Algorithms

Chapter 7: Hashing

GATE CS Lectures
by Monalisa

Section 5: Algorithms

- Searching, sorting, hashing. Asymptotic worst case time and space complexity. Algorithm design techniques : greedy, dynamic programming and divide-and-conquer . Graph traversals, minimum spanning trees, shortest paths
- Chapter 1:Algorithim Analysis:-Algorithm intro , Order of growth ,Asymptotic notation, Time complexity, space complexity, Analysis of Recursive & non recursive program, Master theorem]
- Chapter 2:Brute Force:-Sequential search, Selection Sort and Bubble Sort , Radix sort, Depth first Search and Breadth First Search.
- Chapter 3: Decrease and Conquer :- Insertion Sort, Topological sort, Binary Search .
- Chapter 4: Divide and conquer:-Min max problem , matrix multiplication ,Merge sort ,Quick Sort , Binary Tree Traversals and Related Properties .
- Chapter 5: Transform and conquer:- Heaps and Heap sort, Balanced Search Trees.
- Chapter 6: Greedy Method:-knapsack problem , Job Assignment problem, Optimal merge, Hoffman Coding, minimum spanning trees, Dijkstra's Algorithm.
- Chapter 7: Dynamic Programming:-The Bellman-Ford algorithm ,Warshall's and Floyd's Algorithm ,Rod cutting, Matrix-chain multiplication ,Longest common subsequence ,Optimal binary search trees
- Chapter 8: Hashing.
- Reference : Introduction to Algorithms by Thomas H. Cormen
- Introduction to the Design and Analysis of Algorithms, by Anany Levitin
- My Note

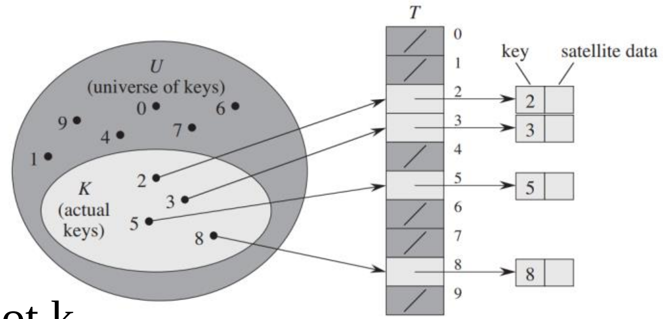
Hashing

- Many applications require the dictionary operations INSERT, SEARCH, and DELETE.
- A hash table is an effective data structure for implementing dictionaries.
- Although searching for an element in a hash table can take as long as searching for an element in a linked list Θ(n) time in the worst case.
- The average time to search for an element in a hash table is O(1).

❖ Direct-address tables

Direct addressing works well when the universe U of keys is reasonably small.

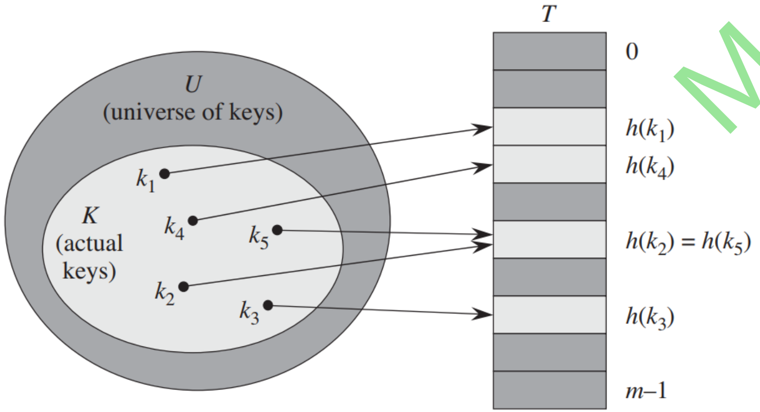
- To represent the dynamic set, we use an array, or ***direct-address table***, denoted by T[0...m-1] in which each position, or ***slot***, corresponds to a key in the universe U .
- DIRECT-ADDRESS-SEARCH(*T, k*) **return** T[k]
- DIRECT-ADDRESS-INSERT(*T, x*) T[x.*key*]= x
- DIRECT-ADDRESS-DELETE(*T,x*) T[x.*key*] =NIL
- Each of these operations takes only O(1) time.

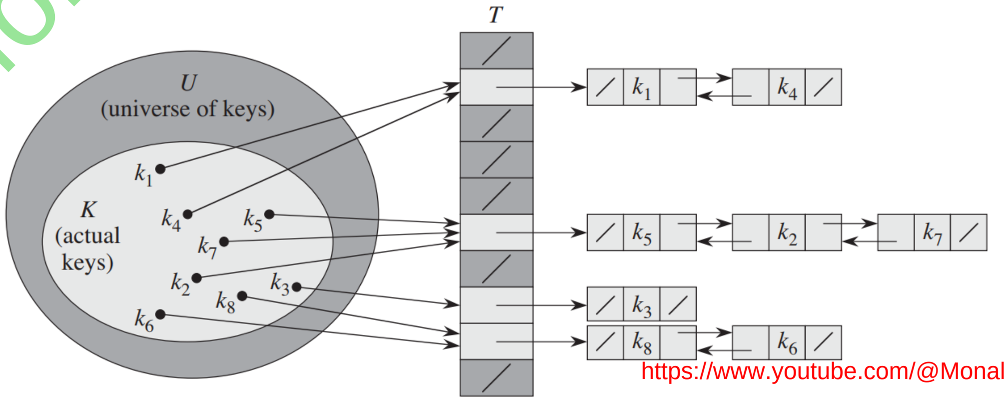


❖ Hash tables

- With direct addressing, an element with key k is stored in slot k.
- With hashing , this element is stored in slot h(k);
- We use a ***hash function*** h to compute the slot from the key k.

- Here, h maps the universe U of keys into the slots of a ***hash table*** T[0….m-1]:
- h : U→{0,1,…m-1},
- Where the size m of the hash table is typically much less than |U|.
- We say that an element with key k ***hashes*** to slot h(k); or h(k) is the ***hash value*** of key k.
- Multiple keys may hash to the same slot. We call this situation a ***collision***.
- **Collision resolution by chaining**
- In ***chaining***, we place all the elements that hash to the same slot into the same linked list .
- The linked list can be either singly or doubly linked; we show it as doubly linked because deletion is faster that way.
- CHAINED-HASH-INSERT(T,x)   insert x at the head of list T [h(x.*key*)]
- CHAINED-HASH-SEARCH(T,k)   search for an element with key k in list T[h(k)]
- CHAINED-HASH-DELETE(T,x)   delete x from the list T [h(x.*key*)]





- The worst-case running time for insertion is O(1). We can delete an element in O(1) time if the lists are doubly linked.
- For searching, the worst case running time is proportional to the length of the list.
- **Analysis of hashing with chaining**
- Given a hash table T with m slots that stores n elements, we define the ***load factor*** α for T as n/m, that is, the average number of elements stored in a chain.
- α can be less than, equal to, or greater than 1.
- The worst-case behavior of hashing with chaining is terrible: all n keys hash to the same slot, creating a list of length n.
- The worst-case time for searching is thus Θ (n) plus the time to compute the hash function.
- The average-case performance of hashing depends on how well the hash function h distributes the set of keys to be stored among the m slots, on the average.
- If any given element is equally likely to hash into any of the m slots.
- We call this the assumption of ***simple uniform hashing***.
- For j=0,1,..m-1 ,length of the list T[j] by n_j,so that $n=n_0+n_1+\ldots n_{m-1}$, and the expected value of n_j is $E[n_j]= \alpha =n/m$.

Hash functions

- **Interpreting keys as natural numbers**
- Most hash functions assume that the universe of keys is the set N={0,1,…n}of natural numbers. Thus, if the keys are not natural numbers, we find a way to interpret them as natural numbers.
- **The division method**
- In the ***division method*** for creating hash functions, we map a key **k** into one of **m** slots by taking the remainder of k divided by m. That is, the hash function is h(k)= k mod m.
- For example, if the hash table has size m =12 and the key is k = 100, then h(k)= 4.
- When using the division method, we usually avoid certain values of m. For example, m should not be a power of 2, since if m = 2^p, then h(k) is just the p lowest-order bits of k.
- **The multiplication method**
- The ***multiplication method*** for creating hash functions operates in two steps. First,we multiply the key k by a constant A in the range 0 < A < 1 and extract the fractional part of kA.
- Then, we multiply this value by m and take the floor of the result. In short, the hash function is $h(k)= \lfloor m(kA \bmod 1)\rfloor$.
- Where "*k*A mod 1" means the fractional part of kA, that is, $kA - \lfloor kA \rfloor$.
- We typically choose it to be a power of 2 ($m=2^p$ for some integer p), since we can then easily implement the function on most computers.
- Folding ,Mid square ,Truncation are some other methods.

- Collision resolution mechanism: ***open hashing*** (also called ***separate chaining***) and ***closed hashing*** (also called ***open addressing***).
- **Open addressing/Closed hashing**
- In ***open addressing***, all elements occupy the hash table itself.
- That is, each table entry contains either an element of the dynamic set or NIL.
- No lists and no elements are stored outside the table, unlike in chaining.
- The load factor α can never exceed 1.
- We could store the linked lists for chaining inside the hash table, in the unused hash-table slots, but the advantage of open addressing is that it avoids pointers altogether.
- To perform insertion using open addressing, we successively examine, or ***probe***, the hash table until we find an empty slot in which to put the key.
- With open addressing, for every key k, the ***probe sequence*** {h(k,0),h(k,1)…. h(k,m-1)} .
- HASH-INSERT(T,k)

```
i = 0
Repeat
    j=h(k,i)
     if T[j] == NIL
        T[j]= k
         return j
    else i =i+ 1
until i == m
error "hash table overflow"
```

- The search can terminate (unsuccessfully) when it finds an empty slot, since k would have been inserted there and not later in its probe sequence.
- This argument assumes that keys are not deleted from the hash table.
- HASH-SEARCH(*T*,*k*)

```
i = 0
Repeat
        j=h(k,i)
        if T[j] == k
            return j
        i= i + 1
until T [j] == NIL or i == m
return NIL
```

- In Worst case searching running time O(m).
- Deletion from an open-address hash table is difficult. We can solve this problem by marking the slot, storing in it the special value DELETED instead of NIL.
- We will examine three commonly used techniques to compute the probe sequences required for open addressing: linear probing, quadratic probing, and double hashing.
- **Linear probing**
- The method of ***linear probing*** uses the hash function $h(k, i)=(h'(k)+ i) \bmod m$.
- For i= 0,1,…m-1. Given key k, we first probe T[$h'(k)$], We next probe slot T[$h'(k)$+1], and so on up to slot T[m-1].

- Linear probing suffers from a problem known as ***primary clustering***.
- Clusters arise because an empty slot preceded by i full slots gets filled next with probability (i+1)/m.
- Long runs of occupied slots tend to get longer,and the average search time increases.
- **ISRO2016-29**
- A Hash Function f defined as f(key)=key mod 7. With linear probing while inserting the keys 37,38,72,48,98,11,56 into a table indexed from 0, in which location key 11 will be stored?
- A.3 B.4 C.5 D.6

- f(37)=37 mod 7=2
- f(38)=38 mod 7=3
- f(72)=72 mod 7=2, $h(k, i)=(h'(k)+i) \bmod m$
- f(72)=(2+1) mod 7=3 ⇒ (2+2) mod 7=4
- f(48)=48 mod 7=6
- f(98)=98 mod 7=0
- f(11)=11 mod 7=4 ⇒ (4+1) mod 7=5
- f(56)=98 mod 7=0 ⇒ (0+1) mod 7=1
- Ans : C.5

Index	Key
0	98
1	56
2	37
3	38
4	72
5	11
6	48

- **Quadratic probing**
- ***Quadratic probing*** uses a hash function of the form $h(k, i)= (h'(k)+c_1 i+c_2 i^2) \bmod m$,
- Where $h'(k)$ is an auxiliary hash function, c_1 and c_2 are positive constants , and i=0,1,… m-1.
- The initial position probed is T[h'(k)] later positions probed are offset by amounts that depend in a quadratic manner on the probe number i.
- To make full use of the hash table, the values of c_1, c_2, and m are constrained.
- If two keys have the same initial probe position, then their probe sequences are the same, since $h(k_1,0)=h(k_2,0)$ implies $h(k_1,i)=h(k_2,i)$.
- This property leads to a milder form of clustering, called ***secondary clustering***.
- **Double hashing**
- ***Double hashing*** uses a hash function of the form $h(k,i)=(h_1(k)+ih_2(k)) \bmod m$,
- Where both h_1 and h_2 are auxiliary hash functions.
- The value $h_2(k)$ must be relatively prime to the hash-table size m.
- A convenient way to ensure this condition is to let m be a power of 2 and to design h_2 so that it always produces an odd number.
- Another way is to let m be prime and to design h_2 so that it always returns a positive integer less than m.

- For example, we could choose m prime and let $h_1(k)=k \bmod m$, $h_2(k)=1+(k \bmod m')$
- where m' is chosen to be slightly less than m .
- When m is prime or a power of 2, double hashing improves over linear or quadratic probing.
- **Analysis of open-address hashing**
- We express our analysis of open addressing in terms of the load factor α = n/m of the hash table.
- With open addressing, at most one element occupies each slot, and thus n≤ m, which implies $\alpha \leq 1$.
- We assume that we are using uniform hashing. In this idealized scheme, the probe sequence {h(k,0),h(k,1)…. h(k,m-1)} used to insert or search for each key k.
- ***Theorem***
  Given an open-address hash table with load factor α = n/m < 1, the expected number of probes in an unsuccessful search is at most 1/(1- α), assuming uniform hashing.
- ***Corollary***
  Inserting an element into an open-address hash table with load factor α requires at most 1/(1- α) probes on average, assuming uniform hashing.
- ***Theorem***
  Given an open-address hash table with load factor α < 1, the expected number of probes in a successful search is at most $\frac{1}{\alpha} Ln \frac{1}{1-\alpha}$ assuming uniform hashing and assuming that each key in the table is equally likely to be searched for.

Index	Key
0	22
1	88
2	
3	
4	4
5	15
6	28
7	17
8	59
9	31
10	10

- **Exercises *11.4-1***
- Consider inserting the keys 10, 22, 31, 4, 15, 28, 17, 88, 59 into a hash table of length m =11 using open addressing with the auxiliary hash function $h'(k) = k$.Illustrate the result of inserting these keys using linear probing, using quadratic probing with c_1=1 and c_2=3, and using double hashing with $h_1(k)$=k and $h_2(k)=1+(k \bmod (m-1))$.
- **linear probing:** *h(k, i)=(h′(k)+ i) mod m* ,So *h(k, i)=(k+ i) mod 11.*
- $h(10)=10 \bmod 11=10$ $h(22)=22 \bmod 11=0$
- $h(31)=31 \bmod 11=9$ $h(4)=4 \bmod 11=4$
- $h(15)=15 \bmod 11=4$, $:(15+1) \bmod 11 = 5$
- $h(28)=28 \bmod 11=6$ $h(17)=17 \bmod 11=6$ $:(17+1) \bmod 11 = 7$
- $h(88)=88 \bmod 11=0$ $:(88+1) \bmod 11 = 1$
- $h(59)=59 \bmod 11=4$ $:(59+1) \bmod 11 = 5$
- $(59+2) \bmod 11 = 6$, $(59+3) \bmod 11 = 7$
- $(59+4) \bmod 11 = 8$

Index	Key
0	22
1	
2	
3	17
4	4
5	
6	28
7	59
8	15
9	31
10	10

- **Exercises *11.4-1***
- Consider inserting the keys 10, 22, 31, 4, 15, 28, 17, 88, 59 into a hash table of length m=11 using open addressing with the auxiliary hash function $h'(k) = k$. Illustrate the result of inserting these keys using linear probing, using quadratic probing with c_1=1 and c_2=3, and using double hashing with $h_1(k)$=k and $h_2(k)$=1+ (k mod (m-1)).
- **Quadradic probing:**
- $h(k, i)= (h'(k)+c_1 i+c_2 i^2) \bmod m$,So $h(k, i)=(k+ i+3i^2) \bmod 11$.
- $h(10)$=10 mod 11=10, $h(22)$=22 mod 11=0, $h(31)$=31 mod 11=9
- $h(4)$=4 mod 11=4, $h(15)$=15 mod 11=4 :(15+1+3) *mod* 11 =8
- $h(28)$=28 mod 11=6 $h(17)$=17 mod 11=6 :(17+1+3) *mod* 11 =10
- (17+2+3*4) *mod* 11 =9, (17+3+3*9) *mod* 11 =3
- $h(88)$=88 mod 11=0 :(88+1+3) *mod* 11 =4, (88+2+3*4) *mod* 11 =3
- (88+3+3*9) *mod* 11=8, (88+4+3*16) *mod* 11=8, (88+5+3*25) *mod* 11 =3
- (88+6+3*36) *mod* 11 =4, (88+7+3*49) *mod* 11 =0.
- $h(59)$=59 mod 11=4 :(59+1+3) *mod* 11 =7
- No slot available for 88.

Index	Key
0	22
1	
2	59
3	17
4	4
5	15
6	28
7	88
8	
9	31
10	10

- **Exercises *11.4-1***
- Consider inserting the keys 10, 22, 31, 4, 15, 28, 17, 88, 59 into a hash table of length m=11 using open addressing with the auxiliary hash function $h'(k) = k$. Illustrate the result of inserting these keys using linear probing, using quadratic probing with c_1=1 and c_2=3, and using double hashing with $h_1(k)$=k and $h_2(k)$=1+ (k mod (m-1)).
- **Double hashing:**
- $h(k,i)=(h_1(k)+ih_2(k))$ mod m ,So *h(k, i)=(k+ i(1+k mod 10)) mod 11.*
- *h*(10)=10 mod 11=10, *h*(22)=22 mod 11=0, *h*(31)=31 mod 11=9
- *h*(4)=4 mod 11=4, *h*(15)=15 mod 11=4
- :(15+1+5) *mod* 11 =10 , (15+2*6) *mod* 11 =5
- *h*(28)=28 mod 11=6 *h*(17)=17 mod 11=6 :(17+1+7) *mod* 11 =3
- *h*(88)=88 mod 11=0 :(88+1+8) *mod* 11 =9, (88+2*9) *mod* 11 =7
- *h*(59)=59 mod 11=4 :(59+1+9) *mod* 11 =3, (59+2*10) *mod* 11 =2

- **linear probing**

Index	Key
0	22
1	88
2	
3	
4	4
5	15
6	28
7	17
8	59
9	31
10	10

Quadradic probing

Index	Key
0	22
1	
2	
3	17
4	4
5	
6	28
7	59
8	15
9	31
10	10

Double hashing

Index	Key
0	22
1	
2	59
3	17
4	4
5	15
6	28
7	88
8	
9	31
10	10

- **Exercises** ***11.4-3***
- Consider an open-address hash table with uniform hashing. Give upper bounds on the expected number of probes in an unsuccessful search and on the expected number of probes in a successful search when the load factor is 3/4 and when it is 7/8.
- α=3/4
- Unsuccessful search : $\frac{1}{1-{}^{3}/_{4}}$= 4 probes
- Successful search: $\frac{1}{{}^{3}/_{4}} Ln \frac{1}{1-{}^{3}/_{4}} \approx$ 1.848 probes
- α=7/8
- Unsuccessful search : $\frac{1}{1-{}^{7}/_{8}}$= 8 probes
- Successful search: $\frac{1}{{}^{7}/_{8}} Ln \frac{1}{1-{}^{7}/_{8}} \approx$ 2.377 probes