**Heaps, Stacks and Queues**

viernes, 21 de abril de 2023

11:43 p. m.

Heaps

The Roadmap Intro: [Heap Data Structure | Illustrated Data Structures](https://www.youtube.com/watch?v=F_r0sJ1RqWk)

Programiz ref: <https://www.programiz.com/dsa/heap-data-structure>

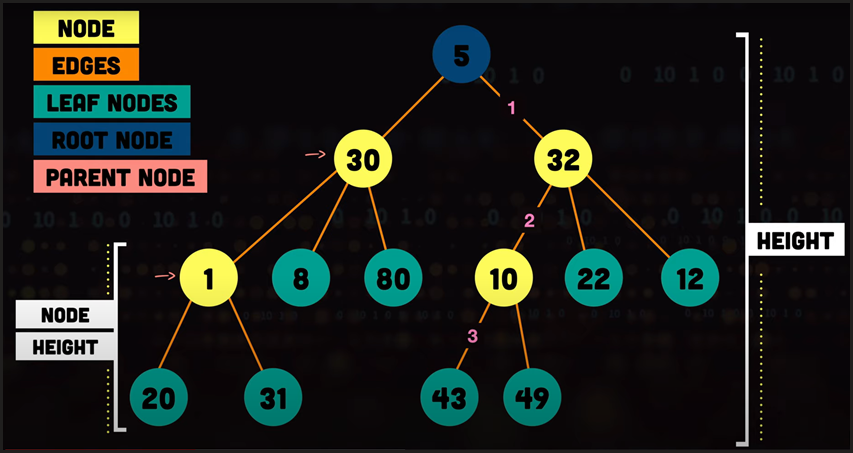
Scrapbook ref: <https://stephanosterburg.gitbook.io/scrapbook/coding/coding-interview/data-structures/heaps-stacks-queues>

Educative - Heap implementation for Python: <https://www.educative.io/answers/heap-implementation-in-python>

First, Trees

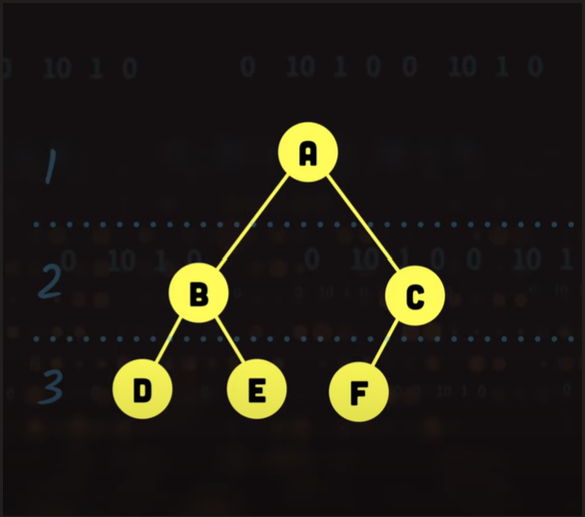
This data structure is a Hierarchical data structure, meaning that is not linear like linked lists, queues or stacks. The heaps are based on the "Tree" data structure, which is composed of nodes, edges, leads nodes, a root node and other parent nodes.

The tree height is determined by the distance (in edges) between the root node and the leaf nodes, for this case is 3. and the node height is determined the same way but this applies to parent nodes up to the leaves, so for the case of the node (1), its node height is 1, but for the node (30) will be 2.



There are many variations for this data structure but the ones we are interested on and that are related to heaps are binary trees. A binary tree is a tree data structure where all the parent nodes have a maximum of 2 child nodes, and more specifically for heaps the tree must be a Completed Binary Tree.

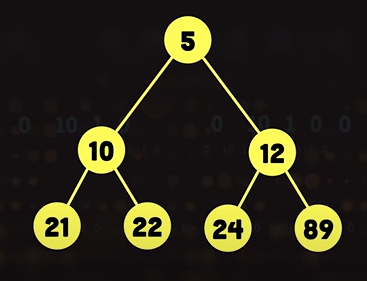
A Completed Binary Tree is a tree in which all levels of the tree are filled, except last level which may be filled from the left.



And, now Heaps

Heaps are essentially complete binary tree that could be either Min Heap or Max Heap.

Min Heap: A tree where all the parents have values that are lesser or equal than the child's.



Max Heap: A tree where all the parents have values that are greater or equal than the child's.



Heap Operations

Insertion: From top to bottom, left to right. Depending if is a Min or a Max heap, the way to insert a new node is inserting them as a child and comparing it to it parent, for Min, if the child is less than the parent the they must be swapped and analogically, for Max is the same but the condition is the opposite, meaning greater than the parent. Complexity: O(log n).

Delete: The deletion process is to go take out the root and take the last node (which will always be the far right most at the last level of the tree and take it as the new root and iteratively start comparing the parent to its children and bubbling up to swap to maintain the condition either for Min or Max. The comparison between the parent and the children will be always to swap the parent with the smallest/greatest depending the kind of heap, until the conditions are met. Complexity: O(log n).

Tabla

Descripción generada automáticamente

Heap Operations in Python

For managing heaps Python counts with a built-in module called 'heapq' and among others count with the following functions:

* heapify(iterable): This function converts an iterable (such as a list) into a valid heap, in-place. It has a time complexity of O(n), making it useful for quickly converting an existing list into a heap.

* heappush(heap, item): This function adds an item to the heap while maintaining the heap property. It has a time complexity of O(log n), where n is the number of elements in the heap.

* heappop(heap): This function removes and returns the smallest element from the heap while maintaining the heap property. It has a time complexity of O(log n).

* heapreplace(heap, item): This function removes and returns the smallest element from the heap and then pushes the new item onto the heap. It's more efficient than calling heappop() followed by heappush(), as it avoids the need to search the heap twice.

* heappushpop(heap, item): This function is similar to heapreplace(), but it pushes the new item onto the heap before popping the smallest element. It can be more efficient than using separate heappush() and heappop() calls.

* heapq.nsmallest(n, iterable): This function returns the n smallest elements from an iterable, using a heap-based algorithm. It has a time complexity of O(n log n).

* heapq.nlargest(n, iterable): This function returns the n largest elements from an iterable, using a heap-based algorithm. It has a time complexity of O(n log n).

* \*heapq.merge(iterables): This function takes multiple sorted input iterables and returns an iterator that produces the sorted combination of all elements using a heap-based merge algorithm.

* heapq.heapreplace(heap, item): Similar to heapreplace(), but this function allows replacing the smallest element in the heap with a new item while ensuring the heap property.

* heapq.heapqpop(item): Similar to heappop(), but it's useful when working with min-heaps.

* heapq.heapqpushpop(heap, item): Similar to heappushpop(), but for min-heaps.

Strengths and Weaknesses

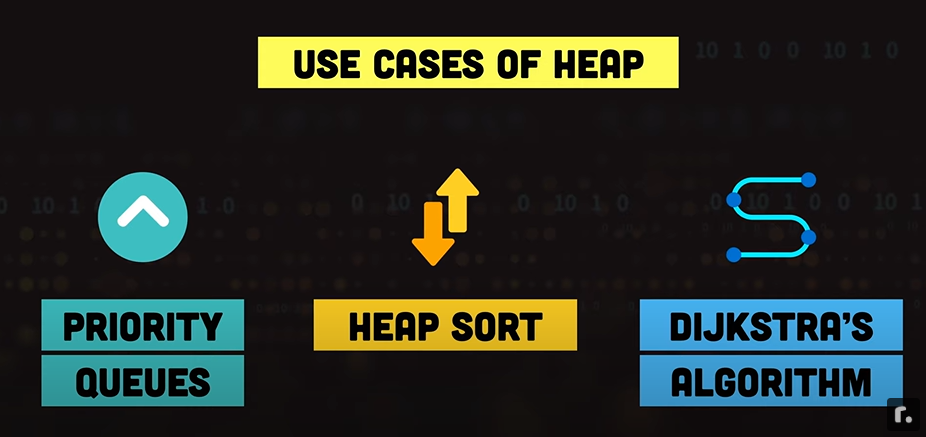
**Strengths**

* A min-heap is able to quickly extract the minimum value on the heap. Repeated extractions from a min-heap into an array will yield a sorted array.

**Weaknesses**

* There's no convenient way of searching for a particular key value in a heap. Entries are only partially ordered; clever use of the heap property can allow for some pruning of searches.

Heap Use Cases



The most common use cases for this data structure is basically whenever a largest or smallest elements is needed to have it at hand and with the same logic, the structure arranged accordingly. Here another some common use cases for heaps:

1. Priority Queue: Heaps are often used to implement priority queues. A priority queue is a data structure that allows efficient insertion, deletion, and retrieval of the element with the highest (or lowest) priority. In medical systems, task scheduling, and Dijkstra's algorithm for finding the shortest path, priority queues are essential.

1. Heap Sort: Heapsort is an efficient sorting algorithm that uses the heap data structure to sort elements in ascending or descending order. It has a time complexity of O(n log n) and is an in-place sorting algorithm.

1. Median Finding: In scenarios where you need to find the median of a set of numbers (the middle value when the numbers are sorted), heaps can be used to efficiently find the median in constant time or logarithmic time.

1. K-Smallest/K-Largest Elements: Heaps can help efficiently find the K-smallest or K-largest elements in a large dataset without having to sort the entire dataset.

1. Dijkstra's Shortest Path Algorithm: The heap data structure is used in Dijkstra's algorithm to efficiently find the shortest path in a weighted graph from a given source vertex to all other vertices.

1. Huffman Coding: In data compression algorithms like Huffman coding, heaps are used to construct the optimal prefix-free binary code for a given set of characters based on their frequencies.

1. Merge Sorted Lists: When merging multiple sorted lists, a heap can be used to efficiently merge them and obtain a sorted result.

1. Event Scheduling: In event-driven simulation or scheduling systems, heaps can be used to efficiently manage and process events in chronological order.

Heaps are designed to do one specific thing well, so the answer to *when you should use a heap* is repetitive: You use one when you have to do repeated minimum (or maximum) extractions. The problems where you would want to do this might look quite different from each other, however. Below we have selected some examples of common interview questions that benefit from heaps.

* Finding the minimum distance between two nodes in a graph: The standard approach to this problem is to use Dijkstra's algorithm. One of the key steps in Dijkstra's algorithm is to select the node closest to a node that you have already completed, which is a minimum calculation.

* Getting the next event that is scheduled to occur: Storing events in a heap with a timestamp as the key gives you a fast way to extract the next event (the event with the smallest timestamp will occur next).

* Keep track of the median value while streaming: This is the running median problem, where two heaps are maintained: a max-heap for values below the current median and a min-heap for values above the current median. When a new value is inserted, it is placed in the low or high pile as appropriate (and the maximum of the low values or minimum of the low values are extracted as necessary to keep the two heaps sizes' different by at most one element).

* Find the first k non-repeating characters in a string in a single traversal.

Fibonacci's Heap

[Fibonacci heaps in 6 minutes — Intro](https://www.youtube.com/watch?v=0vsX3ZQFREM)

Texto

Descripción generada automáticamente con confianza media

Fibonacci's Heap Use Cases

Fibonacci heaps are specialized data structures that offer efficient amortized time complexity for certain operations compared to other types of heaps, such as binary heaps. However, they are also more complex to implement and maintain. As a result, they are not commonly used in practice for most scenarios. Despite this, there are specific use cases where Fibonacci heaps can provide advantages:

1. Dijkstra's Algorithm: In some cases, when Dijkstra's algorithm is used to find the shortest paths in a graph with non-negative edge weights, Fibonacci heaps can offer better time complexity for decrease-key operations, which are common in the algorithm. This can result in improved overall performance compared to binary heaps.

1. Prim's Algorithm: Similar to Dijkstra's algorithm, Fibonacci heaps can be used in Prim's algorithm to find minimum spanning trees in graphs with non-negative edge weights. The improved amortized time complexity for decrease-key operations can be advantageous in this context.

1. Computing Shortest Paths with Negative Weights: If you're dealing with graphs that include negative edge weights and you need to compute single-source shortest paths, Fibonacci heaps can handle decrease-key operations for negative weight updates more efficiently than binary heaps.

1. Network Flow Algorithms: Certain network flow algorithms, like the Capacity Scaling algorithm, can benefit from the better amortized time complexity of decrease-key operations that Fibonacci heaps provide.

1. Approximate Algorithms: In cases where approximate solutions are acceptable, the potential improvements in runtime efficiency offered by Fibonacci heaps might be worth considering.

1. Lazy Merging in Priority Queues: Fibbonacci heaps support a "lazy" approach to merging heaps, where two heaps are combined efficiently and merged fully later on. This can be useful in algorithms that require dynamic merging of priority queues.

It's important to note that while Fibonacci heaps offer better theoretical time complexity for certain operations, they often come with a higher constant factor and increased memory usage compared to simpler data structures like binary heaps. As a result, in many practical situations, the advantages of Fibonacci heaps might not outweigh their complexity and overhead, and simpler alternatives like binary heaps or specialized data structures might be preferred.

Stacks

The Roadmap Intro: [Stack Data Structure | Illustrated Data Structures](https://www.youtube.com/watch?v=I5lq6sCuABE)

Programiz ref: <https://www.programiz.com/dsa/stack>

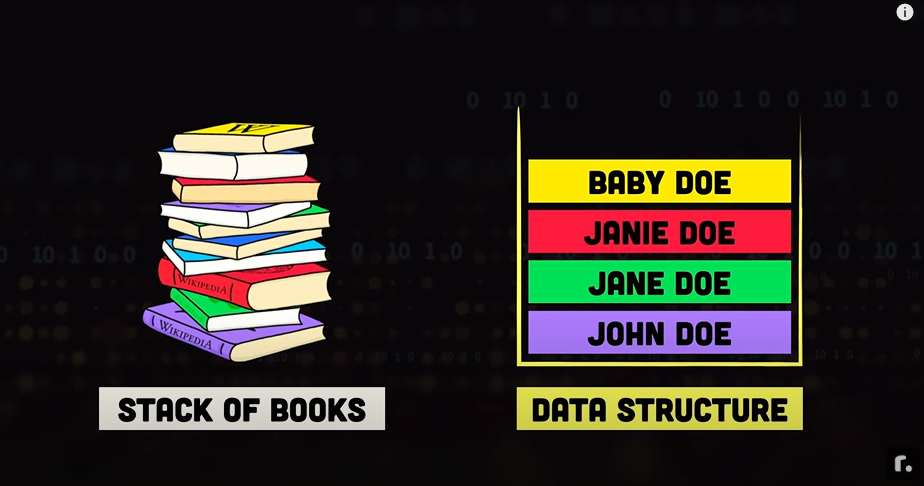
Scrapbook ref: <https://stephanosterburg.gitbook.io/scrapbook/coding/coding-interview/data-structures/heaps-stacks-queues>

Geeksforgeeks ref: <https://www.geeksforgeeks.org/stack-in-python/>

RealPython ref: <https://realpython.com/queue-in-python/>

RealPython Implementation ref: <https://realpython.com/how-to-implement-python-stack/>

A Stack is a linear collection of items that are inserted and removed in a particular order. The particular order stacks follow is LIFO (Last in First Out).



Stack Operations

Insertion: Is basically a Push, to keep the stack property, the element added would be at the top of the stack pushing the existing elements back. Complexity: O(1).

Deletion: Is basically a Pop, to keep the stack property, the taken out would be the top element of the stack pulling the existing elements forward. Complexity: O(1).

isEmpty: A check to see if there is elements present in the object. Complexity: O(1).

Peek: A check to see what is the current leading element in the stack. Complexity: O(1).

Strengths and Weaknesses

**Strengths**

* **Simplicity**: Stacks are easy to understand and implement. Their simplicity makes them suitable for a wide range of applications.

* **LIFO Principle**: The Last-In-First-Out (LIFO) property of stacks makes them ideal for modeling scenarios where the most recently added item is the one of interest, such as function calls or undo operations.

* **Constant-Time Operations:** Basic stack operations (push, pop, peek) have a constant time complexity, which ensures efficient performance regardless of the size of the stack.

* **Memory Management**: Stacks can be useful for managing memory allocation and deallocation in low-level programming languages where memory management is critical.

* **Expression Parsing and Evaluation**: Stacks are particularly useful for parsing and evaluating expressions, especially when handling parentheses and operator precedence.

* **Backtracking and State Management**: Stacks are valuable in algorithms involving backtracking and maintaining a history of states. They provide a simple way to store and restore states.

**Weaknesses**

* **Limited Access**: Stacks offer limited access to elements. You can only access the top element, which can be a drawback if you need to access elements deep within the stack.

* **No Random Access**: Unlike arrays, stacks do not support random access to elements. If you need to access elements at arbitrary positions, stacks are not the right choice.

* **Fixed Capacity (Array Implementation)**: If stacks are implemented using arrays, they may have a fixed capacity. If the stack becomes full, adding more elements can result in overflow errors. Dynamic resizing can mitigate this issue, but it adds complexity.

* **Limited Use Cases**: While stacks are versatile, they're not suitable for all scenarios. Their LIFO nature doesn't match scenarios where items need to be accessed in a different order.

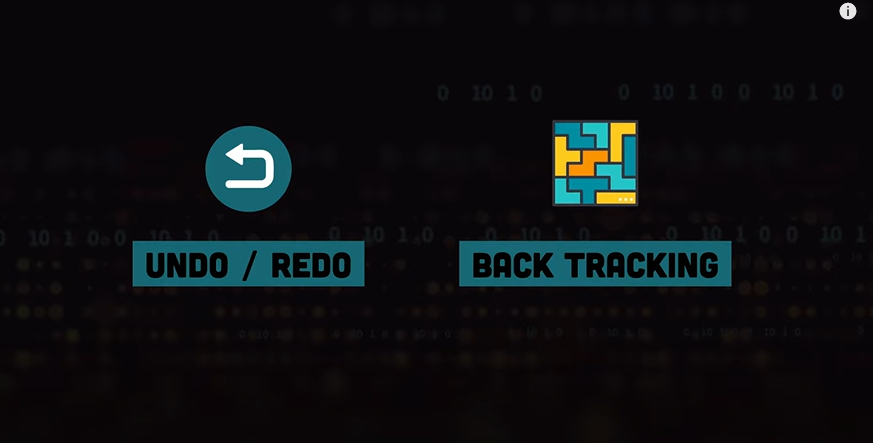
* **Not Suitable for Sorting**: Stacks are not well-suited for sorting algorithms because of their limited access and removal capabilities.

* **Inefficient for Some Operations**: While push, pop, and peek are efficient, other operations like searching for an element within a stack can be inefficient as you have to pop elements off the stack to search.

* **Resource Overhead (Linked List Implementation)**: If stacks are implemented using a linked list, there can be some memory overhead due to the need to store references/pointers for each element.

In summary, stacks are excellent for scenarios involving history tracking, state management, and LIFO behaviors, but they may not be the best choice when you need random access, efficient searching, or support for dynamic resizing. Choosing the right data structure depends on the specific requirements of your application.

Heap Use Cases



1. Undo & Redo Ops: Essentially keeping track of operations or states, with this data structure is easy to do that.

1. Backtrack: Like in solving a maze, if is needed to record previous steps, a way to keeping the record could be through a stack.

1. Function Call Stack: Stacks are used to manage function calls in programming languages. When a function is called, its local variables and return address are stored on the stack. When the function completes execution, it's popped off the stack, and the program resumes from where it left off. This ensures that function calls are managed in a LIFO manner.

1. Expression Evaluation: Stacks can be used to evaluate arithmetic expressions, especially those involving parentheses. They help in maintaining the order of operations, evaluating sub-expressions, and keeping track of operators and operands.

1. Browser History: Stacks are used to implement the back and forward navigation in web browsers. Each page visited is pushed onto the stack, and going back pops pages off the stack.

1. Text Editors and IDEs: Text editors and integrated development environments (IDEs) often use stacks to implement features like undo/redo, indentation, and bracket matching.

1. Memory Management: In some low-level programming languages, stacks are used for managing memory allocation and deallocation, especially in situations where the allocation order must match the deallocation order (e.g., managing local variables and function calls).

1. Parentheses Matching: Stacks can be used to check if a given string of parentheses (or other brackets) is balanced and properly nested.

1. Simulation and Simulators: Stacks can be used in simulation scenarios to model real-world systems that involve stacking or queuing, such as simulating traffic flow or customer queues.

Implementing a Stack with LifoQueue module

It is possible to call the function 'LifoQueue' from the queue module in python to implement a Stack. I could be also useful to note that a Stack could also be implemented with a singly linked list structure.

This implementation counts with this methods:

* maxsize – Number of items allowed in the queue.

* empty() – Return True if the queue is empty, False otherwise.

* full() – Return True if there are maxsize items in the queue. If the queue was initialized with maxsize=0 (the default), then full() never returns True.

* get() – Remove and return an item from the queue. If the queue is empty, wait until an item is available.

* get\_nowait() – Return an item if one is immediately available, else raise QueueEmpty.

* put(item) – Put an item into the queue. If the queue is full, wait until a free slot is available before adding the item.

* put\_nowait(item) – Put an item into the queue without blocking. If no free slot is immediately available, raise QueueFull.

* qsize() – Return the number of items in the queue.

LifoQueue dir():

['all\_tasks\_done', 'empty', 'full', 'get', 'get\_nowait', 'join', 'maxsize', 'mutex', 'not\_empty', 'not\_full', 'put', 'put\_nowait', 'qsize', 'queue', 'task\_done', 'unfinished\_tasks']

Queue

The Roadmap Intro: [Queue Data Structure | Illustrated Data Structures](https://www.youtube.com/watch?v=mDCi1lXd9hc)

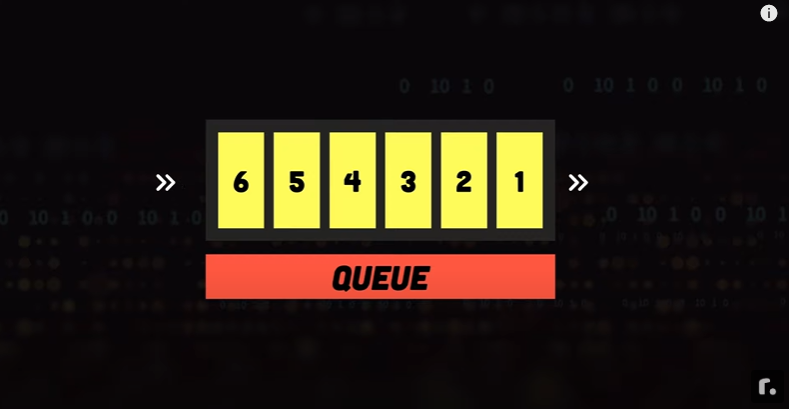
Programiz ref: <https://www.programiz.com/dsa/queue>

Scrapbook ref: <https://stephanosterburg.gitbook.io/scrapbook/coding/coding-interview/data-structures/heaps-stacks-queues>

Geeksforgeeks ref: <https://www.geeksforgeeks.org/queue-in-python/>

RealPython ref: <https://realpython.com/queue-in-python/>

A Queue is a linear collection of items that are inserted and removed in a particular order. The particular order Queues follow is FIFO (FIFO in First Out).



Queue Operations

Enqueue (Insertion): Is basically a Push, to keep the Queue property, the element added would be at the end of the queue leading by the existing elements upfront. Complexity: O(1).

Dequeue (Deletion): Is basically a Pop, to keep the Queue property, the element taken out would be the element upfront of the Queue making the second the head of the Queue. Complexity: O(1).

isEmpty: A check to see if there is elements present in the object. Complexity: O(1).

Peek: A check to see what is the current leading element on the Queue. Complexity: O(1).

Rear: A check to see what is the current trailing element on the Queue. Complexity: O(1).

Strengths and Weaknesses

**Strengths**

* **Simplicity**: Queues are relatively simple to implement and understand, making them a good choice for basic tasks like managing tasks in a printer queue or handling requests in a web server.

* **FIFO Principle**: One of the main strengths of a queue is that it maintains the order in which elements are added. This makes it ideal for scenarios where the order of processing matters, such as task scheduling or printing tasks.

* **Constant-Time Operations:** Basic Queue operations (push, pop, peek) have a constant time complexity, which ensures efficient performance regardless of the size of the stack.

* **Useful in Synchronization**: Queues are commonly used in multithreaded or parallel programming to manage synchronization between different threads or processes. A producer-consumer scenario is a classic example where a queue can help manage communication between threads.

**Weaknesses**

* **Limited Access**: Unlike arrays or lists, queues don't allow direct access to elements in the middle. You can only access and remove elements from the front and add elements to the back. This limitation can be a problem when you need to access or modify elements at arbitrary positions.

* **No Random Access**: Unlike arrays, Queues do not support random access to elements. If you need to access elements at arbitrary positions, queues are not the right choice.

* **Fixed Size Queues**: Some implementations of queues have a fixed size. Once the queue is full, you can't add more elements unless you remove some elements first. This can be a limitation in scenarios where the number of elements varies dynamically.

* **Memory Overhead**: In some implementations, there might be a memory overhead associated with managing the queue. For example, you might need to store additional pointers or bookkeeping data.

* **Not Suitable for Sorting**: Stacks are not well-suited for sorting algorithms because of their limited access and removal capabilities.

* **Not Suitable for Priority-Based Processing**: Queues don't inherently support prioritizing elements based on certain criteria. If you need to process elements based on different priorities, a priority queue might be a better choice.

In summary, queues are great for maintaining order and managing tasks in a sequential manner. They're simple, efficient for specific use cases, and play a crucial role in various programming scenarios. However, they might not be the best choice when direct element access or efficient searching is required.

Queues Use Cases

1. Task Scheduling: Queues are commonly used for scheduling tasks in various systems. For example, in an operating system, tasks are placed in a queue to be executed by the CPU in a first-come, first-served manner.

1. Print Queue: When multiple print jobs are sent to a printer, they are typically managed using a queue. The first job to enter the queue is the first to be printed.

1. Breadth-First Search (BFS): In graph algorithms like BFS, a queue is used to keep track of the nodes that need to be explored. This ensures that nodes are processed in the order they were encountered, leading to an exploration of the graph in a breadth-first manner.

1. Web Server Request Handling: Web servers often use queues to manage incoming requests from clients. Each request is placed in a queue, and the server processes them in the order they were received.

1. Multithreading and Parallel Programming: In multithreaded or parallel programming, queues are used to manage communication between threads or processes. For example, a producer-consumer scenario involves one thread producing data and placing it in a queue, while another thread consumes and processes that data.

Types of Queues

Circular Queues: A circular queue, also known as a circular buffer or a ring buffer, is a type of queue data structure in which the last element is connected to the first element, forming a circle. This circular structure offers some advantages over a regular linear queue in certain scenarios. The main practical use of a circular queue is when you have a fixed-size buffer or memory space and you want to efficiently manage and utilize that space.

Implementing a circular queue in Python makes sense when you have specific use cases that align with its benefits, such as efficient memory usage, circular buffering, and resource management. However, for simpler scenarios where memory efficiency and circular behavior aren't critical, using Python's built-in collections might be more practical.

Link: <https://www.programiz.com/dsa/circular-queue>

Priority Queues: A priority queue is a special type of queue in which each element is associated with a priority value. And, elements are served on the basis of their priority. That is, higher priority elements are served first. However, if elements with the same priority occur, they are served according to their order in the queue.

Generally, the value of the element itself is considered for assigning the priority. For example, The element with the highest value is considered the highest priority element. However, in other cases, we can assume the element with the lowest value as the highest priority element, but we can also set priorities according to our needs.

Priority Queues could be implemented based on lists, linked lists or heaps.

The most common implementation of priority queues are with heaps but for didactic reasons I will implement a Priority Queue class based on list built-in type.

Interfaz de usuario gráfica, Aplicación, Teams

Descripción generada automáticamente

Link: <https://www.programiz.com/dsa/priority-queue>

Deques: Deque or Double Ended Queue is a type of queue in which insertion and removal of elements can either be performed from the front or the rear. Thus, it does not follow FIFO rule (First In First Out).

Also is important to note that any of the deque's operation have a complexity of O(1)

Link: <https://www.programiz.com/dsa/deque>