ADVANCED DATA STRUCTURES

24CS2255F

CO1: Sorting & Hashing:

1. Introduction To Sorting

Sorting is the process of arranging a collection of data elements into a specific order, such as numerical or lexicographical (alphabetical) ascending or descending order. This organization makes data easier to search, retrieve, and analyze efficiently.

Sorting methods are generally categorized into two types based on where the data resides during the sorting process:

Internal Sorting

Internal sorting refers to sorting algorithms that operate entirely within the computer's **main memory (RAM)**. This method is possible when the entire dataset is small enough to fit into the available main memory at once.

- **Characteristics:**
    - **Speed:** Generally faster because all operations occur in RAM, which has much faster access speeds than external storage devices.
    - **Memory Access:** Benefits from the random-access nature of main memory.
    - **Use Cases:** Suitable for small to medium-sized datasets.
- **Examples of Algorithms:**
    - Bubble Sort
    - Insertion Sort
    - Selection Sort
    - Quick Sort
    - Heap Sort

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 $O(n^2)$ where **n** is the number of items.

Bubble Sort Algorithm

Bubble Sort is an elementary sorting algorithm, which works by repeatedly exchanging adjacent elements, if necessary. When no exchanges are required, the file is sorted.

We assume **list** is an array of **n** elements. We further assume that **swap** function swaps the values of the given array elements.

Step 1 − Check if the first element in the input array is greater than the next element in the array.

Step 2 − If it is greater, swap the two elements; otherwise move the pointer forward in the array.

Step 3 − Repeat Step 2 until we reach the end of the array.

Step 4 − Check if the elements are sorted; if not, repeat the same process (Step 1 to Step 3) from the last element of the array to the first.

Step 5 − The final output achieved is the sorted array.

Algorithm: Sequential-Bubble-Sort (A)

fori ← 1 to length [A] do

for j ← length [A] down-to i +1 do

  if A[A] < A[j-1] then

    Exchange A[j] ↔ A[j-1]

Pseudocode

We observe in algorithm that Bubble Sort compares each pair of array element unless the whole array is completely sorted in an ascending order. This may cause a few complexity issues like what if the array needs no more swapping as all the elements are already ascending.

To ease-out the issue, we use one flag variable **swapped** which will help us see if any swap has happened or not. If no swap has occurred, i.e. the array requires no more processing to be sorted, it will come out of the loop.

Pseudocode

```
voidbubbleSort(int numbers[], intarray_size){
  inti, j, temp;
  for (i = (array_size - 1); i>= 0; i--)
  for (j = 1; j <= i; j++)
  if (numbers[j-1] > numbers[j]){
    temp = numbers[j-1];
    numbers[j-1] = numbers[j];
    numbers[j] = temp;
  }
}
```

Analysis

Here, the number of comparisons are

$$1 + 2 + 3 + ... + (n - 1) = n(n - 1)/2 = O(n2)$$

Clearly, the graph shows the n^2 nature of the bubble sort.

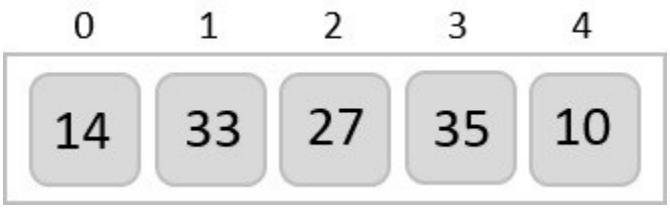
In this algorithm, the number of comparison is irrespective of the data set, i.e. whether the provided input elements are in sorted order or in reverse order or at random.

Memory Requirement

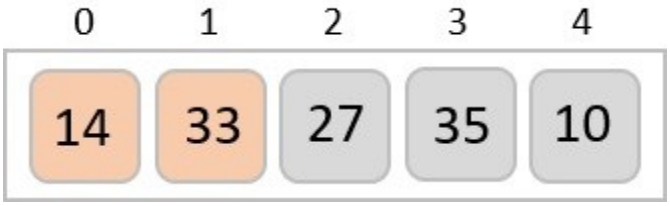
From the algorithm stated above, it is clear that bubble sort does not require extra memory.

Example

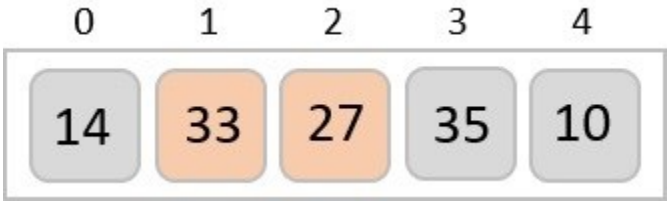
We take an unsorted array for our example. Bubble sort takes (n2) time so we're keeping it short and precise.



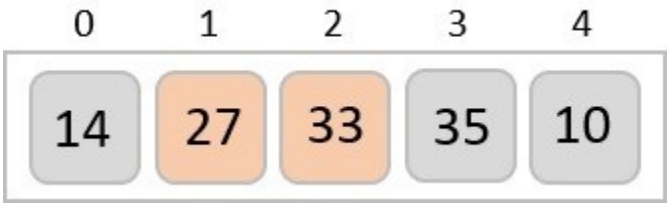
Bubble sort starts with very first two elements, comparing them to check which one is greater.



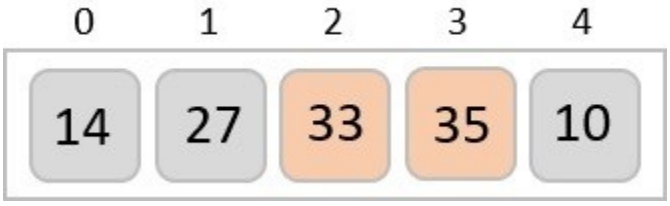
In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.



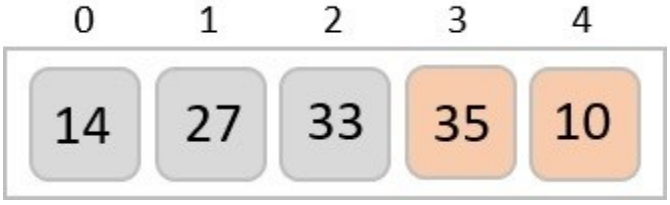
We find that 27 is smaller than 33 and these two values must be swapped.



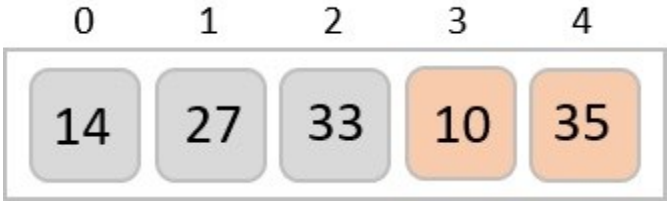
Next we compare 33 and 35. We find that both are in already sorted positions.



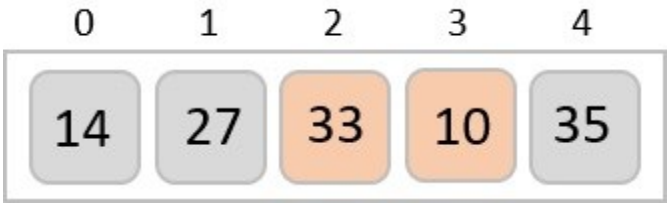
Then we move to the next two values, 35 and 10.

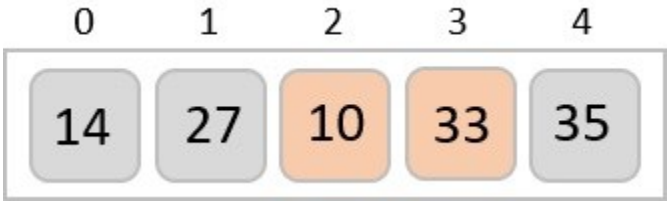


We know then that 10 is smaller 35. Hence they are not sorted. We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −

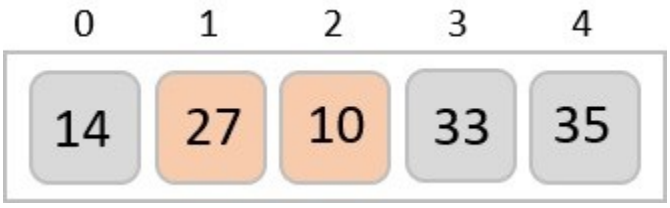


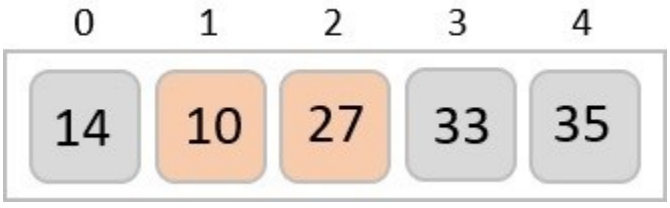
To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −

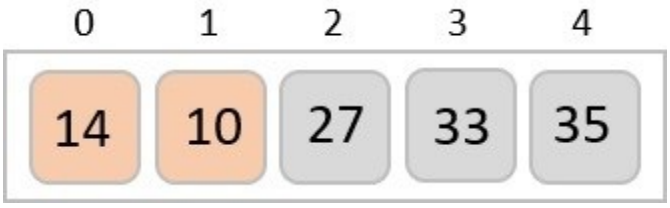


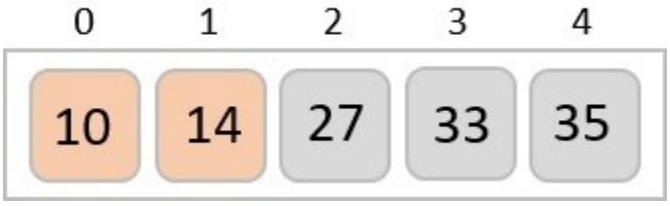


Notice that after each iteration, at least one value moves at the end.

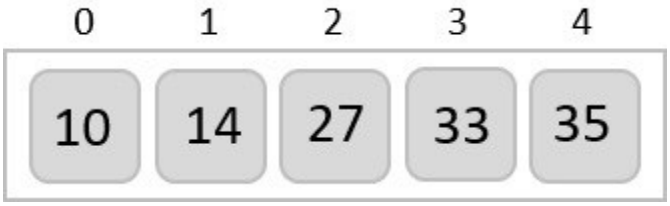








And when there's no swap required, bubble sort learns that an array is completely sorted.



Now we should look into some practical aspects of bubble sort.

Implementation

One more issue we did not address in our original algorithm and its improvised pseudocode, is that, after every iteration the highest values settles down at the end of the array. Hence, the next iteration need not include already sorted elements. For this purpose, in our implementation, we restrict the inner loop to avoid already sorted values.

Insertion Sort

Insertion sort is a very simple method to sort numbers in an ascending or descending order. This method follows the incremental method. It can be compared with the technique how cards are sorted at the time of playing a game.

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'inserted' in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, insertion sort.

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of (n^2), where n is the number of items.

Insertion Sort Algorithm

Now we have a bigger picture of how this sorting technique works, so we can derive simple steps by which we can achieve insertion sort.

Step 1 − If it is the first element, it is already sorted. return 1;

Step 2 − Pick next element

Step 3 − Compare with all elements in the sorted sub-list

Step 4 − Shift all the elements in the sorted sub-list that is greater than the value to be sorted

Step 5 − Insert the value

Step 6 − Repeat until list is sorted

Pseudocode

```
Algorithm: Insertion-Sort(A)
for j = 2 to A.length
   key = A[j]
   i = j  1
   while i > 0 and A[i] > key
```

```
    A[i + 1] = A[i]
    i = i -1
  A[i + 1] = key
```

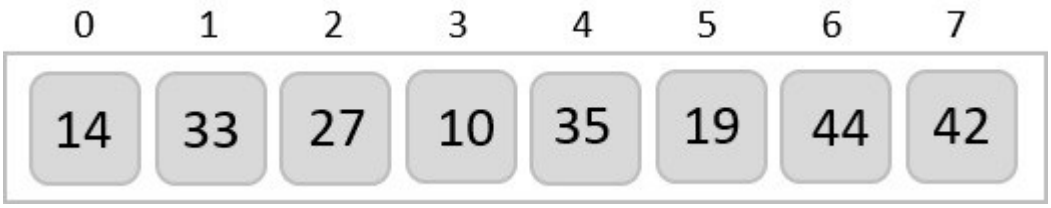
Analysis

Run time of this algorithm is very much dependent on the given input.

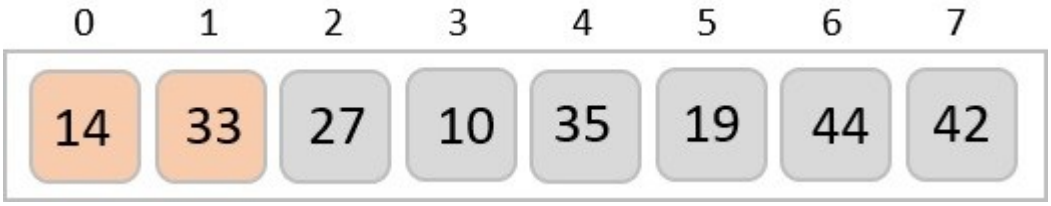
If the given numbers are sorted, this algorithm runs in O(n) time. If the given numbers are in reverse order, the algorithm runs in $O(n^2)$ time.

Example

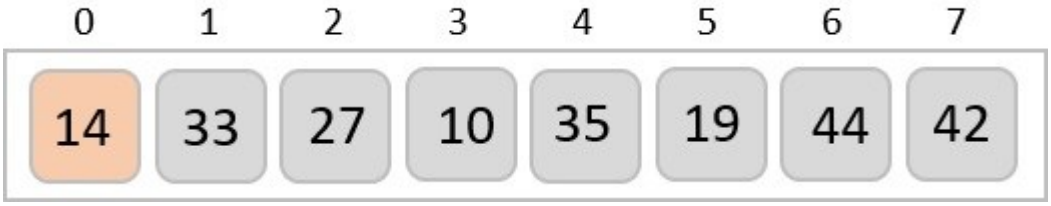
We take an unsorted array for our example.



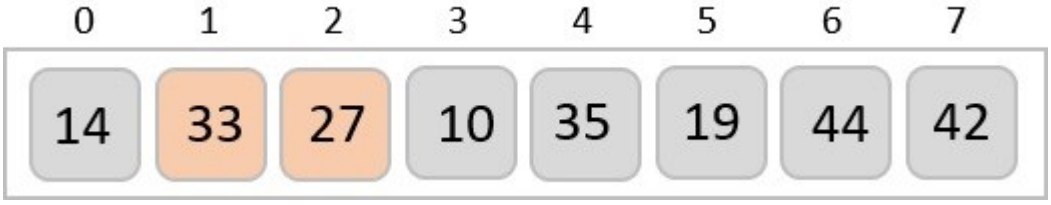
Insertion sort compares the first two elements.



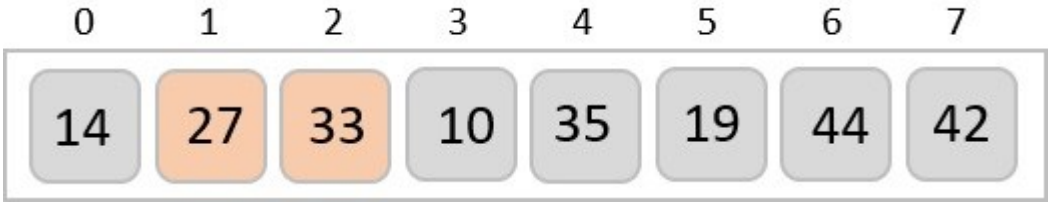
It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.



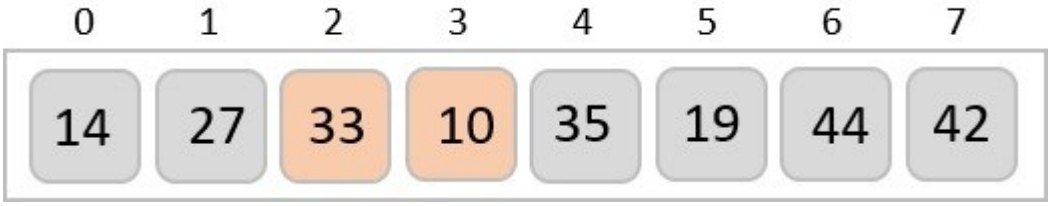
Insertion sort moves ahead and compares 33 with 27.



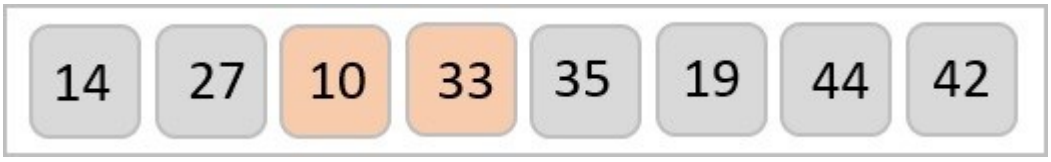
And finds that 33 is not in the correct position. It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.



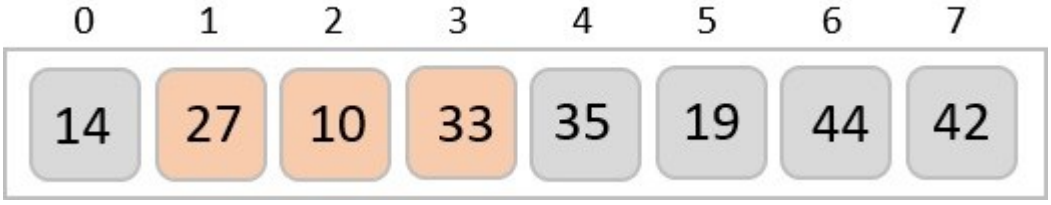
By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10. These values are not in a sorted order.



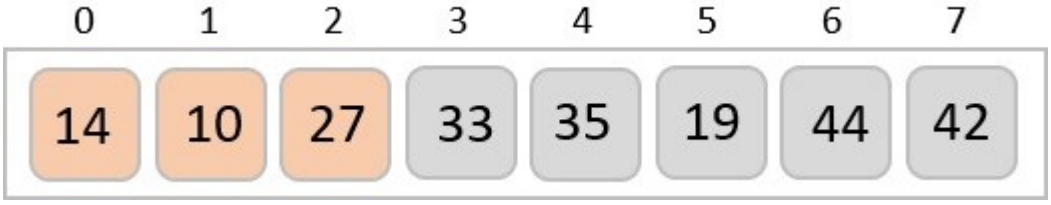
So they are swapped.



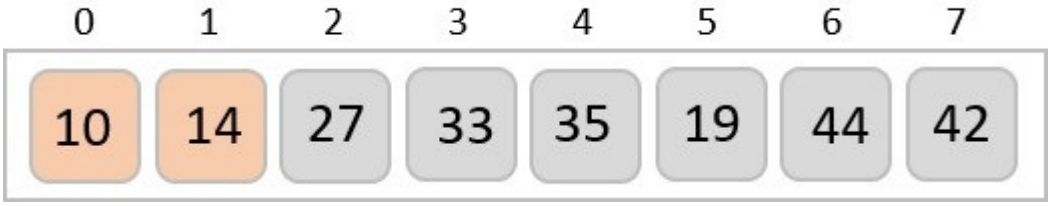
However, swapping makes 27 and 10 unsorted.



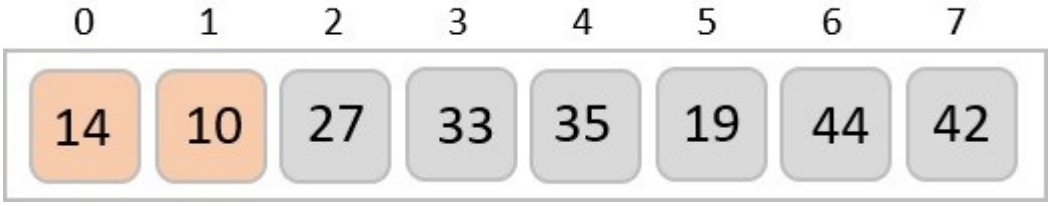
Hence, we swap them too.



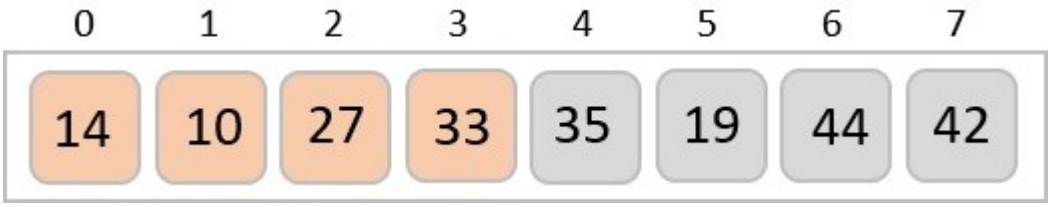
Again we find 14 and 10 in an unsorted order.



We swap them again.



By the end of third iteration, we have a sorted sub-list of 4 items.



This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort.

Python code:

```
def insertion_sort(array, size):
  for i in range(1, size):
    key = array[i]
    j = i
    while (j > 0) and (array[j-1] > key):
      array[j] = array[j-1]
      j = j-1
    array[j] = key

arr = [67, 44, 82, 17, 20]
n = len(arr)
print("Array before Sorting: ")
print(arr)
insertion_sort(arr, n);
print("Array after Sorting: ")
print(arr)
```

output:

Array before Sorting:

[67, 44, 82, 17, 20]

Array after Sorting:

[17, 20, 44, 67, 82]

Selection sort

Selection sort is a simple sorting algorithm. This sorting algorithm, like insertion sort, is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundaries by one element to the right.

This algorithm is not suitable for large data sets as its average and worst case complexities are of $O(n^2)$, where n is the number of items.

Selection Sort Algorithm

This type of sorting is called Selection Sort as it works by repeatedly sorting elements. That is: we first find the smallest value in the array and exchange it with the element in the first position, then find the second smallest element and exchange it with the element in the second position, and we continue the process in this way until the entire array is sorted.

1. Set MIN to location 0.

2. Search the minimum element in the list.

3. Swap with value at location MIN.

4. Increment MIN to point to next element.

5. Repeat until the list is sorted.

Pseudocode

Algorithm: Selection-Sort (A)

```
fori← 1 to n-1 do
  min j ←i;
  min x ← A[i]
  for j ←i + 1 to n do
    if A[j] < min x then
      min j ← j
      min x ← A[j]
  A[min j] ← A [i]
  A[i] ← min x
```

Analysis

Selection sort is among the simplest of sorting techniques and it works very well for small files. It has a quite important application as each item is actually moved at the most once.

Section sort is a method of choice for sorting files with very large objects (records) and small keys. The worst case occurs if the array is already sorted in a descending order and we want to sort them in an ascending order.

Nonetheless, the time required by selection sort algorithm is not very sensitive to the original order of the array to be sorted: the test if [] < *A[j]* < *min x* is executed exactly the same number of times in every case.

Selection sort spends most of its time trying to find the minimum element in the unsorted part of the array. It clearly shows the similarity between Selection sort and Bubble sort.

- Bubble sort selects the maximum remaining elements at each stage, but wastes some effort imparting some order to an unsorted part of the array.
- Selection sort is quadratic in both the worst and the average case, and requires no extra memory.

For each i from 1 to n - 1, there is one exchange and n - i comparisons, so there is a total of n - 1 exchanges and

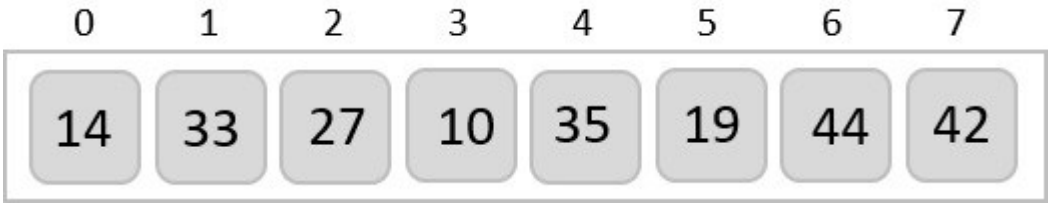
(n 1) + (n 2) + ...+2 + 1 = n(n 1)/2 comparisons.

These observations hold, no matter what the input data is.

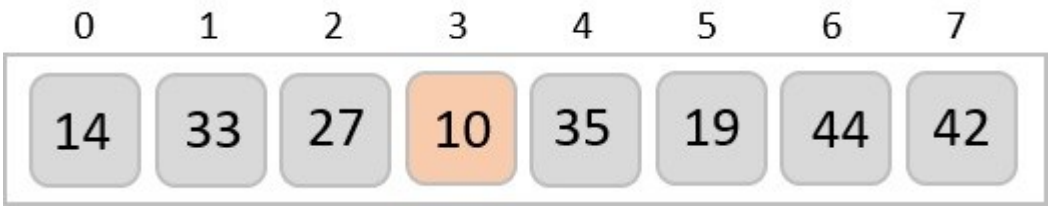
In the worst case, this could be quadratic, but in the average case, this quantity is O(n log n). It implies that the running time of Selection sort is quite insensitive to the input.

Example

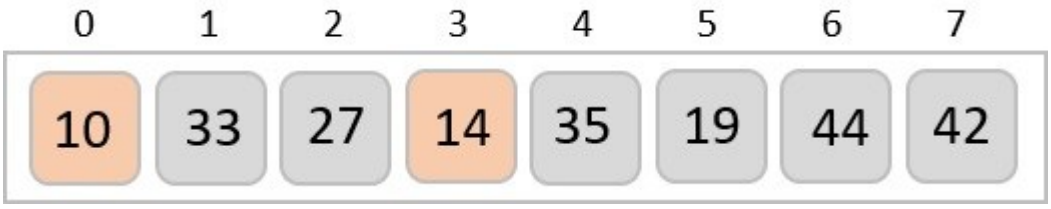
Consider the following depicted array as an example.



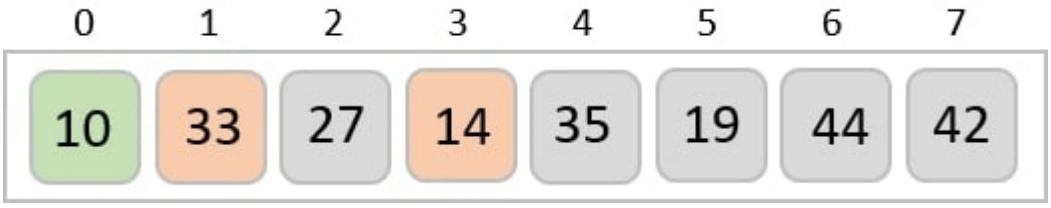
For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.



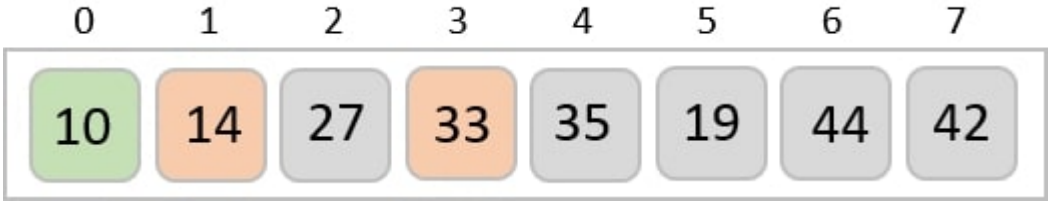
So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.



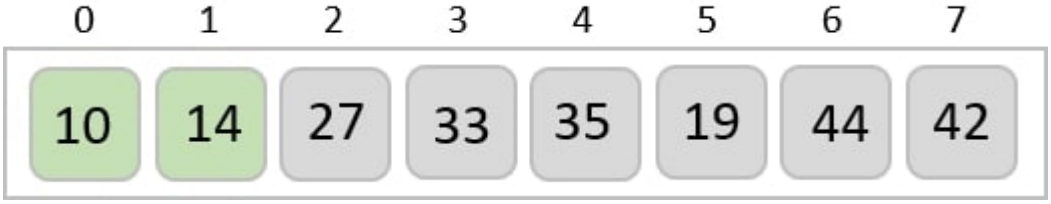
For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.



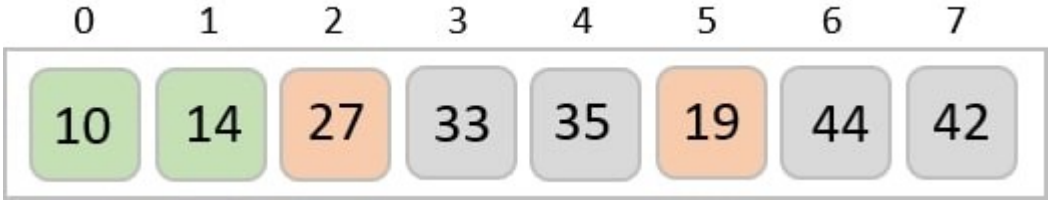
We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.

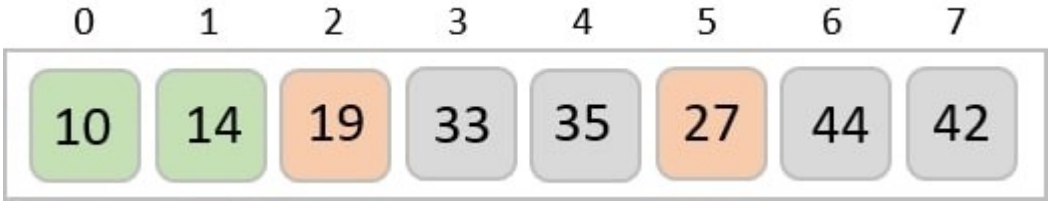


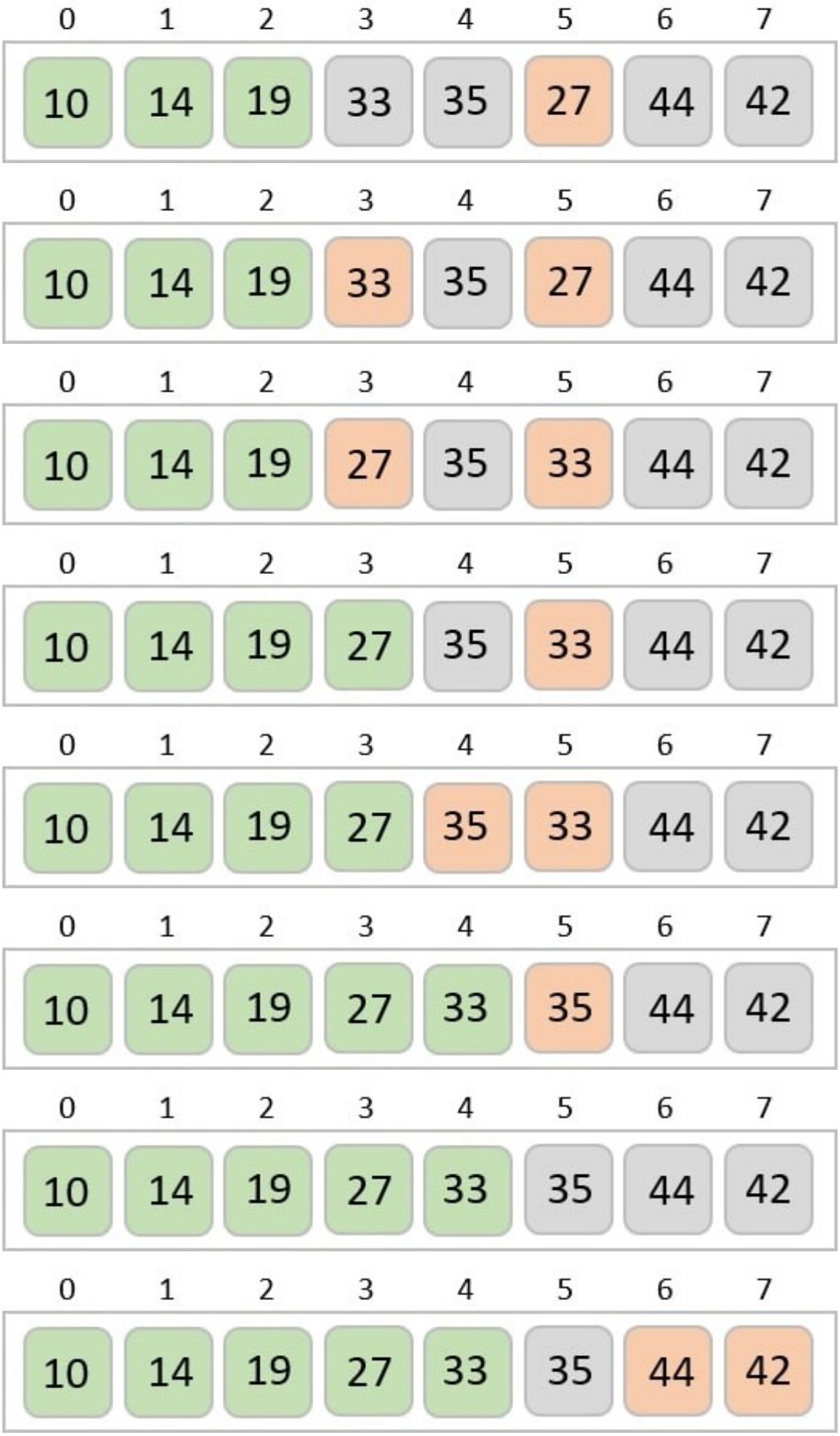
After two iterations, two least values are positioned at the beginning in a sorted manner.

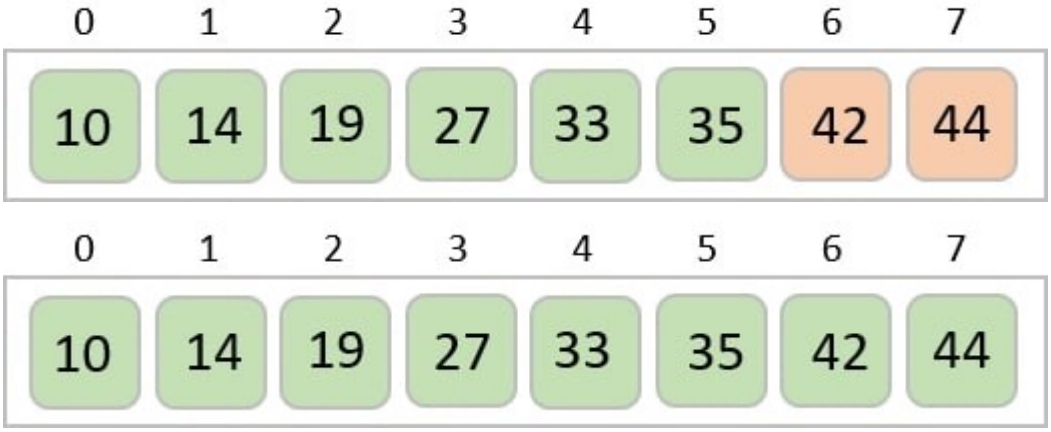


The same process is applied to the rest of the items in the array −






0 1 2 3 4 5 6 7
10 14 19 33 35 27 44 42
0 1 2 3 4 5 6 7
10 14 19 33 35 27 44 42
0 1 2 3 4 5 6 7
10 14 19 27 35 33 44 42
0 1 2 3 4 5 6 7
10 14 19 27 35 33 44 42
0 1 2 3 4 5 6 7
10 14 19 27 35 33 44 42
0 1 2 3 4 5 6 7
10 14 19 27 33 35 44 42
0 1 2 3 4 5 6 7
10 14 19 27 33 35 44 42
0 1 2 3 4 5 6 7
10 14 19 27 33 35 44 42



Implementation

The selection sort algorithm is implemented in four different programming languages below. The given program selects the minimum number of the array and swaps it with the element in the first index. The second minimum number is swapped with the element present in the second index. The process goes on until the end of the array is reached.

```
def insertion_sort(array, size):
   for i in range(size):
      imin = i
      for j in range(i+1, size):
         if arr[j] < arr[imin]:
            imin = j
      temp = array[i];
      array[i] = array[imin];
      array[imin] = temp;

arr = [12, 19, 55, 2, 16]
n = len(arr)
print("Array before Sorting: ")
print(arr)
insertion_sort(arr, n);
print("Array after Sorting: ")
print(arr)
```

output:

Array before Sorting:

[12, 19, 55, 2, 16]

Array after Sorting:

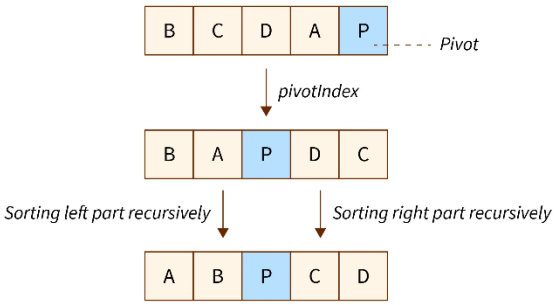
[2, 12, 16, 19, 55]

Quick Sort

Quick sort is one of the most efficient sorting algorithms and is based on the **divide and conquer algorithm**. An element is picked by this sorting algorithm, generally known as **pivot**, and an array is partitioned into two sub-arrays one sub-array is less than the pivot element and the other sub-array is greater than the pivot element.

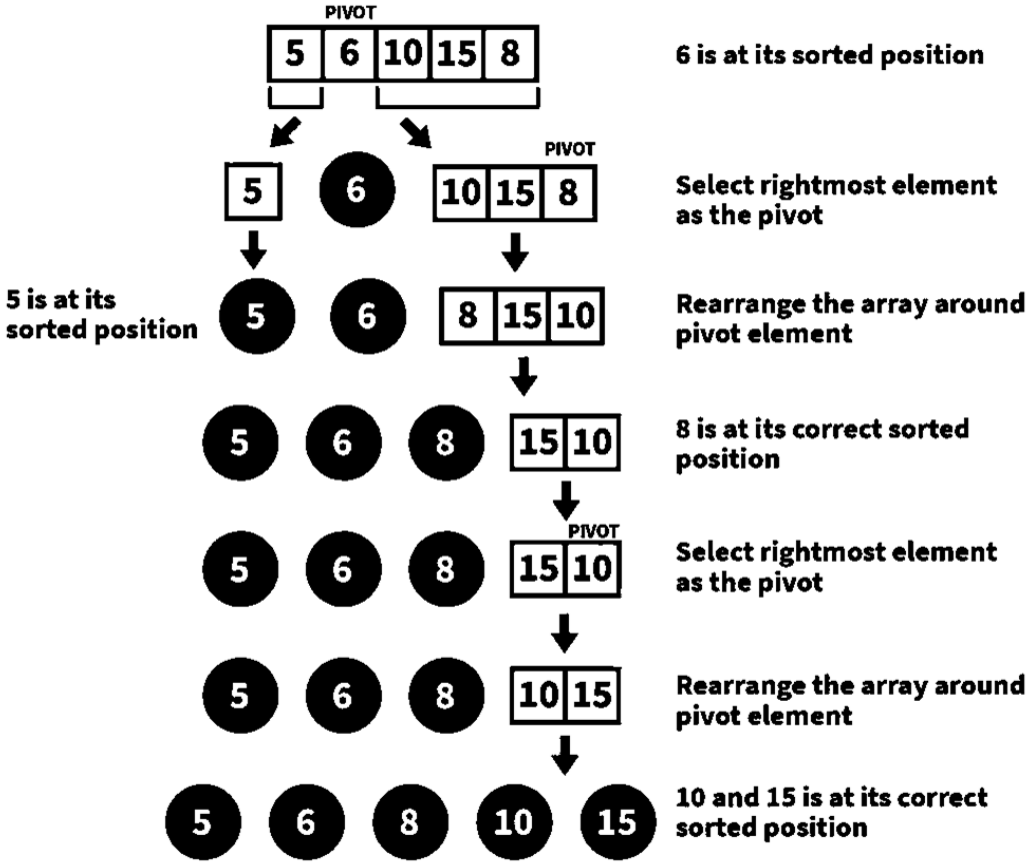
In order to sort the dataset as indicated below, the same procedure is repeated for sub-arrays:

what is an internal sorting algorithm (Quick Sort):



Working of Quick Sort Algorithm

Recall the list/array that had the elements – $10, 8, 5, 15, 6$ in it. To sort these elements in ascending order using quick sort, let us follow the under-given steps –



Python code:

```
# Quick sort in Python

# finding the partition point & rearranging the array
def partition(array, low, high):

  # selecting the rightmost element as pivot
  pivot = array[high];

  # setting the left pointer to point at the lowest index initially
  left = low;

  #setting the left pointer to point at the lowest index initially
  right = high - 1;

  #running a loop till left is smaller than right
  while (left <= right):
    #incrementing the value of left until the value at left'th
    # index is smaller than pivot
    while (array[left] < pivot):
      left = left + 1

    #decrementing the value of right until the value at right'th
    #index is greater than pivot
    while (array[right] > pivot):
      right = right - 1;

    if (left < right):
```

```
        #swapping the elements at left & right index
        array[left], array[right] = array[right], array[left]

    #swapping pivot with the element where left and right meet
    array[left], array[high] = array[high], array[left]

    # return the partition point
    return left

# function to perform quicksort
def quickSort(array, low, high):
 if low < high:

    # since this function returns the point where the array is
    #partitioned, it is used to track the subarrays/partitions in the
    #array
    pi = partition(array, low, high)

    # recursively calling the function on left subarray
    quickSort(array, low, pi - 1)

    # recursively calling the function on right subarray
    quickSort(array, pi + 1, high)
```

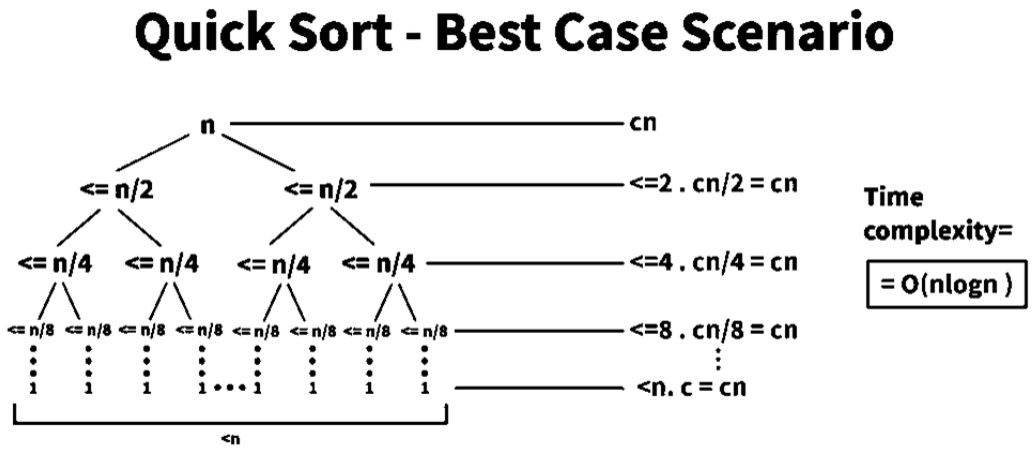
Complexity Analysis of Quick Sort

1. Quick Sort Time Complexity

Best-Case:

The best-case occurs when the pivot is almost in the middle of the array, and the partitioning is done in the middle. So, let us assume that we have a total of n elements in the array, and we are dividing the array from the middle.

In this case, with each partition, we will have at most n/2 elements. And we have to perform the partition until only one element is left in each subarray. The following is the tree representation of the given condition

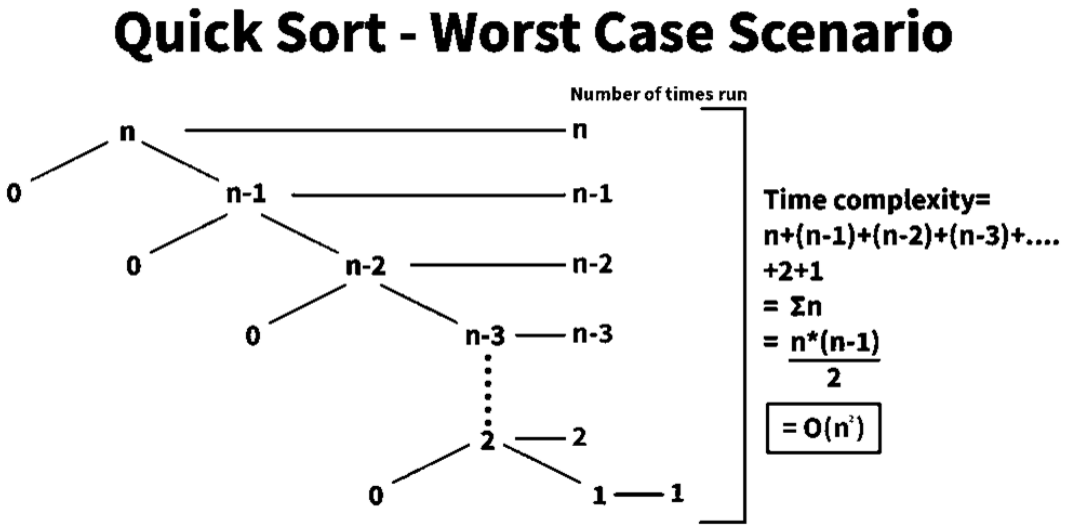


Notice that the above-given tree is a binary tree. And for a binary tree, we can know that its height is equal to logn. Therefore, the complexity of this part is O(logn).

Also, the time complexity of the partition() function would be equal to O(n). This is because, under the function, we are iterating over the array to swap rearranging the array around the pivot element.

Therefore, from the above results, we can conclude that the best-case time complexity for the quick sort algorithm is O(nlogn).

Worst Case:
The Worst-case occurs when either of the two partitions is unbalanced. This generally happens when the greatest or smallest element is selected as the pivot.
In this case, the pivot lies at the extreme of the array, and one subarray is always empty while the other contains n – 1 elements.
This way, in the first call, the array is divided into two subarrays of 0 & n – 1 elements respectively. For the second call, the array with n – 1 elements is divided into two subarrays of 0 & n – 2 elements respectively. This continues till only one element is left.
The time complexity, in this case, would be the sum of all the complexities at each step.
T(quicksort) = T(n) + T(n – 1) + T(n – 2) + …….. + 2 + 1
T(quicksort) = O((n * (n – 1))/2)
T(quicksort) = &O(n ^ 2)&
The following is the tree representation of this condition –



2. Quick Sort Space Complexity
In quick sort, the time complexity is calculated on the basis of space used by the recursion stack. In the worst case, the space complexity is O(n) because in the worst case, n recursive calls are made. And, the average space complexity of a quick sort algorithm is equal to O(logn).

Heap Sort

Heap Sort is an efficient sorting technique based on the heap data structure.
The heap is a nearly-complete binary tree where the parent node could either be minimum or maximum. The heap with minimum root node is called min-heap and the root node with maximum root node is called max-heap. The elements in the input data of the heap sort algorithm are processed using these two methods.
The heap sort algorithm follows two main operations in this procedure −

- Builds a heap H from the input data using the heapify (explained further into the chapter) method, based on the way of sorting ascending order or descending order.
- Deletes the root element of the root element and repeats until all the input elements are processed.

The heap sort algorithm heavily depends upon the heapify method of the binary tree. So what is this heapify method?
Heapify Method

The *heapify* method of a binary tree is to convert the tree into a heap data structure. This method uses recursion approach to heapify all the nodes of the binary tree.

Note − The binary tree must always be a complete binary tree as it must have two children nodes always.

The complete binary tree will be converted into either a max-heap or a min-heap by applying the *heapify* method.

To know more about the heapify algorithm, please click here.

Heap Sort Algorithm

As described in the algorithm below, the sorting algorithm first constructs the heap ADT by calling the Build-Max-Heap algorithm and removes the root element to swap it with the minimum valued node at the leaf. Then the heapify method is applied to rearrange the elements accordingly.

Algorithm:

```
Heapsort(A)
BUILD-MAX-HEAP(A)
for i = A.length downto 2
exchange A[1] with A[i]
A.heap-size = A.heap-size - 1
MAX-HEAPIFY(A, 1)
```

Analysis

The heap sort algorithm is the combination of two other sorting algorithms: insertion sort and merge sort.

The similarities with insertion sort include that only a constant number of array elements are stored outside the input array at any time.

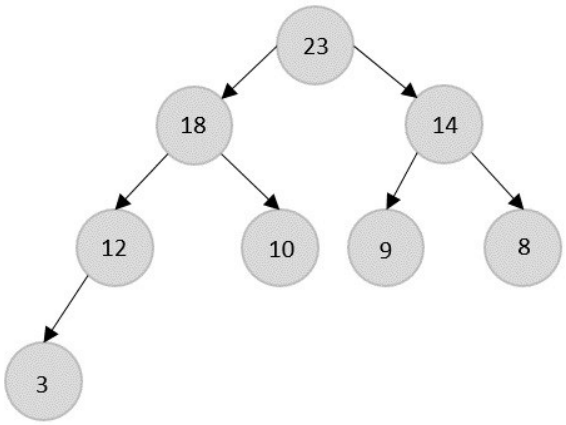
The time complexity of the heap sort algorithm is O(nlogn), similar to merge sort.

Example

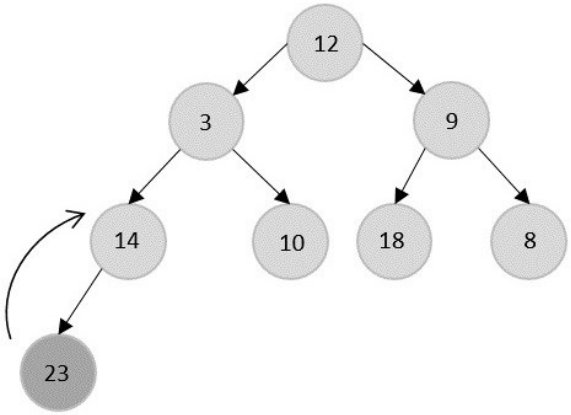
Let us look at an example array to understand the sort algorithm better −

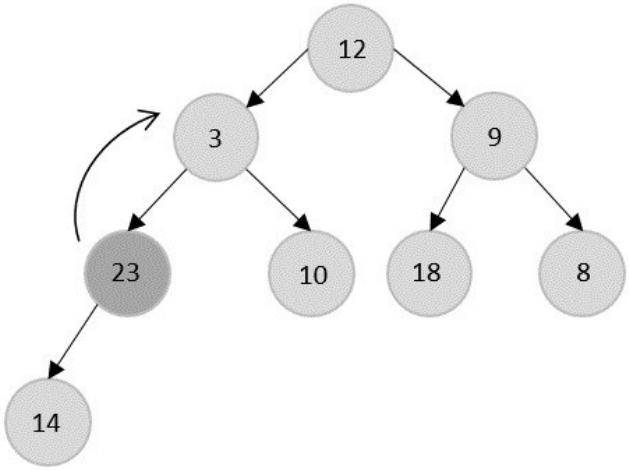
12	3	9	14	10	18	8	23

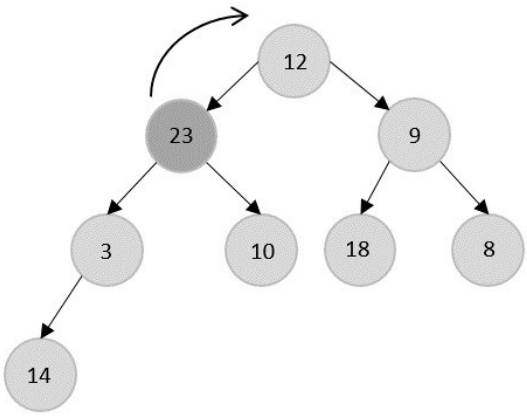
Building a heap using the BUILD-MAX-HEAP algorithm from the input array −

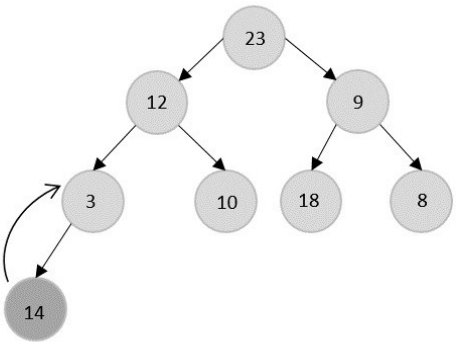


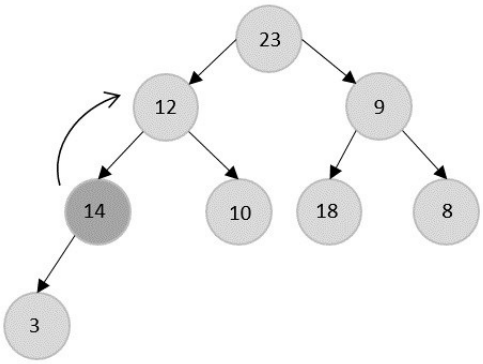
Rearrange the obtained binary tree by exchanging the nodes such that a heap data structure is formed.

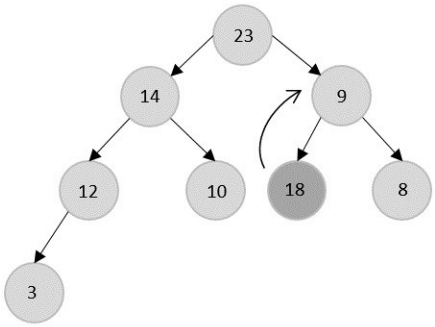

12
3
9
14
10
18
8
23

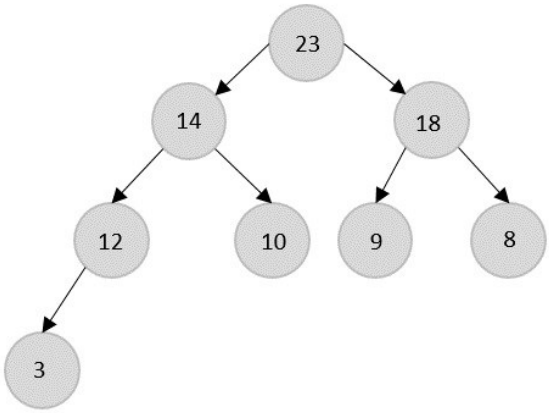

12
3
9
23
10
18
8
14


12
23
9
3
10
18
8
14


23
12
9
3
10
18
8
14



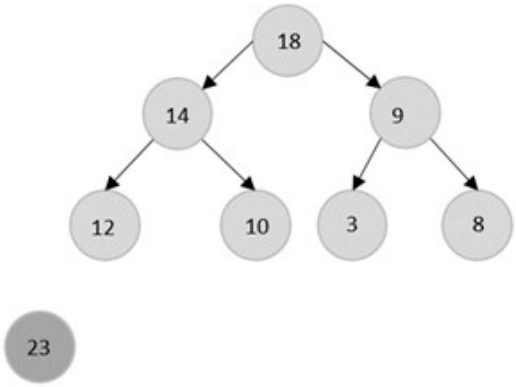




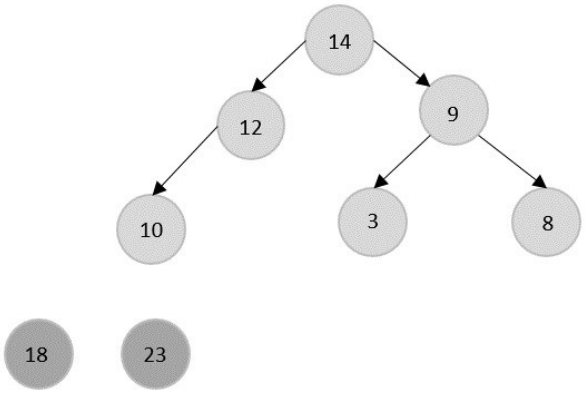
The Heapify Algorithm

Applying the heapify method, remove the root node from the heap and replace it with the next immediate maximum valued child of the root.

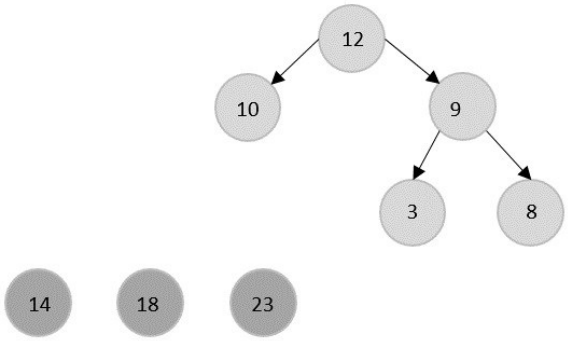
The root node is 23, so 23 is popped and 18 is made the next root because it is the next maximum node in the heap.



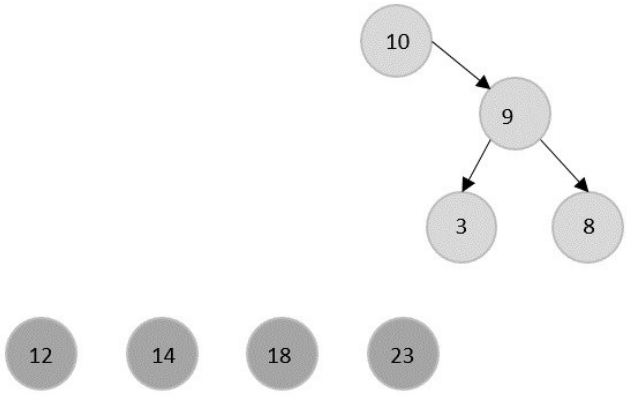
Now, 18 is popped after 23 which is replaced by 14.



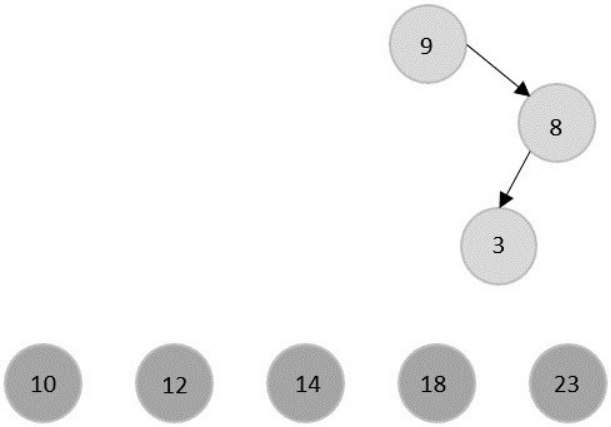
The current root 14 is popped from the heap and is replaced by 12.



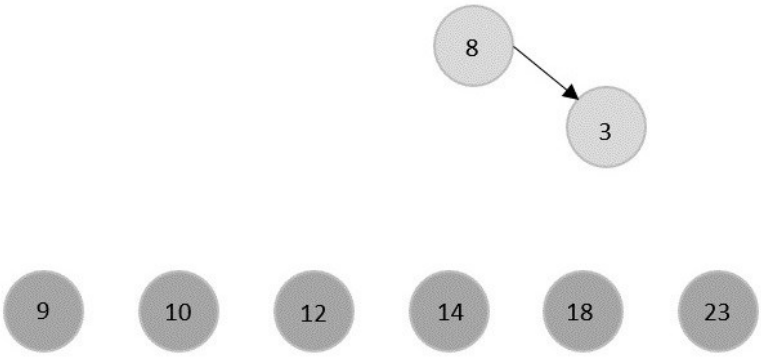
12 is popped and replaced with 10.



Similarly all the other elements are popped using the same process.

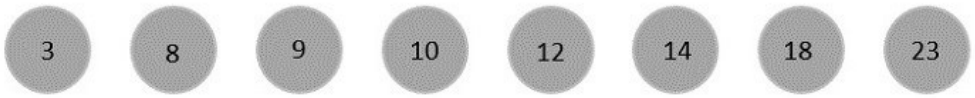


Here the current root element 9 is popped and the elements 8 and 3 are remained in the tree.



Then, 8 will be popped leaving 3 in the tree.

After completing the heap sort operation on the given heap, the sorted elements are displayed as shown below −



Every time an element is popped, it is added at the beginning of the output array since the heap data structure formed is a max-heap. But if the heapify method converts the binary tree to the min-heap, add the popped elements are on the end of the output array.
The final sorted list is,

3	8	9	10	12	14	18	23

Implementation

The logic applied on the implementation of the heap sort is: firstly, the heap data structure is built based on the max-heap property where the parent nodes must have greater values than the child nodes. Then the root node is popped from the heap and the next maximum node on the heap is shifted to the root. The process is continued iteratively until the heap is empty**.**

```
def heapify(heap, n, i):
  maximum = i
  l = 2 * i + 1
  r = 2 * i + 2
  # if left child exists
  if l < n and heap[i] < heap[l]:
    maximum = l
  # if right child exits
  if r < n and heap[maximum] < heap[r]:
    maximum = r
  # root
  if maximum != i:
    heap[i],heap[maximum] = heap[maximum],heap[i] # swap root.
    heapify(heap, n, maximum)
def heapSort(heap):
  n = len(heap)
  # maxheap
  for i in range(n, -1, -1):
    heapify(heap, n, i)
```

```
  # element extraction
  for i in range(n-1, 0, -1):
    heap[i], heap[0] = heap[0], heap[i] # swap
    heapify(heap, i, 0)
# main
heap = [4, 3, 1, 0, 2]
heapSort(heap)
n = len(heap)
print("Heap array: ")
print(heap)
print ("The Sorted array is: ")
print(heap)
```

output:

Heap array:
[0, 1, 2, 3, 4]
The Sorted array is:
[0, 1, 2, 3, 4]

Difference Between Internal and External Sort

Internal Sort	External Sort
Internal sorting is used when the input data can be adjusted in the main memory all at once.	External sorting is used when the input data cannot be adjusted in the main memory all at once.
The data to be sorted should be small enough to fit in the main memory.	It is used when data to be sorted cannot fit into the main memory all at once.
The storage device used in this method is only main memory (RAM)	Both Secondary memory (Hard Disk) and Main memory are used in this method.
While sorting is in progress, all the data to be sorted is stored in the main memory at all times.	While sorting, data is loaded into the Main memory in small chunks, all data is stored outside the memory like on Hard Disk.
Algorithms Used for Internal Sort are Bubble sort, Insertion Sort, Quick sort, Heap sort, etc.	Algorithms Used for External Sort are Merge sort, Tape Sort, External radix sort, etc.

- what is an internal sorting algorithm: Any sorting algorithm that uses the main memory exclusively during the sort is known as an internal sorting algorithm.
- The internal sorting technique is used when data items are less than enough to be held in the main memory (RAM) and the data can be accessed randomly.
- Bubble sort works by comparing adjacent elements, and repeatedly swapping them until all elements are in the correct order.
- Quick sort is one of the most efficient sorting algorithms and is based on the divide and conquer approach.
- While sorting is in progress, all the data to be sorted is stored in the main memory at all times in the internal sort whereas data is loaded in small chunks in the main memory in the external sort.

External Sorting:

External sorting is used for massive datasets that are too large to fit entirely into the main memory. Instead, the data is stored in slower **external memory** (like hard drives or SSDs), and only small, manageable chunks are brought into main memory at a time for processing.

- **Characteristics:**
    - **Data Handling:** Manages huge volumes of data by reading and writing to disk in blocks.
    - **I/O Operations:** The primary goal of external sorting algorithms is to minimize the number of input/output (I/O) operations, as disk access is significantly slower than memory access.
    - **Use Cases:** Necessary for large-scale data processing, such as sorting large database files or payroll information.
- **Examples of Algorithms:**
    - **External Merge Sort:** This is the most common external sorting method. It sorts small, manageable "runs" (chunks) of data in main memory, writes them to disk, and then merges the sorted runs into a single, fully sorted file.
    - K-way External Merge Sort
    - Buffer Management

2. Implementation Of External Sorting:

2.1 K Way Merge

K-Way Merge Sort is an **external sorting technique** used to sort very large data files that cannot fit into the main memory (RAM).
It merges **K sorted lists/files at the same time** instead of merging only two lists like in normal merge sort.

Need for K-Way Merge Sort

• Large data cannot be fully loaded into RAM
• Normal sorting becomes slow for huge data
• Reduces the number of merge passes
• Efficient for database and big data applications

Working Principle

It works in two phases:

Phase 1: Run Generation

• Data is divided into small blocks
• Each block is sorted in memory
• Stored as sorted runs on disk

Phase 2: K-Way Merging

• Open K sorted runs
• Compare first elements of all runs

- Select the smallest element
- Repeat until all elements are merged

Example

Let:
R1 = [4, 9, 15, 30]
R2 = [1, 8, 20, 25]
R3 = [3, 7, 12, 40]
After merging:
Final Output =
[1, 3, 4, 7, 8, 9, 12, 15, 20, 25, 30, 40]

Solutions:

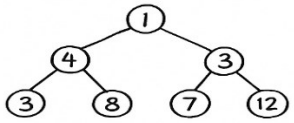
Run	**Elements**
R1	4, 9, 15, 30
R2	1, 8, 20, 25
R3	3, 7, 12, 40

Step	**Elements Compared**	**Smallest Selected**	**Output So Far**	**Remaining in Runs**
1	4, 1, 3	1	[1]	R1:[4,9,15,30], R2:[8,20,25], R3:[3,7,12,40]
2	4, 8, 3	3	[1,3]	R1:[4,9,15,30], R2:[8,20,25], R3:[7,12,40]
3	4, 8, 7	4	[1,3,4]	R1:[9,15,30], R2:[8,20,25], R3:[7,12,40]
4	9, 8, 7	7	[1,3,4,7]	R1:[9,15,30], R2:[8,20,25], R3:[12,40]
5	9, 8, 12	8	[1,3,4,7,8]	R1:[9,15,30], R2:[20,25], R3:[12,40]
6	9, 20, 12	9	[1,3,4,7,8,9]	R1:[15,30], R2:[20,25], R3:[12,40]
7	15, 20, 12	12	[1,3,4,7,8,9,12]	R1:[15,30], R2:[20,25], R3:[40]
8	15, 20, 40	15	[1,3,4,7,8,9,12,15]	R1:[30], R2:[20,25], R3:[40]
9	30, 20, 40	20	[1,3,4,7,8,9,12,15,20]	R1:[30], R2:[25], R3:[40]
10	30, 25, 40	25	[1,3,4,7,8,9,12,15,20,25]	R1:[30], R2:[], R3:[40]
11	30, -, 40	30	[1,3,4,7,8,9,12,15,20,25,30]	R1:[], R2:[], R3:[40]
12	-, -, 40	40	[1,3,4,7,8,9,12,15,20,25,30,40]	R1:[], R2:[], R3:[]

K-WAY MERGE SORT - Example

Initial Runs

R1 = [4, 9, 15, 30]
R2 = [1, 8, 20, 25]
R3 = [3, 7, 12, 40]



Merging Steps (Single Diagram)

Step	Heap (min-heap)	Extracted	Output So Far
0	1	1 (R2)	1
1	3 4	4 (R1)	3
2	7 6 7 12	8 (R3)	7
3	6	8 (R5)	12
4	9 8	9 (R2)	12
5	12 3 10 20	12 (R3)	15
6	7	15 (R1)	16
7	12 15	19 (R2)	20
8	15 2 10 20	23 (R3)	28
9	8	30 (R3)	30
10	3 12	32 (R2)	40

Final Output

1, 3, 4, 7, 7, 8, 9, 12, 15, 20, 25, 30, 25, 30, 40

Algorithm (Write in exams like this)

Algorithm: K-WAY-MERGE

Step 1: Create a min-heap H
Step 2: Insert first element of each list into H
Step 3: Repeat
    a) Remove the smallest element from H
    b) Add it to the output list
    c) Insert next element from the same list into H
Step 4: Stop when heap is empty

Python Code:

```
import heapq

def k_way_merge(lists):
    heap = []
    result = []

    for i in range(len(lists)):
        if len(lists[i]) > 0:
            heapq.heappush(heap, (lists[i][0], i, 0))

    while heap:
        value, list_i, ele_i = heapq.heappop(heap)
        result.append(value)
```

```
        if ele_i + 1 < len(lists[list_i]):
            next_val = lists[list_i][ele_i + 1]
            heapq.heappush(heap, (next_val, list_i, ele_i + 1))
    return result
lists = [
    [4, 9, 15, 30],
    [1, 8, 20, 25],
    [3, 7, 12, 40]
]
print(k_way_merge(lists))
```

Time Complexity

Let
N = Total number of elements
K = Number of lists
Time Complexity:
O(N log K)

2.2 Buffer Management

External Sorting is used when the data to be sorted **cannot fit entirely into main memory (RAM)** and resides on secondary storage (like hard disks). A common algorithm for external sorting is **External Merge Sort**.

Buffer management is crucial in external sorting because:

1. **Minimize Disk I/O:** Reading/writing from/to disk is slow. Using buffers reduces the number of I/O operations by reading/writing blocks of records at once.
2. **Efficient Memory Usage:** Memory is limited. Buffers help manage available RAM by storing only portions of data.
3. **Smooth Merging:** During k-way merge, each input stream can have a buffer to hold chunks of sorted data to avoid frequent disk reads.

Analogy: Imagine you're merging multiple huge books into one sorted book, but you can only hold a few pages in your hand at a time. Buffers are your "handful of pages."

External Merge Sort has **two phases**:

Phase 1: Run Generation

- Divide the large dataset into smaller chunks that fit into RAM.
- Sort each chunk in memory.
- Write each sorted chunk (called a **run**) to disk.

Phase 2: Multi-way Merge

- Merge all sorted runs using a **buffer for each run**.
- Only a small part of each run is loaded into memory.
- Continuously write merged output to disk using an **output buffer**.

Buffer management strategy:

- Input buffer per run (size depends on memory)
- Output buffer to accumulate merged records
- Flush output buffer to disk when full

Algorithm:

1. Read buffer size B
2. Read number of page requests N
3. Read the list of page requests
4. Initialize an empty buffer (queue)

5. For each page in page requests:
    a. If page is present in buffer:
        Print "HIT"
    b. Else:
        Print "MISS"
        If buffer is full:
            Remove the oldest page (FIFO)
        Insert the new page into buffer

6. End

Example:

Buffer Management in External Sorting (FIFO Policy)

You are given a very large file divided into pages. Since the entire file cannot fit into main memory, a buffer of limited size is used to load pages from secondary storage.

For each page request:

- If the page is already present in the buffer, it is a **HIT**
- If the page is not present, it is a **MISS** and the page must be loaded into the buffer
- When the buffer is full, the **FIFO (First In First Out)** replacement policy is used to remove a page

Write a program to simulate buffer management and print **HIT** or **MISS** for each page request.

Problem:

Input Format

- First line: Integer B – buffer size
- Second line: Integer N – number of page requests
- Third line: N space-separated integers representing page requests

Output Format

- For each page request, print **HIT** or **MISS** on a new line

Constraints

- $1 \leq B \leq 10$
- $1 \leq N \leq 100$
- $0 \leq$ Page Number $\leq 10^5$

Example 1

4

10

1 2 3 4 1 5 2 6 3 4

Solution :

Request	Buffer State	Result
1	[1]	MISS
2	[1,2]	MISS
3	[1,2,3]	MISS
4	[1,2,3,4]	MISS
1	[1,2,3,4]	HIT
5	[2,3,4,5]	MISS
2	[2,3,4,5]	HIT
6	[3,4,5,6]	MISS
3	[3,4,5,6]	HIT
4	[4,5,6,4]	MISS

Python Code:

```
from collections import deque
def buffer_management():
    B = int(input())     # Buffer size
    N = int(input())     # Number of page requests
    pages = list(map(int, input().split()))

    buffer = deque()

    for page in pages:
        if page in buffer:
            print("HIT")
        else:
            print("MISS")
            if len(buffer) == B:
                buffer.popleft()   # FIFO removal
            buffer.append(page)

# Run the function
buffer_management()
```

Explanation

- The buffer simulates **main memory**
- Pages are checked for presence:
  - Found → **HIT**
  - Not found → **MISS**
- When buffer is full, the **oldest page** is removed
- FIFO ensures simplicity and fairness

Time Complexity

- Page lookup: **O(B)**
- Total complexity: **O(N × B)**
- Since buffer size is small, this is efficient

Applications

- Database buffer pools
- Operating systems page replacement
- External merge sort I/O optimization

Example 2

Buffer Size = 6
Number of Page Requests = 20
Page Request Sequence
1 2 3 4 5 6 1 2 7 8 3 4 9 1 2 10 5 6 7 8

Solution:

Step	Page	HIT / MISS	Buffer Content (Old → New)
1	1	MISS	1
2	2	MISS	1 2
3	3	MISS	1 2 3
4	4	MISS	1 2 3 4
5	5	MISS	1 2 3 4 5
6	6	MISS	1 2 3 4 5 6
7	1	HIT	1 2 3 4 5 6
8	2	HIT	1 2 3 4 5 6
9	7	MISS	2 3 4 5 6 7
10	8	MISS	3 4 5 6 7 8
11	3	HIT	3 4 5 6 7 8
12	4	HIT	3 4 5 6 7 8
13	9	MISS	4 5 6 7 8 9
14	1	MISS	5 6 7 8 9 1
15	2	MISS	6 7 8 9 1 2
16	10	MISS	7 8 9 1 2 10
17	5	MISS	8 9 1 2 10 5
18	6	MISS	9 1 2 10 5 6

19	7	MISS	1 2 10 5 6 7
20	8	MISS	2 10 5 6 7 8

- **Total Requests** = 20
- **HITS** = 4
- **MISSES** = 16
- **Replacement Policy** = FIFO

3.1 Static Vs Dynamic Hashing,

3.2 Hash Functions (Division, Mid Square, Folding, Multiplication, Digit Analysis)

3. Hashing Techniques:

Introduction:

When a dictionary with n entries is represented as a binary search tree, the dictionary operations search, insert, and delete take **O(n)** time. These dictionary operations may be performed in **O(log n)** time using a balanced binary search tree. In this chapter, we examine a technique called hashing, that enables us to perform the dictionary operations search, insert and delete in **O(1)** time expected.

Hashing:

Hashing is a technique that is used to uniquely identify a specific object from a group of similar objects. Some examples of how hashing is used in our lives include:

- In universities or colleges, each student is assigned a unique roll number that can be used to retrieve information about them.
- In libraries, each book is assigned a unique number that can be used to determine information about the book, such as its exact position in the library or the users it has been issued to etc.

In both these examples the students and books were hashed to a unique number.

Assume that you have an object and you want to assign a key to it to make searching easy. To store the key/value pair, you can use a simple array like a data structure where keys (integers) can be used directly as an index to store values.

In hashing, large keys are converted into small keys by using **hash functions**. The values are then stored in a data structure called **hash table**. The idea of hashing is to distribute entries (key/value pairs) uniformly across an array. Each element is assigned a key (converted key). By using that key you can access the element in **O(1)** time. Using the key, the algorithm (hash function) computes an index that suggests where an entry can be found or inserted.

Hashing is implemented in two steps:

1. **An element is converted into an integer by using a hash function. This element can be used as an index to store the original element, which falls into the hash table.**

2. **The element is stored in the hash table where it can be quickly retrieved using hashed key.**

There are many possibilities for representing the dictionary and one of the best methods for representing is hashing. Hashing is a type of a solution which can be used in almost all situations. Hashing is a technique which uses less key comparisons. This method generally used the hash functions to map the keys into a table, which is called a hash table.

Hashing is of two types.

1) Static Hashing and
2) Dynamic Hashing

Static Hashing

1) Hash Table

Hash table is a type of data structure which is used for storing and accessing data very quickly. Insertion of data in a table is based on a key value. Hence every entry in the hash table is defined with some key. By using this key data can be searched in the hash table by few key comparisons and then searching time is dependent upon the size of the hash table.

2) Hash Function

Hash function is a function which is applied on a key by which it produces an integer, which can be used as an address of hash table. Hence one can use the same hash function for accessing the data from the hash table. In this the integer returned by the hash function is called hash key.

Types of Hash Functions

There are various types of hash function which are used to place the data in a hash table,

1) **Division Method**

2) **Mid square Method**

3) **Digit folding Method**

4) **Digit Analysis Method/Binary/Radix Method**

1)Division Method

In this method the hash function is dependent upon the remainder of a division.

For example if the record **52, 68, 99, 84** is to be placed in a hash table and let us take the table size is 10.

Then:

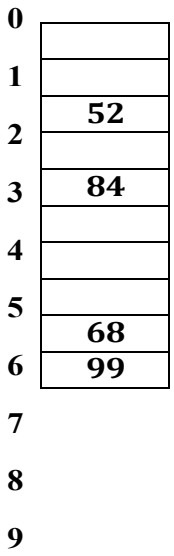
h(key)= key % table_size
h(52) = 52 % 10 = 2

h(68) = 68 % 10 = 8

h(99) = 99 % 10 = 9

h(84) = 84 % 10 = 4

Division Method



Hash Table

 # Program to insert elements into a hash table using Division Method

```
def hash_function(e, size):
    return e % size

# Input size of hash table
size = int(input("Enter size of hash table: "))

# Initialize hash table
hash_table = [-1] * size

# Insert elements
for i in range(size):
    element = int(input("Enter element to insert: "))
    index = hash_function(element, size)
    hash_table[index] = element   # Direct insertion (no collision handling)

# Display hash table
print("\nValues in Hash Table:")
for i in range(size):
    print(hash_table[i])
```

2. Mid Square Method

In this method firstly key is squared and then mid part of the result is taken as the index. For example: consider that if we want to place a record of **3101** and the size of table is 1000.

So, **3101*3101=9616201**

i.e. h (3101) = 162 (middle 3 digit).

k	k*k	Square	index h(k)
15	15 * 15	225	2
22	22 * 22	484	8
16	16 * 16	256	5
29	29 * 29	841	4
31	31 * 31	961	6
5	5 * 5	25	0
3	3 * 3	9	9

Mid Square Method

0	5
1	
2	15
3	
4	29
5	16
6	31
7	
8	22
9	3

Hash Table

If the square value contains even number of digits the index is 0. If it

contains odd value index is middle value.

If the square value is a single digit, the value will be placed in that index only. Value 3 is stored at location 9.

```
# Mid-Square Hashing Method

def mid_square_hash(key, table_size):
    square = key * key
    square_str = str(square)

    # Find middle index
    mid = len(square_str) // 2

    # Extract one or two middle digits
    if len(square_str) >= 2:
        middle_digits = square_str[mid - 1: mid + 1]
    else:
        middle_digits = square_str

    return int(middle_digits) % table_size


def main():
    size = int(input("Enter size of hash table: "))
    hash_table = [-1] * size

    n = int(input("Enter number of elements: "))

    for _ in range(n):
        key = int(input("Enter key: "))
        index = mid_square_hash(key, size)

        # Linear probing if collision occurs
        start = index
        while hash_table[index] != -1:
            index = (index + 1) % size
            if index == start:
                print("Hash table is FULL")
                return

        hash_table[index] = key

    print("\nValues in Hash Table:")
    for i in range(size):
        print(f"Index {i}: {hash_table[i]}")


# Run the program
main()
```

Sample Input

Enter size of hash table: 10
Enter number of elements: 5
Enter key: 23
Enter key: 45
Enter key: 12
Enter key: 78
Enter key: 39

Sample Output

Values in Hash Table:
Index 0: -1
Index 1: 12
Index 2: 23
Index 3: 78
Index 4: -1
Index 5: 45
Index 6: -1
Index 7: 39
Index 8: -1
Index 9: -1

3. Digit Folding Method

In this method, we partition the identifier k into several parts. All parts, except for the last one have the same length. We then add the parts together to obtain the hash address for k. There are two ways of carrying out this addition. In the first method, we shift all parts except for the last one, so that the least significant bit of each part lines up with the corresponding bit of the last part. We then add the parts together to obtain h(k). This method is known as **shift folding.**

The second method, known as **folding at the boundaries**, the key is folded at the partition boundaries, and digits falling into the same position are added together to obtain h(k). This is equivalent to reversing every other partition before adding.

Example 1: Suppose that k=12320324111220, and we partition it into parts that are 3 decimal digits long. The partitions are P1=123, P2=203, P3=241, P4=112 and P5=20

Shift Folding

$$h(k) = \sum_{i=1}^{5} P_i$$

= P1+P2+P3+P4+P5
= 123 + 203 + 241 + 112 + 20
= 699

Folding at the boundaries

When folding at boundaries is used, we first reverse P2 and P4 to obtain 302 and 211 respectively. Next the five partitions are added to obtain

$$h(k) = \sum_{i=1}^{5} P_i$$

$$= P1+P2+P3+P4+P5$$
$$= 123 + 302 + 241 + 211 + 20$$
$$= 897$$

Example 2) For example: consider a record of 12465512 then it will be divided into parts.
i.e. 124, 655, 12. After dividing the parts combine these parts by adding it.

Shift folding

H(key) =124+655+12 = 791
791 is the index to store the value 12465512

Folding at the boundaries

H(key) =124+556+12 = 692
692 is the index to store the value 12465512

```
def digit_folding_hash(key, size):
    key_str = str(key)
    sum_parts = 0

    # Split into parts of 2 digits
    for i in range(0, len(key_str), 2):
        part = key_str[i:i+2]
        sum_parts += int(part)

    return sum_parts % size


# -------- Main Program --------
size = int(input("Enter size of hash table: "))
hash_table = [-1] * size

for i in range(size):
    element = int(input("Enter element to insert: "))
    index = digit_folding_hash(element, size)
    hash_table[index] = element

print("\nValues in Hash Table:")
for i in range(size):
    print(hash_table[i])
```

Test Case – Digit Folding Method

Input

10
5
1234 5678 2345 9876 1111

Explanation

Key	Groups	Sum	Index
1234	12 + 34	46	6
5678	56 + 78	134	4
2345	23 + 45	68	8
9876	98 + 76	174	4
1111	11 + 11	22	2

Output

-1
-1
1111
-1
9876
-1
1234
-1
2345
-1

4) Digit Analysis Method:

In this method we will examine, digit analysis, is used with static files. A static file is one in which all the identifiers are known in advance.

Using this method, we first transform the identifiers into numbers using some radix, r.

We then examine the digits of each identifier, deleting those digits that have the most skewed distributions. We continue deleting digits until the number of remaining digits is small enough to give an address in the range of the hash table. The digits used to calculate the hash address must be the same for all identifiers and must not have abnormally high peaks or valleys (the standard deviation must be small).
For Example store values 4,8,3,7 in the hash table considering the radix 2.

Consider the hash table having a size of 8 elements. The index is in binary. The no of digits in index is 3. So we are going to generate 3 digited index from our has function.

k	value in binary	h(k)
4	0100	100
8	1000	000
3	0011	011
7	0111	111

Digit Analysis Method

000	8
001	
010	
011	3
100	4
101	
110	
111	7

Hash Table

Example:

- **Key = 12345**
- **Select digits at positions [1, 3, 5] → 1, 3, 5**
- **Hash value = 135 % table_size**

This method is useful when certain digits are more uniformly distributed.

Python Code (Digit Analysis Method)

```
def digit_analysis_hash(key, positions, table_size):
    """
    key        : integer key to be hashed
    positions  : list of digit positions (1-based indexing)
    table_size : size of hash table
    """
    key_str = str(key)
    selected_digits = ""

    for pos in positions:
        if pos <= len(key_str):
            selected_digits += key_str[pos - 1]

    if selected_digits == "":
        return 0

    return int(selected_digits) % table_size


# -------- CodeChef Style Input --------
size = int(input())                    # Hash table size
positions = list(map(int, input().split()))  # Digit positions
n = int(input())                       # Number of keys

hash_table = [-1] * size

for _ in range(n):
    key = int(input())
    index = digit_analysis_hash(key, positions, size)

    # Linear probing for collision handling
    start = index
    while hash_table[index] != -1:
        index = (index + 1) % size
        if index == start:
            print("Hash table is FULL")
            break

    if hash_table[index] == -1:
        hash_table[index] = key

# -------- Output --------
for i in range(size):
    print(hash_table[i])
```

Input Format (Example)

10
1 3 5
5
12345
67891
54321
98765
13579

Explanation

- Hash table size = 10
- Digit positions chosen = 1st, 3rd, 5th
- Keys are hashed using selected digits
- Linear probing is used for collision resolution

Dry Run (Example)

Key	Selected Digits	Hash Value
12345	1,3,5 → 135	135 % 10 = 5
67891	6,8,1 → 681	681 % 10 = 1
54321	5,3,1 → 531	531 % 10 = 1 (collision → probe)
98765	9,7,5 → 975	975 % 10 = 5 (collision → probe)
13579	1,5,9 → 159	159 % 10 = 9

Output

-1

67891

54321

-1

-1

12345

98765

-1

-1

13579

Time Complexity

- Hash computation: **O(d)** where d = number of selected digits
- Insertion (with probing): **O(1)** average
- Worst case: **O(n)**

Difference Between Hashing Methods

Method	Definition	Formula	Example	Key Idea
Division Method	Computes hash address by taking the remainder when the key is divided by table size	h(k) = k mod m	If k = 123, m = 10 → 123 mod 10 = 3	Simple and fast, depends on table size
Mid-Square Method	Square the key and extract middle digits as hash value	h(k) = middle digits of k^2	k = 44 → 44^2 = 1936 → middle = 93	Reduces clustering caused by similar keys
Digit Folding Method	Divide key into parts and add them together	h(k) = sum of folded parts	k = 123456 → 12 + 34 + 56 = 102	Useful for large numeric keys
Digit Analysis Method	Select specific digits from key that vary most and combine them	No fixed formula	k = 987654 → choose digits 9,6,4 → 964	Best when digit patterns are known

Python Program for ALL 4 Methods Together

```
def division_method(key, size):
    return key % size
def mid_square_method(key, size):
    square = key * key
    square_str = str(square)
    mid = len(square_str) // 2
    return int(square_str[mid]) % size
def digit_folding_method(key, size):
    key_str = str(key)
    total = 0
    for i in range(0, len(key_str), 2):
        total += int(key_str[i:i+2])
    return total % size
def digit_analysis_method(key, size, positions):
    key_str = str(key)
    selected = ""
    for pos in positions:
        if pos < len(key_str):
            selected += key_str[pos]
    return int(selected) % size

# ---------- TESTING ----------
size = 10
key = 12345
print("Division Method:", division_method(key, size))
print("Mid Square Method:", mid_square_method(key, size))
print("Digit Folding Method:", digit_folding_method(key, size))
print("Digit Analysis Method:", digit_analysis_method(key, size, [2, 3]))
```

Sample Output

Division Method: 5

Mid Square Method: 9

Digit Folding Method: 1

Digit Analysis Method: 4

Advantages

- **Division Method** → Simple, fast
- **Mid-Square Method** → Better distribution
- **Digit Folding** → Handles large keys well
- **Digit Analysis** → Best when digit patterns are known

Characteristics of Good Hashing Function

1) The hash function should generate different hash values for the similar string.
2) The hash function is easy to understand and simple to compute.
3) The hash function should produce the keys which will get distributed, uniformly over an array.
4) A number of collisions should be less while placing the data in the hash table.
5) The hash function is a perfect hash function when it uses all the input data.

4. Overflow Handling (Collision)

Collision

It is a situation in which the hash function returns the same hash key for more than one record, it is called as collision. Sometimes when we are going to resolve the collision it may lead to a overflow condition and this overflow and collision condition makes the poor hash function.

Collision resolution technique

If there is a problem of collision occurs then it can be handled by apply some technique. These techniques are called as collision resolution techniques. There are generally four techniques which are described below.

1) **Chaining**
2) **Linear Probing (Open addressing)**
3) **Quadratic Probing (Open addressing) and**
4) **Double Hashing (Open addressing).**

1) Chaining

It is a method in which additional field with data i.e. chain is introduced. A chain is maintained at the home bucket. In this when a collision occurs then a linked list is maintained for colliding data.

For Example: Let us consider a hash table of size 10 and we apply a hash function of H(key)=key % size of table. Let us take the keys to be inserted are **31, 33, 77, 61**.

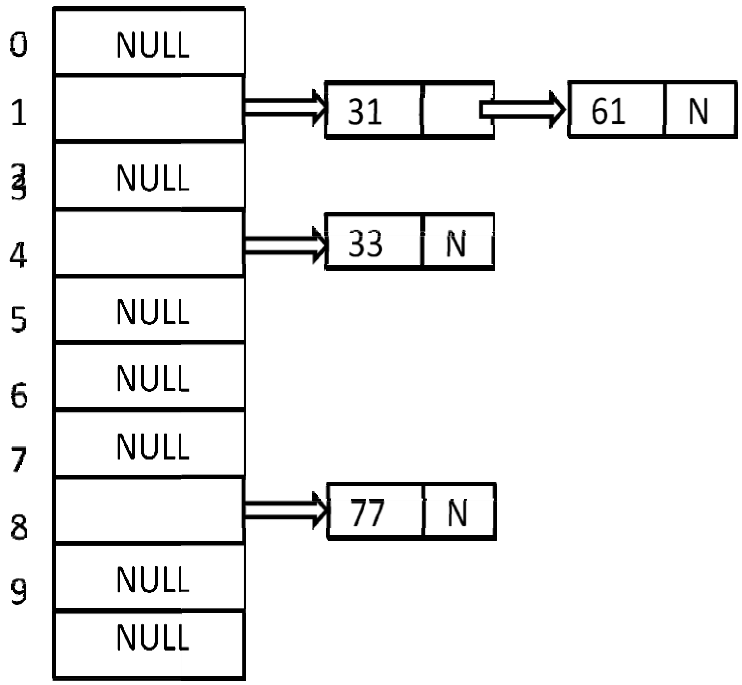
In the diagram we can see at same bucket 1 there are two records which are maintained by linked list or we can say by chaining method.

H(31) = 31%10 = 1

H(33) = 33%10 = 3

H(77) = 77%10 = 7

H(61) = 61%10 = 1



2) Linear probing (Open addressing)

It is very easy and simple method to resolve or to handle the collision. In this, collision can be solved by placing the second record linearly down, whenever the empty place is found. In this method there is a problem of clustering which means at some place block of a data is formed in a hash table.

Example: Let us consider a hash table of size 10 and hash function is defined as H(key)=key % table_size. Consider that following keys are to be inserted that are **56, 64, 36, 71**.

56 % 10 = 6

64 % 10 = 4

0	NULL
1	NULL
2	NULL
3	64
4	56
5	NULL
6	NULL
7	NULL
8	
9	

36 % 10 = 6

The index 6 is already filled with 56
It is not empty
Collision occurred
To resolve this check the next location
i.e. 6+1 = 7
index 7 is NULL so insert 36 at index 7.

0	NULL
1	NULL
2	NULL
3	64
4	NULL
5	56
6	36
7	NULL
8	NULL
9	

71 % 10 = 1

As the index 1 is null
We can insert 71 at index 1

0	
1	NULL
2	71
3	NULL
4	NULL
5	64
6	NULL
7	56
8	36
9	NULL
	NULL

In this diagram we can see that 56 and 36 need to be placed at same bucket but by linear probing technique the records linearly placed downward if place is empty i.e. it can be seen 36 is placed at index 7.

3) Quadratic Probing (Open addressing)

This is a method in which solving of clustering problem is done. In this method the hash function is defined by the

H(key)=(H(key)+x*x)%table_size

Let us consider we have to insert following elements that are:-
67, 90, 55, 17, 49.

67%10 = 7

90%10 = 0

55%10 = 5

17%10 = 7

49%10 = 9

0	90
1	
2	
3	
4	55
5	67
6	
7	
8	
9	

In this we can see if we insert 67, 90, and 55 it can be inserted easily but in the case of 17 hash function is used in such a manner that :-
To insert 17

The initial index generated is 17%10 = 7
But the index 7 is already filled with 67. Collision occurred.
To resolve this we try
(7+0*0)%10 = 7

(when x=0 it provide the index value 7 only) by making the increment in value of x. let x =1 so , **(7+1*1)%10 = 8.**
in this case bucket 8 is empty hence we will place 17 at index 8.

67%10 = 7

90%10 = 0

55%10 = 5

17%10 = 7

49%10 = 9

0	90
1	
2	
3	
4	55
5	67
6	17
7	49
8	
9	

4) Double hashing (Open addressing)

It is a technique in which two hash functions are used when there is an occurrence of collision. In this method 1 hash function is simple as same as division method. But for the second hash function there are two important rules which are

1. It must never evaluate to zero.
2. Must sure about the buckets, that they are probed.

The hash functions for this technique are:

H1(key) = key % table_size
H2(key) = P-(key mod P)

Where, **p** is a prime number which should be taken smaller than the size of a hash table.

Example: Let us consider we have to insert **67, 90,55,17,49**.

67%10 = 7

90%10 = 0

55%10 = 5

17%10 = 7

= 7 - (17%7)

= 7 - 3

= 4

49%10 = 9

0	90
	17
1	
2	
3	55
4	
	67
5	
	49
6	
7	
8	
9	

In this we can see 67, 90 and 55 can be inserted in a hash table by using first hash function but in case of 17 the bucket is full and in this case we have to use the second hash function which is

H2(key) = P - (key mod P)

where **P is a prime number which should be taken smaller than the hash table so value of P will be 7.**

i.e. **H2(17) = 7 - (17%7)**

= 7 - 3

= 4

that means we have to take **4 jumps for placing 17**. Therefore 17 will be placed at index 1.

Key density

The identifier density or key density of a hash table is the ratio n/T, where n is the number of identifiers in the table and T is the total number of possible keys.

Suppose our keys are at most six characters long, where a character may be a decimal digit or an upper case letter, and that the first character is a letter. Then the number of possible keys is $T = \sum_{0 \le i \le 5} 26 \times 36^i > 1.6 * 10^9$. So the key density n/T is usually very small.

Loading density

The loading density or loading factor of a hash table is

$$\alpha = n / (sb)$$

Where n is the number of identifiers in the table b is
number of buckets
s is number of slots per bucket

Example :

Consider the hash table ht with buckets **b = 26** and slots **s = 2**.

We have **n = 10** distinct identifiers, each representing a C library function. This table has a loading factor, α, of 10/52 = 0.19.

$$\propto = \frac{n}{bs}$$

The hash function must map each of the possible identifiers onto one of the number, 0-25.

We can construct a fairly simple hash function by associating the letter, a-z, with the number, 0-25, respectively, and then defining the hash function, f (x), as the first character of x.

Using this scheme, the library functions acos, define, float, exp, char, atan, ceil, floor, clock, and ctime hash into buckets 0, 3, 5, 4, 2, 0, 2, 5, 2, and 2, respectively.

index	Slot 0	Slot 1
0	acos	atan
1		
2	char	ceil
3	define	
4	exp	
5	float	floor
6		

clock ctime

7		
8		
9		

25		

Table) Hash table with 26 buckets and two slots per bucket

The identifier clock hashes into the bucket ht[2]. Since this bucket is full, we have an overflow.

Dynamic Hashing

To ensure good performance, it is necessary to increase the size of a hash table whenever its loading density exceeds a prescribed threshold. So, for example if we currently have b buckets in our hash table and using the division hash function with divisor D=b. When an insert causes the loading density to exceed the pre-specified threshold, we use array doubling to increase the number of buckets to 2b.

2-010	**10**
4-100	**00**
5-101	**01**
3-011	**11**

00	**4**
01	**5**
10	**2**
11	**3**

Now if we want to insert any mode values in the hash table they will overflow. To avoid this we can use dynamic hashing. We can add some more memory and readjust the values already stored in the hash table and add the new values in the hash table.

Using 2 digited index there is a possibility of having 4 buckets. If we use 3 digited index there is a possibility of using 8 buckets. The storage of hash table will be doubled to allow you to store more values.

2-010
4-100
5-101
3-011
7-111
6-110

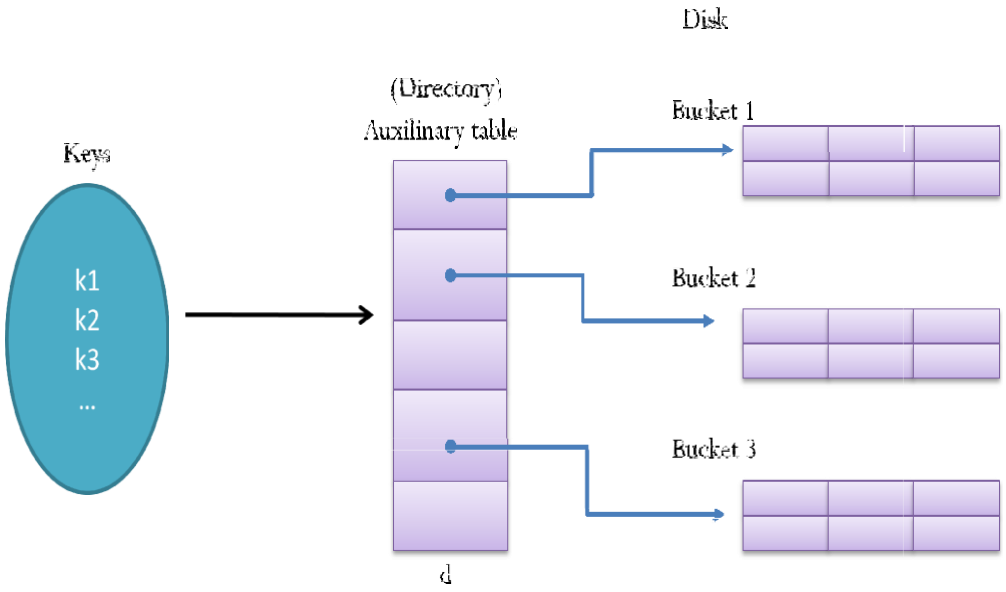
000	
001	
010	**2**
011	**3**
100	**4**
101	**5**
110	**6**
111	**7**

We consider two forms of Dynamic hashing- one uses a directory and the other does not.

1) Dynamic Hashing Using Directories

2)Directory Less Dynamic Hashing

Dynamic Hashing Using Directories



Dynamic Hashing

- The dynamic hashing method is used to overcome the problems of static hashing like bucket overflow.
- In this method, data buckets grow or shrink as the records increases or decreases. This method is also known as Extendable hashing method.

- This method makes hashing dynamic, i.e., it allows insertion or deletion without res lting in poor performance.

How to search a key

- First, calculate the hash address of the key.
- Check how many bits are used in the directory, and these bits are called as i.
- Take the least significant i bits of the hash address. This gives an index of the directory.
- Now using the ind x, go to the directory and find buc et address where the record mi ht be.

How to insert a new record

- Firstly, you have to follow the same procedure for retrieval in ending up some bucket.
- If there is still space in that bucket, then place the record i r it.
- If the bucket is full, then we will split the bucket and redi s records.

Example 1: Conside r the following grouping of keys and insert them into buckets, depending on the prefix of their hash address:

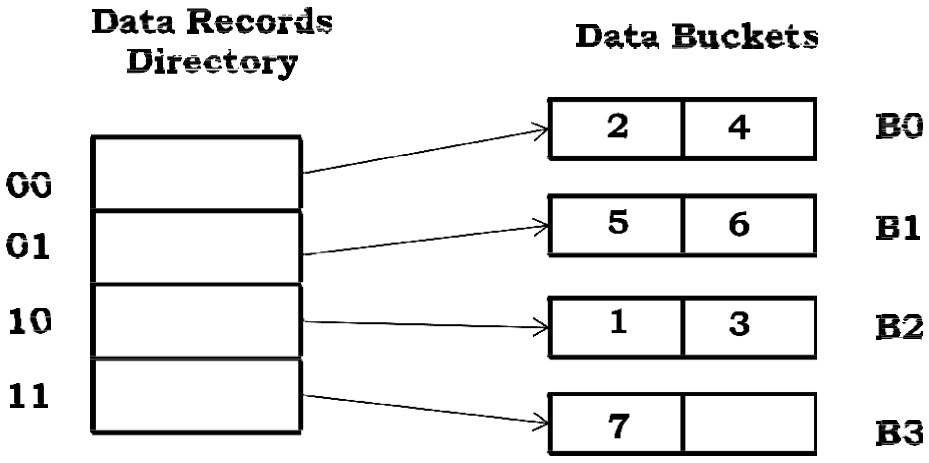
Key	Hash address
1	11010
2	00000
3	11110
4	00000
5	01001
6	10101
7	10111

Using two bits there is a possibility of producing 4 different codes. tory Assume that the dire chas 4 codes and 4 buckets. Each bucket has

2 slots. Consider 00 pointing to Bucket B0, 01 pointing to Bucket B1, 10 pointing to bucket B2 and 11 pointing to bucket B3. Each bucket can store 2 values.

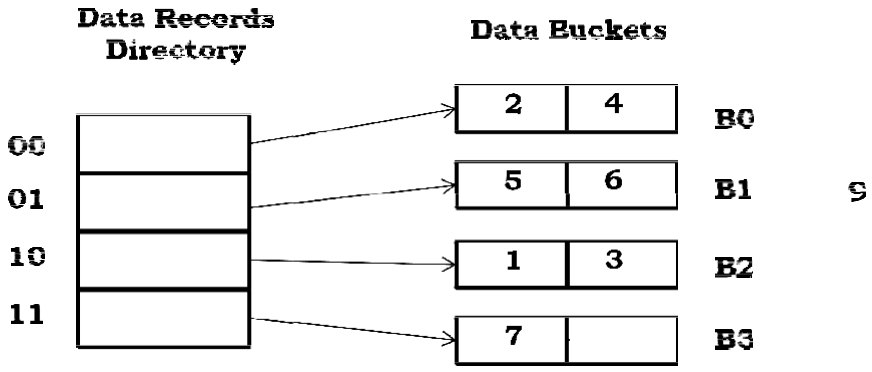
The last two bits of 2 and 4 are 00. So it will go into bucket B0. The last two bits of 5 and 6 are 01, so it will go into bucket B1. The last two bits of 1 and 3 are 10, so it will go into bucket B2. The last two bits of 7 are 11, so it will go into B3.

Key	Hash Address
1	11010
2	00000
3	11110
4	00000
5	01001
6	10101
7	10111



Insert key 9 with hash address 10001 into the above structure:

- Since **key 9** has hash address **10001**, it must go into the bucket B1. But bucket B1 is full, so it will get split.



- The splitting will separate 5, 9 from 6 since last three bits of key 5 and key 9 are 001, so it will go into bucket B1, and the last three bits of key 6 are 101, so it will go into bucket B5.

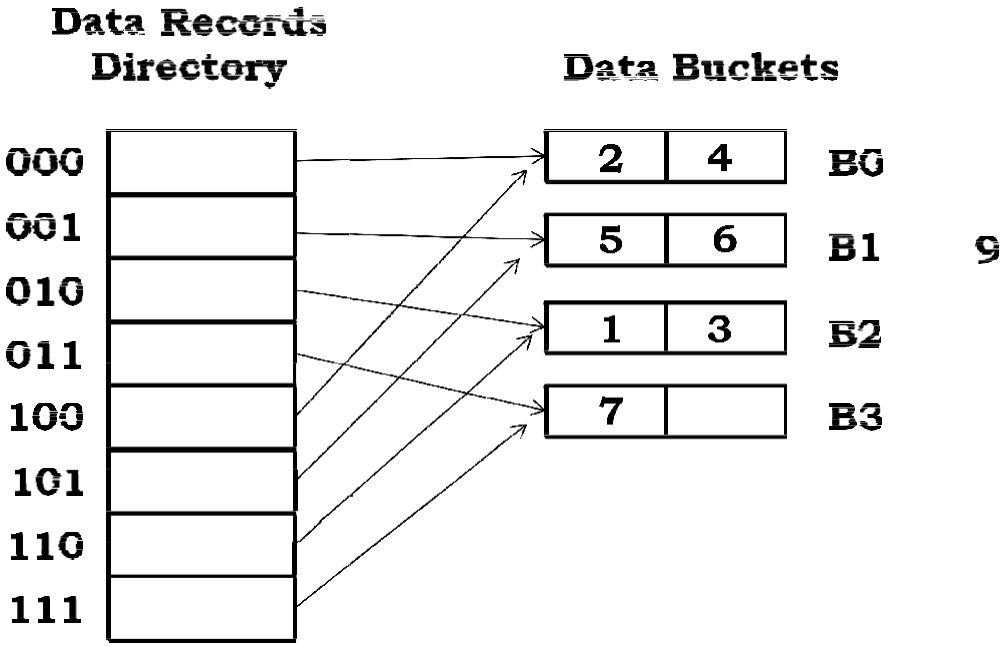
- Keys 2 and 4 are still in B0. The record in B0 pointed by the 000 and 100 entry because last two bits of both the entry are 00.

- Keys 1 and 3 are still in B2. The record in B2 pointed by the 010 and 110 entry because last two bits of both the entry are 10.

- Key 7 are still in B3. The record in B3 pointed by the 1 1 1 and 011 entry because last two bits of both the entry are 11.

5 - 01001
6 - 10101
9 - 10001

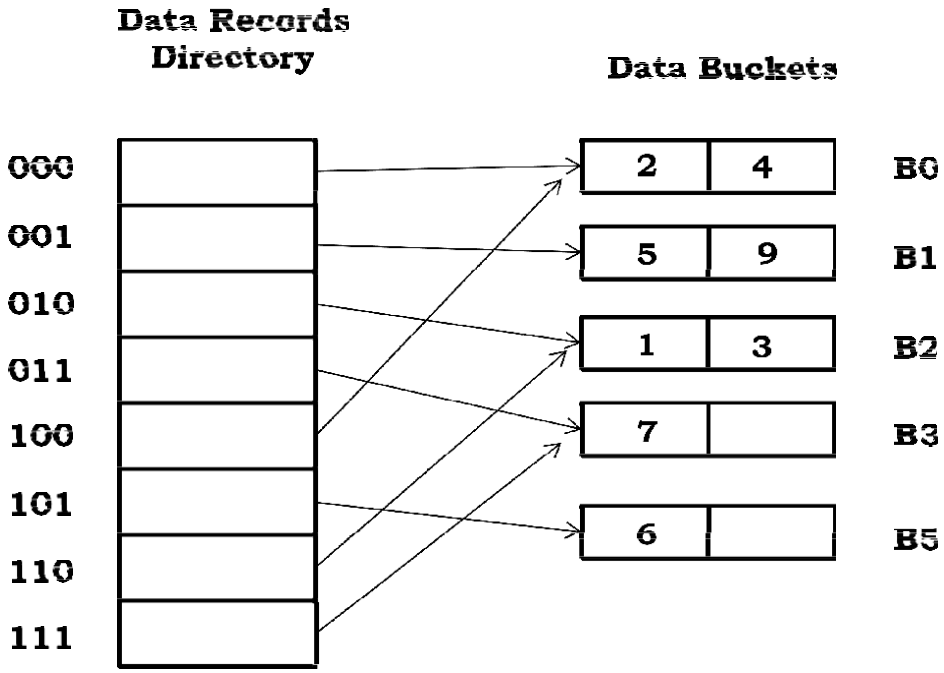
5 - 01001
6 - 10101
9 - 10001



5 - 01001
6 - 10101
9 - 10001

5 - 01001
6 - 10101
9 - 10001

B1
B5
B1



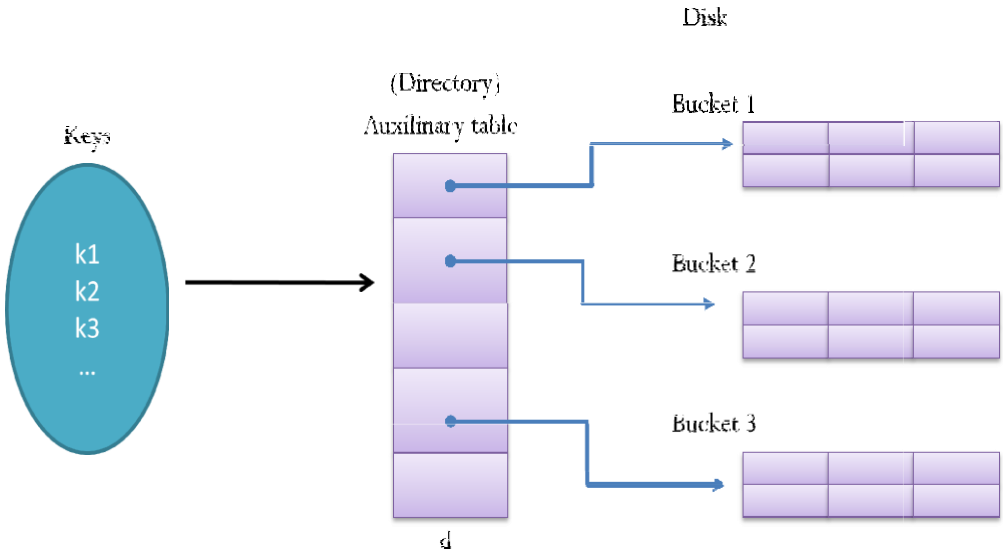
Advantages of dynamic hashing

- In this method, the performance does not decrease as the data grows in the system. It simply increases the size of memory to accommodate the data.
- In this method, memory is well utilized as it grows and shrinks with the data. There will not be any unused memory lying.
- This method is good for the dynamic database where data grows and shrinks frequently.

Disadvantages of dynamic hashing

- In this method, if the data size increases then the bucket size is also increased. These addresses of data will be maintained in the bucket address table. This is because the data address will keep changing as buckets grow and shrink. If there is a huge increase in data, maintaining the bucket address table becomes tedious.
- In this case, the bucket overflow situation will also occur. But it might take little time to reach this situation than static hashing.

Dynamic Hashing Using Directories Uses an auxiliary table to record the pointer of each bucket



Example 2) Define the hash function h(k) transforms k into 6-bit binary integer. Insert keys in the hash table using dynamic hashing using directory.

k	h(k)
A0	**100 000**
A1	**100 001**
B0	**101 000**
B1	**101 001**
C2	**110 010**
C3	**110 011**
C5	**110 101**
C1	**110 001**
C4	**110 100**

The size of directory d is 2^r, where r is the number of bits use all d to identify h(k).

Initially, Let r = 2. Thus, the size of directory d = 2^2 = 4.

Suppose h(k, p) is defi ed as the p least significant bits in h(k), where p is also called dictionary depth.

E. g.

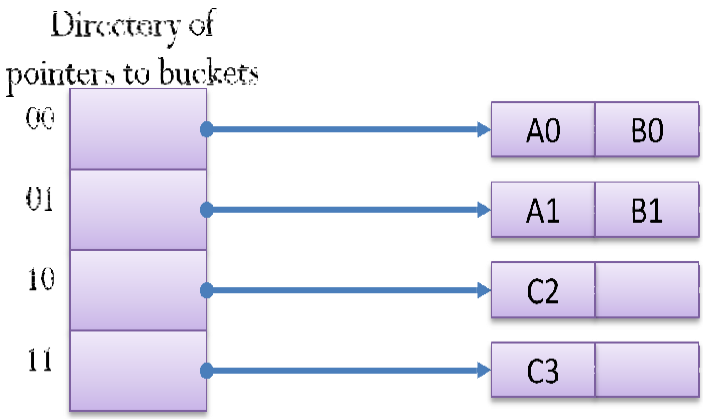
h(C5) = 110 101

h(C5, 2) = 01

h(C5, 3) = 101

Consider the following keys have been already stored. The le ast r is 2 to differentiate all the input keys.

k	h(k)
A0	100 000
A1	100 001
B0	101 000
B1	101 001
C2	110 010
C3	110 011



Depth 2

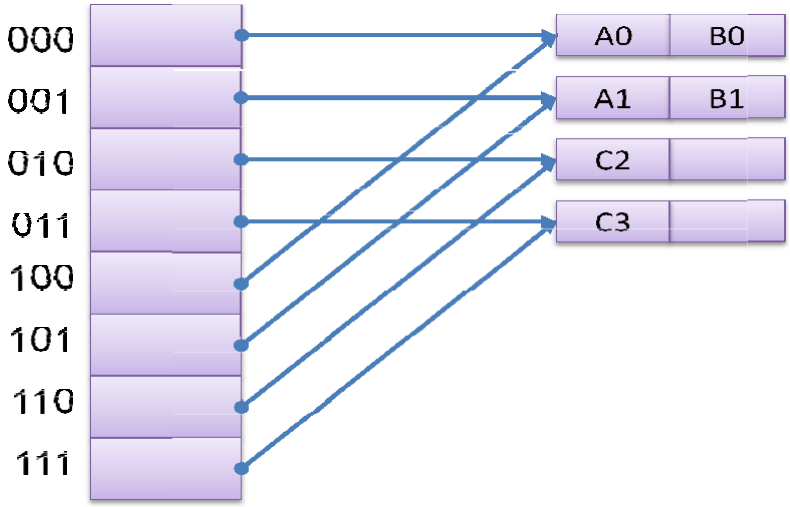
When C5 (110101) is to enter in to the buckets of the has h table, normally consider the least 2 digits of the hash address which is **01**

Key	Hash address
A1 -	100001
B1 -	101001
C5 -	110101

In the directory we can see this is pointing to a bucket where both slots are already filled with A1 and B1. The Key C5 is over flow from This bucket. Now let us consider the least 3 bits of the hash address.

Key	Hash Address
A1	100 001
B1	101 001
C5	110 101

If we consider 3 bits as directory index we can have 8 pointers in the directory.



Depth 3

When C5 (110101) is to enter

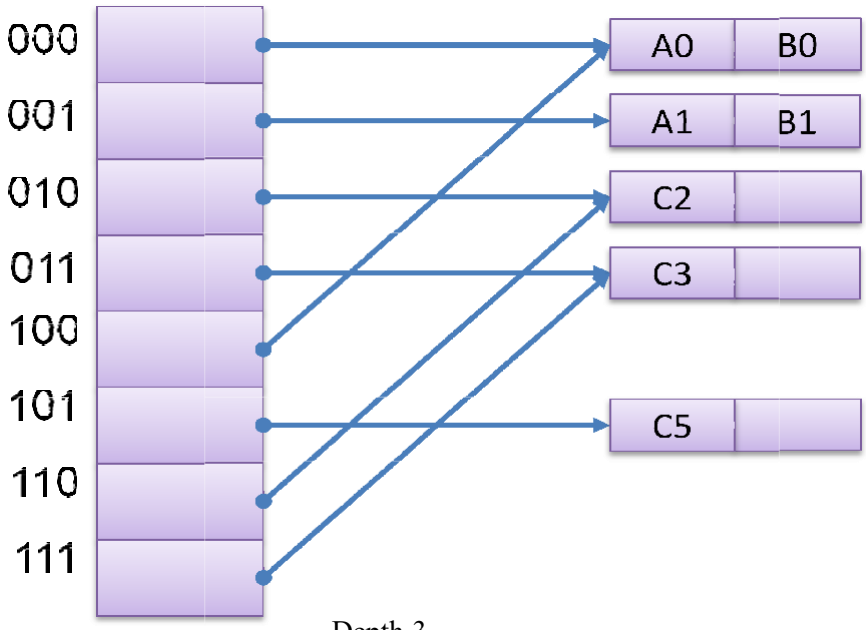
Since r=2 and h(C5, 2) = 01, follow the pointer of d[01]. A1

and B1 have been at d[01]. Bucket overflows.

Find the least u such that h(C5, u) is not the same with some keys in h(C5, 2) (01) bucket.
In this case, u = 3.

Since u > r, expand the size of d to 2^u and duplicate the pointers half to the new (why?).

Rehash identifiers 01 (A1 and B1) and C5 using new hash function h(k, u).



Depth 3

Let r = u = 3.

When **C1 (110001)** is to e tar

Since r=3 and h(C1, 3) = 001, follow the pointer of d[001]. A1 and B1 have been at d[001]. Bucket overflows.

Key	Hash address
A1 -	100001
B1 -	101001
C1 -	110001

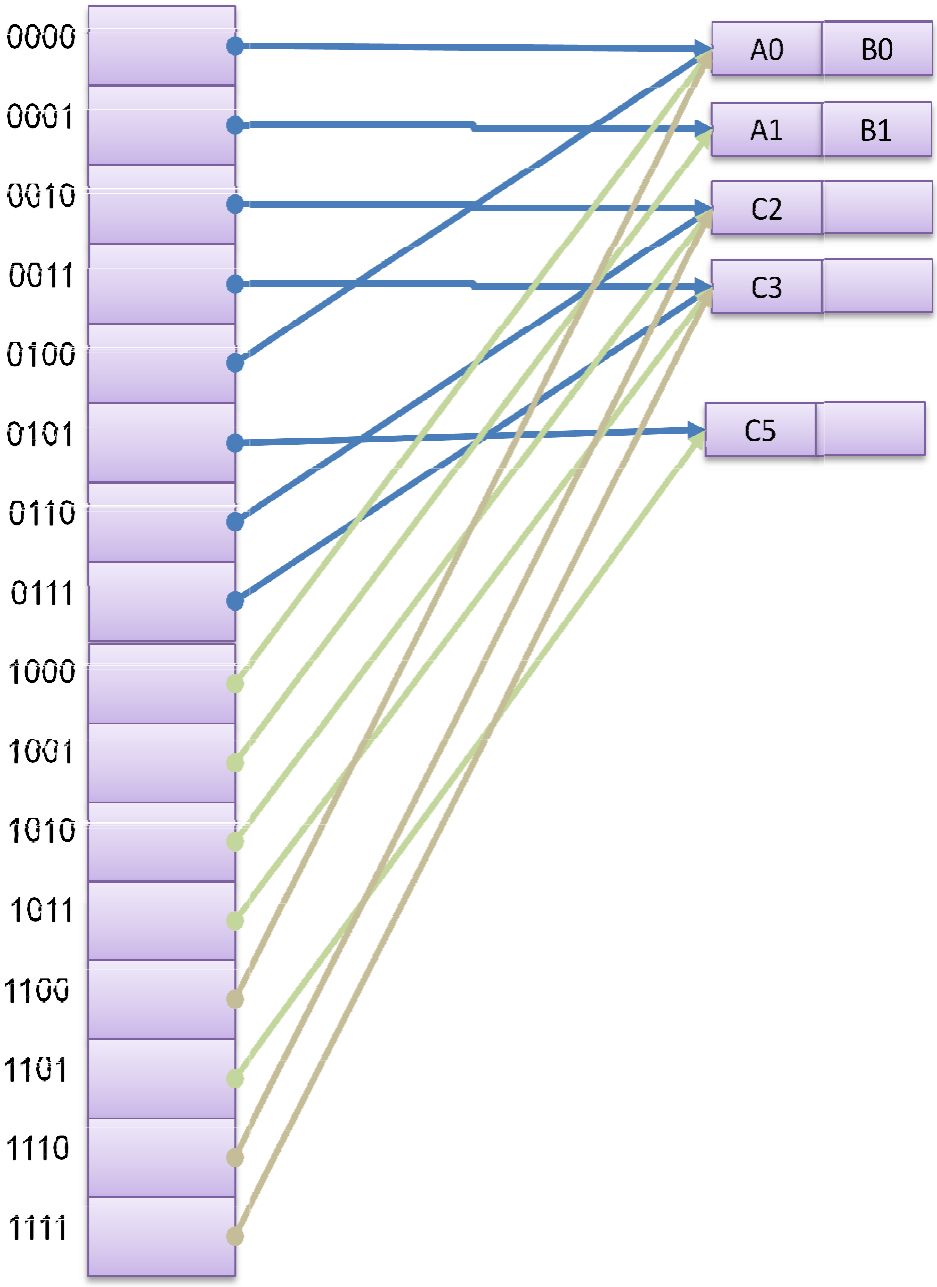
Find the least u such that h(C1, u) is not the same with some keys in h(C1, 3) (001) bucket.

In this case, u = 4.

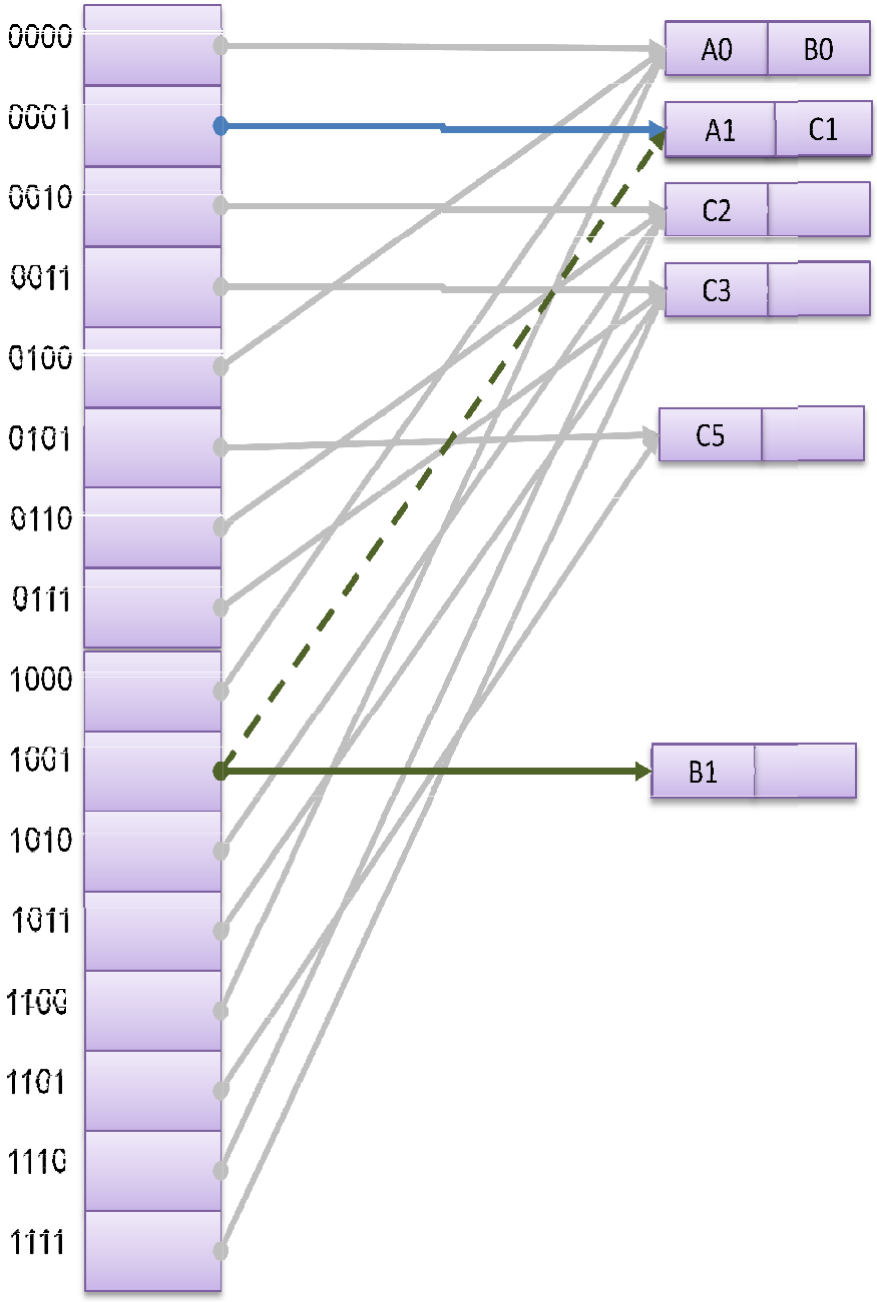
Key	Hash address
A1 -	100001
B1 -	101001
C1 -	110001

A1, C1 will be placed in one bucket and B1 will be placed in new bucket.

Since u > r, expand the size of d to 2^u and duplicate the pointers half. to the new


0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
A0
B0
A1
B1
C2
C3
C5

Depth 4


0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
A0
B0
A1
C1
C2
C3
C5
B1

depth 4

When C4 (110100) is to enter

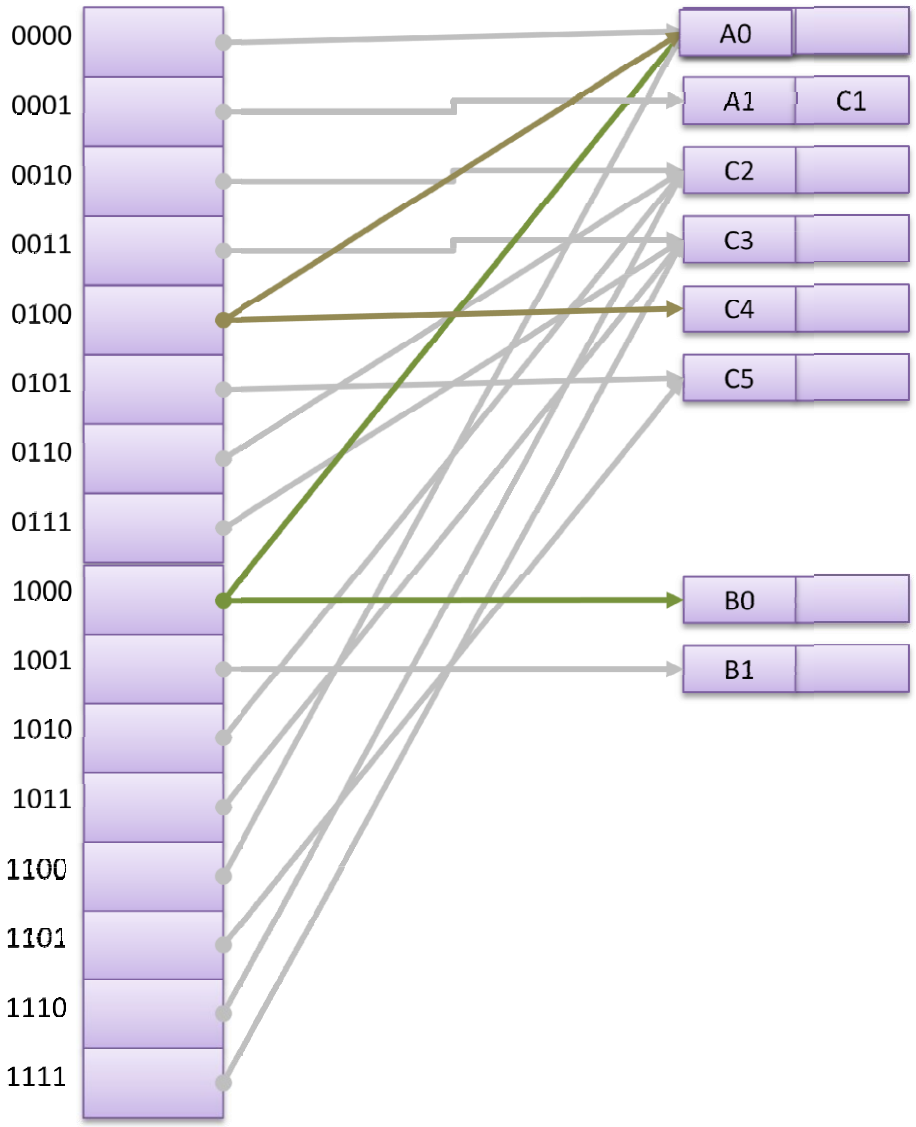
Since r=4 and h(C4, 4) = 0100, follow the pointer of d[0100].
A0 (100000) and B0 (101000) have been at d[0100]. Bucket overflows.

A0 ---- 100000
B0 ---- 101000

C1 ---- 110001

C4 ---- 110100

Find the least u such that h(C1, u) is not the same with so h(C1, e keys in 4) (0100) bucket.
In this case, u = 3.
Since u = 3 < r = 4, d is not required to expand its size.



Advantages

le used in

- Only doubling directory rather than the whole hash ta static hashing.
- Only rehash the ent ies in the buckets that overflows.

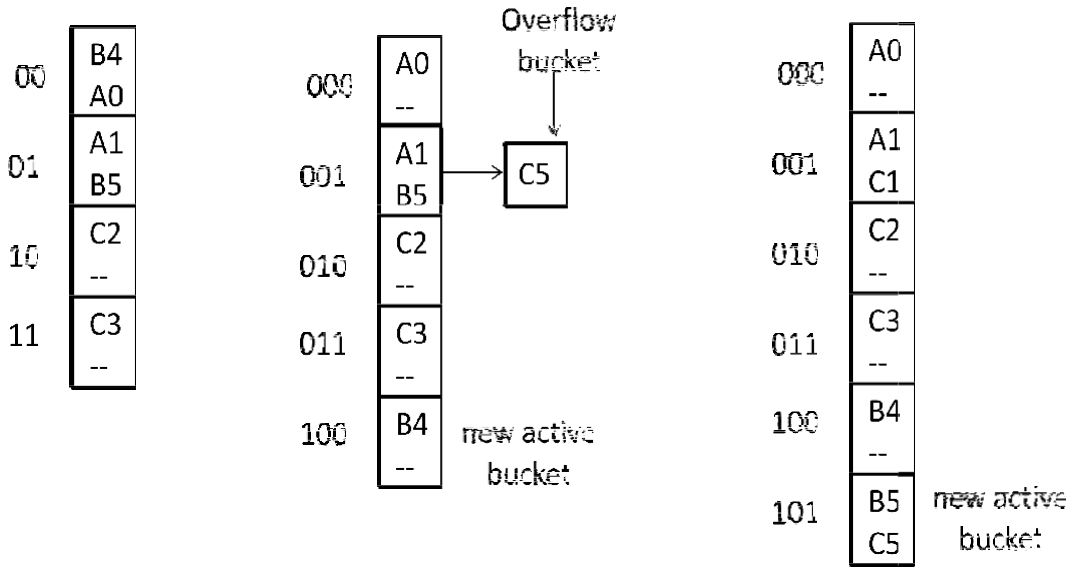
Directory **Less Dynamic Hashing**

Directory less Dynamic hashing is also known as linear dynamic hashing

k	h(k)
A_0	100 000
A_1	100 001
B_0	101 000
B_1	101 001
B_4	101 100
B_5	101 101
C1	110 001
C2	110 010
C3	110 011
C5	110 101

Figure a) shows a directory less hash table ht with r=2 the number of bits of h(k) used to index into the hash table and q=0. The number of active buckets are 4 indexed (00 01 10 11). Each active bucket has 2 slots.

Insert B4, A0, A1, B5, C2 and C3. Each dictionary pair is either in an active or an overflow bucket.



a) r=2, q=0 b)insert C5, r=2, q=1 c) insert C1, r=2,q=2

d is directory buckets=2^r

additional links q may be added where $0<=q<=2^r$.

Figure b) r=2, q=1, h(k,3) has been used for chains 000 and 100.
Insert C5
q =1 means one bucket added
h(k,3) means using lower 3 digits as index.
after adding a bucket readjust keys.
C5 is in overflow bucket

Figure c) r=2 , q=2, h(k,3) has been used for chains 000 , 100 and 101.
Insert C1, q=2 means 2 buckets added

An example of directory less hashing after two insertions.

Initially, there are four pages/buckets, each addressed by two bits (Figure (a). Two of the pages/buckets are full, and two have one identifier each.

When C5 is inserted, it hashes to the page/bucket whose address is 01 (Figure (b). Since that page/bucket is full, an overflow node is allocated to hold C5. At the same time, we add a new page at the end of the storage, rehash the identifiers in the first page, and split them between the first and new page. B4 moves to last page as the last 3 bits of B4 are 100. The last 3 bits of key C5 are 101. And we don't have a bucket with address 101 so C5 still in overflow node.

In the next step, we insert the identifier C1. Last 3 bits of C1 are 001. Since it hashes to the same page as C5, we use another overflow node to store it.

A1 - 100 001

B5 - 101 101

C5 - 110 101

C1 - 110 001

We add another new page to the end of the file and rehash the identifiers in the second page. The address of new bucket added to the has table is 101.

Last 3 bits of A1, C1 are 001. Last 3 Bits of B5 and C5 are 101. So B5 and C5 will be storing in the new bucket and C1 will be storing in 001 bucket so that there will be no overflow.

Hashing Collision Resolution Techniques in Python

1. **Chaining**
2. **Linear Probing (Open Addressing)**
3. **Quadratic Probing (Open Addressing)**
4. **Double Hashing (Open Addressing)**

Common Hash Functions

```
def hash1(key, size):
    return key % size

def hash2(key, size):
    return 1 + (key % (size - 1))
```

Chaining

Concept

- Each index contains a **list**
- Collisions are stored in the same bucket

Python Code

```
def chaining_hashing(keys, size):
    table = [[] for _ in range(size)]

    for key in keys:
        index = hash1(key, size)
        table[index].append(key)

    return table
```

Test Case

```
size = 7
keys = [50, 700, 76, 85, 92, 73, 101]

table = chaining_hashing(keys, size)
print("Chaining Hash Table:")
for i in range(size):
    print(i, "->", table[i])
```

Linear Probing (Open Addressing)

Concept

- If collision occurs → check **next index**
- Formula:

index = (h(key) + i) % size

Python Code

```
def linear_probing(keys, size):
    table = [-1] * size

    for key in keys:
        index = hash1(key, size)
        while table[index] != -1:
            index = (index + 1) % size
```

```
        table[index] = key

    return table
```

Test Case

```
print("\nLinear Probing Hash Table:")
print(linear_probing(keys, size))
```

Quadratic Probing (Open Addressing)

Concept

- Uses square of probe count
- Formula:

index = (h(key) + i²) % size

Python Code

```
def quadratic_probing(keys, size):
    table = [-1] * size

    for key in keys:
        h = hash1(key, size)
        i = 0
        while table[(h + i*i) % size] != -1:
            i += 1
        table[(h + i*i) % size] = key

    return table
```

Test Case

```
print("\nQuadratic Probing Hash Table:")
print(quadratic_probing(keys, size))
```

Double Hashing (Open Addressing)

Concept

- Uses **two hash functions**
- Formula:

index = (h1(key) + i × h2(key)) % size

Python Code

```
def double_hashing(keys, size):
    table = [-1] * size

    for key in keys:
        h1 = hash1(key, size)
        h2 = hash2(key, size)
        i = 0
        while table[(h1 + i*h2) % size] != -1:
            i += 1
        table[(h1 + i*h2) % size] = key

    return table
```

Test Case

```
print("\nDouble Hashing Hash Table:")
```

```
print(double_hashing(keys, size))
```

Sample Output (Conceptual)

```
Chaining Hash Table:
0 -> [700]
1 -> [50]
2 -> [85]
3 -> [73]
4 -> [76]
5 -> [92]
6 -> [101]

Linear Probing Hash Table:
[700, 50, 85, 73, 76, 92, 101]

Quadratic Probing Hash Table:
[700, 50, 85, 73, 76, 92, 101]

Double Hashing Hash Table:
[700, 50, 85, 73, 76, 92, 101]
```

Comparison Summary

Method	Extra Space	Clustering	Performance
Chaining	Yes	No	Good
Linear Probing	No	Primary clustering	Average
Quadratic Probing	No	Secondary clustering	Better
Double Hashing	No	Least clustering	Best

Example:

Below is a **CodeChef-style Python program** that implements **all four hashing collision-resolution techniques**:

1. **Chaining**
2. **Linear Probing (Open Addressing)**
3. **Quadratic Probing (Open Addressing)**
4. **Double Hashing (Open Addressing)**

with **CodeChef input format** and **test cases**.

Problem Statement (CodeChef Style)

You are given a hash table of size M and N keys to insert.
Using the specified hashing technique, insert all keys into the hash table and display the final hash table.

Input Format

T
method

M N
k1 k2 k3 ... kN
Where:

- T = number of test cases
- method:
  - 1 → Chaining
  - 2 → Linear Probing
  - 3 → Quadratic Probing
  - 4 → Double Hashing
- M = size of hash table
- N = number of keys
- k1..kN = keys to insert

Output Format

- Print the hash table after inserting all keys
- For chaining, print lists
- For probing, print table values (-1 for empty slots)

Python Code (CodeChef Compatible)

```
def chaining(m, keys):
    table = [[] for _ in range(m)]
    for key in keys:
        index = key % m
        table[index].append(key)
    return table


def linear_probing(m, keys):
    table = [-1] * m
    for key in keys:
        index = key % m
        i = 0
        while table[(index + i) % m] != -1:
            i += 1
        table[(index + i) % m] = key
    return table


def quadratic_probing(m, keys):
    table = [-1] * m
    for key in keys:
        index = key % m
        i = 0
        while table[(index + i * i) % m] != -1:
            i += 1
        table[(index + i * i) % m] = key
    return table


def double_hashing(m, keys):
    table = [-1] * m
```

```
    def hash2(key):
        return 1 + (key % (m - 1))

    for key in keys:
        h1 = key % m
        h2 = hash2(key)
        i = 0
        while table[(h1 + i * h2) % m] != -1:
            i += 1
        table[(h1 + i * h2) % m] = key
    return table


# -------- CodeChef Driver Code --------
t = int(input())

for _ in range(t):
    method = int(input())
    m, n = map(int, input().split())
    keys = list(map(int, input().split()))

    if method == 1:
        result = chaining(m, keys)
        for i in range(m):
            print(i, "->", result[i])

    elif method == 2:
        result = linear_probing(m, keys)
        print(*result)

    elif method == 3:
        result = quadratic_probing(m, keys)
        print(*result)

    elif method == 4:
        result = double_hashing(m, keys)
        print(*result)
```

Sample Test Cases

Test Case 1 – Chaining

Input

1
1
7 5
50 700 76 85 92

Output

0 -> [700]
1 -> []

2 -> [50]
3 -> []
4 -> []
5 -> [85, 92]
6 -> [76]

Test Case 2 – Linear Probing

Input
1
2
7 5
50 700 76 85 92
Output
700 -1 50 76 85 92 -1

Test Case 3 – Quadratic Probing

Input
1
3
7 5
50 700 76 85 92
Output
700 -1 50 76 -1 85 92

Test Case 4 – Double Hashing

Input
1
4
7 5
50 700 76 85 92
Output
700 -1 50 76 92 85 -1

Example 2:

Hash Table Collision Resolution
You are given a hash table of size M and N keys to be inserted using different hashing techniques.
Implement the following collision resolution methods:

1. **Chaining**
2. **Linear Probing (Open Addressing)**
3. **Quadratic Probing (Open Addressing)**
4. **Double Hashing (Open Addressing)**

The primary hash function is:
h(k) = k % M
Write programs to insert keys and display the final hash table.
Input Format (Common)
M N
k1 k2 k3 ... kN
Where:

- M → hash table size

- N → number of keys
- ki → keys to be inserted

Constraints

- $1 \le M \le 20$
- $1 \le N \le M$
- $0 \le key \le 10^5$
-

1) Chaining

Python Code (CodeChef Compatible)

```
def chaining_hashing(M, keys):
    table = [[] for _ in range(M)]

    for key in keys:
        index = key % M
        table[index].append(key)

    for i in range(M):
        print(f"{i}:", *table[i])
```

Test Case

Input

7 5
10 20 15 7 32

Output

0:
1:
2:
3:
4:
5:
6:
(Actual chains depend on modulo result)

Dry Run Table

Key	h(k)=k%7	Action
10	3	Insert at index 3
20	6	Insert at index 6
15	1	Insert at index 1
7	0	Insert at index 0
32	4	Insert at index 4

2) Linear Probing (Open Addressing)

Formula

$h(k, i) = (h(k) + i) \% M$

Python Code

```
def linear_probing(M, keys):
    table = [-1] * M

    for key in keys:
        index = key % M
        i = 0
        while table[(index + i) % M] != -1:
            i += 1
        table[(index + i) % M] = key

    print(*table)
```

Dry Run Table

Key	h(k)	Probes	Final Index
10	3	0	3
20	6	0	6
15	1	0	1
7	0	0	0
32	4	0	4

3) Quadratic Probing (Open Addressing)

Formula

$h(k, i) = (h(k) + i^2) \% M$

Python Code

```
def quadratic_probing(M, keys):
    table = [-1] * M

    for key in keys:
        index = key % M
        i = 0
        while table[(index + i*i) % M] != -1:
            i += 1
        table[(index + i*i) % M] = key

    print(*table)
```

Dry Run Table

Key	h(k)	i^2	Final Index
10	3	0	3
20	6	0	6
15	1	0	1
7	0	0	0
32	4	0	4

4) Double Hashing (Open Addressing)

Formula

$h1(k) = k \% M$

$h2(k) = 1 + (k \% (M - 1))$

$h(k, i) = (h1(k) + i * h2(k)) \% M$

Python Code

```
def double_hashing(M, keys):
    table = [-1] * M

    for key in keys:
        h1 = key % M
        h2 = 1 + (key % (M - 1))
        i = 0
        while table[(h1 + i * h2) % M] != -1:
            i += 1
        table[(h1 + i * h2) % M] = key

    print(*table)
```

Dry Run Table

Key	h1	h2	i	Index
10	3	4	0	3
20	6	3	0	6
15	1	2	0	1
7	0	2	0	0
32	4	3	0	4

Comparison Table (Exam Favorite)

Method	Extra Space	Clustering	Performance
Chaining	Yes	No	Good
Linear Probing	No	Primary	Poor
Quadratic Probing	No	Secondary	Better
Double Hashing	No	Minimal	Best

Difference Between All Hashing Methods

Method	Formula	Collision Handling	Example
Division	$h(k)=k\%size$	Basic hashing	$50\%7=1$
Mid-Square	Middle digits of k^2	Reduces clustering	$44^2=1936 \rightarrow 93$
Digit Folding	Sum of digit groups	Simple	$1234 \rightarrow 12+34=46$
Digit Analysis	Select specific digits	Data-dependent	ID numbers

ALL THE BEST