HASHING

Motivation

- Arrays provide an indirect way to access a **set**.
- Many times we need an association between two sets, or a set of **keys** and associated data.
- Ideally we would like to access this data directly with the keys.
- We would like a data structure that supports fast search, insertion, and deletion.
  - Do not usually care about sorting.
- The abstract data type is usually called a **Dictionary** or **Partial Map**
- **Applications are in Symbol table ,Direct Access files.**

Dictionaries

- What is the best way to implement this?
  - Linked Lists?
  - Double Linked Lists?
  - Queues?
  - Stacks?
  - Multiple indexed arrays (e.g., data[key[i]])?
- To answer this, ask what is the complexity of the :
  - Insertion
  - Deletion
  - Search operations.

Direct Addressing

- Let's look at an easy case, suppose:
  - The range of keys is 0..m-1
  - Keys are distinct
- Possible solution
  - Set up an array T[0..m-1] in which
    - T[i] = x If $x \in T$ and key[x] = i
    - T[i] = NULL otherwise
  - This is called a *direct-address table*
    - Operations take O(1) time!
    - *So what's the problem?*

Direct Addressing

- Direct addressing works well when the range *m* of keys is relatively small
- But what If the keys are 32-bit integers?
  - Problem 1: direct-address table will have 2^{32} entries, more than 4 billion
  - Problem 2: even If memory is not an issue, the time to initialize the elements to NULL may be
- Solution: map keys to smaller range 0..*p*-1
  - Desire $p = \boldsymbol{O}(m)$.

Hash Tables- overview

- All search structures so far
  - Relied on a comparison operation
  - Linear Performance *O(n)*
  - Non linear *O(* log *n)*
  - **Assume I have a function**
  - $f(key) \rightarrow integer$

    I.e. one that maps a key to an integer
- What performance might I expect now?

Hash Table

key – DBMS primary key Unique key

- A *hash table* is a list in which each member is accessed through a *key*.
- The key is used to determine where to store the value in the table. Table
- The function that produces a location from the key is called the *hash* function.
- For example, if it were a hash table of strings, the hash function might compute the sum of the ASCII values of the first 5 characters of the string, modulo the size of the table.

Hashing

- **Component of Hashing**
- **Hash key :** Unique attribute from data set.
- **Hash function :**maps key K into an address.(integer value)
- Hash Table : Array / linked list

Example ::

keyword — Symbol table, Static Symbol table, keyboard static, Symbol table

collision

unique hash

- For example, if the string were "**return**", then the key would **be**

(114 + 101 + 116 + 117 + 114) % 100 = 62

roll key %80 — Table size

- Thus, "return" would be located in position 62 of the hash table.

Unique code — XYZXY — 62 — hashcode

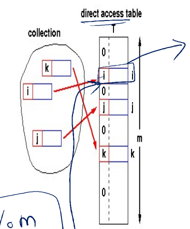
62 return

9

9

Hash Tables - Structure

- Simplest case:
  - Assume items have integer keys in the range 1 .. *m*
  - Use the value of the key itself to select a slot in a direct access table in which to store the item
  - To search for an item with key, *k*, just look in slot *k*
    - If there's an item there, you've found it
    - If the tag is 0, it's missing.
  - Constant time, *O*(1)



i%m = location O(1)

10

10

Hash Tables – Constraints Hash Key

- Constraints
  - Keys must be unique
  - Keys must lie in a small range
  - For storage efficiency, keys must be dense in the range
  - If they're sparse (lots of gaps between values), a lot of space is used to obtain speed
    - Space for speed trade-off

11

11

Hash Table

- Hash Tables provide ***O***(1) support for all of these operations!
- The key is rather than index an array directly, index it through some function, ***h***(*x*), called a *hash function*.
  - myArray[*h*(index)]
- Key questions:
  - What is the set that the *x* comes from?
  - What is *h()* and what is its range?

12

12

Hash Functions

- A hash function, ***h***, maps keys of a given type to integers in a fixed interval *[0, N − 1]*
- Example:

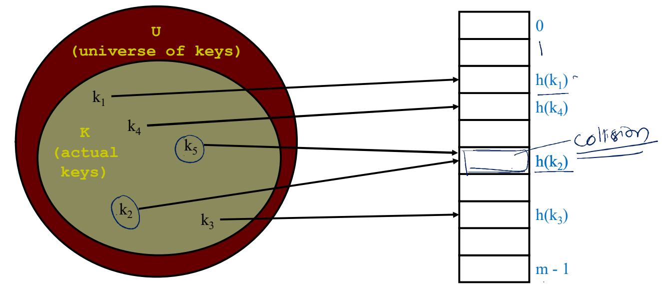
  hash value $\boxed{h(x)} = x \bmod N$ hash table size

  is a hash function for integer keys
- The integer $h(x)$ is called the hash value of x.
- A hash table for a given key type consists of
  - Hash function ***h***
  - Array (called table) of size ***N***
- The goal is to store item (k, o) at index $i = h(k)$

13

13

Hash Functions



14

14

Issues in Hashing

- Collision:
- Overflow.
- Synonym

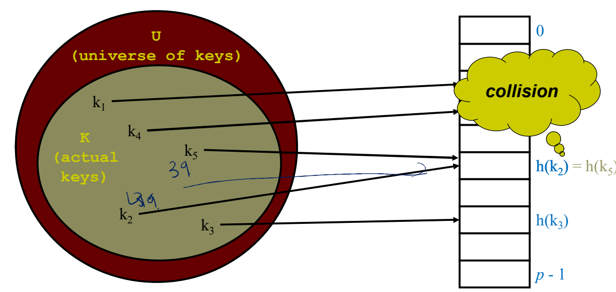
hash func produces same hash value

15

15

Hash Functions

- A **collision** occurs when *h(x)* maps two keys to the same location.



16

16

Properties of Hash function

- Uniformaly distributed to avoid .
  - Collision
  - Overflow.
- Easy to compute.

Hash function type

- Many hashing Functions :
  1. Division methods.(mod by)
  2. Multisquare method(Square the key and take mid of it
  3. Folding method. (Eg key is very large : 21345678 = > 21+ 34+56+87 = 198 omit 1 and the final answer is 98.
  4. Others are Algebraic coding, length dependent method, digit analysis.

$10 \% 10 \to 0$

$10^2 = 100$

$7 = 49$

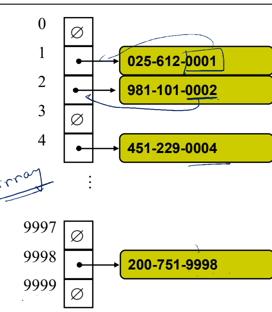
Hash table

- Array of buckets.
- Buckets of size 1 : max -1.

Example

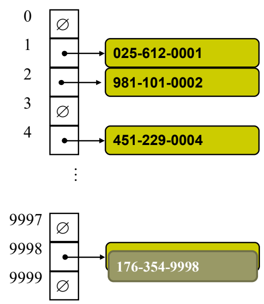
- We design a hash table storing employees records using their social security number, SSN as the key.
  - SSN is a nine-digit positive integer
- Our hash table uses an array of size N = 10,000 and the hash function $h(x)$ = last four digits of x

array



Example

- Our hash table uses an *array* of size **N = 100.**
- We have ***n*** **= 49** employees.
  - Need a method to handle ***collisions***.
- As long as the chance for collision is low, we can achieve this goal.
- Setting **N = 1000** and looking at the last four digits will *reduce* the chance of collision.



Collisions

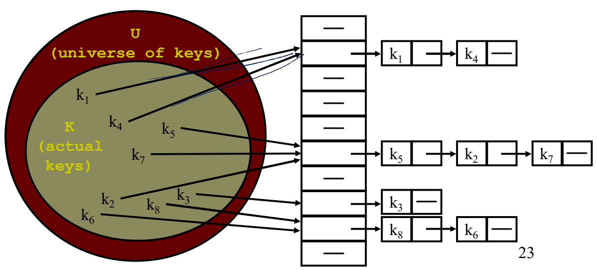
Key = 22

- Two primary techniques for resolving collisions:
  - **Chaining (open hashing)** – keep a collection at each key slot.
  - **Open addressing (closed hashing)** – If the current slot is full use the ***next open*** one.

O(1)
O(log n)
O(n)
sequential
size fixed
Table size fixed

Chaining

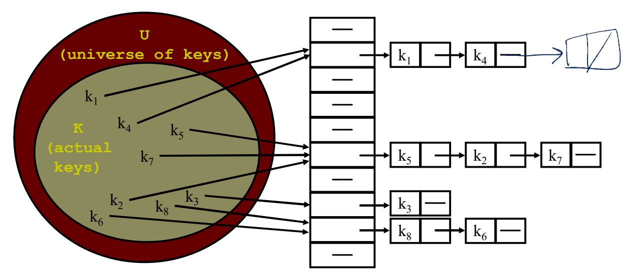
- Chaining puts elements that hash to the same slot in a linked list:



Chaining

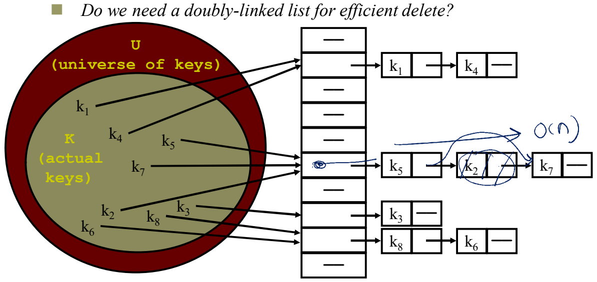
- *How do we insert an element?*

h(x)



Chaining

- How do we delete an element? SLL
  - Do we need a doubly-linked list for efficient delete?



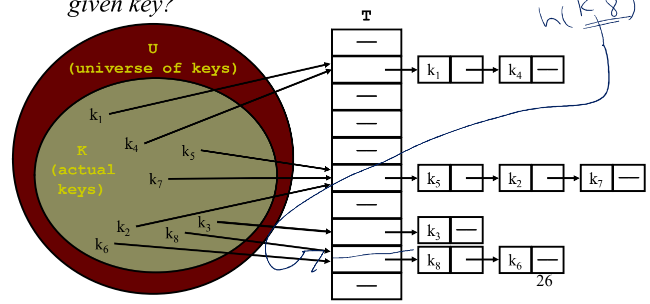
25

25

Chaining

- How do we search for a element with a given key?

h(x1) ⇓ h(k 8)



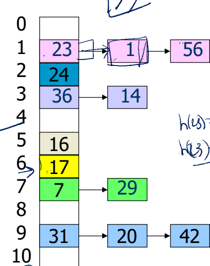
26

26

Chaining /open hashing

$h(x) = x \% 11$

Insert/deletion-once the "bucket" is found through *Hash*, insert and delete are list operations



$x=56$
$h(56) = 56\%11$
$= 1$

$h(23) = 23\%11$
$h(23) = 1$

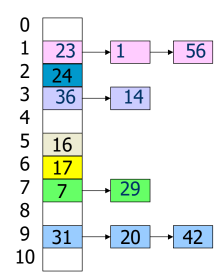
$x = 1$
$h(1) = 1\%11$
$h(1) = 1$

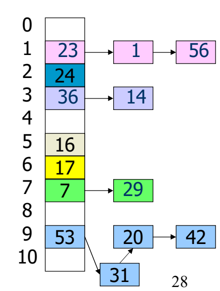
$h(17)$
$= 17\%11$
$= 6$
found !!!
Not found!

27

27

53 = 4 x 11 + 9
53 mod 11 = 9





28

28

Open Addressing

size bucket is fixed
next empty slot

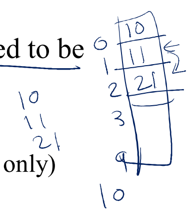
- Basic idea:
  - To insert: If slot is full, try another slot, …, until an open(empty) slot is found (***probing***)
  - To search, follow same sequence of probes as would be used when inserting the element
    - If reach element with correct key, return it
    - If reach a NULL pointer, element is not in table

Open Addressing

- The colliding item is placed in a different cell of the same table.
  - No dynamic memory.
  - Fixed Table size.
  - To pace the item in next available slot probing is used.
- **Load factor:** n/N, where n is the number of items to store and N the size of the hash table.
  - Cleary, $n \leq N$, or $n/N \leq 1$. — all position will
- To get a reasonable performance, $n/N < \boxed{0.5}$.

Collision solving techniques in closed hashing ::

- They key question is what should the next cell to try be?
- Random would be great, but we need to be able to Repeat it.
- Three common techniques:
  - Linear Probing (useful for discussion only)
  - Quadratic Probing
  - Double Hashing



Linear Probing

- **Linear probing** handles collisions by placing the colliding item in the ***next*** (circularly) available table cell.
- Each table cell inspected is referred to as a ***probe.***
- Colliding items lump together, causing future collisions to cause a longer sequence of probes.

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

73%13

0	1	2	3	4	5	6	7	8	9	10	11	12

⇩

		41			18	44	59	32	22	31	73	
0	1	2	3	4	5	6	7	8	9	10	11	12

Search with Linear Probing

- Consider a hash table *A* that uses linear probing
- get(*k*)
  - We start at cell *h*(*k*)
  - We probe consecutive locations until one of the following occurs
    - **An item with key *k* is found, or**
    - ***An empty cell is found*, or**
    - ***N* cells have been unsuccessfully probed**
  - **To ensure the efficiency, If *k* is not in the table, we want to find an empty cell as soon as possible. The load factor can NOT be close to 1.**

Linear Probing

- Search for key=20.
  - h(20)=20 mod 13 =7.
  - Go through rank 8, 9, …, 12, 0.
- Search for key=15
  - h(15)=15 mod 13=2.
  - Go through rank 2, 3 and return null.
- Example:
  - ***h*(*x*) = *x*** mod 13
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, 12, 20 in this order

0	1	2	3	4	5	6	7	8	9	10	11	12

⇩

20		41			18	44	59	32	22	31	73	12
0	1	2	3	4	5	6	7	8	9	10	11	12

Updates with Linear Probing

- To handle insertions and deletions, we introduce a special object, called ***AVAILABLE***, which replaces deleted elements
- delete(*k*)
  - We search for an entry with key ***k***
  - If such an entry **(*k, o*)** is found, we replace it with the special item ***AVAILABLE*** and we return element ***o***
  - ***Have to modify other methods to skip available cells.***
- Insert **(*k, o*)**
  - We throw an exception If the table is full
  - We start at cell ***h*(*k*)**
  - We probe consecutive cells until one of the following occurs
    - A cell ***i*** is found that is either empty or stores ***AVAILABLE***, or
    - *N* cells have been unsuccessfully probed
  - We store entry **(*k, o*)** in cell ***i***

- If collision happens, alternative cells are tried until an empty cell is found.
- Linear probing :
  Try next available position

0	42
1	1
2	24
3	14
4	
5	16
6	28
7	7
8	
9	31
10	9

Linear Probing (insert 12)

12 mod 11 = 1

12 mod 11 = 1+1
= 1+1+1
= 1+1+1+1

0	42	42
1	1	1
2	24	24
3	14	14
4	12	12
5	16	16
6	28	28
7	7	7
8		
9	31	31
10	9	9

+1

Search with linear probing (Search 15)

15 = 1 x 11 + 4
15 mod 11 = 4

15 mo

0	42
1	1
2	24
3	14
4	12
5	16
6	28
7	7
8	
9	31
10	9

15

NOT FOUND !

O(n) ~~O~~ X O(1)

Deletion with linear probing: LAZY (Delete 9)

9 = 0 x 11 + 9
9 mod 11 = 9

0	42	42
1	1	1
2	24	24
3	14	14
4	12	12
5	16	16
6	28	28
7	7	7
8		
9	31	31
10	9	D

FOUND !

Load Factor in Linear Probing

- For *any* $\lambda < 1$, linear probing will find an empty slot
- Search cost (for large table sizes)
  - successful search: $\frac{1}{2}\left(1+\frac{1}{(1-\lambda)}\right)$
  - unsuccessful search: $\frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$
- Linear probing suffers from *primary clustering*
- Performance quickly degrades for $\lambda > 1/2$

O(n)

Linear Probing chaining without replacement

Record No	Natural address	Forward Pointer
1236	1236	
1237	1237	
1238		
1239	1239	
1240	1240	
1241	1241	
1242		
1243	1243	
1244		

coalesce

Record No	Natural address	Forward Pointer
1236	1236	1238
1237	1237	
1238	1236	1242
1239	1239	
1240	1240	
1241	1241	
1242	1238	1244
1243	1243	
1244	1236	1245

1236,1238, 1236

41

Linear Probing chaining with replacement

Record No	Natural address	Forward Pointer
1236	1236	
1237	1237	
1238		
1239	1239	
1240	1240	
1241	1241	
1242		
1243	1243	
1244		

Record No	Natural address	Forward Pointer
1236	1236	
1237	1237	
1238		
1239	1239	
1240	1240	
1241	1241	
1242		
1243	1243	
1244		

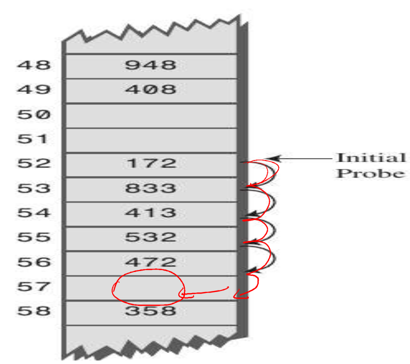
1236,1238, 1236

42

Primary Clustering

- Phenomenon in which two keys that hashes into different values compete with each other in successive rehashes is called Primary clustering .
- One way of eliminating primary clustering is to allow rehash function to depend on the number of times that the function is applied to a particular hash function.
- Ex. Rh(i,j)=(i+j) % tablesize

43



44

Quadratic Probing

- Primary clustering occurs with linear probing because the same linear pattern:

- Instead of searching forward in a linear fashion, consider searching forward using a quadratic function

45

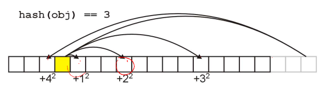
45

Quadratic Probing

- Suppose that an element should appear in bin *h*:
  - If bin *h* is occupied, Then check the following sequence of bins:

    $h + 1^2,\ h + 2^2,\ h + 3^2,\ h + 4^2,\ h + 5^2, \ldots$

    $h + 1,\ h + 4,\ h + 9,\ h + 16,\ h + 25, \ldots$
- For example, with M = 17:



46

46

Quadratic Probing

- For example, suppose an element was to be inserted in bin 23 in a hash table with 31 bins

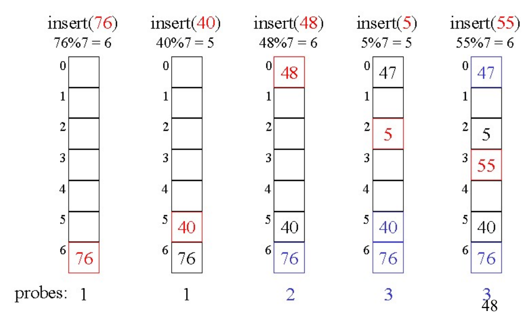
- The sequence in which the bins would be checked is:

  23, 24, 27, 1, 8, 17, 28, 10, 25, 11, 30, 20, 12, 6, 2, 0

47

47

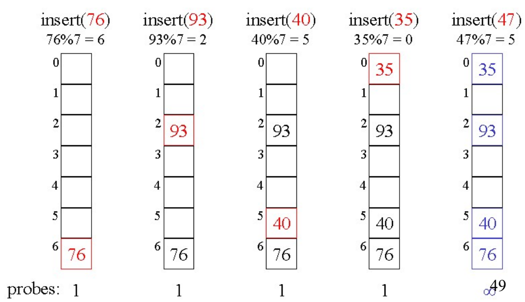
Quadratic Probing Example ☺



48

48

Quadratic Probing Example ☹



Quadratic Probing

- Even If two bins are initially close, the sequence in which subsequent bins are checked varies greatly
- Again, with M = 31 bins, compare the first 16 bins which are checked starting with 22 and 23:

22, 23, 26, 0, 7, 16, 27, 9, 24, 10, 29, 19, 11, 5, 1, 30

23, 24, 27, 1, 8, 17, 28, 10, 25, 11, 30, 20, 12, 6, 2, 0

Quadratic Probing

- Thus, quadratic probing solves the problem of primary clustering
- Unfortunately, there is a second problem which must be dealt with
- Suppose we have M = 8 bins:

  $1^2 \equiv 1$, $2^2 \equiv 4$, $3^2 \equiv 1$
- In this case, we are checking bin $h + 1$ twice having checked only one other bin

Quadratic Probing

- Unfortunately, there is no guarantee that

  $h \pm i^2 \bmod M$

  will cycle through 0, 1, ..., M – 1
- Solution:
  - require that M be prime
  - in this case, $h \pm i^2 \bmod M$ for $i = 0, ..., (M - 1)/2$ will cycle through exactly $(M + 1)/2$ values before repeating

Quadratic Probing

- Example with M = 11:
  0, 1, 4, 9, 16 ≡ 5, 25 ≡ 3, 36 ≡ 3
- With M = 13:
  0, 1, 4, 9, 16 ≡ 3, 25 ≡ 12, 36 ≡ 10, 49 ≡ 10
- With M = 17:
  0, 1, 4, 9, 16, 25 ≡ 8, 36 ≡ 2, 49 ≡ 15, 64 ≡ 13, 81 ≡ 13

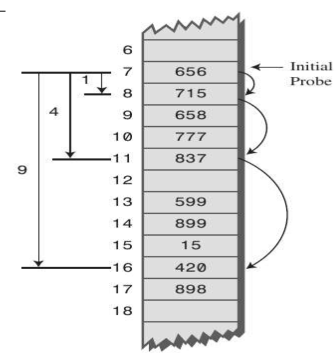
- Thus, quadratic probing avoids primary clustering
- Unfortunately, we are not guaranteed that we will use all the slots.

Load Factor in Quadratic Probing

- For *any* $\lambda \leq \frac{1}{2}$, quadratic probing will find an empty slot; for greater λ, quadratic probing *may* find a slot
- Quadratic probing does not suffer from primary clustering
- Quadratic probing *does* suffer from *secondary* clustering
  - How could we possibly solve this?

Secondary Clustering

- **Secondary Clustering** is the tendency for a collision resolution scheme such as quadratic probing to create long runs of filled slots *away* from the hash position of keys.
- If the primary hash index is x, probes go to x+1, x+4, x+9, x+16, x+25 and so on, this results in Secondary Clustering.
- Secondary clustering is less severe in terms of performance hit than primary clustering, and is an attempt to keep clusters from forming by using Quadratic Probing. The idea is to probe more widely separated cells, instead of those adjacent to the primary hash site.



Secondary Clustering

- The phenomenon of primary clustering will not occur with quadratic probing
- However, If multiple items all hash to the same initial bin, the same sequence of numbers will be followed. This is termed *secondary clustering*
- The effect is less significant than that of primary clustering

Double Hashing (Solution Of secondary clustering)

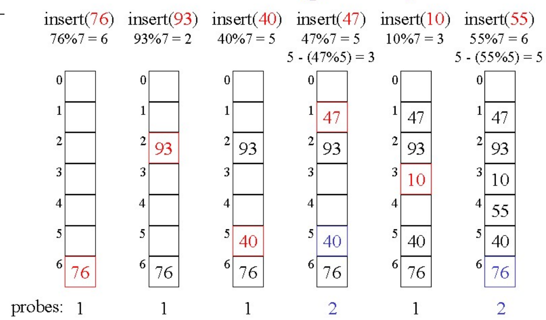
- Use two hash functions
- If **M** is prime, eventually will examine every position in the table
- double_hash_insert(K)

```
{
 if(table is full) error
probe = h1(K)
offset = h2(K)
while (table[probe] occupied)
    probe = (probe + offset) mod M
table[probe] = K
}
```


Double Hashing

- Many of same (dis)advantages as linear probing
- Distributes keys more uniformly than linear probing does
- Notes:
  - h2(x) should never return zero.
  - M should be prime.

Double Hashing Example



Load Factor in Double Hashing

- For *any* $\lambda < 1$, double hashing will find an empty slot (given appropriate table size and $hash_2$)
- Search cost appears to approach optimal (random hash):
  - successful search: $\frac{1}{\lambda} \ln \frac{1}{1-\lambda}$
  - unsuccessful search: $\frac{1}{1-\lambda}$
- No primary clustering and no secondary clustering
- One extra hash calculation

Open Addressing Summary

- In general, the hash function contains two arguments now:
  - Key value
  - Probe number
    h(k,p), p=0,1,...,m-1
- Probe sequences
  <h(k,0), h(k,1), ..., h(k,m-1)>
  - Should be a permutation of <0,1,...,m-1>
  - There are m! possible permutations
  - Good hash functions should be able to produce all m! probe sequences

Open Addressing Summary

- None of the methods discussed can generate more than ***m^2*** different probing sequences.
- Linear Probing:
  - Clearly, only *m* probe sequences.
- Quadratic Probing:
  - The initial key determines a fixed probe sequence, so
  - only m distinct probe sequences.
- Double Hashing
  - Each possible pair ($h_1(k)$,$h_2(k)$) yields a distinct probe, so m^2 permutations.

Algorithm for separate chaining

```
typedef struct node
{
   int data;
   struct node *next;
}Node;

Node  *hasht(100);
```

Contd..

Procedure INITIALIZE(A, N):A is an array of pointers. N is max. size

1. I←0

1. Repeat thru step 3 while I<N
   A[I]=NULL

3. I← I +1

INSERTION IN HASHTABLE(chaining)

PROCEDURE INSERTH(A , N ,KEY)

1.LOC ← KEY % N

2.P ← A[LOC]

3.Repeat while (P!=NULL AND KEY!=DATA(P))
P←NEXT(P)

4. if(P=NULL)
Then Q <= NODE
DATA(Q)← X
NEXT(Q) ← A[LOC]
A[LOC] ← Q

5. RETURN

SEARCH FROM HASHTABLE(chaining)

FUNCTION SEARCH(A, N, KEY):

1. LOC←KEY % N

1. P←A[LOC]

1. Repeat while (P!=NULL AND X!=DATA(P))
   P ← NEXT(P)

4. RETURN P

Algorithm for Linear Probing

Procedure initializeL (FLAG, N):FLAG is an array of Integers. N is max. size

- I←0

- Repeat thru step 3 while I<N
  FLAG[I]=0

- I← I +1

Insertion (Linear probing without replacement without chaining)

PROCEDURE INSERTHL(A , FLAG ,N ,KEY):

A AND FLAG ARE INTEGER ARRAY OF SIZE N. KEY IS A KEY VALUE TO BE INSERTED

```
1.    J ← KEY % N
      I←0

2.    Repeat while I<N
         If (FLAG[J] = 0)
         Then A[J] = KEY
                FLAG[J] =1
                BREAK
         Else
                I←I+1
                J←(J+1)%KEY

3.    RETURN
```


Search (Linear probing without replacement without chaining)

FUNCTION SEARCH(A, FLAG,N, KEY):

```
1.    LOC←KEY % N
      I← 0

2.    Repeat while I<N
         If( A[LOC]=KEY AND FLAG[LOC]=1)
              RETURN (LOC)
         Else
              I←I+1
              LOC←(LOC + 1) % N

3.    RETURN (-1)
```


Insertion (Linear probing with replacement without chaining)

PROCEDURE INSERTHL(A , FLAG ,N ,KEY):

A AND FLAG ARE INTEGER ARRAY OF SIZE N. KEY IS A KEY VALUE TO BE INSERTED

```
1.    LOC← KEY % N
      I←0

2.    If (FLAG[LOC] = 0)
      Then A[LOC] ← KEY
           FLAG[LOC] ←1
           RETURN

3.    I← 0
      J←LOC
      Repeat while I<N AND FLAG[J] =1
           J←(J+1) % N
           I←I+1
```


CONTD..

```
4.    If I=N
      Then WRITE('TABLE IS FULL')
      Else If (A[LOC] % N != LOC)
        Then   A[J] ← A[LOC]
               FLAG[J] ← 1
               A[LOC]←KEY
               FLAG[LOC]← 1
      Else
               A[J]←KEY
               FLAG[J]← 1

5.    RETURN
```

Search (Linear probing with replacement without chaining)

FUNCTION SEARCH(A, FLAG,N, KEY):

1. LOC←KEY % N
   I← 0

2. Repeat while I<N
   If(A[LOC]=KEY AND FLAG[LOC]=1)
   RETURN (LOC)
   Else
   I←I+1
   LOC←(LOC + 1) % N

3. RETURN (-1)

Three situation (with replacement with chaining)

1. Hashed location is empty

1. Hashed location is occupied by an element which is not a synonym of the current element (mapped location contains an element which belongs to a different chain)

1. Hashed location is occupied by an element which is synonym of the current element.

Structure of Hashtable

```
Struct ht
{
    int DATA;
    int FLAG;
    int CHAIN;
};
Struct ht A[10];
```


Algorithm for Linear Probing (with chaining)

Procedure initializeL (A, N):

- I←0
- Repeat thru step 3 While I<N
  FLAG(A[I])=0
  CHAIN(A[I])= -1
- I← I +1

Insertion (Linear probing with replacement with chaining)

PROCEDURE INSERTHL(A , N ,KEY):

1. LOC← KEY % N
   I←0

2. If (FLAG(A[LOC]) = 0) (1st situation)
   Then DATA(A[LOC]) ← KEY
   FLAG (A[LOC]) ←1
   RETURN

3. I← 0
   J←LOC
   Repeat while I<N AND FLAG(A[J]) =1
   J←(J+1) % N
   I←I+1

77

77

CONTD..

4. If I=N
   Then WRITE('TABLE IS FULL')
   RETURN

5. If (DATA(A[LOC]) % N != LOC) (2nd situation)
   Then I←DATA(A[LOC]) % N
   Repeat while(CHAIN(A[I])!=LOC)
   I←CHAIN(A[I])
   CHAIN(A[I])←CHAIN(A[LOC])

   Repeat while (CHAIN(A[I] != -1)
   I←CHAIN(A[I])
   CHAIN(A[I]) ← J

   DATA(A[J]) ←DATA(A[LOC])
   FLAG(A[J]) ← 1
   CHAIN(A[J])← -1
   DATA(A[LOC])←KEY
   FLAG(A[LOC])← 1
   CHAIN(A[LOC]) ← -1
   RETURN

78

78

CONTD..

6. If (DATA(A[LOC]) % N = LOC) (3rd situation)
   Then DATA(A[J]) ←KEY
   FLAG(A[J]) ← 1
   CHAIN(A[J])← -1

   I←LOC
   Repeat while (CHAIN(A[I] != -1)
   I ← CHAIN(A[I])
   CHAIN(A[I]) ← J
   RETURN

7. RETURN

79

79

Search (Linear probing with replacement with chaining)

FUNCTION SEARCH(A , N, KEY):

1. I←0
   LOC←KEY % N

2. Repeat while (I<N AND FLAG(A[LOC])=1 AND DATA(A[LOC]) %N != LOC)
   I←I+1
   LOC←(LOC+1)%N

3. IF(FLAG(A[LOC] != 1 OR I=N)
   Then RETURN -1
   Else
   Repeat while LOC != -1
   If (DATA(A[LOC])=KEY)
   RETURN LOC
   Else
   LOC←CHAIN(A[LOC])

1. RETURN -1

80

80

Choosing A Hash Function

- Clearly choosing the hash function well is crucial.
- ***What are desirable characteristics of the good hash function?***
  - Should distribute keys uniformly into slots
  - Should minimize number of collisions
  - Should be easy to compute
- An essential requirement of the hash function is to ***map equal keys to equal indices***

81

81

Popular Compression Maps

1. Division Method
1. Multiplicative Method
1. Midsquare Method
1. Folding Method

82

82

DIVISION METHOD

- Division
  - Use a mod function

    $h(k) = k \bmod m$

  - Choice of m?
    - Powers of 2 are generally not good!

      $h(k) = k \bmod 2^n$

      selects last n bits of k

      $k \bmod 2^8$ selects these bits

      01100101110 $\boxed{00011010}$

      - All combinations are not generally equally likely
  - Prime numbers close to 2^n seem to be good choices

    eg want ~4000 entry table, choose m = 4093

83

83

The Multiplication Method

- This method is based on obtaining an address of a key based on multiplication value. If K is a non negative key and constant A(0 <A <1) Then compute KA mod 1 which is fractional part of KA. Multiply this fractional part by m and take floor value to get the address (index). i.e
- For a constant A, $0 < A < 1$ and 0<=h(k) <m
- $h(k) = \lfloor \underbrace{m\ (kA \bmod 1)} \rfloor$

  What does this term represent?

84

84

The Multiplication Method

- Multiply the key by constant, A, $0 < A < 1$
- Extract the fractional part of the product
  $(kA - \lfloor kA \rfloor)$
- Multiply this by m
  $h(k) = \lfloor m * (kA - \lfloor kA \rfloor) \rfloor$

- Now m is not critical and a power of 2 can be chosen
- So this procedure is fast on a typical digital computer
  - Set $m = 2^p$
  - Multiply k (w bits) by $\lfloor A \cdot 2^w \rfloor$ □ $2w$ bit product
  - Extract p most significant bits of lower half
  - $A = ½(\sqrt{5} - 1)$ seems to be a good choice (*see* Knuth)

Midsquare method

- Square the value of key and take the number of digits required to form an address from the middle position of squared value.
- Ex. Suppose a key value is 16 Then its square is 256. now If we want address of one digit Then select the address 5.
- Poor performance compared to previous two methods.

Folding Method

- Break up a key into several segments that are added or exclusive-ored together to form a hash value

- Ex. Suppose internal bit representation of key is: 010111001010110 and 5 bits are allowed in index Then divide string into group of 5 bits i.e. 01011 10010 10110 make exclusive or which will produce 01111 which is 15 as a binary integer.

- Two keys ,in which both keys consist of the same group of k bits in different order, hashes into same k bit value i.e generates collision.

Hash Tables - Load factor

- ***simple uniform hashing***: each key in table is equally likely to be hashed to any slot.
- Collisions are very probable!
- Table load factor

  $$\alpha = \frac{n}{m}$$ **n = number of items**, **m = number of slots**

  must be kept low
- Separate chaining
  - linked lists attached to each slot

    gives best performance

    but uses more space!

Hash Tables - General Design

- Choose the table size
  - Large tables reduce the probability of collisions!
  - Table size, *m*
  - *n* items
  - Collision probability $\alpha = n / m$
- Choose a table organisation
  - Does the collection keep growing?
    - Linked lists
  - Size relatively static?
    - Re-hash
- Choose a Good hash function

Analysis of Chaining

- *What will be the **average** cost of an unsuccessful search for a key?* = $O(1+\alpha)$
- *What will be the **average** cost of a successful search?* =
  $O(1 + \alpha/2) = O(1 + \alpha)$
- So the cost of searching = $O(1 + \alpha)$
- *If the number of keys n is proportional to the number of slots in the table, what is α?*

  $\alpha = O(1)$
  - In other words, we can make the expected cost of searching constant If we make α constant

Analysis of Open Addressing

- Consider the load factor, α, and assume each key is uniformly hashed.
- Probability that we hit an occupied cell is Then α.
- Probability that the next probe hits an occupied cell is also α.
- Will terminate If an unoccupied cell is hit: $\alpha(1- \alpha)$.
- From Theorem 11.6, the expected number of probes in an ***unsuccessful*** search is at most $1/(1- \alpha)$.
- Theorem 11.8: Expected number of probes in a successful search is at most:

$$\frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right)$$