UNIT V SEARCHING, SORTING AND HASHING TECHNIQUES

Searching-Linear Search – Binary Search – Sorting – Bubble Sort – Selection Sort- Insertion sort – Shell Sort –Radix Sort – Hashing – Hash Function – Separate Chaining – Open Addressing – Rehashing – Extendible Hashing

LINEAR SEARCH

Linear search, also called as sequential search, is a very simple method used for searching an array for a particular value. It works by comparing the value to be searched with every element of the array one by one in a sequence until a match is found. Linear search is mostly used to search an unordered list of elements (array in which data elements are not sorted).

For example, if an array A[] is declared and initialized as
int A[] = {10, 8, 2, 7, 3, 4, 9, 1, 6, 5};

Algorithm for Linear Search

```
LINEAR_SEARCH(A, N, VAL)
Step 1: [INITIALIZE] SET POS=-1
Step 2: [INITIALIZE] SETI=1
Step 3: Repeat Step 4 while I<=N
Step 4: IF A[I] = VAL SET POS=I PRINT POS Go to Step 6 [END OF IF] [END OF LOOP] Step 6:
EXIT SETI=I+1
Step 5: IF POS = –1 PRINT VALUE IS NOT PRESENT
```

BINARY SEARCH

Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

Example:
Now, let us consider how this mechanism is applied to search for a value in a sorted array.
Consider an array A[] that is declared and initialized as
int A[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
and the value to be searched is VAL = 9. The algorithm will proceed in the following manner.
BEG = 0, END = 10, MID = (0 + 10)/2 = 5
Now, VAL = 9 and A[MID] = A[5] = 5
A[5]islessthanVAL,therefore,wenowsearchforthevalueinthesecondhalfofthearray.So, we change the values of BEGand MID.
Now, BEG = MID + 1 = 6, END = 10, MID = (6 + 10)/2 =16/2 = 8
VAL = 9 and A[MID] = A[8] = 8
A[8] is less than VAL, therefore, we now search for the value in the second half of the segment.
So, again we change the values of BEG and MID.
Now, BEG = MID + 1 = 9, END = 10, MID = (9 + 10)/2 = 9

Now, VAL = 9 and A[MID] = 9.
Inthisalgorithm,weseethatBEGandENDarethebeginningandendingpositionsofthesegmentthat we are looking to search for the element. MID is calculated as (BEG + END)/2. Initially,BEG= lower_bound and END = upper_bound. The algorithm will terminate when A[MID] =VAL. Whenthe
algorithmends,wewillsetPOS=MID.POSisthepositionatwhichthevalueispresentinthearray.

However, if VAL is not equal to A[MID], then the values of BEG, END, and MID will be changed depending on whether VAL is smaller or greater than A[MID].

(a) IfVAL<A[MID],thenVALwillbepresentintheleftsegmentofthe array.So,thevalueofENDwill be changed as END = MID – 1.
(b) IfVAL>A[MID],thenVALwillbepresentintherightsegmentofthearray.So,thevalue ofBEG will be changed as BEG = MID + 1.

Finally, if VAL is not present in the array, then eventually, END will be less than BEG. When this happens, the algorithm will terminate and the search will be unsuccessful.

Algorithm for Binary Search

BINARY_SEARCH(A, lower_bound, upper_bound, VAL)
Step 1: [INITIALIZE] SET BEG = lower_bound END = upper_bound, POS=-1
Step 2: Repeat Steps 3 and 4 while BEG <= END
Step 3: SET MID = (BEG + END)/2
Step 4: IF A[MID] = VAL SET POS = MID PRINT POS Go to Step 6 ELSE IF A[MID] > VAL SET END = MID-1 ELSE SET BEG = MID+1 [END OF IF] [END OF LOOP]
Step 5: IF POS=-1 PRINT "VALUE IS NOT PRESENT IN THE ARRAY" [END OF IF]
Step 6: EXIT

SORTING

Sorting means arranging the elements of an array so that they are placed in some relevant order which may be either ascending or descending. That is, if A is an array, then the elements of A are arranged in a sorted order (ascending order) in such a way that A[0] < A[1] < A[2] < ...... < A[N].

For example, if we have an array that is declared and initialized as
int A[] = {21, 34, 11, 9, 1, 0, 22};
Then the sorted array (ascending order) can be given as:
A[] = {0, 1, 9, 11, 21, 22, 34;

There are two types of sorting:
Internal sorting : which deals with sorting the data stored in the computer's memory
External sorting: which deals with sorting the data stored in files. External sorting is applied when there is voluminous data that cannot be stored in the memory

BUBBLE SORT

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of O(n2) where n is the number of items.

Example:
Todiscussbubblesortindetail,letusconsideranarrayA[]thathasthefollowing elements:
A[] = {30, 52, 29, 87, 63, 27, 19, 54}

Pass 1:
(a) Compare 30 and 52. Since 30 < 52, no swapping is done.
(b) Compare 52 and 29. Since 52 > 29, swapping is done. 30, **29, 52**, 87, 63, 27, 19,54
(c) Compare 52 and 87. Since 52 < 87, no swapping is done.
(d) Compare 87 and 63. Since 87 > 63, swapping is done. 30, 29, 52, **63, 87**, 27, 19,54
(e) Compare 87 and 27. Since 87 > 27, swapping is done. 30, 29, 52, 63, **27, 87**, 19,54
(f) Compare 87 and 19. Since 87 > 19, swapping is done. 30, 29, 52, 63, 27, **19**, **87**,54
(g) Compare 87 and 54. Since 87 > 54, swapping is done. 30, 29, 52, 63, 27, 19, **54,87**

Observethataftertheendofthefirstpass,thelargestelementisplacedatthehighestind exof the array. All the other elements are stillunsorted.

Pass 2:
(a) Compare 30 and 29. Since 30 > 29, swapping is done. **29, 30**, 52, 63, 27, 19, 54, 87
(b) Compare 30 and 52. Since 30 < 52, no swapping is done.
(c) Compare 52 and 63. Since 52 < 63, no swapping is done.
(d) Compare 63 and 27. Since 63 > 27, swapping is done. 29, 30, 52, **27, 63**, 19, 54,87
(e) Compare 63 and 19. Since 63 > 19, swapping is done. 29, 30, 52, 27, **19, 63**, 54, 87
(f) Compare 63 and 54. Since 63 > 54, swapping is done. 29, 30, 52, 27, 19, **54, 63**,87

Observethataftertheendofthesecondpass,thesecondlargestelementisplacedatthese cond highest index of the array. All the other elements are stillunsorted.

Pass 3:
(a) Compare 29 and 30. Since 29 < 30, no swapping is done.
(b) Compare 30 and 52. Since 30 < 52, no swapping is done.
(c) Compare 52 and 27. Since 52 > 27, swapping is done. 29, 30, **27, 52**, 19, 54, 63,87
(d) Compare 52 and 19. Since 52 > 19, swapping is done. 29, 30, 27, **19, 52**, 54, 63,87
(e) Compare 52 and 54. Since 52 < 54, no swapping is done.

Observethataftertheendofthethirdpass,thethirdlargestelementisplacedatthethird highest index of the array. All the other elements are stillunsorted.

Pass 4:

(a) Compare 29 and 30. Since 29 < 30, no swapping is done.
(b) Compare 30 and 27. Since 30 > 27, swapping is done. 29, **27, 30**, 19, 52, 54, 63,87
(c) Compare 30 and 19. Since 30 > 19, swapping is done. 29, 27, **19, 30**, 52, 54, 63,87
(d) Compare 30 and 52. Since 30 < 52, no swapping is done.

Observe that after the end of the fourth pass, the fourth largest element is placed at the fourth highest index of the array. All the other elements are still unsorted.

Pass 5:

(a) Compare 29 and 27. Since 29 > 27, swapping is done. **27, 29,** 19, 30, 52, 54, 63, 87
(b) Compare 29 and 19. Since 29 > 19, swapping is done. 27, **19, 29**, 30, 52, 54, 63,87
(c) Compare 29 and 30. Since 29 < 30, no swapping is done.

Observethatafterthe endofthefifthpass,thefifthlargestelementisplacedatthefifthhighest index of the array. All the other elements are stillunsorted.

Pass 6:

(a) Compare 27 and 19. Since 27 > 19, swapping is done. **19, 27,** 29, 30, 52, 54, 63, 87
(b) Compare 27 and 29. Since 27 < 29, no swapping is done.

Observethatafterthe endofthesixthpass,thesixthlargestelementisplacedatthesixthlargest index of the array. All the other elements are stillunsorted.

(a) Compare 19 and 27. Since 19 < 27, no swapping is done

Algorithm for Bubble sort:

```
BUBBLE_SORT(A, N)
Step 1: Repeat Step 2 For 1 = to N-1
Step 2:      Repeat For J =  to N - I
Step 3:              IF A[J] > A[J + 1] SWAP A[J] andA[J+1]
                 [END OF INNER LOOP]
             [END OF OUTER LOOP]
Step 4: EXIT
```

SELECTION SORT

In selection sort, the first element in the list is selected and it is compared repeatedly with remaining all the elements in the list. If any element is smaller than the selected element (for Ascending order), then both are swapped. Then we select the element at second position in the list and it is compared with remaining all elements in the list. If any element is smaller than the selected element, then both are swapped. This procedure is repeated till the entire list is sorted**.**

The selection sort algorithm is performed using following steps:

Step 1: Select the first element of the list (i.e., Element at first position in the list).
Step 2: Compare the selected element with all other elements in the list.
Step 3: For every comparison, if any element is smaller than selected element (for Ascending order), then these two are swapped.

Step 4: Repeat the same procedure with next position in the list till the entire list is sorted.

Example:
Sort the array given below using selection sort.

39	9	81	45	90	27	72	18

PASS	POS	ARR[0]	ARR[1]	ARR[2]	ARR[3]	ARR[4]	ARR[5]	ARR[6]	ARR[7]
1	1	9	39	81	45	90	27	72	18
2	7	9	18	81	45	90	27	72	39
3	5	9	18	27	45	90	81	72	39
4	7	9	18	27	39	90	81	72	45
5	7	9	18	27	39	45	81	72	90
6	6	9	18	27	39	45	72	81	90
7	6	9	18	27	39	45	72	81	90

Algorithm for selection sort
SMALLEST(ARR, K, N, POS)

```
Step 1: [INITIALIZE] SET SMALL = ARR[K] Step 2: [INITIALIZE] SET POS = K
Step 3: Repeat for J = K+1 to N-1
        IF SMALL > ARR[J]
        SET SMALL = ARR[J]
        SET POS = J [END OF IF]
        [END OF LOOP]
Step 4: RETURN POS
```

SELECTION SORT(ARR, N)

```
Step 1: Repeat Steps 2 and 3 for K = 1 to N-1
Step 2:         CALL SMALLEST(ARR, K, N, POS)
Step 3:         SWAP A[K] with ARR[POS]
            [END OF LOOP]
Step 4: EXIT
```

INSERTION SORT

Insertion sort is a simple sorting algorithm that works the way we sort playing cards in our hands. Insertion sorts works by taking element from the list one by one and inserting them in their current position into a new sorted list.

Technique Insertion sort works as follows:

- The array of values to be sorted is divided into two sets. One that stores sorted values and another that contains unsorted values.
- The sorting algorithm will proceed until there are elements in the unsorted set.
- Suppose there are n elements in the array. Initially, the element with index 0 (assuming LB = 0) is in the sorted set. Rest of the elements are in the unsorted set.
- The first element of the unsorted partition has array index 1 (if LB = 0).

- During each iteration of the algorithm, the first element in the unsorted set is picked up and inserted into the correct position in the sorted set.

Example:
Consider an unsorted array as follows,
20 10 60 40 30 15

ORIGINAL	20	10	60	40	30	10	POSITIONS MOVED
After i = 1	10	20	60	40	30	15	1
After i = 2	10	20	60	40	30	15	0
After i = 3	10	20	40	60	30	15	1
After i = 4	10	20	30	40	60	15	2
After i = 5	10	15	20	30	40	60	4
Sorted Array	10	15	20	30	40	60	

Algorithm for Insertion Sort

```
INSERTION-SORT (ARR, N)
Step 1: Repeat Steps 2 to 5 for K = 1 to N – 1 Step 2:        SET TEMP = ARR[K]
Step 3:    SET J = K - 1
Step 4:    Repeat while TEMP <= ARR[J]
               SET ARR[J + 1] = ARR[J]
               SET J = J - 1 [END OF INNER LOOP]
Step 5:    SET ARR[J + 1] = TEMP [END OF LOOP]
Step 6: EXIT
```

Advantages of Insertion Sort

The advantages of this sorting algorithm are as follows:

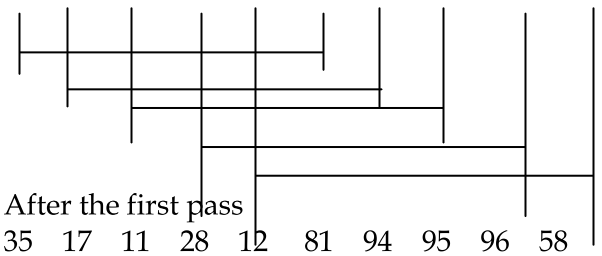
- It is easy to implement and efficient to use on small sets of data.
- It can be efficiently implemented on data sets that are already substantially sorted.
- It performs better than algorithmslike selection sort and bubble sort. Insertion sort algorithm is simpler than shell sort, with only a small trade-off in efficiency. It is over twice as fast as the bubble sort and almost 40 per cent faster than the selection sort.

SHELL SORT

The shell sort, sometimes called the "diminishing increment sort," improves on the insertion sort by breaking the original list into a number of smaller sub lists, each of which is sorted using an insertion sort. The unique way that these sub lists are chosen is the key to the shell sort. Instead of breaking the list into sub lists of contiguous items, the shell sort uses an increment i, sometimes called the gap, to create a sub list by choosing all items that are i items apart.

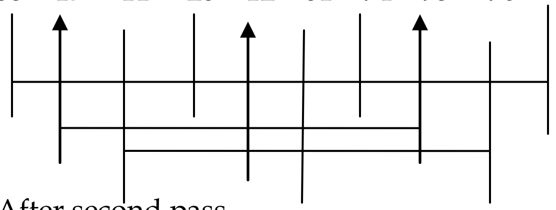
Example:
Consider an unsorted array as follows.
81 94 11 96 12 35 17 95 28 58
Here N=10, the first pass as K = 5 (10/2)

81 94 11 96 12 35 17 95 28 58

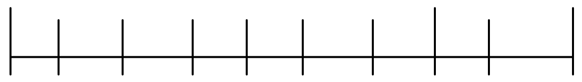


After the first pass
35 17 11 28 12 81 94 95 96 58
In second pass, K is reduced to 3

35 17 11 28 12 81 94 95 96 58



After second pass,
28 12 11 35 17 81 58 95 96 94
In third pass , K is reduced to 1
28 12 11 35 17 81 58 95 96 94



The final sorted array is
11 12 17 28 35 58 81 94 95 98

<u>Algorithm for shell sort</u>

```
Shell_Sort(Arr, n)
Step 1: SET FLAG = 1, GAP_SIZE = N
Step 2: Repeat Steps 3 to 6 while FLAG = 1 OR GAP_SIZE > 1
Step 3:     SET FLAG =
Step 4:     SET GAP_SIZE = (GAP_SIZE + 1) / 2
Step 5:     Repeat Step 6 for I =  to I < (N - GAP_SIZE)
Step 6:         IF Arr[I + GAP_SIZE] >Arr[I]
                    SWAP Arr[I + GAP_SIZE], Arr[I] SET FLAG =
Step 7: END
```

RADIX SORT

The idea of Radix Sort is to do digit by digit sort starting from least significant digit to most significant digit. Radix sort uses counting sort as a subroutine to sort.

The Radix Sort Algorithm
1) Do following for each digit i where i varies from least significant digit to the most significant digit.
a) Sort input array using counting sort (or any stable sort) according to the i’th digit.

Example:
Original, unsorted list:
170, 45, 75, 90, 802, 24, 2, 66

Sorting by least significant digit (1s place) gives: [*Notice that we keep 802 before 2, because 802 occurred before 2 in the original list, and similarly for pairs 170 & 90 and 45 & 75.]
170, 90, 802, 2, 24, 45, 75, 66
Sorting by next digit (10s place) gives: [*Notice that 802 again comes before 2 as 802 comes before 2 in the previous list.]
802, 2, 24, 45, 66, 170, 75, 90
Sorting by most significant digit (100s place) gives:
2, 24, 45, 66, 75, 90, 170, 802

Algorithm for Radix Sort

```
Step 1: Find the largest number in ARR as LARGE
Step 2: [INITIALIZE] SET NOP = Number of digits in LARGE Step 3: SET
PASS =
Step 4: Repeat Step 5 while PASS <= NOP-1
Step 5:         SET I =  and INITIALIZE buckets
Step 6:         Repeat Steps 7 to 9 while I<N-1
Step 7:              SET DIGIT = digit at PASSth place in A[I]
Step 8:              Add A[I] to the bucket numbered DIGIT
Step 9:              INCEREMENT bucket count for bucket numbered DIGIT [END OF
                LOOP]
Step 1 0 :      Collect the numbers in the bucket [END OF LOOP]
Step 11: END
```

HASHING

- Hashing is a technique used for performing insertion, deletions, and finds in constant average time.
- The Hash table data structure is an array of some fixed size, containing the keys. A key is a value associated with each record.

Hashing Function
A Hashing function is a key – to – address transformation, which acts upon a given key to compute the relative position of the key in an array.

A Simple Hash Function

HASH (KEYVALUE) = KEYVALUE MOD TABLESIZE

Example: Hash (92)
Hash (92) = 92 mod 10 =2
The key value '92' is placed in the relative location '2'.

ROUTINE FOR SIMPLE HASH FUNCTION:
Hash(const char *key, intTableSize)

```
{
        intHashVal = 0;
        While( *key != '\0' )
                HashVal +=*key++;
        Return HashVal % TableSize;
}
```

<u>DIFFERENT HASH FUNCTION</u>

Division Method

It is the most simple method of hashing an integer x. This method divides x by M and then uses the remainder obtained. In this case, the hash function can be given as h(x) = x mod M

Example:
Calculate the hash values of keys 1234 and 5462
Solution
Setting M = 97,
hash values can be calculated as:
h(1234) = 1234 % 97 = 70
h(5642) = 5642 % 97 = 16

Multiplication Method

The steps involved in the multiplication method are as follows:
Step 1: Choose a constant A such that 0 < A < 1
Step 2: Multiply the key k by A.
Step 3: Extract the fractional part of kA
Step 4: Multiply the result of Step 3 by the size of hash table (m).
Hence, the hash function can be given as:
h(k) = Î m (kA mod 1) ˚

Example:
Given a hash table of size 1000, map the key 12345 to an appropriate location in the hash table
Solution We will use A = 0.618033, m = 1000, and k = 12345
h(12345) = Î 1000 (12345 ¥ 0.618033 mod 1) ˚ h(12345)
= Î 1000 (7629.617385 mod 1) ˚ h(12345)
= Î 1000 (0.617385) ˚ h(12345)
= Î 617.385 ˚ h(12345)
= 617

Mid-Square Method

The mid-square method is a good hash function which works in two steps:
Step 1: Square the value of the key. That is, find k2 .
Step 2: Extract the middle r digits of the result obtained in Step 1

Example:
Calculate the hash value for keys 1234 and 5642 using the mid-square method. The hash table has 100 memory locations.

Solution: Note that the hash table has 100 memory locations whose indices vary from 0 to 99. This means that only two digits are needed to map the key to a location in the hash table, so r = 2.
When k = 1234, k2 = 1522756, h (1234) = 27 When k = 5642, k2 = 31832164, h (5642) = 21 Observe that the 3rd and 4th digits starting from the right are chosen

Folding Method
The folding method works in the following two steps:
Step 1: Divide the key value into a number of parts. That is, divide k into parts k1 , k2 , ..., kn where each part has the same number of digits except the last part which may have lesser digits than the other parts.
Step 2: Add the individual parts. That is, obtain the sum of k1 + k2 + ... + kn . The hash value is produced by ignoring the last carry, if any.

Example:
Given a hash table of 100 locations, calculate the hash value using folding method for keys 5678, 321, and 34567
Solution
Sincethereare100memorylocationstoaddress,wewillbreakthekeyintopartswhereeachpart (except the last) will contain two digits. The hash values can be obtained as shownbelow:

key	5678	32 1	34567
Parts	56 and 78	32 and 1	34, 56 and 7
Sum	134	33	97
Hash value	34 (ignore the last carry)	33	97

COLLISIONS

Collision occurs when a hash value of a record being inserted hashes to an address (i.e. Relative position) that already contain a different record. (ie) When two key values hash to the same position.

Collision Resolution
The process of finding another position for the collide record.

Some of the collision Resolution Techniques
- Separate chaining
- Open Addressing
- Double Hashing

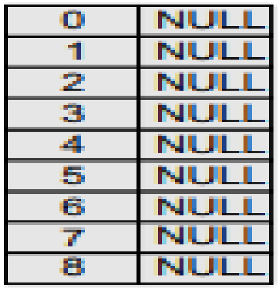
<u>**Struture of the node**</u>
```
typedefstructnode_HT
{
int value;
struct node *next;
 }node;
```
Code to initialize a chained hash table
```
void initializeHashTable (node *hash_table[], int m)
{ int i;
for(i=0i<=m;i++)
hash_table[i]=NULL;
}
```

Code to insert a value
```
node *insert_value( node *hash_table[], intval)
{
 node *new_node;
new_node = (node *)malloc(sizeof(node));
new_node value = val;
new_node next = hash_ table[h(x)];
hash_table[h(x)] = new_node;
}
```
Code to search a value
```
node *search_value(node *hash_table[], intval)
{
node *ptr;
ptr = hash_table[h(x)];
while ( (ptr!=NULL) && (ptr –> value != val))
ptr = ptr –> next;
   if (ptr–>value == val)
return ptr;
else
return NULL;
 }
```

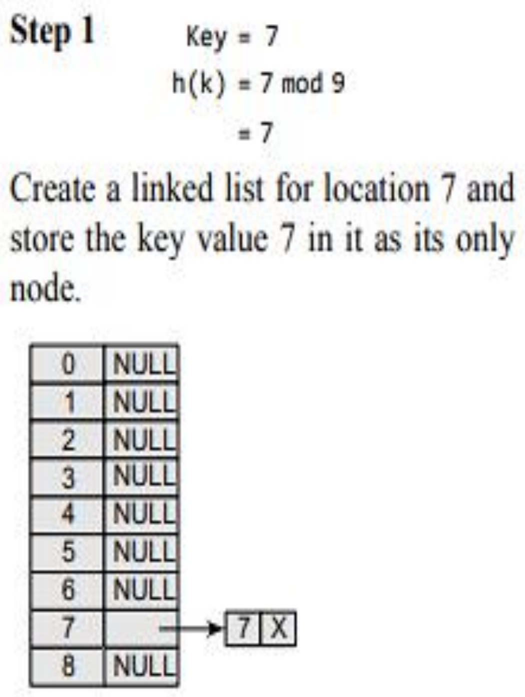
Code to delete a value
```
void delete_value (node *hash_table[], intval)
{
node *save, *ptr;
save = NULL;
ptr = hash_table[h(x)];
while ((ptr != NULL) && (ptr value != val))
{
save = ptr;
ptr = ptr next;
 }
      if (ptr != NULL)
{
save next = ptr next;
free (ptr);
}
else
printf("\n VALUE NOT FOUND"); }
```

Example:
Insert the keys 7, 24, 18, 52, 36, 54, 11, and 23 in a chained hash table of 9 memory locations. Use h(k) = k mod m

In this case, m=9. Initially, the hash table can be given as:

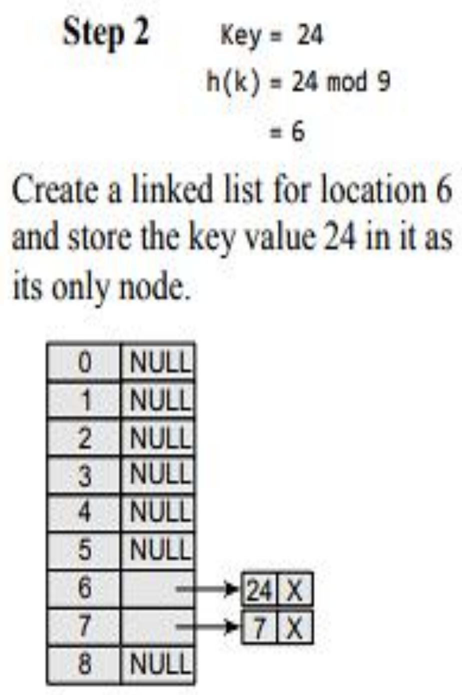


0	NULL
1	NULL
2	NULL
3	NULL
4	NULL
5	NULL
6	NULL
7	NULL
8	NULL



Step 1 Key = 7
h(k) = 7 mod 9
= 7

Create a linked list for location 7 and store the key value 7 in it as its only node.

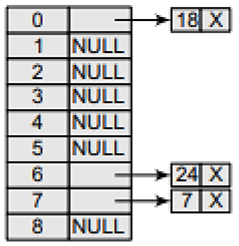


Step 2 Key = 24
h(k) = 24 mod 9
= 6

Create a linked list for location 6 and store the key value 24 in it as its only node.

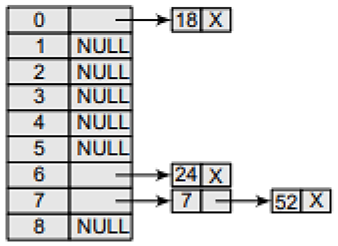
Step 3 Key = 18
h(k)= 18 mod 9 = 0

Create a linked list for location 0 and store the key value 18 in it as its only node.



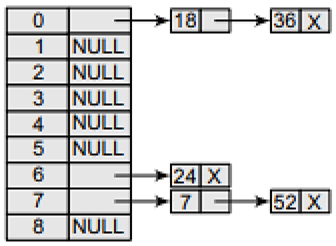
Step 4 Key = 52
h(k) = 52 mod 9 = 7

Insert 52 at the end of the linked list of location 7.



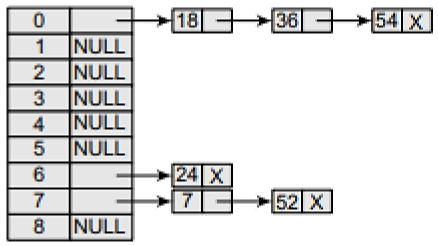
Step 5: Key = 36
h(k)= 36 mod 9 = 0

Insert 36 at the end of the linked list of location 0.



Step 6: Key = 54
h(k) = 54 mod 9 = 0

Insert 54 at the end of the linked list of location 0.



Step 7: Key = 11
h(k)= 11 mod 9 = 2

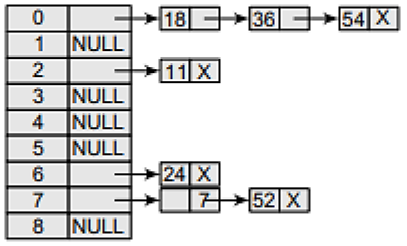
Create a linked list for location 2 and store the key value 11 in it as its only node.

Step 8: Key = 23
h(k) = 23 mod 9 = 5

Create a linked list for location 5 and store the key value 23 in it as its only node.

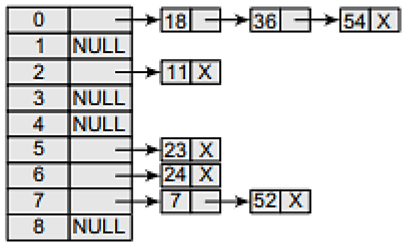
Step 7: Key = 11
h(k)= 11 mod 9 = 2

Create a linked list for location 2 and store the key value 11 in it as its only node.



Step 8: Key = 23
h(k) = 23 mod 9 = 5

Create a linked list for location 5 and store the key value 23 in it as its only node.



Advantages
1) Simple to implement.
2) Hash table never fills up, we can always add more elements to chain.
3) Less sensitive to the hash function or load factors.
4) It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

Disadvantages:

1) Cache performance of chaining is not good as keys are stored using linked list. Open addressing provides better cache performance as everything is stored in same table.
2) Wastage of Space (Some Parts of hash table are never used)
3) If the chain becomes long, then search time can become O(n) in worst case.
4) Uses extra space for links

OPEN ADDRESSING:

Like separate chaining, open addressing is a method for handling collisions. In Open Addressing, all elements are stored in the hash table itself. So at any point, size of the table must be greater than or equal to the total number of keys (Note that we can increase table size by copying old data if needed).

- Insert(k): Keep probing until an empty slot is found. Once an empty slot is found, insert k.
- Search(k): Keep probing until slot's key doesn't become equal to k or an empty slot is reached.
- Delete(k): Delete operation is interesting. If we simply delete a key, then search may fail. So slots of deleted keys are marked specially as "deleted".

Insert can insert an item in a deleted slot, but the search doesn't stop at a deleted slot.

LINEAR PROBING:

The simplest approach to resolve a collision is linear probing. In this technique, if a value is already stored at a location generated by h(k),
then the following hash function is used to resolve the collision:

$$h(k, i) = [h¢(k) + i] \bmod m$$

Example:

Consider a hash table of size 10. Using linear probing, insert the keys 72, 27, 36, 24, 63, 81, 92, and 101 into the table.
Let h¢(k) = k mod m, m = 10
Initially, the hash table can be given as:

0	1	2	3	4	5	6	7	8	9
–1	–1	–1	–1	–1	–1	–1	–1	–1	–1

Step1 Key =72

h(72, 0) = (72 mod 10 + 0) mod 10
= (2) mod 10
= 2

Since T[2] is vacant, insert key 72 at this location.

–1	–1	72	–1	–1	–1	–1	–1	–1	–1

Step2 Key =27

h(27, 0) = (27 mod 10 + 0) mod 10
= (7) mod 10
= 7

Since T[7] is vacant, insert key 27 at this location.

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	–1	–1	–1	27	–1	–1

Step3 Key =36

h(36, 0) = (36 mod 10 + 0) mod 10
= (6) mod 10
= 6

Since T[6] is vacant, insert key 36 at this location.

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	–1	–1	36	27	–1	–1

Step4 Key =24

h(24, 0) = (24 mod 10 + 0) mod 10
= 4

Since T[4] is vacant, insert key 24 at this location.

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	24	–1	36	27	–1	–1

Step5 Key =63

h(63, 0) = (63 mod 10 + 0) mod 10
= (3) mod 10
= 3

Since T[3] is vacant, insert key 63 at this location.

0	1	2	3	4	5	6	7	8	9
–1	–1	72	63	24	–1	36	27	–1	–1

Step6 Key =81

h(81, 0) = (81 mod 10 + 0) mod 10
= (1) mod 10
= 1

Since T[1] is vacant, insert key 81 at this location.

0	1	2	3	4	5	6	7	8	9
0	81	72	63	24	–1	36	27	–1	–1

Step7 Key =92

h(92, 0) = (92 mod 10 + 0) mod 10
= (2) mod 10
= 2

Now T[2] is occupied, so we cannot store the key 92 in T[2]. Therefore, try again for the next location. Thus probe, i = 1, this time.

Key =92
h(92, 1) = (92 mod 10 + 1) mod 10
= (2 + 1) mod 10
= 3

Now T[3] is occupied, so we cannot store the key 92 in T[3]. Therefore, try again for the next location. Thus probe, i = 2, this time.

Key =92
h(92, 2) = (92 mod 10 + 2) mod 10
= (2 + 2) mod 10
= 4

Now T[4] is occupied, so we cannot store the key 92 in T[4]. Therefore, try again for the next location. Thus probe, i = 3, this time.

Key =92
h(92, 3) = (92 mod 10 + 3) mod 10
= (2 + 3) mod 10
= 5

Since T[5] is vacant, insert key 92 at this location.

0	1	2	3	4	5	6	7	8	9
–1	81	72	63	24	92	36	27	–1	–1

Step8 Key =101

h(101, 0) = (101 mod 10 + 0) mod 10
= (1) mod 10
= 1

Now T[1]is occupied, so we cannot store the key 101 in T[1]. Therefore, try again for the next location. Thus probe, i = 1, this time.

Key =101
h(101, 1) = (101 mod 10 + 1) mod 10
= (1 + 1) mod 10
=2

T[2]isalsooccupied,sowecannotstorethekeyinthislocation.Theprocedurewillberepeated until the hash function generates the address of location 8 which is vacant and can be used to store the value init

QUADRATIC PROBING

Quadratic probing is similar to linear probing and the only difference is the interval between

successive probes or entry slots. Here, when the slot at a hashed index for an entry record is already occupied, you must start traversing until you find an unoccupied slot. The interval between slots is computed by adding the successive value of an arbitrary polynomial in the original hashed index.
Let us assume that the hashed index for an entry is index and at index there is an occupied slot. The probe sequence will be as follows:
index = index % hashTableSize
index = (index + 1) % hashTableSize
index = (index + 4) % hashTableSize
index = (index + 9) % hashTableSize

Quadratic Probing Example
Consider a hash table of size 10. Using quadratic probing, insert the keys 72, 27, 36, 24, 63, 81, and 101 into the table. Take c1 = 1 and c2 = 3

Solution:

Let h¢(k)=kmodm, m=10
Initially, the hash table can be given as:

0	1	2	3	4	5	6	7	8	9
–1	–1	–1	–1	–1	–1	–1	–1	–1	–1

We have,

h(k,i)=[h¢(k)+ci+ci^2]modm

1 2

Step1 Key =72
h(72, 0) = [72 mod 10 + 1 ¥ 0 + 3 ¥ 0] mod 10
= [72 mod 10] mod 10
= 2 mod 10
= 2
Since T[2] is vacant, insert the key 72 in T[2]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	–1	–1	–1	–1	–1	–1

Step2 Key =27
h(27, 0) = [27 mod 10 + 1 ¥ 0 + 3 ¥ 0] mod 10
= [27 mod 10] mod 10
= 7 mod 10
= 7
Since T[7] is vacant, insert the key 27 in T[7]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	–1	–1	–1	27	–1	–1

Step3 Key =36

h(36, 0) = [36 mod 10 + 1 ¥ 0 + 3 ¥ 0] mod 10
= [36 mod 10] mod 10
= 6 mod 10
= 6

Since T[6] is vacant, insert the key 36 in T[6]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	–1	–1	36	27	–1	–1

Step4 Key =24

h(24, 0) = [24 mod 10 + 1 ¥ 0 + 3 ¥ 0] mod 10
= [24 mod 10] mod 10
= 4 mod 10
= 4

Since T[4] is vacant, insert the key 24 in T[4]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	24	–1	36	27	–1	–1

Step5 Key =63

h(63, 0) = [63 mod 10 + 1 ¥ 0 + 3 ¥ 0] mod 10
= [63 mod 10] mod 10
= 3 mod 10
= 3

Since T[3] is vacant, insert the key 63 in T[3]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	63	24	–1	36	27	–1	–1

Step6 Key =81

h(81,0) = [81 mod 10 + 1 ¥ 0 + 3 ¥ 0] mod 10
= [81 mod 10] mod 10
= 81 mod 10
= 1

Since T[1] is vacant, insert the key 81 in T[1]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	81	72	63	24	–1	36	27	–1	–1

Step7 Key =101

h(101,0) = [101 mod 10 + 1 ¥ 0 + 3 ¥ 0] mod 10
= [101 mod 10 + 0] mod 10
= 1 mod 10
= 1

Since T[1] is already occupied, the key 101 cannot be stored in T[1]. Therefore, try again for next location. Thus probe, i = 1, this time.

Key =101

h(101,0) = [101 mod 10 + 1 ¥ 1 + 3 ¥ 1] mod 10
= [101 mod 10 + 1 + 3] mod 10
= [101 mod 10 + 4] mod 10
= [1 + 4] mod 10
= 5 mod 10
= 5

Since T[5] is vacant, insert the key 101 in T[5]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	81	72	63	24	101	36	27	–1	–1

DOUBLE HASHING

Double hashing is similar to linear probing and the only difference is the interval between successive probes. Here, the interval between probes is computed by using two hash functions. Let us say that the hashed index for an entry record is an index that is computed by one hashing function and the slot at that index is already occupied. You must start traversing in a specific probing sequence to look for an unoccupied slot.

The Double Hashing is:

f(i) = i . hash2 (x)

Where hash2(X) =R – (X mod R)

To choose a prime R < size

Example:

Consider a hash table of size = 10. Using double hashing, insert the keys 72, 27, 36, 24, 63, 81, 92, and 101 into the table. Take h1 = (k mod 10) and h2 = (k mod 8)

Solution:

Initially, the hash table can be given as:

–1	–1	–1	–1	–1	–1	–1	–1	–1	–1

We have,

$h(k, i) = [h_1(k) + ih_2(k)] \bmod m$

Step1 Key =72

h(72, 0) = [72 mod 10 + (0 ¥ 72 mod 8)] mod 10
= [2 + (0 ¥ 0)] mod 10

= 2 mod 10
= 2
Since T[2] is vacant, insert the key 72 in T[2]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	–1	–1	–1	–1	–1	–1

Step2 Key =27
h(27, 0) = [27 mod 10 + (0 ¥ 27 mod 8)] mod 10
= [7 + (0 ¥ 3)] mod 10
= 7 mod 10
= 7
Since T[7] is vacant, insert the key 27 in T[7]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	–1	–1	–1	27	–1	–1

Step3 Key =36
h(36, 0) = [36 mod 10 + (0 ¥ 36 mod 8)] mod 10
= [6 + (0 ¥ 4)] mod 10
= 6 mod 10
= 6
Since T[6] is vacant, insert the key 36 in T[6]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	–1	–1	36	27	–1	–1

Step4 Key =24
h(24, 0) = [24 mod 10 + (0 ¥ 24 mod 8)] mod 10
= [4 + (0 ¥ 0)] mod 10
= 4 mod 10
= 4
Since T[4] is vacant, insert the key 24 in T[4]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	24	–1	36	27	–1	–1

Step5 Key =63

h(63, 0) = [63 mod 10 + (0 ¥ 63 mod 8)] mod 10
= [3 + (0 ¥ 7)] mod 10
= 3 mod 10
= 3

Since T[3] is vacant, insert the key 63 in T[3]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	63	24	–1	36	27	–1	–1

Step6 Key =81
h(81, 0) = [81 mod 10 + (0 ¥ 81 mod 8)] mod 10
= [1 + (0 ¥ 1)] mod 10
= 1 mod 10
= 1

Since T[1] is vacant, insert the key 81 in T[1]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	81	72	63	24	–1	36	27	–1	–1

Step7 Key =92
h(92, 0) = [92 mod 10 + (0 ¥ 92 mod 8)] mod 10
= [2 + (0 ¥ 4)] mod 10
= 2 mod 10
= 2

Now T[2] is occupied, so we cannot store the key 92 in T[2]. Therefore, try again for the next location. Thus probe, i = 1, this time.

Key =92
h(92, 1) = [92 mod 10 + (1 ¥ 92 mod 8)] mod 10
= [2 + (1 ¥ 4)] mod 10
= (2 + 4) mod 10
= 6 mod 10
= 6

Now T[6] is occupied, so we cannot store the key 92 in T[6]. Therefore, try again for the next location. Thus probe, i = 2, this time.

Key =92
h(92, 2) = [92 mod 10 + (2 ¥ 92 mod 8)] mod 10
= [2 + (2 ¥ 4)] mod 10
= [2 + 8] mod 10
= 10 mod 10
= 0

Since T[0] is vacant, insert the key 92 in T[0]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
92	81	72	63	24	–1	36	27	–1	–1

Step8 Key =101

h(101, 0) = [101 mod 10 + (0 ¥ 101 mod 8)] mod 10
= [1 + (0 ¥ 5)] mod 10
= 1 mod 10
= 1

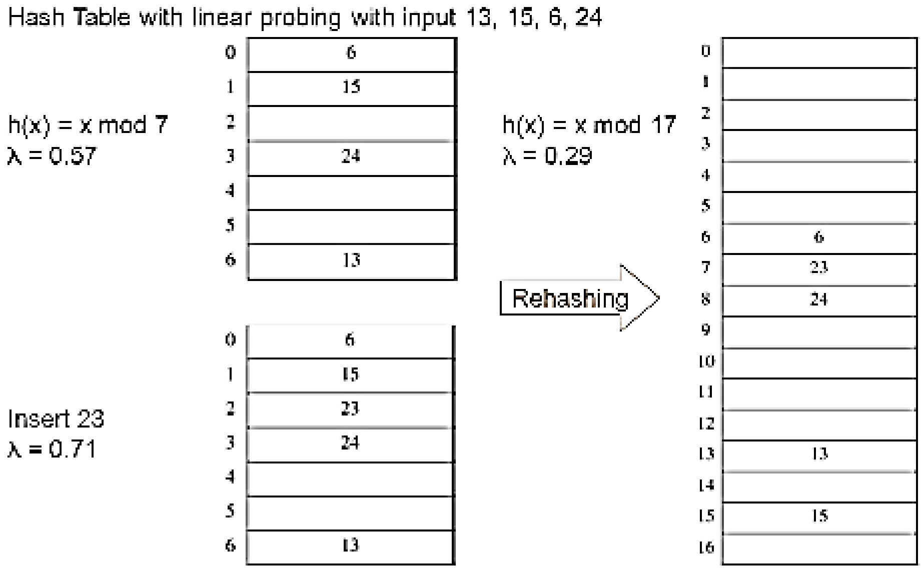
Now T[1] is occupied, so we cannot store the key 101 in T[1]. Therefore, try again for the next location. Thus probe, i = 1, this time.

Key =101

h(101, 1) = [101 mod 10 + (1 ¥ 101 mod 8)] mod 10
= [1 + (1 ¥ 5)] mod 10
= [1 + 5] mod 10
=6

REHASHING

- If the table is close to full, the search time grows and may become equal to the table size.
- When the load factor exceeds a certain value (e.g. greater than 0.5) we do rehashing :
- Build a second table twice as large as the original and rehash there all the keys of the original table.
- Rehashing is expensive operation, with running time O(N)
  However, once done, the new hash table will have good performance.



Routine for rehashing for open addressing hash tables

Hashtable Rehash(Hashtable H)

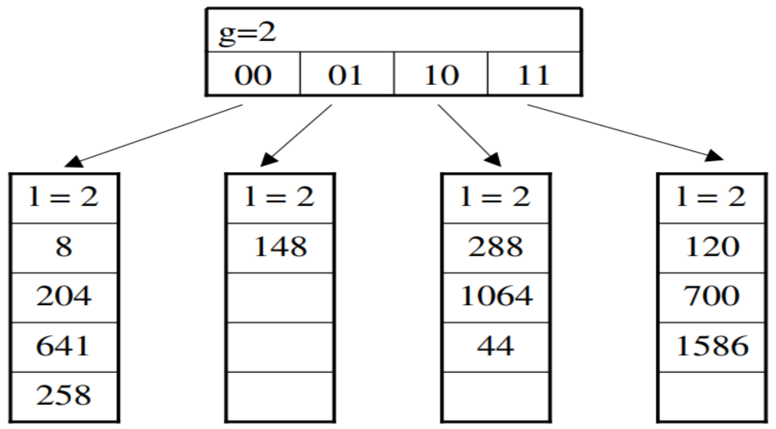
```
{
Int I, oldsize;
Cell  *oldcells;
Oldcells = H →Thecells;
Oldsize – H →Tablesize;
H = InitializeTable (2 * oldsize);
For(i=0; i<oldsize; i++)
If(oldcells [i]. Info == Legitimate)
Insert(oldcells [i]. Element, H);
Free(oldcells);
Return H;
}
```

EXTENDIBLE HASHING

Used when the amount of data is too large to fit in main memory and external storage is used.
N records in total to store, M records in one disk block
The problem: in ordinary hashing several disk blocks may be examined to find an element - a time consuming process.
Extendible hashing: no more than two blocks are examined.

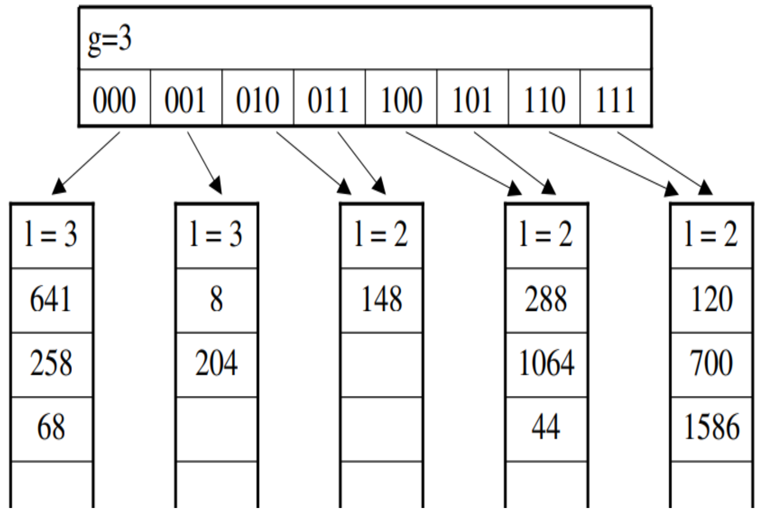
Idea:
Keys are grouped according to the first m bits in their code.
Each group is stored in one disk block.
If the block becomes full and no more records can be inserted, each group is split into two, and m+1 bits are considered to determine the location of a record.
Extendible Hashing Example • Suppose that g=2 and bucket size = 4. • Suppose that we have records with these keys and hash function h(key) = key mod 64:

key	**h(key) = key mod 64**	**bit pattern**
288	32	100000
8	8	001000
1064	40	101000
120	56	111000
148	20	010100
204	12	001100
641	1	000001
700	60	111100
258	2	000010
1586	50	110010
44	44	101010

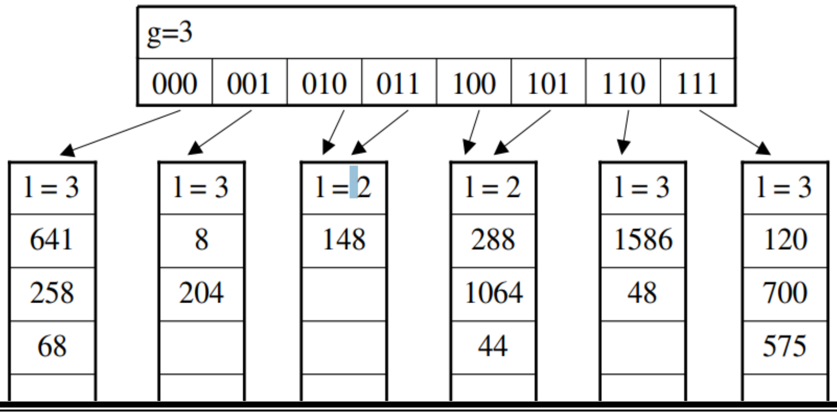


Bucket and directory split

- Insert 68
- 68 mod 64 = 4 = 000100



- Insert 48 and 575
- 48 mod 64 = 48 = 110000
- 575 mod 64 = 63 = 111111

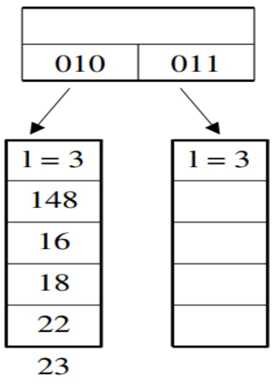


Multiple splits

- Insert 16, 18, 22, 23
- 16 mod 64 = 16 = 010000
- 18 mod 64 = 18 = 010010
- 22 mod 64 = 22 = 010110
- 23 mod 64 = 23 = 010111

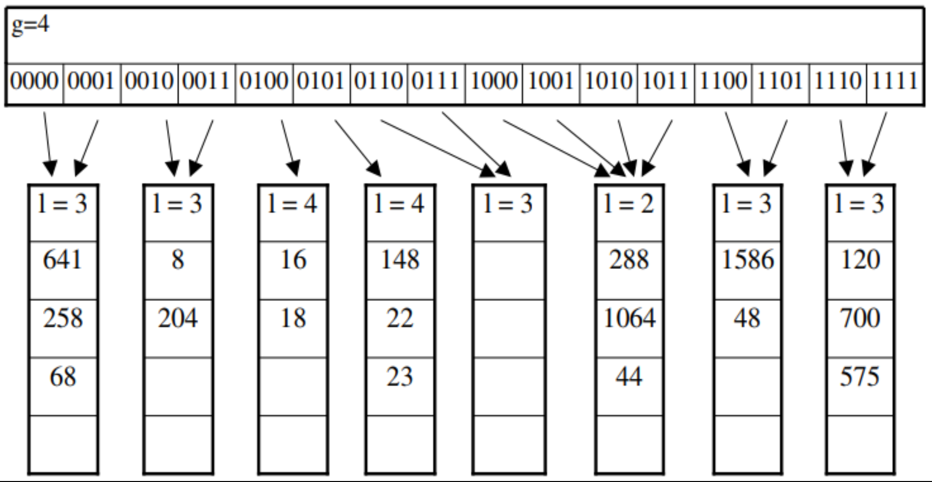
Setting l=3 gives this intermediate (partial) picture…

Continue to next page…



Multiple splits, continued

- Setting l=4 (and thus g=4) gives this final result…


g=4
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
l = 3
641
258
68
l = 3
8
204
l = 4
16
18
l = 4
148
22
23
l = 3
l = 2
288
1064
44
l = 3
1586
48
l = 3
120
700
575