**What is an ABSTRACT DATA TYPES?**

An abstract data type (ADT) is a high-level description of a data structure that specifies its behaviour without specifying the details of how it is implemented. In other words, an ADT defines a set of operations that can be performed on a data structure and the behaviour of those operations, but it does not specify the internal representation or implementation of the data structure.

ADTs are used to encapsulate and abstract the complexity of data structures, allowing programmers to work with data in a more intuitive and modular way. They provide a clear interface for interacting with the data while hiding the underlying details. This abstraction is crucial in software development because it promotes separation of concerns, making it easier to manage and maintain complex software systems.

Common examples of abstract data types include:

1. Stack: An ADT that represents a collection of elements with two main operations, "push" and "pop." It follows the Last-In-First-Out (LIFO) principle.

2. Queue: An ADT that represents a collection of elements with two primary operations, "enqueue" and "dequeue." It follows the First-In-First-Out (FIFO) principle.

3. List: An ADT that represents an ordered collection of elements with operations like "insert," "delete," and "retrieve."

4. Set: An ADT that represents a collection of distinct elements with operations like "add," "remove," and "check for membership."

5. Map (Dictionary): An ADT that represents a collection of key-value pairs with operations like "insert," "delete," and "retrieve" based on keys.

6. Graph: An ADT that represents a set of nodes and edges, often used for modelling relationships between objects.

7. Tree: An ADT used for hierarchical structures with a root node and child nodes.

ADTs are typically implemented in various programming languages using concrete data structures and algorithms. For example, a stack ADT can be implemented using an array or a linked list, but the way the stack is used (pushing and popping elements) remains consistent regardless of the underlying implementation.

By using abstract data types, programmers can focus on the high-level design and functionality of their code, leading to more maintainable and modular software systems.

**Explain the computational complexity with three notations.**

Computational complexity is a way to analyse and describe the efficiency of algorithms by measuring how their performance scales with input size. Three commonly used notations for expressing computational complexity are Big O notation, Big Omega notation, and Big Theta notation. Each notation provides different insights into the behaviour of an algorithm:

1. Big O Notation (O-notation):

- Big O notation provides an upper bound on the growth rate of an algorithm. It describes the worst-case scenario in terms of time or space complexity.

- Notation: O(f(n)), where f(n) represents an upper bound on the algorithm's growth rate.

- Example: If an algorithm has a time complexity of O(n^2), it means that the algorithm's running time grows no faster than n^2 as the input size (n) increases. It represents an upper limit on how inefficient the algorithm can be.

2.Big Omega Notation (Ω-notation):

- Big Omega notation provides a lower bound on the growth rate of an algorithm. It describes the best-case scenario in terms of time or space complexity.

- Notation: Ω(f(n)), where f(n) represents a lower bound on the algorithm's growth rate.

- Example: If an algorithm has a time complexity of Ω(n), it means that the algorithm's running time is at least linear, and it grows as fast or faster than n as the input size increases. It represents the lower limit on how efficient the algorithm can be.

3.Big Theta Notation (Θ-notation):

- Big Theta notation provides a tight bound on the growth rate of an algorithm. It describes both the upper and lower bounds on the algorithm's complexity, indicating that it has a specific, predictable behaviour.

- Notation: Θ(f(n)), where f(n) represents both upper and lower bounds on the algorithm's growth rate.

- Example: If an algorithm has a time complexity of Θ(n), it means that the algorithm's running time grows linearly with the input size and is both the best and worst-case scenario. The algorithm's behaviour is well-defined and predictable.

These notations are used to express the efficiency of algorithms and data structures and help programmers and computer scientists compare different algorithms and make informed choices when selecting algorithms for specific tasks. It's important to note that these notations are simplifications and are used for asymptotic analysis, focusing on the behaviour of algorithms as input size approaches infinity. Actual performance can vary based on the specific implementation, hardware, and constant factors.

**Differentiate between primitive data structure and non-primitive data structure.**

Primitive data structures and non-primitive data structures are fundamental concepts in computer science and data organization. Here's how they differ:

Primitive Data Structures:

Primitive Data Structures directly operate according to the machine instructions. These are the primitive data types. Data types like int, char, float, double, and pointer are primitive data structures that can hold a single value.

1.Basic Data Types: Primitive data structures are the basic or fundamental data types provided by a programming language. They are usually built into the language and are used to represent simple values.

2. Fixed Size: These data structures have a fixed size and memory allocation. They are typically of a fixed length, and their size is known at compile time.

3. Efficiency: Primitive data structures are efficient in terms of memory usage and performance because they are directly supported by hardware and the programming language.

4. Examples: Common examples of primitive data structures include integers, floating-point numbers, characters, and Booleans. In some languages, they may also include simple types like Enums.

5. Immutable: Primitive data types are often immutable, meaning their values cannot be changed once assigned. For example, you can't change the value of an integer; you can only create a new integer with a different value.

Non-Primitive Data Structures (Abstract Data Types):

Non-primitive data structures are complex data structures that are derived from primitive data structures. Non – Primitive data types are further divided into two categories. 5.2.1 Linear Data Structure 5.2.2 Non – Linear Data Structure

1. Derived Data Types: Non-primitive data structures are derived or composite data types built by combining one or more primitive or non-primitive data types. They are not part of the core language but are created by programmers.

2. Dynamic Size: Non-primitive data structures are typically dynamic in size. They can grow or shrink as needed during program execution and are often allocated on the heap or dynamically in memory.

3. Complexity: Non-primitive data structures can be more complex and versatile than primitive data structures. They are used to solve more complex problems and provide flexibility in data representation and manipulation.

4. Examples: Common examples of non-primitive data structures include arrays, lists, stacks, queues, trees, graphs, and custom data structures created by programmers.

5. Mutable: Non-primitive data structures are often mutable, meaning their contents can be modified after creation. For example, you can add or remove elements from a list or tree.

In summary, primitive data structures are the basic, fixed-size data types provided by a programming language for representing simple values, while non-primitive data structures, often referred to as abstract data types (ADTs), are derived, dynamic-sized data types created by combining primitive and non-primitive data types to handle more complex data and problem-solving requirements. Non-primitive data structures are essential for solving a wide range of real-world problems and building more sophisticated software applications.

**Differentiate between linear data structure and non-linear data structure.**

Linear data structures and non-linear data structures are two categories of data structures that differ in the way they organize and represent data elements. Here's a differentiation between the two:

Linear Data Structure:

1. Organization: In a linear data structure, data elements are organized in a sequential or linear manner, where each element has a unique predecessor and successor, except for the first and last elements. It forms a one-dimensional arrangement.

2. Traversal: Elements in a linear data structure can be traversed in a single, linear sequence, from the beginning to the end or vice versa.

3. Examples: Common examples of linear data structures include arrays, linked lists, stacks, and queues. These data structures are well-suited for modelling scenarios where data elements have a clear order or sequence.

4. Characteristics: Linear data structures are often used for tasks like storing and accessing data in a straightforward manner. They are efficient for operations that involve sequential processing of data.

Non-Linear Data Structure:

1. Organization: Non-linear data structures do not have a simple, linear organization. Elements are connected in a way that forms a more complex or branching structure. Data elements may have multiple predecessors and successors, leading to a tree-like or graph-like structure.

2. Traversal: Elements in a non-linear data structure cannot be traversed in a simple linear sequence. Traversal typically involves navigating through branches or connections between elements.

3. Examples: Common examples of non-linear data structures include trees (such as binary trees and AVL trees), graphs (including directed and undirected graphs), and hash tables (which use a non-linear mapping of keys to values).

4. Characteristics: Non-linear data structures are used to represent relationships and connections between data elements in a more complex way. They are often employed when modelling hierarchical or network-like structures, such as file systems, computer networks, and hierarchical data.

In summary, the key distinction between linear and non-linear data structures is the way they organize and represent data. Linear data structures are characterized by a simple, one-dimensional organization with a clear sequence of elements, while non-linear data structures feature more complex, branching, and interconnected relationships among data elements. The choice of which data structure to use depends on the specific requirements of the problem being solved and the relationships between the data elements.

**: Differentiate between static data structure and dynamic data structure.**

Static data structures and dynamic data structures differ in terms of their memory allocation and flexibility. Here's a differentiation between the two:

Static Data Structure:

1. Memory Allocation: Static data structures have fixed memory allocation at compile-time. The amount of memory needed for these structures is determined in advance, and it cannot be changed during program execution.

2. Size: The size of static data structures remains constant throughout the program's execution. It is determined when the structure is defined and does not change.

3. Flexibility: Static data structures are not flexible in terms of size. If the allocated memory is insufficient, it can lead to inefficiency or even program failure in cases where the size of data exceeds the allocated space.

4. Examples: Common examples of static data structures include arrays and simple data structures like structs in C or records in some programming languages.

5. Memory Efficiency: Static data structures are generally memory-efficient because they allocate a fixed amount of memory. However, this efficiency comes at the cost of limited flexibility.

Dynamic Data Structure:

1. Memory Allocation: Dynamic data structures allocate memory at runtime, as needed. They can grow or shrink based on the data elements being stored.

2. Size: The size of dynamic data structures can change during program execution, allowing for efficient memory utilization. Memory is allocated or deallocated dynamically as elements are added or removed.

3. Flexibility: Dynamic data structures are highly flexible in terms of size. They can adapt to varying data requirements, which makes them suitable for managing dynamic or unpredictable datasets.

4. Examples: Common examples of dynamic data structures include linked lists, dynamic arrays (e.g., ArrayList in Java or std::vector in C++), stacks, queues, trees, and graphs.

5. Memory Efficiency: Dynamic data structures may consume more memory than their static counterparts because they allocate memory as needed. However, they provide greater flexibility and can handle varying data sizes efficiently.

In summary, the key difference between static and dynamic data structures lies in how they handle memory allocation and size flexibility. Static data structures have a fixed, predetermined size allocated at compile-time, while dynamic data structures allocate and deallocate memory at runtime, allowing them to adapt to changing data requirements. The choice between static and dynamic data structures depends on the specific needs of the application, with dynamic structures being more suitable for situations where data size varies or is unpredictable.

Km=

explain algorithm of reverse polish notation

Reverse Polish Notation (RPN), also known as postfix notation, is a mathematical notation in which operators follow their operands. RPN is used to represent mathematical expressions in a way that can be evaluated using a stack-based algorithm. It's particularly useful for parsing and evaluating mathematical expressions without the need for parentheses to indicate operator precedence. Here's a step-by-step algorithm for evaluating RPN expressions:

**Algorithm for Evaluating Reverse Polish Notation (RPN):**

1. Create an empty stack to hold operands and intermediate results.
2. Split the RPN expression into individual tokens (operands and operators). You can use spaces or other delimiters to separate tokens.
3. Iterate through each token in the expression from left to right.
   * If the token is an operand, push it onto the stack.
   * If the token is an operator, pop the required number of operands from the stack (usually two for binary operators), apply the operator to those operands, and push the result back onto the stack.
4. Continue this process until you've processed all the tokens in the expression.
5. Once you've processed all the tokens, the final result should be the only item left on the stack. Pop it to obtain the result of the RPN expression.

**Example:** Let's evaluate the RPN expression: "3 4 2 \* 1 5 - 2 3 \* + \* 6 -"

* Initialize an empty stack.
* Process each token from left to right:
  + "3" (operand) - Push onto the stack: Stack: [3]
  + "4" (operand) - Push onto the stack: Stack: [3, 4]
  + "2" (operand) - Push onto the stack: Stack: [3, 4, 2]
  + "\*" (operator) - Pop two operands, perform the operation, and push the result: Stack: [3, 8]
  + "1" (operand) - Push onto the stack: Stack: [3, 8, 1]
  + "5" (operand) - Push onto the stack: Stack: [3, 8, 1, 5]
  + "-" (operator) - Pop two operands, perform the operation, and push the result: Stack: [3, 8, -4]
  + "2" (operand) - Push onto the stack: Stack: [3, 8, -4, 2]
  + "3" (operand) - Push onto the stack: Stack: [3, 8, -4, 2, 3]
  + "\*" (operator) - Pop two operands, perform the operation, and push the result: Stack: [3, 8, -12]
  + "+" (operator) - Pop two operands, perform the operation, and push the result: Stack: [-9]
  + "\*" (operator) - Pop two operands, perform the operation, and push the result: Stack: [54]
  + "6" (operand) - Push onto the stack: Stack: [54, 6]
  + "-" (operator) - Pop two operands, perform the operation, and push the result: Stack: [48]
* The final result is 48, which is the only element left on the stack.

This algorithm allows you to evaluate RPN expressions efficiently without the need for operator precedence rules or parentheses. It's often used in calculators and programming languages like Forth and PostScript.

write down advantages of linked list over array and explain in details

Linked lists and arrays are both data structures used for storing and managing collections of elements, but they have different advantages and use cases. Here are some advantages of linked lists over arrays, explained in detail:

1. **Dynamic Size:**
   * **Linked List:** Linked lists have a dynamic size. Elements can be easily inserted or removed at any position in a linked list without the need to reallocate memory or shift other elements. This flexibility is especially useful when dealing with data of varying or unpredictable sizes.
   * **Array:** Arrays have a fixed size that is typically determined at the time of declaration. To accommodate more elements in an array, you may need to create a new, larger array, copy the elements from the old array to the new one, and update references. This can be an inefficient process.
2. **Memory Allocation Flexibility:**
   * **Linked List:** Linked lists allocate memory for each element individually. This allows for efficient memory usage, as you only allocate memory for the elements you need. It's particularly useful when memory is a concern, or you have a large number of elements with varying sizes.
   * **Array:** Arrays allocate a block of memory that is contiguous. The entire block must be allocated at once, which can lead to memory wastage if the array size is significantly larger than the number of elements it needs to hold.
3. **Constant-Time Insertions and Deletions (for singly and doubly linked lists):**
   * **Linked List:** In a singly linked list, insertions and deletions at the beginning or after a given node can be done in constant time O(1) because you only need to update a few pointers. In doubly linked lists, you can also efficiently delete a node if you have a reference to it.
   * **Array:** In arrays, insertions and deletions often require shifting elements to accommodate new values or fill gaps left by removed elements. In the worst case, this can take linear time O(n), where n is the number of elements.
4. **No Need for Predefined Size:**
   * **Linked List:** Linked lists do not require you to specify a size upfront, making them suitable for scenarios where the number of elements can change over time or is unknown.
   * **Array:** In contrast, arrays typically need a predefined size, which can lead to either underestimation (not enough space) or overestimation (wasted memory) of the required space.
5. **Ease of Insertions in the Middle:**
   * **Linked List:** Linked lists make it relatively easy to insert elements at any position within the list, as long as you have a reference to the node before the insertion point.
   * **Array:** In arrays, inserting an element in the middle requires shifting elements after the insertion point, resulting in time-consuming operations.

While linked lists have these advantages, it's essential to note that arrays are also valuable data structures in many situations, particularly when you require direct access to elements by index or when memory allocation efficiency is not a primary concern. The choice between linked lists and arrays depends on the specific requirements of your application and the trade-offs between dynamic sizing, memory efficiency, and constant-time insertions and deletions.

explain AVL tree with example

An AVL tree, named after its inventors Adelson-Velsky and Landis, is a self-balancing binary search tree. It's designed to maintain its balance by ensuring that the heights of the left and right subtrees of any node differ by at most one. This balancing property ensures that the tree remains relatively balanced, resulting in efficient operations such as insertion, deletion, and searching with a time complexity of O(log n), where "n" is the number of nodes in the tree.

Here's an explanation of AVL trees with an example:

**AVL Tree Properties:**

1. For any node in the AVL tree, the heights of its left and right subtrees (the balance factor) can differ by at most one.
2. Each subtree of an AVL tree is itself an AVL tree.

trace the algorithm showing stack to calculate the factorial of n=5

To calculate the factorial of a number "n" using a stack, you can use a simple iterative algorithm that pushes and pops values onto and from the stack. Here's a step-by-step trace of the algorithm to calculate the factorial of n=5:

1. Initialize a stack and a variable **result** to 1.
   * Stack: []
   * result: 1
2. Start a loop from **i = 2** to **n = 5** (inclusive).

**Iteration 1: i = 2**

* + Push **i** onto the stack.
  + Stack: [2]

**Iteration 2: i = 3**

* + Push **i** onto the stack.
  + Stack: [2, 3]

**Iteration 3: i = 4**

* + Push **i** onto the stack.
  + Stack: [2, 3, 4]

**Iteration 4: i = 5**

* + Push **i** onto the stack.
  + Stack: [2, 3, 4, 5]

1. Perform a loop to calculate the factorial by popping values from the stack.

**Iteration 1:**

* + Pop **5** from the stack and multiply it with **result**.
  + result = 5 \* 1 = 5
  + Stack: [2, 3, 4]

**Iteration 2:**

* + Pop **4** from the stack and multiply it with **result**.
  + result = 4 \* 5 = 20
  + Stack: [2, 3]

**Iteration 3:**

* + Pop **3** from the stack and multiply it with **result**.
  + result = 3 \* 20 = 60
  + Stack: [2]

**Iteration 4:**

* + Pop **2** from the stack and multiply it with **result**.
  + result = 2 \* 60 = 120
  + Stack: []

1. End of the loop.
2. The value of **result** is now the factorial of 5.
   * result = 120

#include <stdio.h>

#define MAX\_STACK\_SIZE 100

int stack[MAX\_STACK\_SIZE];

int top = -1;

void push(int value) {

if (top < MAX\_STACK\_SIZE - 1) {

stack[++top] = value;

} else {

printf("Stack overflow!\n");

}

}

int pop() {

if (top >= 0) {

return stack[top--];

} else {

printf("Stack underflow!\n");

return -1; // Return -1 to indicate an error

}

}

int calculateFactorial(int n) {

if (n < 0) {

return -1; // Factorial is not defined for negative numbers

} else if (n == 0) {

return 1; // 0! is defined as 1

}

int i, result = 1;

// Push values onto the stack from 2 to n

for (i = 2; i <= n; i++) {

push(i);

}

// Calculate the factorial by popping values from the stack

while (top >= 0) {

result \*= pop();

}

return result;

}

int main() {

int n, factorial;

printf("Enter a non-negative integer: ");

scanf("%d", &n);

factorial = calculateFactorial(n);

if (factorial != -1) {

printf("%d! = %d\n", n, factorial);

} else {

printf("Factorial is not defined for negative numbers.\n");

}

return 0;

}

what is sparse matrix ? explain memory representation of sparse matrix

A sparse matrix is a matrix in which most of the elements are zero or have some default or non-significant value. Sparse matrices often occur in various real-world applications, such as text analysis, image processing, network analysis, and scientific computing, where most of the data is zero or lacks meaningful information. Storing all elements of a sparse matrix as if they were dense (non-sparse) can be highly inefficient in terms of memory usage. Therefore, specialized data structures are used to represent sparse matrices more efficiently.

**Memory Representation of Sparse Matrices:**

Sparse matrices are typically represented using one of the following common techniques to save memory:

1. **Compressed Sparse Row (CSR) or Compressed Row Storage (CRS):** In this representation, you store the non-zero elements of the matrix in three separate one-dimensional arrays:
   * **values**: An array containing the values of the non-zero elements, stored in row-major order.
   * **columns**: An array containing the column indices corresponding to the non-zero elements.
   * **row\_ptr**: An array that points to the beginning of each row in the **values** and **columns** arrays.

differentiate between array and stack

Arrays and stacks are both data structures used for organizing and storing collections of elements, but they have distinct characteristics and purposes. Here's a differentiation between arrays and stacks:

**Arrays:**

1. **Data Organization:**
   * Arrays are a fundamental data structure that organizes elements of the same data type in a contiguous block of memory.
   * Elements in an array can be accessed by their indices, which are typically integers.
2. **Size:**
   * Arrays have a fixed size determined at the time of declaration, and it remains constant throughout their lifetime.
   * The size must be known in advance, and changing it requires creating a new array and copying data, which can be inefficient.
3. **Operations:**
   * Arrays support basic operations such as element access, insertion, and deletion, but these operations may be less efficient when compared to specialized data structures for these tasks.
4. **Random Access:**
   * Arrays allow for efficient random access to elements since you can access elements by their index in O(1) time.

**Stacks:**

1. **Data Organization:**
   * Stacks are an abstract data type (ADT) that can be implemented using arrays or linked lists.
   * They follow the Last-In-First-Out (LIFO) principle, where the last element added is the first one to be removed.
2. **Size:**
   * Stacks can be implemented as dynamic data structures, allowing elements to be added and removed without the need to specify a fixed size.
3. **Operations:**
   * Stacks support two primary operations: push (to add an element to the top of the stack) and pop (to remove the top element).
   * They are well-suited for managing function calls, tracking state, and solving problems where elements need to be processed in reverse order.
4. **Access Pattern:**
   * Stacks are generally used for processing elements in a specific order, often related to nested structures or reversing data.

In summary, while arrays are a low-level data structure for storing elements in contiguous memory, stacks are a higher-level abstraction that focuses on specific access patterns, especially LIFO processing. Arrays provide efficient random access to elements, while stacks emphasize adding and removing elements in a prescribed order, making them suitable for a different set of use cases. Depending on your needs, you may choose to use an array when random access and fixed size are essential, or a stack when LIFO behaviour and dynamic sizing are more relevant.

give advantages and use of circular linked list

ChatGPT

**Advantages of Circular Linked Lists:**

1. **Efficient Cyclic Operations:** Circular linked lists are well-suited for cases where cyclic operations are needed. For example, in applications like scheduling or simulation, where you want a process or event to loop back to the beginning once the end is reached.
2. **Constant-Time Insertions and Deletions:** Insertions and deletions at the beginning and end of a circular linked list can be done in constant time, just like with a regular singly linked list. This makes circular linked lists efficient for certain operations.
3. **Efficient Memory Utilization:** Circular linked lists can save memory compared to doubly linked lists because they have fewer pointers per node, but they still offer some of the benefits of linked data structures. This can be advantageous in embedded systems or scenarios with limited memory.
4. **Representing Circular Data:** Circular linked lists are suitable for representing cyclic data structures, such as circular buffers or queues, where elements wrap around after reaching the end of the data structure.

**Use of Circular Linked Lists:**

1. **Round-Robin Scheduling:** Circular linked lists are commonly used in scheduling algorithms, such as round-robin scheduling in operating systems. In this context, each element in the list represents a process, and they take turns executing in a cyclic order.
2. **Circular Queues:** Circular linked lists are used in the implementation of circular queues, which are data structures where elements are dequeued and enqueued in a circular manner. They are useful in scenarios like managing printer job queues or handling tasks in a cyclic manner.
3. **Game Development:** In game development, circular linked lists can be used to manage game object updates in a loop, where the last object's update triggers the update of the first object, creating a continuous loop of updates.
4. **Music and Media Players:** Circular linked lists can be used in the implementation of playlist management for music or media players, where you can create a loop of songs or media items to play in a continuous cycle.
5. **Simulation and Modelling:** In simulations and modelling, circular linked lists can be used to model cyclic behaviours or processes, where the simulation loops back to the beginning after a certain point.
6. **Data Structures:** Circular linked lists can be used to implement various other data structures, such as a circular doubly linked list (a circular version of a doubly linked list) or a circular singly linked list with a sentinel node, where the last node points back to the head, simplifying certain operations.

Overall, circular linked lists are valuable in situations where elements need to cycle or loop in a continuous manner and for efficient insertions and deletions at the beginning or end of the list. They find applications in scheduling, circular queues, game development, playlist management, and various other scenarios where cyclic behaviour is relevant.

explain priority queue

ChatGPT

A priority queue is an abstract data type that stores a collection of elements, each associated with a priority. The primary purpose of a priority queue is to efficiently retrieve and process elements based on their priorities. Elements with higher priorities are typically processed or removed before elements with lower priorities.

A priority queue supports two fundamental operations:

1. **Insertion (Enqueue):** This operation allows you to insert an element into the priority queue along with its associated priority. The element is placed in a position such that the elements with the highest priorities are at the front of the queue.
2. **Extraction (Dequeue):** This operation retrieves and removes the element with the highest priority from the priority queue. After the extraction, the priority queue is updated to maintain its structure, ensuring that the next element to be dequeued will also have a high priority.

Priority queues can be implemented using various data structures, and the choice of implementation affects the time complexity of these operations. Common implementations of priority queues include:

1. **Binary Heap:** Binary heaps, whether in the form of a min-heap or a max-heap, are one of the most common implementations of priority queues. They provide efficient insertion and extraction operations with time complexities of O(log n), where "n" is the number of elements in the queue.
2. **Fibonacci Heap:** Fibonacci heaps are more complex data structures that offer even more efficient amortized time complexities for some operations, such as constant-time insertions and improved decrease-key operations. However, they are more complex to implement.
3. **Pairing Heap:** Pairing heaps are simpler than Fibonacci heaps but still provide good performance for priority queue operations. They are often used in practice for their simplicity.
4. **Unsorted Array or Linked List:** While not the most efficient, an unsorted array or linked list can be used to implement a priority queue with constant-time insertions, but extracting the highest-priority element would take linear time (O(n)). This is practical for small data sets or situations where insertion performance is more critical.

Priority queues are used in a wide range of applications, including:

* Dijkstra's algorithm for finding the shortest path in a graph.
* Prim's algorithm for finding minimum spanning trees.
* Huffman coding for data compression.
* Task scheduling and job processing in operating systems.
* A\* search algorithms in pathfinding for games and robotics.
* Simulations and event-driven systems where events occur at different times and must be processed in order of their priority.

In summary, a priority queue is a data structure that allows efficient insertion and extraction of elements based on their associated priorities. It is a fundamental tool in computer science and is used in various algorithms and applications to solve problems that involve prioritizing and processing elements.

Explain the following terms with reference to TREE. (i) node (ii) root (iii) child (iv) parent (v) path (vi) leaf (vii) depth (viii) height (ix) degree (x) level

ChatGPT

In the context of a tree data structure, the following terms are commonly used to describe various aspects of the tree:

1. **Node:**
   * A node is a fundamental building block of a tree. It represents an element within the tree.
   * A node contains data (a value or key) and may have links to zero or more child nodes.
2. **Root:**
   * The root is the topmost node in a tree. It serves as the starting point for traversing the tree.
   * Every tree has exactly one root node.
3. **Child:**
   * A child is a node that is connected to another node called its parent.
   * In a tree, each node, except the root, has one parent, and zero or more children.
4. **Parent:**
   * A parent is a node that has one or more child nodes connected to it.
   * The node from which an edge (connection) emanates to reach its child is referred to as the parent of the child node.
5. **Path:**
   * A path in a tree is a sequence of nodes in which each node is connected to the next node by an edge (link).
   * The path starts at a particular node and ends at another node in the tree.
6. **Leaf:**
   * A leaf node is a node in the tree that has no children, meaning it is at the outermost level and does not have any child nodes connected to it.
7. **Depth:**
   * The depth of a node is the number of edges in the path from the root node to that node.
   * The root node has a depth of 0, and the depth of other nodes is determined by how many edges they are away from the root.
8. **Height:**
   * The height of a tree is the maximum depth among all nodes in the tree. In other words, it is the length of the longest path from the root to a leaf.
   * The height of a tree is often used as a measure of its overall balance and efficiency.
9. **Degree:**
   * The degree of a node is the number of children it has. A node's degree can vary in a tree, and nodes with zero children are called leaf nodes.
10. **Level:**
    * The level of a node refers to its position in the tree with respect to the root. The root is at level 0, and its immediate children are at level 1, and so on.
    * The level of a node is equal to its depth plus one.

**Applications of Queue**

▪ Queue used as waiting lists for a single shared resource like printer, disk, CPU, etc.

▪ It used as buffer on MP3 player, iPod playlist.

▪ It used in Operating System for handling interrupts, Job Scheduling and process scheduling.

▪ Queue of people at any service point such as ticketing etc.

▪ Queue of airplanes waiting for landing instructions.

▪ Queue is used in BFS (Breadth First Search) algorithm. It helps in traversing a tree or graph.

▪ Queue is used in networking to handle congestion