SREENIVASA INSTITUTE OF TECHNOLOGY AND MANAGEMENT STUDIES
(Autonomous)

Subject: Data Structures
I B.Tech, II Sem
Unit 1 Question bank

Short Answer Questions:

1. Define data structure.
Ans. Data structures are the fundamental building blocks of computer programming. They define how data is organized, stored, and manipulated within a program. Understanding data structures is very important for developing efficient and effective algorithms. It is a way of arranging data on a computer so that it can be accessed and updated efficiently. It is also used for processing, retrieving, and storing data. Examples include arrays, linked lists, stacks, and trees.

2. What are the different types of data structures?
Ans. There are two different types of Data Structures: 1. Linear Data Structure, 2. Non-Linear Data Structure.
Linear Data Structure: Elements are arranged in one dimension, also known as linear dimension. Example: lists, stack, queue, etc.
Non-Linear Data Structure: Elements are arranged in one-many, many-one and many-many dimensions. Example: tree, graph, table, etc.

3. Differentiate linear and nonlinear data structures.
Ans. Linear Data Structure:
Data structure where data elements are arranged sequentially or linearly where each and every element is attached to its previous and next adjacent is called a **linear data structure**. In linear data structure, single level is involved. Therefore, we can traverse all the elements in single run only. Linear data structures are easy to implement because computer memory is arranged in a linear way. Its examples are array, stack, queue, linked list, etc.
Non-linear Data Structure:
Data structures where data elements are not arranged sequentially or linearly are called **non-linear data structures**. In a non-linear data structure, single level is not involved. Therefore, we can't traverse all the elements in single run only. Non-linear data structures are not easy to implement in comparison to linear data structure. It utilizes computer memory efficiently in comparison to a linear data structure. Its examples are trees and graphs.

4. List the various operations that can be performed on data structure.
Ans. Various operations that can be performed on the data structure are
- Create
- Insertion of element
- Deletion of element
- Searching for the desired element
- Sorting the elements in the data structure
- Reversing the list of elements
- Copying the elements
- Merging two data structures.

5. Describe abstract data type with example.
Ans. An Abstract Data Type (ADT) is a conceptual model that defines a set of operations and behaviors for a data structure, without specifying how these operations are implemented or how data is organized in memory. The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called "abstract" because it provides an implementation-independent view. Examples of ADTs: 1. Stacks: A linear data structure that follows the LIFO (Last In, First Out) principle. Stacks are commonly implemented using arrays or linked lists.
2. Queues: A linear data structure that allows data to be accessed from both ends, the front and the rear. Queues are a natural component of many simulation models of real processes. Other Examples: List, Map, Set, Table, Tree, and Vector.

6. List out the areas in which data structures are applied extensively.

Ans. Data structures are applied extensively in a wide array of computer science areas, including but not limited to compiler design, operating systems, database management, artificial intelligence, and more.

1. Compiler Design:
   Data structures like parse trees, symbol tables, and abstract syntax trees are vital for efficiently translating source code into executable form.
2. Operating Systems:
   Data structures such as queues, stacks, and priority queues are used for managing processes, memory, and scheduling tasks within an operating system.
3. Database Management Systems:
   Trees (like B-trees), hash tables, and various indexing techniques are crucial for efficient data storage, retrieval, and manipulation within database systems.
4. Algorithms:
   Efficient data structures are essential for designing and implementing fast algorithms, such as those for searching, sorting, and graph traversal.
5. Artificial Intelligence (AI):
   Data structures like decision trees, graphs, and knowledge bases play a vital role in AI algorithms, enabling machines to learn, make decisions, and solve complex problems.

6. What are the advantages of an ADT?

Ans. Advantages of ADT

- Encapsulation: ADTs help encapsulate data and operations into a single unit, making managing and modifying the data structure easier.
- Abstraction: You can work with ADTs without knowing the implementation details, which can simplify programming and reduce errors.
- Data Structure Independence: ADTs can be implemented using different data structures, to make it easier to adapt to changing needs and requirements.
- Information Hiding: ADTs protect data integrity by controlling access and preventing unauthorized modifications.
- Modularity: ADTs can be combined with other ADTs to form more complex data structures increasing flexibility and modularity in programming.

7. What is time complexity of an algorithm?

Ans. Time Complexity: The time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input. Note that the time to run is a function of the length of the input and not the actual execution time of the machine on which the algorithm is running on.

The valid algorithm takes a finite amount of time for execution. The time required by the algorithm to solve given problem is called ***time complexity*** of the algorithm. Time complexity is very useful measure in algorithm analysis.

It is the time needed for the completion of an algorithm. To estimate the time complexity, we need to consider the time required for each fundamental instruction and the number of times the instruction is executed.

8. What is space complexity of an algorithm?

Ans. Space complexity is a measure of the amount of memory an algorithm uses, expressed in terms of the size of the input. It refers to the amount of memory storage required to execute the algorithm and solve a problem. A low space complexity means that an algorithm requires relatively little memory to solve the problem, while a high space complexity means that it requires a large amount of memory, potentially leading to slow performance or memory limitations.

9. Why is searching required? What are the different searching techniques used?

Ans. Searching is required to locate specific data within a collection, and efficient techniques like linear search, binary search, and hashing are crucial for fast and reliable retrieval. Searching algorithms are essential for solving various problems, including finding information in databases, searching for specific items in a list, and locating relevant web pages.

10. What are the drawbacks of using static memory allocation, and how does dynamic memory allocation overcome those limitations?

Ans. Drawbacks of Static Memory Allocation:

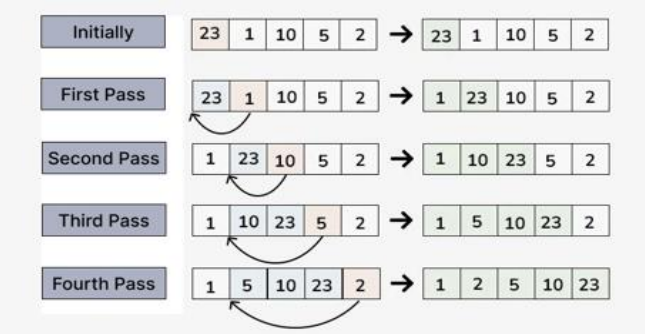
Fixed Size: In static memory allocation, the amount of memory a variable or data structure occupies is

determined at compile time and cannot be changed during runtime.
Potential for Wastage: If you allocate more memory than needed, you waste space. Conversely, if you allocate too little, you'll run into problems when your program needs more memory.
Lack of Flexibility: Static allocation is not suitable for applications where memory requirements can vary during program execution.
Compile-Time Overhead: The compiler needs to know the size of all data structures upfront, which can make the compilation process slower and less flexible.
Dynamic Memory Allocation Overcomes These Limitations:
Runtime Allocation: Dynamic memory allocation allows you to allocate memory during program execution, meaning you can request memory as needed.
Flexibility: You can allocate and deallocate memory as your program's requirements change, making dynamic allocation suitable for applications with variable memory needs.
Efficient Memory Usage: You can allocate only the amount of memory you need, reducing wastage.
Data Structures like Linked Lists: Dynamic memory allocation is essential for implementing data structures like linked lists, where the size of the data structure can change during runtime.
Dynamic memory allocation is typically managed using functions like malloc(), calloc(), realloc() and free().

11. What is the process of insertion sort, and how is it used to sort a collection of data elements in ascending or descending order?

Ans. Insertion sort is a simple sorting algorithm that works by iteratively inserting each element of an unsorted list into its correct position in a sorted portion of the list. It is like sorting playing cards in your hands. You split the cards into two groups: the sorted cards and the unsorted cards. Then, you pick a card from the unsorted group and put it in the right place in the sorted group.

- We start with second element of the array as first element in the array is assumed to be sorted.
- Compare second element with the first element and check if the second element is smaller then swap them.
- Move to the third element and compare it with the first two elements and put at its correct position
- Repeat until the entire array is sorted.



12. What is the process of linear search in programming, and how is it used to search for a specific element in a collection of data?

Ans. Linear search in programming is a straightforward algorithm that sequentially checks each element in a collection (like an array or list) until a match for the target element is found or the end of the collection is reached.

Search Process:

- The search starts at the beginning of the collection.
- It compares each element in the collection with the target element you are searching for.
- If a match is found, the search stops, and the index (position) of the matched element is returned.
- If the end of the collection is reached without finding a match, the search returns a value indicating that the target element was not found (e.g., -1).

13. What is the process of binary search in programming, and how is it used to search for a specific element in a collection of data?

Ans. Binary search is an efficient algorithm for finding an element in a sorted collection (like an array) by repeatedly dividing the search interval in half, comparing the target value with the middle element, and narrowing the search space until the element is found or the interval is empty.

explanation:

1. Sorted Collection: Binary search requires the data to be sorted in ascending or descending order.
2. Identify the Middle Element: Determine the middle element of the current search interval (initially the entire array).
3. Compare: Compare the target value with the middle element:

If the target value matches the middle element: The search is successful, and the index of the middle element is returned.

If the target value is less than the middle element: The search continues in the left half of the array (excluding the middle element).

If the target value is greater than the middle element: The search continues in the right half of the array (excluding the middle element).

4. Repeat: Repeat steps 2 and 3 with the new search interval until the target value is found or the search interval becomes empty.
5. Not Found: If the search interval becomes empty without finding the target value, the element is not present in the collection, and a "not found" indicator (e.g., -1) is returned.

14. What is the process of selection sort, and how is it used to sort a collection of data elements in ascending or descending order?

Ans. Selection sort is a simple comparison-based sorting algorithm that sorts a collection of data elements by repeatedly finding the minimum element (for ascending order) or maximum element (for descending order) from the unsorted part and swapping it with the first element of the unsorted part.

explanation:

- Divide and Conquer: Selection sort conceptually divides the input list into two parts: a sorted portion and an unsorted portion.
- Find the Minimum/Maximum: In each iteration, it finds the minimum (ascending) or maximum (descending) element within the unsorted portion.
- Swap: This minimum/maximum element is then swapped with the first element of the unsorted portion, effectively extending the sorted portion by one element.
- Repeat: This process is repeated until the entire list is sorted.

15. What is the process of bubble sort, and how is it used to sort a collection of data elements in ascending or descending order?

Ans. Bubble sort is a simple sorting algorithm that works by repeatedly stepping through the list, comparing adjacent elements and swapping them if they are in the wrong order, until the list is sorted. This process can be used to sort data elements in either ascending or descending order.

Here's a more detailed explanation:

Process of Bubble Sort:

Comparison: The algorithm starts by comparing the first two elements in the list.

Swapping (if needed): If the elements are in the wrong order (e.g., larger element before smaller element for ascending order), they are swapped.

Iteration: The algorithm then moves to the next pair of adjacent elements and repeats the comparison and swapping process.

Pass: This process continues until the end of the list, completing one "pass".

Repeat: The algorithm then repeats the passes until no more swaps are needed, indicating that the list is sorted.

Ascending/Descending Order: The direction of the comparison (swapping if the left element is greater than the right for ascending or the reverse for descending) determines whether the list is sorted in ascending or descending order.

16. What do you mean by sorting? Mention the types of sorting.

Ans. Sorting is a fundamental operation in computer science that involves arranging a set of data in a specific order, such as numerical or alphabetical order.

This ordered arrangement can be either ascending (smallest to largest) or descending (largest to smallest). Sorting is crucial for many algorithms and data structures, as it simplifies searching and other operations.

Types of Sorting Algorithms:
Comparison-based sorting: These algorithms compare elements to determine their relative order.
Bubble Sort: Iteratively compares adjacent elements and swaps them if they are in the wrong order.
Selection Sort: Finds the minimum (or maximum) element and places it at the beginning of the sorted portion of the list.
Insertion Sort: Iteratively inserts each element into its correct position within the sorted portion of the list.
Merge Sort: Divides the list into smaller sublists, sorts them, and then merges them back together.
Quick Sort: Selects a pivot element and partitions the list around it, recursively sorting the partitions.

17. What do you mean by searching? Mention the types of searching.

Ans. Searching is the process of finding a given value position in a list of values. It decides whether a search key is present in the data or not. It is the algorithmic process of finding a particular item in a collection of items. It can be done on internal data structure or on external data structure.
Types of Searching Algorithms:
Linear Search (or Sequential Search):It sequentially checks each element in the data structure, starting from the beginning, until the target element is found or the end is reached. It is best for small, unsorted datasets.
Binary Search: It repeatedly divides the search interval in half, comparing the target element with the middle element of the interval. It requires the data to be sorted, and is much faster than linear search for large, sorted datasets.

18. Differentiate Linear Search and Binary search.

Linear Search	Binary Search
In linear search input data need not to be in sorted.	In binary search input data need to be in sorted order.
It is also called sequential search.	It is also called half-interval search.
The time complexity of linear search O(n).	The time complexity of binary search O(log n).
Multidimensional array can be used.	Only single dimensional array is used.
Linear search performs equality comparisons	Binary search performs ordering comparisons
It is less complex.	It is more complex.
It is very slow process.	It is very fast process.

19. Differentiate bubble sort, and selection sort.

S.No.	Bubble Sort	Selection Sort
1.	Bubble sort is a simple sorting algorithm which continuously moves through the list and compares the adjacent pairs for proper sorting of the elements.	Selection sort is a sorting algorithm which takes either smallest value (ascending order) or largest value (descending order) in the list and place it at the proper position in the list.
2.	Bubble sort compares the adjacent elements and move accordingly.	Selection sort selects the smallest element from the unsorted list and moves it at the next position of the sorted list.
3.	Bubble sort performs a large number of swaps or moves to sort the list.	Selection sort performs comparatively less number of swaps or moves to sort the list.
4.	Bubble sort is relatively slower.	Selection sort is faster as compared to bubble sort.
5.	The efficiency of the bubble sort is less.	The efficiency of the selection sort is high.
6.	Bubble sort performs sorting of an array by exchanging elements.	Selection sort performs sorting of a list by the selection of element.
7.	Time complexity: O(n^2) in the worst and average cases, O(n) in the best case (when the input array is already sorted) Space complexity: O(1)	Time complexity: O(n^2) in all cases (worst, average, and best) Space complexity: O(1)

20. Differentiate bubble sort, and insertion sort.

Bubble sort	Insertion sort
Bubble sort compares adjacent elements and swaps them if they are in the wrong order. This process is repeated until no more swaps are needed, indicating that the	Insertion sort builds a sorted array one element at a time. It iterates through the array, picking up an element and inserting it into its correct position within the sorted

	list is sorted. Time Complexity: Best Case: O(n) (already sorted array) Average Case: $O(n^2)$ Worst Case: $O(n^2)$ (reverse sorted array) Space Complexity: O(1) (in-place sorting) Stability: Stable (preserves the relative order of equal elements) Efficiency: Generally less efficient than insertion sort, especially for larger datasets.	portion. Time Complexity: Best Case: O(n) (already sorted array) Average Case: $O(n^2)$ Worst Case: $O(n^2)$ (reverse sorted array) Space Complexity: O(1) (in-place sorting) Stability: Stable (preserves the relative order of equal elements) Efficiency: Generally more efficient than bubble sort for smaller datasets and nearly sorted datasets.	

Long Answers Questions

1. Write C program to perform searching operation using linear and binary search.

Linear Search:

```
#include <stdio.h>
int main()
{
  int array[100], search, c, n;

  printf("Enter number of elements in array\n");
  scanf("%d", &n);
  printf("Enter %d integer(s)\n", n);
  for (c = 0; c < n; c++)
    scanf("%d", &array[c]);
  printf("Enter a number to search\n");
  scanf("%d", &search);
  for (c = 0; c < n; c++)
  {
    if (array[c] == search)    /* If required element is found */
    {
      printf("%d is present at location %d.\n", search, c+1);
      break;
    }
  }
  if (c == n)
    printf("%d isn't present in the array.\n", search);
  return 0;
}
```

Output:

```
Enter number of elements in array
5
Enter 5 integer(s)
11
36
68
94
54
Enter a number to search
94
94 is present at location 4.
```

Binary Search:

```
// Binary Search in C

#include <stdio.h>
int binarySearch(int array[], int x, int low, int high) {
  // Repeat until the pointers low and high meet each other
  while (low <= high) {
    int mid = low + (high - low) / 2;
    if (x == array[mid])
      return mid;
    if (x > array[mid])
      low = mid + 1;
    else
      high = mid - 1;
  }

  return -1;
}

int main(void) {
  int array[] = {3, 4, 5, 6, 7, 8, 9};
  int n = sizeof(array) / sizeof(array[0]);
  int x = 4;
  int result = binarySearch(array, x, 0, n - 1);
  if (result == -1)
    printf("Not found");
  else
    printf("Element is found at index %d", result);
  return 0;
}
```

Output:
Element is found at index 1

2. What is meant by time complexity and space complexity? How do we analyze the time complexity and space complexity of linear data structures?

Ans. Linear Data Structures

Linear Data structure are those data structures in which data is stored sequentially, and each element is connected to the previous or next element so that they can be accessed in a single run. Some examples of linear data structures are arrays, stacks, queues, etc.

Let us discuss their time and space complexity.

Time Complexity

Time complexity can be understood as a concept that constitutes the quantification of time taken by an algorithm or code snippet to execute. The time complexity is also a measure of the efficiency, such that the lesser the time is taken by the algorithm, the more its efficiency will be.

We will be discussing only the worst-case time complexity in this article.

Space Complexity

Space Complexity can be understood as the amount of memory space occupied by a code snippet or algorithm. It is one of the two measures of the efficiency of an algorithm. The lesser the space it takes, the more efficient it is.

Now let's discuss some linear data structures.

Array

A Data Structure in which a similar type of data is stored at contiguous memory locations is called an array. Array is the most frequently used linear data structure in computer science and due to its connectivity with the previous and next element, it became the inspiration for data structures like linked lists, queues, etc.

Queue

A queue can be defined as a linear data structure, which is simply a collection of entries that are tracked in order,

such that the addition of entries happens at one end of the queue, while the removal of entries takes place from the other end. Its order is also known as First In First Out (FIFO).

Stack

A stack is a linear data structure, following a particular order in which operations can be performed. Its order is also known as LIFO(Last In First Out). Stacks are implemented using arrays and linked lists. Let us discuss its efficiency in terms of time and space complexity.

Linked List

A linked list can be understood as a data structure that consists of nodes to store data. Each node consists of a data field and a pointer to the next node. The various elements in a linked list are linked together using pointers. In Linked List, unlike arrays, elements are not stored at contiguous memory locations but rather at different memory locations.

Data Structure	Insert	Delete	Access	Search
Array	O(N)	O(N)	O(1)	O(N)
String	O(N)	O(N)	O(1)	O(N)
Queue	O(1)	O(1)	O(N)	O(N)
Stack	O(1)	O(1)	O(N)	O(N)
Linked List	O(1), if inserted on head O(N), elsewhere	O(1), if deleted on head O(N), elsewhere	O(N)	O(N)

Where 'N' is the size of the respective data structure.

Space Complexity of Linear Data Structures:

Space complexity for each operation in a linked list linear data structures is **O(1)**, as no extra space is required for any operation.

3. Differentiate Linear and Nonlinear data structures

Factor	Linear Data Structure	Non-Linear Data Structure
Data Element Arrangement	In a linear data structure, data elements are sequentially connected, allowing users to traverse all elements in one run.	In a non-linear data structure, data elements are hierarchically connected, appearing on multiple levels.
Implementation Complexity	Linear data structures are relatively easier to implement.	Non-linear data structures require a higher level of understanding and are more complex to implement.
Levels	All data elements in a linear data structure exist on a single level.	Data elements in a non-linear data structure span multiple levels.
Traversal	A linear data structure can be traversed in a single run.	More complex, requiring specialized algorithms like depth-first search or breadth-first search
Memory Utilization	Linear data structures do not efficiently utilize memory.	Non-linear data structures are more memory-friendly.
Time Complexity	The time complexity of a linear data structure is directly proportional to its size, increasing as input size increases.	The time complexity of a non-linear data structure often remains constant, irrespective of its input size.
Applications	Linear data structures are ideal for application software development.	Non-linear data structures are commonly used in image processing and Artificial Intelligence.
Examples	Linked List, Queue, Stack, Array.	Tree, Graph, Hash Map.

4. Write a program to reverse an array.

Ans.

```
#include <stdio.h>

void reverseArray(int arr[], int size) {
   int temp[size]; // Temporary array
   int j = 0;

   // Copy elements in reverse order
   for (int i = size - 1; i >= 0; i--) {
      temp[j++] = arr[i];
   }

   // Copy reversed elements back to the original array
   for (int i = 0; i < size; i++) {
      arr[i] = temp[i];
   }
}

int main() {
   int arr[] = {1, 2, 3, 4, 5};
   int size = sizeof(arr) / sizeof(arr[0]);

   // Display the original array
   printf("Original array: ");
   for (int i = 0; i < size; i++) {
      printf("%d ", arr[i]);
   }
   printf("");

   // Reverse the array
   reverseArray(arr, size);

   // Display the reversed array
   printf("Reversed array: ");
   for (int i = 0; i < size; i++) {
      printf("%d ", arr[i]);
   }
   printf("");
   return 0;
}
```

Output:

Original array: 1 2 3 4 5

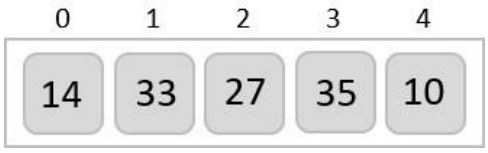
Reversed array: 5 4 3 2 1

5. What is sorting? Explain and write an algorithm for bubble sorting and trace the algorithm with an example.

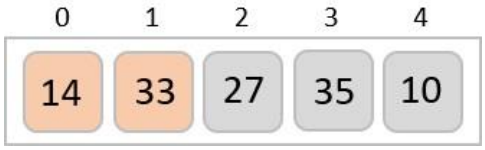
In computer science, a sorting algorithm is an algorithm that puts elements of a list into an order. The most frequently used orders are numerical order and lexicographical order, and either ascending or descending. Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based 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 works by repeatedly exchanging adjacent elements, if necessary. When no exchanges are required, the file is sorted.
Algorithm:
We assume **list** is an array of **n** 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.

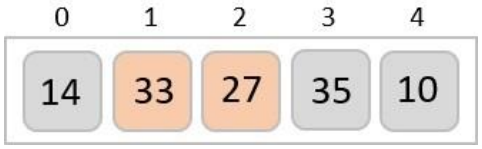
Example
We take an unsorted array for our example. Bubble sort takes O(n2) time so we're keeping it short and precise.



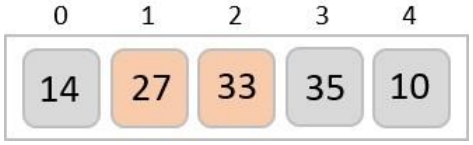
Bubble sort starts with very first two elements, comparing them to check which one is greater.



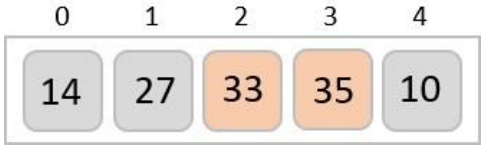
In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.



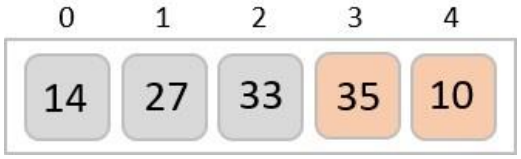
We find that 27 is smaller than 33 and these two values must be swapped.



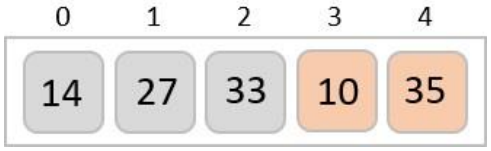
Next we compare 33 and 35. We find that both are in already sorted positions.



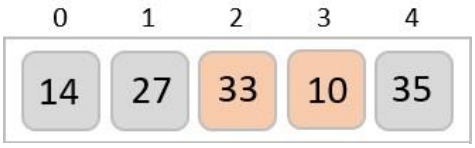
Then we move to the next two values, 35 and 10.

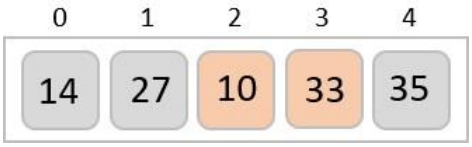


We know then that 10 is smaller 35. Hence they are not sorted. We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −

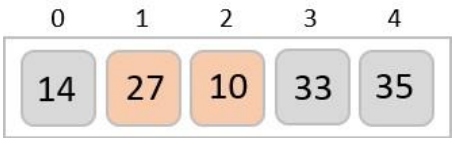


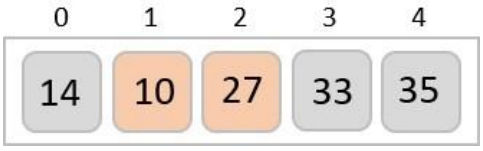
To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −

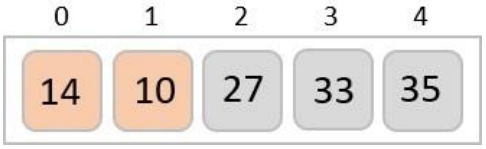


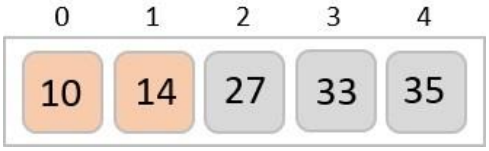


Notice that after each iteration, at least one value moves at the end.

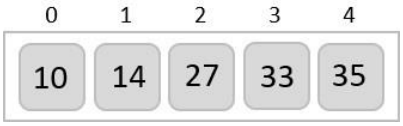








And when there's no swap required, bubble sort learns that an array is completely sorted.



6. Describe the Insertion Sort algorithm, trace the steps for sorting the following list, and derive the time complexity.

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 $O(n^2)$, 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.

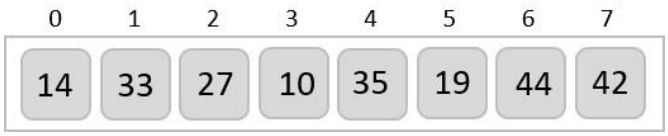
Step 1 − If it is the first element, it is already sorted. return 1;

Step 2 − Pick next element

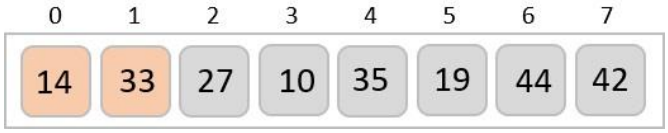
Step 3 − Compare with all elements in the sorted sub-list
Step 4 − Shift all the elements in the sorted sub-list that is greater than the value to be sorted
Step 5 − Insert the value
Step 6 − Repeat until list is sorted

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.

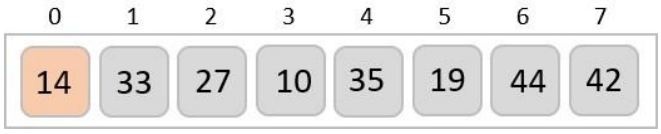
Example
We take an unsorted array for our example.



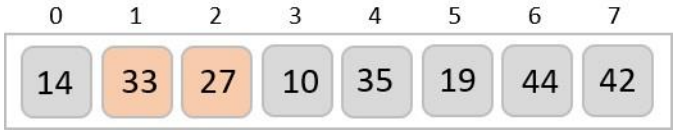
Insertion sort compares the first two elements.



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

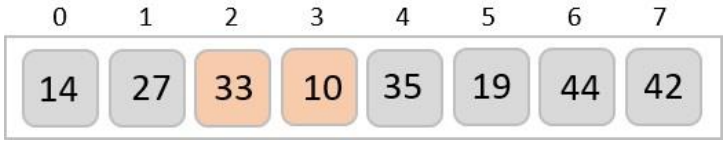


Insertion sort moves ahead and compares 33 with 27.



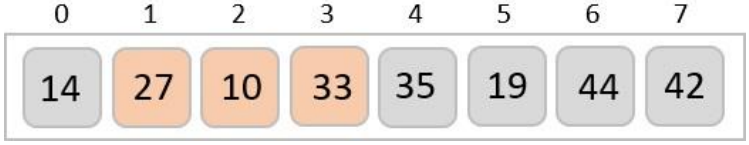
And finds that 33 is not in the correct position. It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.

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.

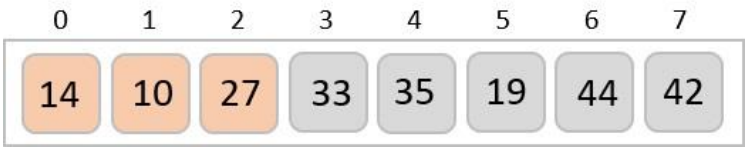


So they are swapped.

However, swapping makes 27 and 10 unsorted.

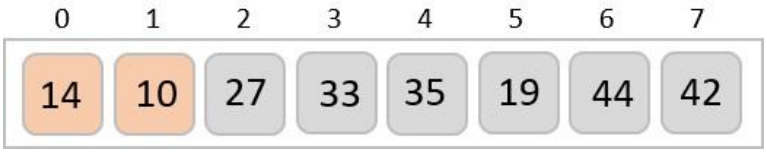


Hence, we swap them too.

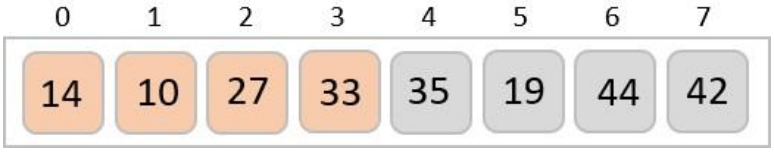


Again we find 14 and 10 in an unsorted order.

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.

7. Describe the selection Sort algorithm, trace the steps for sorting the following list, and derive the time complexity.

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.

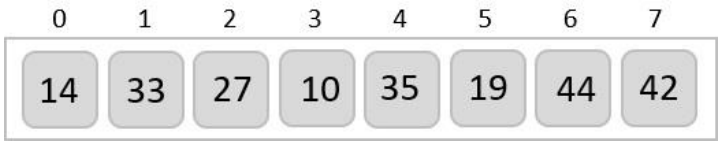
2. Search the minimum element in the list.

3. Swap with value at location MIN.

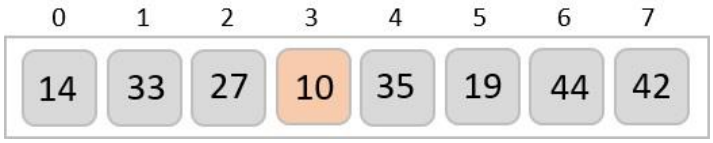
4. Increment MIN to point to next element.

5. Repeat until the list is sorted.

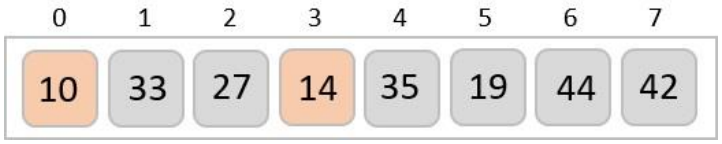
Example
Consider the following depicted array as an example.



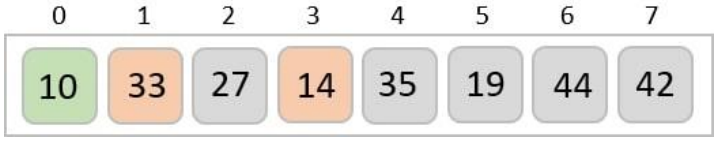
For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.



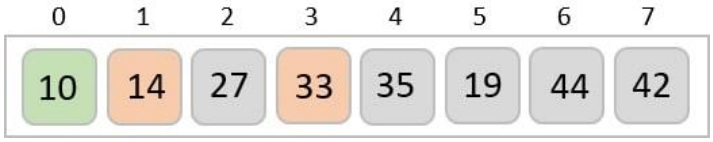
So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.



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

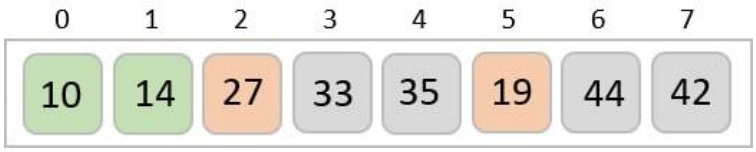


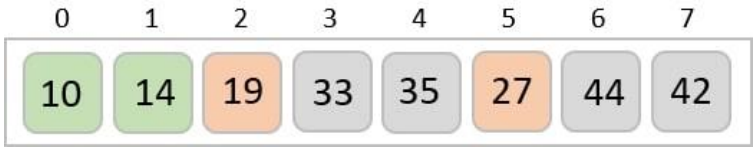
We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.

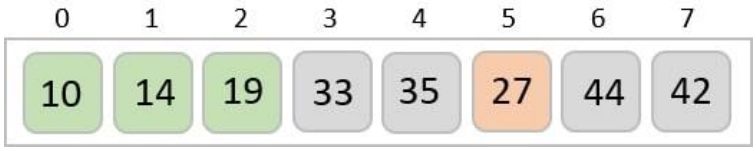


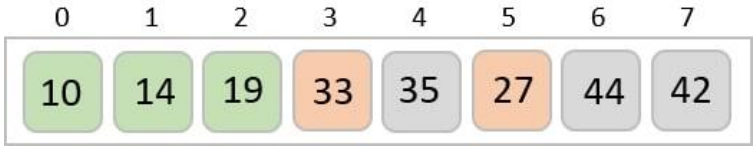
After two iterations, two least values are positioned at the beginning in a sorted manner.

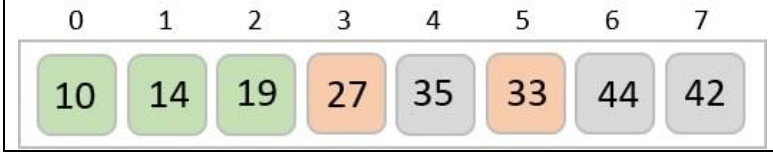
The same process is applied to the rest of the items in the array −

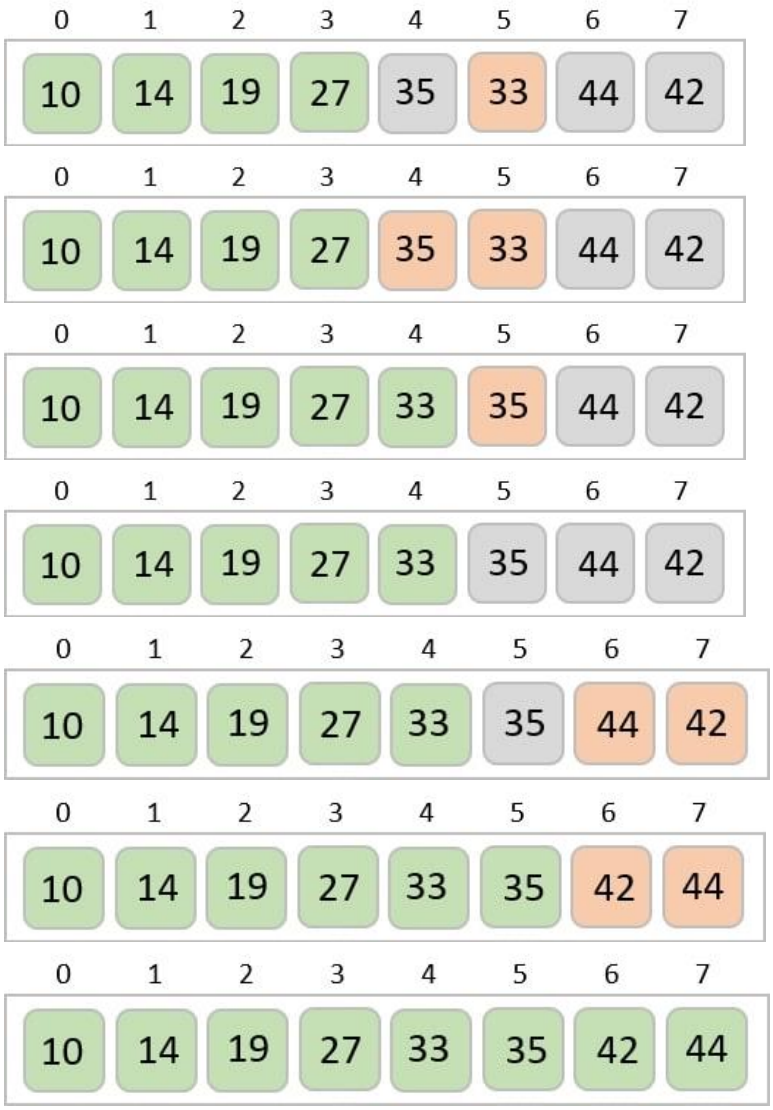












8. Write C program to perform bubble sort.

```
// Bubble sort in C

#include <stdio.h>

// perform the bubble sort
void bubbleSort(int array[], int size) {

  // loop to access each array element
  for (int step = 0; step < size - 1; ++step) {

    // loop to compare array elements
    for (int i = 0; i < size - step - 1; ++i) {

      // compare two adjacent elements
      // change > to < to sort in descending order
      if (array[i] > array[i + 1]) {

        // swapping occurs if elements
        // are not in the intended order
        int temp = array[i];
```

```
        array[i] = array[i + 1];
        array[i + 1] = temp;
      }
    }
  }
}

// print array
void printArray(int array[], int size) {
  for (int i = 0; i < size; ++i) {
    printf("%d  ", array[i]);
  }
  printf("\n");
}

int main() {
  int data[] = {-2, 45, 0, 11, -9};

  // find the array's length
  int size = sizeof(data) / sizeof(data[0]);

  bubbleSort(data, size);

  printf("Sorted Array in Ascending Order:\n");
  printArray(data, size);
}
```

Output:
Sorted Array in Ascending Order:
-9 -2 0 11 45

9. What is data structure? Explain its role in problem solving? Explain the different types of data structure with suitable examples.

Ans. A data structure is a method of organizing and storing data, enabling efficient access and manipulation. It plays a crucial role in problem-solving by optimizing data organization and retrieval, leading to faster and more efficient algorithms.

Role in Problem Solving:

Efficiency:

Data structures are designed to optimize operations like searching, insertion, deletion, and sorting, leading to faster and more efficient algorithms.

Organization:

They provide a systematic way to store and manage large amounts of data, making it easier to access and manipulate.

Algorithm Design:

The choice of data structure can significantly impact the performance and complexity of algorithms, making it a crucial aspect of problem-solving.

Code Reusability:

Well-designed data structures can be reused across different projects, saving time and effort in software development.

Types of Data Structures:

1. Linear Data Structures:

Arrays:

A collection of elements stored in contiguous memory locations, allowing for direct access to any element using an index.

Example: An array to store a list of student IDs: [101, 102, 103, 104].

Linked Lists:

A collection of nodes, where each node contains data and a pointer to the next node, allowing for dynamic allocation and insertion/deletion.

Example: A linked list to store a list of names: Node(John) -> Node(Mary) -> Node(Peter).

Stacks:

A linear data structure where elements are added and removed in a Last-In, First-Out (LIFO) manner.
Example: A stack to store the history of visited web pages in a browser.

Queues:

A linear data structure where elements are added and removed in a First-In, First-Out (FIFO) manner.
Example: A queue to manage a list of tasks to be processed.

2. Non-Linear Data Structures:

Trees:

A hierarchical data structure where elements are organized in a parent-child relationship.
Example: A file system directory structure.

Graphs:

A data structure that represents relationships between entities, consisting of nodes (vertices) and edges.
Example: A social network where users are nodes and connections are edges.

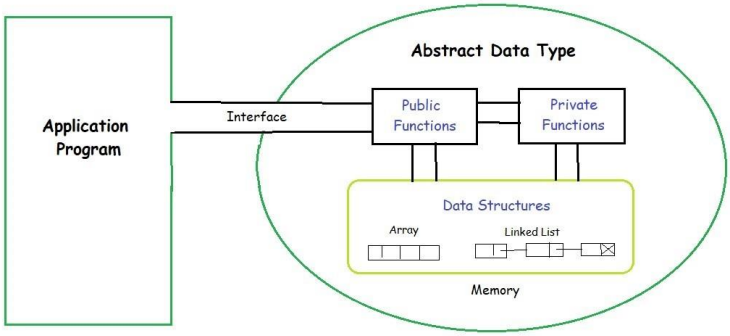
Hash Tables:

A data structure that uses a hash function to map keys to values, allowing for fast retrieval of data.
Example: A dictionary where words are keys and their meanings are values.

10. Explain about abstract data types with examples.

Ans. An Abstract Data Type (ADT) is a conceptual model that defines a set of operations and behaviors for a data structure, without specifying how these operations are implemented or how data is organized in memory. The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called "abstract" because it provides an implementation-independent view.

The process of providing only the essentials and hiding the details is known as abstraction.



The above figure shows the ADT model. There are two types of models in the ADT model, i.e., the public function and the private function. The ADT model also contains the data structures that we are using in a program. In this model, first encapsulation is performed, i.e., all the data is wrapped in a single unit, i.e., ADT. Then, the abstraction is performed means showing the operations that can be performed on the data structure and what are the data structures that we are using in a program.

Examples of ADTs

Now, let's understand three common ADT's: List ADT, Stack ADT, and Queue ADT.

1. List ADT

The List ADT (Abstract Data Type) is a sequential collection of elements that supports a set of operations without specifying the internal implementation. It provides an ordered way to store, access, and modify data.

The List ADT need to store the required data in the sequence and should have the following operations**:**

- **get():** Return an element from the list at any given position.
- **insert():** Insert an element at any position in the list.
- **remove():** Remove the first occurrence of any element from a non-empty list.
- **removeAt():** Remove the element at a specified location from a non-empty list.
- **replace():** Replace an element at any position with another element.
- **size():** Return the number of elements in the list.
- **isEmpty():** Return true if the list is empty; otherwise, return false.
- **isFull():** Return true if the list is full; otherwise, return false.

2. Stack ADT

The Stack ADT is a linear data structure that follows the LIFO (Last In, First Out) principle. It allows elements to be added and removed only from one end, called the top of the stack.
In Stack ADT, the order of insertion and deletion should be according to the FILO or LIFO Principle. Elements are inserted and removed from the same end, called the top of the stack. It should also support the following operations:

- **push():** Insert an element at one end of the stack called the top.
- **pop():** Remove and return the element at the top of the stack, if it is not empty.
- **peek():** Return the element at the top of the stack without removing it, if the stack is not empty.
- **size():** Return the number of elements in the stack.
- **isEmpty():** Return true if the stack is empty; otherwise, return false.
- **isFull():** Return true if the stack is full; otherwise, return false.

3. Queue ADT
The Queue ADT is a linear data structure that follows the FIFO (First In, First Out) principle. It allows elements to be inserted at one end (rear) and removed from the other end (front).
The Queue ADT follows a design similar to the Stack ADT, but the order of insertion and deletion changes to FIFO. Elements are inserted at one end (called the rear) and removed from the other end (called the front). It should support the following operations:

- **enqueue():** Insert an element at the end of the queue.
- **dequeue():** Remove and return the first element of the queue, if the queue is not empty.
- **peek():** Return the element of the queue without removing it, if the queue is not empty.
- **size():** Return the number of elements in the queue.
- **isEmpty():** Return true if the queue is empty; otherwise, return false.

Features of ADT
Abstract data types (ADTs) are a way of encapsulating data and operations on that data into a single unit. Some of the key features of ADTs include:

- **Abstraction:** The user does not need to know the implementation of the data structure only essentials are provided.
- **Better Conceptualization:** ADT gives us a better conceptualization of the real world.
- **Robust:** The program is robust and has the ability to catch errors.
- **Encapsulation**: ADTs hide the internal details of the data and provide a public interface for users to interact with the data. This allows for easier maintenance and modification of the data structure.
- **Data Abstraction**: ADTs provide a level of abstraction from the implementation details of the data. Users only need to know the operations that can be performed on the data, not how those operations are implemented.
- **Data Structure Independence**: ADTs can be implemented using different data structures, such as arrays or linked lists, without affecting the functionality of the ADT.
- **Information Hiding:** ADTs can protect the integrity of the data by allowing access only to authorized users and operations. This helps prevent errors and misuse of the data.
- **Modularity**: ADTs can be combined with other ADTs to form larger, more complex data structures. This allows for greater flexibility and modularity in programming.

11. **Write c program for selection sort and insertion sort**.

Ans. C Program to Implement Selection Sort Algorithm

```
// C program for implementation of selection sort
#include <stdio.h>

void selectionSort(int arr[], int N) {

  // Start with the whole array as unsored and one by
   // one move boundary of unsorted subarray towards right
  for (int i = 0; i < N - 1; i++) {

    // Find the minimum element in unsorted array
    int min_idx = i;
    for (int j = i + 1; j < N; j++) {
```

```
            if (arr[j] < arr[min_idx]) {
                min_idx = j;
            }
        }

        // Swap the found minimum element with the first
        // element in the unsorted part
        int temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}

int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int N = sizeof(arr) / sizeof(arr[0]);
    printf("Unsorted array: \n");
    for (int i = 0; i < N; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

     // Calling selection sort
    selectionSort(arr, N);

    printf("Sorted array: \n");
    for (int i = 0; i < N; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
    return 0;
}
```

Output

```
Unsorted array:

64 25 12 22 11

Sorted array:

11 12 22 25 64
```

UNIT - 2
DATA STRUCTURE
QUESTION BANK

1) Define Linked list?

Ans). A **Linked List** is a **linear data structure** that consists of a **sequence of elements (nodes)**, where each node contains two parts:

1. **Data:** The value stored in the node.
2. **Pointer (Next):** A reference to the next node in the sequence.

2) What are the different types of linked list?

Ans). In **data structures**, linked lists are categorized into the following main types:

1. **Singly Linked List (SLL)**
2. Doubly Linked List (DLL)
3. Circular Linked List (CLL)
4. Circular Doubly Linked List

3) Define the Singly list?

A **Singly Linked List** consists of nodes where:

- Each node contains **two parts**:
  Data → Stores the value.
  Next pointer → Points to the next node.
- The **last node** points to NULL, indicating the end of the list.
- **Traversal is one-way** (from head to tail).

Structure and Example

[Data|Next] → [Data|Next] → [Data|Next] → NULL

4) Define the Doubly linked list?

A **Doubly Linked List** consists of nodes where:

- Each node contains **three parts**:
  - **Prev pointer** → Points to the previous node.
  - **Data** → Stores the value.
  - **Next pointer** → Points to the next node.
- Allows **bi-directional traversal**.

Structure and Example

NULL ← [Prev|Data|Next] ↔ [Prev|Data|Next] → NULL

5) Define the Circular Linked list?

A **Circular Linked List** is a variation where:

- The **last node** points back to the **first node**, forming a circle.
- Can be either **singly circular** or **doubly circular**.

Structure and Example

[Data|Next] → [Data|Next] → [Data|Next] → (points back to first node)

6) Singly linked list advantages and dis advantages?

Advantages:

7) Simple and easy to implement.
8) Efficient for **insertion and deletion** operations.

Disadvantages:

9) Cannot traverse backward.
10) Accessing a specific node requires traversal from the beginning.

7) Doubly Linked List advantages and dis advantages?

Advantages:

- **Bi-directional traversal** (both forward and backward).
- Easier to **delete nodes** (as it contains a reference to the previous node).

Disadvantages:

- Uses **extra memory** for storing an additional pointer.
- Slightly more complex compared to singly linked lists.

8) Circular linked list advantage and dis advantages?

Advantages:

- Can be traversed **continuously** in a loop.
- Useful for **round-robin scheduling**.

Disadvantages:

- More complex to implement.
- Risk of infinite loops if not handled properly.

9). How can represent in singly linked list?

Representation of Singly Linked List in Data Structure

In a **Singly Linked List (SLL)**:

- Each **node** contains two parts:
    - **Data** → Holds the value of the node.
    - **Next Pointer** → Points to the **next node** in the list.
- The **last node** points to NULL, indicating the end of the list.

10). How can represent in doubly linked list?

Representation of Doubly Linked List in Data Structure

A **Doubly Linked List (DLL)** is a **linear data structure** where:

- Each **node** contains **three parts**:
    1. **Prev pointer** → Points to the **previous node**.
    2. **Data** → Holds the value.
    3. **Next pointer** → Points to the **next node**.
- The **first node's previous pointer** is NULL (start of the list).
- The **last node's next pointer** is NULL (end of the list).
- It allows **bi-directional traversal**.

11). How can represent in circular linked list?

Representation of Circular Linked List in Data Structure

A **Circular Linked List (CLL)** is a **linear data structure** where:

- Each **node** contains **two parts**:
    1. **Data** → Holds the value.
    2. **Next pointer** → Points to the **next node** in the list.
- The **last node's next pointer** points to the **first node**, forming a **circular connection**.
- It can be:
    - **Singly Circular Linked List** → Only next pointer exists.
    - **Doubly Circular Linked List** → Both next and prev pointers exist.

12). What are the operation of doubly linked list?

A **Doubly Linked List (DLL)** supports various operations, including:

- **Insertion**
- **Deletion**
- **Traversal**
- **Searching**
- **Reversing the list**

13). Difference between array and linked list?

Array	Linked List
A **fixed-size** data structure that stores elements in **contiguous memory locations**.	A **dynamic-size** data structure consisting of **nodes** with data and pointers.
static allocation: Memory is allocated at **compile-time**.	**Dynamic allocation**: Memory is allocated at **runtime**.
Fixed size → Cannot be easily resized.	**Dynamic size** → Can grow or shrink in memory.
Direct access using the index → O(1) time complexity.	**Sequential access** → Traverses nodes one by one O(n).
Faster → Random access is possible.	**Slower** → Sequential access only.

14). Applications of linked list?

1.Dynamic Memory Management
2. Implementing Stack and Queue
3. Implementing Graphs
4. Polynomial Manipulation
5. Circular Lists in Gaming

15). illustrate the use of the linked list?

Ans). To **illustrate** the use of linked lists, let's go through **real-world scenarios** and visualize how they work.

1). Linked List in Memory Management (Operating System).
2). Music Playlist Navigation.
3). Web Browser History.
4). Stack Implementation Using Linked List.
5). Graph Representation Using Linked List.

16). How can insert the elements in doubly linked list?

Ans). Insertion of Elements in Doubly Linked List (DLL)

In a **doubly linked list**, each node contains:

- **Data**
- A pointer to the **next node**
- A pointer to the **previous node**

Insertion operations in a **Doubly Linked List** can be performed in three ways:

1. **At the beginning**
2. **At the end**
3. **At a specific position.**

17). Difference between singly linked list and doubly linked list?

Singly Linked list	Doubly Linked list
A list where each node contains **data** and a pointer to the **next node**.	A list where each node contains **data**, a pointer to the **next node**, and a pointer to the **previous node**..
Each node contains: data → next.	Each node contains: data → next → prev
One-directional → Can be traversed only forward**.**	Two-directional → Can be traversed forward and backward.
Requires less memory → Only one pointer per node.	Requires more memory → Two pointers per node.
Simpler to implement due to only one pointer.	More complex to implement due to two pointers.

18). Difference between array and circular list?

Array	Circular Linked list
A collection of **contiguous memory locations**.	A linked list where the **last node points to the first node**, forming a **circular structure**.
Static memory allocation → Fixed size.	**Dynamic** memory allocation → Flexible size.
Requires **continuous memory** blocks.	Nodes can be scattered in **different memory locations**.
Linear traversal → Moves from the first to the last element.	**Circular** traversal → Moves continuously in a loop.
The last element points to **NULL**.	The last node points to the **first node**.

5- & 10-MARKS QUESTIONS

DATA STRUCTURE
UNIT – 2

1). Application of Linked list?
Ans). Here are some of the key applications:

1.Dynamic Memory Allocation

- **Linked lists** are used for dynamic memory management, where memory is allocated or deallocated during program execution.
- In **C language**, malloc () and free () functions internally use linked lists to manage the heap memory.

2. Implementation of Stack and Queue

- **Stacks** can be implemented using a **singly linked list**.
- **Queues** are often implemented using a **doubly linked list** for efficient insertion and deletion operations at both ends.
- **Dequeue (Double-ended queue)** can be easily implemented using doubly linked lists.

3.Managing Directories and File Systems

- **Operating systems** use linked lists to manage files and directories.
- Each file or directory is represented as a node containing the name and the address of the next file or directory.
- **File Allocation Table (FAT)** in operating systems uses linked lists for memory allocation.

4. Implementing Hash Tables

- **Chaining in hash tables** uses linked lists to handle collisions.
- When multiple keys hash to the same index, a linked list is used to store the keys at that index.

5. Undo and Redo Operations

- Applications like **text editors** use linked lists to implement **undo and redo** operations.
- Each change is stored as a node in the linked list, allowing you to traverse back and forth.

6.Graph Representation

- **Adjacency lists** in graphs use linked lists to store the neighbours of each node.
- This is an efficient way to represent sparse graphs.

7.Polynomial Arithmetic

- Linked lists are used to represent and manipulate **polynomials**.
- Each node represents a term with coefficients and exponents.

2) Explain the Singly linked list?
Ans). A **Singly Linked List (SLL)** is a type of linked list where:

- Each **node** contains **two parts**:
  - **Data**: The actual value stored in the node.
  - **Pointer (Next)**: The address of the next node in the sequence.
- The **last node** points to NULL, indicating the end of the list.

Structure of a Singly Linked List
[Data | Next] → [Data | Next] → [Data | Next] → NULL

Example:
10 → 20 → 30 → 40 → NULL
0 is the first node (head), pointing to the next node (20).
20 points to **30**, and so on.
40 is the last node, pointing to NULL.

Basic Operations on Singly Linked List

Insertion
Insertion in a singly linked list can occur at:

- **Beginning**
- **End**
- **Specific Position**

🔷 **Insertion at the Beginning**

- Create a new node.
- Point the new node's next to the current head.
- Update the head to the new node.

Example:
Original List: 20 → 30 → 40 → NULL
Insert: 10 at the beginning
New List: 10 → 20 → 30 → 40 → NULL
Insertion at the End

- Create a new node.
- Traverse the list to reach the last node.
- Point the last node's next to the new node.

Example:'
Original List: 10 → 20 → 30 → NULL
Insert: 40 at the end
New List: 10 → 20 → 30 → 40 → NULL

Insertion at a Specific Position
Create a new node.
Traverse the list until the desired position.
Point the new node's next to the current node at the position.
Update the previous node's next to the new node.

Example: Insert 25 at position 2 in the list 10 → 20 → 30 → NULL

New List: 10 → 20 → 25 → 30 → NULL.

Deletion
Deletion in a singly linked list can occur at:

- **Beginning**
- **End**
- **Specific Position**

Deletion at the Beginning

- Update the head to point to the next node.
- Free the memory of the original head.

Example:
Original List: 10 → 20 → 30 → NULL
Delete the first node
New List: 20 → 30 → NULL

Deletion at the End

- Traverse the list to find the **second last node**.
- Update the next of the second last node to NULL.

Example:
Original List: 10 → 20 → 30 → NULL
Delete the last node
New List: 10 → 20 → NULL

Deletion at a Specific Position

- Traverse the list to reach the node before the target node.
- Update the next pointer of the previous node to skip the target node.

Example: Delete node at position 2 in 10 → 20 → 30 → 40 → NULL
New List: 10 → 20 → 40 → NULL

Advantages of Singly Linked List

1. **Dynamic Size:** It allows dynamic memory allocation, making it easy to add or remove nodes.
2. **Efficient Insert/Delete:** Insertion and deletion are faster compared to arrays, especially at the beginning.
3. **Memory Utilization:** It uses memory efficiently by allocating space only when required.

Disadvantages of Singly Linked List

1. **No Backward Traversal:** You can only traverse the list in **one direction**.
2. **Extra Memory Usage:** Each node uses extra memory to store the pointer.
3. **Random Access Not Possible:** You must traverse the list sequentially to access a node, unlike arrays that allow direct access by index.

3) Explain the Doubly linked list?
A **Doubly Linked List (DLL)** is a type of linked list where:

- Each **node** contains **three parts**:
  - **Data**: The actual value stored in the node.
  - **Pointer to the next node (next)**.
  - **Pointer to the previous node (prev)**.
- It allows **traversal in both directions** (forward and backward).

Structure of a Doubly Linked List
[Prev | Data | Next] ↔ [Prev | Data | Next] ↔ [Prev | Data | Next] ↔ NULL
Example:
NULL ← 10 ↔ 20 ↔ 30 ↔ 40 → NULL

- **10** is the head node with prev = NULL and next pointing to **20**.
- **20** points back to **10** and forward to **30**.
- **40** is the last node with next = NULL.

Basic Operations on Doubly Linked List:
Insertion
Insertion in a doubly linked list can occur at:

- **Beginning**
- **End**
- **Specific Position**

Insertion at the Beginning

- Create a new node.
- Point the next of the new node to the current head.
- Update the prev of the current head to the new node.
- Update the head pointer to the new node.

Example:
Original List: 20 ↔ 30 ↔ 40 → NULL
Insert: 10 at the beginning
New List: 10 ↔ 20 ↔ 30 ↔ 40 → NULL

Insertion at the End

- Create a new node.
- Traverse the list to reach the **last node**.
- Point the next of the last node to the new node.
- Set the prev of the new node to the last node.
- Update the next of the new node to NULL.

Example:
Original List: 10 ↔ 20 ↔ 30 → NULL
Insert: 40 at the end
New List: 10 ↔ 20 ↔ 30 ↔ 40 → NULL

Insertion at a Specific Position

- Create a new node.
- Traverse the list to the **target position**.
- Update the pointers:
    - Set the next of the new node to the current node at the position.
    - Set the prev of the new node to the previous node.
    - Adjust the prev and next pointers of the surrounding nodes.

Example: Insert 25 at position 2 in the list 10 ↔ 20 ↔ 30 → NULL
New List: 10 ↔ 20 ↔ 25 ↔ 30 → NULL

Deletion

Deletion in a doubly linked list can occur at:

- **Beginning**
- **End**
- **Specific Position**

Deletion at the Beginning

- Update the head pointer to the **next** node.
- Set the prev pointer of the new head to NULL.
- Free the memory of the removed node.

Example:
Original List: 10 ↔ 20 ↔ 30 → NULL
Delete the first node
New List: 20 ↔ 30 → NULL

Deletion at the End

- Traverse the list to the **last node**.

- Update the next of the **second last node** to NULL.
- Free the memory of the removed node.

Example:
Original List: 10 ↔ 20 ↔ 30 → NULL
Delete the last node
New List: 10 ↔ 20 → NULL

Deletion at a Specific Position

- Traverse the list to the **target position**.
- Adjust the pointers:
    - Set the next of the previous node to the next of the target node.
    - Set the prev of the next node to the prev of the target node.
- Free the memory of the removed node.

Example: Delete node at position 2 in 10 ↔ 20 ↔ 30 ↔ 40 → NULL
New List: 10 ↔ 20 ↔ 40 → NULL

Advantages of Doubly Linked List

1. **Bidirectional Traversal:** You can traverse the list in **both directions** (forward and backward).
2. **Efficient Deletion:** Easier to delete a node compared to singly linked list.
3. **Insertion and Deletion:** Faster compared to arrays, especially in the middle.
4. **No Need for Previous Pointer:** Unlike singly linked list, you can directly access the previous node.

Disadvantages of Doubly Linked List

1. **Extra Memory Usage:** Requires extra memory for the prev pointer in each node.
2. **More Complex Operations:** Insertion and deletion involve more pointer adjustments.
3. **Higher Memory Consumption:** Compared to singly linked lists, DLL uses more memory.

4) Explain the Circular linked list?

A **Circular Linked List (CLL)** is a variation of the linked list where:

- **The last node** points back to the **first node**, forming a circle.
- Unlike singly and doubly linked lists, circular linked lists **do not have NULL pointers** at the end.
- It can be **singly** or **doubly circular**:
    - **Singly Circular Linked List:** Each node points to the next node, and the last node points back to the head.
    - **Doubly Circular Linked List:** Each node has two pointers (next and prev), and the last node points back to the head, while the head's prev points to the last node.

Structure of a Circular Linked List:
[Data | Next] → [Data | Next] → [Data | Next] → (Back to Head)

Example:
10 → 20 → 30 → 40 → (Back to 10)

- **10** is the head, pointing to **20**.
- **40** is the last node, pointing back to **10**, forming a circular structure.

Doubly Circular Linked List
[Prev | Data | Next] ↔ [Prev | Data | Next] ↔ [Prev | Data | Next] ↔ (Circular Connection)
Example:
NULL ←→ 10 ←→ 20 ←→ 30 ←→ 40 ←→ (Back to 10)
10 is the head node with prev pointing to **40** and next pointing to **20**.
40 is the last node, pointing back to **10**, forming a circle.

Basic Operations on Circular Linked List:

Insertion

Insertion in a circular linked list can occur at:

- **Beginning**
- **End**
- **Specific Position**

Insertion at the Beginning

- Create a new node.
- Point the new node's next to the current head.
- Traverse to the **last node** and update its next to point to the new node.
- Update the head to the new node.

Example:

Original List: 20 → 30 → 40 → (Back to 20)

Insert: 10 at the beginning

New List: 10 → 20 → 30 → 40 → (Back to 10)

Insertion at the End

- Create a new node.
- Traverse the list to reach the **last node**.
- Point the next of the last node to the new node.
- Set the next of the new node to the head, forming a circle.

Example:

Original List: 10 → 20 → 30 → (Back to 10)

Insert: 40 at the end

New List: 10 → 20 → 30 → 40 → (Back to 10)

Insertion at a Specific Position

- Create a new node.
- Traverse the list to the **target position**.
- Adjust the next pointers:
    - Set the next of the new node to the next node at the position.
    - Set the next of the previous node to the new node.

Example: Insert 25 at position 2 in the list 10 → 20 → 30 → (Back to 10)

New List: 10 → 20 → 25 → 30 → (Back to 10)

Deletion

Deletion in a circular linked list can occur at:

- **Beginning**
- **End**
- **Specific Position**

Deletion at the Beginning

- Update the head to the **next node**.
- Traverse to the **last node** and update its next to the new head.
- Free the memory of the removed node.

Example:
Original List: 10 → 20 → 30 → (Back to 10)
Delete the first node
New List: 20 → 30 → (Back to 20)

- Traverse the list to reach the **second last node**.
- Update the next of the second last node to point to the head.
- Free the memory of the removed node.

Example:
Original List: 10 → 20 → 30 → (Back to 10)
Delete the last node
New List: 10 → 20 → (Back to 10)

Deletion at a Specific Position

- Traverse the list to the **target position**.
- Adjust the pointers:
    - Set the next of the previous node to the next node of the target node.
- Free the memory of the removed node.

Example: Delete node at position 2 in 10 → 20 → 30 → 40 → (Back to 10)
New List: 10 → 20 → 40 → (Back to 10)

Advantages of Circular Linked List

1. **Efficient Circular Traversal:** You can traverse the entire list from any point, making it useful for applications that require continuous looping.
2. **Dynamic Size:** No need to define the size in advance.
3. **Efficient Insertion/Deletion:** Insertions and deletions are efficient, especially at the **beginning** and **end**.

Disadvantages of Circular Linked List

1. **Complex Traversal:** Traversal requires special handling to avoid infinite loops.
2. **More Complex Implementation:** Requires extra logic to manage the circular link.
3. **No direct access:** Unlike arrays, random access is not possible.

5). Comparing Arrays and linked list?

Array	Linked list
A collection of elements stored in **contiguous memory locations**.	A collection of **nodes** where each node contains data and a pointer/reference to the next (or previous) node.
Static memory allocation (fixed size).	**Dynamic memory allocation (size can grow or shrink).**
Direct/Random access (O(1) time complexity).	**Sequential access** (O(n) time complexity).
Inserting/deleting elements requires shifting subsequent elements (O(n) time complexity).	**Efficient insertion/deletion (O(1) if position is known).**
May lead to **wasted memory** if the size is over-allocated.	Memory efficient with no unused space (but requires extra space for pointers).
Fixed size (defined at declaration).	Flexible size (grows and shrinks dynamically).
Elements are stored in **contiguous** memory locations.	Nodes are scattered across memory with **pointers** linking them.

6). write a program to reverse an array in c language

```
#include <stdio.h>

int main() {
   int n, i, temp;

   // Input the size of the array
   printf("Enter the size of the array: ");
   scanf("%d", &n);

   int arr[n];

   // Input array elements
   printf("Enter %d elements of the array:\n", n);
   for (i = 0; i < n; i++) {
      scanf("%d", &arr[i]);
   }

   // Reversing the array using two-pointer technique
   for (i = 0; i < n / 2; i++) {
      temp = arr[i];
      arr[i] = arr[n - i - 1];
      arr[n - i - 1] = temp;
   }

   // Display the reversed array
   printf("\nReversed array:\n");
   for (i = 0; i < n; i++) {
      printf("%d ", arr[i]);
   }
   printf("\n");

   return 0;
};
```

OUTPUT:
Enter the size of the array: 5
Enter 5 elements of the array:
1 2 3 4 5

Reversed array:
5 4 3 2 1

7) write algorithm for insert and delete a node from doubly linked list ?
Ans : Algorithm for Insertion and Deletion in a Doubly Linked List:
1. Insertion in a Doubly Linked List
There are three cases for inserting a node:

- **At the beginning**

- **At the end**
- **At a specific position**

Algorithm for Inserting at the Beginning

1. **Create a new node**.
2. Set new_node → next = head.
3. If the list is not empty:
    - Set head → prev = new_node.
4. Update head = new_node.

Time Complexity: O(1)

Algorithm for Inserting at the End

1. **Create a new node**.
2. If the list is empty:
    - Set head = new_node.
3. Otherwise:
    - Traverse the list to the last node.
    - Set last → next = new_node.
    - Set new_node → prev = last.

Time Complexity: O(n)

Algorithm for Inserting at a Specific Position

1. **Create a new node**.
2. If inserting at the beginning:
    - Follow the **beginning insertion algorithm**.
3. Otherwise:
    - Traverse the list to the desired position (pos).
    - Set:
        - new_node → next = current → next
        - new_node → prev = current
    - Update the pointers:
        - current → next → prev = new_node
        - current → next = new_node

Time Complexity: O(n)

2. Deletion in a Doubly Linked List

There are three cases for deleting a node:

- **Delete the first node**
- **Delete the last node**
- **Delete a node at a specific position**

Algorithm for Deleting the First Node

1. **Check if the list is empty:**
    - If head == NULL, return.
2. Set temp = head.
3. Update head = head → next.
4. If head != NULL:
    - Set head → prev = NULL.
5. Free the temp node.

Time Complexity: O(1)

Algorithm for Deleting the Last Node

1. **Check if the list is empty:**
    - If head == NULL, return.
2. Traverse to the last node.
3. Set second_last → next = NULL.
4. Free the last node.

Time Complexity: O(n)

Algorithm for Deleting a Node at a Specific Position

1. **Check if the list is empty**:
    - If head == NULL, return.
2. Traverse the list to the desired position (pos).
3. Update the pointers:
    - current → prev → next = current → next
    - current → next → prev = current → prev
4. Free the current node.

Time Complexity: O(n)

8) Specify the use of Header node in a linked list?

Use of Header Node in a Linked List

A **Header Node** is a special node at the **beginning of a linked list** that does not contain any actual data but acts as a placeholder or reference point. It typically stores **meta-information** (e.g., list size) or serves as a pointer to the first node in the list.

Purpose of Header Node

1. **Simplifies List Operations:**
    - Insertion and deletion operations become simpler as you don't need to handle special cases for the first node separately.
    - The header node acts as a consistent starting point.
2. **Improved Traversal Efficiency:**
    - The header node provides a single reference point, making traversal and searching easier and more structured.
3. **Storage of Meta-Information:**
    - It can store the **size of the list**, making size-related operations efficient.
    - It may also contain a pointer to the last node, improving the efficiency of tail operations.
4. **Consistency in List Operations:**
    - The header node makes **empty list handling** easier by providing a consistent reference point, even when the list has no data nodes.

Types of Header Nodes

1. **Dummy Header Node:**
    - A node with no significant data, used purely for simplifying operations.
    - It always points to the first data node.
2. **Sentinel Header Node:**
    - A special marker node used to indicate the **end of the list** (in circular linked lists).
    - It simplifies the termination condition in traversals.

Advantages of Using a Header Node

- Simplifies **insertion and deletion** operations by avoiding special cases for the first node.
- Improves **consistency** and readability of the linked list operations.
- Makes **list management** easier by storing meta-data like size or end pointers.

9). create a doubly linked list by inserting following elements in a list 13,45,23,20,25

```
#include <stdio.h>
#include <stdlib.h>

// Define the structure for a doubly linked list node
struct Node {
   int data;
   struct Node* prev;
   struct Node* next;
};

// Function to create a new node
struct Node* createNode(int data) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = data;
   newNode->prev = NULL;
   newNode->next = NULL;
   return newNode;
}

// Function to insert a node at the end of the doubly linked list
void insertEnd(struct Node** head, int data) {
   struct Node* newNode = createNode(data);

   if (*head == NULL) {
      *head = newNode;  // If the list is empty, make the new node the head
      return;
   }

   struct Node* temp = *head;

   // Traverse to the last node
   while (temp->next != NULL) {
      temp = temp->next;
   }

   // Insert the new node at the end
   temp->next = newNode;
   newNode->prev = temp;
}

// Function to display the doubly linked list
void display(struct Node* head) {
   struct Node* temp = head;

   printf("\nDoubly Linked List: ");
   while (temp != NULL) {
      printf("%d ", temp->data);
```

```
        temp = temp->next;
    }
    printf("\n");
}

// Main function
int main() {
    struct Node* head = NULL;

    // Inserting the elements into the doubly linked list
    insertEnd(&head, 13);
    insertEnd(&head, 45);
    insertEnd(&head, 23);
    insertEnd(&head, 20);
    insertEnd(&head, 25);

    // Display the doubly linked list
    display(head);

    return 0;
};
```

Explanation

1. **Node Structure Definition:**
    - data: To store the value of the node.
    - prev: Pointer to the previous node.
    - next: Pointer to the next node.
2. **Functions Used:**
    - createNode(int data) → Allocates memory for a new node and assigns the value.
    - insertEnd(struct Node** head, int data) → Inserts the node at the **end** of the doubly linked list.
    - display(struct Node* head) → Prints the doubly linked list.
3. **Insertion Process:**
    - The program inserts 13 → 45 → 23 → 20 → 25 sequentially.
    - The nodes are linked together in both directions using next and prev pointers.

Output :

Doubly Linked List: 13 45 23 20 25

10). explain the insertion operation in single linked list. How nodes are inserted after a specified node?

Insertion Operation in a Singly Linked List

The **insertion operation** in a singly linked list involves adding a new node into the list. The operation can be performed in three cases:

1. **At the beginning**
2. **At the end**
3. **After a specified node**

1. Inserting a Node at the Beginning

- **Steps:**
    1. Create a new node.
    2. Assign the next pointer of the new node to the current head.

3. Update the head pointer to the new node.

- **Time Complexity:** O(1)

Inserting a Node at the End

- **Steps:**
  1. Create a new node.
  2. Traverse the list until you reach the last node.
  3. Set the next pointer of the last node to the new node.
  4. Set the next of the new node to NULL.
- **Time Complexity:** O(n)

Inserting a Node After a Specified Node

- This is the most common type of insertion where you add a new node **after a given node**.

Algorithm for Inserting After a Specified Node

1. **Create a New Node**
   - Allocate memory for the new node.
   - Assign the data value to the new node.
2. **Traverse the List**
   - Start from the head and find the specified node.
3. **Insert the New Node**
   - Set:
     - new_node → next = specified_node → next
     - specified_node → next = new_node

Time Complexity: O(n) in the worst case (if the specified node is at the end). **Space Complexity:** O(1) for the new node.

C Program to Insert After a Specified Node in a Singly Linked List:

```
#include <stdio.h>
#include <stdlib.h>

// Define the structure for a singly linked list node
struct Node {
   int data;
   struct Node* next;
};

// Function to create a new node
struct Node* createNode(int data) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = data;
   newNode->next = NULL;
   return newNode;
}

// Function to insert a node at the end
void insertEnd(struct Node** head, int data) {
   struct Node* newNode = createNode(data);

   if (*head == NULL) {
      *head = newNode;  // If the list is empty, make the new node the head
      return;
```

```
    }

    struct Node* temp = *head;

    while (temp->next != NULL) {
        temp = temp->next;
    }

    temp->next = newNode;
}

// Function to insert a new node after a specified node
void insertAfter(struct Node* prev_node, int data) {
    if (prev_node == NULL) {
        printf("The given previous node cannot be NULL\n");
        return;
    }

    struct Node* newNode = createNode(data);

    newNode->next = prev_node->next;
    prev_node->next = newNode;
}

// Function to display the linked list
void display(struct Node* head) {
    struct Node* temp = head;

    printf("\nSingly Linked List: ");
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

// Main function
int main() {
    struct Node* head = NULL;

    // Insert elements at the end
    insertEnd(&head, 10);
    insertEnd(&head, 20);
    insertEnd(&head, 30);
    insertEnd(&head, 40);

    printf("Original List:");
    display(head);
```

```
    // Inserting after the second node (after 20)
    printf("\nInserting 25 after 20...\n");
    insertAfter(head->next, 25);  // Inserts 25 after 20

    display(head);

    return 0;
};
```

Explanation

1. **Structure Definition:**
    - Each Node contains:
        - data: To store the value.
        - next: Pointer to the next node.
2. **Functions Used:**
    - createNode(int data) → Creates a new node.
    - insertEnd() → Adds a node at the end of the list.
    - insertAfter() → Adds a node **after a specified node**.
    - display() → Prints the list.
3. **Insertion After a Specified Node:**
    - insertAfter(head->next, 25) → Inserts 25 **after the second node** (20).

Output

Singly Linked List: 10 -> 20 -> 30 -> 40 -> NULL

Inserting 25 after 20...
Singly Linked List: 10 -> 20 -> 25 -> 30 -> 40 -> NULL

UNIT-3

STACK

1. **What is a stack?**

A stack is a linear data structure that follows the **Last In, First Out (LIFO)** principle. This means that the most recently added (pushed) element is the first one to be removed (popped).

2. **What are the two main operations of a stack?**
    - **Push** (inserts an element)
    - **Pop** (removes the top element)
3. **What is the time complexity of push and pop operations?**

Both **Push** and **Pop** operations take **O(1) (constant time)** because they operate only at the top of the stack.

4. **What is stack overflow?**

5. Stack overflow occurs when we try to push an element into a stack that has reached its maximum size (for an array-based stack).

Example:

If a stack is implemented using a fixed-size array (e.g., size 5), pushing a 6th element will cause **stack overflow**.

5. **What is stack underflow?**

Stack underflow happens when we try to pop an element from an empty stack. Since there are no elements to remove, it leads to an error.

6. **How can a stack be implemented?**

Using **arrays** (fixed size) or **linked lists** (dynamic size).

7. **Which is better: array-based or linked-list-based stack?**
    - Array-based stacks are faster but have a fixed size.
    - Linked-list-based stacks are flexible but use extra memory.
8. **What is the top of the stack?**

The **stack top** refers to the **element at the highest position** in a stack. It is the most recently added item and is the first to be removed when performing a **pop** operation.

9. **Where are stacks used in real life?**
    - Undo/Redo in editors
    - Backtracking (e.g., maze solving, recursion)
    - Expression evaluation (postfix, prefix)
    - Function call management in recursion
10. **How does a stack handle function calls?**

The **call stack** stores function calls and returns them in **LIFO** order

11. What is the top of the stack?

The **top** element is the last inserted element, which will be removed first when popping.

Example:

Stack: [10, 20, 30]

Top element = 30 (last pushed element)

12.what are the Advantages of Stack ?

1. **Follows LIFO (Last In, First Out) Order**
    - Ensures the most recently added data is accessed first, which is useful in function calls, undo operations, and expression evaluation.
2. **Efficient Memory Management**
    - Stack memory allocation is **automatic** and **fast**, making function calls and local variable storage more efficient than heap allocation.
3. **Simplifies Function Call Management**
    - The **call stack** keeps track of function calls and local variables, ensuring smooth execution of recursive and nested functions.
4. **Backtracking and Undo Operations**
    - Stacks are used in algorithms that require **backtracking** (e.g., solving mazes, Depth First Search (DFS), undo/redo features in text editors).
5. **Expression Evaluation and Parsing**
    - Stacks simplify evaluating expressions in **postfix notation** and **parsing syntax** in compilers.
6. **Efficient Operations (O(1) Complexity)**
    - Push and pop operations take **constant time (O(1))**, making them very fast.
7. **Less Memory Wastage** (Compared to Queues)
    - Since elements are removed from the top, there is **no need for shifting** like in queues.
8. **Easy to Implement**
    - Stacks can be implemented using **arrays** or **linked lists** with minimal effort.

13.what are the Disadvantages of Stack?

1. **Limited Size (Fixed in Array Implementation)**
    - If implemented using an **array**, the stack has a **fixed size**, which can lead to **stack overflow** if too many elements are pushed.
2. **Stack Overflow and Underflow**
    - **Stack Overflow** occurs when pushing an element into a full stack.
    - **Stack Underflow** happens when popping an element from an empty stack.

3. **No Random Access**
    - Unlike arrays, stacks **do not allow direct access** to elements in the middle; only the **top element** can be accessed.
4. **Extra Memory in Linked List Implementation**
    - If implemented using a **linked list**, extra memory is needed for pointers, making it less space-efficient than an array-based stack.
5. **Limited Flexibility for Large Data Handling**
    - Stacks are **not ideal for large dynamic data storage** since removing an element from the middle is not possible without modifying the structure.
6. **Complex Debugging and Error Handling**
    - Since the stack only allows access to the top element, debugging or retrieving past elements can be difficult.

14.what are the Real-Time Applications of Stack

1. **Function Calls (Call Stack)**
    - When a function is called, it is pushed onto the **call stack**. Once the function finishes execution, it is popped off.
    - Used in **recursion**, where multiple function calls are stacked until the base case is reached.
2. **Undo/Redo in Text Editors**
    - Every action (typing, deleting, formatting) is pushed onto a stack.
    - Undo: Pops the last action and restores the previous state.
    - Redo: Reapplies the last undone action.
3. **Backtracking Algorithms**
    - Used in **maze solving, puzzle solving, and DFS (Depth First Search)**.
    - If a wrong path is chosen, the algorithm **backtracks** by popping from the stack.
4. **Expression Evaluation and Syntax Parsing**
    - Used in **compilers and calculators** for:
        - Converting infix expressions to postfix/prefix.
        - Evaluating mathematical expressions.
5. **Browser Forward and Back Buttons**
    - Every visited page is **pushed** onto a stack.
    - Clicking "Back" **pops** the current page and returns to the previous page.
    - Clicking "Forward" pushes the popped page back.
6. **Undo Operations in Graphics Software**
    - Drawing or editing actions are stored in a stack for easy reversal.

7. **Memory Management in OS (Stack vs Heap)**
    - The stack is used for **local variables and function calls**, while the heap is for **dynamic memory allocation**.
8. **Balancing Parentheses in Expressions**
    - Used in **checking balanced brackets ({[()]})** by pushing opening brackets and popping when a matching closing bracket is found.
9. **Compiler Syntax Checking**
    - Stacks help check **correct nesting of loops, function calls, and expressions** in programming languages.
10. **Reversing a String or Data**

- Characters are pushed onto a stack and then popped out in reverse order.

15. What are the different types of stack implementations? Compare them

Stacks can be implemented in two main ways:

1. Array-Based Stack

- Uses a fixed-size array to store elements.
- The top of the stack is updated with each push or pop operation.

ADV:

- Faster access (O(1) time complexity for push/pop).
- Simple to implement.

DIS:

- Fixed size, leading to stack overflow if full.
- Memory may be wasted if the stack is not fully used.

2. Linked List-Based Stack

- Uses a linked list where each node contains the data and a pointer to the next node.

Advantages:

- No fixed size (dynamically allocated).
- No memory wastage.

Disadvantages:

- Extra memory needed for pointers.
- Slightly slower due to pointer manipulation.

16. Explain the working of a stack with its operations and real-world analogy.

Answer:

A **stack** is a **linear data structure** that follows the **Last In, First Out (LIFO)** principle. This means the last element added (pushed) onto the stack will be the first one to be removed (popped).

Operations on Stack:

1. **Push** – Adds an element to the top of the stack.
2. **Pop** – Removes and returns the top element of the stack.
3. **Peek (Top)** – Returns the top element without removing it.
4. **isEmpty** – Checks if the stack is empty.
5. **isFull** (for array-based stacks) – Checks if the stack is full.

Real-World Analogy:

Imagine a stack of plates in a cafeteria:

- When a new plate is placed, it is added on top (**Push**).
- When someone takes a plate, they pick the top one (**Pop**).
- The plate at the bottom is only accessible after removing all others (**LIFO** principle).

17. How is a stack used in recursion? Explain with an example in C.

Answer:

A stack is used to manage function calls in **recursion**. Every function call is **pushed onto the call stack**, and when the function completes, it is **popped** off the stack.

Example: Factorial Calculation Using Recursion

```
#include <stdio.h>
// Recursive function to calculate factorial
int factorial(int n) {
```

```
  if (n == 0)
      return 1;
   return n * factorial(n - 1);
}
// Main function
int main() {
   int num = 5;
   printf("Factorial of %d is %d\n", num, factorial(num));
   return 0;
}
```

Stack Behavior for factorial(5)**:**

```
factorial(5)
factorial(4)
factorial(3)
factorial(2)
factorial(1)
factorial(0)  # Base case, returns 1
```

Each function call is **pushed onto the stack** until factorial(0) is reached. Then, it starts **popping** the calls and computing the final result.

Output:

Factorial of 5 is 120

18. How is Stack Used in Call Logs?

- **LIFO Principle:** The most recent call is displayed first, while older calls move down the list.
- **Push Operation:** When a new call is made or received, it is **pushed** onto the call log stack.
- **Pop Operation:** When viewing or deleting call logs, the latest (top) call is **popped** first.
- **Limited Size:** If the call log reaches a maximum limit (e.g., 100 entries), the **oldest calls are removed** to make space for new ones (similar to a circular stack).

19.How Stack is Used in Viewing History?

- **LIFO Principle:** The most recent activity (page visited, file opened) is stored at the **top** of the stack.
- **Push Operation:** When a new activity occurs (e.g., opening a webpage), it is **pushed** onto the stack.
- **Pop Operation:** When the user presses "Back," the latest activity is **popped** off the stack, returning to the previous activity.
- **Forward Navigation (Redo):** Another stack is used to store activities when moving forward.

20.Real-World Use Cases of Stack in Viewing History

1. **Web Browsers:**
    - When a user visits a webpage, it is **pushed onto the stack**.
    - Clicking the **Back button** pops the current page and navigates to the previous one.
    - Clicking **Forward** can be implemented using another stack.
2. **File Explorers:**
    - When navigating through folders, each visited folder is **pushed** onto the stack.
    - Pressing **Back** pops the current folder and returns to the previous one.
3. **Mobile Applications:**
    - App screens (activities) are stacked so that when a user presses the **Back button**, the last screen is **popped** and removed.

1.Explain the properties of Stack?

Properties of Stack

A **stack** is a linear data structure that follows the **LIFO (Last In, First Out)** principle. Here are its key properties:

1. **LIFO Principle**
    - The last element inserted is the first one to be removed.
    - Example: If you push A, B, C onto a stack, C will be popped first.
2. **Operations**
    - **Push**: Adds an element to the top of the stack.
    - **Pop**: Removes the top element from the stack.
    - **Peek (Top)**: Returns the top element without removing it.
    - **isEmpty**: Checks if the stack is empty.

- **isFull**: Checks if the stack is full (for array-based stacks).

3. **Memory Allocation**
   - Can be implemented using **arrays** (fixed size) or **linked lists** (dynamic size).
4. **Access Restriction**
   - Only the **top** element can be accessed or modified directly.
5. **Recursive Nature**
   - Stacks are used in **recursion** (function call stack).
6. **Applications**
   - **Expression evaluation** (Postfix, Prefix, Infix)
   - **Backtracking** (like in maze solving, undo operations)
   - **Function calls** (maintains activation records)
   - **Browser history and undo-redo operations**

2.Write a C program to implement a stack using an array?

```
#include <stdio.h>

#include <stdlib.h>

#define MAX 5 // Maximum size of the stack

int stack[MAX], top = -1;

// Function to push an element onto the stack

void push() {

    int value;

    if (top == MAX - 1) {

        printf("Stack Overflow! Cannot push more elements.\n");

        return;

    }

    printf("Enter the value to push: ");

    scanf("%d", &value);

    stack[++top] = value;

    printf("%d pushed into the stack.\n", value);

}
```

```
// Function to pop an element from the stack
void pop() {
    if (top == -1) {
        printf("Stack Underflow! No elements to pop.\n");
        return;
    }
    printf("%d popped from the stack.\n", stack[top--]);
}
// Function to return the top element without removing it
void peek() {
    if (top == -1) {
        printf("Stack is empty!\n");
        return;
    }
    printf("Top element is: %d\n", stack[top]);
}
// Function to display all elements in the stack
void display() {
int i;
    if (top == -1) {
        printf("Stack is empty!\n");
        return;
    }
```

```
    printf("Stack elements: ");
    for (i = top; i >= 0; i--)
        printf("%d ", stack[i]);
    printf("\n");
}

// Main function with menu-driven interface
int main() {
    int choice;
    while (1) {
        printf("\nStack Operations:\n");
        printf("1. Push\n2. Pop\n3. Peek\n4. Display\n5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1: push();
                    break;
            case 2: pop();
                    break;
            case 3: peek();
                    break;
            case 4: display();
                    break;
```

```
            case 5: exit(0);
            default: printf("Invalid choice! Please enter a valid option.\n");
        }
    }
    return 0;
}
```

Sample test data

1. Push

2. Pop

3. Peek

4. Display

5. Exit

Enter your choice: 1

Enter the value to push: 10

10 pushed into the stack.

Enter your choice: 1

Enter the value to push: 20

20 pushed into the stack.

Enter your choice: 1

Enter the value to push: 30

30 pushed into the stack.

Enter your choice: 4

Stack elements: 30 20 10

Enter your choice: 3

Top element is: 30

Enter your choice: 2

30 popped from the stack

Enter your choice: 4

Stack elements: 20 10

Enter your choice: 5

3. Explain the concept of a stack in detail with an example.

Answer:

A **stack** is a linear data structure that follows the **LIFO (Last In, First Out)** principle, meaning that the last element inserted into the stack is the first one to be removed. It is similar to a stack of plates, where the topmost plate is the first to be removed.

A stack can be implemented using:

1. **Arrays** (Static implementation)
2. **Linked Lists** (Dynamic implementation)

Operations on Stack:

1. **Push:** Adds an element to the top of the stack.
2. **Pop:** Removes the top element from the stack.
3. **Peek:** Retrieves the top element without removing it.
4. **isEmpty:** Checks whether the stack is empty.
5. **isFull:** Checks whether the stack is full (only in an array implementation).

Example:

Consider a stack with the following sequence of operations:

Push(10) → Push(20) → Push(30) → Pop() → Push(40)

Step-by-step changes in the stack:

Operation: Push(10) Stack: [10]

Operation: Push(20) Stack: [10, 20]
Operation: Push(30) Stack: [10, 20, 30]
Operation: Pop() Stack: [10, 20] (30 is removed)
Operation: Push(40) Stack: [10, 20, 40]

Applications of Stack:

1. **Expression Evaluation:** Converting infix expressions to postfix/prefix.
2. **Undo/Redo Mechanism:** Used in text editors and applications.
3. **Recursion Handling:** Function calls use a stack to store return addresses.
4. **Backtracking:** Used in maze-solving algorithms and game development.

---

4. What are the advantages and disadvantages of using a stack?

Answer:

Advantages of Stack:

1. **Efficient for LIFO Operations:** Best suited for scenarios where the last inserted element needs to be accessed first.
2. **Memory Management:** Used in function calls and recursion (call stack).
3. **Simple and Fast:** Push and Pop operations take **O(1) time complexity**.

Disadvantages of Stack:

1. **Limited Size in Array Implementation:** Fixed size can lead to stack overflow.
2. **Difficult to Access Middle Elements:** Unlike linked lists, direct access to elements other than the top is not possible.
3. **Dynamic Memory Usage (Linked List Implementation):** Requires extra memory for pointers.

---

5. How is a stack used in recursion? Explain with an example.

Answer:

Recursion is a process where a function calls itself until a base condition is met. Internally, recursion uses a **call stack** to keep track of function calls.

How Stack is Used in Recursion:

Each function call is **pushed** onto the stack.

1. When a function completes execution, it is **popped** from the stack.
2. The stack stores return addresses and local variables of each function call.
3. Example: Factorial Using Recursion

```
#include <stdio.h>
int factorial(int n) {
   if (n == 0) return 1; // Base case
   return n * factorial(n - 1); // Recursive call
}
int main() {
   int num = 5;
   printf("Factorial of %d is %d\n", num, factorial(num));
   return 0;
}
```

Stack Representation for factorial(5)
Call: factorial(5) → Push 5
Call: factorial(4) → Push 4
Call: factorial(3) → Push 3
Call: factorial(2) → Push 2
Call: factorial(1) → Push 1
Call: factorial(0) → Push 0 (Base Case, Return 1)
Return: factorial(1) → Pop 1
Return: factorial(2) → Pop 2
Return: factorial(3) → Pop 3
Return: factorial(4) → Pop 4
Return: factorial(5) → Pop 5

Final result: factorial(5) = 5 × 4 × 3 × 2 × 1 = 120

---

6. Differentiate between stack and queue.

Answer:

Feature	Stack (LIFO)	Queue (FIFO)
Principle	Last In, First Out (LIFO)	First In, First Out (FIFO)
Insertion (Push/Enqueue)	Performed at the **top**	Performed at the **rear**
Deletion (Pop/Dequeue)	Done from the **top**	Done from the **front**

Feature	Stack (LIFO)	Queue (FIFO)
Implementation	Arrays, Linked Lists	Arrays, Linked Lists, Circular Queues
Example Use Case	Function calls, Undo operations	Scheduling processes, Printer queue

7.Explain how stacks are used in expression evaluation and conversion.

Answer:

Stacks play a crucial role in **expression evaluation** and **conversion** between different types of mathematical expressions:

Infix Expression: Operators are between operands (e.g., A + B).

- **Postfix Expression:** Operators follow operands (e.g., A B +).
- **Prefix Expression:** Operators precede operands (e.g., + A B).

Why Use Stacks?

1. **Operator Precedence Handling:** Stacks help maintain the correct order of operations.
2. **Efficient Processing:** Conversion and evaluation become easy using a stack.

---

1. Infix to Postfix Conversion Using Stack

Example: Convert A + B * C to postfix form.

Step	Symbol	Stack (Operators)	Postfix Expression
1	A	-	A
2	+	+	A
3	B	+	A B
4	*	+ *	A B
5	C	+ *	A B C
6	End	Pop stack	A B C * +

Final postfix expression: **A B C * +**

---

2. Evaluating a Postfix Expression Using Stack

Example: Evaluate **5 3 + 8 ***

Step Symbol Stack (Operands)

1 5 5

2 3 5 3

3 + 8 (5 + 3)

4 8 8 8

5 * 64 (8 × 8)

Final result: **64**

Infix, Prefix, and Postfix Notation Using Stack

In **mathematical expressions**, there are three common notations:

1. **Infix Notation** – Operators are placed **between operands** (e.g., A + B).
2. **Prefix Notation (Polish Notation)** – Operators are placed **before operands** (e.g., + A B).
3. **Postfix Notation (Reverse Polish Notation - RPN)** – Operators are placed **after operands** (e.g., A B +).

Conversion Between Notations

Notation Example Expression (A + B * C)

Infix A + (B * C)

Prefix + A * B C

Postfix A B C * +

---

1. Infix to Postfix Conversion Using Stack

Algorithm

1. **Initialize an empty stack** for operators.
2. **Scan the infix expression from left to right**.
3. **If operand (A-Z or 0-9), append to output**.
4. **If an operator (+, -, *, /)**:
    - Pop operators from the stack that have **higher or equal precedence** and append them to the output.
    - Push the current operator onto the stack.
5. **If '(' (left parenthesis), push it onto the stack**.
6. **If ')' (right parenthesis), pop from the stack to the output until '(' is found**.
7. **At the end, pop all remaining operators from the stack**.

C Program: Infix to Postfix Conversion

```
#include <stdio.h>
#include <ctype.h>  // For isalnum()
#include <string.h>
#define MAX 100  // Stack size
// Stack structure
struct Stack {
    char arr[MAX];
    int top;
};
// Initialize stack
void initialize(struct Stack* stack) {
    stack->top = -1;
}
// Check if stack is empty
int isEmpty(struct Stack* stack) {
    return stack->top == -1;
}
// Push an element
void push(struct Stack* stack, char value) {
    stack->arr[++stack->top] = value;
}
// Pop an element
```

```
char pop(struct Stack* stack) {
    return stack->arr[stack->top--];
}
// Get precedence of operators
int precedence(char op) {
    if (op == '+' || op == '-') return 1;
    if (op == '*' || op == '/') return 2;
    return 0;  // For non-operators
}
// Convert infix to postfix
void infixToPostfix(char* infix, char* postfix) {
    struct Stack stack;
    initialize(&stack);
    int j = 0;
    for (int i = 0; infix[i] != '\0'; i++) {
        char ch = infix[i];
        // If operand, add to output
        if (isalnum(ch)) {
            postfix[j++] = ch;
        }
        // If '(', push to stack
        else if (ch == '(') {
            push(&stack, ch);
        }
        // If ')', pop until '(' is found
        else if (ch == ')') {
            while (!isEmpty(&stack) && stack.arr[stack.top] != '(') {
                postfix[j++] = pop(&stack);
            }
            pop(&stack);  // Remove '('
        }
        // If operator, handle precedence
        else {
            while (!isEmpty(&stack) && precedence(stack.arr[stack.top]) >= precedence(ch)) {
                postfix[j++] = pop(&stack);
            }
            push(&stack, ch);
        }
    }
    // Pop remaining operators
    while (!isEmpty(&stack)) {
        postfix[j++] = pop(&stack);
    }
```

```
 postfix[j] = '\0';  // Null terminate string
}
// Main function
int main() {
    char infix[] = "A+B*C";
    char postfix[MAX];
    infixToPostfix(infix, postfix);
    printf("Postfix Expression: %s\n", postfix);
    return 0;
}
```

Output:

Postfix Expression: ABC*+

---

2. Infix to Prefix Conversion Using Stack

Algorithm

1. **Reverse the infix expression**.
2. **Convert infix to postfix (using stack)**.
3. **Reverse the result to get the prefix expression**.

C Program: Infix to Prefix Conversion

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define MAX 100
// Stack structure
struct Stack {
    char arr[MAX];
    int top;
};
// Initialize stack
void initialize(struct Stack* stack) {
    stack->top = -1;
}
// Check if stack is empty
int isEmpty(struct Stack* stack) {
    return stack->top == -1;
}
// Push an element
```

```
void push(struct Stack* stack, char value) {
    stack->arr[++stack->top] = value;
}
// Pop an element
char pop(struct Stack* stack) {
    return stack->arr[stack->top--];
}
// Get precedence of operators
int precedence(char op) {
    if (op == '+' || op == '-') return 1;
    if (op == '*' || op == '/') return 2;
    return 0;
}
// Reverse a string
void reverse(char* str) {
    int n = strlen(str);
    for (int i = 0; i < n / 2; i++) {
        char temp = str[i];
        str[i] = str[n - i - 1];
        str[n - i - 1] = temp;
    }
}
// Convert infix to prefix
void infixToPrefix(char* infix, char* prefix) {
    // Reverse infix expression
    reverse(infix);
    // Replace ( with ) and vice versa
    for (int i = 0; infix[i] != '\0'; i++) {
        if (infix[i] == '(') infix[i] = ')';
        else if (infix[i] == ')') infix[i] = '(';
    }
    // Convert to postfix
    char postfix[MAX];
    struct Stack stack;
    initialize(&stack);
    int j = 0;
    for (int i = 0; infix[i] != '\0'; i++) {
        char ch = infix[i];
        if (isalnum(ch)) {
            postfix[j++] = ch;
        } else if (ch == '(') {
            push(&stack, ch);
        } else if (ch == ')') {
```

```
 while (!isEmpty(&stack) && stack.arr[stack.top] != '(') {
            postfix[j++] = pop(&stack);
        }
        pop(&stack);
    } else {
        while (!isEmpty(&stack) && precedence(stack.arr[stack.top]) >= precedence(ch)) {
            postfix[j++] = pop(&stack);
        }
        push(&stack, ch);
    }
  }
  while (!isEmpty(&stack)) {
    postfix[j++] = pop(&stack);
  }
  postfix[j] = '\0';
  // Reverse postfix to get prefix
  reverse(postfix);
  strcpy(prefix, postfix);
}
// Main function
int main() {
  char infix[] = "A+B*C";
  char prefix[MAX];
  infixToPrefix(infix, prefix);
  printf("Prefix Expression: %s\n", prefix);
  return 0;
}
```

Output:

Prefix Expression: +A*BC

Summary

Notation	Example	Conversion Using Stack
Infix	(A + B) * C	Given directly
Postfix	A B + C *	Use **stack** to push/pop operators
Prefix	+ A * B C	Reverse + Convert to postfix + Reverse

UNIT- 4

Queues: Introduction to queues: properties and operations, implementing queues using arrays and linked lists, Applications of queues in breadth-first search, scheduling, etc. Deques: Introduction to deques (double-ended queues), Operations on deques and their applications.

1. Define Queue.

A Queue is an ordered list in which all insertions take place at one end called the rear, while all deletions take place at the other end called the front. Rear is initialized to -1 and front is initialized to 0. Queue is also referred as First In First Out (FIFO) list.

2. What are the various operations performed on the Queue?

The various operations performed on the queue are

CREATE(Q) – Creates Q as an empty Queue.

Enqueue(Q,X) – Adds the element X to the Queue.

Dequeue(Q) – Deletes a element from the Queue.

ISEMTPTY(Q) – returns true if Queue is empty else false.

ISFULL(Q) - returns true if Queue is full else false

3. How do you test for an empty Queue?

The condition for testing an empty queue is rear=front-1. In linked list implementation of queue the condition for an empty queue is the header node link field is NULL.

4. What is enqueue operation?

Enqueue - adding an element to the queue at the rear end If the queue is not full, this function adds an element to the back of the queue, else it prints “OverFlow”.

5. What is dequeue operation?

Dequeue – removing or deleting an element from the queue at the front end If the queue is not empty, this function removes the element from the front of the queue, else it prints “UnderFlow”.

6. What are the types of queue

The following are the types of queue: ● Double ended queue ● Circular queue ● Priority queue

7. Define Circular Queue.

Another representation of a queue, which prevents an excessive use of memory by arranging elements/ nodes Q1,Q2,…Qn in a circular fashion. That is, it is the queue, which wraps around upon reaching the end of the queue

8. Define Dequeue.

Deque stands for Double ended queue. It is a linear list in which insertions and deletion

are made from either end of the queue structure.

9. Distinguish between stack and queue.

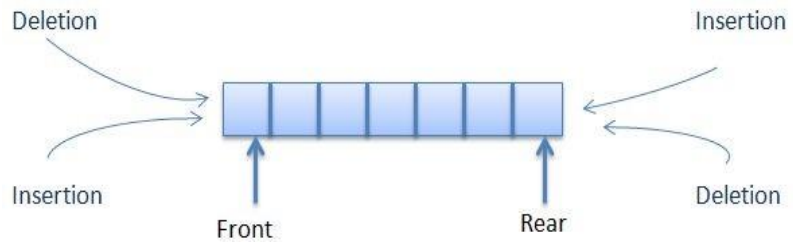
STACK	QUEUE
Insertion and deletion are made at one end.	Insertion at one end rear and deletion at other end front.
The element inserted last would be removed first. So LIFO structure.	The element inserted first would be removed first. So FIFO structure.
Full stack condition: If(top==Maxsize) Physically and Logically full stack	Full stack condition: If(rear = = Maxsize) Logically full. Physically may or may not be full.

10. How do you test for an empty queue?

To test for an empty queue, we must check whether REAR==HEAD where REAR is a pointer pointing to the last node in a queue and HEAD is a pointer that points to the dummy header. In the case of array implementation of queue, the condition to be checked for an empty queue is REAR<FRONT.

11. **Define double ended queue**

It is a special type of queue that allows insertion and deletion of elements at both Ends. It is also termed as DEQUE.



12. What are the methods to implement queue in C?

The methods to implement queues are: Array based, Linked list based

13. What are the applications of queue?

The following are the areas in which queues are applicable

a. Simulation

b. Batch processing in an operating system
c. Multiprogramming platform systems
d. Queuing theory
e. Printer server routines
f. Scheduling algorithms like disk scheduling , CPU scheduling
g. I/O buffer requests

14. What are the basic features of Queue?

1. Queue is an ordered list of elements of similar data types.
2. Queue is a FIFO (First in First Out) structure.
3. Once a new element is inserted into the Queue, all the elements inserted before the new element in the queue must be removed, to remove the new element.

1. What is queue and basic operations on queue?

A queue is a linear data structure where elements are stored in the FIFO (First In First Out) principle where the first element inserted would be the first element to be accessed. A queue is an Abstract Data Type (ADT). It is open at both its ends. The data is inserted into the queue through one end and deleted from it using the other end. Queue is very frequently used in most programming languages. Queue operations also include initialization of a queue, usage and permanently deleting the data from the memory.

The most fundamental operations in the queue ADT include: enqueue(), dequeue(), peek(), isFull(), isEmpty(). These are all built-in operations to carry out data manipulation and to check the status of the queue. Queue uses two pointers − front and rear. The front pointer accesses the data from the front end (helping in enqueueing) while the rear pointer accesses data from the rear end (helping in dequeuing).

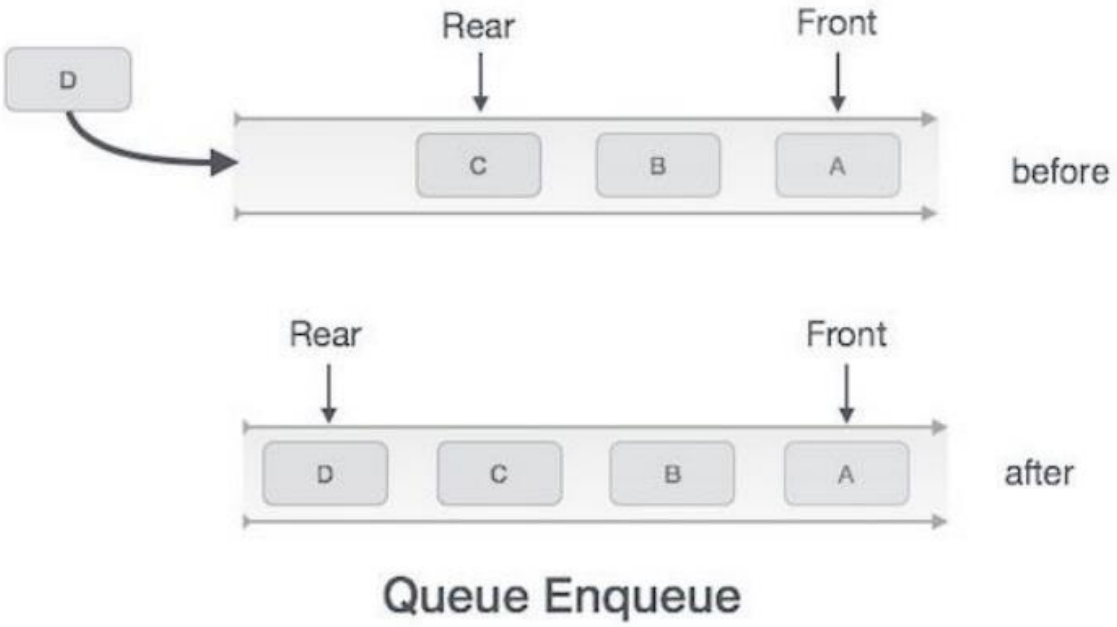
2. Write the algorithms for the insertion and deletion operations on queue.

Queue Insertion Operation: Enqueue()

The *enqueue()* is a data manipulation operation that is used to insert elements into the Queue. The following algorithm describes the **enqueue()** operation in a simpler way.

Algorithm

1. START
2. if(rear == length) Check if the queue is full .
3. If the queue is full, produce overflow error and exit.
4. rear++ (If the queue is not full, increment rear pointer to point the next empty space.)
5. q[rear] = item (Add data element to the queue location, where the rear is pointing.)
6. if(front==0) then front++
7. return success.
8. END



Queue Deletion Operation: dequeue()

The *dequeue()* is a data manipulation operation that is used to remove elements from the stack. The following algorithm describes the dequeue() operation in a simpler way.

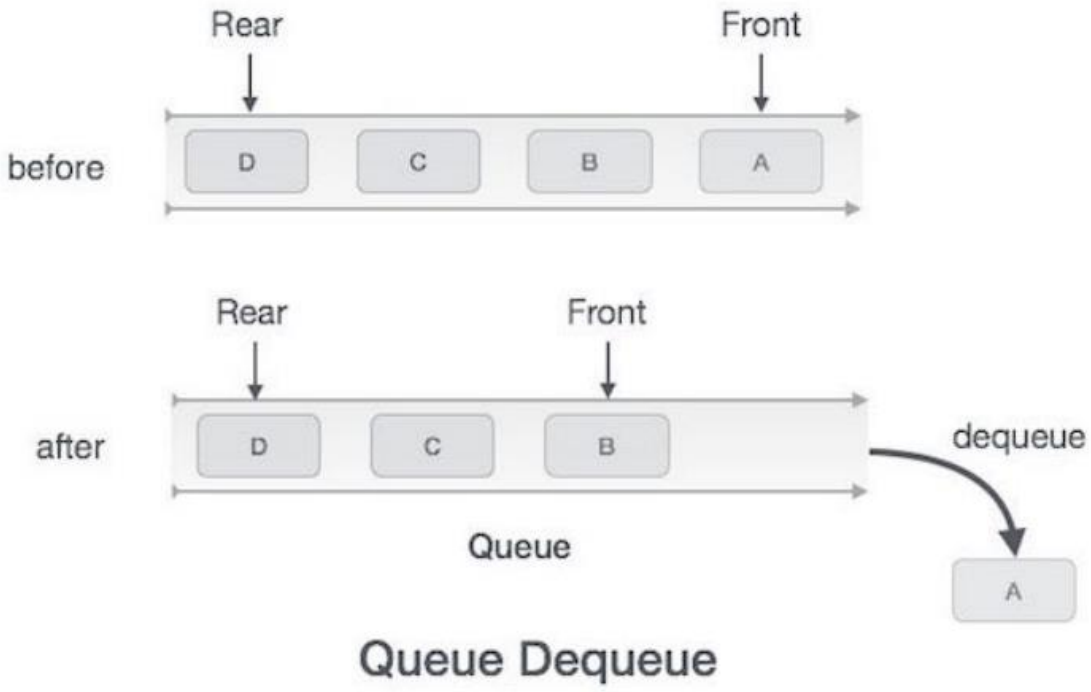
Algorithm

1. START
2. if((rear==0) and (front==0)) (Check if the queue is empty.)
3. If the queue is empty, produce underflow error and exit.
4. If the queue is not empty, access the data where front is pointing.
4. item=q[front]
5. if (front==rear)
6. front=0, rear=0.
7. else

8. front ++ (Increment front pointer to point to the next available data element.)

9. Return success.

10. END



Queue Dequeue

3. Write the algorithms to check the status and to display the elements of the queue.

Algorithm for checking the status of the Queue

Steps:

1. if((rear==0) and (front==0))

2. print Queue is empty

3. else

4. if(rear==length)

5. print Queue is full

6. else

7. print no. of items in queue and free space available in queue(rear, length-rear)

8. end if

9. end if

10. stop

4. Write a c program to implement queue using array?

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>

void enqueue();
void dequeue();
void display();
void status();
void front_element();
void rear_element();

int item, front=0, rear=0, length, q[50],i;

void main()
{

   int choice;
   printf("enter the length");
   scanf("%d", &length);

   while(1)
   {
      printf("\n Menu \n1.display \n2.enqueue \n3.dequeue\n4.Status \n5.first element \n5.last element \n7.exit ");

      printf("\nenter the choice: ");
      scanf("%d", &choice);
```

```
        switch(choice)
        {
           case 1: display();
                   break;

           case 2: enqueue();
                   break;

           case 3: dequeue();
                   break;

           case 4: status();
                   break;

           case 5: front_element();
                   break;

           case 6: rear_element();
                   break;

           case 7: exit(0);
        }
    }

}


void enqueue()
{
    if(rear ==length)
        printf("Queue is full. enqueue not possible");
    else{
        printf("enter the value");
        scanf("%d",&item);
        rear++;
        q[rear]=item;
        if(front==0)
            front++;
    }
}

void dequeue()
```

```
{
   if(front==0 && rear==0)
      printf("enque is empty. Dequeue not possible\n ");

else
{

   if(front==rear)
   {

      item=q[front];
      front=0;
      rear=0;

   }
   else
   {
      item=q[front];
      front++;

   }
   printf("the deleted item=%d", item);
}

}

void status()
{
   if(rear==0&&front==0)
      printf("queue is empty\n");

   else
   {
      if(rear==length)
         printf("queue is full\n");
      else
      {

         printf("the filled spaces=%d\n", rear );
         printf("the free spaces=%d\n", length-rear );
      }
   }
}
```

```
void display()
{
   i=front;
   while(i<=rear)
   {
      printf("\t%d", q[i]);
      i++;

   }
}

void front_element()
{
   if(front==0&&rear==0)
      printf("\n queue is empty");
   else
   {
      printf("\n First element is :%d", q[front]);
   }
}

void rear_element()
{
   if(front==0&&rear==0)
      printf("\n queue is empty");
   else
   {
      printf("\n Rear element is :%d", q[rear]);
   }
}
```

5. Write a c program to implement queue using Single Linked List?

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>

struct node
{
   int data;
   struct node *next;
```

```
}*qheader, *front,*rear,*new, *ptr;

void creation();
void enqueue();
void dequeue();
void display();
void status();

int item;

void main()
{
   int choice;
   while(1)
   {
      printf("\n Menu \n1.creation \n2.enqueue \n3.dequeue\n4.Status \n5.Display \n6.exit ");
      printf("\nenter the choice: ");
      scanf("%d", &choice);

      switch(choice)
      {
         case 1: creation();
               break;

         case 2: enqueue();
               break;
         case 3: dequeue();
               break;
         case 4: status();
               break;
         case 5: display();
               break;
         case 6: exit(0);
               }
   }
}

void creation()
{
   qheader = (struct node*) malloc(sizeof(struct node));
   qheader ->data=NULL;
   qheader ->next=NULL;
   front=qheader;
   rear=qheader;
}
void enqueue()
```

```
{
   new= (struct node*) malloc(sizeof(struct node));
   if(new==NULL)
   {
      printf("\n insufficient memory, Push not possible");
   }


   else
   {
      printf("\n enter the item to insert: ");
      scanf("%d", &item);
      new->data=item;
      rear->next=new;
      new->next=NULL;
      if(front==qheader&&rear==qheader)
      {
         front=new;
         rear=new;
      }
      else
      {
         rear=new;
      }
   }
}

void dequeue()
{
 if(front==qheader&&rear==qheader)
 {
    printf("queue is empty, dequeue not possible");
 }
else
{
   ptr=front;
   if(front==rear)
   {
      front=qheader;
      rear=qheader;
      qheader->next=NULL;
   }
   else
   {
      qheader->next=ptr->next;
      front=ptr->next;
```

```
    }
    free(ptr);
}
}

void status()
{
    int count=0;
    if(front==qheader&&rear==qheader)
 {
     printf("queue is empty, dequeue not possible");
 }
    else
    {
        ptr=front;
        while(ptr->next!=NULL)
        {
            count++;
            ptr=ptr->next;
        }
     printf("Number of elements in queue=%d",count);
}
}

void display()
{
    if(front==qheader&&rear==qheader)
 {
     printf("queue is empty, dequeue not possible");
 }
 Else
{
printf("elements of the queue are: ");
    ptr=front;
    while(ptr->next!=NULL)
    {
        printf("\t%d", ptr->data);
        ptr=ptr->next;
    }
}
}
```

6. Write the applications of Queues.

Applications of Queue are

1. Breadth First Search (BFS) Algorithm

Breadth-first search (BFS) is a general technique for traversing a graph. One starts at some arbitrary node as the start node and expand shallowest unexpanded node before exploring deeper levels. The algorithm uses a queue to keep track of nodes to visit where new successors go to the end of the queue. Each entry in the queue stores all edges of the node processed and new nodes are added to the rear of the queue.

Step 1: Initially all nodes are undiscovered. Mark the start node as discovered Enqueue the start node S Front of queue which initially is the start node S.

Now enqueue all neighbours of node S into the Queue

Step2: Repeat Until queue is Empty

a. Select the node F in front of queue & mark it as discovered and process it by examining its neighbours.

b. Repeat until all neighbours of F have been processed

If a neighbour of F has not yet been discovered, add it to the rear of queue else do not add it to queue

c. Now delete the original node from the queue and go to step 2

2. Printing Job Management

Queues has the application in the management of printer jobs.

Many users send their printing jobs to a public printer. The printer will put them into a queue according to the arrival time and print the jobs one by one.

Assume that the documents are A.doc, B.doc, C.doc first arrive for printing. Therefore, the three jobs are enqueued and A.doc is sent for printing. Next A.doc finishes and is dequeued. Therefore B.doc starts printing. Meanwhile D.doc arrives and is enqueued. while B.doc is still printing. Next B.doc finishes and is dequeued. Now C.doc the document at the front of the queue is sent for printing. Next C.doc finishes and is dequeued. Now the final item in the queue, D.doc is sent for printing. This process will continue as long as documents arrive for printing.

3. **Handling Website Traffic**

Increasing website traffic is the primary goal for websites. However, extreme spikes in internet traffic frequently cause website infrastructure to fail. Particularly, the inventory systems and payment gateways are two of the most vulnerable systems needed to be preserved.

Another common concern is to ensure good user experience and fairness in terms of directing users from one webpage to another, in heavy-traffic situations.

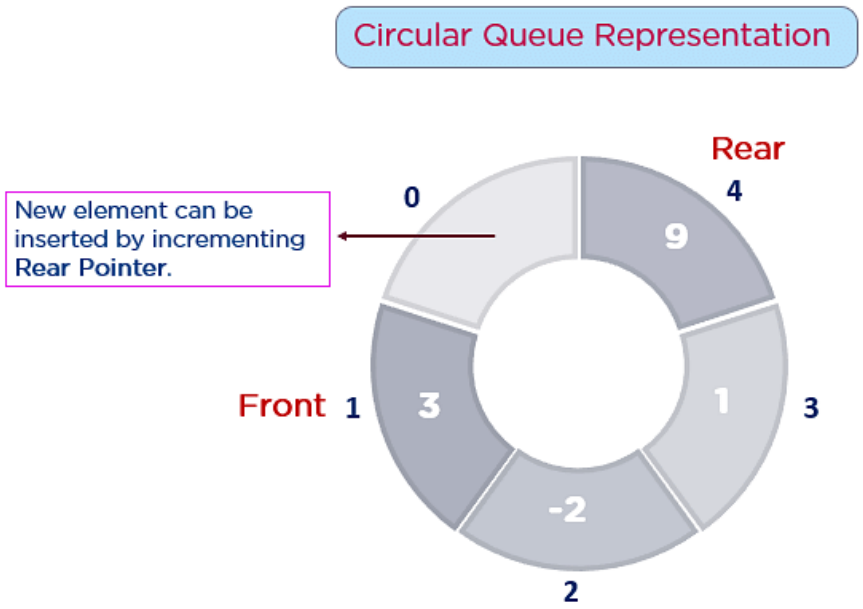
One of the best website traffic management solutions is by implementing a virtual HTTP request queue, thus making it one of the most used applications of queues.

CPU Scheduling

This is one of the most common applications of queues where a single resource is shared among multiple consumers, or asked to perform multiple tasks.
Operating system store all the processes in the memory in the form of a queue.
It makes sure that the tasks are performed based on the sequence of requests. When we need to replace a page, the oldest page is selected for removal, underlining the First-In-First-Out (FIFO) principle.

7. Explain circular queue and its implementation?

A circular queue is an extended version of a linear queue as it follows the First In First Out principle with the exception that it connects the last node of a queue to its first by forming a circular link. Hence, it is also called a Ring Buffer.



The Circular Queue is similar to a Linear Queue in the sense that it follows the FIFO (First In First Out) principle but differs in the fact that the last position is connected to the first position, replicating a circle.

Operations
Front - Used to get the starting element of the Circular Queue.
Rear - Used to get the end element of the Circular Queue.
enQueue(value) - Used to insert a new value in the Circular Queue. This operation takes place from the end of the Queue.
deQueue() - Used to delete a value from the Circular Queue. This operation takes place from the front of the Queue.

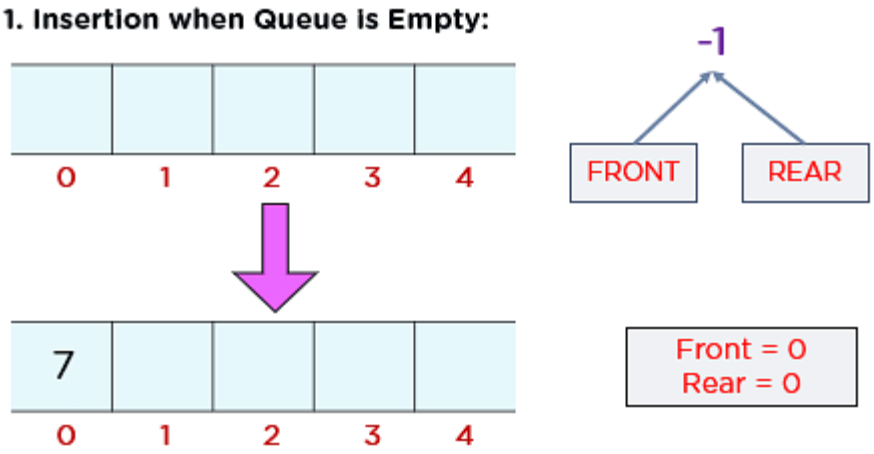
Steps for Implementing Queue Operations
Enqueue() and Dequeue() are the primary operations of the queue, which allow you to manipulate the data flow. These functions do not depend on the number of elements inside

the queue or its size; that is why these operations take constant execution time, i.e., O(1) [time-complexity].

1. Enqueue(x) Operation

Steps to insert (enqueue) a data element into a circular queue -

- Step 1: Check if the queue is full (Rear + 1 % Maxsize = Front)
- Step 2: If the queue is full, there will be an Overflow error
- Step 3: Check if the queue is empty, and set both Front and Rear to 0
- Step 4: If Rear = Maxsize - 1 & Front != 0 (rear pointer is at the end of the queue and front is not at 0th index), then set Rear = 0
- Step 5: Otherwise, set Rear = (Rear + 1) % Maxsize
- Step 6: Insert the element into the queue (Queue[Rear] = x)
- Step 7: Exit

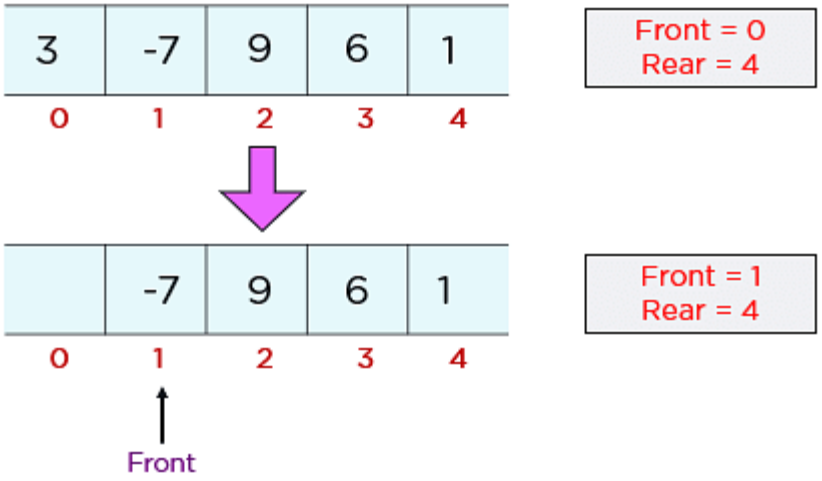


2. Dequeue() Operation

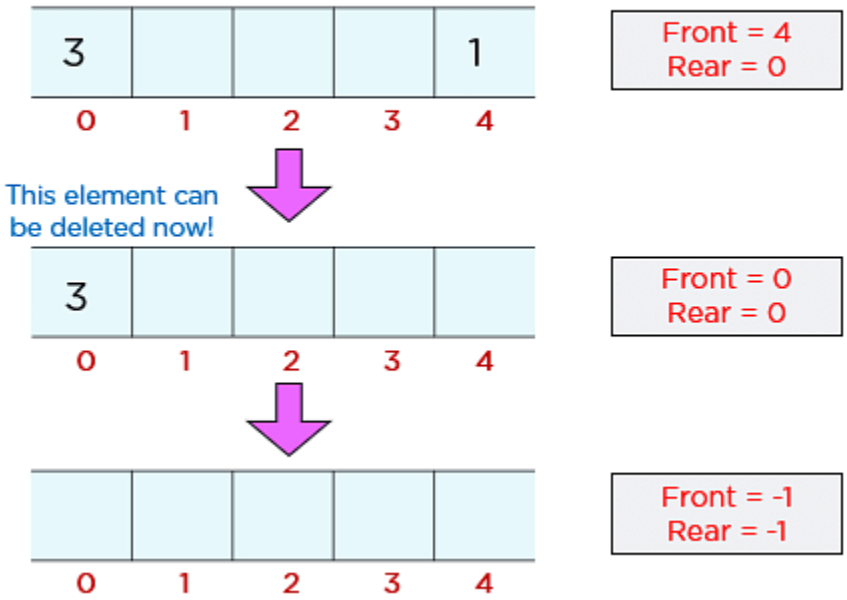
Obtaining data from the queue comprises two subtasks: access the data where the front is pointing and remove the data after access. You should take the following steps to remove data from a circular queue -

- Step 1: Check if the queue is empty (Front = -1 & Rear = -1)
- Step 2: If the queue is empty, Underflow error
- Step 3: Set Element = Queue[Front]
- Step 4: If there is only one element in a queue, set both Front and Rear to -1 (IF Front = Rear, set Front = Rear = -1)
- Step 5: And if Front = Maxsize -1 set Front = 0
- Step 6: Otherwise, set Front = Front + 1
- Step 7: Exit

1. Deletion when rear at the end of queue and front at the beginning of the queue



2. Deletion when front reached at end of queue but there is element rear is at beginning of queue



8. Explain double ended queue and its operations?

A double-ended queue, commonly referred to as a deque (pronounced "deck"), is a versatile data structure that **allows for the insertion and deletion of elements from both ends.** A double-ended queue in the data structure is an extension of the linear queue structure.

Deque provides greater flexibility in data handling. Unlike a standard queue, which follows the First-In-First-Out (FIFO) principle, a deque **can function in both FIFO and Last-In-First-Out (LIFO) modes**, making it suitable for various applications in programming and algorithms.

Double-ended queues in the data structure are especially useful in scenarios where you need to manage a collection of items with dynamic size and require efficient access to both ends. They can be implemented using arrays or linked lists, each offering different performance characteristics.

Types of Double ended queues Implementations

Deques can be implemented in several ways, each with its strengths and weaknesses. The most common implementations are:

1. Array-Based Deque

- In this implementation, a fixed-size array is used to store the elements of the deque. Two pointers (or indices) are maintained: one for the front and one for the rear.
- Advantages include constant time complexity (O(1)) for access and modification operations.
- However, a significant limitation is that resizing the array when it becomes full can be costly, requiring all elements to be copied to a new larger array.

2. Linked List-Based Deque

- This implementation utilizes nodes that contain data and pointers to the next and previous nodes.
- Elements can be added or removed from either end without needing to copy existing elements.
- The time complexity for all deque operations (insertion, deletion, and access) remains (O(1)).
- However, linked lists require extra memory for pointers and can be less cache-friendly compared to arrays.

9. Write the algorithm to implement Deque operations?

Operations of Deque

1. insertfront(): Adds an item at the front of Deque
2. insertrear(): Adds an item at the rear end of deque
3. deletefront(): Deletes from the front end of deque
4. **deleterear():** Delete an item from the rear end of Deque

Insertfront():

1. if(front==0) then
2. print insertion at front is not possible
3. else
4. if (front == -1) then
5. front = 0, rear=0
6. if(front>0) then
7. front –
8. deque[front] = item
9. end if
10. end if

11. end if
12. stop

Insert at rear:

Step:
1. if(rear==size)
2. print insert at rear end not possible
3. else
4. if(rear<size)
5. rear++
6. if(front == -1)
7. front =0, rear = 0
8. deque[rear] = item
9. end if
10. end if
11. end if
12. stop

Delete at front

Steps:
1. if(front == -1) then
2. print delete at front is not possible
3. else
4. print deleted item q[front]
5. if(front == rear) then
6. front=-1, rear=-1
7. else
8. front++
9. end if
10. end if
11. stop

Delete at rear:
Steps:
1. if(front == -1) and (rear==-1) then
2. print Delete at rear end is not possible
3. else
4. print deleted itemdeque[rear]
5. if(front==rear) then
6. front=-1, rear=-1
7. else
8. rear—
9. end if
10. end if

11. stop

Display
Algorithm for Display items
Steps:
1. if(front==-1) and (rear==-1) then
2. print Deque is empty
3. else
4. i=front
5. while(i<=rear) do
6. print deque[i]
7. i++
8. end while
9. end if
10. stop

10. Write a C program to implement Deque operations.

```
#include <stdio.h>
#include <stdlib.h>


struct Node
{
 struct Node *prev;
 int data;
 struct Node *next;

}*header=NULL,*ptr=NULL;


void create(int A[],int n)
{
 int i;
 struct Node *t,*last;
 header=(struct Node *)malloc(sizeof(struct Node));
 header->prev=NULL;
 header->data=A[0];
 header->next=NULL;
```

```
 last=header;
 for(i=1;i<=n;i++)
 {
 t=(struct Node*)malloc(sizeof(struct Node));
 t->data=A[i];
 t->next=last->next;
 t->prev=last;
 last->next=t;
 last=t;
 }
}

void Display( )
{
   ptr=header;
 while(ptr->next!=NULL)
 {
  printf("\n%d\n ",ptr->data);
  ptr=ptr->next;
 }
}
void insert_first()
{
   struct Node *t;
   t=(struct Node*)malloc(sizeof(struct Node));
   int x;
   printf("enter the value to insert at first position: ");
   scanf("%d", &x);
   t->data = x;
   t->prev = NULL;
   t->next= header;
   header->prev=t;
```

```
    header=t;

}

void insert_any()
{
    struct Node *t;
    t=(struct Node*)malloc(sizeof(struct Node));
    int x, pos,i;
    printf("enter the value to insert at given position: ");
    scanf("%d", &x);
    printf("enter the position to insert the given value: ");
    scanf("%d", &pos);
    ptr=header;
    t->data = x;
    for(i=0;i<pos-1;i++){
        ptr=ptr->next;}

    t->next = ptr->next;

    t->prev= ptr;
    ptr->next->prev=t;
    ptr->next=t;
}


int main()
{
 int a[5]={1,2,3,4,5};
 create(a,5);
 printf("\n After creating the Linked List: \n");
 Display();
```

```
 insert_first();
 printf("\n After inserting element at the first position in the Linked List: \n");
 Display();
 insert_any();
  printf("\n After inserting element at the given position in the Linked List: \n");
 Display();

 return 0;
}



//Double Linked List Delete Program

#include <stdio.h>
#include <stdlib.h>

struct Node
{
 struct Node *prev;
 int data;
 struct Node *next;

}*header=NULL,*ptr=NULL;

void create(int A[],int n)
{
 int i;
 struct Node *t,*last;
 header=(struct Node *)malloc(sizeof(struct Node));
 header->prev=NULL;
 header->data=A[0];
```

```
 header->next=NULL;
  last=header;
 for(i=1;i<=n;i++)
 {
 t=(struct Node*)malloc(sizeof(struct Node));
 t->data=A[i];
 t->next=last->next;
 t->prev=last;
 last->next=t;
 last=t;
 }
}

void Display( )
{
    ptr=header;
 while(ptr->next!=NULL)
 {
  printf("\n%d\n ",ptr->data);
  ptr=ptr->next;
 }
}

void delete_first()
{
    ptr=header;
    header=header->next;
    free(ptr);
    header->prev=NULL;
}

void delete_any()
```

```
{
   ptr=header;
   int i, pos;

   printf("enter the pos to delete");
   scanf("%d", &pos);
   for(i=0;i<pos-1;i++)
      ptr=ptr->next;

   ptr->prev->next=ptr->next;

   ptr->next->prev=ptr->prev;

   free(ptr);

}

int main()
{
 int a[5]={10,20,30,40,50};
 create(a,5);
 printf("\n After creating the Linked List: \n");
 Display();
 delete_first();
 printf("\n After deleting the first element from the Linked List: \n");
 Display();
 delete_any();
 printf("\n After deleting the element at given position from the Linked List: \n");
 Display();
 return 0;
}
```

UNIT-V

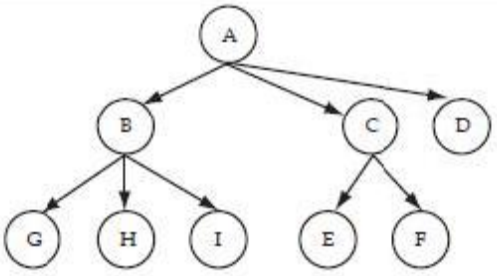
PART-A

1.What is a tree?

A tree is a set of one or more nodes T such that:

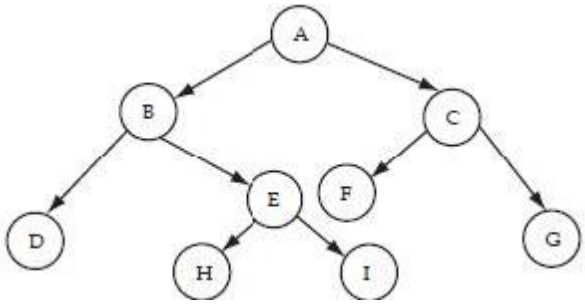
1. There is a specially designated node called root, and

2. Remaining nodes are partitioned into n >= 0 disjoint set of nodes T1, T2,…,Tn each of which is a tree.



2.What is a binary tree data structure?

A binary tree is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called left subtree and right subtree. It is the most commonly used non-linear data structure



3.what are the properties of binary tree?

1. If a binary tree contains n nodes, then it contains exactly n – 1 edges;

2. A Binary tree of m levels has 2m– 1 nodes or less.

4.What are the types of binary tree representations?

A binary tree data structure is represented using two methods. Those methods are

1)Array Representation

2)Linked List Representation

5. what are the applications of trees?

Following are some important application of trees

Decision Tree – Machine learning Algorithm

Sorting

File Systems

Search Engines

Binary Expression Trees

Data Compression- Huffman coding

6.Name different types of the traversals.

There are three types of binary tree traversals.

1)In - Order Traversal

2)Pre - Order Traversal

3)Post - Order Traversal

7. What is binary search tree?

A binary search tree (BST) is a binary tree. It may be empty. If it is not empty, then all nodes follows the below mentioned properties −

- Every element has a unique key.
- The keys in a nonempty left subtree (right subtree) are smaller (larger) than the key in the root of subtree.
- The keys in a nonempty right subtree larger than the key in the root of subtree.

The left and right subtrees are also binary search trees.

8.what are the properties of binary search tree?

For any given node: All keys in its left subtree are less than the node's key.

All keys in its right subtree are greater than the node's key.

Performing an in-order traversal of a BST visits the keys in ascending sorted order.

9.Define Hash table.

The Hash table data structure stores elements in key-value pairs where

Key- unique integer that is used for indexing the values

Value - data that are associated with keys.

10.define hash function.

A hash function is a special mathematical function that takes a key and converts it into a specific index within the hash table. This function ensures that the same key always produces the same index. Good hash functions distribute keys evenly across the hash table to minimize collisions.

11. What are the types of hash functions

The types of hash functions are:

1. Division Method.
2. Mid Square Method.
3. Folding Method.
4. Multiplication Method.

12. What is collision?

When the hash function generates the same index for multiple keys, there will be a conflict (what value to be stored in that index). This is called a hash collision.

13. What are collision resolution techniques?

collisions in hash tables can be minimized using collision resolution techniques

1.Separate chaining

2.Open addressing

14. What are the Techniques used for open addressing?

The techniques used for open addressing are

–Linear Probing

–Quadratic Probing

–Double Hashing

15. What is double hashing?

Double hashing uses two hash functions,

– one to find the initial location to place the key and a

– second to determine the size of the jumps in the probe sequence.

PART-B

1. What is binary search tree? Explain its operations with example.

Definition: A binary search tree (BST) is a binary tree. It may be empty. If it is not empty, then all nodes follows the below mentioned properties −

- **Every element has a unique key.**
- **The keys in a nonempty left subtree (right subtree) are smaller (larger) than the key in the root of subtree.**
- **The keys in a nonempty right subtree larger than the key in the root of subtree.**
- **The left and right subtrees are also binary search trees.**

The basic operations that can be performed on binary search tree data structure, are following −

1. Search − search an element in a binary search tree.

2.Insert − insert an element into a binary search tree / create a tree.

3.Delete − Delete an element from a binary search tree.

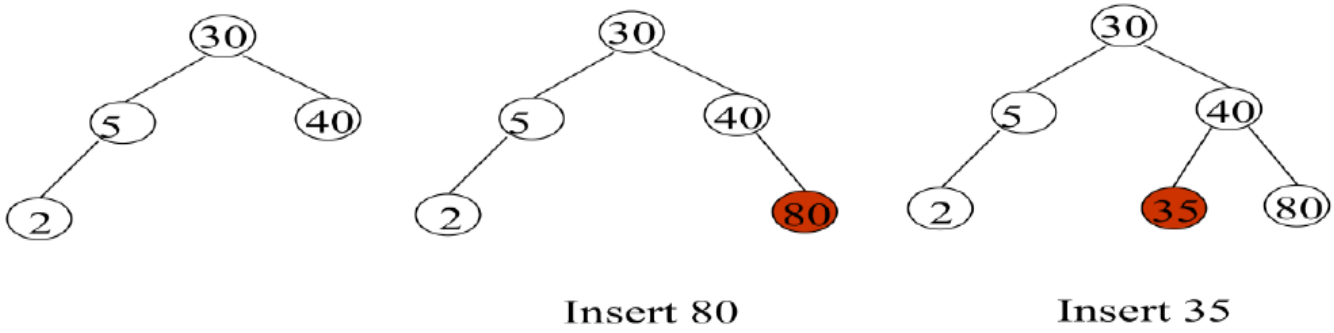
4. Traversal- visit and print every element in a binary search tree

Searching a Binary Search Tree

Let an element k is to search in binary search tree. Start search from root node of the search tree. If root is NULL, search tree contains no nodes and search unsuccessful. Otherwise, compare k with the key in the root. If k equals the root's key, terminate search, if k is less than key value, search element k in left subtree otherwise search element k in right subtree. The function search recursively searches the subtrees.

Inserting into a Binary Search Tree

The very first insertion creates the tree. Afterwards, whenever an element is to be inserted. First locate its proper location. Start search from root node then if data is less than key value, search empty location in left sub tree and insert the data. Otherwise search empty location in right sub tree and insert the data.



Deleting a node

Remove operation on binary search tree is more complicated, than insert and search. Basically, in can be divided into two stages:

- **search for a node to remove**
- **if the node is found, run remove algorithm.**

Remove algorithm in detail

Now, let's see more detailed description of a remove algorithm. First stage is identical to algorithm for lookup, except we should track the parent of the current node. Second part is more tricky. There are three cases, which are described below.

i.Node to be removed has no children. --This case is quite simple. Algorithm sets corresponding link of the parent to NULL and disposes the node.

Example. Remove -4 from a BST.

ii.Node to be removed has one child. In this case, node is cut from the tree and algorithm links single child (with it's subtree) directly to the parent of the removed node.

iii.Node to be removed has two children. --This is the most complex case. The deleted node can be replaced by either largest key in its left subtree or the smallest in its right subtree. Preferably which node has one child.

4. Traversal

Displaying (or) visiting order of nodes in a binary tree is called as Binary Tree Traversal.

There are three types of binary tree traversals.

In - Order Traversal

Pre - Order Traversal

Post - Order Traversal

We may perform In-order Traversal as it displays the nodes in ascending order.

In - Order Traversal

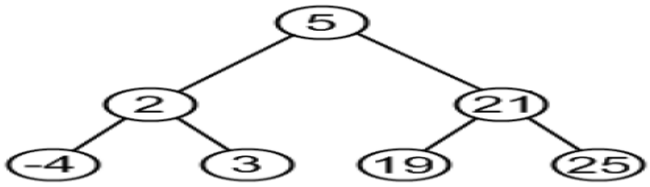
Until all nodes are traversed −

Step 1 − Recursively traverse left subtree.

Step 2 − Visit root node.

Step 3 − Recursively traverse right subtree.

Example:



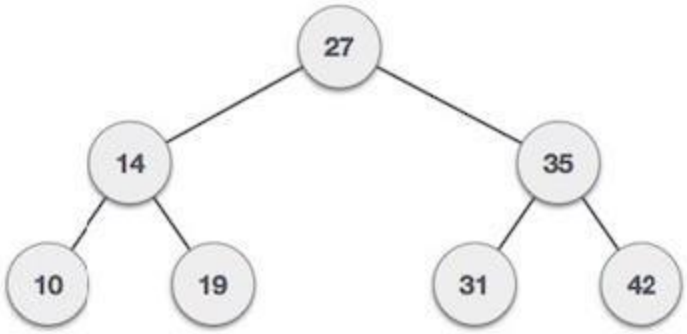
In-order: -4,2,3,5,19,21,25

2. what is binary search tree? Describe the process of inserting a node into a binary search tree.Write the algorithm for BST insertion.

Definition: A binary search tree (BST) is a binary tree. It may be empty. If it is not empty, then all nodes follows the below mentioned properties −

- **Every element has a unique key.**
- **The keys in a nonempty left subtree (right subtree) are smaller (larger) than the key in the root of subtree.**
- **The keys in a nonempty right subtree larger than the key in the root of subtree.**
- **The left and right subtrees are also binary search trees.**

Example of Binary search tree



Inserting into a Binary Search Tree

The very first insertion creates the tree. Afterwards, whenever an element is to be inserted. First locate its proper location. Start search from root node then if data is less than key value, search empty location in left sub tree and insert the data. Otherwise search empty location in right sub tree and insert the data.

In binary search tree, new node is always inserted as a leaf node. The insertion operation is performed as follows...

Step 1: Create a newNode with given value and set its left and right to NULL.

Step 2: Check whether tree is Empty.

Step 3: If the tree is Empty, then set set root to newNode.

Step 4: If the tree is Not Empty, then check whether value of newNode is smaller or larger than the node (here it is root node).

Step 5: If newNode is smaller than or equal to the node, then move to its left child. If newNode is larger than the node, then move to its right child.

Step 6: Repeat the above step until we reach a node (e.i., reach to NULL) where search terminates.

Step 7: After reaching a last node, then insert the newNode as left child if newNode is smaller or equal to that node else insert it as right child.

Algorithm

Create newnode

If root is NULL

then create root node

return

If root exists then

compare the data with node.data

while until insertion position is located

If data is greater than node.data

goto right subtree

else

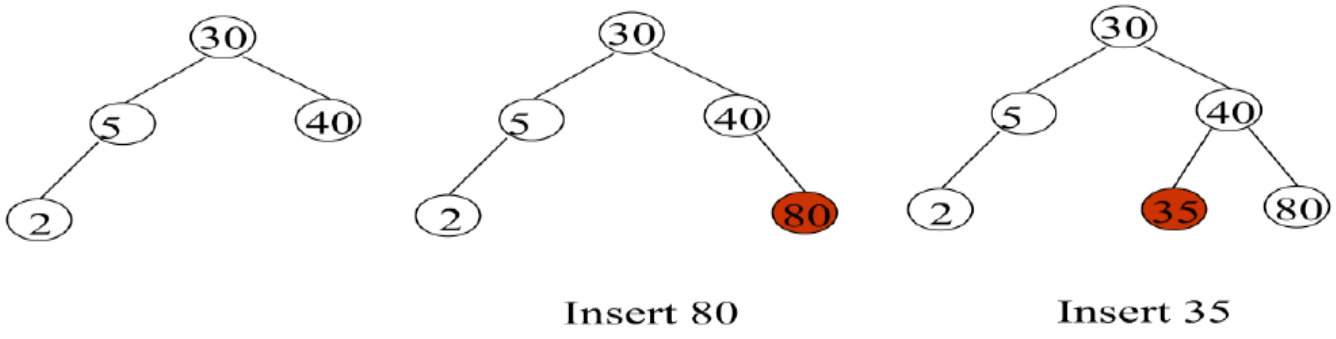
goto left subtree

endwhile

insert newnode

end If

Example:



3. what is binary search tree(BST)? Write a C function to implement BST insertion operation.

Definition: A binary search tree (BST) is a binary tree. It may be empty. If it is not empty, then all nodes follows the below mentioned properties −

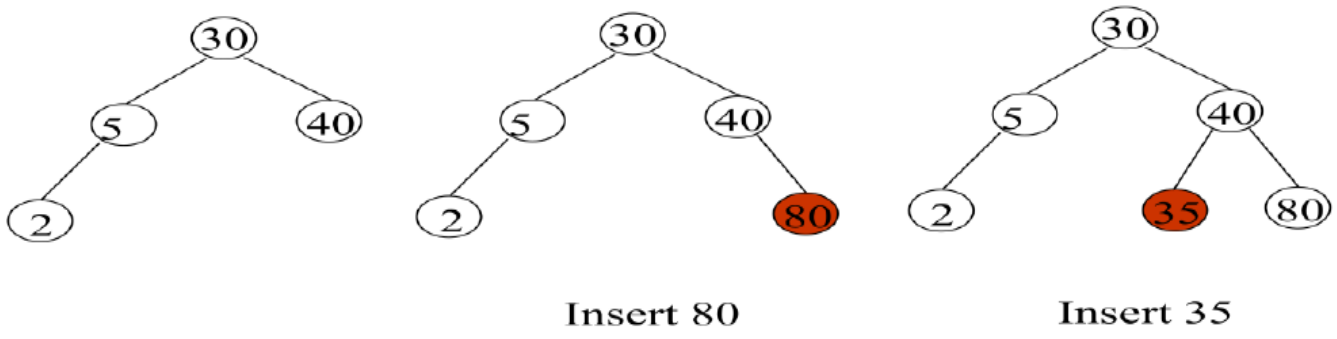
- **Every element has a unique key.**
- **The keys in a nonempty left subtree (right subtree) are smaller (larger) than the key in the root of subtree.**
- **The keys in a nonempty right subtree larger than the key in the root of subtree.**
- **The left and right subtrees are also binary search trees.**

Implementation of Insertion operation in BST

```
void insert(int data)
{
struct node *tempNode = (struct node*) malloc(sizeof(struct node));
struct node *current;
struct node *parent;
tempNode->data = data;
tempNode->leftChild = NULL;
tempNode->rightChild = NULL;
//if tree is empty, create root node
if(root == NULL) {
root = tempNode;
}else {
current = root;
parent = NULL;
while(1) {
parent = current;
if(data < parent->data) {
        current = current->leftChild;
```

```
            if(current == NULL) {
            parent->leftChild = tempNode;
            return;
             }
    }
else {
            current = current->rightChild;
            if(current == NULL) {
            parent->rightChild = tempNode;
            return;
            }
     }
 }
}
}
```

Example:



4. Describe the process of Deleting a node from a binary search tree with an example.

Deleting a node

Remove operation on binary search tree is more complicated, than insert and search. Basically, in can be divided into two stages:

- **search for a node to remove**
- **if the node is found, run remove algorithm.**

Remove algorithm in detail

Now, let's see more detailed description of a remove algorithm. First stage is identical to algorithm for lookup, except we should track the parent of the current node. Second part is more tricky. There are three cases, which are described below.

i.Node to be removed has no children. --This case is quite simple.

We use the following steps to delete a leaf node from BST...

Step 1: Find the node to be deleted using search operation

Step 2: Delete the node using free function (If it is a leaf) and terminate the function.

Example. Remove -4 from a BST.

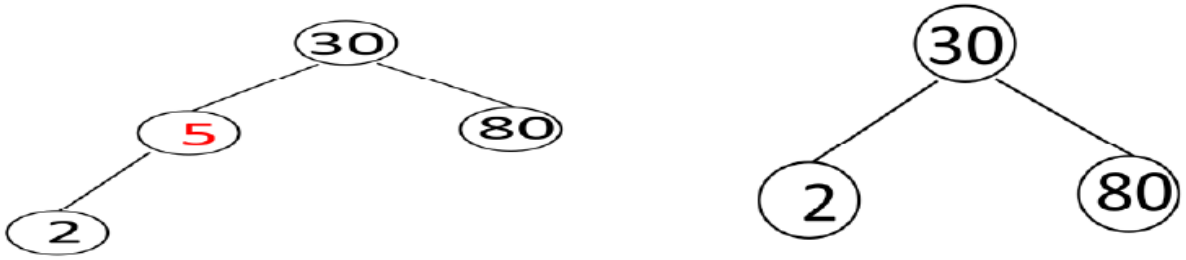
ii.Node to be removed has one child. In this case, node is cut from the tree and algorithm links single child (with it's subtree) directly to the parent of the removed node.

We use the following steps to delete a node with one child from BST...

Step 1: Find the node to be deleted using search operation

Step 2: If it has only one child, then create a link between its parent and child nodes.

Step 3: Delete the node using free function and terminate the function.



iii.Node to be removed has two children. --This is the most complex case. The deleted node can be replaced by either largest key in its left subtree or the smallest in its right subtree. Preferably which node has one child.

We use the following steps to delete a node with two children from BST...

Step 1: Find the node to be deleted using search operation

Step 2: If it has two children, then find the largest node in its left subtree (OR) the smallest node in its right subtree.

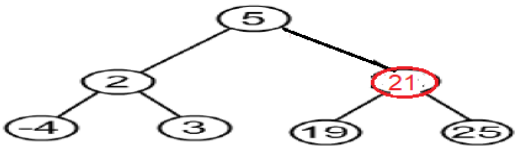
Step 3: Swap both deleting node and node which found in above step.

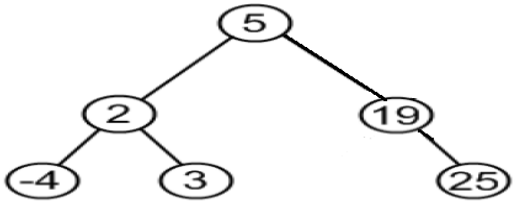
Step 4: Then, check whether deleting node came to case 1 or case 2 else goto steps 2

Step 5: If it comes to case 1, then delete using case 1 logic.

Step 6: If it comes to case 2, then delete using case 2 logic.

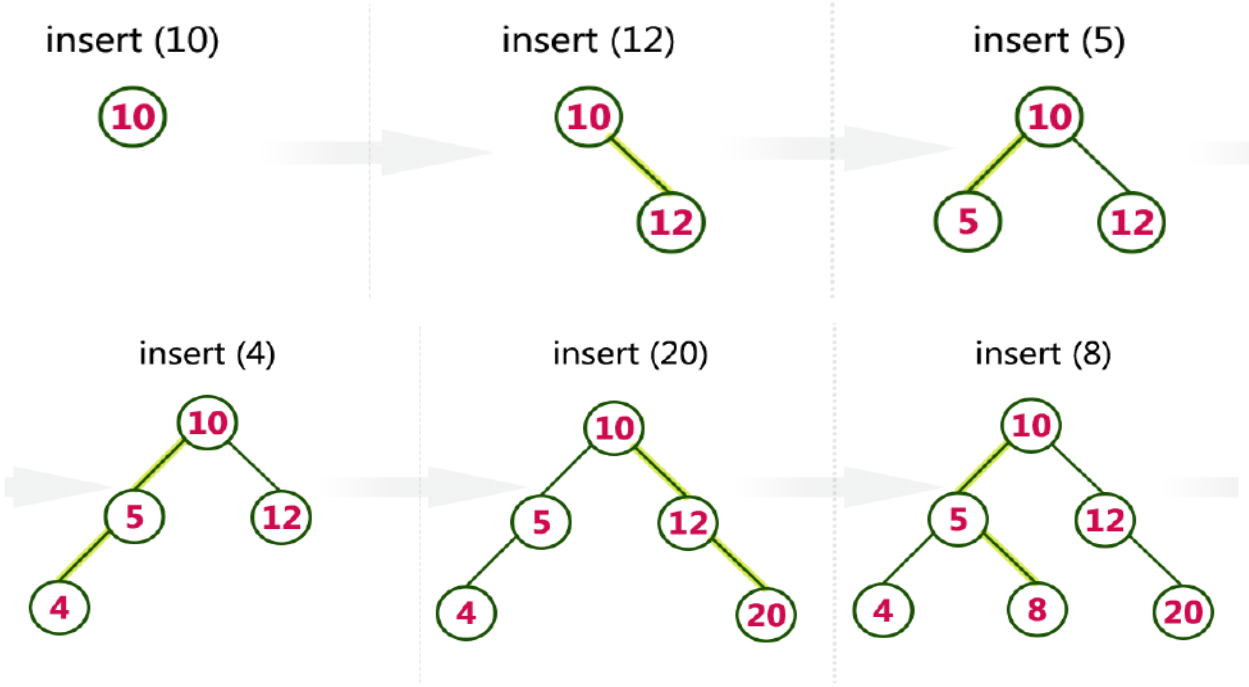
Step 7: Repeat the same process until node is deleted from the tree.

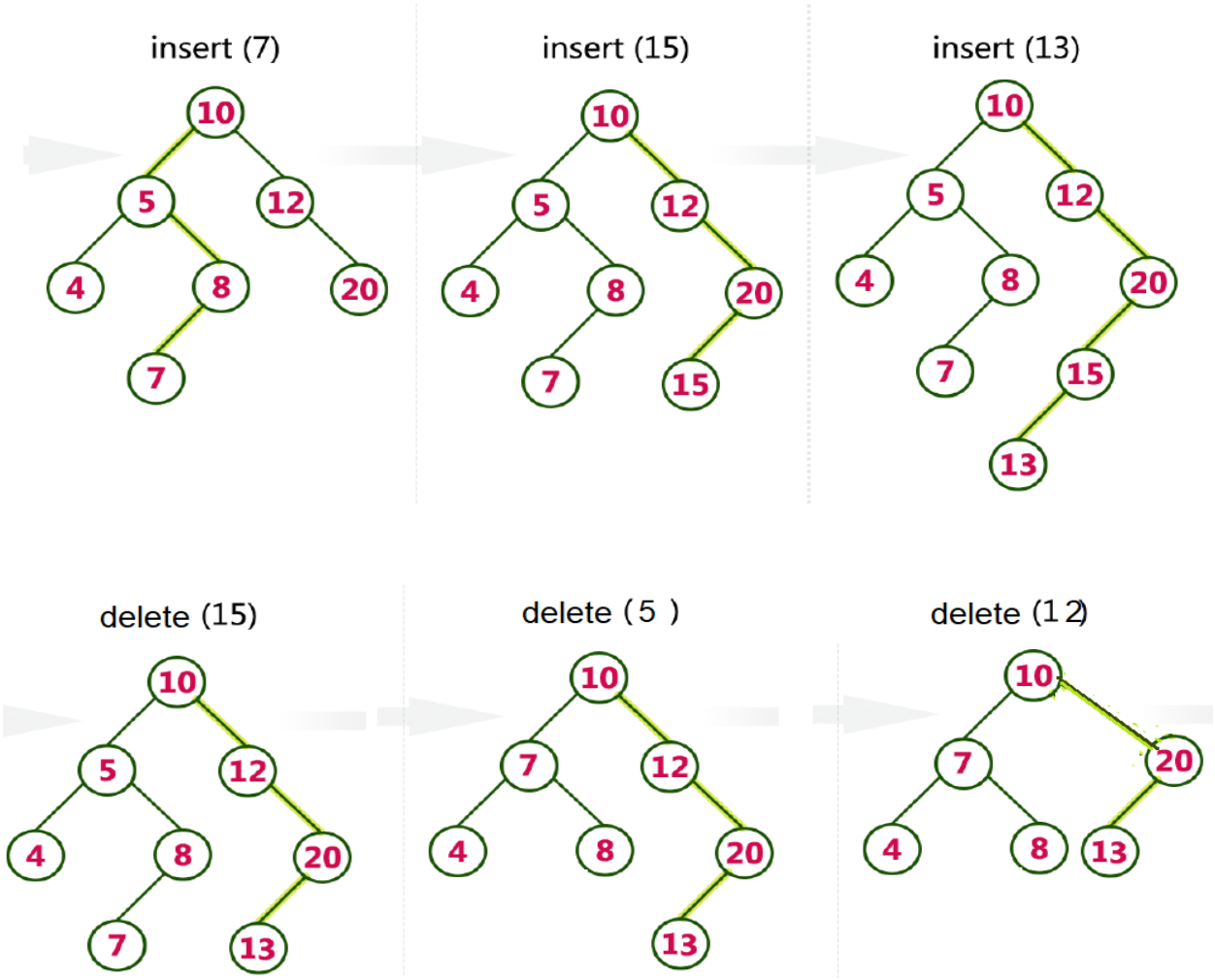




5. Construct a Binary Search Tree by inserting the following sequence of numbers...

10,12,5,4,20,8,7,15 and 13 Apply delete operation on the following key elements 13,20 and 5.



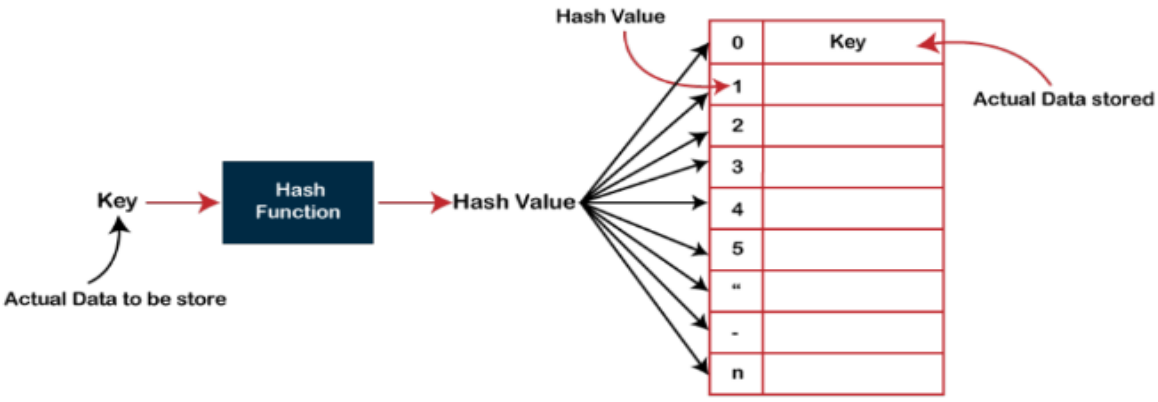


6. Explain the basic implementation of hash tables and the operations supported by hash tables, such as insertion, deletion and search.

In Hashing technique, the hash table and hash function are used. Using the hash function, we can calculate the address at which the value can be stored.

The main idea behind the hashing is to create the (key/value) pairs. If the key is given, then the algorithm computes the index at which the value would be stored. It can be written as:

Index = hash(key) % array-size.



The Hash table data structure stores elements in key-value pairs where

- **Key**- unique integer that is used for indexing the values
- **Value** - data that are associated with keys.

The hash table can be implemented with the help of an array, each location of hash table is referred as bucket. Hash table uses a special function known as a hash function that maps a given value with a key to a location in a hash table.

Hash function takes the key as an input and returns a small integer value as an output.

- The small integer value is called as a hash value.
- Hash value of the key is then used as an index for storing it into the hash table.

Operations of a hash table are

Search − Searches an element in a hash table.

Insert − inserts an element in a hash table.

Delete − Deletes an element from a hash table.

Assume a table with 8 slots:

Hash key = key % table size

36 % 8 = 4

18 % 8 = 2

72 % 8 = 0

[0]	72
[1]	
[2]	18
[3]	
[4]	36
[5]	
[6]	
[7]	

1. Search (Finding a Value by Key):

- Calculate the hash value of the key using the hash function.
- Use the hash value as the index to locate the potential value in the hash table.
- If the location contains the key, return the corresponding value.
- If the location is empty or contains a different key (collision), use a collision resolution technique to find the correct value.

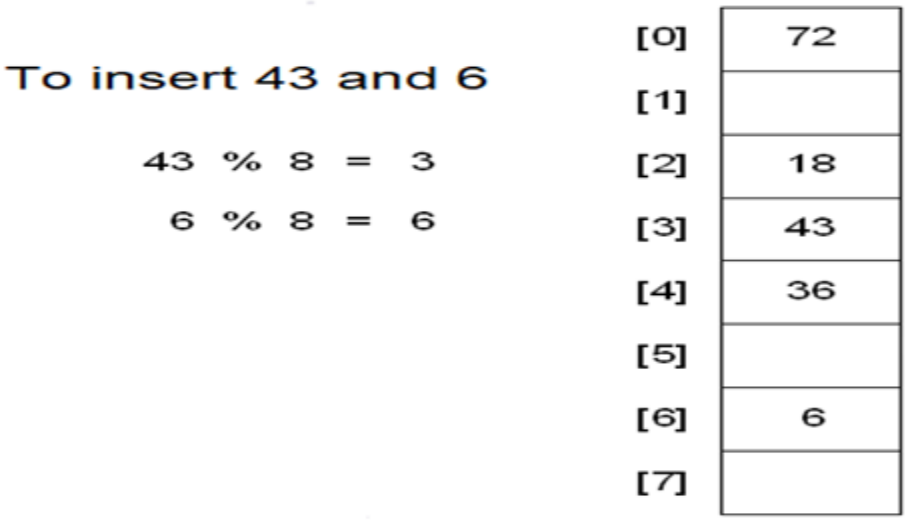
Example: To search a key value 36 hash value is computed, 36 % 8 = 4;

4 is used as index to locate the value in hash table.

2.Insertion (Adding a Key-Value Pair):

- Calculate the hash value of the key using the hash function.
- Use the hash value as the index to determine the location in the hash table where the value will be stored.
- If the location is empty, store the key-value pair there.
- If the location is occupied (collision), use a collision resolution technique (e.g., separate chaining, open addressing).

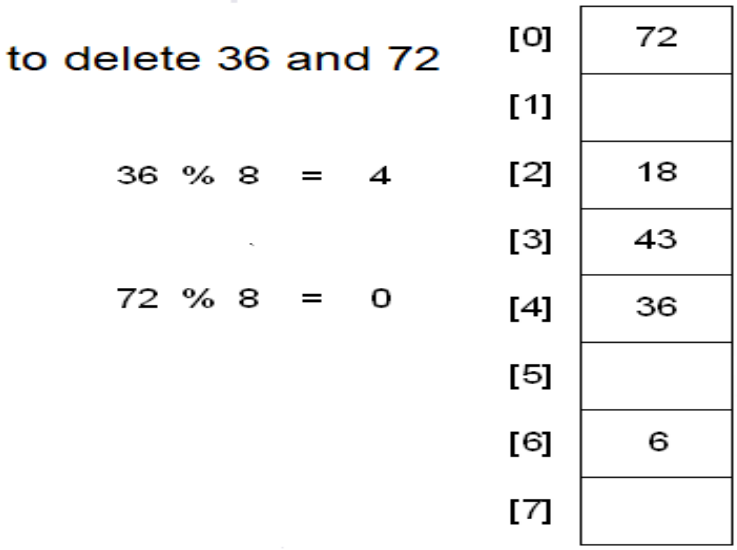
Example:



3. Deletion (Removing a Key-Value Pair):

- Calculate the hash value of the key using the hash function.
- Use the hash value as the index to locate the key-value pair in the hash table.
- If the location contains the key, remove the key-value pair.
- If the location is empty or contains a different key (collision), use a collision resolution technique to find the correct key-value pair to delete.

Example



7. What is hash function? Explain the different hash functions with examples.

Types of Hash Functions

The primary types of hash functions are:

Division Method.

Mid Square Method.

Folding Method.

Multiplication Method.

Division Method

The easiest and quickest way to create a hash value is through division. The k-value is divided by M in this hash function, and the result is used.

Formula:

h(K) = k mod M

(where k = key value and M = the size of the hash table)

Advantages:

This method is effective for all values of M.

The division strategy only requires one operation, thus it is quite quick.

Disadvantages:

Since the hash table maps consecutive keys to successive hash values, this could result in poor performance.

There are times when exercising extra caution while selecting M's value is necessary.

Example of Division Method

k = 1987

M = 13

h(1987) = 1987 mod 13

h(1987) = 4

Mid Square Method

The following steps are required to calculate this hash method:

k*k, or square the value of k

Using the middle r digits, calculate the hash value.

Formula:

h(K) = h(k x k)

(where k = key value)

Advantages:

This technique works well because most or all of the digits in the key value affect the result.

All of the necessary digits participate in a process that results in the middle digits of the squared result.

The result is not dominated by the top or bottom digits of the initial key value.

Disadvantages:

The size of the key is one of the limitations of this system; if the key is large, its square will contain twice as many digits.

Probability of collisions occurring repeatedly.

Example of Mid Square Method

k = 60

Therefore,k = k x k

k = 60 x 60

k = 3600

Thus, h(60) = 60

Folding Method

The process involves two steps:

except for the last component, which may have fewer digits than the other parts, the key-value k should be divided into a predetermined number of pieces, such as k1, k2, k3,..., kn, each having the same amount of digits.

Add each element individually. The hash value is calculated without taking into account the final carry, if any.

Formula:

k = k1, k2, k3, k4, ….., kn

s = k1+ k2 + k3 + k4 +….+ kn

h(K)= s

(Where, s = addition of the parts of key k)

Advantages:

Creates a simple hash value by precisely splitting the key value into equal-sized segments.

Without regard to distribution in a hash table.

Disadvantages:

When there are too many collisions, efficiency can occasionally suffer.

Example of Folding Method

k = 12345

k1 = 67; k2 = 89; k3 = 12

Therefore,s = k1 + k2 + k3

s = 67 + 89 + 12

s = 168

Multiplication Method

Determine a constant value. A, where (0, A, 1)

Add A to the key value and multiply.

Consider kA's fractional portion.

Multiply the outcome of the preceding step by M, the hash table's size.

Formula:

h(K) = floor (M (kA mod 1))

(Where, M = size of the hash table, k = key value and A = constant value)

Advantages:

Any number between 0 and 1 can be applied to it, however, some values seem to yield better outcomes than others.

Disadvantages:

The multiplication method is often appropriate when the table size is a power of two since multiplication hashing makes it possible to quickly compute the index by key.

Example of Multiplication Method

k = 5678

A = 0.6829

M = 200

Now, calculating the new value of h(5678):

h(5678) = floor[200(5678 x 0.6829 mod 1)]

h(5678) = floor[200(3881.5702 mod 1)]

h(5678) = floor[200(0.5702)]

h(5678) = floor[114.04]

h(5678) = 114

So, with the updated values, h(5678) is 114

If the key is 123, the table size is 10, and A is 0.618, then you calculate (123 * 0.618) % 1, get the fractional part, multiply by 10, and take the floor value to get the index.

8.What is a collision? What are the different collision resolution techniques? Explain Separate chaining with examples.

Collision Resolution

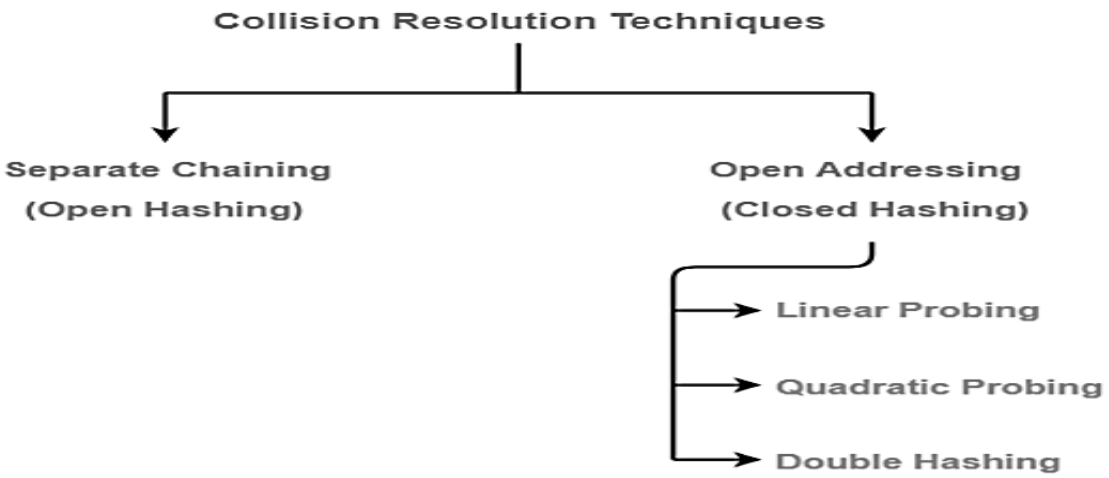
When two keys or hash values compete with a single hash table slot, then Collision occur. To resolve collision, we use collision resolution techniques. Collisions can be reduced with a selection of a good hash function.

Collision Resolution Techniques

There are two types of collision resolution techniques.

Separate chaining (open hashing)

Open addressing (closed hashing)



Separate Chaining

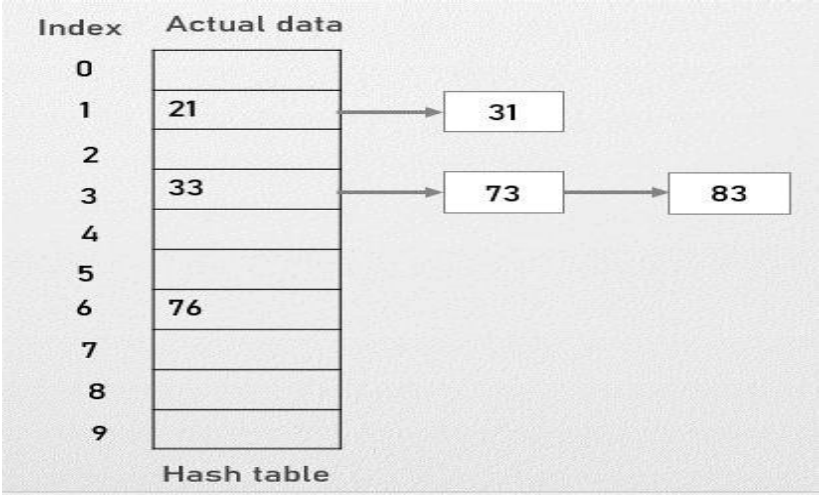
To handle the collision,

–This technique creates a linked list to the slot for which collision occurs.

–The new key is then inserted in the linked list.

–These linked lists to the slots appear like chains.

–That is why, this technique is called as separate chaining.



For Searching

In worst case, all the keys might map to the same bucket of the hash table.

In such a case, all the keys will be present in a single linked list.

Sequential search will have to be performed on the linked list to perform the search.

Worst case complexity for searching is O(n).

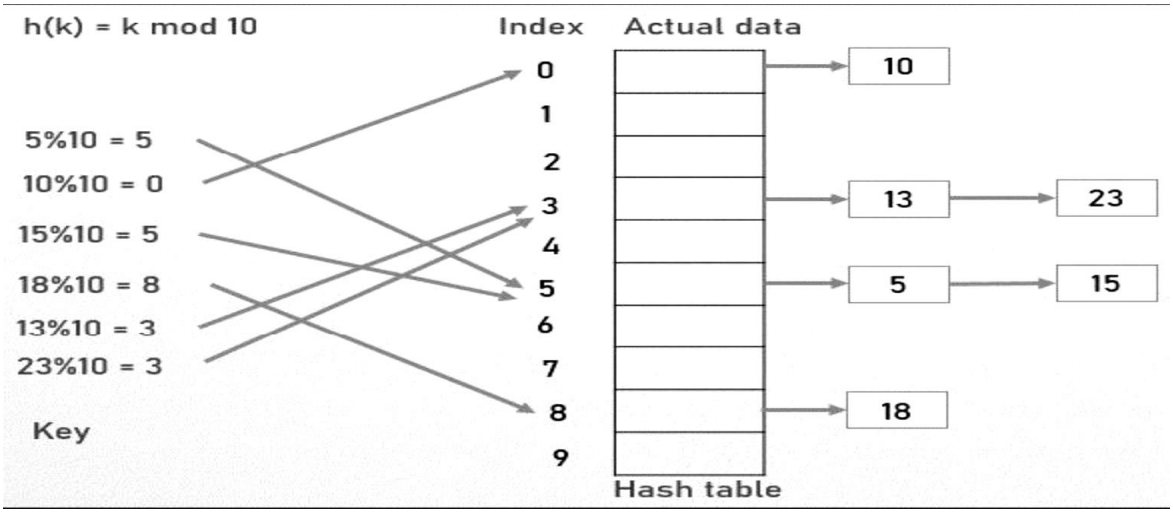
For Deletion

In worst case, the key might have to be searched first and then deleted.

In worst case, time taken for searching is O(n).

Worst case complexity for deletion is O(n).

Example-Separate Chaining



Advantages of separate chaining

- It is easy to implement.
- The hash table never fills full, so we can add more elements to the chain.
- It is mostly used when it is unknown how many and how frequently keys may be inserted or
- deleted.
- It is less sensitive to the function of the hashing.

Disadvantages of separate chaining

- The cache performance of chaining is not good.
- The memory wastage is too much in this method.
- It requires more space for element links.
- If the chain becomes long, then search time can become O(n) in the worst case.

9.What is a collision? What are the different collision resolution techniques? Explain open addressing with examples.

Collision:

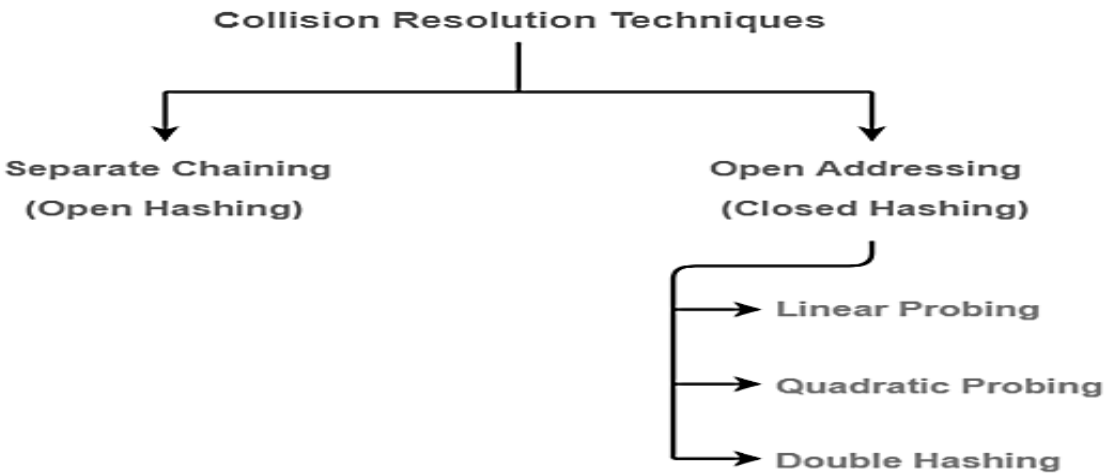
When two keys or hash values compete with a single hash table slot, then Collision occur. To resolve collision, we use collision resolution techniques. Collisions can be reduced with a selection of a good hash function.

Collision Resolution Techniques

There are two types of collision resolution techniques.

Separate chaining (open hashing)

Open addressing (closed hashing)



Open Addressing

•In open addressing,

–Unlike separate chaining, all the keys are stored inside the hash table.

–No key is stored outside the hash table.

•Techniques used for open addressing are-

–Linear Probing

–Quadratic Probing

–Double Hashing

Operations in Open Addressing

•Insert Operation:

–Hash function is used to compute the hash value for a key to be inserted.

–Hash value is then used as an index to store the key in the hash table.

•In case of collision,

–Probing is performed until an empty bucket is found.

–Once an empty bucket is found, the key is inserted.

–Probing is performed in accordance with the technique used for open addressing.

Search Operation:

•To search any particular key,

–Its hash value is obtained using the hash function used.

–Using the hash value, that bucket of the hash table is checked.

–If the required key is found, the key is searched.

–Otherwise, the subsequent buckets are checked until the required key or an empty bucket is found.

–The empty bucket indicates that the key is not present in the hash table.

Delete Operation:

–The key is first searched and then deleted.

–After deleting the key, that particular bucket is marked as “deleted”.

1. Linear Probing

In linear probing,

–When collision occurs, we linearly probe for the next bucket.

–We keep probing until an empty bucket is found.

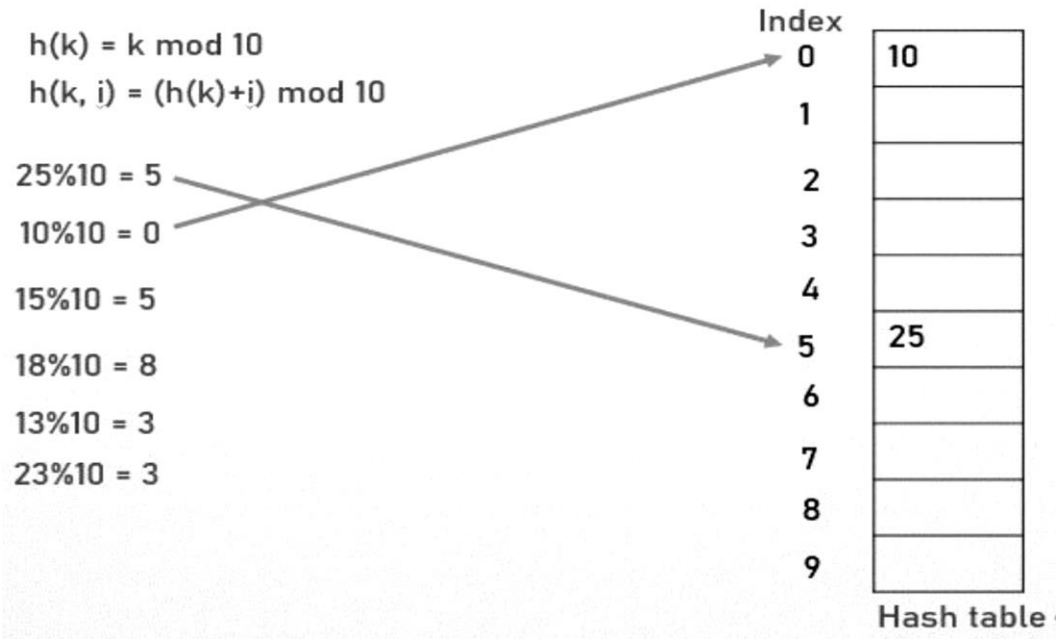
–Let hash(x) be the slot index computed using a hash function and n be the table size

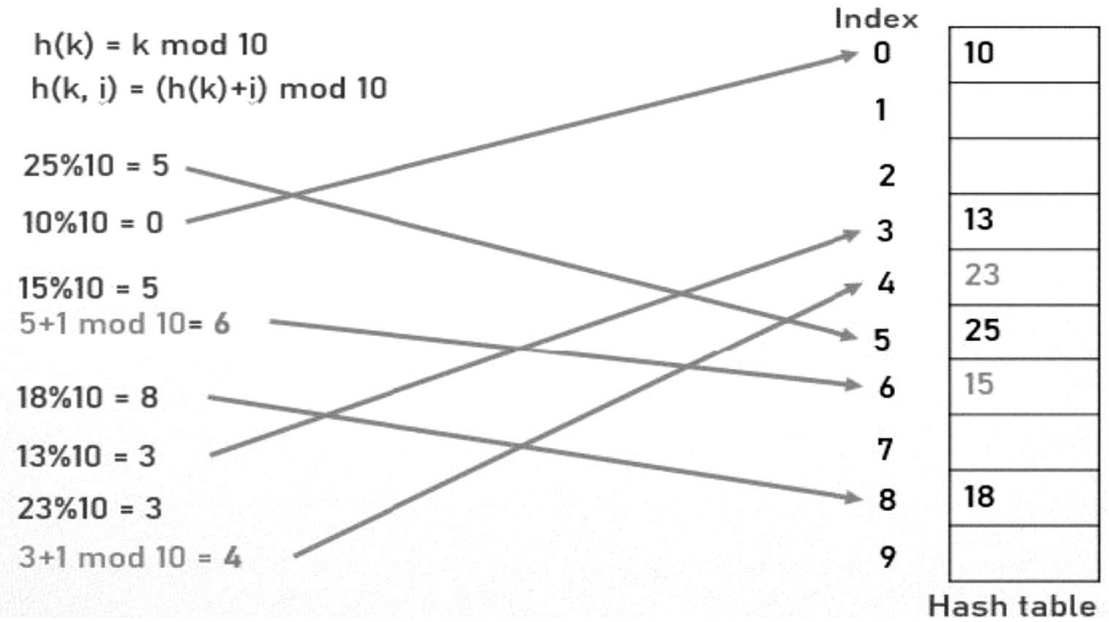
–If slot hash(x) % n is full, then we try (hash(x) + 1) % n

–If (hash(x) + 1) % n is also full, then we try (hash(x) + 2) % n

–If (hash(x) + 2) % n is also full, then we try (hash(x) + 3) % n so on…

Example: Linear Probing





•**Advantage-**

–It is easy to compute.

•**Disadvantage-**

–The main problem with linear probing is clustering.

–Many consecutive elements form groups.

–Then, it takes time to search an element or to find an empty bucket.

2. Quadratic Probing

In quadratic probing,

–When collision occurs, we probe for i^2'th bucket in ithiteration.

–We keep probing until an empty bucket is found.

–In quadratic probing the sequence is that $H+1^2$, $H+2^2$, $H+3^2$,....$H+K^2$

–The hash function for quadratic probing is

$h_i(X)$ = (Hash(X) + $(i)^2$) % Table Size for i = 0, 1, 2, 3,...etc.

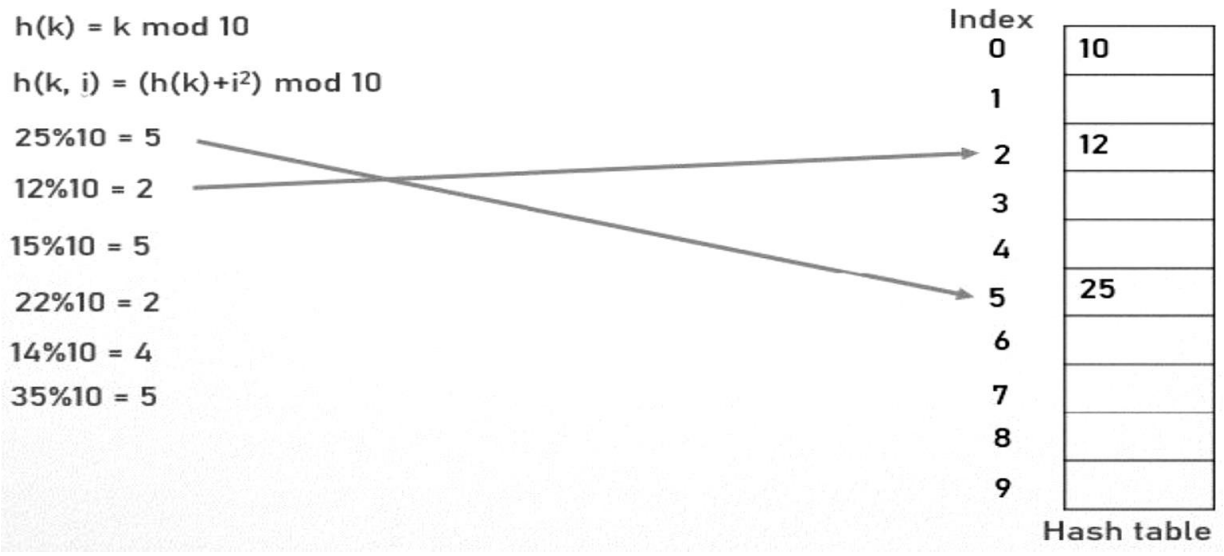
–If the slot hash(x) % S is full, then we try (hash(x) + 1*1) % S.

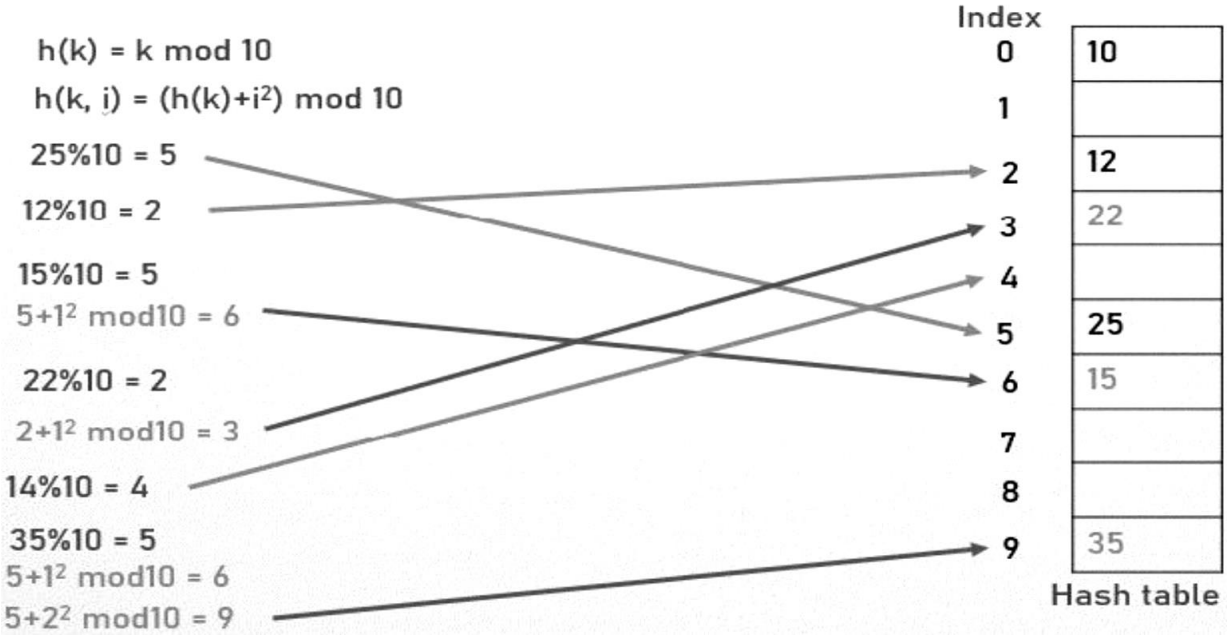
–If (hash(x) + 1*1) % S is also full, then we try (hash(x) + 2*2) % S.

–If (hash(x) + 2*2) % S is also full, then we try (hash(x) + 3*3) % S.

–This process continue until an empty slot is found.

Example : Quadratic probing





Advantage:

– Primary clustering problem resolved

Disadvantage:

– Secondary clustering

– No guarantee for finding slots

3.Double Hashing

Double hashing uses two hash functions,

– one to find the initial location to place the key and a

– second to determine the size of the jumps in the probe sequence.

–The ith probe is

h(k, i) = (hash1(key) + i * hash2(key)) mod TABLE_SIZE

Here hash1() and hash2() are hash functions

First hash function is:

hash1(key) = key mod TABLE_SIZE

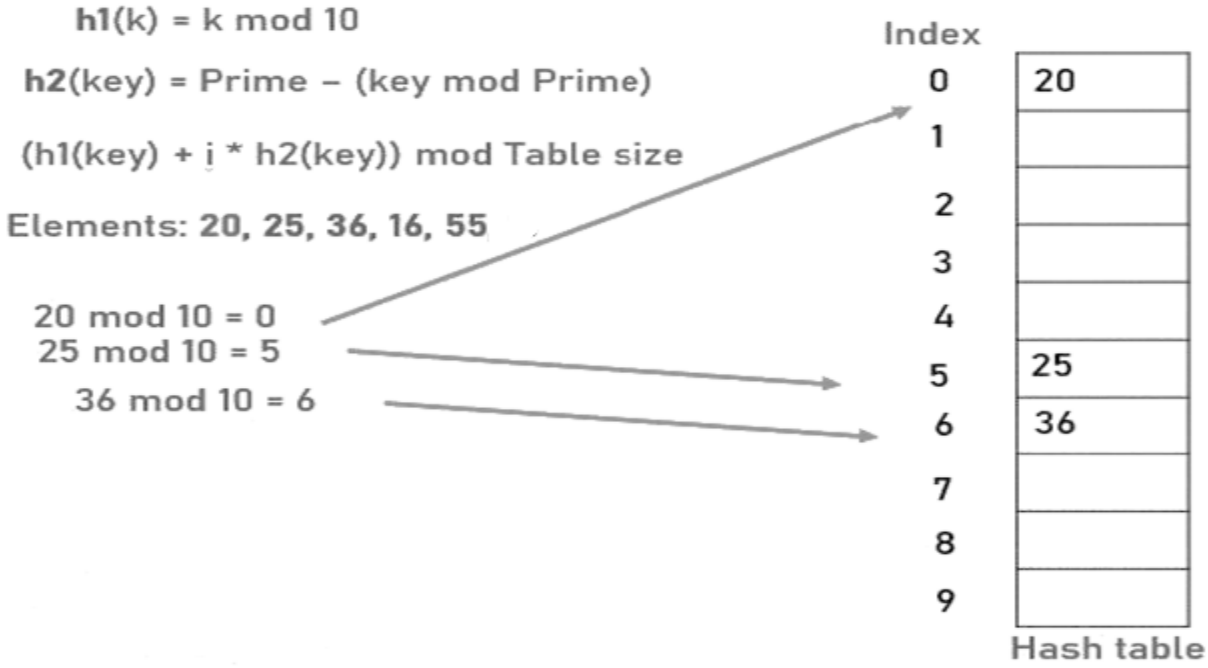
Second hash function is :

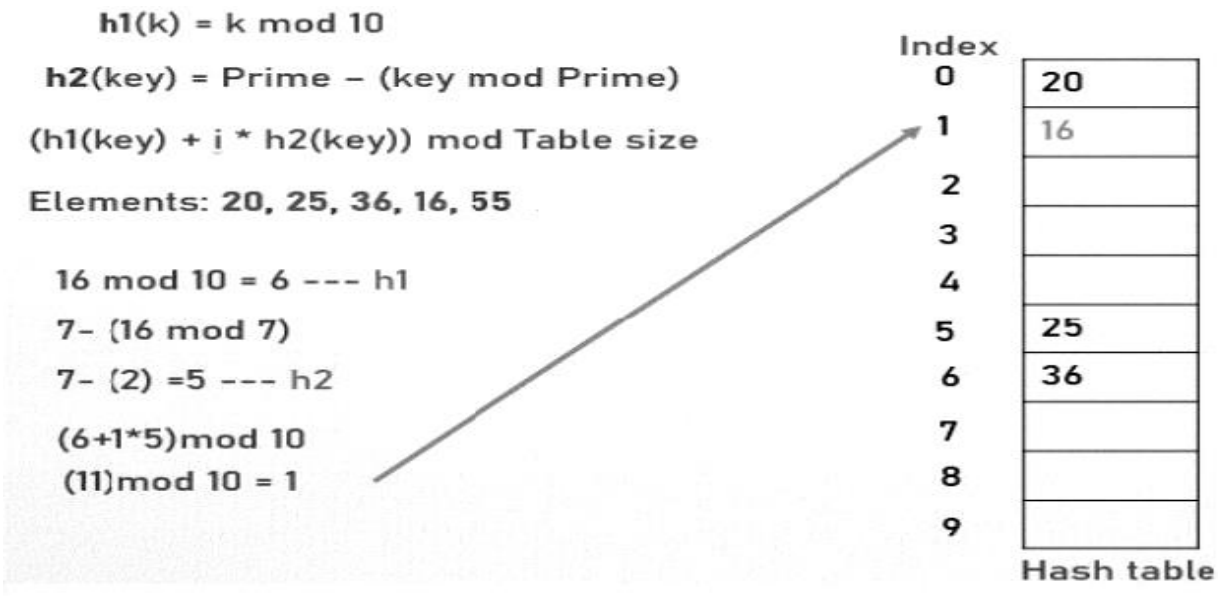
hash2(key) = PRIME – (key mod PRIME)

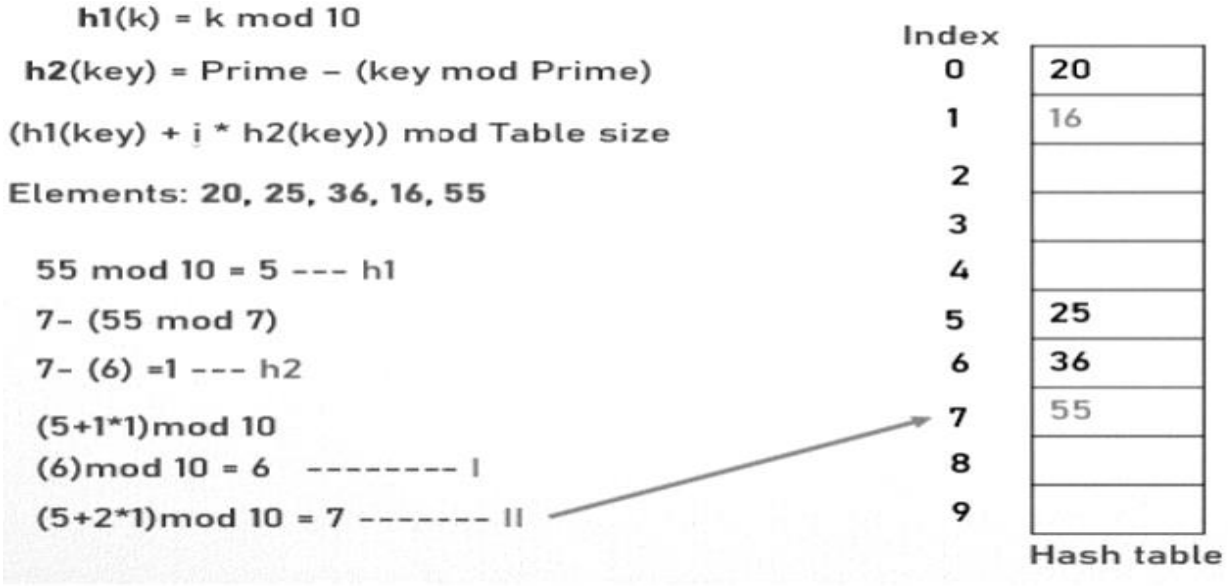
Where PRIME is a prime smaller than the TABLE_SIZE

Example: Double hashing

Elements: 20, 25, 36, 16, 55, 17 table size= 10 prime number = 7







Double hashing highlights

– Computational cost is higher.

– No primary clustering.

– No secondary clustering.

– Double hashing can find the next free slot faster than the linear probing approach.

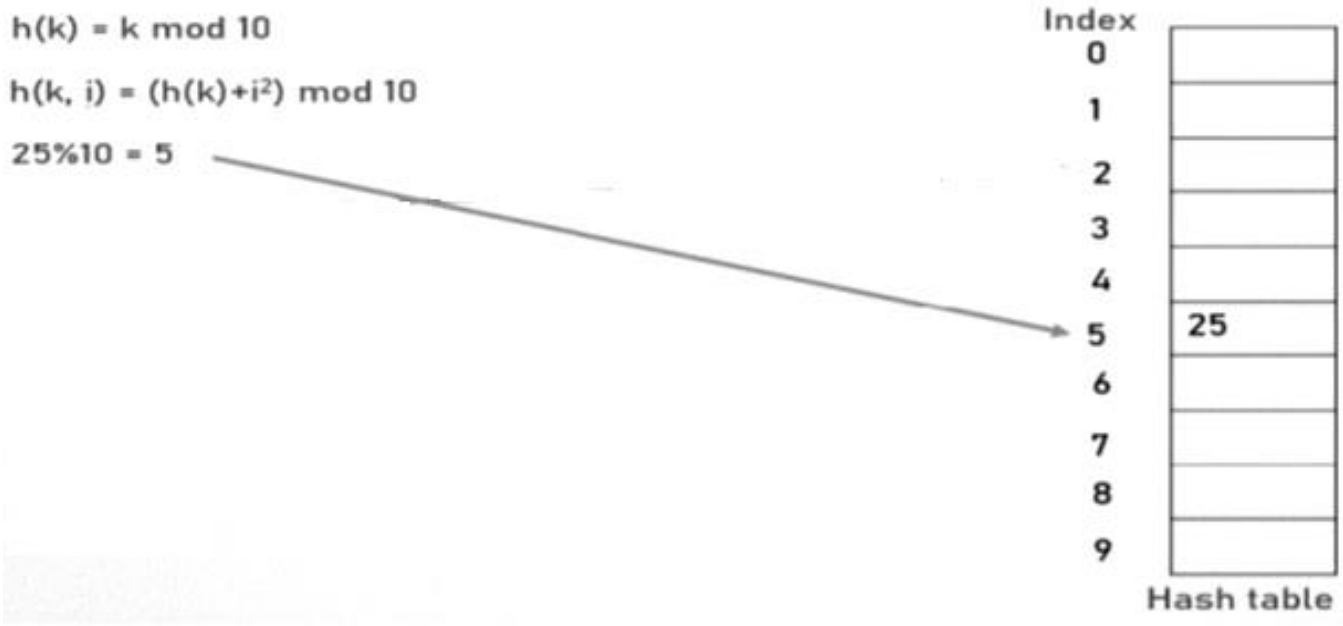
– Double hashing is used for uniform distribution of records throughout a hash table.

– Double hashing is useful if an application requires a smaller hash table.

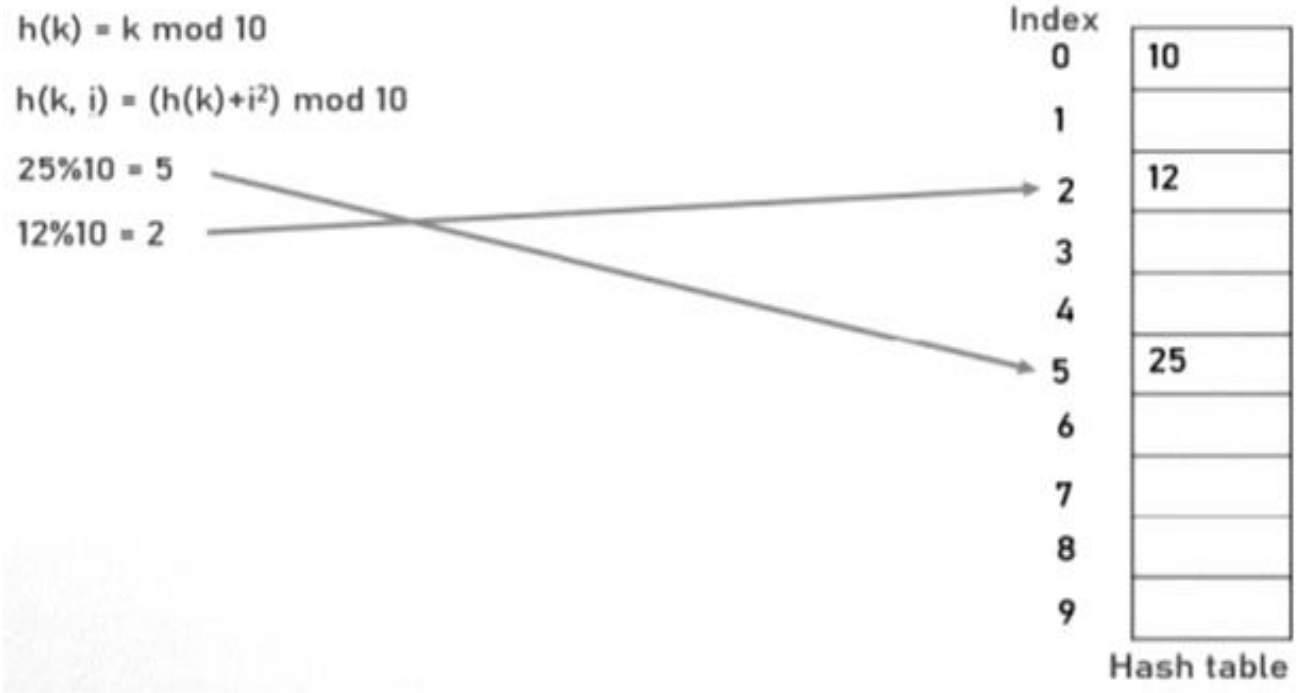
10.using the hash function key mod 10 insert the following sequence of keys in the hash table

25,12,15,22,14 and 35 use quadratic probing technique for collision resolution.

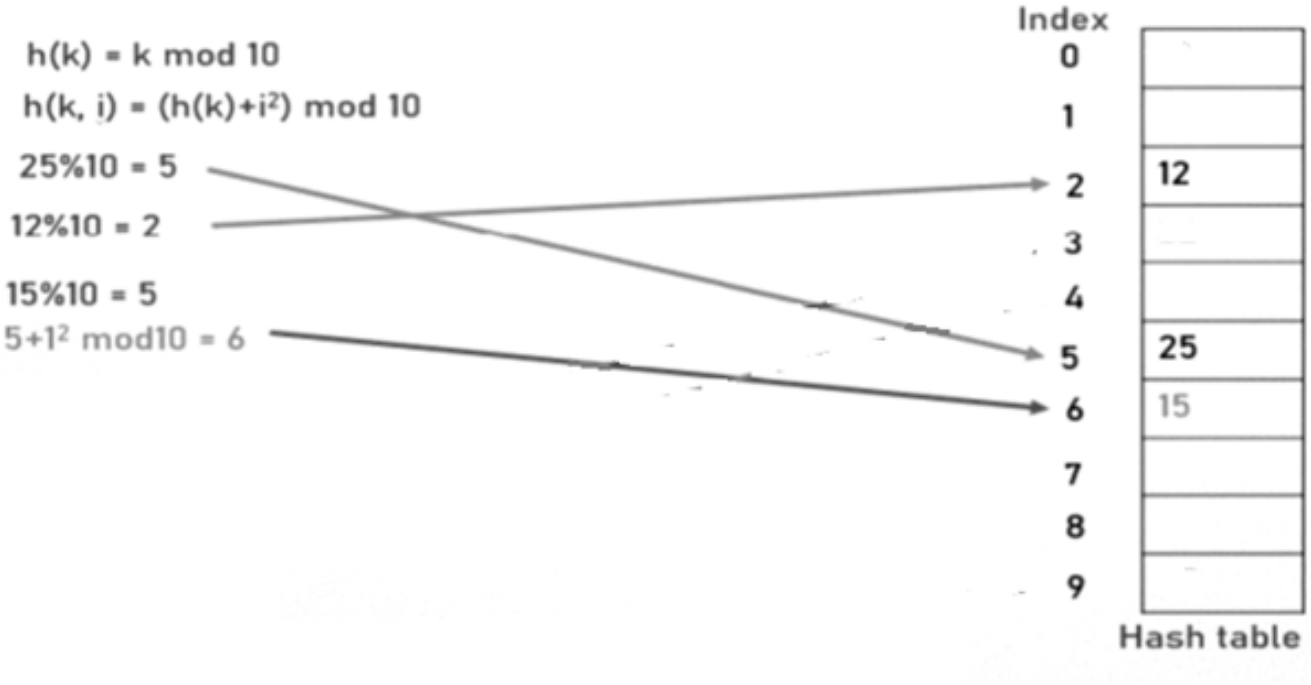
Step-1



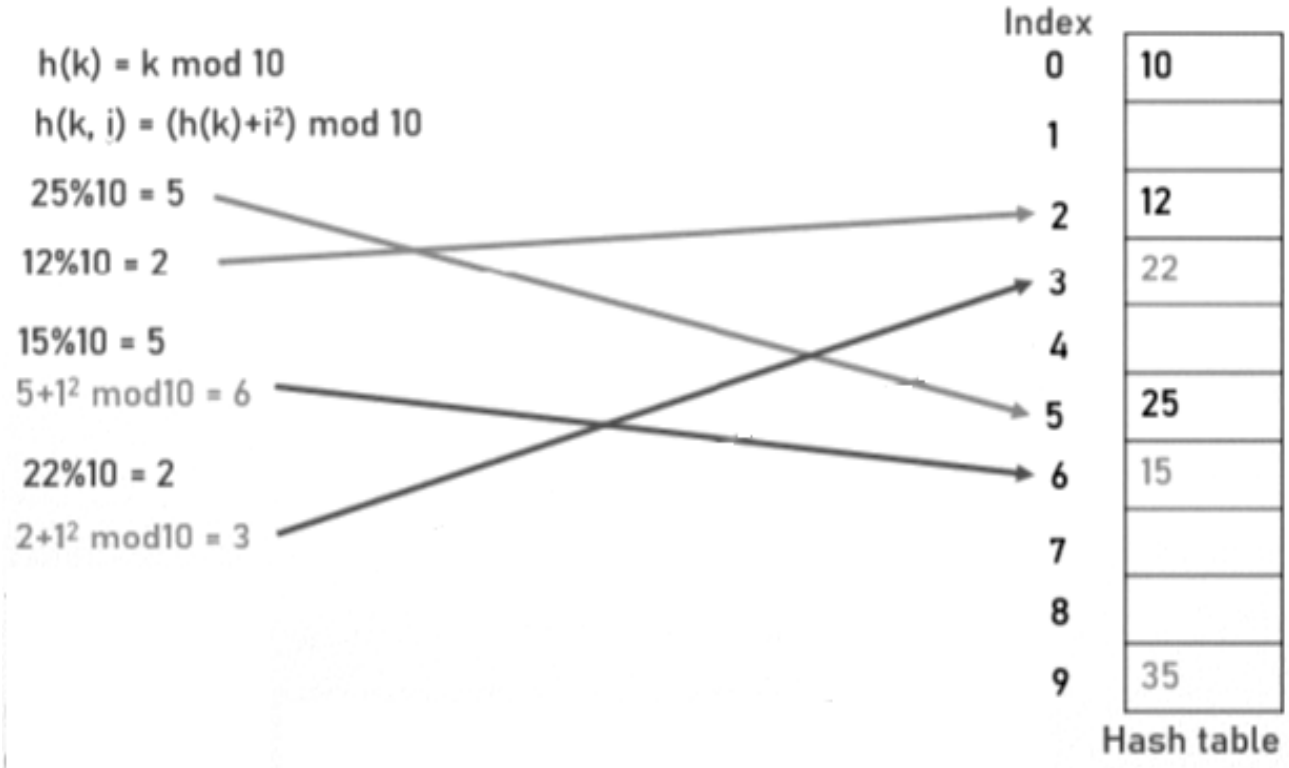
Step-2



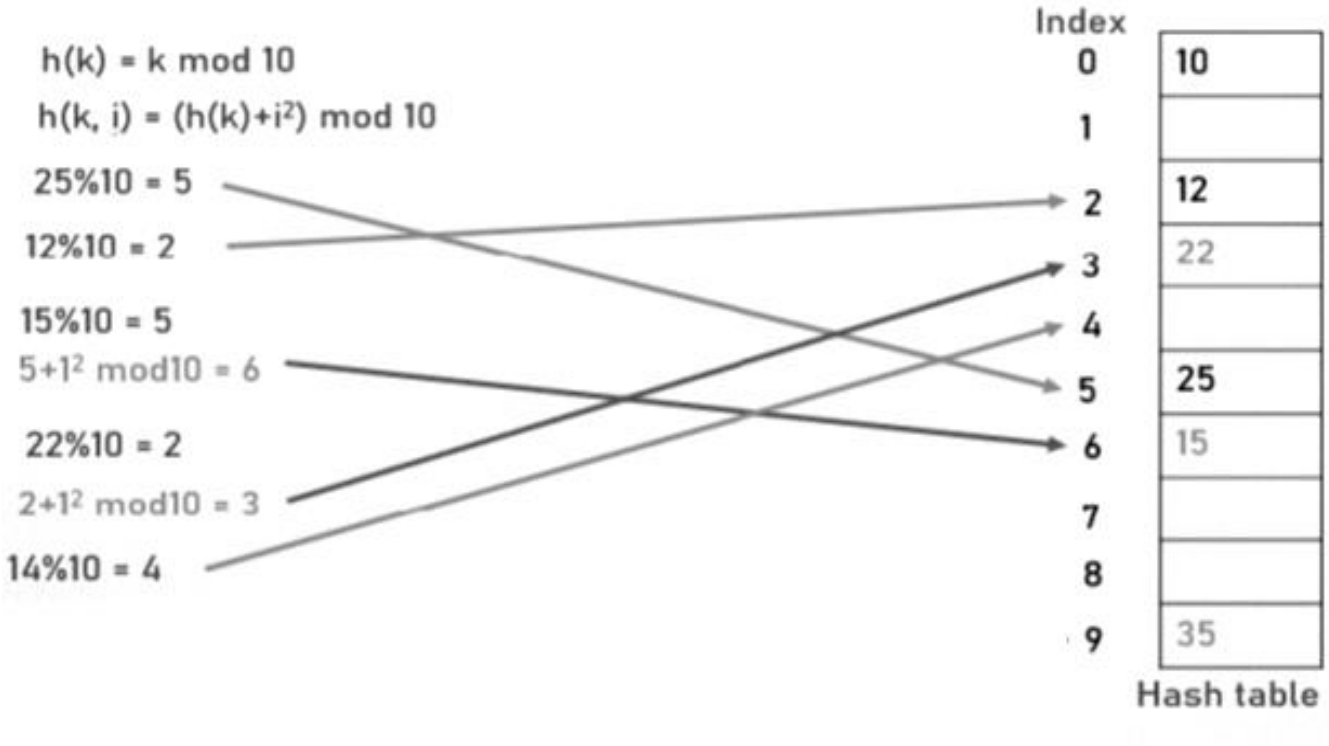
Step-3


h(k) = k mod 10
h(k, i) = (h(k)+i²) mod 10
25%10 = 5
12%10 = 2
15%10 = 5
5+1² mod10 = 6
Index
0
1
2
3
4
5
6
7
8
9
12
25
15
Hash table

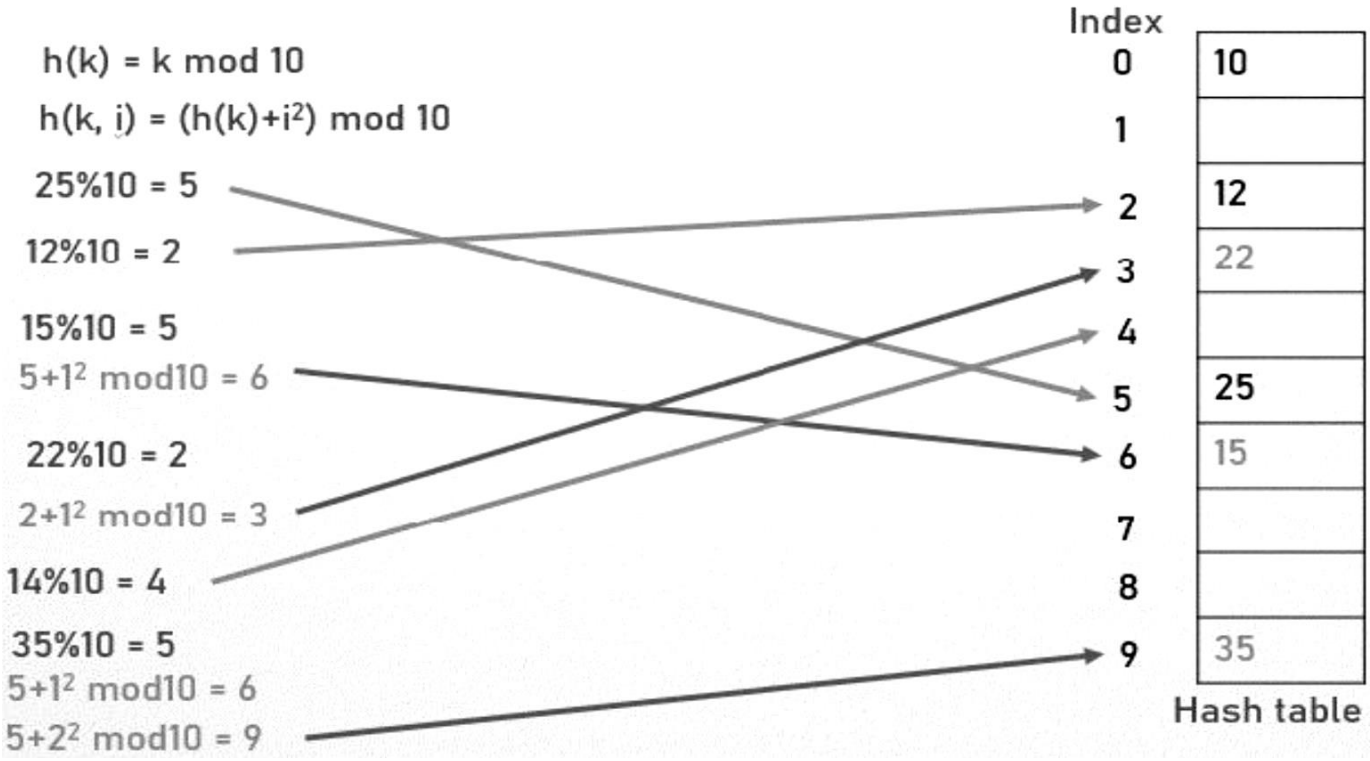
Step-4


h(k) = k mod 10
h(k, i) = (h(k)+i²) mod 10
25%10 = 5
12%10 = 2
15%10 = 5
5+1² mod10 = 6
22%10 = 2
2+1² mod10 = 3
Index
0
1
2
3
4
5
6
7
8
9
10
12
22
25
15
35
Hash table

Step-5


h(k) = k mod 10
h(k, i) = (h(k)+i²) mod 10
25%10 = 5
12%10 = 2
15%10 = 5
5+1² mod10 = 6
22%10 = 2
2+1² mod10 = 3
14%10 = 4
Index
0
1
2
3
4
5
6
7
8
9
10
12
22
25
15
35
Hash table

Step-6


h(k) = k mod 10
h(k, i) = (h(k)+i²) mod 10
25%10 = 5
12%10 = 2
15%10 = 5
5+1² mod10 = 6
22%10 = 2
2+1² mod10 = 3
14%10 = 4
35%10 = 5
5+1² mod10 = 6
5+2² mod10 = 9
Index
0
1
2
3
4
5
6
7
8
9
10
12
22
25
15
35
Hash table