ALGORITMOS

Bernardo Marujo, nº mec 107322

Algoritmos e Estruturas de Dados
Engenharia de Computadores e Informática

Conteúdo

Capítulo 1

QUEUE

A queue is a data structure that stores elements in a linear order. The elements are stored in such a way that the first element added to the queue is the first one to be removed. This means that the elements are removed in the order in which they were added, and new elements are added to the end of the queue. The queue data structure is commonly used for tasks that require processing items in the order in which they are received. This is because a queue is a First In, First Out (FIFO) data structure, meaning that the first element added to the queue is the first one to be removed. Queues are commonly used in computer systems for tasks such as scheduling processes, storing messages, and handling requests in a server. Queues can be implemented using arrays, linked lists, and other data structures, and they can be implemented in a variety of programming languages.

Capítulo 2

STACK

A stack is a linear data structure that follows the Last In First Out (LIFO) principle. This means that the most recently added item is the first one to be removed, just like a stack of plates where you would take the plate from the top. The two basic operations performed on a stack are push and pop. Push refers to adding an element to the top of the stack, and pop refers to removing an element from the top of the stack. The top of the stack is the element most recently added, and it is the next element to be removed. There are many use cases for stacks, such as implementing undo/redo functionality in an application, backtracking algorithms in computer science, evaluating expressions, and more. In computer science, a stack can be implemented as an array or a linked list, but the basic operations remain the same. Stacks are easy to understand and use, and they have a constant time complexity for both push and pop operations, making them very efficient in certain use cases.

Capítulo 3

BINARY SEARCH

Binary search is a search algorithm used to efficiently find an item in a sorted list or array by repeatedly dividing the search interval in half. It works by first comparing the target value to the middle element of the list or array. If the target value matches the middle element, the search is complete and the middle element's index is returned. If the target value is less than the middle element, the search continues in the lower half of the list or array; if it is greater, the search continues in the upper half. The process is repeated until the target value is found or the search interval is reduced to zero, indicating that the target value is not present in the list or array. Binary search is an efficient search algorithm that has a time complexity of O(log n), where n is the number of elements in the list or array.

Capítulo 4

CIRCULAR BUFFER

A circular buffer, also known as a circular queue or ring buffer, is a type of data structure that uses a single, fixed-size buffer as if it were connected end-to-end. The buffer can be thought of as a queue, where elements are added to one end (the tail) and removed from the other end (the head). However, when the head reaches the end of the buffer, it wraps around and starts at the beginning again. In this way, the buffer acts as a continuous sequence of memory locations, so that new elements can be added even if the buffer is full, by overwriting the oldest data.

Circular buffers are used in various applications, such as data streaming, audio and video processing, and real-time communication protocols, where data must be stored temporarily in a buffer before being processed or transmitted. The advantage of circular buffers over linear buffers is that they do not require frequent memory allocation and deallocation, which can be time-consuming and computationally expensive.

A circular buffer can be implemented using an array, where the head and tail pointers are used to keep track of the first and last items in the buffer. The head and tail pointers can be incremented and decremented as necessary, and when they reach the end of the buffer, they can be wrapped around to start at the beginning again. To ensure that the buffer is not overwritten when it is full, a variable called the “count” or “fill level” can be used to keep track of the number of elements in the buffer. When the buffer is full, a new item cannot be added until an item is removed, freeing up space in the buffer.

Capítulo 5

HEAP

A heap is a special type of binary tree data structure in computer science. It is a complete binary tree that satisfies the heap property, where the value of the parent node is either greater than or equal to (for a max heap) or less than or equal to (for a min heap) the values of its children nodes.

In a max heap, the highest value element is stored at the root node. In a min heap, the smallest value element is stored at the root node. The values of the remaining nodes are stored in a way that follows the heap property.

Heaps are often used in priority queue implementation, where elements with high priority are extracted first from the queue. They can also be used for sorting algorithms like heap sort, where the largest or smallest elements can be efficiently extracted from the heap.

Heaps are represented using arrays, where the left and right children of a node are stored at indices 2 * i + 1 and 2 * i + 2, respectively. The parent node of an element stored at index i can be found at index (i - 1) / 2.

Overall, heaps provide efficient methods for inserting, extracting, and manipulating elements in a data structure, making them useful in a variety of applications

Capítulo 6

DEQUE

A deque, short for Double Ended Queue, is a data structure that allows elements to be inserted and removed from both the front and the back of the queue. Unlike a regular queue, where elements can only be inserted at the back and removed from the front, a deque allows for more flexible operations on the data. The elements in a deque are stored in an ordered sequence, and the two ends of the deque are referred to as the "front"and "rear."The front of the deque is where new elements are removed, and the rear is where new elements are inserted. Deques are used in various applications such as scheduling algorithms, in computer networks for buffering data, and in various data processing algorithms where efficient insertions and deletions from both ends of a queue are required.

Capítulo 7

SINGLY LINKED LIST

A singly linked list is a linear data structure in which elements are stored in nodes and each node has a reference to the next node, forming a chain. The first node is known as the head node, and the last node has a reference to null, indicating the end of the list. Unlike an array, a linked list can dynamically grow or shrink in size, making it useful for implementing dynamic data structures. The main advantage of linked lists over arrays is that elements can be easily inserted or removed from the middle of the list without having to shift the elements, as is the case with arrays. However, linked lists have a slower access time compared to arrays, as elements must be searched sequentially starting from the head node.

Capítulo 8

DOUBLY LINKED LIST

A Doubly Linked List is a data structure that consists of a list of nodes, where each node stores an element and two pointers, one pointing to the previous node and one pointing to the next node. The two pointers allow for traversal in both forward and backward direction, hence the name "doubly linked."

The first node in a doubly linked list is known as the head, and the last node is known as the tail. The head node's previous pointer points to NULL, and the tail node's next pointer points to NULL.

A doubly linked list can be useful in various scenarios where data needs to be stored in a linear order and elements need to be added or removed from both ends of the list. Some common operations that can be performed on a doubly linked list include insertion, deletion, and traversal.

One advantage of a doubly linked list over a singly linked list is that it allows for efficient traversal in both forward and backward direction, as well as easy access to the previous element, which is not possible in a singly linked list. However, doubly linked lists also have a higher memory overhead compared to singly linked lists, as each node requires two pointers rather than one.

Capítulo 9

HASH TABLE

A hash table is a data structure that is used to implement an associative array, a structure that can map keys to values. It uses a hash function to compute an index into an array of buckets or slots, from which the desired value can be found. In a hash table, the key-value pairs are stored in an array, and the keys are hashed to find their index in the array.

Hash tables offer an efficient way to look up values based on their keys. When a key is hashed, the hash function produces an index in the array, which is then used to look up the corresponding value. If there are multiple keys that produce the same index, a collision resolution strategy is used to resolve the conflict and determine which value should be stored in that index.

The time complexity of hash table operations depends on the hash function, the size of the array, and the collision resolution strategy used. In the ideal case, hash table operations have an average time complexity of O(1), making hash tables a fast and efficient data structure for a wide variety of use cases, including database indexing, cache implementation, and more.

A hash table is a data structure that provides an associative array abstract data type, a structure that can map keys to values. Hash tables use a hash function to compute an index into an array of buckets or slots, from which the desired value can be found.

There are two main variants of hash tables: open addressing and chaining.

Open addressing: In open addressing, all elements are stored in the same array. If a collision occurs (when two keys are hashed to the same index), the hash table uses a probing sequence to find the next available slot to store the value. The probing sequence can be linear, quadratic, double hashing, etc.

Chaining: In chaining, each element in the array is a linked list of elements with the same hash value. When a collision occurs, the new element is added to the linked list. Chaining is simple to implement, but it can lead to longer linked lists, which can cause slowdowns.

The main difference between open addressing and chaining is how they handle collisions. Open addressing tries to find the next available slot in the array, while chaining stores all elements with the same hash value in a linked list. Open addressing can have better performance when the hash table is well-distributed, but it can also lead to long probing sequences when the hash table is poorly distributed. Chaining, on the other hand, is simple to implement and can handle poorly distributed hash tables, but it can have a performance penalty when the linked lists are long.

Capítulo 10

BINARY TREE

A binary tree is a tree data structure in computer science, used to represent a hierarchical structure, where each node has at most two children, which are referred to as the left child and the right child. The terms "left"and "right"are used to distinguish between children and do not necessarily imply any type of ordering. The root node is the topmost node in the tree and serves as the starting point for traversing the tree.

Each node in a binary tree has a value associated with it, and the value of the child nodes must be different from their parent node. The left child node contains a value that is less than or equal to its parent node, and the right child node contains a value that is greater than its parent node. This relationship forms a binary search tree, which is a special type of binary tree used to store data in a way that enables efficient searching and sorting operations.

A leaf node is a node that does not have any children. The height of a binary tree is the length of the longest path from the root to a leaf node. A balanced binary tree is a tree where the height of the left and right subtrees of every node differ by no more than one.

Binary trees have several use cases, such as in searching and sorting algorithms, expression trees, and decision trees in machine learning.

Capítulo 11

PRIORITY QUEUE

A priority queue is a data structure in which each element has a priority assigned to it and the elements are stored in such a way that the element with the highest priority is always at the front and can be retrieved before the other elements. The order of elements with the same priority can be determined by additional rules such as the order in which they were added to the queue.

In computer science, a priority queue is often implemented as a binary heap, a binary search tree or a data structure that implements the priority queue abstract data type. In a binary heap, elements can be added to the heap in any order, but when it comes to removing elements, the highest priority element is always removed first. In a binary search tree, elements are sorted by priority, and removing an element always removes the highest priority element.

Priority queues are used in many algorithms and applications such as process scheduling, graph algorithms, and data compression. They are also used in simulation and modeling, where events with different priorities must be processed in order.

Capítulo 12

TRIES

Trie (also known as Prefix Tree) is a tree-like data structure used for efficient querying and searching of sequences (such as strings or arrays). It stores the sequences in a way that allows for fast prefix lookups, making it ideal for tasks such as searching for words in a large dictionary or searching for substrings within a string.

Each node in the Trie represents a single character in the sequence and the path from the root node to a leaf node represents a complete sequence. Each node has multiple children, one for each character in the alphabet (or set of characters used in the sequences being stored). The child nodes represent possible extensions of the sequence represented by the parent node.

The Trie stores the end of a complete sequence by marking the end of the sequence at the leaf node for that sequence. This allows for efficient lookups for complete sequences as well as prefixes of sequences.

Tries have efficient time complexity for common operations, such as insertion, deletion, and lookup, with an average time complexity of O(k) where k is the length of the key (sequence) being looked up. This makes them particularly useful for large datasets and for tasks that require quick searching and retrieval of data.

Capítulo 13

BUBBLE SORT

Bubble sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order. The algorithm gets its name from the way smaller elements "bubble"to the top as the sort is performed.

Bubble sort works by repeatedly swapping adjacent elements that are out of order, until the list is sorted. The algorithm starts at the beginning of the list and compares the first two elements. If the first element is larger than the second, the two elements are swapped. The algorithm then moves to the next pair and repeats the process. This continues until the end of the list is reached. At this point, the largest element will have "bubbled"to the end of the list. The process is then repeated, ignoring the final element, until the list is completely sorted.

The best-case scenario for Bubble Sort occurs when the input list is already sorted, in which case no swaps are needed and the algorithm runs in O(n) time.

The worst-case scenario occurs when the input list is sorted in the reverse order, in which case the algorithm will have to repeatedly compare and swap elements until the entire list is sorted. This results in $O(n^2)$ time complexity.

The average-case scenario is also $O(n^2)$ time complexity, making Bubble Sort not the most efficient sorting algorithm for large data sets. However, it is a simple algorithm to understand and implement, making it useful for small data sets or as a demonstration of basic sorting concepts.

Capítulo 14

HEAP SORT

Heap sort is a comparison-based sorting algorithm that uses a data structure called a binary heap, which is a complete binary tree, to sort elements. A binary heap can either be a max heap or a min heap, and the sorting algorithm will be based on the type of heap used.

In a max heap, the parent node is always greater than or equal to its children. In a min heap, the parent node is always smaller than or equal to its children.

The steps of the heap sort algorithm are as follows:

Build a binary heap from the input data.

Extract the maximum or minimum element from the binary heap and place it at the end of the sorted array.

Repeat step 2 until all elements have been extracted from the binary heap.

The best case scenario for heap sort is when the input data is already sorted in the desired order. In this case, the binary heap can be constructed in linear time, and the extraction of the maximum or minimum element can be done in constant time. However, since the extraction of the maximum or minimum element requires the adjustment of the binary heap, the sorting process takes O(nlogn) time in the best case.

The worst case scenario for heap sort occurs when the input data is in the reverse order of the desired sort. In this case, the binary heap will require the maximum number of comparisons to be constructed, and the extraction of the maximum or minimum element will require the maximum number of adjustments to be made. The sorting process will take O(nlogn) time in the worst case.

The average case scenario for heap sort is O(nlogn) time, which is the same as the best and worst case scenario. This is because the number of comparisons and adjustments required to build the binary heap and extract the maximum or minimum element is proportional to the logarithm of the number of elements in the input data.

Capítulo 15

INSERTION SORT

Insertion sort is a simple sorting algorithm that sorts an array by repeatedly removing one element and inserting it into the correct position in a sorted sub-list. It is efficient for small data sets and is also often used as a building block for more complex algorithms like merge sort.

The best case scenario for insertion sort occurs when the array is already sorted, and each element is inserted into its proper position in the array in one operation. In this case, the time complexity of insertion sort is O(n), where n is the number of elements in the array.

The worst case scenario for insertion sort occurs when the array is sorted in the reverse order, and each element requires shifting all elements to the right of it to make room for the insertion. In this case, the time complexity of insertion sort is $O(n^2)$, where n is the number of elements in the array.

The average case scenario for insertion sort occurs when the array is randomly ordered. In this case, the time complexity of insertion sort is $O(n^2)$, where n is the number of elements in the array.

Capítulo 16

MERGE SORT

Merge sort is a divide-and-conquer based sorting algorithm. It works by dividing the unsorted list into n sublists, each containing one element, and then repeatedly merging sublists to produce new sorted sublists until there is only one sublist remaining.

The best case scenario of Merge sort occurs when the list is already sorted. In this case, the algorithm will divide the list into two sublists, compare each element and combine the two sublists into a single sorted list. This will result in a linear time complexity of O(n).

The worst case scenario of Merge sort occurs when the list is in reverse order. In this case, the algorithm will divide the list into n sublists and repeatedly merge the sublists. This will result in a time complexity of O(n log n), which is the average time complexity for Merge sort.

The average case scenario of Merge sort occurs when the elements in the list are randomly ordered. In this case, the algorithm will divide the list into n sublists and repeatedly merge the sublists until the list is sorted. This will result in a time complexity of O(n log n), which is the average time complexity for Merge sort.

Merge sort is considered a stable sorting algorithm, which means that it preserves the relative order of equal elements in the sorted list. It is also considered an efficient sorting algorithm for large datasets, as it has a guaranteed time complexity of O(n log n). Additionally, Merge sort is a recursive algorithm that requires extra space to store the sublists during the sorting process. This extra space can be a disadvantage if memory is limited. However, the algorithm is efficient in terms of cache utilization, as it requires only a small amount of extra memory. In terms of complexity, Merge sort is generally slower than Quick sort, but it is more consistent and stable. It is also more efficient than Insertion sort and Bubble sort for larger datasets. Overall, Merge sort is a well-balanced sorting algorithm that is suitable for a wide range of applications.

Capítulo 17

QUICK SORT

Quick sort is a divide-and-conquer based sorting algorithm. It works by selecting a pivot element from the list and partitioning the other elements into two sub-lists, according to whether they are less than or greater than the pivot. The sub-lists are then sorted recursively.

The best case scenario for Quick sort occurs when the pivot is chosen such that the sub-lists are evenly balanced. This results in a time complexity of O(n log n).

The worst case scenario for Quick sort occurs when the pivot is chosen as the largest or smallest element in the list. This results in one sub-list being empty and the other containing n-1 elements. In this case, the time complexity is O(n^2).

The average case scenario for Quick sort occurs when the pivot is chosen randomly. This results in a time complexity of O(n log n).

Quick sort is considered an efficient sorting algorithm for large datasets, as it has a good average case time complexity of O(n log n). However, it can be less efficient than other sorting algorithms in the worst case scenario. Quick sort is also an in-place sorting algorithm, meaning it only requires a constant amount of extra memory to sort the list.

In conclusion, QuickSort is a highly efficient and reliable sorting algorithm with an average time complexity of O(n log n), but its performance is highly dependent on the choice of pivot element.

Capítulo 18

SELECTION SORT

Selection sort is a simple sorting algorithm that works by repeatedly finding the minimum element from the unsorted part of the array and swapping it with the first unsorted element.

The algorithm works as follows:

.Start with the first element and find the smallest element in the array.

.Swap the smallest element with the first element.

.Repeat the above steps for the remaining unsorted part of the array.

The best case scenario for selection sort occurs when the list is already sorted in ascending order. In this case, the algorithm will only make n-1 comparisons to sort the list, resulting in a time complexity of $O(n^2)$.

The worst case scenario for selection sort occurs when the list is sorted in reverse order. In this case, the algorithm will make n(n-1)/2 comparisons, resulting in a time complexity of O(n^2).

The average case scenario for selection sort occurs when the elements in the list are randomly ordered. In this case, the algorithm will make n(n-1)/2 comparisons, resulting in a time complexity of $O(n^2)$.

Selection sort is considered an inefficient sorting algorithm for large datasets, as it has a time complexity of $O(n^2)$. It is also not a stable sorting algorithm, meaning that it does not preserve the relative order of equal elements in the sorted list.