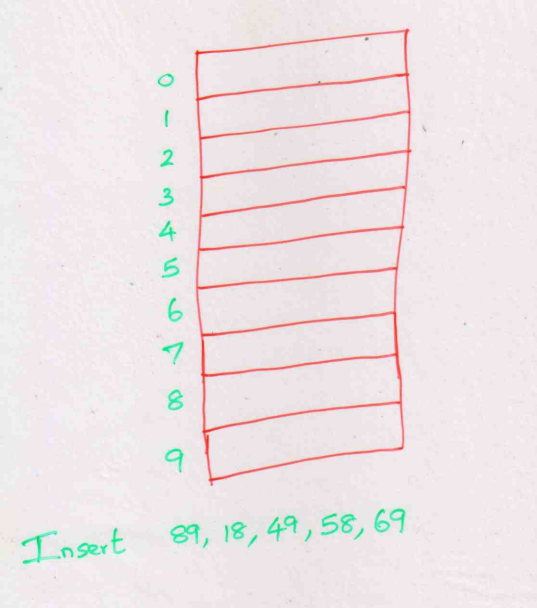
Linear Probing

- When F is a linear function of i, probing becomes linear
- The simplest linear probing function is $F(i) = i$
- That means, we will first try the home index Hash(key). If it is occupied, we will sequentially try the subsequent cells until an empty cell is found
- Probing is done with wraparound — circular array

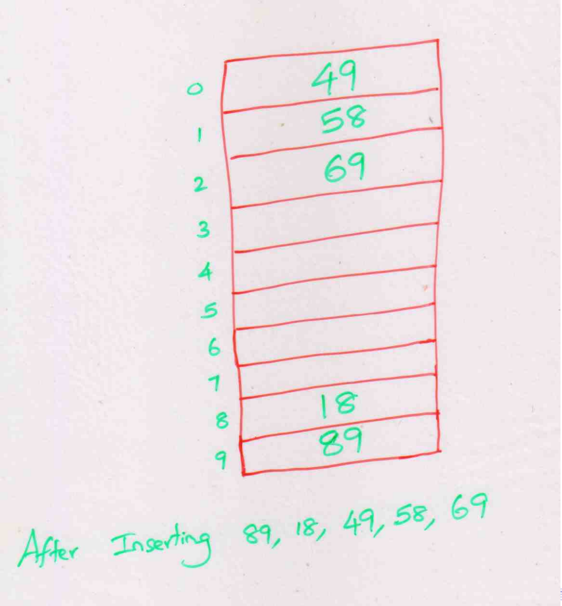
Linear Probing

- When F is a linear function of i, probing becomes linear
- The simplest linear probing function is $F(i) = i$
- That means, we will first try the home index Hash(key). If it is occupied, we will sequentially try the subsequent cells until an empty cell is found
- Probing is done with wraparound — circular array
- A new object can be inserted as long as there are free cells in the array

Linear Probing



Linear Probing



Analysis of linear probing

- An important drawback of linear probing is "primary clustering" — formation of blocks of objects in the array

Analysis of linear probing

- An important drawback of linear probing is "primary clustering" — formation of blocks of objects in the array
- This increases the number of probes required as the load factor increases

Analysis of linear probing

- An important drawback of linear probing is "primary clustering" — formation of blocks of objects in the array
- This increases the number of probes required as the load factor increases
- Analysis shows that the expected number of probes is

$$\frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$$

Analysis of linear probing

- An important drawback of linear probing is "primary clustering" — formation of blocks of objects in the array
- This increases the number of probes required as the load factor increases
- Analysis shows that the expected number of probes is

$$\frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$$

- When $\lambda = 0.5$, this works out to be about 2.5 probes

Analysis of linear probing

- An important drawback of linear probing is "primary clustering" — formation of blocks of objects in the array
- This increases the number of probes required as the load factor increases
- Analysis shows that the expected number of probes is

$$\frac{1}{2}\left(1 + \frac{1}{(1-\lambda)^2}\right)$$

- When $\lambda = 0.5$, this works out to be about 2.5 probes
- But, increases to about 8.5 when $\lambda = 0.75$, and to approximately 50 probes when $\lambda = 0.9$

Analysis of linear probing

- An important drawback of linear probing is "primary clustering" — formation of blocks of objects in the array
- This increases the number of probes required as the load factor increases
- Analysis shows that the expected number of probes is

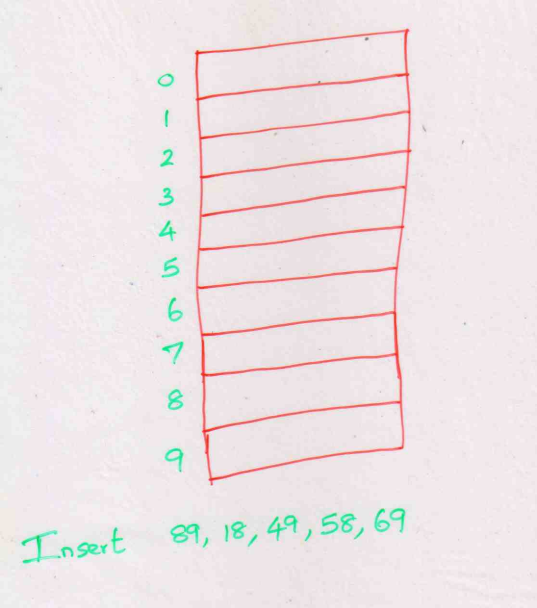
$$\frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$$

- When $\lambda = 0.5$, this works out to be about 2.5 probes
- But, increases to about 8.5 when $\lambda = 0.75$, and to approximately 50 probes when $\lambda = 0.9$
- Ideally, λ should be kept below 0.5

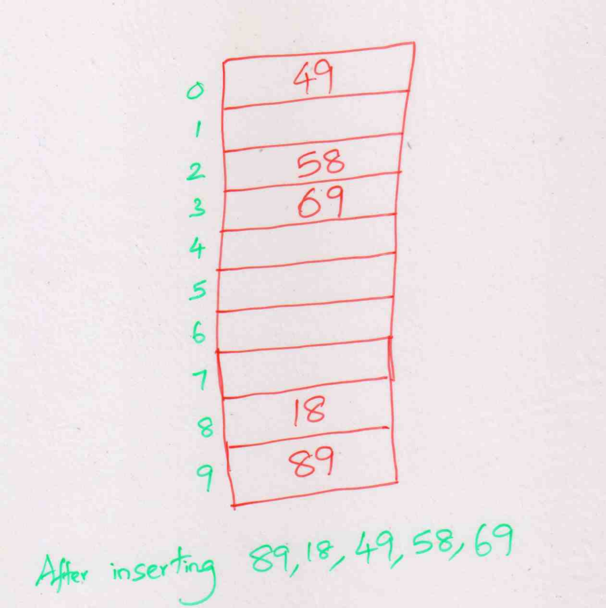
Quadratic probing

- Quadratic probing may be tried to eliminate the primary clustering problem
- $F(i) = i^2$ is a popular choice!
- The number of alternative locations may be severely reduced

Quadratic Probing



Quadratic Probing



Analysis of quadratic probing

- Since not all cells are probed, insertion may fail even when there are free cells in the array

Analysis of quadratic probing

- Since not all cells are probed, insertion may fail even when there are free cells in the array
- No guarantee of finding an empty cell when the load factor is more than 0.5

Analysis of quadratic probing

- Since not all cells are probed, insertion may fail even when there are free cells in the array
- No guarantee of finding an empty cell when the load factor is more than 0.5
- However, it has been proved that insertion is always possible if the table is at least half empty and TableSize is prime

Analysis of quadratic probing

- Since not all cells are probed, insertion may fail even when there are free cells in the array
- No guarantee of finding an empty cell when the load factor is more than 0.5
- However, it has been proved that insertion is always possible if the table is at least half empty and TableSize is prime
- For all the keys that are mapped to the same index, same alternate locations are tried resulting in "secondary clustering"

Double hashing

- One interesting option for probing is to use another hash function!
- $F(i) = i \cdot Hash_2(Key)$

Double hashing

- One interesting option for probing is to use another hash function!
- $F(i) = i \cdot Hash_2(Key)$
- The second hash function selects a step size for linear probing
- Note that the step size is different for different keys!

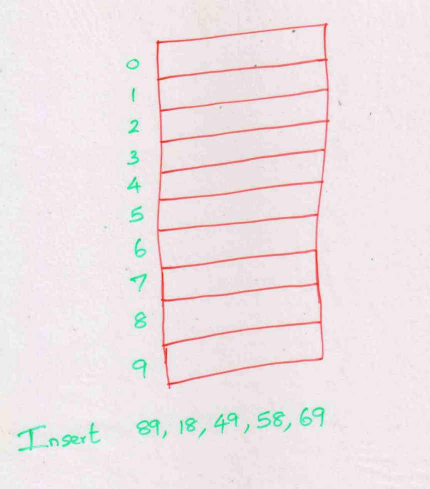
Double hashing

- One interesting option for probing is to use another hash function!
- $F(i) = i \cdot Hash_2(Key)$
- The second hash function selects a step size for linear probing
- Note that the step size is different for different keys!
- Also note that this second hash function should not result in 0 for any key! (Why?)

Double hashing

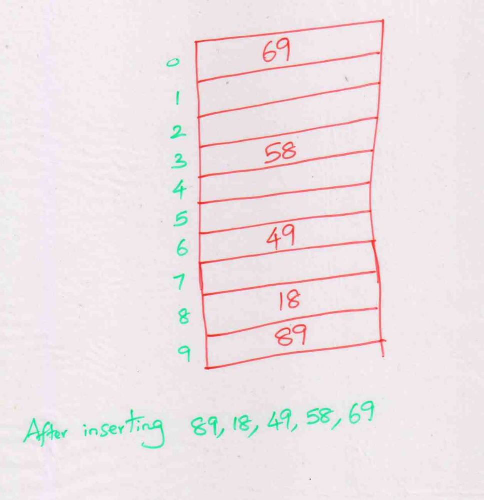
- One interesting option for probing is to use another hash function!
- $F(i) = i \cdot Hash_2(Key)$
- The second hash function selects a step size for linear probing
- Note that the step size is different for different keys!
- Also note that this second hash function should not result in 0 for any key! (Why?)
- Typically, $Hash_2(Key) = R - (Key \% R)$, where R is a prime number smaller that TableSize

Double hashing

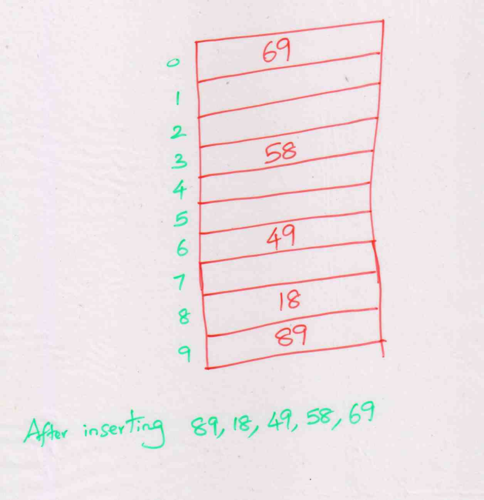


- Let us choose $R = 7$

Double hashing



Double hashing



- Try inserting 60 and 23!

Rehashing

- For closed hashing to be successful, load factor λ needs to be kept under control

Rehashing

- For closed hashing to be successful, load factor λ needs to be kept under control
- One solution is to dynamically control the TableSize

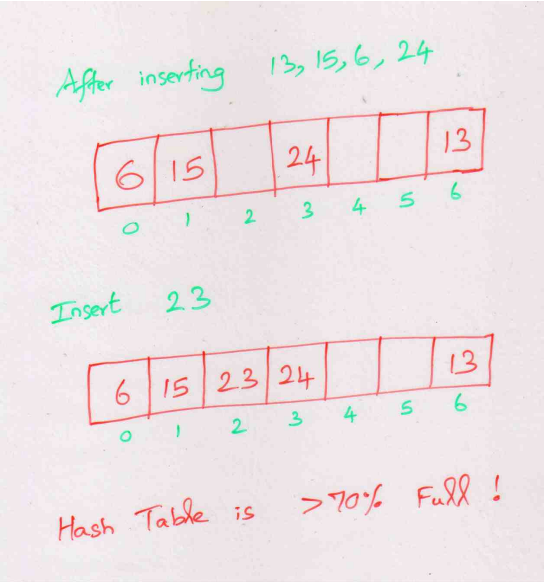
Rehashing

- For closed hashing to be successful, load factor λ needs to be kept under control
- One solution is to dynamically control the TableSize
- When the load factor crosses a threshold, array size may be doubled and all the objects be rehashed to the new array

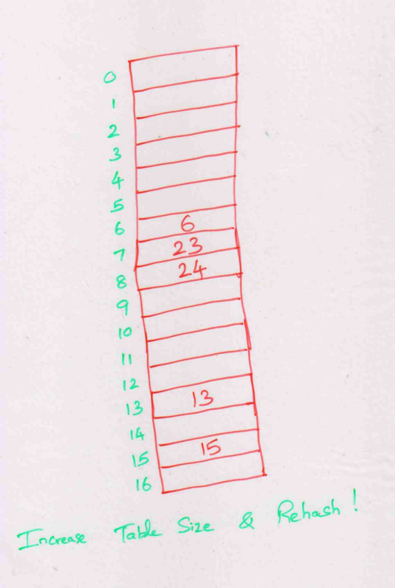
Rehashing

- For closed hashing to be successful, load factor λ needs to be kept under control
- One solution is to dynamically control the TableSize
- When the load factor crosses a threshold, array size may be doubled and all the objects be rehashed to the new array
- Rehashing is a good opportunity to purge all the deleted objects!

Rehashing


After inserting 13, 15, 6, 24
6
15
24
13
0 1 2 3 4 5 6
Insert 23
6
15
23
24
13
0 1 2 3 4 5 6
Hash Table is >70% Full !

Rehashing


0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
6
23
24
13
15
Increase Table Size & Rehash !

Analysis

- Entire array needs to be scanned and all the n objects need to be rehashed

Analysis

- Entire array needs to be scanned and all the n objects need to be rehashed
- Hence the complexity is $O(n)$

Analysis

- Entire array needs to be scanned and all the n objects need to be rehashed
- Hence the complexity is $O(n)$
- However, next rehashing is likely to occur only after n more insertions

Analysis

- Entire array needs to be scanned and all the n objects need to be rehashed
- Hence the complexity is $O(n)$
- However, next rehashing is likely to occur only after n more insertions
- So, the amortized complexity can be considered as constant

Cryptographic hash functions

- Let's say a string s is hashed to a 32-bit integer I, that is, hashCode(s) $= I$

Cryptographic hash functions

- Let's say a string s is hashed to a 32-bit integer I, that is, hashCode$(s) = I$
- By knowing I, is it possible to find s or some other string t, such that hashCode$(t) = I$?

Cryptographic hash functions

- Let's say a string s is hashed to a 32-bit integer I, that is, $\text{hashCode}(s) = I$
- By knowing I, is it possible to find s or some other string t, such that $\text{hashCode}(t) = I$?
- If the answer is "No" or "infeasible", then such a hash function will be suitable for cryptographic applications!

Cryptographic hash functions

- Let's say a string s is hashed to a 32-bit integer I, that is, $\text{hashCode}(s) = I$
- By knowing I, is it possible to find s or some other string t, such that $\text{hashCode}(t) = I$?
- If the answer is "No" or "infeasible", then such a hash function will be suitable for cryptographic applications!
- Examples: Password hashing, checksum of data for integrity check, message digest

Cryptographic hash functions

- Let's say a string s is hashed to a 32-bit integer I, that is, $\text{hashCode}(s) = I$
- By knowing I, is it possible to find s or some other string t, such that $\text{hashCode}(t) = I$?
- If the answer is "No" or "infeasible", then such a hash function will be suitable for cryptographic applications!
- Examples: Password hashing, checksum of data for integrity check, message digest
- Popular choices: SHA-2 and SHA-3 (Keccak) families of algorithms, MD5 (MD6 has also been introduced), RIPEMD, BLAKE, etc.

Summary

- Several applications need collections that support only insertion, deletion, search
- Hashing is an ideal solution that can achieve constant average time
- We have discussed some simple hash functions and the issues involved
- Collision is a major issue in implementing hashing technique
- Separate chaining is one of the solutions to handle collision — use a secondary data structure, such as linked lists, to store all the objects hashing to the same index
- Load factor λ needs to be close to 1 for effective separate chaining

Summary

- We have also explored the open addressing hashing (also known as closed hashing) techniques in this lecture
- In particular, we discussed three probing techniques, namely linear, quadratic, and double hashing
- Analysis reveals that the load factor should be kept below 0.5 for effective closed hashing
- Rehashing may be necessary to control the load factor
- Hash codes are also useful in several cryptographic applications