Hash functions and hash tables

Hashing, Introduction to hash tables, Hash functions, Collision resolution Techniques, separate Chaining, open addressing, Linear probing, Quadratic probing, Double hashing, rehashing, Chained hash tables.

Goal

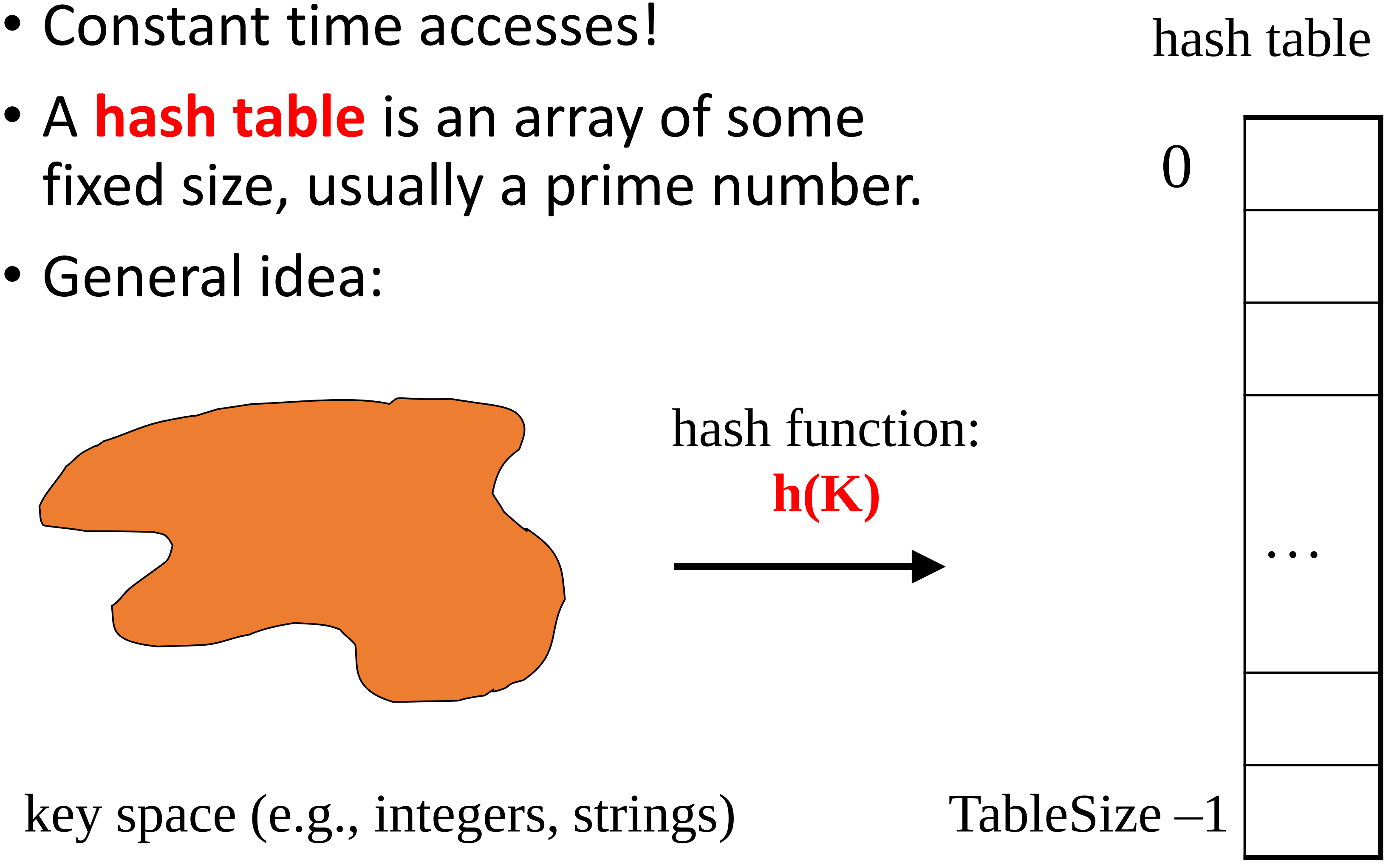
Develop a structure that will allow user to insert/delete/find records in

constant average time (O(1))

- structure will be a table (relatively small)
- table completely contained in memory
- implemented by an array
- capitalizes on ability to access any element of the array in constant time

Hash Tables

- Constant time accesses!
- A **hash table** is an array of some fixed size, usually a prime number.
- General idea:

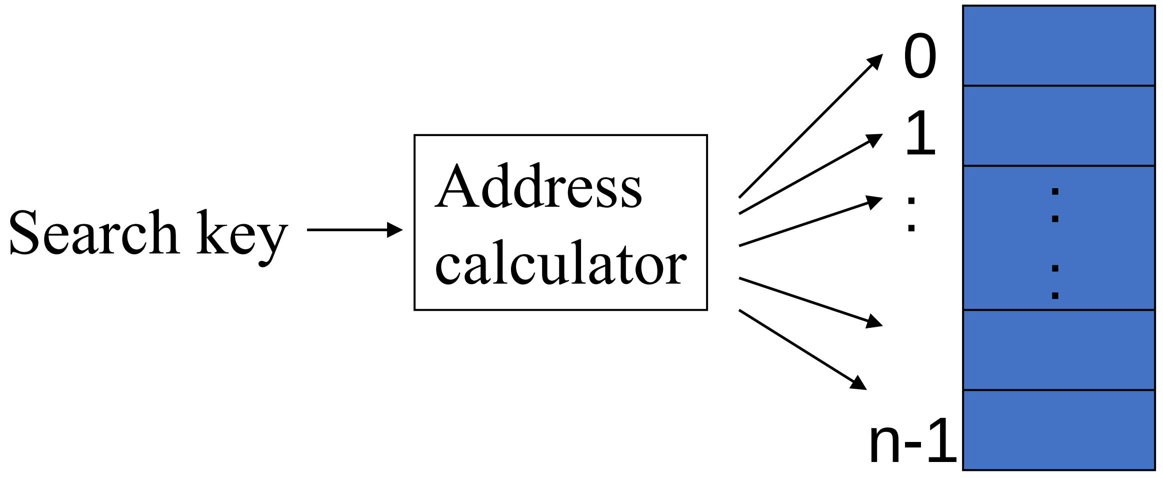


Hashing

- Hashing
  - Enables access to table items in time that is relatively constant and independent of the items
- Hash function
  - Maps the search key of a table item into a location that will contain the item
- Hash table
  - An array that contains the table items, as assigned by a hash function

Hashing Operations

- Address calculator



- tableInsert(newItem)

  i = the array index that the address calculator gives you for newItem's search key

  table[i] = newItem

Hash Function

- Determines position of key in the array.
- Assume table (array) size is N
- Function $f(x)$ maps any key x to an int between 0 and $N-1$
- For example, assume that $N=15$, that key x is a non-negative integer between 0 and MAX_INT, and hash function $f(x) = x \% 15$.

Hash Function

Let f(x) = x % 15. Then,

if x =	25	129	35	2501	47	36
f(x) =	10	9	5	11	2	6

Storing the keys in the array is straightforward:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
_	_	47	_	_	35	36	_	_	129	25	2501	_	_	_

Thus, delete and find can be done in O(1), and also insert, except…

Example

- key space = integers
- TableSize = 10

- **h**(K) = K mod 10

- **Insert**: 7, 18, 41, 94

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Another Example

- key space = integers
- TableSize = 6

- **h**(K) = K mod 6

- **Insert**: 7, 18, 41, 34

0	
1	
2	
3	
4	
5	

Hash Function

What happens when you try to insert: x = 65 ?

N=15

x = 65

f(x) = 5

```
0    1    2    3    4    5    6    7    8    9   10    11   12   13   14
_    _   47    _    _   35   36    _    _  129   25 2501    _    _    _
                        65(?)
```

This is called a **collision**.

Handling Collisions

- What to do when inserting an element and already something present?

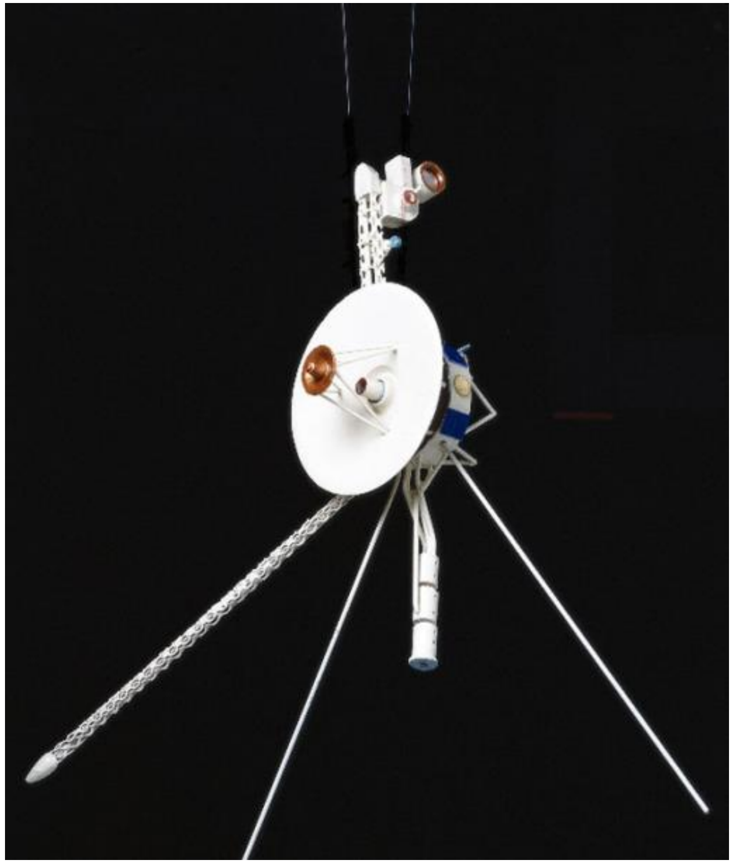
Hashing

- A perfect hash function (**ideally** …)
  - **Maps each search key into a unique location** of the hash table
  - Possible if all the search keys are known
- Collisions
  - Occur when the hash function maps more than one item into the same array location
- Collision-resolution schemes
  - Assign locations in the hash table to items with different search keys when the items are involved in a collision
- Requirements for a hash function
  - Be easy and fast to compute
  - Place items evenly throughout the hash table

Collisions

- When two values hash to the same array location, this is called a collision
- Collisions are normally treated as “first come, first served”—the first value that hashes to the location gets it
- We have to find something to do with the second and subsequent values that hash to this same location

Handling Collisions



- Separate Chaining
- Open Addressing
  - Linear Probing
  - Quadratic Probing
  - Double Hashing

Resolving Collisions

- Two approaches to collision resolution
  - Approach 1: Open addressing
    - A category of collision resolution schemes that probe for an empty, or open, location in the hash table
      - The sequence of locations that are examined is the probe sequence
    - **Linear probing**
      - Searches the hash table sequentially, starting from the original location specified by the hash function
      - Possible problem
        - Primary clustering

Resolving Collisions

- Approach 1: Open addressing (Continued)
  - **Quadratic probing**
    - Searches the hash table beginning with the original location that the hash function specifies and continues at increments of 1^2, 2^2, 3^2, and so on
    - Possible problem
      - Secondary clustering
  - **Double hashing**
    - Uses two hash functions h_1 and h_2, where $h_2(key) \neq 0$ and $h_2 \neq h_1$
    - Searches the hash table starting from the location that one hash function determines and considers every n^{th} location, where n is determined from a second hash function
- Increasing the size of the hash table
  - The hash function must be applied to every item in the old hash table before the item is placed into the new hash table

Handling Collisions

Linear Probing

Algorithm / Procedure

Algorithm:

1. Calculate the hash key. i.e., **key = data % size**
2. Check, if **hashTable[key]** is empty
   - store the value directly by **hashTable[key] = data**
3. If the hash index already has some value then
   1. check for next index using **key = (key+1) % size**
4. Check, if the next index is available hashTable[key] then store the value. Otherwise try for next index.
5. Do the above process till we find the space.

Linear Probing

Let key x be stored in element f(x)=t of the array

```
0    1    2    3    4    5    6    7    8    9   10    11   12   13   14
         47                 35   36             129   25  2501
                            65(?)
```

What do you do in case of a collision?

If the hash table is not full, attempt to store key in the next array element (in this case (t+1)%N, (t+2)%N, (t+3)%N …)

until you find an empty slot.

Linear Probing

Where do you store 65 ?

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
		47			35	36	65		129	25	2501			
					↑	↑	↑							

attempts

Where would you store: 29?

Linear Probing

If the hash table is not full, attempt to store key in array elements (t+1)%N, (t+2)%N, …

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
		47			35	36	65		129	25	2501			29
														↑

attempts

Where would you store: 16?

Linear Probing

If the hash table is not full, attempt to store key in array elements (t+1)%N, (t+2)%N, …

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	16	47			35	36	65		129	25	2501			29
	↑													

Where would you store: 14?

Linear Probing

If the hash table is not full, attempt to store key in array elements (t+1)%N, (t+2)%N, …

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
14	16	47			35	36	65		129	25	2501			29
↑														↑
														attempts

Where would you store: 99?

Linear Probing

If the hash table is not full, attempt to store key in array elements (t+1)%N, (t+2)%N, …

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
14	16	47			35	36	65		129	25	2501	99		29
									↑	↑	↑	↑		

attempts

Where would you store: 127 ?

Linear Probing

- If the hash table is not full, attempt to store key in array elements (t+1)%N, (t+2)%N, …

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	16	47			35	36	65	127	129	25	2501	29	99	14
							↑	↑						

attempts

Example: Let us consider a simple hash function as "key mod 5" and a sequence of keys that are to be inserted are 50, 70, 76, 85, 93.

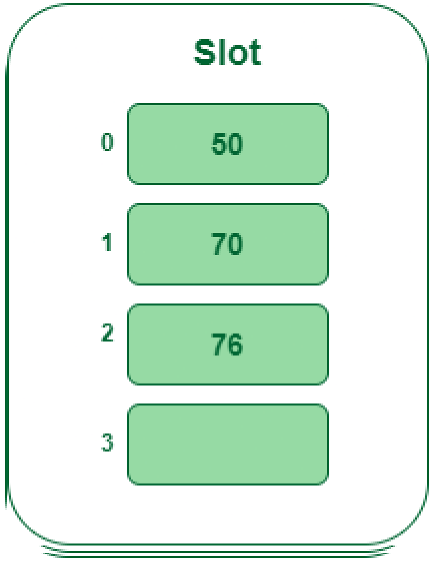
•**Step1:** First draw the empty hash table which will have a possible range of hash values from 0 to 4 according to the hash function provided.

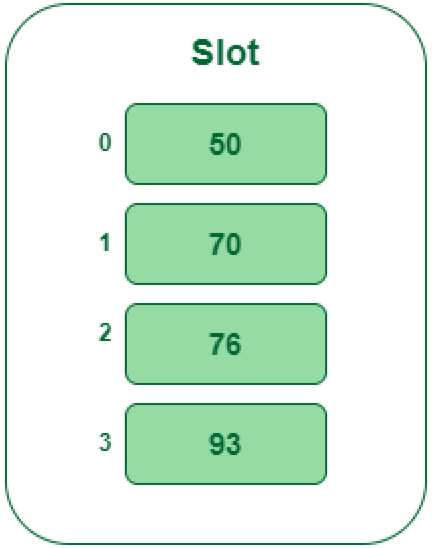
•**Step 2:** Now insert all the keys in the hash table one by one. The first key is 50. It will map to slot number 0 because 50%5=0. So insert it into slot number 0.

•**Step 3:** The next key is 70. It will map to slot number 0 because 70%5=0 but 50 is already at slot number 0 so, search for the next empty slot and insert it.

•**Step 4:** The next key is 76. It will map to slot number 1 because 76%5=1 but 70 is already at slot number 1 so, search for the next empty slot and insert it.

•**Step 5:** The next key is 93 It will map to slot number 3 because 93%5=3, So insert it into slot number 3.

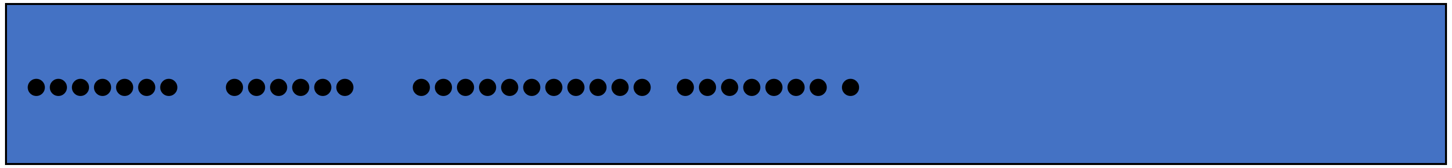




Linear Probing

- Eliminates need for separate data structures (chains), and the cost of constructing nodes.

- Leads to problem of clustering. Elements tend to cluster in dense intervals in the array.



- Search efficiency problem remains.
- Deletion becomes trickier….

Handling Collisions

Quadratic Probing

Algorithms/Procedure

Let hash(x) be the slot index computed using the hash function and n be the size of the hash table.

1. If the slot hash(x) % n is full, then we try $(hash(x) + 1^2) \% n$.
2. If $(hash(x) + 1^2) \% n$ is also full, then we try $(hash(x) + 2^2) \% n$.
3. If $(hash(x) + 2^2) \% n$ is also full, then we try $(hash(x) + 3^2) \% n$.

This process will be repeated for all the values of **i** until an empty slot is found

Quadratic Probing

Let key x be stored in element f(x)=t of the array

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
		47			35	36			129	25	2501			
					65(?)									

What do you do in case of a collision?

If the hash table is not full, attempt to store key in array elements $(t+1^2)\%N$, $(t+2^2)\%N$, $(t+3^2)\%N$ …

until you find an empty slot.

Quadratic Probing

Where do you store 65 ? f(65)=t=5

```
0    1    2    3    4    5    6    7    8    9   10    11   12   13   14
         47                  35   36             129   25  2501            65
                              ↑    ↑               ↑                        ↑
                              t  t+1             t+4                      t+9
                              attempts
```

Where would you store: 29?

Quadratic Probing

If the hash table is not full, attempt to store key in array elements $(t+1^2)\%N$, $(t+2^2)\%N$ …

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
29		47			35	36			129	25	2501			65
↑														↑
t+1														t attempts

Where would you store: 16?

Quadratic Probing

If the hash table is not full, attempt to store key in array elements $(t+1^2)\%N$, $(t+2^2)\%N$ …

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
29	16	47			35	36			129	25	2501			65
	↑													
	t													

attempts

Where would you store: 14?

Quadratic Probing

If the hash table is not full, attempt to store key in array elements $(t+1^2)\%N$, $(t+2^2)\%N$ …

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
29	16	47	14		35	36			129	25	2501			65
↑			↑											↑
t+1			t+4											t attempts

Where would you store: 99?

Quadratic Probing

If the hash table is not full, attempt to store key in array elements $(t+1^2)\%N$, $(t+2^2)\%N$ …

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
29	16	47	14		35	36			129	25	2501		99	65
									↑	↑			↑	
									t	t+1			t+4	

attempts

Where would you store: 127 ?

Quadratic Probing

If the hash table is not full, attempt to store key in array elements $(t+1^2)\%N$, $(t+2^2)\%N$ …

Where would you store: 127 ?

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
29	16	47	14		35	36	127		129	25	2501		99	65

↑
t
attempts

Quadratic Probing

- Tends to distribute keys better than linear probing
- Alleviates problem of clustering
- Runs the risk of an infinite loop on insertion, unless precautions are taken.
- E.g., consider inserting the key 16 into a table of size 16, with positions 0, 1, 4 and 9 already occupied.

- **When two keys hash to the same location, they will probe to the same alternative location. This may cause secondary clustering.**
- **In order to avoid this secondary clustering, double hashing method is created where we use extra multiplications and divisions.**

Handling Collisions

Double Hashing

Double Hashing

- Use a hash function for the decrement value
  - $Hash(key, i) = H_1(key) - (H_2(key) * i)$
- Now the decrement is a function of the key
  - The slots visited by the hash function will vary even if the initial slot was the same
  - Avoids clustering
- Theoretically interesting, but in practice slower than quadratic probing, because of the need to evaluate a second hash function.

Algorithms/ Procedure

- You must perform the following steps to find an empty slot:
  1. Verify if hash1(key) is empty. If yes, then store the value on this slot.
  2. If hash1(key) is not empty, then find another slot using hash2(key).
  3. Verify if hash1(key) + hash2(key) is empty. If yes, then store the value on this slot.
  4. Keep incrementing the counter and repeat with hash1(key)+2hash2(key), hash1(key)+3hash2(key), and so on, until it finds an empty slot.

Double Hashing

Let key x be stored in element f(x)=t of the array

Array:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
		47			35	36			129	25	2501			
					65(?)									

What do you do in case of a collision?

Define a second hash function $f_2(x)=d$. Attempt to store key in array elements (t+d)%N, (t+2d)%N, (t+3d)%N …

until you find an empty slot.

Double Hashing

- Typical second hash function

$$f_2(x) = R - (x \% R)$$

where R is a prime number, $R < N$

Double Hashing

Where do you store 65 ? $f(65)=t=5$

Let $f_2(x) = 11 - (x \% 11)$ $f_2(65)=d=1$

Note: $R=11$, $N=15$

Attempt to store key in array elements $(t+d)\%N$, $(t+2d)\%N$, $(t+3d)\%N$ …

Array:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
		47			35	36	65		129	25	2501			
					↑	↑	↑							
					t	t+1	t+2							

attempts

Double Hashing

If the hash table is not full, attempt to store key in array elements (t+d)%N, (t+d)%N …

Let $f_2(x) = 11 - (x \% 11)$ $f_2(29)=d=4$

Where would you store: 29?

Array:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
		47			35	36	65		129	25	2501			29
														↑
														t
														attempt

Double Hashing

If the hash table is not full, attempt to store key in array elements (t+d)%N, (t+d)%N …

Let $f_2(x) = 11 - (x \% 11)$ $f_2(16) = d = 6$

Where would you store: 16?

Array:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	16	47			35	36	65		129	25	2501			29
	↑													
	t													
	attempt													

Where would you store: 14?

Double Hashing

If the hash table is not full, attempt to store key in array elements (t+d)%N, (t+d)%N …

Let $f_2(x) = 11 - (x \% 11)$ $f_2(14) = d = 8$

Array:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
14	16	47			35	36	65		129	25	2501			29
↑							↑							↑
t+16							t+8							t

attempts

Where would you store: 99?

Double Hashing

If the hash table is not full, attempt to store key in array elements (t+d)%N, (t+d)%N …

Let $f_2(x) = 11 - (x \% 11)$ $f_2(99)=d=11$

Array:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
14	16	47			35	36	65		129	25	2501	99		29
	↑				↑				↑			↑		
	t+22				t+11				t			t+33		

attempts

Where would you store: 127 ?

Double Hashing

If the hash table is not full, attempt to store key in array elements (t+d)%N, (t+d)%N …

Let $f_2(x) = 11 - (x \% 11)$ $f_2(127) = d = 5$

Array:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
14	16	47			35	36	65		129	25	2501	99		29
		↑					↑					↑		
		t+10					t					t+5		

attempts

Infinite loop!

Double Hashing

- The advantage of Double hashing is that it is one of the best forms of probing, producing a uniform distribution of records throughout a hash table.
- This technique does not yield any clusters.
- It is one of the effective methods for resolving collisions.
- The disadvantages are: **Double hashing is more difficult to implement than any other. Double hashing can cause thrashing**.

Handling Collisions

Separate Chaining

Algorithms/Procedure

- 1. Declare an array of a linked list with the hash table size.
- 2. Initialize an array of a linked list to NULL.
- 3. Find hash key.
- 4. If chain[key] == NULL

  Make chain[key] points to the key node.
- 5. Otherwise(collision),

  Insert the key node at the end of the chain[key].

Separate Chaining

Let each array element be the head of a chain.

```
0   1   2   3   4   5   6   7   8   9   10   11   12   13   14
        ↓           ↓   ↓           ↓   ↓    ↓
       47          65  36         129   25 2501
                    ↓
                   35
```

Where would you store: 29, 16, 14, 99, 127 ?

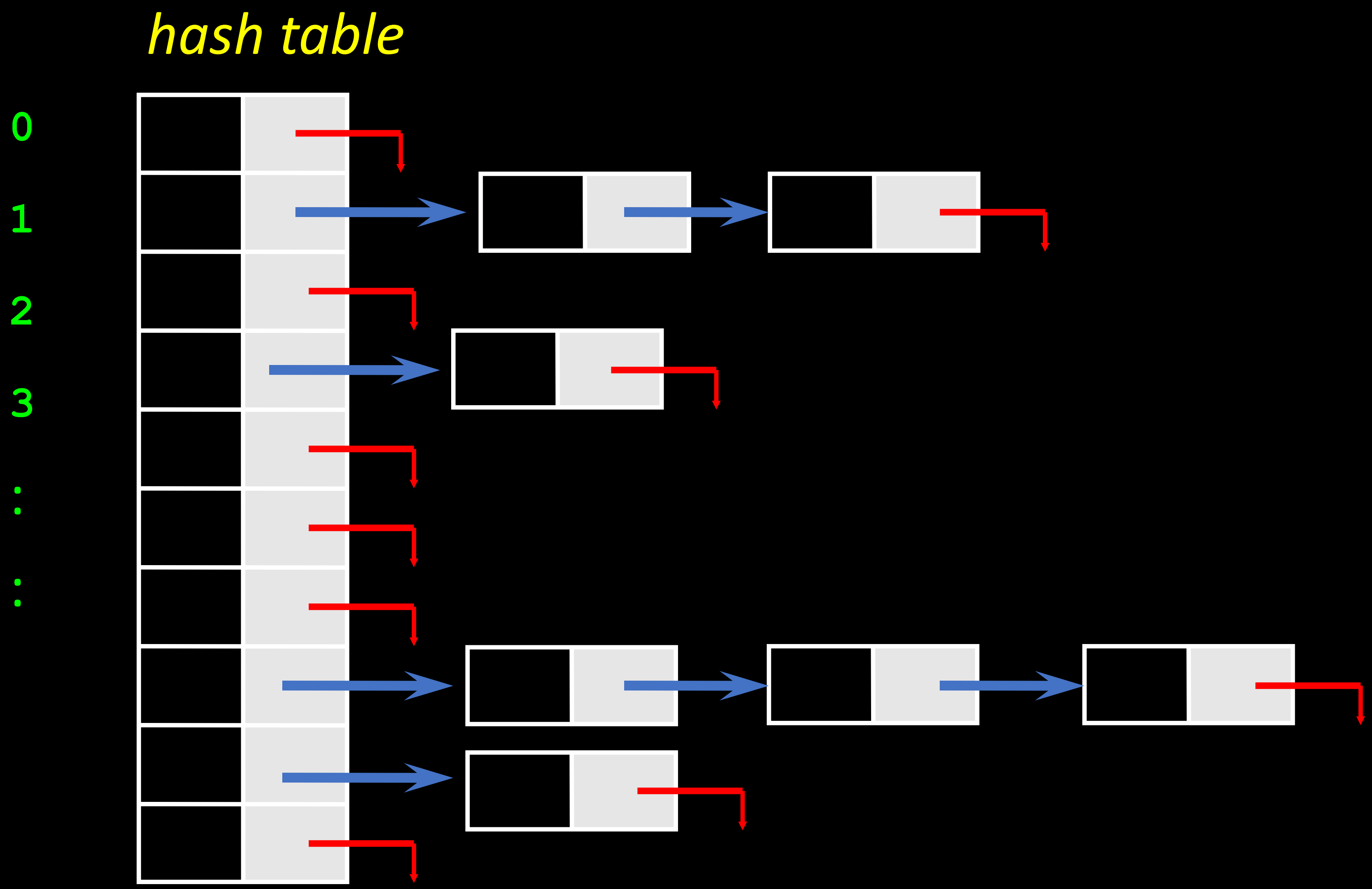
Separate Chaining

Let each array element be the head of a chain:

Where would you store: 29, 16, 14, 99, 127 ?

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	↓	↓			↓	↓	↓		↓	↓	↓			↓
	16	47			65	36	127		99	25	2501			14
					↓				↓					↓
					35				129					29

New keys go at the front of the relevant chain.


hash table
0
1
2
3
:
:

Separate Chaining: Disadvantages

- Parts of the array might never be used.
- As chains get longer, search time increases to O(n) in the worst case.
- Constructing new chain nodes is relatively expensive (still constant time, but the constant is high).
- Is there a way to use the “unused” space in the array instead of using chains to make more space?

Factors affecting efficiency

- Choice of hash function
- Collision resolution strategy
- Load Factor

- Hashing offers excellent performance for insertion and retrieval of data.

The Efficiency of Hashing

- An analysis of the average-case efficiency of hashing involves the load factor
  - Load factor α
    - Ratio of the current number of items in the table to the maximum size of the array `table`
    - Measures how full a hash table is
    - Should not exceed 2/3
  - Hashing efficiency for a particular search also depends on whether the search is successful
    - Unsuccessful searches generally require more time than successful searches

Performance of Hashing

- m = Length of Hash Table
- n = Total keys to be inserted in the hash table

- **Load Factor and Rehashing :**
  - Load factor is defined as (m/n) where n is the total size of the hash table and m is the preferred number of entries which can be inserted before a increment in size of the underlying data structure is required.
- Load factor lf = n/m
- Expected time to search = O(1 +lf)
- Expected time to insert/delete = O(1 + lf)

- The time complexity of search insert and delete is
- O(1) if lf is O(1)

Hashing in Data Structures

- Insert - T[h(key)] = value;
- Delete - T[h(key)] = NULL;
- Search - return T[h(key)];

- Open Hashing (Separate Chaining)
- h(key) = key%table size

- Closed Hashing (Open Addressing):
- Linear Probing:
- rehash(key) = (n+1)%tablesize

- Quadratic Probing:
- rehash(key) = (n+ k²) % tablesize

- Double Hashing:
- h2(key) != 0 and h2 != h1

Python Implementation of Hashing

```python
# Function to display hashtable
def display_hash(hashTable):

    for i in range(len(hashTable)):
        print(i, end = " ")

        for j in hashTable[i]:
            print("-->", end = " ")
            print(j, end = " ")

        print()

# Creating Hashtable as
# a nested list.
HashTable = [[] for _ in range(10)]

# Hashing Function to return
# key for every value.
def Hashing(keyvalue):
    return keyvalue % len(HashTable)
```

```python
# Insert Function to add
# values to the hash table
def insert(Hashtable, keyvalue, value):

    hash_key = Hashing(keyvalue)
    Hashtable[hash_key].append(value)

# Driver Code
insert(HashTable, 10, 'Allahabad')
insert(HashTable, 25, 'Mumbai')
insert(HashTable, 20, 'Mathura')
insert(HashTable, 9, 'Delhi')
insert(HashTable, 21, 'Punjab')
insert(HashTable, 21, 'Noida')

display_hash (HashTable)
```

Any questions ?