Data Structures and Algorithms (CS-2001)

KALINGA INSTITUTE OF INDUSTRIAL TECHNOLOGY

School of Computer Engineering

4 Credit

Lecture Note

Chapter Contents

Sr #	Major and Detailed Coverage Area	Hrs
8	**Searching**	4
	Linear Search, Binary Search, Hashing	

Searching

Computer systems are often used to store large amounts of data from which individual records must be retrieved according to some search criterion. Thus the efficient storage of data to facilitate **fast searching is an important issue**.

Following are the typical searching methodology used:

- **Linear Search**
- **Binary Search**
- **Hashing**

Linear Search

Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every items is checked and if a match founds then that particular item is returned otherwise search continues till the end of the data collection. The run time complexity is **O(n)**

How linear search works?

It sequentially checks each element of the list for the target value until a match is found or until all the elements have been searched.

Algorithm

```
LinearSearch (Array A, Value x)

Step 1: Start
Step 2: Set i to 1
Step 3: Set n to length of A
Step 4: if i > n then go to step 9
Step 5: if A[i] = x then go to step 8
Step 6: Set i to i + 1 [continuation of algorithm]
Step 7: Go to Step 4
Step 8: Print Element x found at position i and go to step 10
Step 9: Print element not found
Step 10: Stop
```

Linear Search C code

```
#include <stdio.h>

int main()
{
  int array[100], search, c, n;

  printf("Enter the number of elements in array\n");
  scanf("%d",&n);

  printf("Enter %d integer(s)\n", n);

  for (c = 0; c < n; c++)
    scanf("%d", &array[c]);

  printf("Enter the number to search\n");
  scanf("%d", &search);
```

```
//continuation of program
for (c = 0; c < n; c++)
  {
    if (array[c] == search)
    {
      printf("%d is present at location %d.\n", search, c+1);
      break;
    }
  }

  if (c == n)
   printf("%d is not present in array.\n", search);

  return 0;
}
```

Linear Search Recursive C code

```
#include <stdio.h>

/* Recursive function to search x in arr[l..r] */
int recSearch(int arr[], int l, int r, int x)
{
    if (r < l)
      return -1;
    if (arr[l] == x)
      return l;
    return recSearch(arr, l+1, r, x);
}
int main()
{
    int arr[] = {12, 34, 54, 2, 3}, i;
   int n = sizeof(arr)/sizeof(arr[0]);
```

```
//continuation of program
int x = 3; // x is the element to be searched for
int index = recSearch(arr, 0, n-1, x);
if (index != -1)
     printf("Element %d is present at index %d", x, index);
else
      printf("Element %d is not present", x);
return 0;
}
```

Linear Search cont...

Time Complexity

Case	Best Case	Worst Case	Average Case
Item is present	1	n	n/2
Item not present	n	n	n

Class Work

Your CR (Class Representative) went for a walk in a garden. There are many trees in the garden and each tree has an English alphabet on it. While CR was walking, he/she noticed that all trees with vowels on it are not in good state. She/he decided to take care of them. So, he/she asked you to tell him the count of such trees in the garden.

Note : The following letters are vowels: 'A', 'E', 'I', 'O', 'U' ,'a','e','i','o' and 'u'.

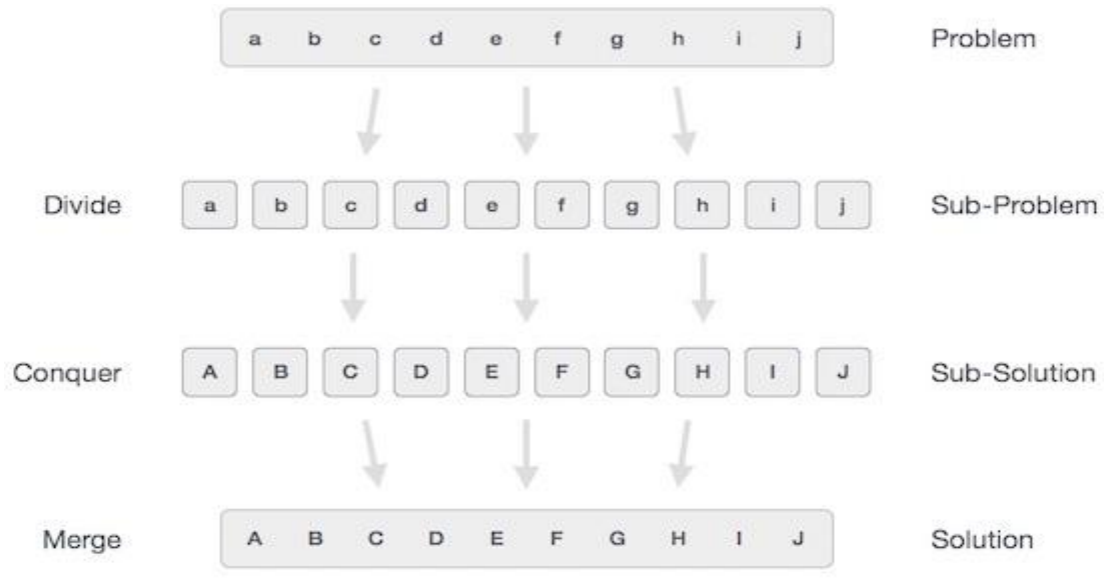
Input : "nBBZLaosnm"
Output : 2

Input : "JHkIsnZtTL"
Output : 1

Explanation: number of vowels in 1st input is 2 and in second input is 1

Divide & Conquer

In divide and conquer approach, the problem in hand, is divided into smaller sub-problems and then each problem is solved independently. When we keep on dividing the sub-problems into even smaller sub-problems, we may eventually reach at a stage where no more dividation is possible. Those "atomic" smallest possible sub-problem (fractions) are solved. The solution of all sub-problems is finally merged in order to obtain the solution of original problem.



Divide & Conquer cont…

Broadly, we can understand divide-and-conquer approach as three step process.

- **Divide/Break:** This step involves breaking the problem into smaller sub-problems. Sub-problems should represent as a part of original problem. This step generally takes recursive approach to divide the problem until no sub-problem is further dividable. At this stage, sub-problems become atomic in nature but still represents some part of actual problem.
- **Conquer/Solve:** This step receives lot of smaller sub-problem to be solved. Generally at this level, problems are considered 'solved' on their own.
- **Merge/Combine:** When the smaller sub-problems are solved, this stage recursively combines them until they formulate solution of the original problem.

The following computer algorithms are based on divide-and-conquer programming approach

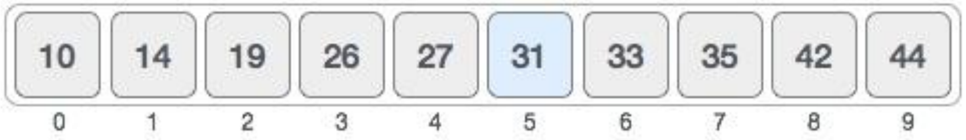
- Binary Search

This algorithmic approach works recursively and conquer & merge steps works so close that they appear as one.

Binary Search

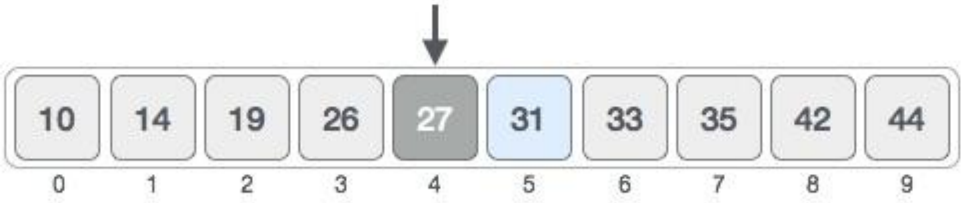
Binary search is a fast search algorithm with run-time complexity of **O(log n)**. This search algorithm works on the principle of **divide and conquer**. For this algorithm to work properly the data collection should be in sorted form. It search a particular item by comparing the middle most item of the collection. If match occurs then index of item is returned. If middle item is greater than item then item is searched in sub-array to the right of the middle item other wise item is search in sub-array to the left of the middle item. This process continues on sub-array as well until the size of sub-array reduces to zero.

How binary search works?



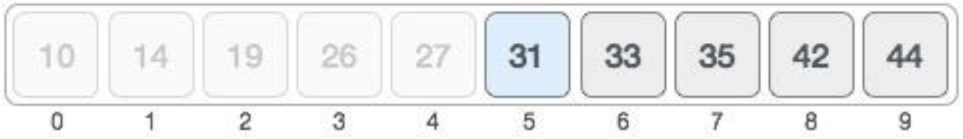
Before the sort computation starts, **bottom** is initialized to 0 and **top** is initialized to n-1 i.e. **9.**

First, we shall determine the half of the array by using this formula : **mid = (top + bottom)/ 2.** Here it is, (9 + 0) / 2 = 4 (integer value of 4.5). So 4 is the mid of array.

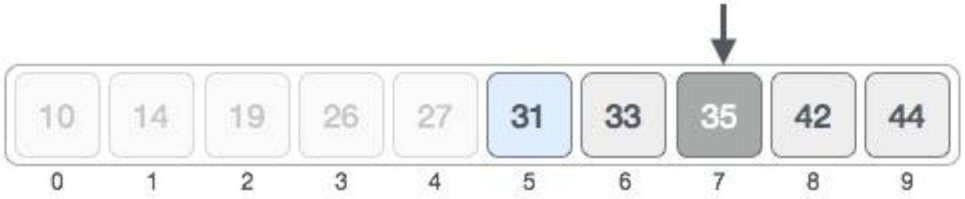


Binary Search cont…

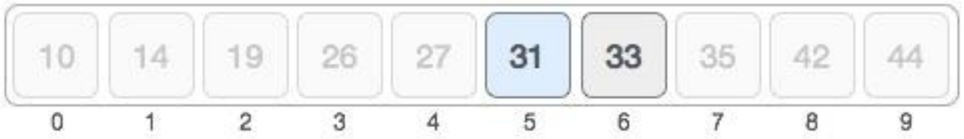
Now we compare the value stored at location 4, with the value being searched i.e. 31. We find that value at location 4 is 27, which **is not a match**. Because value is greater than 27 and we have a sorted array so we also know that target value must be in **upper portion** of the array. So make **bottom = mid + 1** i.e. 4 + 1 = 5



So at this point, bottom is 5 and top is 9. Second, we need to find the **new mid value** again i.e. mid = (bottom + top) /2 = (5 + 9) / 2 = 14 / 2 = 7. So 7 is the mid of the array

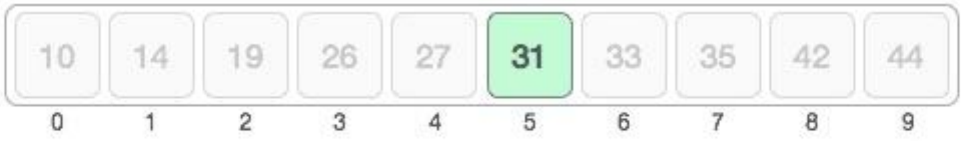


Now we compare the value stored at location 7, with the value being searched i.e. 31. We find that value at location 7 is 35, which **is not a match.** Because value is less than 35 and we have a sorted array so we also know that target value must be in **lower portion** of the array. So make **top = mid - 1** i.e. 7 - 1 = 6



Binary Search cont…

So at this point, bottom is 5 and top is 6. Third, we need to find the **new mid value again** i.e. mid = (bottom + top) /2 = (5 + 6) / 2 = 11 / 2 = 5. The value stored at location 5 **is a match** and conclude that the target value 31 is stored at location 5.



Binary search pseudo code

```
INPUT: A[], n, ITEM
bottom ← 0
top ← n – 1
REPEAT
   mid ← (bottom + top ) / 2
   IF (ITEM < A[mid]) THEN
     top ← mid – 1
   ELSE IF (ITEM > A[mid]) THEN
     bottom ← mid + 1
   END IF
WHILE (ITEM != A[mid] AND bottom <= top)
```

```
[continuation of Pseudo code]
IF (ITEM  = A[mid] THEN
   OUTPUT: ITEM FOUND
ELSE
    OUTPUT: ITEM NOT FOUND
```

Binary Search C code

```
#include <stdio.h>

int main()
{
 int n, a[30], item, i, j, mid, top, bottom;
 printf("Enter # of elements :\n");
 scanf("%d", &n);
 printf("Enter elements in ascending order\n");
 for (i = 0; i < n; i++)
 {
  scanf("%d", &a[i]);
 }
 printf("\nEnter the item to  search\n");
 scanf("%d", &item);
 bottom = 0;
top = n - 1;
```

```
//continuation of program
do
{
  mid = (bottom + top) / 2;
  if (item < a[mid])
   top = mid - 1;
  else if (item > a[mid])
   bottom = mid + 1;
 } while (item != a[mid] && bottom <= top);
 if (item == a[mid])
  printf("Binary search successful!!\n");
else
  printf("\n  Search failed);
 return 0;
}
```

Binary Search Recursive C code

```
// A recursive binary search function. It returns location of x in
// given array arr[l..r] is present, otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
  if (r >= l)
  {
     int mid = l + (r - l)/2;
     // If the element is present at the middle itself
     if (arr[mid] == x)  return mid;
     // If element is smaller than mid, then it can only be present
     // in left subarray
     if (arr[mid] > x) return binarySearch(arr, l, mid-1, x);
     // Else the element can only be present in right subarray
     return binarySearch(arr, mid+1, r, x);
  }
  // We reach here when element is not present in array
  return -1;
}
```

```
//continuation of program
int main(void)
{
  int arr[] = {2, 3, 4, 10, 40};
  int n = sizeof(arr)/ sizeof(arr[0]);
  int x = 10;
  int result = binarySearch(arr, 0, n-1, x);
  if (result == -1)
     printf("Element is not present in array")
  else
     printf("Element is present at index %d", result);
  return 0;
}
```

Binary Search cont…

Time Complexity

Case	Best Case	Worst Case	Average Case
Item is present	1	$\log_2(n)$	$\log_2(n/2)$
Item not present	$\log_2(n)$	$\log_2(n)$	$\log_2(n)$

Class Work

Its been a few days since John is acting weird and finally you(best friend) came to know that its because his proposal has been rejected.

He is trying hard to solve this problem but because of the rejection thing he can't really focus. Can you help him? The question is: Given a number n , find if n can be represented as the sum of 2 desperate numbers (not necessarily different) , where desperate numbers are those which can be written in the form of (a*(a+1))/2 where a > 0 .

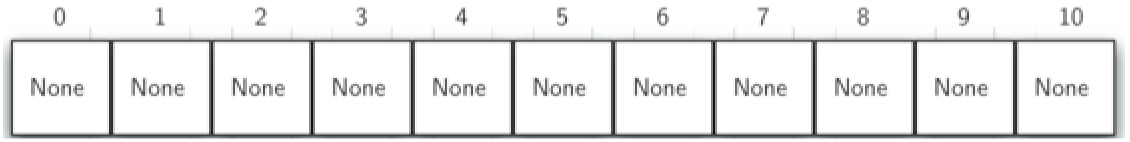
Input : The first input line contains an integer n (1 ≤ n ≤ 10^9).

Output : Print "YES", if n can be represented as a sum of two desperate numbers, otherwise print "NO".

Hashing

Hashing is a concept that is used to search an item in **O(1)** time. It is a completely different approach from the comparison-based methods (binary search, linear search). Rather than navigating through a list data structure comparing the search key with the elements, hashing tries to reference an element in a table directly based on its search key k.

A **hash table** is a collection of items which are stored in such a way as to make it easy to find them later. Each position of the hash table, often called a **slot**, can hold an item and is named by an integer value starting at 0. For example, we will have a slot named 0, a slot named 1, a slot named 2, and so on. Initially, the hash table contains no items so every slot is empty. We can implement a hash table by using a list with each element initialized to **None**. Below figure shows a hash table of size m=11. In other words, there are m slots in the table, named 0 through 10.



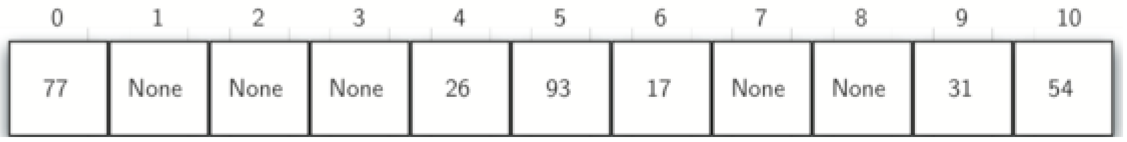
0	1	2	3	4	5	6	7	8	9	10
None	None	None	None	None	None	None	None	None	None	None

Hash Function

The mapping between an item and the slot where that item belongs in the hash table is called the **hash function**. The hash function will take any item in the collection and return an integer in the range of slot names, between 0 and m-1. Assume that we have the set of integer items 54, 26, 93, 17, 77, and 31. The hash function, sometimes referred to as the "remainder method" simply takes an item and divides it by the table size, returning the remainder as its hash value h(item) = item % 11. Table shown below gives all of the hash values for example items.

Item	Hash Index
54	54 % 11 = 10
26	26 % 11 = 4
93	93 % 11 = 5
17	17 % 11 = 6
77	77 % 11 = 0
31	31 % 11 = 9

Once the hash values have been computed, we can insert each item into the hash table at the designated position as shown below. Note that 6 of the 11 slots are now occupied. This is referred to as the **load factor**, and is commonly denoted by λ=number of item/table size. For this example, λ=6/11.

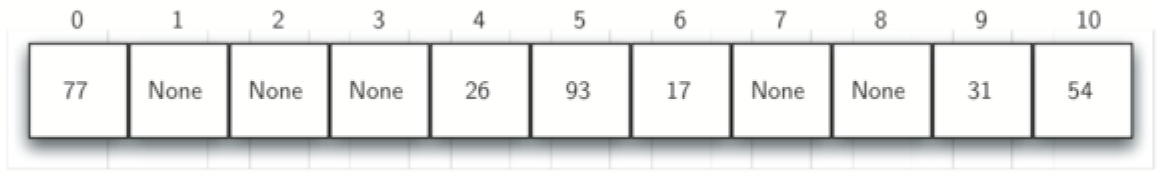


0	1	2	3	4	5	6	7	8	9	10
77	None	None	None	26	93	17	None	None	31	54

Now when we want to search for an item, we simply use the hash function to compute the slot name for the item and then check the hash table to see if it is present. This searching operation is **O(1)**, since a constant amount of time is required to compute the hash value and then index the hash table at that location. If everything is where it should be, we have found a **constant time search** algorithm.

Collision

Let’s insert items 65 and 37 to the following hash table.



0	1	2	3	4	5	6	7	8	9	10
77	None	None	None	26	93	17	None	None	31	54

Item	Hash	Hash Index	Comments
65	65 % 11	10	Slot is occupied
37	37 % 11	4	Slot is occupied

***Note** - Hash Index can be called as Hash Address or Address*

So the hash function is not yielding to distinct values. So (54, 65) & (26,37) yielding to same hash value. This situation is called as **collision** (also called **clash**) and some method to be used to resolve it. So how to handle the collision?

- Search from there for an empty location
- Use a second/third/fourth/fifth hash function
- Use the array location as the header of a linked list of values that hash to this location

Hash Function

Given a collection of items, a hash function that maps each item into a unique slot is referred to as a **perfect hash function**. If we know the items and the collection will never change, then it is possible to construct a perfect hash function. Unfortunately, given an arbitrary collection of items, there is no systematic way to construct a perfect hash function.

One way to always have a perfect hash function is to increase the size of the hash table so that each possible value in the item range can be accommodated. This guarantees that each item will have a unique slot. Although this is practical for small numbers of items, it is not feasible when the number of possible items is large. For example, if the items were nine-digit Social Security numbers, this method would require almost one billion slots. If we only want to store data for a class of 25 citizen, we will be wasting an enormous amount of memory.

So goal is to create a hash function that minimizes the number of collisions, easy to compute, and evenly distributes the items in the hash table. There are a number of common ways to extend the simple remainder method.

Popular Hash Functions

- Folding method
- Midsquare method
- Division method
- Subtraction method
- Digit extraction method
- Rotation hashing method

Folding Method

The folding method for constructing hash functions begins by dividing the item into equal-size pieces (the last piece may not be of equal size). These pieces are then added together to give the resulting hash value.

Example -

if our item was the phone number 436-555-4601, we would take the digits and divide them into groups of 2 (43,65,55,46,01). After the addition, 43+65+55+46+01, we get 210. If we assume our hash table has 11 slots, then we need to perform the extra step of dividing by 11 and keeping the remainder. In this case 210 % 11 is 1, so the phone number 436-555-4601 hashes to slot 1. Sometimes, for **extra milling**, even number parts are each **reversed** before the addition. So the groups of 2 (43,65,55,46,01) becomes (43, 56, 55, 64 and 01). After the addition, 43+56+55+64+01, we get 210.

Midsquare Method

We first square the item, and then extract some portion of the resulting digits. For example, if the item were 44, we would first compute 44^2=1936. By extracting the middle two digits, 93, and performing the remainder step, we get 5 (93 % 11). Below table shows item under both the reminder method and the mid-square method.

Item	Address	Mid-Square Explanation
54	3	54^2=2916, 91%11 = 3
26	7	26^2=676, 7%11 = 7
93	9	93^2=8649, 64%11 = 9
17	8	17^2=289, 8%11 = 8
77	4	77^2=5929, 92%11 = 4
31	6	31^2=961, 6%11 = 6

Division Method

Definition of Hash Function: **H(x) = x mod m + 1**

Where m is some predetermined divisor integer (i.e. the table size), x is the preconditioned item, and mod stands for modulo. **Note that adding 1 is only necessary if the table starts at key 1 (if it starts at 0, the algorithm simplifies to H(x) = x mod m.** So, in other words: given an item, divide the preconditioned key of that item by the table size (+1). The remainder is the hash key.

Example

Given a hash table with 10 slots, what is the hash key for 'Cat'? Since 'Cat' = 6798116 when converted to ASCII, then x = 6798116

We are given the table size (i.e., m = 10, starting at 0 and ending at 9).

H(x) = x mod m

H(6798116) = 6798116 mod 10

= 6

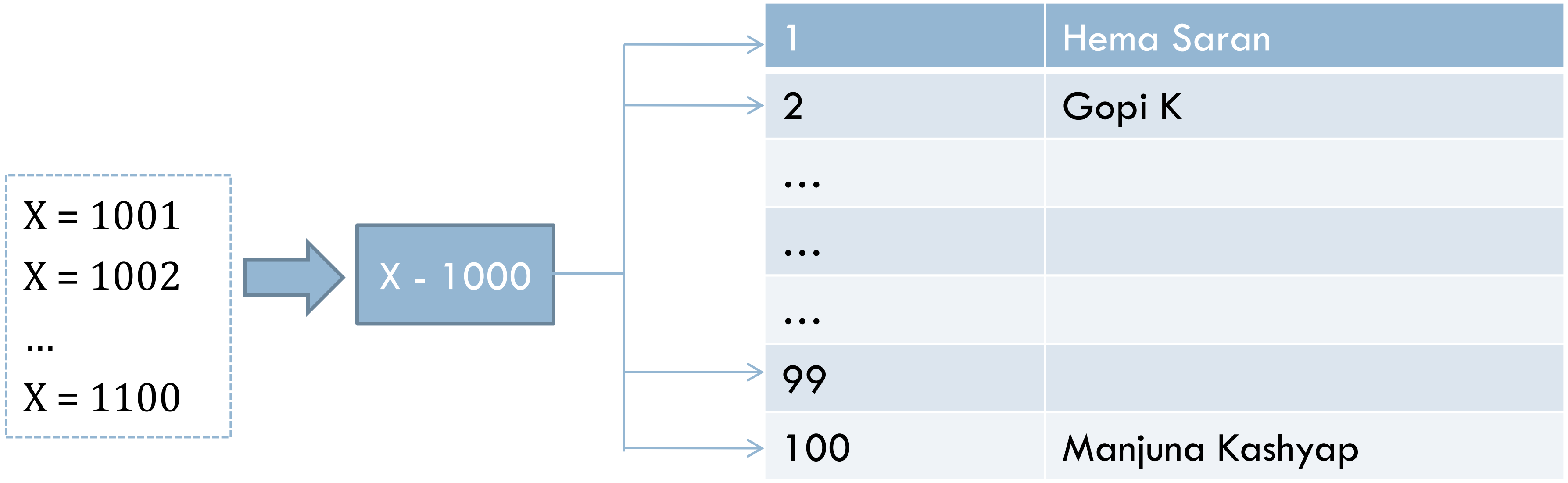
'Cat' is inserted into the table at address 6.

Subtraction Method

The items are not consecutive and don’t start from 1. In such cases, we subtract a number from the item/key to determine the address.

Example

A company have 100 employees and employee number starts from 1000



1	Hema Saran
2	Gopi K
…	
…	
…	
99	
100	Manjuna Kashyap

Digit Extraction Method

Keys are extracted from the key and made use as its address i.e. select specific digits from the key k and use it as an address.

Example 1

Suppose we want to hash a 6 digit employee number say 123456 to a three digit address, we could select the first, third and fourth digits from left and use them as address, so the address will be 124

Example 2

Suppose the roll number of a student is 160252 and to hash the number to a 3 digit address selecting first, third and fourth digits from right, so the address will be 220

Example 3

Suppose the roll number of a student is 160252 and to hash the number to a 3 digit address selecting first, third and fourth digits from left, so the address will be 102

Rotation Hashing Method

This method is useful when keys are assigned serially, as in the case of serial numbers. This method is generally not used by itself, but is used in combination with other hashing methods.

Example

Original Key	Rotation	Rotated Key
600101	600101	160010
600102	600102	260010
600103	600103	360010
600104	600104	460010
600105	600105	560010

Collision Resolution Techniques

There are 2 broad ways of collision resolution:

- Closed Hashing – Array based implementation. Also called as **Open Addressing**
- Open Hashing – Array of linked list implementation. Also called as **Separate Chaining**

The difference has to do with whether collisions are stored **outside of the hash table** (open hashing) or whether collisions result in storing one of the records at **another slot in the hash table** (closed hashing)

Open Addressing includes:

- Linear Probing (Linear Search)
- Quadratic Probing (Nonlinear Search)
- Double Hashing (Uses two hash function)

Linear Probing

Search the next empty location by looking into the next cell until we found an empty cell.

Example

Assume that Hash Table size is 20 and the hash function = x mod 20 and key to insert are 1, 2, 42, 4, 12, 14, 17, 13 and 37

Key	Hash	Address	Address after Linear Probing	# of probes
1	1%20	1	1	1
2	2%20	2	2	1
42	42%20	2	3 (i.e. 2 + 1)	2
4	4%20	4	4	1
12	12%20	12	12	1
14	14%20	14	14	1
17	17%20	17	17	1
13	13%20	13	13	1
37	37%20	17	18 (i.e. 17 + 1)	2

Probes are also know as comparisons

Linear Probing Analysis

- Approximate average number of comparisons (probes) that a search requires:

$$\frac{1}{2}\left[1+\frac{1}{1-\lambda}\right] \textit{ for a successful search}$$

$$\frac{1}{2}\left[1+\frac{1}{(1-\lambda)^2}\right] \textit{ for an unsuccessful search}$$

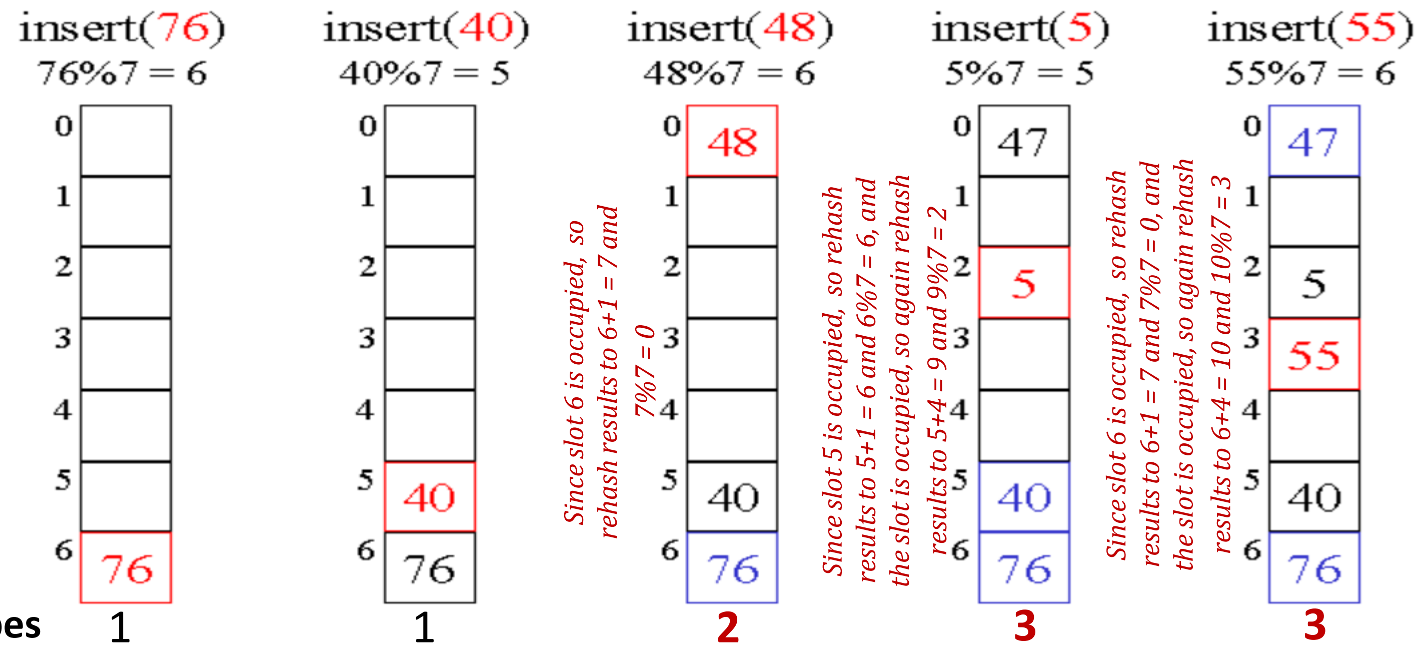
λ=load factor

- As the load factor increases, number of collision increases causing increased search time
- To maintain efficiency, it is important to prevent the hash table from filling up

Quadratic Probing

One main disadvantages of linear probing is that records tend to cluster (i.e. appear next to each other) when the load factor is more than 50 %. Such a clustering substantially increases the average search time of the record. So 2 techniques (Quadratic Probing and Double Hashing) to minimize the clustering.

Quadratic Probing: Instead of using a constant "skip" value, we use a rehash function that increments the hash value by 1, 3, 5, 7, 9, and so on. This means that if the first hash value is h, the successive values are h+1, h+4, h+9, h+16, and so on. In other words, quadratic probing uses a skip consisting of **successive perfect squares** i.e. k^2 where k >=1



insert(76)
76%7 = 6
insert(40)
40%7 = 5
insert(48)
48%7 = 6
insert(5)
5%7 = 5
insert(55)
55%7 = 6
0
1
2
3
4
5
6
76
40
76
48
40
76
47
5
40
76
47
5
55
40
76
Since slot 6 is occupied, so rehash results to 6+1 = 7 and 7%7 = 0
Since slot 5 is occupied, so rehash results to 5+1 = 6 and 6%7 = 6, and the slot is occupied, so again rehash results to 5+4 = 9 and 9%7 = 2
Since slot 6 is occupied, so rehash results to 6+1 = 7 and 7%7 = 0, and the slot is occupied, so again rehash results to 6+4 = 10 and 10%7 = 3
# of Probes
1
1
2
3
3

Double Hashing

It works on a similar idea to linear and quadratic probing. Use a big table and hash into it. Whenever a collision occurs, choose another slot in table to put the value. The difference here is that instead of choosing next opening, a second hash function is used to determine the location of the next slot. For example, given hash function H1 and H2 and key. do the following:

- Check location hash1(key). If it is empty, put record in it.
- If it is not empty calculate hash2(key).
- check if hash1(key)+hash2(key) is open, if it is, put it in
- repeat with hash1(key)+2hash2(key), hash1(key)+3hash2(key) and so on, until an opening is found.

Let hash1(k) = k % 20 and hash2(k) = k % 6 + 1 and length of the circular array is 20

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Let's insert 34, 55, 12, 8, 45, 37, 88, 98, 54 and 32

		32				45			8				12	88	34	55	54	37	98	
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Probes		4				1			1				1	2	1	1	3	1	1	

Quadratic and Double Hashing Analysis

- Approximate average number of comparisons (probes) that a search requires:

$$\left[\frac{1}{\lambda}(\log_e \frac{1}{1-\lambda})\right] = \frac{-\log_e(1-\lambda)}{\lambda} \quad \textit{for a successful search}$$

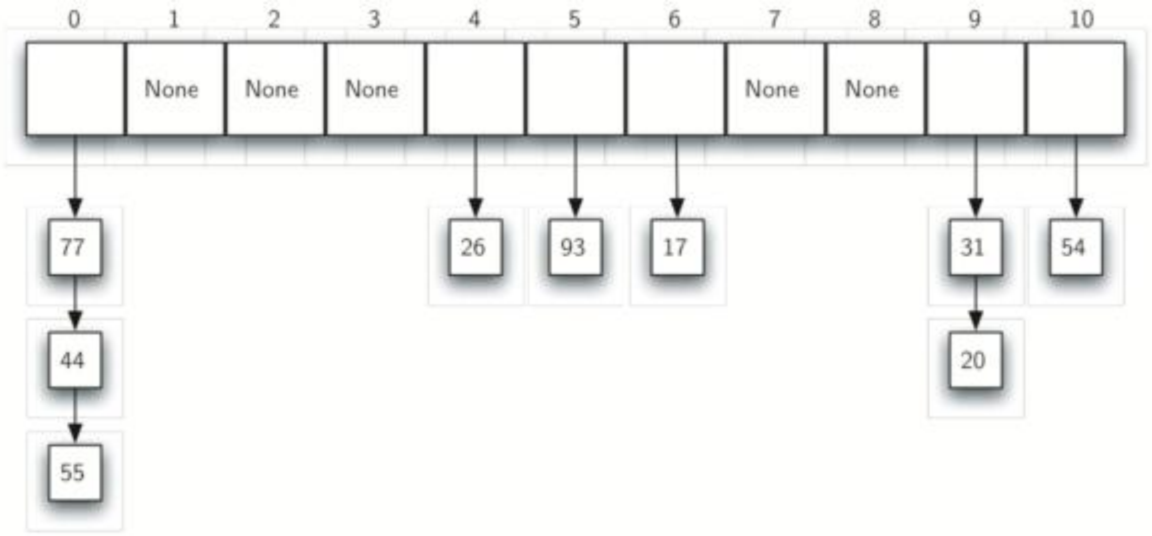
$$\frac{1}{1-\lambda} \quad \textit{for an unsuccessful search}$$

λ=load factor

- On average, both methods require fewer comparisons than linear probing

Closed Hashing

It allow each slot to hold a reference to a collection (or chain) of items. Chaining allows many items to exist at the same location in the hash table. When collisions happen, the item is still placed in the proper slot of the hash table. As more and more items hash to the same location, the difficulty of searching for the item in the collection increases.



When we want to search for an item, we use the hash function to generate the slot where it should reside. Since each slot holds a collection, we use a searching technique to decide whether the item is present. The advantage is that on the average there are likely to be many fewer items in each slot, so the search is perhaps more efficient.

Closed Hashing Analysis

- Approximate average number of comparisons (probes) that a search requires:

$$1+\frac{\lambda}{2} \quad \textit{for a successful search}$$

$$\lambda \quad \textit{for an unsuccessful search}$$

λ=load factor

- It is the most efficient collision resolution scheme
- Requires more storage (needs storage for pointers)
- It easily performs the deletion operation. Deletion is more difficult in open-addressing

What constitutes a good Hash function?

- A hash function should be easy and fast to compute
- A hash function should scatter the data evenly throughout the hash table
  - How well does the hash function scatter random data?
  - How well does the hash function scatter the non-random data?
- General principles
  - Hash function should use entire key in the calculation
  - If a hash function uses modulo arithmetic, the table size should be prime

Application of Hashing

- Compiler use hash tables to implement the symbol table (a data structure to keep track of declared variables)
- Game programs use hash tables to keep track of positions it has encountered (transposition table)
- Online spelling checker
- Substring pattern matching
- Document comparison
- Searching

Assignments

- A hash function is defined as $h_1(r, g, b) = r \wedge g \wedge b$ where ^ represents exclusive-or. Compute the following-
  - $h_1(255,18,15)$
  - $h_1(127,0,255)$
- A hash function is defined as $h_2(r, g, b) = 1024 * r + 32 * g + b$. Compute the following-
  - $h_2(255,18,15)$
  - $h_2(127,0,255)$
- If we use a hash table of size N = 521, and compute hash table indexes, then which hash function (h_1 or h_2) is likely to cause fewer collisions.
- Consider the 6-digit employee numbers - 123456, 654321, 112233, 223344, 999999, 888888, 111111, 222222, 333333, 444444, 555555, 666666, 777777. Find the 2-digit hash address of each number using
  - Division method
  - Mid-square method
  - Folding method with reversing
  - Folding method without reversion

Assignments

- We have a N (very large number of) sales records. Each record consists of the **id** number of the customer and the price. There are k customers, where k is still large, but not nearly as large as N. We want create a list of customers together with the total amount spent by each customer. That is, for each customer id, we want to know the sum of all the prices in sales records with that id. Design a sensible algorithm for doing this.
- What is the **average** and **worst** time complexity for insertion, deletion and access operation for the hash table.
- Suppose an unsorted linked list is in memory. Write a procedure SEARCH(INFO, LINK, START, ITEM, LOC) which
  - Finds the location LOC of ITEM in the list or sets LOC = NULL for an successful search
  - When the search is successful, interchanges ITEM with the element in front of it.
- Mathematically compute the worst case time complexity of binary search

Thank You

Home Work

- Write an algorithm RANDOM(DATA, N, K) which assigns N random integers between 1 and K to the array DATA.
- Write a C function **int Search(int A[], int n, int key)**, that returns 1 when key is present in 1st half of the array, returns 2 when the key is present in 2nd half of the array & returns 0 for unsuccessful search.
- Write the pseudo code of the search algorithm(s) whose worst case time complexity are O(n), O(log n) and O(1)
- What is a divide and conquer algorithm? Explain the concept of divide and conquer through the pseudo code binary search algorithm. Write down its time complexity for best, average and worst case.
- Assume a Hash Table of size 20 and the hash function = x mod 20 and key to insert are 11, 29, 42, 39, 40, 12, 14, 17, 13, 99 and 37. Compute the hash index/address using
  - Linear Probing
  - Quadratic Probing
  - Double Hashing

Supplementary Reading

- https://www.tutorialspoint.com/data_structures_algorithms/linear_search_algorithm.htm
- https://www.tutorialspoint.com/data_structures_algorithms/binary_search_algorithm.htm
- https://www.tutorialspoint.com/data_structures_algorithms/hash_data_structure.htm
- http://www.studytonight.com/data-structures/search-algorithms
- https://www.w3schools.in/data-structures-tutorial/searching-techniques/
- http://interactivepython.org/courselib/static/pythonds/SortSearch/searching.html
- http://btechsmartclass.com/DS/U4_T1.html
- http://www.geeksforgeeks.org/hashing-data-structure/
- http://www.geeksforgeeks.org/searching-algorithms/
- http://nptel.ac.in/courses/106102064/5
- http://nptel.ac.in/courses/106103069/15

FAQ

What is linear search?

Linear search tries to find an item in a sequentially arranged data type. These sequentially arranged data items known as array or list, are accessible in incrementing memory location. Linear search compares expected data item with each of data items in list or array.

What is hashing?

Hashing is a technique to convert a range of key values into a range of indexes of an array. By using hash tables, we can create an associative data storage where data index can be find by providing its key values.

What is binary search?

A binary search works only on sorted arrays. This search selects the middle which splits the entire list into two parts. First the middle is compared. This search first compares the target value to the mid of the list. If match occurs then index of item is returned. If it is not found, If middle item is greater than item then item is searched in sub-array to the right of the middle item other wise item is search in sub-array to the left of the middle item. This process continues on sub-array as well until the size of sub-array reduces to zero.

FAQ

What is Interpolation Search?

Interpolation search is an improved variant of binary search. This search algorithm works on the probing position of the required value. For this algorithm to work properly, the data collection should be in a sorted form and equally distributed. Runtime complexity of interpolation search algorithm is O(log (log n)) as compared to O(log n) of Binary Search. Navigate to following URL for further details

https://www.tutorialspoint.com/data_structures_algorithms/interpolation_search_algorithm.htm