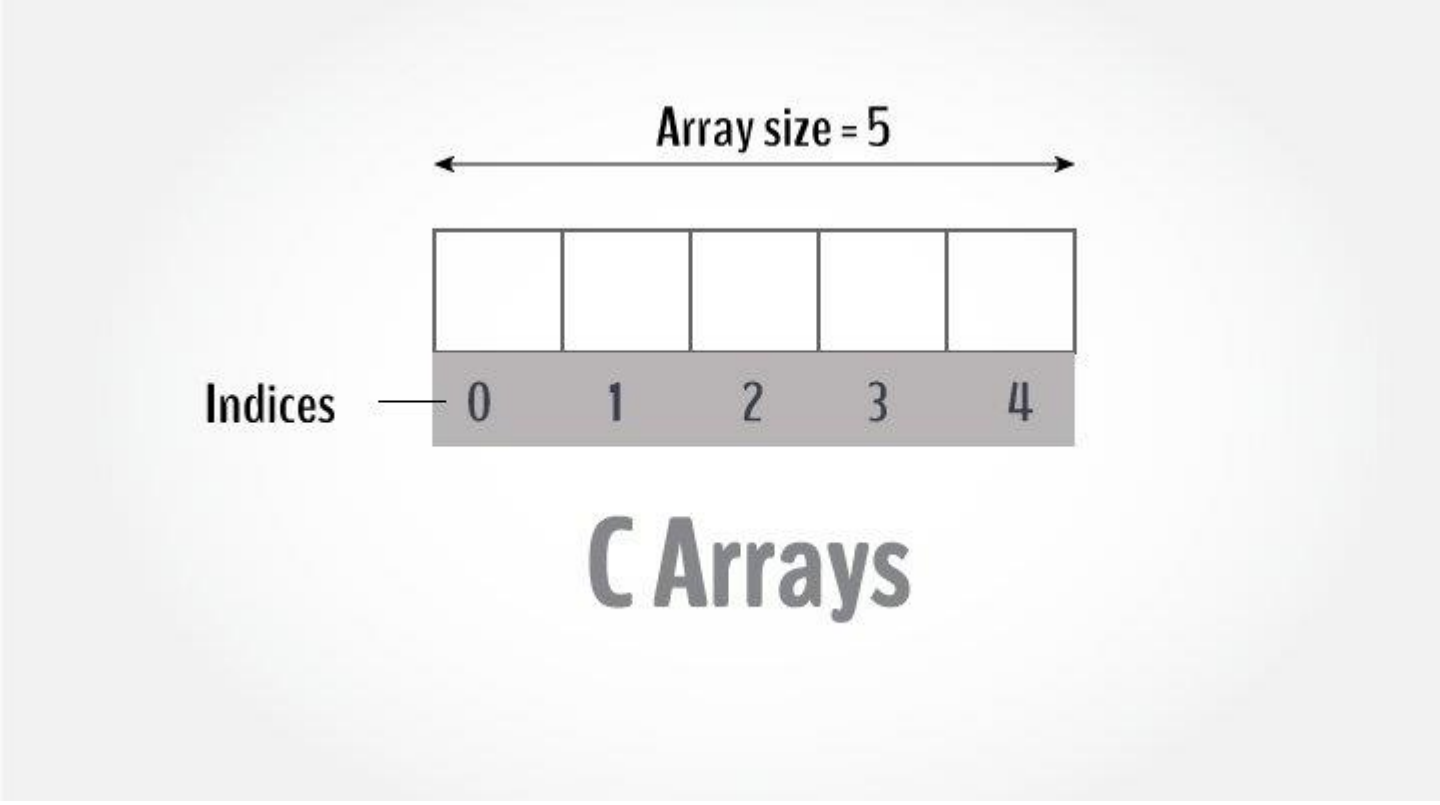
Unit 1 (Introduction to Data structures)

What are data structures

1: Data structures are ways of organizing and storing data in a computer so that it can be accessed and modified efficiently.
2: Data structures are fundamental to computer science and are used in a wide range of applications, from basic programming tasks to complex algorithms and systems.
3: Data structures can be classified into two main categories: linear and non-linear. Linear data structures include arrays, linked lists, and stacks, while non-linear data structures include trees and graphs.
4: Each data structure has its own set of advantages and use cases, and choosing the right one for a particular task can have a significant impact on the performance of an algorithm or program.

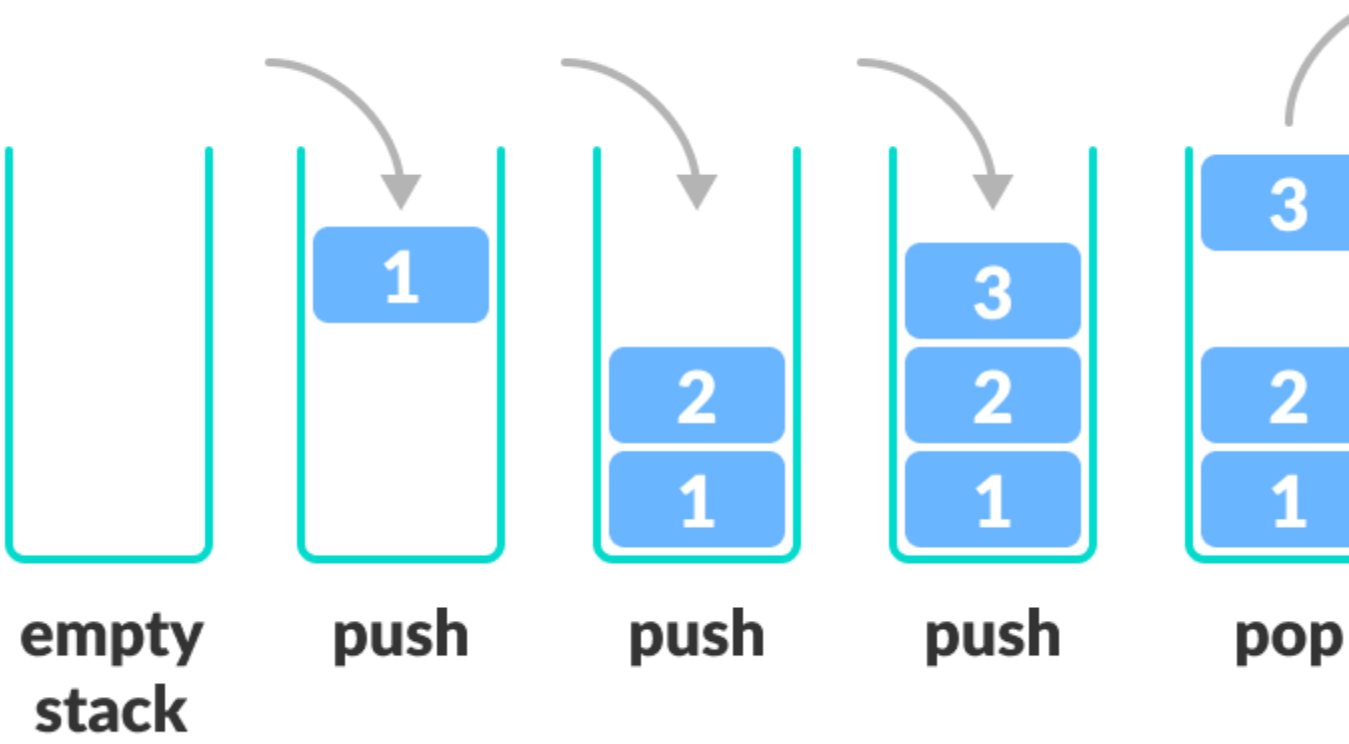
Elementary data organization

1: Arrays: An array is a collection of elements that are stored in contiguous memory locations. Elements can be accessed and modified by their index.



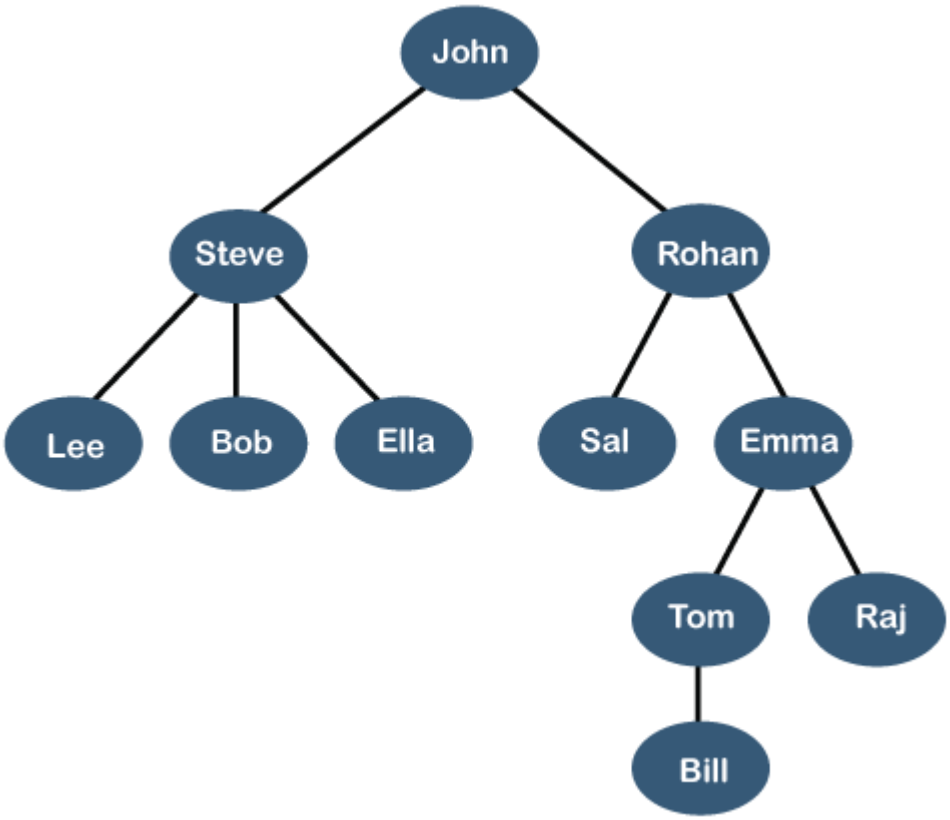
2: Linked Lists: A linked list is a collection of elements called nodes, where each node contains a value and a reference to the next node. The first node is called the head, and the last node is called the tail.

3: Stacks: A stack is a collection of elements that follows the Last In First Out (LIFO) principle. Elements can only be added or removed from the top of the stack.

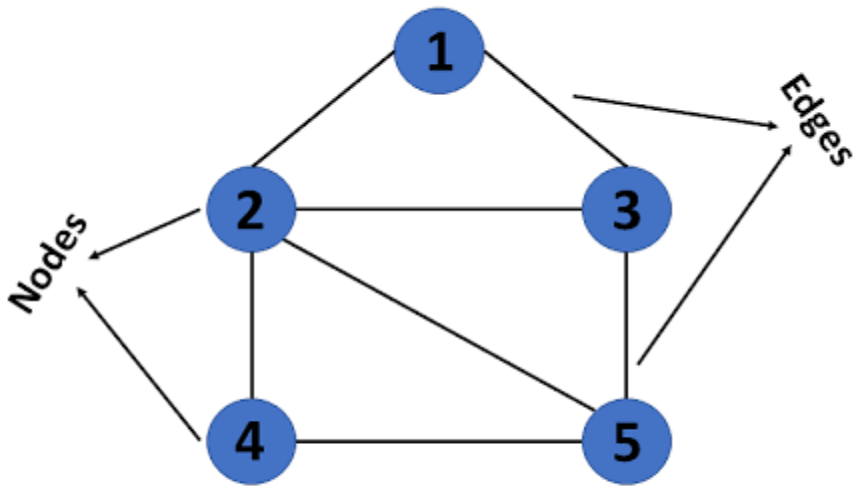


4 : Queues: A queue is a collection of elements that follows the First In First Out (FIFO) principle. Elements can only be added to the rear and removed from the front of the queue.

5: Trees: A tree is a non-linear data structure that consists of nodes and edges. Each node can have one or more child nodes and one parent node. The node without a parent is called the root node.



6: Graphs: A graph is a non-linear data structure that consists of nodes and edges. Each node can be connected to one or more other nodes, and the connections between nodes are called edges. Graphs can be directed or undirected.



---

traversing

1: Traversing in data structures refers to the process of visiting each element of a data structure in a specific order.

2: The order in which the elements are visited can vary depending on the type of data structure and the specific traversal algorithm being used.

3: Breadth-first search (BFS): This algorithm visits all the elements at the current level before moving on to the next level. It is commonly used to traverse trees and graphs.

4: Depth-first search (DFS): This algorithm visits all the elements at the current level before moving on to the next level. It is commonly used to traverse trees and graphs.

5: In-order traversal: This algorithm is used to traverse binary trees. It visits the left subtree, the root, and then the right subtree.

6: Pre-order traversal: This algorithm is used to traverse binary trees. It visits the root, the left subtree, and then the right subtree.

7: Post-order traversal: This algorithm is used to traverse binary trees. It visits the left subtree, the right subtree, and then the root.

Inserting

1: Inserting refers to the process of adding new data elements to a data structure.

2: The method of insertion can vary depending on the type of data structure and the specific algorithm being used

3: Array: Inserting into an array requires shifting all elements after the insertion point to make room for the new element.

4: Linked List: Insertion into a linked list is relatively simple, as it only requires changing the pointers of the previous and next nodes to include the new node.

5: Stack: Insertion into a stack is also simple, and it's known as push operation. The new element is added to the top of the stack.

6: Queue: Insertion into a queue is also simple and it's known as enqueue operation. The new element is added to the rear of the queue.

7: Tree: Insertion into a tree can be more complex, depending on the type of tree. For example, in a binary search tree, the new element must be inserted in the correct position in the tree to maintain the ordering property.

8: Hash table: Insertion into a hash table is relatively simple, as it only requires adding the new element to the correct position in the table based on its hash value.

Deleting

1: Deleting refers to the process of removing data elements from a data structure.

2: The method of deletion can vary depending on the type of data structure and the specific algorithm being used.

3: Array: Deleting an element from an array requires shifting all elements after the deletion point to close the gap left by the deleted element.

4: Linked List: Deleting an element from a linked list requires changing the pointers of the previous and next nodes to remove the element from the list.

5: Stack: Deleting an element from a stack is also simple, and it's known as pop operation. The top element is removed from the stack.

6: Queue: Deleting an element from a queue is also simple and it's known as dequeue operation. The front element is removed from the queue.

7: Tree: In a binary search tree, the element to be deleted must be located and then replaced by its in-order predecessor or successor, to maintain the ordering property.

8: Hash table: Deleting an element from a hash table requires removing the element from the correct position in the table based on its hash value and also updating the corresponding pointers or index.

Searching

1: Searching refers to the process of finding a specific data element in a data structure.

2: The method of searching can vary depending on the type of data structure and the specific algorithm being used.

3: Array**:** Searching for an element in an array can be done by iterating through the array and comparing each element to the search key.

4: Linked List**:** Searching for an element in a linked list can be done by iterating through the list and comparing each element to the search key.

5: Stack**:** Searching for an element in a stack can be done by iterating through the stack and comparing each element to the search key.

6: Queue: Searching for an element in a queue can be done by iterating through the queue and comparing each element to the search key.

7: Tree: In a binary search tree, the search can be done by starting at the root and then traversing the left or right subtree based on whether the search key is less than or greater than the current node.

8: Hash table: Searching for an element in a hash table can be done by computing the hash value of the search key and then looking up the corresponding position in the table**.**

Sorting

1: Sorting refers to the process of arranging data elements in a specific order, such as ascending or descending order.

2: The method of sorting can vary depending on the type of data structure and the specific algorithm being used.

3: Array: Sorting an array can be done using various algorithms such as bubble sort, insertion sort, selection sort, merge sort, quick sort, and so on.

4: linked list: Sorting a linked list can be done using merge sort.

5: Tree: Sorting elements in a tree can be done by traversing the tree in-order, which visits the left subtree, the root, and then the right subtree, resulting in elements sorted in ascending order.

6: Hash table: Sorting elements in a hash table can be done by extracting all the elements and applying a sorting algorithm to them.

Merging

1: Merging refers to the process of combining two or more data structures or sets of data into a single data structure.

2: The method of merging can vary depending on the type of data structures being merged and the specific algorithm being used.

3: Array: Merging two arrays can be done by merging them in a specific order. This can be done using various algorithms such as merge sort algorithm.

4: Linked List: Merging two linked lists can be done by iterating through both lists and adding elements to a new list in the correct order.

5: Stack: Merging two stacks can be done by concatenating them or by merging them in a specific order.

6: Queue: Merging two queues can be done by merging them in a specific order.

7: Tree: Merging two trees can be done by merging the elements of one tree into the other, this can be done using various algorithms such as in-order, pre-order, and post-order traversals.

8: Hash table: Merging two hash tables can be done by iterating through the elements of one table and adding them to the other table.

Time complexity

1: Time complexity is a measure of the amount of time an algorithm takes to run as a function of the size of its input.

2: In data structures, it is used to analyze the performance of different operations, such as insertion, deletion, and search.

3: Common time complexities include O(1) for constant time, O(log n) for logarithmic time, and O(n) for linear time. O(n^2) for quadratic time.

4: O(1) is considered the most efficient time complexity, as it does not depend on the size of the input.

5: O(log n) is considered a good time complexity, as it grows slowly with the size of the input.

6: O(n) is considered a fair time complexity, as it grows linearly with the size of the input.

7: O(n^2) is considered a poor time complexity, as it grows quickly with the size of the input, and can quickly become impractical for large inputs.

Space complexity

1: Space complexity is a measure of the amount of memory an algorithm uses as a function of the size of its input.

2: It is similar to time complexity in that it is used to analyze the performance of an algorithm and compare it to other algorithms.

3: Space complexity can be calculated for both the best-case and worst-case scenarios of an algorithm.

4: A lower space complexity indicates a more memory efficient algorithm or data structure.

5: Space complexity can be reduced by using data structure which uses less memory such as using a hash table instead of an array or by using techniques such as memory pooling or memory reusing.

6: The most common space complexities include O(1) for constant space, O(n) for linear space, and O(n^2) for quadratic space.

O notation

1: Big O notation is commonly used to describe the time complexity and space complexity of an algorithm.

2: When used to describe time complexity, it tells us how the running time of an algorithm increases as the size of the input increases.

3: When used to describe space complexity, it tells us how the memory usage of an algorithm or data structure increases as the size of the input increases.

The most common O notations include:

O(1) for constant time or constant space. This means that the running time or memory usage does not increase as the size of the input increases.

O(log n) for logarithmic time. This means that the running time increases logarithmically as the size of the input increases.

O(n) for linear time. This means that the running time increases linearly as the size of the input increases.

O(n log n) for log-linear time. This means that the running time is a combination of linear and logarithmic time, it increases as the input size increases and logarithmically with the size of the input.

O(n^2) for quadratic time. This means that the running time increases as the square of the size of the input increases.

O(2^n) for exponential time. This means that the running time increases exponentially as the size of the input increases.

Ω Notation

1: Big Ω notation is used to describe the best-case time complexity of an algorithm, and it can also be used to describe the space complexity.

2: When used to describe time complexity, it tells us how the best-case running time of an algorithm decreases as the size of the input increases.

3: When used to describe space complexity, it tells us how the best-case memory usage of an algorithm or data structure decreases as the size of the input increases.

The most common Ω notations include:

Ω(1) for constant time or constant space. This means that the best-case running time or memory usage does not decrease as the size of the input increases.

Ω(log n) for logarithmic time. This means that the best-case running time decreases logarithmically as the size of the input increases.

Ω(n) for linear time. This means that the best-case running time decreases linearly as the size of the input increases.

Ω(n log n) for log-linear time. This means that the best-case running time is a combination of linear and logarithmic time, it decreases as the input size increases and logarithmically with the size of the input.

Ω(n^2) for quadratic time. This means that the best-case running time decreases as the square of the size of the input increases.

Ω(2^n) for exponential time. This means that the best-case running time decreases exponentially as the size of the input increases.

θ Notation

1: Θ notation is used to describe both the time complexity and space complexity of an algorithm or data structure.

2: In terms of time complexity, Θ notation can be used to describe the average-case running time of an algorithm as a function of the size of the input.

3: In terms of space complexity, Θ notation can be used to describe the average-case memory usage of an algorithm or data structure as a function of the size of the input**.**

4: the big Θ notation is based on the assumption that all inputs are equally likely, so it may not always give an accurate representation of the average-case performance of an algorithm or data structure.

5: The notation is usually expressed in big O notation and is used to give a more accurate representation of the performance of an algorithm or data structure.

6: For example, if an algorithm has a best-case time complexity of O(1) and a worst-case time complexity of O(n^2), the average-case time complexity would be Θ(n^2).

Unit 2 (Sorting and Searching)

Selection sort

1: Selection sort is a simple sorting algorithm that sorts an array or a list by repeatedly selecting the minimum element from the unsorted part of the data and swapping it with the first element of the unsorted part.
2: The process is repeated until the entire data is sorted.
3: It has a time complexity of O(n^2) in the worst-case scenario, making it inefficient for large datasets.
4: However, it has the advantage of being easy to understand and implement.
Note to me: Selection sort is a sorting algorithm that arranges numerical values in ascending order, i.e., from smallest to largest.selection sort can be used for sorting non-numeric values as well, such as strings or characters. The algorithm works by comparing elements based on their ordering.For example, in the case of strings, selection sort can be used to sort them in lexicographical order, i.e., in dictionary order.

Insertion sort

1: Insertion sort is a simple sorting algorithm that works by repeatedly dividing the unsorted part of an array into smaller parts and inserting the elements into their correct position in the sorted part of the array.

2: This algorithm is one of the simplest algorithm with simple implementation

Basically, Insertion sort is efficient for small data values.

3: Insertion sort is adaptive in nature, i.e. it is appropriate for data sets which are already partially sorted.

• Consider an example: arr[]: {12, 11, 13, 5, 6}

12 11 13 5 6

First Pass:

Initially, the first two elements of the array are compared in insertion sort.

12 11 13 5 6

Here, 12 is greater than 11 hence they are not in the ascending order and 12 is not at its correct position. Thus, swap 11 and 12.

So, for now 11 is stored in a sorted sub-array.

11 12 13 5 6

Second Pass:

Now, move to the next two elements and compare them

11 **12 13** 5 6

Here, 13 is greater than 12, thus both elements seems to be in ascending order, hence, no swapping will occur. 12 also stored in a sorted sub-array along with 11

Third Pass:

Now, two elements are present in the sorted sub-array which are 11 and 12

Moving forward to the next two elements which are 13 and 5

11 12 **13 5** 6

Both 5 and 13 are not present at their correct place so swap them

11 12 **5 13** 6

After swapping, elements 12 and 5 are not sorted, thus swap again

11 **5 12** 13 6

Here, again 11 and 5 are not sorted, hence swap again

5 11 12 13 6

Here, 5 is at its correct position

Fourth Pass:

Now, the elements which are present in the sorted sub-array are 5, 11 and 12

Moving to the next two elements 13 and 6

5 11 12 **13 6**

Clearly, they are not sorted, thus perform swap between both

5 11 12 **6 13**

Now, 6 is smaller than 12, hence, swap again

5 11 **6 12** 13

Here, also swapping makes 11 and 6 unsorted hence, swap again

5 6 11 12 13

Finally, the array is completely sorted.

Difference between insertion and selection sort

1: Selection sort and insertion sort are simple sorting algorithms with quadratic time complexity O(n^2) in the worst-case scenario.However, insertion sort may perform better than selection sort for small arrays or arrays that are partially sorted.

2: Selection sort repeatedly selects the minimum element and swaps it with the first element of the unsorted part, while insertion sort repeatedly divides the unsorted part and inserts elements into their correct position in the sorted part. **3:** Selection sort requires a constant amount of memory, but is not a stable sorting algorithm. Insertion sort requires only a constant amount of memory and is a stable sorting algorithm, preserving the relative order of equal elements in the sorted array.

Bubble sort

1: Bubble sort is a simple sorting algorithm that works by repeatedly swapping adjacent elements if they are in the wrong order.

2: The time complexity of bubble sort is O(n^2) in the worst-case scenario.

3: Thus, this algorithm is not suitable for large data sets.

•Consider an example.

Input: arr[] = {5, 1, 4, 2, 8}

First Pass:

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

(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.

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

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

(1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:

Now, during second iteration it should look like this:

(1 4 2 5 8) –> (1 4 2 5 8)

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

(1 2 4 5 8) –> (1 2 4 5 8)

(1 2 4 5 8) –> (1 2 4 5 8)

Third Pass:

Now, the array is already sorted, but our algorithm does not know if it is completed.

The algorithm needs one whole pass without any swap to know it is sorted.

(1 2 4 5 8) –> (1 2 4 5 8)

(1 2 4 5 8) –> (1 2 4 5 8)

(1 2 4 5 8) –> (1 2 4 5 8)

(1 2 4 5 8) –> (1 2 4 5 8)

Merge sort

1: Merge sort is a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.

O(n log n)

• Consider an example

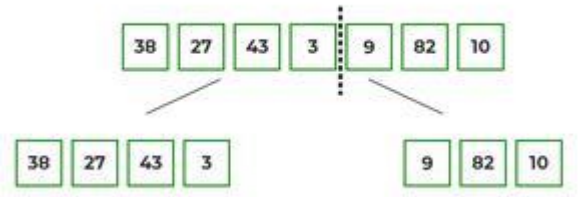
an array arr[] = {38, 27, 43, 3, 9, 82, 10}

At first, check if the left index of array is less than the right index, if yes then calculate its mid point.

Now, as we already know that merge sort first divides the whole array iteratively into equal halves, unless the atomic values are achieved.

Here, we see that an array of 7 items is divided into two arrays of size 4 and 3 respectively.

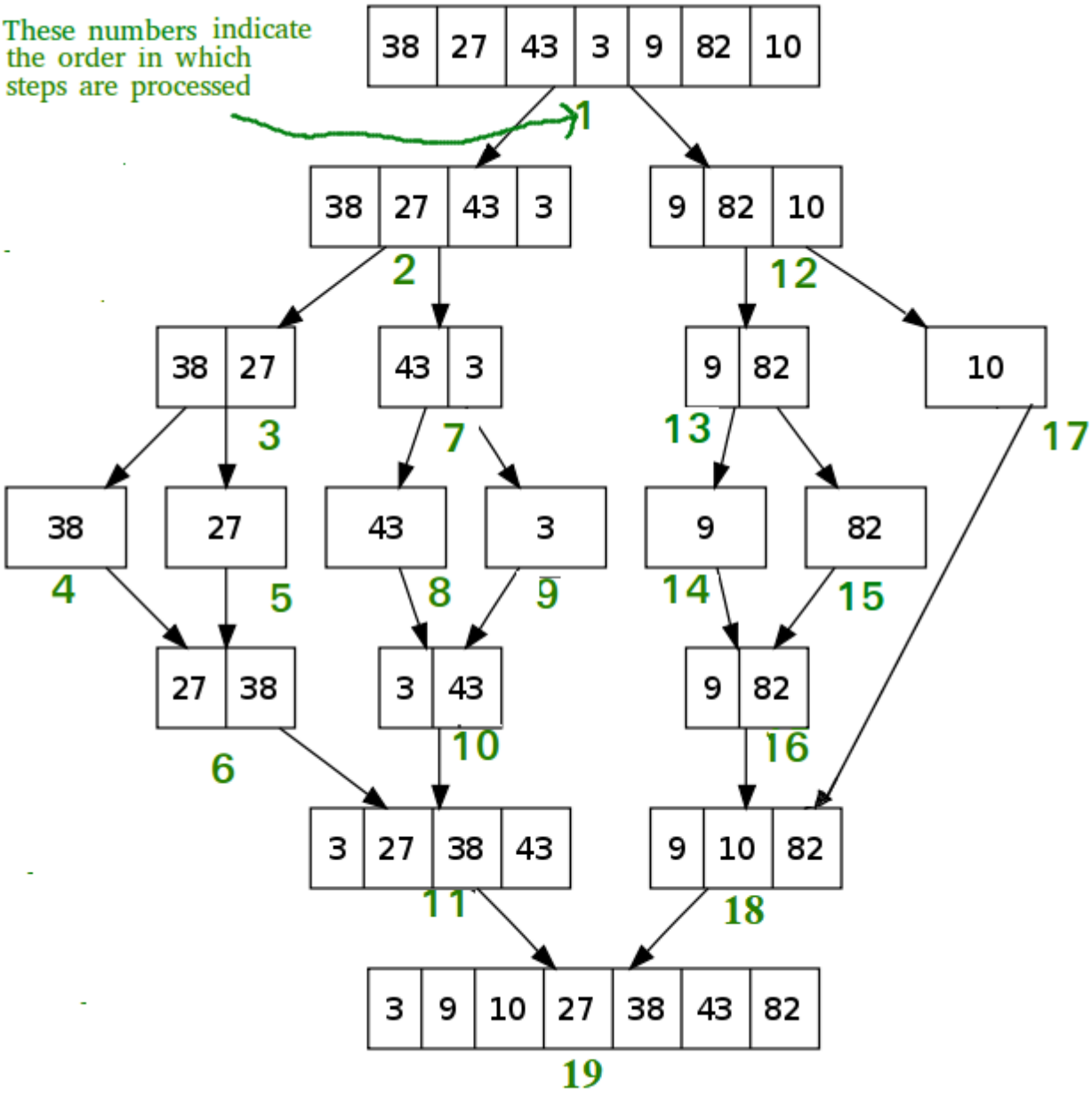
Now, again find that is left index is less than the right index for both arrays, if found yes, then again calculate mid points for both the arrays.



Now, further divide these two arrays into further halves, until the atomic units of the array is reached and further division is not possible.

After dividing the array into smallest units, start merging the elements again based on comparison of size of elements

Firstly, compare the element for each list and then combine them into another list in a sorted manner.



Radix Sort

1: Radix sort is a a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array

2: It can be used to sort integers, floating-point numbers, and other data types that can be represented as a sequence of digits.

Radix sort is the linear sorting algorithm that is used for integers. In Radix sort, there is digit by digit sorting is performed that is started from the least significant digit to the most significant digit.

time complexity of O(nd), where n is the size of the array and d is the number of digits in the largest number.

• consider an example

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

Sorting by least significant digit (1s place) gives: 170, 90, 802, 2, 24, 45, 75, 66.

[*Notice that we keep 802 before 2, because 802 occurred before 2 in the original list, and similarly for pairs 170 & 90 and 45 & 75.]

Sorting by next digit (10s place) gives: 802, 2, 24, 45, 66, 170, 75, 90.

Sorting by the most significant digit (100s place) gives: 2, 24, 45, 66, 75, 90, 170, 802.

[*Notice that 802 again comes before 2 as 802 comes before 2 in the previous list.]

Shell sort

1: In insertion sort, at a time, elements can be moved ahead by one position only.

2: To move an element to a far-away

position, many movements are required that increase the algorithm's execution time.

3: But shell sort overcomes this drawback of insertion sort.

4: It allows the movement and swapping of far-away elements as well.

5: This algorithm first sorts the elements that are far away from each other, then it subsequently reduces the gap between them.

6: This gap is called as **interval.** This interval can be calculated by using the **Knuth's** formula given below -

hh = h * 3 + 1

where, 'h' is the interval having initial value 1.

• working of shell sort algorithm:

https://www.javatpoint.com/shell-sort

Quick Sort algorithm

1: QuickSort is a Divide and Conquer algorithm.

2: It picks an element as a pivot and partitions the given array around the picked pivot.

3: The key process in quickSort is a partition().

4: The target of partitions is, given an array and an element x of an array as the pivot, put x at its correct position in a sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

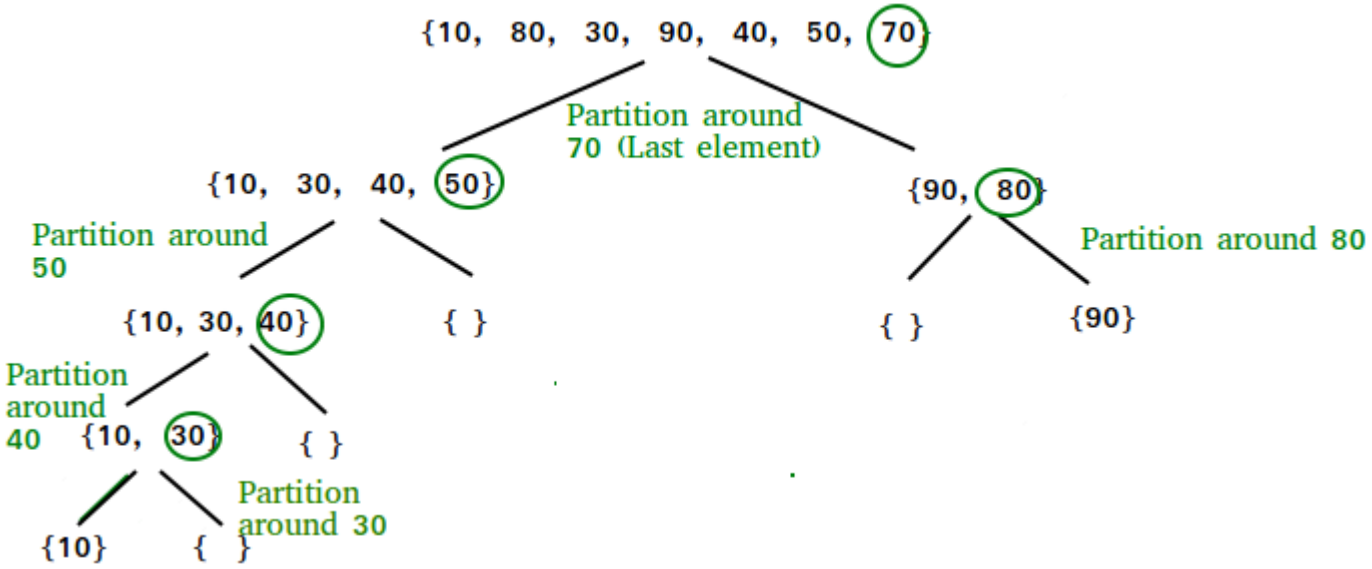
5: There are many different versions of quickSort that pick pivot in different ways.

a: Always pick the first element as a pivot.

b: Always pick the last element as a pivot.

c: Pick a random element as a pivot.

d: Pick median as the pivot.



Linear search

1: Linear search is the simplest method for searching.

2: In Linear search technique of searching; the element to be found in searching the elements to be found is searched sequentially in the list.

3: This method can be performed on a sorted or an unsorted list (usually arrays).

4: In case of a sorted list searching starts from 0th element and continues until the element is found from the list or the element whose value is greater than (assuming the list is sorted in ascending order), the value being searched is reached.

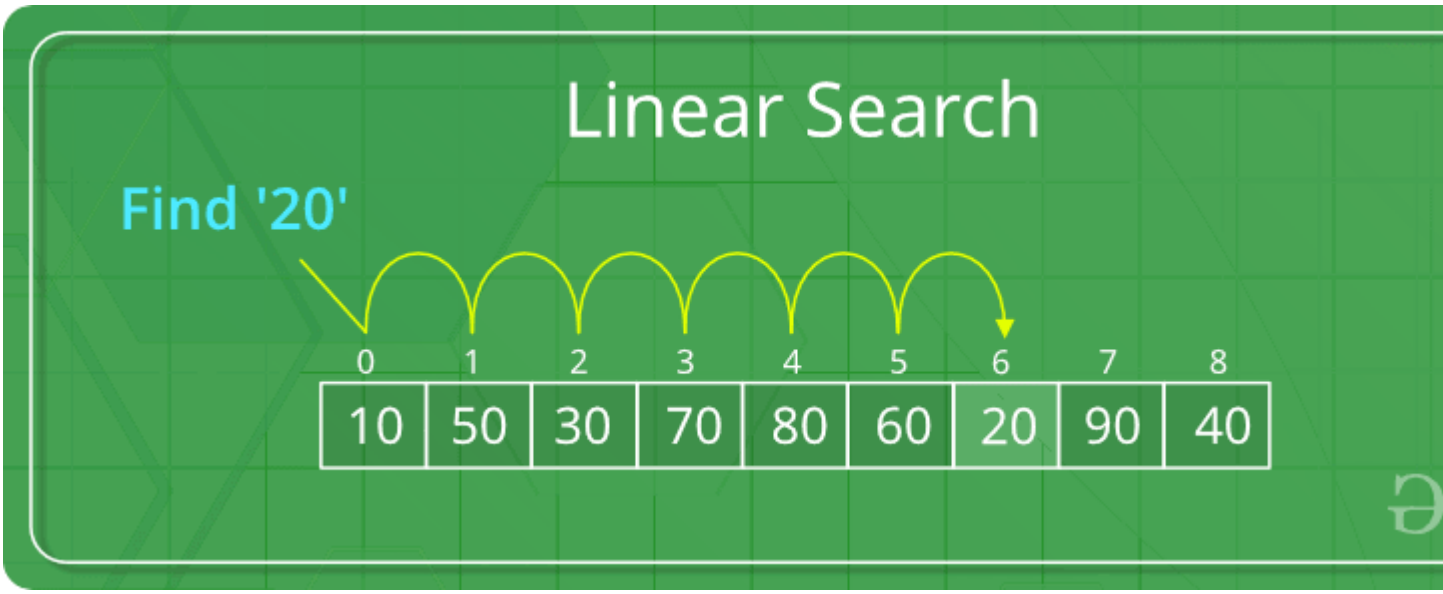
5: As against this, searching in case of unsorted list also begins from the 0th element and continues until the element or the end of the list is reached.

6: The linear search algorithm searches all elements in the array sequentially.

7: Its best execution time is 1, whereas the worst execution time is n, where n is the total number of items in the search array.

8: It is the most simple search algorithm in data structure and checks each item in the set of elements until it matches the search element until the end of data collection.

9: When data is unsorted, a linear search algorithm is preferred.



Binary search

1: Binary Search is a searching algorithm used in a sorted array by repeatedly dividing the search interval in half.

2: The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(Log n).

3:The basic steps to perform Binary Search are:

1: Sort the array in ascending order.

2: Set the low index to the first element of the array and the high index to the last element.

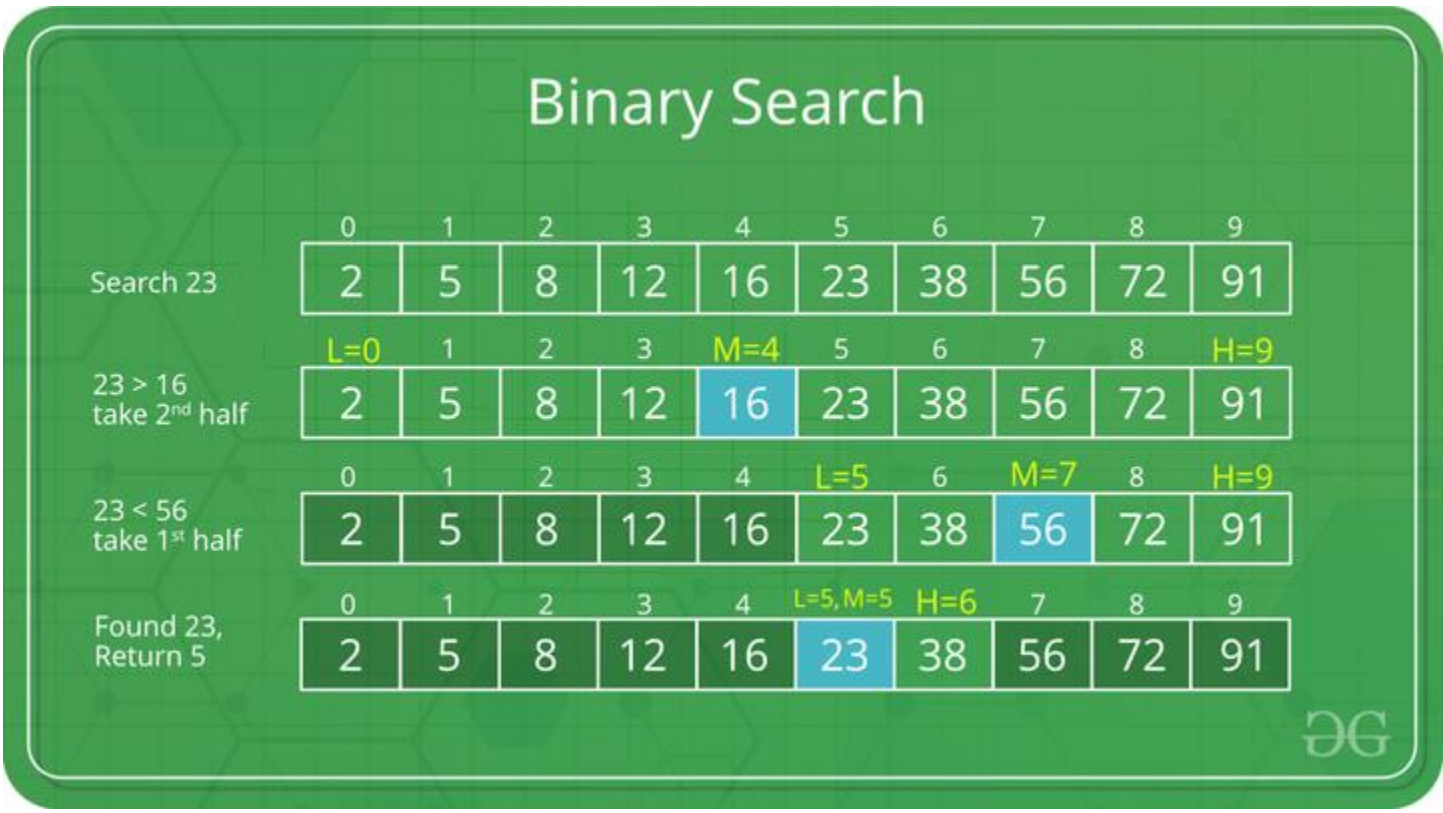
3: Set the middle index to the average of the low and high indices.

4: If the element at the middle index is ex the target element, return the middle index.

5: If the target element is less than the element at the middle index, set the high index to the middle index – 1.

6: If the target element is greater than the element at the middle index, set the low index to the middle index + 1.

7: Repeat steps 3-6 until the element is found or it is clear that the element is not present in the array.


Binary Search
Search 23
0 1 2 3 4 5 6 7 8 9
2 5 8 12 16 23 38 56 72 91
23 > 16
take 2nd half
L=0 1 2 3 M=4 5 6 7 8 H=9
2 5 8 12 16 23 38 56 72 91
23 < 56
take 1st half
0 1 2 3 4 L=5 6 M=7 8 H=9
2 5 8 12 16 23 38 56 72 91
Found 23,
Return 5
0 1 2 3 4 L=5, M=5 H=6 7 8 9
2 5 8 12 16 23 38 56 72 91

Unit 3 (Stacks)

Stack as an abstract data type

1: a stack is an abstract data type that represents a collection of elements arranged in a specific order.
2: It is based on the last-in, first-out (LIFO) principle, which means that the last element added to the stack is the first one to be removed.
3: A stack has two main operations: push and pop.
4: The push operation adds an element to the top of the stack, while the pop operation removes the top element from the stack.
5: In addition to push and pop, a stack may also provide other operations such as peek, which returns the top element without removing it, and isEmpty, which returns a Boolean value indicating whether the stack is empty or not.

Representation of stack through arrays

1: *Fixed-size array implementation:* In this method, a fixed-size array is used to represent the stack. The array has a fixed capacity, and elements are pushed onto and popped off the stack by modifying the array indices. The disadvantage of this method is that if the number of elements in the stack exceeds the capacity of the array, a stack overflow error occurs.

2: *Dynamic array implementation*: In this method, a dynamic array is used to represent the stack. The array can grow or shrink in size as elements are pushed onto or popped off the stack. The advantage of this method is that it can handle stacks of any size, but it may be less efficient than the fixed-size array implementation due to the overhead of resizing the array.

3: *Linked list implementation:* In this method, a linked list is used to represent the stack. Each element of the stack is represented by a node in the linked list, and the top of the stack is represented by the head of the list. Elements can be pushed onto or popped off the stack by adding or removing nodes from the head of the list. The advantage of this method is that it can handle stacks of any size and is generally more efficient than the array implementations.

Reversing a Linked list using stack

Algorithm:

Step 1: Create an empty stack.

Iterate through the list and push each element onto the stack.

Step 2: Create a new empty list.

Step 3: Iterate through the stack and pop each element, appending it to the new list.

Step 4: Return the new list.

Polish notations

1: The way to write arithmetic expression is known as a notation.

2: An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression.

3: These notations are −

Infix notation: We write expression in infix notation, e.g. a - b + c, where operators are used in-between operands. It is easy for us humans to read, write, and speak in infix notation but the same does not go well with computing devices. An algorithm to process infix notation could be difficult and costly in terms of time and space consumption.

Prefix notation: In this notation, operator is prefixed to operands, i.e. operator is written ahead of operands. For example, +ab. This is equivalent to its infix notation a + b. Prefix notation is also known as Polish Notation.

Postfix notation: This notation style is known as Reversed Polish Notation. In this notation style, the operator is postfixed to the operands i.e., the operator is written after the operands. For example, ab+. This is equivalent to its infix notation a + b.

The following table briefly tries to show the difference in all three notations −

Sr no.	Infix	Prefix	postfix
1	(a + b) ∗ c	∗ + a b c	a b + c ∗
2	a / b + c / d	+ / a b / c d	a b / c d / +
3	(a + b) ∗ c) - d	- ∗ + a b c d	a b + c ∗ d -

Unit 4 (Queues)

queue as an abstract data type

1: A queue is an abstract data type that represents a collection of elements in which elements are added at one end, called the "rear" or and removed from the other end, called the "front".
2: In other words, a queue follows the "First-In-First-Out" (FIFO) principle, where the first element added is the first one to be removed.
3: The two main operations that can be performed on a queue are adding an element to the rear, which is called "enqueue", and removing an element from the front, which is called "dequeue".
4: Additionally, there are some other operations that are commonly supported by queues, such as checking if the queue is empty, getting the number of elements in the queue, and accessing the front element without removing it.

Representation of a Queue as an array

1: A queue can be implemented as an array by keeping track of the front and rear of the queue using two indices.

2: To implement a queue using an array,

create an array arr of size n and

take two variables front and rear both of which will be initialized to 0 which means the queue is currently empty.

3: rear is the index up to which the elements are stored in the array and

front is the index of the first element of the array.

---

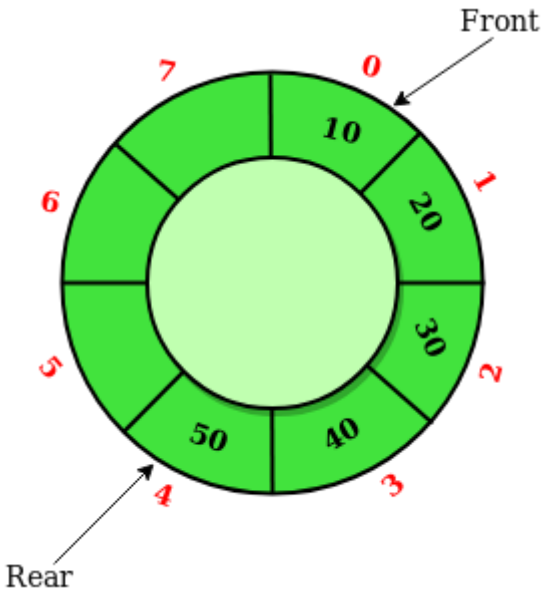
circular queue

1: A circular queue is a variation of a queue data structure in which the front and rear of the queue are connected, forming a circle.

2: This means that the last element in the queue points back to the first element in the queue, creating a circular structure.

3: In a circular queue, when the rear reaches the end of the array, it wraps around to the beginning of the array, and when the front reaches the end of the array, it also wraps around to the beginning.

4: This allows the queue to efficiently use the available space in the array without wasting any space.



Double ended queue

1: A Double Ended Queue (Deque) is an abstract data type that represents a sequence of elements, allowing elements to be added or removed from both ends.

2: It is sometimes also called a "deque" (pronounced "deck"), which is a shorthand for "double-ended queue".

3: A deque allows elements to be added or removed from the front or the back of the queue in constant time, making it very efficient for certain operations.

4: Deques are useful in a variety of scenarios, including as a basic building block for other data structures such as stacks, queues, and lists.

Priority Queue

1: A Priority Queue is an abstract data type that allows elements to be added and removed based on a priority value.

2: Each element is assigned a priority value, and the elements with higher priority are removed first.

3: In other words, a priority queue is a data structure that maintains a set of elements, each with an associated priority, and supports adding elements and removing the element with the highest priority.

4: Priority Queues are commonly used in algorithms and applications that involve scheduling, optimization, or resource allocation.

Dequeue

1: A Deque (double-ended queue) is a data structure that allows elements to be inserted and removed from both ends of the queue.

2: It is similar to a queue and a stack, but unlike a queue, a Deque allows inserting and removing elements from both ends, while a stack only allows inserting and removing elements from one end.

3: Deques are useful when you need to implement certain algorithms that require adding and removing elements from both ends, such as graph traversal, where you may need to explore a graph in both directions, or when you need to maintain a sliding window over a sequence of elements.

Applications of Queue

1: Job scheduling: Queues are often used to schedule tasks or jobs that need to be processed in a specific order. For example, a printer may use a queue to process print jobs in the order they were received.

2: Message queuing: In messaging systems, messages are added to a queue and are processed in the order they are received. This is useful in scenarios where messages need to be processed sequentially, for example, in a chat application.

3: Event-driven programming: In event-driven programming, events are added to a queue and are processed in the order they are received. This is useful for handling asynchronous events, such as user input in a graphical user interface.

Unit 5 (Linked List)

Linked list and terminology

A linked list is a data structure that consists of a sequence of elements, where each element contains a reference or a pointer to the next element in the sequence. The first element in the sequence is called the head of the list, and the last element is called the tail.. Here are some common terminology used when working with linked lists:

1: Node: A node is an individual element in a linked list that contains the data to be stored and a reference to the next node in the list.

2: Head: The head is the first node in the linked list. It is used as the starting point for traversing the list.

3: Tail: The tail is the last node in the linked list. It points to null or an empty reference, indicating the end of the list.

4: Next: Each node in a linked list contains a reference to the next node in the sequence. This reference is commonly called the "next" pointer.

5: Adress: The memory address of an element in the linked list is often referred to as its "location" or "pointer."

6: Null pointer: A null pointer is a special value that is used to represent a pointer or reference that does not refer to a valid memory address.

7: Empty list: An empty list is a linked list that contains no elements. In other words, it is a data structure that has been initialized but contains no data.

8: Information: A linked list is a dynamic data structure that stores a sequence of elements called nodes, in which each node contains two fields: a data field and a pointer that points to the next node in the list.The data field can store any type of information, such as an integer, a character, a string, or a complex object.

Linear list

1: A linear list, is a collection of elements that are stored in a linear order.

2: In a linear list, each element has a unique position or index that identifies its location in the sequence.

3: Linear lists are commonly used in computer programming and data structures, as they provide a simple and efficient way to organize and manipulate data.

4: The main advantage of a linear list is its efficiency in accessing elements, as each element can be accessed in constant time using its index. Q

Circular linked list

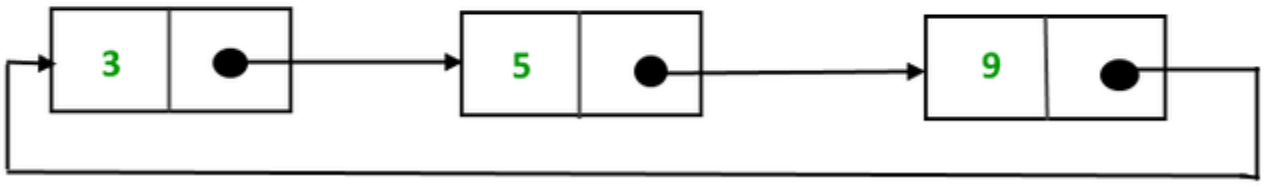
1: The circular linked list is a linked list where all nodes are connected to form a circle.

2: In a circular linked list, the first node and the last node are connected to each other which forms a circle.

3: There is no NULL at the end.

4: We traverse the circular singly linked list until we reach the same node where we started.

5: The circular singly linked list has no beginning or end.

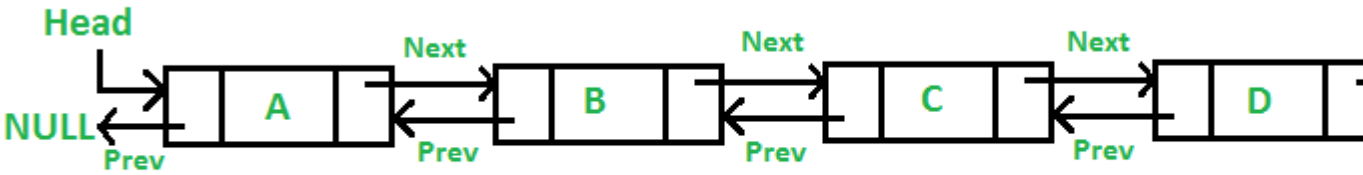


Doubly linked list

1: A doubly linked list is a data structure consisting of a group of nodes, where each node contains three fields: a data element, a pointer to the next node in the sequence, and a pointer to the previous node in the sequence.

2: Unlike a singly linked list, which only has a reference to the next node in the sequence, a doubly linked list allows traversal in both forward and backward directions.

3: double linked list require more memory overhead than singly linked lists because of the additional pointer field for each node.



Traversing

- **STEP 1:** SET PTR = HEAD
- **STEP 2:** IF PTR = NULL

  WRITE "EMPTY LIST"
  GOTO STEP 7
  END OF IF

- **STEP 4:** REPEAT STEP 5 AND 6 UNTIL PTR != NULL
- **STEP 5:** PRINT PTR→ DATA
- **STEP 6:** PTR = PTR → NEXT

  [END OF LOOP]
  STEP 7: EXIT

Unit 6 (Trees)

Terminology of tree

1: Node: The fundamental unit of a tree, which stores data and may have zero or more child nodes.

2: Root: The topmost node of a tree, which has no parent.

3: Parent: A node that has one or more child nodes.

4: Child: A node that has a parent node.

5: Siblings: Nodes that share the same parent node.

6: Leaf: A node that has no child nodes, also known as an external node.

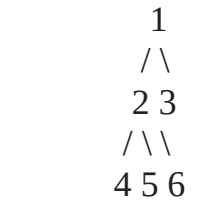
7: Internal node: A node that has one or more child nodes.

8: Depth: The number of edges on the path from the root to a node.

9: Height: The number of edges on the longest path from a node to a leaf.

10: Subtree: A tree rooted at a child node of a parent node.

11: Ancestor and descendent nodes: an ancestor node is a node that is closer to the root than the node in question, while a descendant node is a node that is further away from the root than the node in question.



Here 1 is an ancestor and others are it descendents.

12: path: a path is a sequence of nodes that starts from a given node and ends at a descendant node or at a leaf node (i.e., a node with no children).For example, the length of the path from node 5 to node 10 in the below tree is 3, because there are three edges in the path {5, 8, 9, 10}.

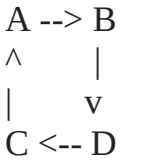
1

13: Directed edge: directed edge is an ordered pair of vertices that represents a one-way connection between two vertices. It is also sometimes referred to as a directed arc or simply an *arc.In* a directed edge, the first vertex is called the "tail" of the edge, and the second vertex is called the "head" of the edge. The direction of the edge is from the tail to the head.

there are six directed edges: (A, B), (A, C), (B, C), (B, D), (C, D), and (D, A).

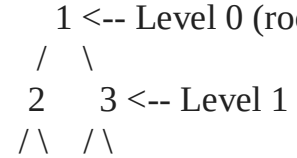
14: out degree: out-degree of a node is the number of children that the node has.

15: in degree: in-degree of a vertex is the number of edges that terminate at the vertex. In other words, the in-degree of a vertex is the number of directed edges that point to the vertex.



The in-degree of vertex A is 1 (there is only one edge that points to A), the in-degree of vertex B is 1 (one edge points to B), the in-degree of vertex C is 1, and the in-degree of vertex D is 2 (two edges point to D).

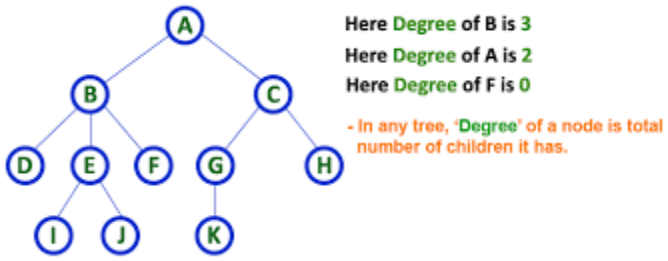
16: level of a node: the level of a node refers to its distance from the root node. The root node is typically defined to be at level 0, and each subsequent level is incremented by 1 as you move down the tree.



4 5 6 7 <-- Level 2

17: degree of a tree: The degree of a tree is the maximum number of children any node in the tree has. In other words, it is the maximum number of edges that are connected to a single node.

18: degree of a node: The degree of a node in a tree is the number of immediate children it has. In other words, it is the number of edges that are connected to the node.



Types of trees

1: Binary tree: a tree in which each node has maximum two children.

2: Binary search tree: a binary tree in which the left child of a node has a key less than its parent node, and the right child of a node has a key greater than its parent node.

3: AVL tree: a self-balancing binary search tree that maintains a balance factor to keep the height of the tree low.

4: Red-black tree: another self-balancing binary search tree that maintains a balance factor using color-coding to keep the height of the tree low.

5: B-tree: a self-balancing tree that is optimized for disk access and can store large amounts of data on a single node.

6: Trie tree: a tree used for efficient information retrieval and prefix matching, commonly used in text search engines.

7: Heap: a tree-like data structure that satisfies the heap property, commonly used in heap sort and priority queue algorithms.

8: Radix tree: a tree used for storing keys in a compact way and optimizing search time.

General tree

1: In a general tree, each node can have an arbitrary number of child nodes, unlike a binary tree where each node can have at most two child nodes.

2: A general tree is typically represented using a linked data structure where each node contains a data element and a list of references to its child nodes.

3: The first node in the tree is called the root node, and it has no parent nodes.

4: Each non-root node has exactly one parent node.

5: General trees are used to represent hierarchical structures such as

family trees, file systems, organization charts, and more.

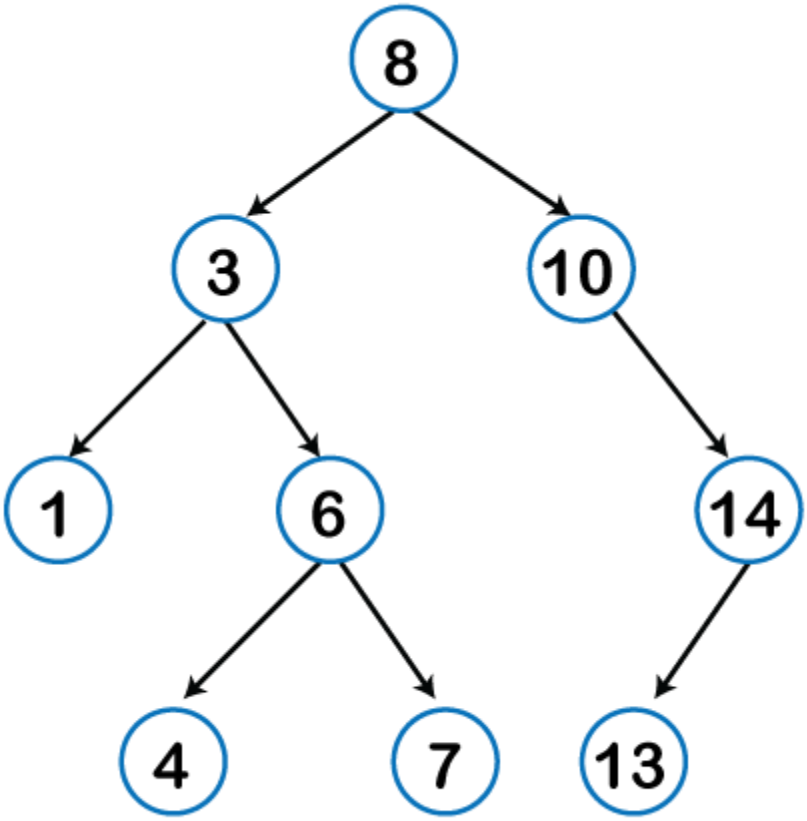
Binary tree

1: A binary tree is a tree data structure in which each node can have at most 2 children.

2: Binary trees are commonly used to represent hierarchical relationships, and they are often used in computer science to represent the structure of a file system or the organization of data in a database.

3: Binary trees can have zero, one, or two children, and they can be categorized as leaf, unary, or binary nodes.

4: take points from general tree.



Binary search tree (BST)

1: A binary search tree (BST) is a type of binary tree, a data structure that is used to store collections of elements.

2: In a BST, each node has at most two children, referred to as the left child and the right child.

3: The left child is always less than the parent, and the right child is always greater than the parent.

4: This property is known as the "BST property."

5: To search an element in a BST, compare it to the root, and then search the left or right subtree recursively based on the element's value.

6: To insert an element, find its correct position based on the BST property, and create a new node there.

7: Deleting an element is complex, as it involves removing a node while maintaining the BST property.

Expression Tree

1: An expression tree is a binary tree where each leaf node contains an operand, and each non-leaf node contains an operator.

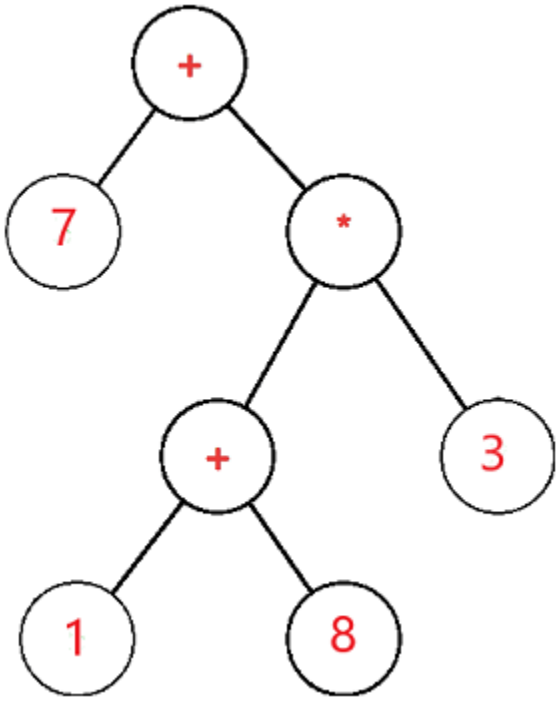
2: The tree represents an arithmetic expression, where the leaves represent the operands, and the non-leaf nodes represent the operators.

3: The operator present in the depth of the tree is always at the highest priority.

4: The operator, which is not much at the depth in the tree, is always at the lowest priority compared to the operators lying at the depth.

5: To evaluate an expression tree, we start at the root node and recursively evaluate the left and right subtrees, using the operator at the root to combine their results.

*expression tree for 7 + ((1+8)*3) would be:*



Notes from teacher:

K= 1, K is index of the root,

[2×K] to find left child

[(2×k)+1] to find right child

Right child is even node.

Left child is odd node.

K-1÷2 to find parent of left child

K÷2 to find parent of right child

Binary tree algorithm:

in order Traversal

- **Step 1:** Repeat Steps 2 to 4 while TREE != NULL
- **Step 2:** INORDER(TREE -> LEFT)
- **Step 3:** Write TREE -> DATA
- **Step 4:** INORDER(TREE -> RIGHT)
  [END OF LOOP]
- **Step 5:** END

Pre order Traversal

Step 1: Repeat Steps 2 to 4 while TREE != NULL

Step 2: Write TREE -> DATA

Step 3: PREORDER(TREE -> LEFT)

Step 4: PREORDER(TREE -> RIGHT)

[END OF LOOP]

Step 5: END

Post order Traversal

Step 1: Repeat Steps 2 to 4 while TREE != NULL

Step 2: POSTORDER(TREE -> LEFT)

Step 3: POSTORDER(TREE -> RIGHT)

Step 4: Write TREE -> DATA

[END OF LOOP]

Step 5: END

Unit 7 (Graph)

Graph terminologies

1: Vertex (or Node): A point in a graph is called a vertex or a node. Each vertex may have a unique identifier or label.

2: Edge: A line or a connection between two vertices is called an edge. An edge can be directed or undirected, weighted or unweighted.

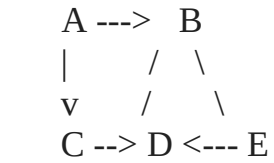
3: graph: graph is a non-linear data structure consisting of a collection of nodes or vertices and edges that connect them.A node or vertex represents a point in the graph, and an edge represents a connection between two nodes.

4: Directed Graph: A graph in which edges have a specific direction associated with them is called a directed graph.

5: Degree: The number of edges that are incident(endpoint) to a vertex is called its degree.

6: in degree: The in-degree of a node is the number of edges pointing towards the node.In-degree is denoted by the symbol 'indeg'.

7: out degree : The out-degree of a node is the number of edges pointing away from the node.Out-degree is denoted by the symbol 'outdeg.



The in-degree of node A is 0 (no incoming edges).
The out-degree of node A is 2 (outgoing edges to nodes B and C).
The in-degree of node B is 1 (incoming edge from node A).
The out-degree of node B is 2 (outgoing edges to nodes D and E).
The in-degree of node C is 1 (incoming edge from node A).
The out-degree of node C is 1 (outgoing edge to node D).
The in-degree of node D is 3 (incoming edges from nodes B, C, and E).
The out-degree of node D is 0 (no outgoing edges).
The in-degree of node E is 1 (incoming edge from node B).
The out-degree of node E is 1 (outgoing edge to node D).

8: Adjacency List: A collection of unordered lists, one for each vertex in the graph. The list contains all the vertices that are adjacent to the vertex represented by the list.

```
A -- B
|    |
C -- D
```

In an adjacency list representation, this graph would be represented as follows:

A: [B, C]
B: [A, D]
C: [A, D]
D: [B, C]

9: Successors: the successors of a vertex in a directed graph are the vertices that can be directly reached from the vertex by following an edge in the direction of the edge.

A --> B
A --> C
B --> D
C --> D

The successors of vertex A are vertices B and C, since there is an outgoing edge from A to each of those vertices. The successors of vertex B are vertex D, and the successors of vertex C are also vertex D. The successor set of vertex D is the empty set since it has no outgoing edges.

10: predecessor: the predecessors of a vertex are the vertices that can reach the given vertex by following an edge in the direction opposite to the edge.

A <-- B
A <-- C
B <-- D
C <-- D

The predecessors of vertex A are the vertices B and C, since there are incoming edges from vertices B and C to A. The predecessor set of vertex B is the vertex A, and the predecessor set of vertex C is also the vertex A. The predecessor set of vertex D is vertices B and C, since there are incoming edges from both vertices B and C to D.

11: Relation: relation can refer to any binary relation defined on the vertices of a graph, such that for any two vertices, the relation defines whether or not they are related or connected.

12: Path**:** A sequence of edges that connects a sequence of vertices is called a path.

13: Cycle**:** A path in which the first and last vertices are the same is called a cycle.

14: Connected Graph: A graph is said to be connected if there is a path from any vertex to any other vertex in the graph.

15: Disconnected Graph: A graph is said to be disconnected if there exist two vertices that are not connected by any path.

16: Weighted Graph**:** A graph in which edges have weights or values assigned to them is called a weighted graph.

17: Undirected Graph: A graph in which edges have no specific direction associated with them is called an undirected graph.

18: Length**:** the length of a path in a graph is defined as the number of edges in the path. The length of a path is used to measure the distance between two vertices in the graph.

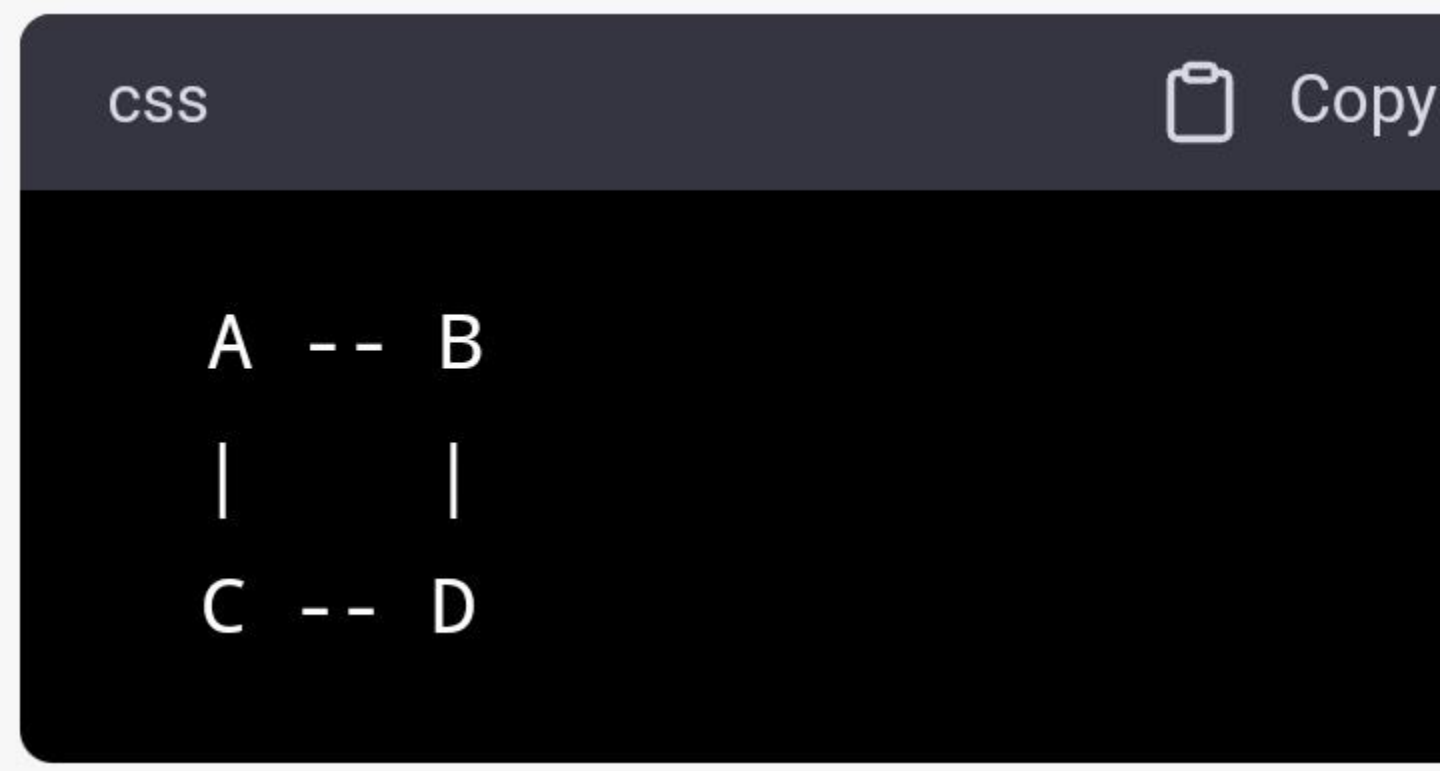
Representation of Array using graph

1: One common way to represent a graph using an arrayis by using an adjacency matrix.

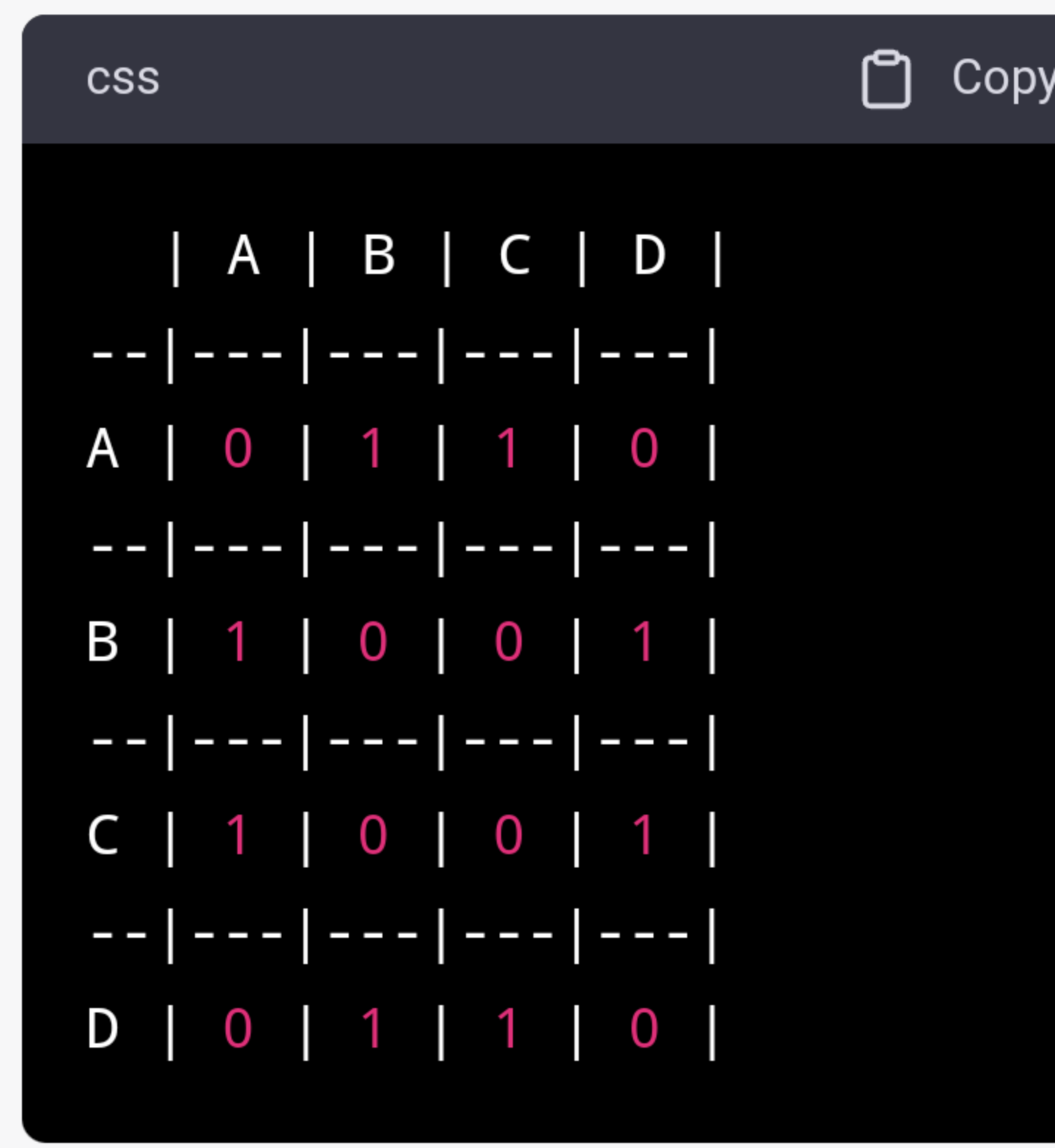
2: An adjacency matrix is a 2D array that has a row and a column for each vertex in the graph, and each element (i, j) of the matrix is 1 if there is an edge from vertex i to vertex j, and 0 otherwise.

3: If the graph is weighted, the matrix elements can represent the weight of the edge instead of just 1 or 0.

For example, consider the following undirected graph:



This graph can be represented using the following adjacency matrix:



In this matrix, the row and column for each vertex represent the adjacency relationships between that vertex and every other vertex. For example, the row for

vertex A indicates that there is an edge from A to B and an edge from A to C, but no edge from A to D.

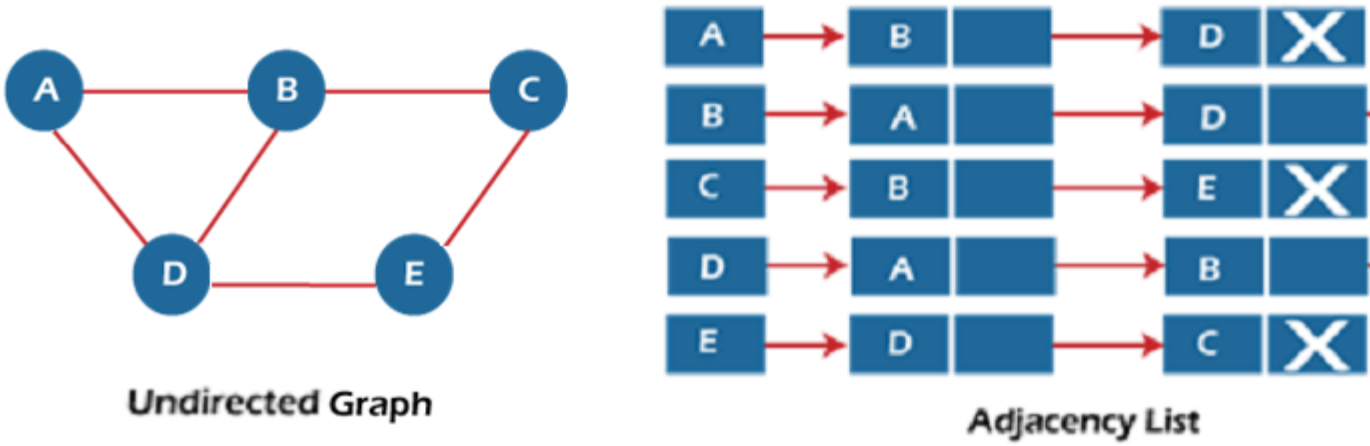
Representation of graph using Linked list

1: An adjacency list is used in the linked representation to store the Graph in the computer's memory.

2: An adjacency list is maintained for each node present in the graph, which stores the node value and a pointer to the next adjacent node to the respective node.

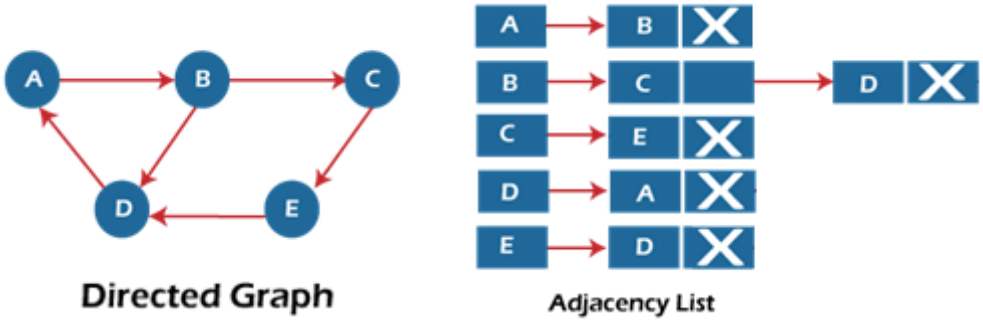
3: If all the adjacent nodes are traversed, then store the NULL in the pointer field of the last node of the list.

Representation of an undirected graph:



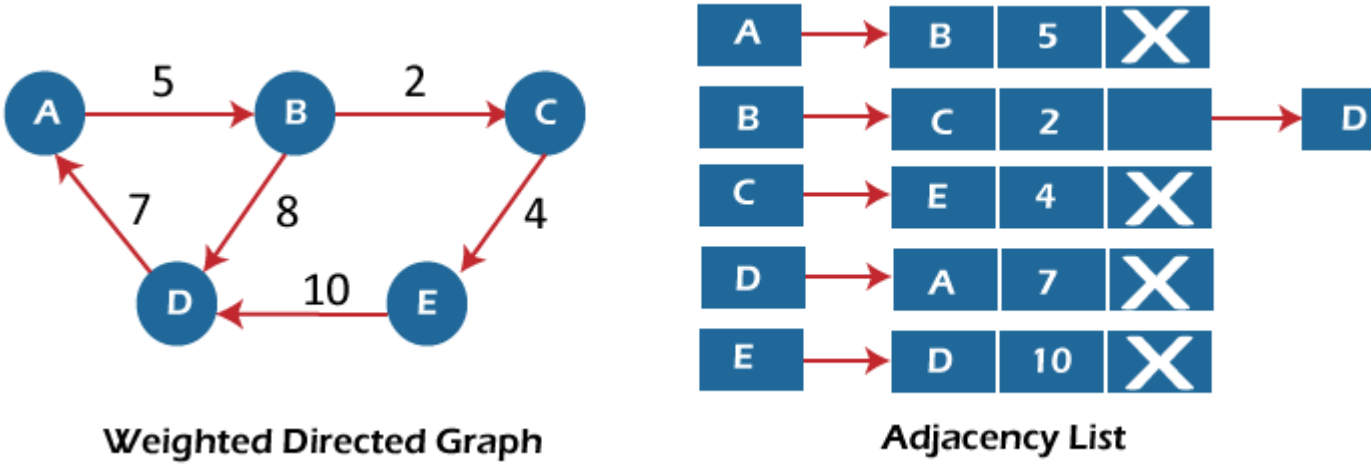
In the above figure, we can see that there is a linked list or adjacency list for every node of the graph. From vertex A, there are paths to vertex B and vertex D. These nodes are linked to nodes A in the given adjacency list.The sum of the lengths of adjacency lists is equal to twice the number of edges present in an undirected graph.

Representation of an directed graph:



For a directed graph, the sum of the lengths of adjacency lists is equal to the number of edges present in the graph.

Representation of an weighted graph:



In the case of a weighted directed graph, each node contains an extra field that is called the weight of the node.

Applications of graph

1: In Computer science graphs are used to represent the flow of computation.

2: Google maps uses graphs for building transportation systems, where intersection of two(or more) roads are considered to be a vertex and the road connecting two vertices

is considered to be an edge, thus their navigation system is based on the algorithm to calculate the shortest path between two vertices.

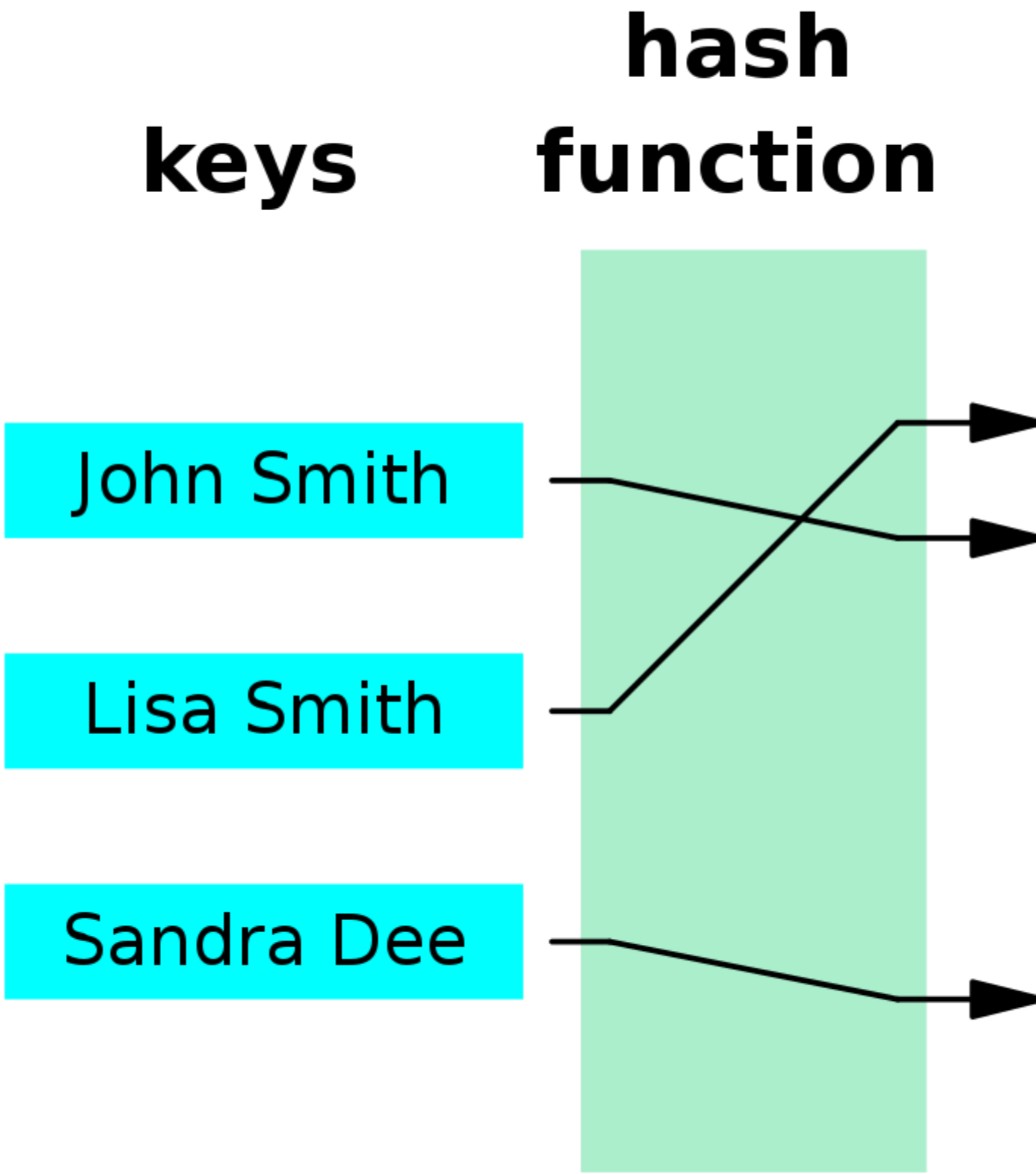
3: In Facebook, users are considered to be the vertices and if they are friends then there is an edge running between them. Facebook is an example of undirected graph.

4: On social media sites, we use graphs to track the data of the users. like showing preferred post suggestions, recommendations, etc.

Unit 8 (Hashing)

Hash function

1: hash function is a function that takes in an input (usually a string or other data type) and returns a fixed-size output, which is often a numeric value or a bit array.
2: The output value, known as the hash value, is generally much smaller than the input value, and it is computed in a deterministic way such that the same input always produces the same output.


keys
hash function
John Smith
Lisa Smith
Sandra Dee

Types of hash function

1: Division method: The hash function divides the value k by M and then uses the remainder obtained.

Formula:

$h(K) = k \bmod M$

Example:

$k = 12345$

$M = 95$

$h(12345) = 12345 \bmod 95$

$= 90$

$k = 1276$

$M = 11$

$h(1276) = 1276 \bmod 11$

$= 0$

2: Multiplication method:

This method involves the following steps:

1: Choose a constant value A such that $0 < A < 1$.

2: Multiply the key value with A.

3: Extract the fractional part of kA.

4: Multiply the result of the above step by the size of the hash table i.e. M.

5: The resulting hash value is obtained by taking the floor of the result obtained in step 4.

Formula:

$$h(K) = floor\ (M\ (kA \bmod 1))$$

Here,

M is the size of the hash table.

k is the key value.

A is a constant value.

Example:

$k = 12345$

A = 0.357840

M = 100

$$h(12345) = floor[\ 100\ (12345*0.357840 \bmod 1)]$$

$$= floor[\ 100\ (4417.5348 \bmod 1)\ ]$$

$$= floor[\ 100\ (0.5348)\ ]$$

$$= floor[\ 53.48\ ]$$

$$= 53$$

3: Folding method: This method involves two steps:

1: Divide the key-value **k** into a number of parts i.e. **k1, k2, k3,….,kn**, where each part has the same number of digits except for the last part that can have lesser digits than the other parts.

2: Add the individual parts. The hash value is obtained by ignoring the last carry if any.

Formula:

k = k1, k2, k3, k4, ..., kn

s = k1+ k2 + k3 + k4 +....+ kn

h(K)= s

Here,

s is obtained by adding the parts of the key k.

Example:

k = 12345

k1 = 12, k2 = 34, k3 = 5

s = k1 + k2 + k3

= 12 + 34 + 5

= 51

h(K) = 51

4: Mid-square method:

It involves two steps to compute the hash value-

1: Square the value of the key k i.e. k2.

2: Extract the middle r digits as the hash value.

Formula:

h(K) = h(k x k)

Here,

k is the key value.

Example:

Suppose the hash table has 100 memory locations. So r = 2 because two digits are required to map the key to the memory location.

k = 60

k x k = 60 x 60

= 3600

h(60) = 60

The hash value obtained is 60.

Collision in hashing

1: In hashing, a collision occurs when two or more distinct keys are mapped to the same index in the hash table.

2: This can happen due to the limited size of the hash table or due to the nature of the hash function used.

3: There are two types of collisions that can occur in hashing:

Direct collisions: Direct collisions occur when two different keys are hashed to the same index in the hash table. This can happen if the hash function produces the same hash value for different keys.

Indirect collisions: Indirect collisions occur when two different keys are hashed to different indexes, but due to the limited size of the hash table, they end up in the same bucket or slot. This is also known as a "collision chain."

Collision Resolution Techniques

Chaining: In this technique, a linked list is used to store multiple values at the same index. When a collision occurs, the new key-value pair is added to the linked list at that index.

Open addressing: In this technique, when a collision occurs, the algorithm searches for the next available slot in the hash table and inserts the key-value pair at that location.

Robin Hood hashing: This technique is similar to open addressing, but it tries to balance the hash table by moving items that are further from their ideal index towards their ideal index.

Note for me:

Sure, let's consider a simple example of a hash table with 5 slots and a hash function that calculates the remainder of the key when divided by 5. We will use chaining to resolve collisions.

Suppose we want to insert the following key-value pairs into the hash table:

"apple" -> 5

"banana" -> 7

"cherry" -> 6

"date" -> 9

"elderberry" -> 4

Using the hash function, we calculate the index for each key as follows:

"apple" -> index 0 (5 % 5 = 0)

"banana" -> index 2 (7 % 5 = 2)

"cherry" -> index 1 (6 % 5 = 1***)***

"date" -> index 4 (9 % 5 = 4)

"elderberry" -> index 4 (4 % 5 = 4)

As we can see, both "date" and "elderberry" have hashed to the same index (index 4), causing a collision.

To resolve this collision using chaining, we create a linked list at index 4 and add both "date" and "elderberry" to it, resulting in the following hash table***:***

Index 0: "apple" -> 5

Index 1: "cherry" -> 6

Index 2: "banana" -> 7

Index 3: empty

Index 4: "date" -> 9 -> "elderberry" -> 4

Now, when we want to retrieve the value associated with "date," we first calculate its index as 4 and then traverse the linked list at that index to find the correct key-value pair.
