UNIT-3 Searching, Sorting and Hashing

Table of Contents

- **Searching:**
  - Concept of Searching
  - Sequential Search
  - Index Sequential Search
  - Binary Search
- **Sorting:**
  - Concept of Sorting
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
  - Quick Sort
  - Merge Sort
  - Heap Sort
  - Radix Sort
- **Hashing:**
  - Concept of Hashing
  - Collision Resolution Techniques used in Hashing

Dr. Sunil Kumar, CSE Dept., MIET Meerut

Concept of Searching

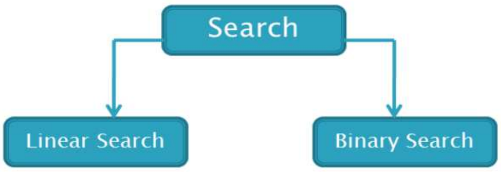
- **Searching in data structure** refers to the **process of finding location LOC of an element in a list**.
- This is one of the important parts of many **data structures** algorithms, as one operation can be performed on an element if and only if we find it.

Why do we need Searching?

- Searching is one of the core computer science operation.
- We know that today's computers store a lot of information.
- To retrieve this information proficiently we need very efficient searching algorithms.

Types of Searching

- Many different searching techniques exist and the most commonly used searching techniques are:

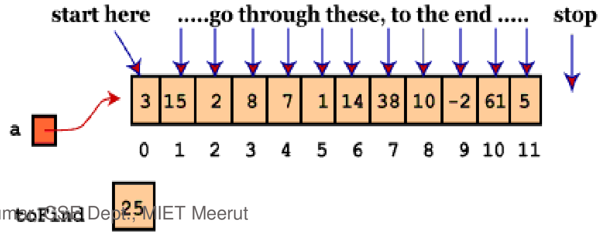


Linear Search

- Linear search or sequential search is a method for finding a particular value in a list that consists of checking every one of its elements, one at a time and in sequence, until the desired one is found.

How Linear Search works

- Linear search in an array is usually programmed by stepping up an index variable until it reaches the last index.
- This normally requires two comparisons for each list item:
  - One to check whether the index has reached the end of the array, and
  - Another one to check whether the item has the desired value.



Linear Search Algorithm

- Repeat For J = 1 to N
- If (ITEM == A[J]) Then
  - Print: ITEM found at location J
- Return [End of If]
- [End of For Loop]
- If (J > N) Then
  - Print: ITEM doesn't exist [End of If]
- Exit


```
#include<stdio.h>
#include<conio.h> void main()
{
int i ,n, item, a[20];
clrscr( );

printf("\nEnter no of elements: ");
   scanf("%d",&n);

printf("\nEnter %d elements: ",n);
   for(i=1; i<=n; i++)
{
scanf("%d",&a[i]);
}
printf("\nEnter the element to be
   searched: "); scanf("%d",&item);

for(i=1; i<=n; i++)
{
if(a[i]==item)
{
printf("\n%d is present at position
   %d", a[i], i);
break;
}
}

if(i>n)
printf("\n Element is not present.");
getch( );
}
```

Complexity of Linear Search

- Linear search on a list of n elements. In the worst case, the search must visit every element once. This happens when the value being searched for is either the last element in the list, or is not in the list.
- However, on average, assuming the value searched for is in the list and each list element is equally likely to be the value searched for, the search visits only n/2 elements. In best case the array is already sorted.

Algorithm	Worst Case	Average Case	Best Case
Linear Search	O(n)	O(n)	O(1)

Indexed Sequential Search

- An index file can be used to effectively overcome the problem associated with sequential files and to speed up the key search.
- In this searching method, first of all, an index file is created, that contains some specific group or division of required record when the index is obtained, then the partial indexing takes less time cause it is located in a specified group.

Note: When the user makes a request for specific records it will find that index group first where that specific record is recorded.

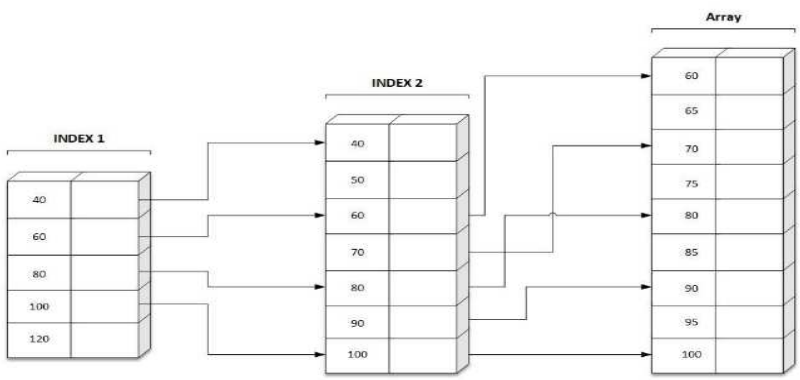
Indexed Sequential Search…

Characteristics of Indexed Sequential Search:

- In Indexed Sequential Search a sorted index is set aside in addition to the array.
- Each element in the index points to a block of elements in the array or another expanded index.
- The index is searched 1st then the array and guides the search in the array.

Note: Indexed Sequential Search actually does the indexing multiple times, like creating the index of an index.

Indexed Sequential Search



Advantages:

- More efficient
- Time required is less

Binary Search

- A binary search or half-interval search algorithm finds the position of a specified input value (the search "key") within an array sorted by key value.
- For binary search, the array should be arranged in ascending or descending order.

How Binary Search Works

- Searching a sorted collection is a common task.
- A dictionary is a sorted list of word definitions.
- Given a word, one can find its definition. A telephone book is a sorted list of people's names, addresses, and telephone numbers.
- Knowing someone's name allows one to quickly find their telephone number and address.

How Binary Search Works

- If the array is sorted, then we can apply the binary search technique.

number

0	1	2	3	4	5	6	7	8
5	12	17	23	38	44	77	84	90

- The basic idea is straightforward. First search the value in the middle position. If key X is less than this value, then search the middle of the left half next. If X is greater than this value, then search the middle of the right half next. Continue in this manner.

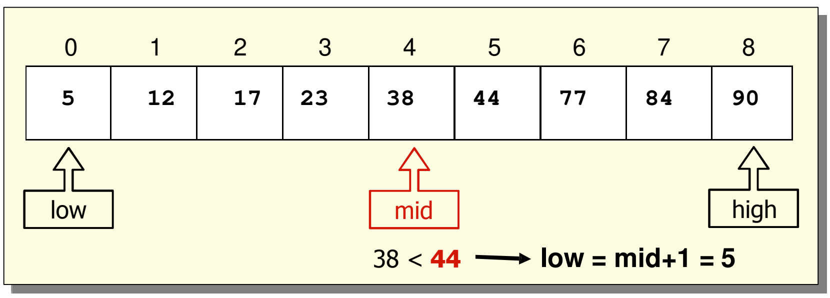
Dr. Sunil Kumar, CSE Dept., MIET Meerut

Sequence of Successful Search

search(44)

	low	high	mid
#1	0	8	4

$$mid = \left\lfloor \frac{low + high}{2} \right\rfloor$$



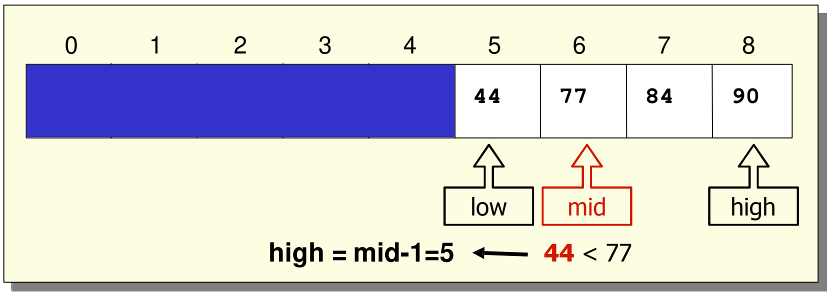
Dr. Sunil Kumar, CSE Dept., MIET Meerut

Sequence of Successful Search

	low	high	mid
#1	0	8	4
#2	5	8	6

search(44)

$$mid = \left\lfloor \frac{low + high}{2} \right\rfloor$$



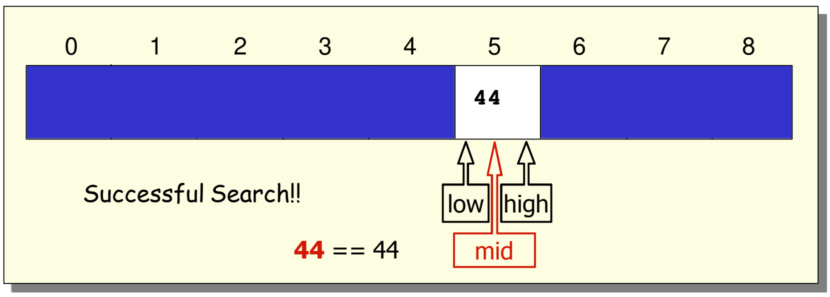
19 Dr. Sunil Kumar, CSE Dept., MIET Meerut

Sequence of Successful Search

	low	high	mid
#1	0	8	4
#2	5	8	6
#3	5	5	5

search(44)

$$mid = \left\lfloor \frac{low + high}{2} \right\rfloor$$



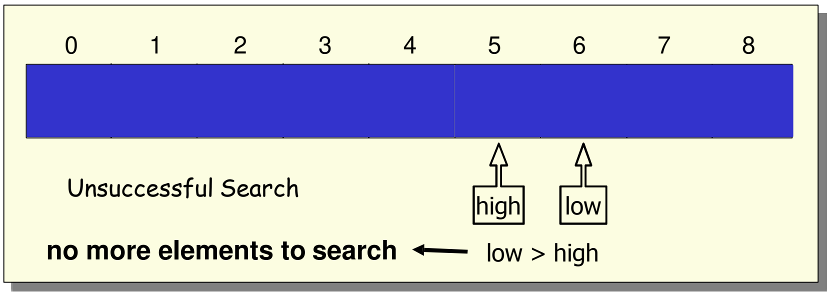
20 Dr. Sunil Kumar, CSE Dept., MIET Meerut

Sequence of Unsuccessful Search

search(45)

	low	high	mid
#1	0	8	4
#2	5	8	6
#3	5	5	5
#4	6	5	

$$mid = \left\lfloor \frac{low + high}{2} \right\rfloor$$



Concept of Sorting

- Sorting is nothing but arrangement/storage of data in sorted order, it can be in ascending or descending order.
- The term Sorting comes into picture with the term Searching.
- There are so many things in our real life that we need to search, like a particular record in database, roll numbers in merit list, a particular telephone number, any particular page in a book etc.

Internal and External Sorting

- Any sort algorithm that uses main memory exclusively during the sorting is called as internal sort.
- Internal sorting is faster than external sorting.

Internal Sorting

- Bubble Sort
- Selection Sort
- Insertion Sort
- Quick Sort
- Heap Sort
- Radix Sort
- Bucket Sort
- Shell Sort

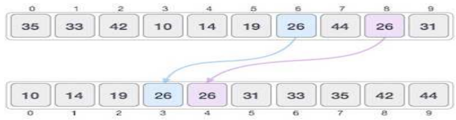
External Sorting

- Any sort algorithm that uses external memory, such as tape or disk, during the sorting is called as external sorting.

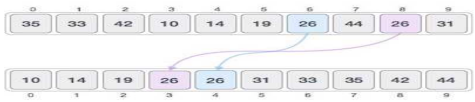
- Merge sort is an example of external sorting.

Stable and Not Stable Sorting

- If a sorting algorithm, after sorting the contents, ***does not change the sequence of similar content*** in which they appear, it is called **stable sorting**.



- If a sorting algorithm, after sorting the contents, ***changes the sequence of similar content*** in which they appear, it is called **unstable sorting.**



Bubble Sort

- Bubble sort is a simple sorting technique.
- This sorting technique 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 sorting 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...

1) Starting with the first element (index = 0), compare the current element with the next element of the array.
2) If the current element is greater than the next element of the array, swap them.
3) If the current element is less than the next element, move to the next element. **Repeat Step 1**.

Bubble Sort

Let us take the array of numbers "5 1 4 2 8", and sort the array from lowest number to greatest number using bubble sort. In each step, elements written in bold are being compared. Three passes will be required.

First Pass

(**5 1** 4 2 8) → (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
(1 **5 4** 2 8) → (1 **4 5** 2 8), Swap since 5 > 4
(1 4 **5 2** 8) → (1 4 **2 5** 8), Swap since 5 > 2
(1 4 2 **5 8**) → (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass

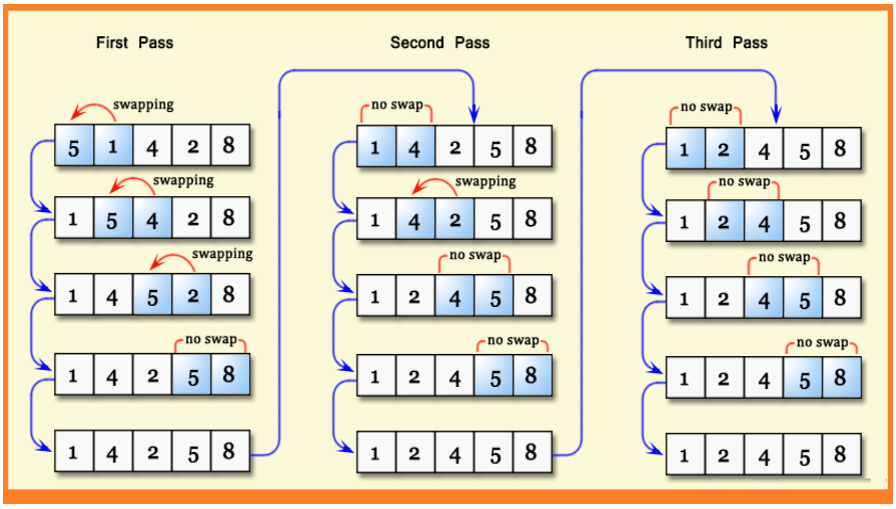
(**1 4** 2 5 8) → (**1 4** 2 5 8)
(1 **4 2** 5 8) → (1 **2 4** 5 8), Swap since 4 > 2
(1 2 **4 5** 8) → (1 2 **4 5** 8)
(1 2 4 **5 8**) → (1 2 4 **5 8**)

Now, the array is already sorted, but the algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

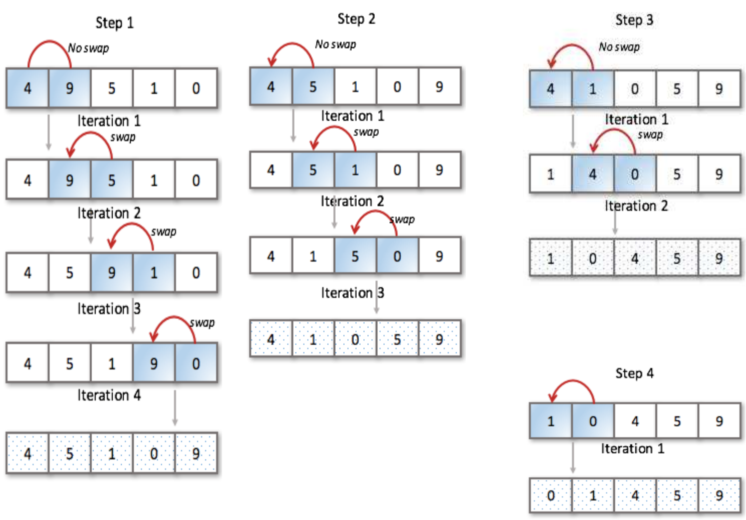
Third Pass

(**1 2** 4 5 8) → (**1 2** 4 5 8)
(1 **2 4** 5 8) → (1 **2 4** 5 8)
(1 2 **4 5** 8) → (1 2 **4 5** 8)
(1 2 4 **5 8**) → (1 2 4 **5 8**)

Bubble Sort



Bubble Sort



Algorithm

- **Bubble Sort (A, N)**
  - **Here A is an array with N elements. This algorithm sorts the elements in the array A.**
  - **Step 1**: Repeat Steps 2 and 3 for k = 1 to N-1.
  - **Step 2:** Set PTR=1.
  - **Step 3**: Repeat while PTR<= N-k:

    a) If A[PTR]> A[PTR+1], then:

    Swap A[PTR] and A[PTR+1].

    b) Set PTR=PTR+1.
  - **Step 4**: EXIT

Complexity of Bubble Sort Algorithm

- In Bubble Sort, n-1 comparisons will be done in 1st pass, n-2 in 2nd pass, n-3 in 3rd pass and so on. So the total number of comparisons will be:

$F(n)=(n-1)+(n-2)+\ldots\ldots\ldots\ldots+2+1=n(n-1)/2$
$= O(n^2)$

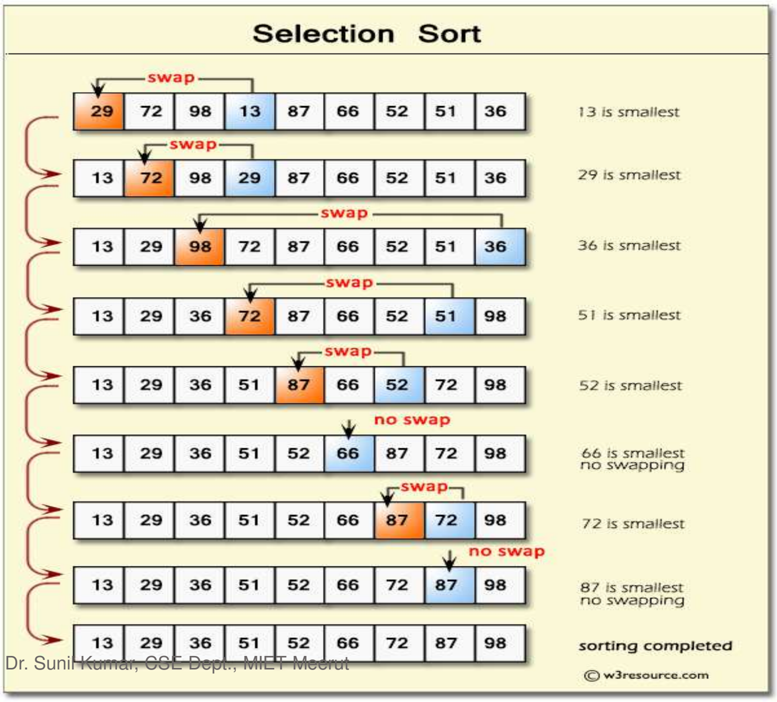
Algorithm	Worst Case	Average Case	Best Case
Bubble Sort	$n(n-1)/2 = O(n^2)$	$n(n-1)/2 = O(n^2)$	$O(n)$

Selection Sort

- Selection sorting is conceptually the simplest sorting algorithm.
- This algorithm first finds the smallest element in the array and exchanges 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 continues in this way until the entire array is sorted.

How Selection Sort Works

Pass	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
K=1 LOC=4	77	33	44	11	88	22	66	55
K=2 LOC=6	11	33	44	77	88	22	66	55
K=3 LOC=6	11	22	44	77	88	33	66	55
K=4 LOC=6	11	22	33	77	88	44	66	55
K=5 LOC=8	11	22	33	44	88	77	66	55
K=6 LOC=7	11	22	33	44	55	77	66	88
K=7 LOC=4	11	22	33	44	55	66	77	88
Sorted	11	22	33	44	55	66	77	88



Algorithm

- **Selection Sort (A, N)**
  - **This algorithm sorts the array A with N elements.**
  - **Step 1**: Repeat Steps 2 and 3 for K = 1 to N-1:
  - **Step 2**: CALL MIN(A, K, N, LOC)
  - **Step 3**: SWAP A[K] with A[LOC]

    Set TEMP= A[K],

    A[K]= A[LOC],

    A[LOC]=TEMP.
  - **Step 4**: EXIT

Algorithm...

- **Procedure MIN (A, K, N, LOC)**
  - This procedure finds the location LOC of the smallest element A[K], A[K+1],…, A[N], where A is an array.
  - **Step 1**: Set MIN= A[K] and LOC=K.
  - **Step 2**: Repeat Steps 2 for j=K+1, K+2,…,N:

    If MIN> A[j], then set MIN=A[j] and LOC=j;
  - **Step 3:** Return.

Complexity of Selection Sort Algorithm

- The number of comparison in the selection sort algorithm is independent of the original order of the element. That is there are n-1 comparison during PASS 1 to find the smallest element, there are n-2 comparisons during PASS 2 to find the second smallest element, and so on. Accordingly $F(n)=(n-1)+(n-2)+\ldots\ldots\ldots\ldots\ldots+2+1=n(n-1)/2$
  $= O(n^2)$

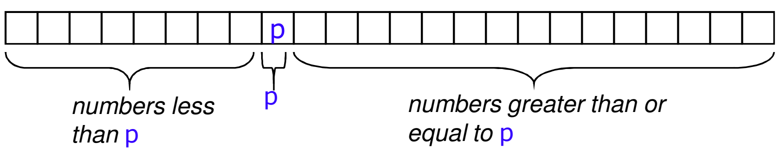
Algorithm	Worst Case	Average Case	Best Case
Selection Sort	$n(n-1)/2 = O(n^2)$	$n(n-1)/2 = O(n^2)$	$O(n^2)$

Quick Sort

- Quick Sort, as the name suggests, sorts any list very quickly.
- Quick sort is not stable search, but it is very fast and requires very less additional space.
- It is ***based on the rule of Divide and Conquer*** (***also called partition-exchange sort***).
- This algorithm divides the list into three main parts:
  - Elements less than the Pivot
  - Pivot element
  - Elements greater than the pivot element

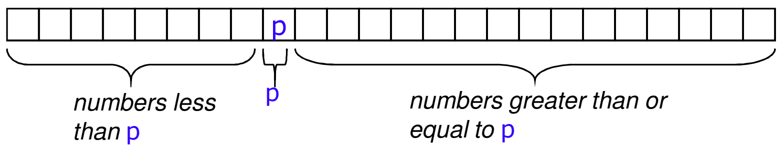
Quicksort I: Basic idea

- Pick some number p from the array
- Move all numbers less than p to the beginning of the array
- Move all numbers greater than (or equal to) p to the end of the array
- Quicksort the numbers less than p
- Quicksort the numbers greater than or equal to p



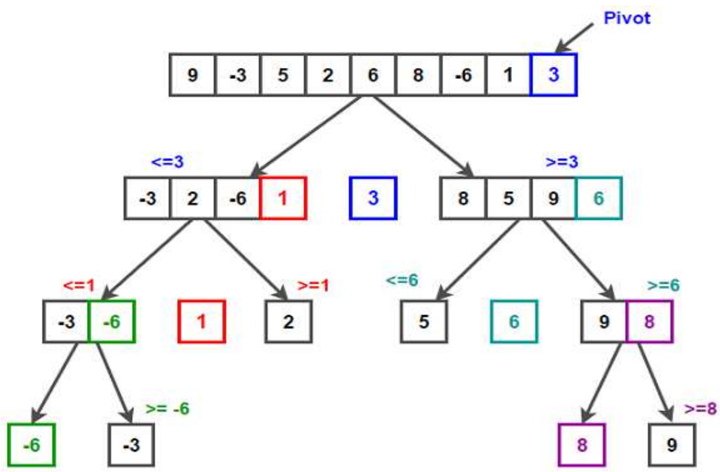
Partitioning (Quicksort II)

- A key step in the Quicksort algorithm is partitioning the array
  - We choose some (any) number p in the array to use as a pivot
  - We partition the array into three parts:



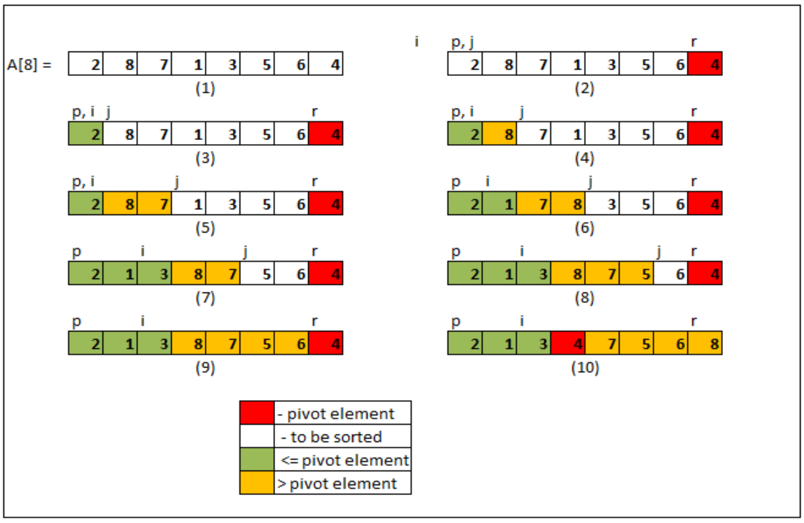
Dr. Sunil Kumar, CSE Dept., MIET Meerut

Quick Sort



Dr. Sunil Kumar, CSE Dept., MIET Meerut

How Quick Sort Works



Algorithm

QUICKSORT(*A, p, r*)

1. **if** $p < r$
2. **then** $q \leftarrow$ PARTITION(A, p, r)
3. QUICKSORT(A, p, q-1)
4. QUICKSORT(A, $q + 1$,r)

Partitioning the array

PARTITION(A, p, r)

1. $x \leftarrow A[r]$
2. $i \leftarrow p - 1$
3. for $j = p$ *to* $r - 1$
4. if $A[j] \leq x$
5. **then** $i = i + 1$
6. exchange A[i] ←→ A[j]
7. exchange $A[i+1] \leftarrow\rightarrow A[r]$
8. **return** i+1

Complexity of Quick Sort Algorithm

- The Worst Case occurs when the list is sorted. Then the first element will require n comparisons to recognize that it remains in the first position.
- Furthermore, the first sublist will be empty, but the second sublist will have n-1 elements. Accordingly the second element require n-1 comparisons to recognize that it remains in the second position and so on.

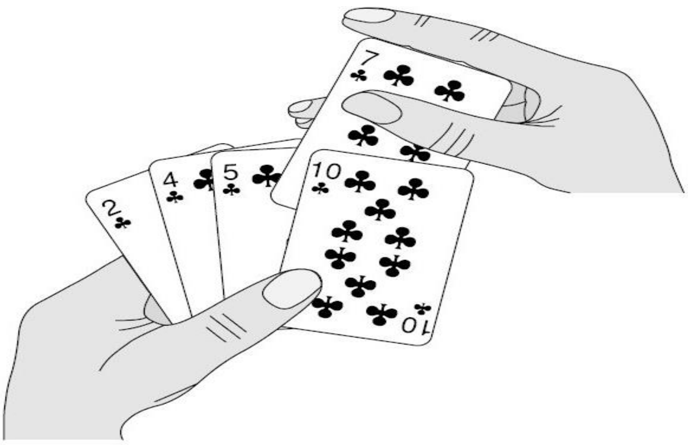
$$F(n)=(n-1)+(n-2)+\ldots\ldots\ldots\ldots+2+1=n(n-1)/2$$
$$= O(n^2)$$

Algorithm	Worst Case	Average Case	Best Case
Quick Sort	$n(n+1)/2 = O(n^2)$	O(n logn)	O(n logn)

Insertion Sort

- It is a simple sorting algorithm that builds the final sorted array (or list) one item at a time.
- This algorithm is less efficient on large lists than more advanced algorithms such as quicksort, heap sort, or merge sort.
- However, insertion sort provides several advantages:
  - Simple implementation
  - Efficient for small datasets
  - Stable; i.e., does not change the relative order of elements with equal keys.
  - In-place; i.e., only requires a constant amount O(1) of additional memory space.

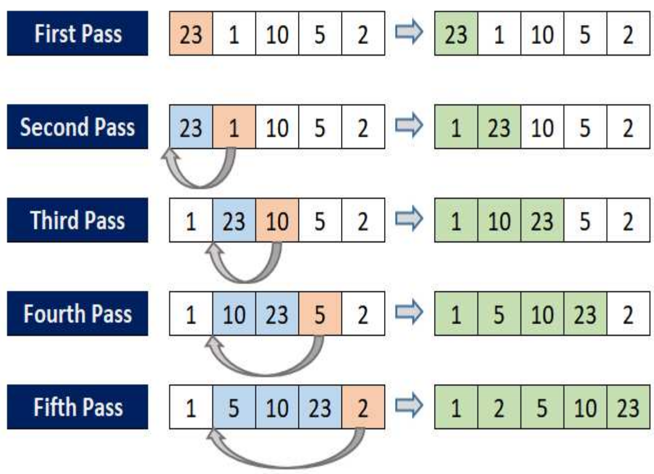
Insertion Sort



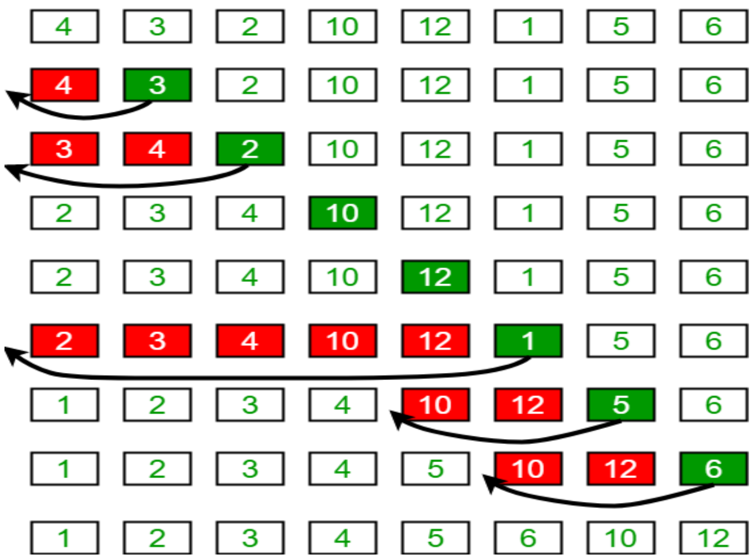
Insertion Sort

1. We start by making the second element of the given array, i.e. element at index 1, the key. The key element here is the new card that we need to add to our existing sorted set of cards(remember the example with cards above).
2. We compare the key element with the element(s) before it, in this case, element at index 0:
   - If the key element is less than the first element, we insert the key element before the first element.
   - If the key element is greater than the first element, then we insert it after the first element.
3. Then, we make the third element of the array as key and will compare it with elements to it's left and insert it at the right position.
4. And we go on repeating this, until the array is sorted.

Insertion Sort



Insertion Sort Execution Example



Algorithm

- **Algorithm : Insertion Sort (A, N)**
  - **Step 1**: Repeat Steps 2 to 5 for K = 1 to N-1
  - **Step 2**: SET TEMP = A[K]
  - **Step 3**: SET J = K-1
  - **Step 4**: Repeat while TEMP <=A[J]
    (a) SET A[J + 1] = A[J]
    (b) SET J = J - 1
  - **Step 5**: SET A[J + 1] = TEMP
  - **Step 6**: EXIT

Complexity of Insertion Sort

- The number f(n) of comparisons in the insertion sort algorithm can be easily computed. First of all, the worst case occurs when the array A is in reverse order and the inner loop must use the maximum number K-1 of comparisons. Hence
  F(n)=(n-1)+(n-2)+……………+2+1=n(n-1)/2
  $= O(n^2)$
- Furthermore, One can show that, on the average, there will be approximately (K-1)/2 comparisons in the inner loop. Accordingly, for the average case. $F(n)=O(n^2)$
- Thus the insertion sort algorithm is a very slow algorithm when n is very large.

Algorithm	Worst Case	Average Case	Best Case
Insertion Sort	$n(n-1)/2 = O(n^2)$	$n(n-1)/4 = O(n^2)$	O(n)

Merge Sort

- Merge Sort is ***based on the rule of Divide and Conquer***. But it doesn't divide the list into two halves.
- In merge sort, ***the unsorted list is divided into N sub-lists, each having one element***, because a list of one element is considered sorted.
- Then, it repeatedly merge these sub lists, to produce new sorted sub lists, and at lasts one sorted list is produced.

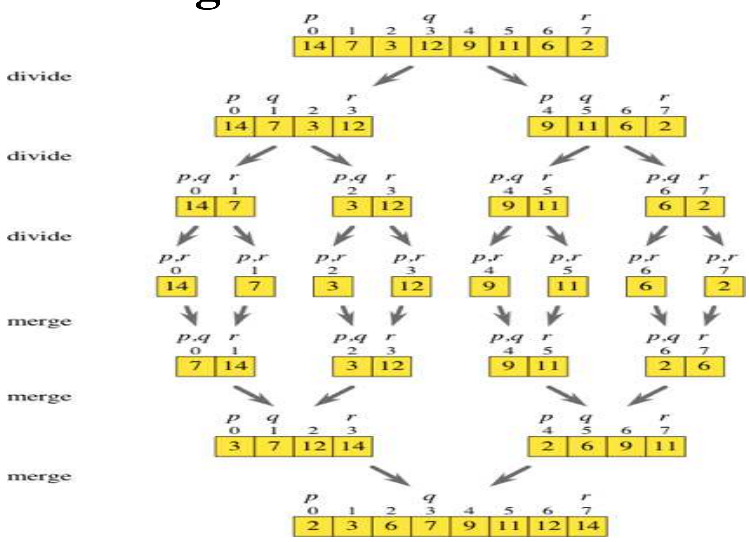
Merge Sort…

- **DIVIDE:** Divide the unsorted list into two sub lists of about half the size.
- **CONQUER:** Sort each of the two sub-lists recursively. If they are small enough just solve them in a straight forward manner.
- **COMBINE:** Merge the two-sorted sub-lists back into one sorted list.

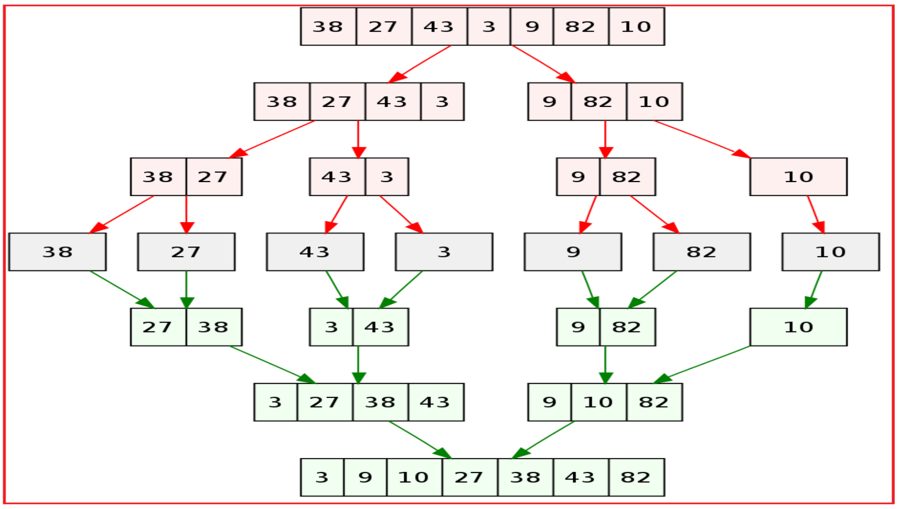
Merge Sort...

- Merge Sort is quite fast, and has a time complexity of O(n log n).
- It is also a stable sort, which means the equal elements are ordered in the same order in the sorted list.

How Merge Sort Works



Another Example



 Dr. Sunil Kumar, CSE Dept., MIET Meerut

Algorithm

ALGO.: Merge-Sort(A, p, r)

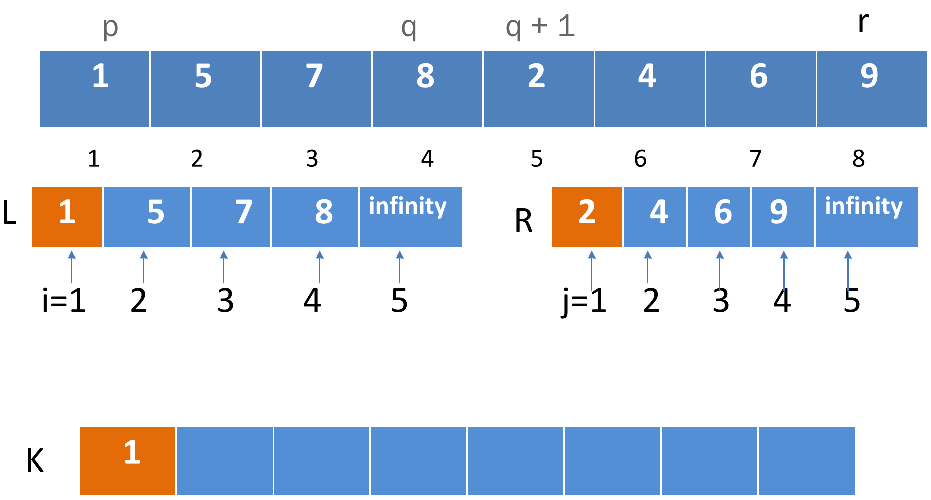
1. if $p < r$ // check for base case
2. then $q \leftarrow \lfloor (p + r)/2 \rfloor$ // divide step
3. Merge-Sort (A, p, q) // conquer step
4. Merge-Sort (A, q + 1, r) // conquer step
5. Merge (A, p, q, r) // conquer step

 Dr. Sunil Kumar, CSE Dept., MIET Meerut

Algorithm...

```
MERGE (A, p, q, r )
    n1 ← q − p + 1
    n2 ← r − q
    Create arrays L[1 . . n1 + 1]
    and R[1 . . n2 + 1]
    for i ← 1 to n1
        do L[i] ← A[p + i − 1]
    for j ← 1 to n2
        do R[j] ← A[q + j ]
    L[n1 + 1] ← ∞
    R[n2 + 1] ← ∞
    i ← 1
    j ← 1
    for k ← p to r
        do if L[i ] ≤ R[ j]
            then A[k] ← L[i]
                 i ← i + 1
            else A[k] ← R[j]
                 j ← j + 1
```


MERGE SORT EXAMPLE :



MERGE SORT EXAMPLE :

p			q	q + 1			r
1	5	7	8	2	4	6	9

L

1	5	7	8	infinity
i=1	2	3	4	5

R

2	4	6	9	infinity
j=1	2	3	4	5

K

1	2						

Dr. Sunil Kumar, CSE Dept., MIET Meerut

MERGE SORT EXAMPLE :

p			q	q + 1			r
1	5	7	8	2	4	6	9

L

1	5	7	8	infinity
i=1	2	3	4	5

R

2	4	6	9	infinity
j=1	2	3	4	5

K

1	2	4					

Dr. Sunil Kumar, CSE Dept., MIET Meerut

MERGE SORT EXAMPLE :

p			q	q + 1			r
1	5	7	8	2	4	6	9

L

1	5	7	8	infinity
i=1	2	3	4	5

R

2	4	6	9	infinity
j=1	2	3	4	5

K

1	2	4	5				

MERGE SORT EXAMPLE :

p			q	q + 1			r
1	5	7	8	2	4	6	9

L

1	5	7	8	infinity
i=1	2	3	4	5

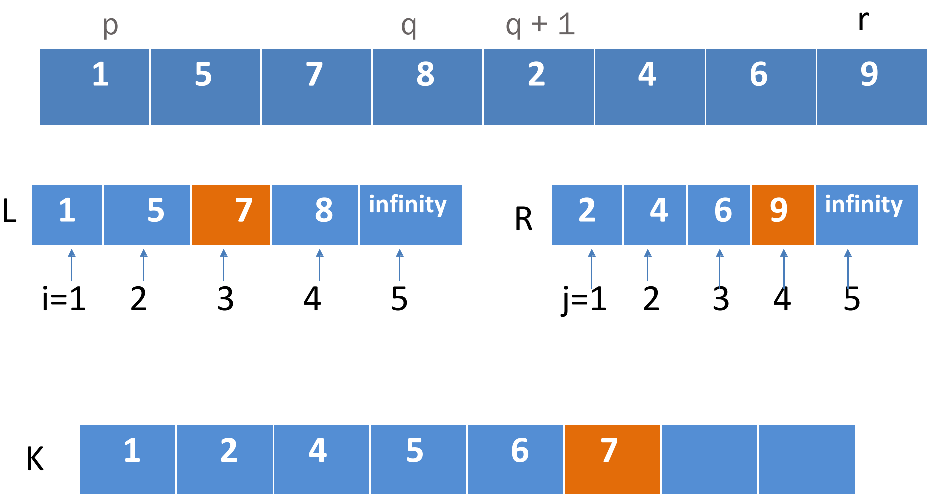
R

2	4	6	9	infinity
j=1	2	3	4	5

K

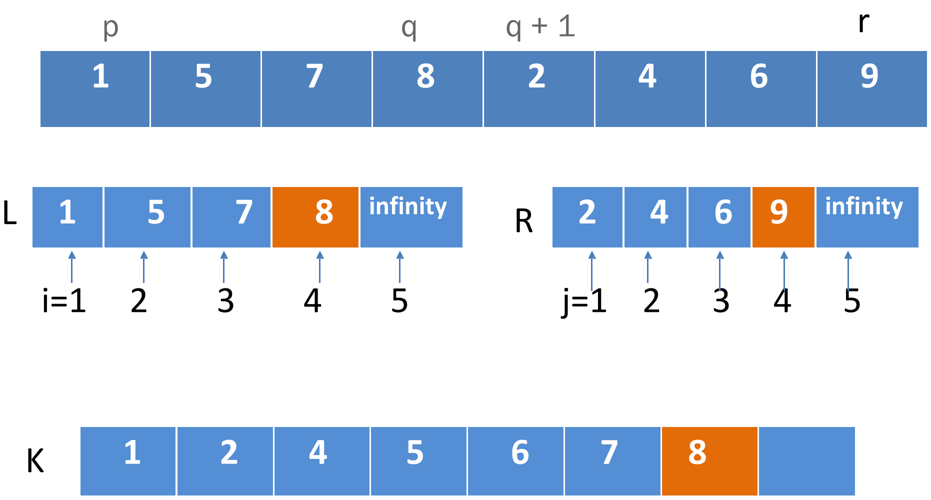
1	2	4	5	6			

MERGE SORT EXAMPLE :



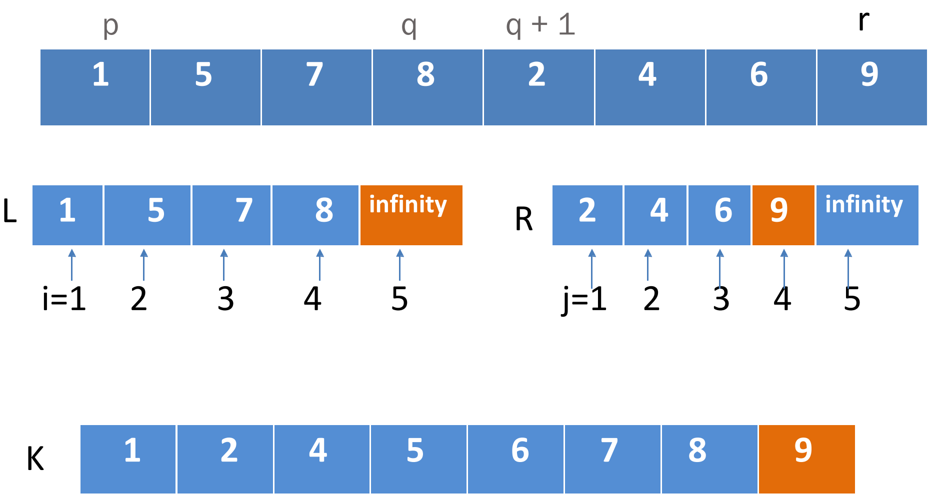
Dr. Sunil Kumar, CSE Dept., MIET Meerut

MERGE SORT EXAMPLE :



Dr. Sunil Kumar, CSE Dept., MIET Meerut

MERGE SORT EXAMPLE :



Complexity of Merge Sort Algorithm

- Let f(n) denote the number of comparisons needed to sort an n-element array A using merge-sort algorithm. The algorithm requires at most log n passes. Each pass merges a total of n elements and each pass require at most n comparisons.
- Thus for both the worst and average case

$$F(n) \leq n \log n$$

- Thus the time complexity of Merge Sort is O(n Log n) in all 3 cases (worst, average and best) as merge sort always divides the array in two halves and take linear time to merge two halves.

Algorithm	Worst Case	Average Case	Best Case
Merge Sort	O(n logn)	O(n logn)	O(n logn)

Heap Sort

- Heap Sort is one of the best sorting methods being in-place and with no quadratic worst-case scenarios.
- Heap sort algorithm is divided into two basic parts:
  - Creating a Heap of the unsorted list.
  - Then a sorted array is created by repeatedly removing the largest/smallest element from the heap, and inserting it into the array. The heap is reconstructed after each removal.

Types of Heap

- Max Heap
- Min Heap

1-Max Heap

Max-heap Definition:

- is a complete binary tree in which the value in each internal node is greater than or equal to the values in the children of that node.

Max-heap property:

- The key of a node is ≥ than the keys of its children.

2-Min heap :

Min-Heap Definition:

is a complete binary tree in which the value in each internal node is lower than or equal to the values in the children of that node.

Min-Heap property:

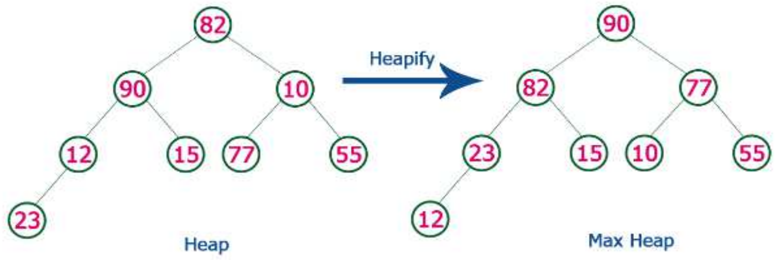
- The key of a node is <= than the keys of its children.

Example

Consider the following list of unsorted numbers which are to be sort using Heap Sort

82, 90, 10, 12, 15, 77, 55, 23

Step 1 - Construct a Heap with given list of unsorted numbers and convert to Max Heap



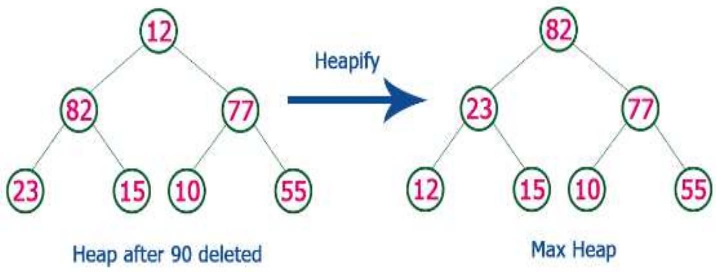
list of numbers after heap converted to Max Heap

90, 82, 77, 23, 15, 10, 55, 12

Dr. Sunil Kumar, CSE Dept., MIET Meerut

Example...

Step 2 - Delete root (**90**) from the Max Heap. To delete root node it needs to be swapped with last node (**12**). After delete tree needs to be heapify to make it Max Heap.



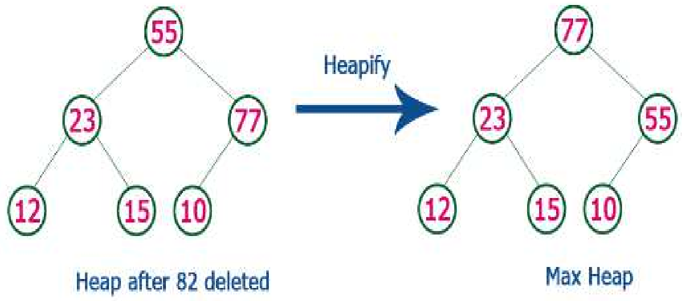
list of numbers after swapping 90 with 12.

12, 82, 77, 23, 15, 10, 55, **90**

Dr. Sunil Kumar, CSE Dept., MIET Meerut

Example...

Step 3 - Delete root (**82**) from the Max Heap. To delete root node it needs to be swapped with last node (**55**). After delete tree needs to be heapify to make it Max Heap.

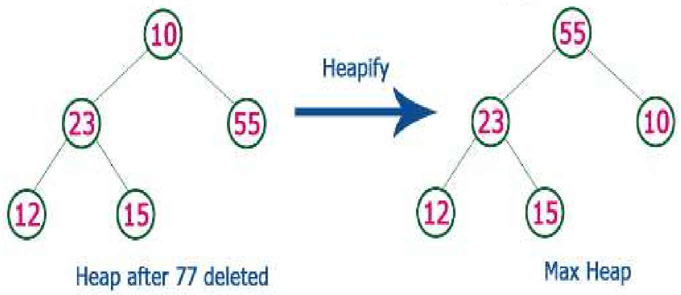


list of numbers after swapping 82 with 55.

12, 55, 77, 23, 15, 10, **82**, **90**

Example...

Step 4 - Delete root (**77**) from the Max Heap. To delete root node it needs to be swapped with last node (**10**). After delete tree needs to be heapify to make it Max Heap.

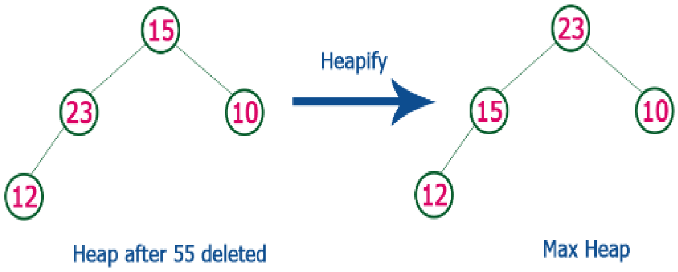


list of numbers after swapping 77 with 10.

12, 55, 10, 23, 15, **77**, **82**, **90**

Example...

Step 5 - Delete root (**55**) from the Max Heap. To delete root node it needs to be swapped with last node (**15**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 55 with 15.

12, 15, 10, 23, **55, 77, 82, 90**

Example...

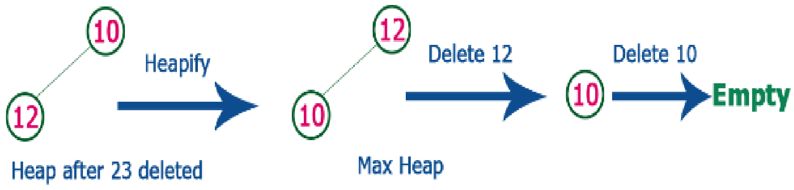
Step 6 - Delete root (**23**) from the Max Heap. To delete root node it needs to be swapped with last node (**12**). After delete tree needs to be heapify to make it Max Heap.

list of numbers after swapping 23 with 12.

12, 15, 10, **23, 55, 77, 82, 90**

Example

Step 7 - Delete root (**15**) from the Max Heap. To delete root node it needs to be swapped with last node (**10**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after Deleting 15, 12 & 10 from the Max Heap.

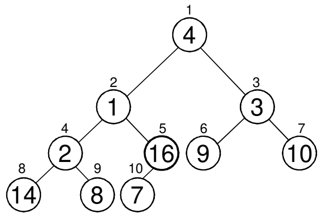
10, 12, 15, 23, 55, 77, 82, 90

Algorithm

- Convert an array A[1 … n] into a max-heap (n = length[A])
- The elements in the subarray A[(⌊n/2⌋+1) .. n] are leaves
- Apply MAX-HEAPIFY on elements between 1 and ⌊n/2⌋

Algo: BUILD-MAX-HEAP(A)

1. n = length[A]
2. **for** i ← ⌊n/2⌋ **down to** 1
3. **do** MAX-HEAPIFY(A, i, n)

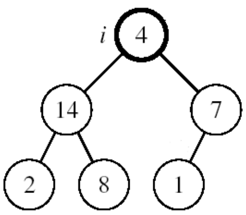


A:

4	1	3	2	16	9	10	14	8	7

Maintaining the Heap Property

- Assumptions:
  - Left and Right subtrees of i are max-heaps
  - A[i] may be smaller than its children



Algo: MAX-HEAPIFY(A, i, n)

1. l ← LEFT(i)
2. r ← RIGHT(i)
3. **if** l ≤ n and A[l] > A[i]
4. **then** largest ←l
5. **else** largest ←i
6. **if** r ≤ n and A[r] > A[largest]
7. **then** largest ←r
8. **if** largest ≠ i
9. **then** exchange A[i] ↔ A[largest]
10. MAX-HEAPIFY(A, largest, n)

HEAP-EXTRACT-MAX

Algo: HEAP-EXTRACT-MAX(A, n)

1. **if** n < 1
2. **then error** "heap underflow"
3. max ← A[1]
4. A[1] ←→ A[n]
5. MAX-HEAPIFY(A, 1, n-1)
6. **return** max

Complexity of Heap Sort Algorithm

- The algorithm has two phases, and we analyze the complexity of each phase separately.
- **Phase 1.**Since H is complete tree, its depth is bounded by log2m where m is the number of elements in H. Accordingly, the total number g(n) of comparisons to insert the n elements of A into H is bounded as g(n) ≤ n log2n
- **Phase 2.**Reheaping uses 4 comparisons to move the node L one step down the tree H. Since the depth cannot exceeds log2m , it uses 4log2m comparisons to find the appropriate place of L in the tree H. h(n)≤4nlog2n
- Thus each phase requires time proportional to nlog2n, the running time to sort n elements array A would be nlog2n

Algorithm	Worst Case	Average Case	Best Case
Heap Sort	O(n logn)	O(n logn)	O(n logn)

Dr. Sunil Kumar, CSE Dept., MIET Meerut

Radix Sort

- A multiple pass *distribution sort* algorithm that distributes each item to a *bucket* according to part of the item's *key* beginning with the least significant part of the key.
- After each pass, items are collected from the buckets, keeping the items in order, then redistributed according to the next most significant part of the key and so on.

Radix Sort

- The idea is to consider the key one character at a time and to divide the entries, not into two sub lists, but into as many sub-lists as there are possibilities for the given character from the key.
- If our keys, for example, are words or other alphabetic strings, then we divide the list into 26 sub-lists at each stage.
- That is, we set up a table of 26 lists and distribute the entries into the lists according to one of the characters in the key.

Example

Example: Sort the numbers 348, 143, 361, 423, 538, 128, 321, 543, 366.

	Input	0	1	2	3	4	5	6	7	8	9
Pass 1	348									348	
	143				143						
	361		361								
	423				423						
	538									538	
	128									128	
	321		321								
	543				543						
	366							366			

Example...

	Input	0	1	2	3	4	5	6	7	8	9
Pass 2	361							361			
	321			321							
	143					143					
	423			423							
	543					543					
	366							366			
	348					348					
	538				538						
	128			128							

Example...

	Input	0	1	2	3	4	5	6	7	8	9
Pass 3	321				321						
	423					423					
	128		128								
	538						538				
	143		143								
	543						543				
	348				348						
	361				361						
	366				366						

Example:

- Sort the numbers 551, 12, 346, 311 using Radix sort.

Pass 1		Pass 2		Pass 3		Pass 4	
Bucket	Values	Bucket	Values	Bucket	Values	Bucket	Values
0		0		0	12	0	12, 311, 346, 551
1	551, 311	1	311, 12	1		1	
2	12	2		2		2	
3		3		3	311, 346	3	
4		4	346	4		4	
5		5	551	5	551	5	
6	346	6		6		6	
7		7		7		7	
8		8		8		8	
9		9		9		9	

Algorithm

- **Step 1 -** Define 10 queues each representing a bucket for each digit from 0 to 9.
- **Step 2 -** Consider the least significant digit of each number in the list which is to be sorted.
- **Step 3 -** Insert each number into their respective queue based on the least significant digit.
- **Step 4 -** Group all the numbers from queue 0 to queue 9 in the order they have inserted into their respective queues.
- **Step 5 -** Repeat from step 3 based on the next least significant digit.
- **Step 6 -** Repeat from step 2 until all the numbers are grouped based on the most significant digit.

Complexity of Radix Sort

The list A of n elements A_1, A_2,……………A_n is given. Let d denote the radix(e.g d=10 for decimal digits, d=26 for letters and d=2 for bits) and each item A_i is represented by means of s of the digits:

Ai = di1 di2……………….. dis

The radix sort require s passes, the number of digits in each item . Pass K will compare each digit with each of the d digits. Hence C(n)≤ d*s*n

Algorithm	Worst Case	Average Case	Best Case
Radix Sort	$O(n^2)$	d*s*n	O(n logn)

Table of Contents

Introduction to Hashing

- Suppose that we want to store 10,000 students records (each with a 5-digit ID) in a given container.
  - A linked list implementation would take **O(n)** time.
  - A height balanced tree would take **O(log n)** access time.
  - Using an array of size 100,000 would take **O(1)** access time but will lead to a lot of space wastage.
- Is there some way that we could get **O(1)** access time without wasting a lot of space?
- **The answer is Hashing.**

Introduction to Hashing...

- Hashing is a technique used for performing insertions, deletions and finds in constant average time O(1).
- The techniques employed here is ***to compute location of desired record to retrieve it in a single access or comparison***.
- This data structure, however, is ***not efficient in operations that require any ordering information among the elements***, such as ***findMin, findMax*** and ***printing the entire table in sorted order***.

Applications:

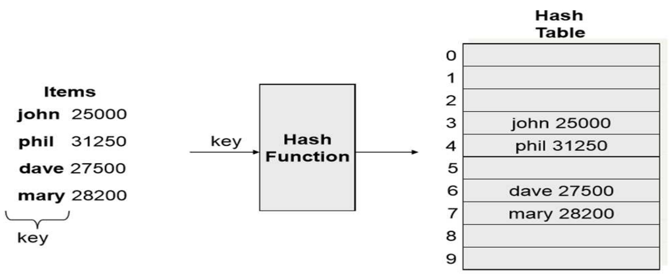
- Database Systems
- Symbol table for compilers
- Data Dictionaries
- Browser caches

Hash Table

- The ideal hash table structure is an array of some fixed size, containing the items.
- A stored item needs to have a data member, called ***key***, that will be used in computing the index value for the item.
  - Key could be an *integer*, a *string*, etc

    e.g. a name or Id that is a part of a large employee structure
- The size of the array is ***TableSize***.
- The items that are stored in the hash table are indexed by values from ***0* to *TableSize – 1***.
- Each key is mapped into some number in the range **0 to *TableSize – 1***.
- The mapping is called a ***hash function***.

Example



Hash Functions (cont'd)

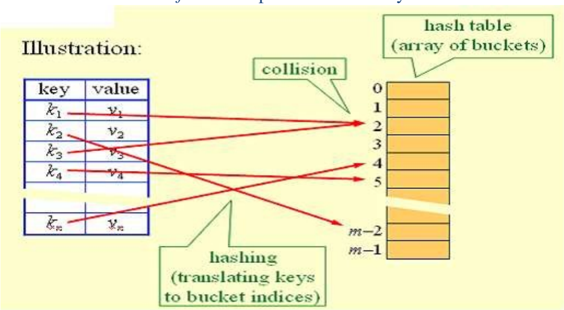
- A ***hash function*** (**h**) is a function which transforms a key from a set, **K**, into an index in a table of size n:

  h: K -> {0, 1, ..., n-2, n-1}
- A key can be a number, a string, a record etc.
- The size of the set of keys, **|K|**, to be relatively very large.
- It is possible for different keys to hash to the same array location. This situation is called ***collision*** and the colliding keys are called ***synonyms***.
- A common hash function is:

  h(x) = x mod SIZE
- if key = 27 and SIZE =10 then

  hash address = 27 % 10 = 7

- A good hash function should:
  - *Minimize* collisions.
  - Be *easy* and *quick* to compute.
  - Distribute key values *evenly* in the hash table.
  - Use *all the information* provided in the key.



Collision Resolution

- If, when an element is inserted, it hashes to the same value as an already inserted element, then we have a collision and need to resolve it.
  i.e. For any two keys k_1 and k_2,
  $H(k_1) = H(k_2) = \beta$
- There are several methods for dealing with this:
  - **Open Hashing: Closed Addressing**
    Separate Chaining
  - **Closed Hashing: Open Addressing**
    - Linear Probing
    - Quadratic Probing
    - Double Hashing

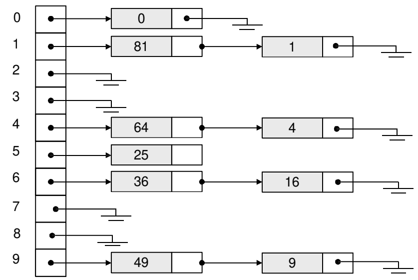
Separate Chaining

- The idea is to keep a list of all elements that hash to the same value.
  - The array elements are pointers to the first nodes of the lists.
  - A new item is inserted to the front of the list.
- Advantages:
  - Better space utilization for large items.
  - Simple collision handling: searching linked list.
  - Overflow: we can store more items than the hash table size.
  - Deletion is quick and easy: deletion from the linked list.

Example

Keys: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81

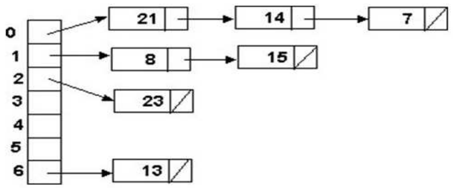
hash(key) = key % 10.



Exercise: Represent the keys {89, 18, 49, 58, 69, 78} in hash table using separate chaining, hash(key) = key % 10.

Example: Load the keys {23, 13, 21, 14, 7, 8, and 15} in this order, in a hash table of size 7 using separate chaining with the hash function: h(key)=key% 7.

h(23) = 23 % 7 = 2
h(13) = 13 % 7 = 6
h(21) = 21 % 7 = 0
h(14) = 14 % 7 = 0 collision
h(7) = 7 % 7 = 0 collision
h(8) = 8 % 7 = 1
h(15) = 15 % 7 = 1 collision



Collision Resolution with Open Addressing

- Separate chaining has the disadvantage of using linked lists.
  - Requires the implementation of a second data structure.
- In an open addressing hashing system, all the data go inside the table.
  - Thus, a bigger table is needed.
  - If a collision occurs, alternative cells are tried until an empty cell is found.

Open Addressing

- In open addressing, there are three common collision resolution strategies:
  - Linear Probing
  - Quadratic Probing
  - Double Hashing

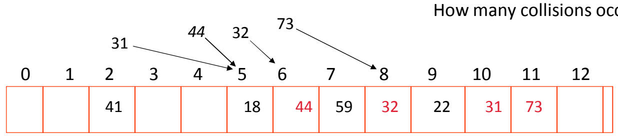
Linear Probing

- Searches the hash table sequentially, starting from the original location specified by the hash function
- Possible problem
  - Primary clustering

Linear Probing Example

- h(k, i) = (h(k) + i) mod m (i is probe number, initially, i = 0)
- **Insert keys: 18 41 22 44 59 32 31 73 (in that order)**

How many collisions occur in this case? **11**



h(k) = k mod 13
m = 13

If a collision occurs, when j = h(k), we try next at A[(j+1)mod m], then A[(j+2)mod m], and so on. When an empty position is found the item is inserted.

Linear probing is easy to implement, but leads to *clustering* (long run of occupied slots). Clusters can lead to poor performance, both for inserting and finding keys.

Another Example: Linear Probing

- **Example: Insert keys {89, 18, 49, 58, 69, 78} with the hash function: h(x)=x mod 10 using linear probing. Use table size 10.**

- **Solution:**
  - **when x=89:**
    h(89)=89%10=9
    insert key 89 in hash-table in location 9
  - **when x=18:**
    h(18)=18%10=8
    insert key 18 in hash-table in location 8

0	
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

Fig. Hash table with keys Using linear probing

h(k) = (h(k) + i) mod m, i= 0 to m-1

- **when x=49:**
  h(49)=49%10=9 **(Collision)**
  so insert key 49 in hash-table in next possible vacant location of 9 is 0
- **when x=58:**
  h(58)=58%10=8 **(Collision)**
  insert key 58 in hash-table in next possible vacant location of 8 is 1 (since 9, 0 already contains values).
- **when x=69:**
  h(69)=69%10=9 **(Collision)**
  insert key 69 in hash-table in next possible vacant location of 9 is 2 (since 0, 1 already contains values).
- **when x = 78:**
  h(78) = 78 % 10 = 8 **(Collision)**
  search next vacant slot in the table which is 3 (since 0,1,2 contain values)
  insert 78 at location 3.

0	49
1	58
2	69
3	78
4	
5	
6	
7	
8	18
9	89

Fig. Hash table with keys Using linear probing

Quadratic Probing:

- Quadratic probing is a collision resolution method that eliminates the primary clustering problem take place in a linear probing.
- Compute: hash value = h(x) = x % table size
- When collision occur then the quadratic probing works as follows:
  (hash value + 1^2)% table size,
- if there is again collision occur then there exist rehashing.
  (hash value + 2^2)%table size
- if there is again collision occur then there exist rehashing.
  (hash value = 3^2)% table size
- In general in i^{th} collision

$$h_i(x)=(hash\ value + i^2)\%size$$

Quadratic Probing

- In general, searches the hash table beginning with the original location that the hash function specifies and continues at increments of 1^2, 2^2, 3^2, and so on
- Possible problem
  - Secondary clustering

Example: Insert keys {89, 18, 49, 58, 69 78} in order with the hash-table size 10 using quadratic probing.   Hash function: h(x)=x mod 10

Solution:

when x=89:
h(89)=89%10=9
insert key 89 in hash-table in location 9

when x=18:
h(18)=18%10=8
insert key 18 in hash-table in location 8

when x=49:
h(49)=49%10=9   (Collision)
so use following hash function,
$h_1(49)=(9 + 1)\%10=0$
hence insert key 49 in hash-table in location 0

when x=58:
h(58)=58%10=8   (Collision)
so use following hash function,
$h_1(58)=(8 + 1)\%10=9$
again collision occur use again the following hash function ,
$h_2(58)=(8+ 2^2)\%10=2$
insert key 58 in hash-table in location 2

0	49
1	
2	58
3	
4	
5	
6	
7	
8	18
9	89

Fig. Hash table with keys Using quadratic probing

- **when x=69:**
  - h(69)=69%10=9   (Collision)
  - so use following hash function, h1(69)=(9 + 1)%10=0
  - again collision occurs use again the following hash function ,
  - h2(69)=(9+ 22)%10=3
  - insert key 69 in hash-table in location 3
- **when x=78:**
  - h(78)=78%10=8   (Collision)
  - so use following hash function, h1(78)=(8 + 1)%10=9 ; again collision occurs
  - use again the following hash function ,
  - h2(78)=(8+ 22)%10=2 ; again collision occurs, compute following step
  - h3(78)=(8+ 32)%10=7
  - insert key 58 in hash-table in location 7

0	49
1	
2	58
3	69
4	
5	
6	
7	78
8	18
9	89

Fig. Hash table with keys Using quadratic probing

Double Hashing

- Uses two hash functions
- Searches the hash table starting from the location that one hash function determines and considers every n^{th} location, where n is determined from a second hash function

Double Hashing Example

- **$h_1(K)$ = K mod m**
- **$h_2(K)$ = K mod (m – 1)**
- The i^{th} probe is $h(k, i) = (h_1(k) + h_2(k) \cdot i) \bmod m$

- **Insert keys: 18 41 22 44 59 32 31 73 (in that order)**

31 → 5, *44* → 5

How many collisions occur in this case? **2**

0	1	2	3	4	5	6	7	8	9	10	11	12
44		41			18	32	59	73	22			31

m = 13

44 % 13 = 5 (collision), next try: (5 + (44 % 12)) % 13 = 13 % 13 = 0

31 % 13 = 5 (collision), next try: (5 + (31 % 12)) % 13 = 12 % 13 = 12