Exam 2 Notes

Heaps: Min Heaps and Max Heaps

Heaps are a specialized tree-based data structure that satisfy the heap property. In a Min Heap, for any given node, the value of the node is less than or equal to the values of its children. Conversely, in a Max Heap, for any given node, the value of the node is greater than or equal to the values of its children. Heaps are commonly used to implement priority queues and for efficient sorting (Heap Sort).

Heap Representation

Heaps are often represented as arrays for efficiency, with the relationships between parent nodes and children nodes implicitly defined by their indices in the array.

Node Relationships in an Array Representation

Given a node at index i in the array:

- The index of the left child is $2i + 1$.
- The index of the right child is $2i + 2$.
- The index of the parent node is $\lfloor (i - 1)/2 \rfloor$, for any node except the root.

Operations and Their Complexities

1. **Insertion**: Inserting a new element into a heap involves adding the element to the end of the array and then adjusting its position to maintain the heap property. This adjustment, or "heapifying up," has a time complexity of $\mathcal{O}(\log n)$.
2. **Deletion of Root**: Removing the root element (the minimum element in a Min Heap or the maximum element in a Max Heap) involves moving the last element in the array to the root position and then "heapifying down" to maintain the heap property. This operation also has a time complexity of $\mathcal{O}(\log n)$.
3. **Find Min/Max**: In a Min Heap or Max Heap, finding the minimum or maximum value, respectively, is a constant-time operation, $\mathcal{O}(1)$, as this value is always at the root of the heap.

Quick Sort

Quick Sort is a highly efficient sorting algorithm that utilizes the divide-and-conquer strategy. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively. This process is repeated until the whole array is sorted.

Steps

1. **Pivot Selection**: The first step is to select a pivot element. There are multiple strategies for this, such as choosing the first element, the last element, the middle element, or even a random element from the array.
2. **Partitioning**: Rearrange the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it. After this step, the pivot is in its final position. This is called the partition operation.
3. **Recursion**: Recursively apply the above steps to the sub-array of elements with smaller values and the sub-array of elements with larger values.

Quick Sort

An example of the Quick Sort Algorithm can be seen in Python below

```python
def quick_sort(arr):
if len(arr) <= 1:
    return arr
else:
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + middle + quick_sort(right)
```

```
10
```

The above implementation is selecting the pivot element as the middle element of the given array.

Runtime Analysis

The runtime analysis of Quick Sort is as follows:

- **Best and Average Case**: The average and best-case performance of quick sort is $\mathcal{O}(n \log(n))$, where n is the number of elements in the array. This occurs when the pivot divides the array into two nearly equal halves, leading to a balanced tree.

- **Worst Case**: The worst-case scenario occurs when the pivot selection results in one partition being significantly smaller than the other, e.g., when the smallest or largest element is always chosen as the pivot. This leads to an unbalanced partition and gives a runtime of $\mathcal{O}(n^2)$. However, this can be mitigated by choosing a good pivot.

Quick Select

Quick Select is an efficient selection algorithm to find the k-th smallest element in an unordered list. It is closely related to the Quick Sort sorting algorithm, utilizing a similar partitioning approach. The key difference is that Quick Select only recurses into the part of the array that contains the k-th smallest element, reducing the average time complexity.

Steps

1. **Choose Pivot**: Select a pivot element from the array. The choice of pivot can be random or based on the Median of Medians strategy to ensure good average performance and avoid worst-case scenarios.

2. **Partition**: Rearrange the array such that all elements less than the pivot come before the pivot, while all elements greater than the pivot come after it. The pivot is then in its final position, with a certain number of elements preceding it.

3. **Target Check**: If the pivot's position is k, the pivot is the k-th smallest element, and the algorithm terminates. If the pivot's position is greater than k, repeat the algorithm on the sub-array of elements before the pivot. If the pivot's position is less than k, repeat on the sub-array of elements after the pivot, adjusting k accordingly.

Runtime Analysis

The average time complexity of Quick Select is $\mathcal{O}(n)$, making it highly efficient for selection problems. This efficiency stems from the fact that it only needs to process one partition of the array, unlike Quick Sort, which must sort both partitions.

- **Best and Average Case**: In the average and best-case scenarios, the partitioning divides the array into parts that diminish in size exponentially, leading to an overall linear time complexity.

- **Worst Case**: Without the Median of Medians strategy, the worst-case time complexity can degrade to $\mathcal{O}(n^2)$, particularly if the smallest or largest element is consistently chosen as the pivot. However, this is mitigated by intelligent pivot selection.

Median Of Medians

The Median of Medians is a selection algorithm that serves as a trick to find a good pivot for sorting and selection algorithms, such as Quick Sort, or for solving the selection problem (finding the k-th smallest element) in linear time. The algorithm is particularly useful because it guarantees a pivot that is a 'good' approximation of the median, ensuring that the partition of the array is reasonably balanced, which helps avoid the worst-case scenario of $\mathcal{O}(n^2)$ time complexity in Quick Sort or other selection algorithms.

Steps

1. **Grouping**: Divide the array into subarrays of n elements each, where n could be any small constant. The last group may have fewer than n elements if the size of the array is not a multiple of n.

2. **Find Medians**: Compute the median of each of these subarrays. If n is small, this can be done quickly through a brute-force approach.

3. **Recursive Median Selection**: Use the Median of Medians algorithm recursively to find the median of these n-element medians. This median will be used as the pivot.

4. **Partition Using the Pivot**: The chosen pivot divides the original array into parts such that one part contains elements less than the pivot, and the other part contains elements greater than the pivot.

5. **Recursive Application for Selection/Sorting**: Depending on whether you're sorting or selecting (e.g., finding the k-th smallest element), proceed with the algorithm recursively on the relevant partition(s).

Runtime Analysis

The choice of n (the size of the groups) affects the constants in the time complexity but not the overall linear time complexity for the selection. The analysis hinges on two key observations:

- At least half of the medians are greater than (or equal to) the median-of-medians, and at least half are less than (or equal to) it.

- Because each median is greater (or smaller) than at least half the elements in its group, the pivot (median-of-medians) effectively guarantees that at least 1/4 of the elements are less than it and at least 1/4 are greater, ensuring a balanced split.

This balancing ensures that the algorithm makes significant progress in reducing the problem size at each step, leading to a linear time complexity, $\mathcal{O}(n)$, for the selection problem.

Binary Search Trees (BST)

A Binary Search Tree (BST) is a node-based binary tree data structure with the following essential properties:

- Each node has at most two children, referred to as the left child and the right child.

- For any given node, all elements in the left subtree are less than the node, and all elements in the right subtree are greater than the node. This property is recursively true for all nodes in the tree.

These characteristics enable efficient performance of operations such as search, insertion, and deletion, with average and best-case time complexities of $\mathcal{O}(\log n)$, where n is the number of nodes in the tree.

Height of a Binary Search Tree

The height of a BST is measured as the number of edges on the longest path from the root node to a leaf node. It is a critical metric that influences the efficiency of various tree operations.

Height of a Node in a Binary Search Tree

The height of a node within a BST is determined by the number of edges on the longest path from that node to a leaf node in its subtree. The calculation follows the same recursive logic as the height of the entire tree, but starts from the specific node in question.

Red-Black Trees

Red-Black Trees are a type of self-balancing binary search tree, where each node contains an extra bit for denoting the color of the node, either red or black. This structure ensures the tree remains approximately balanced, leading to improved performance for search, insertion, and deletion operations. The properties of Red-Black Trees enforce constraints on node colors to maintain balance and ensure operational complexity remains logarithmic.

Properties of Red-Black Trees

Red-Black Trees adhere to the following five essential properties:

1. **Node Color**: Every node is either red or black.
2. **Root Property**: The root node is always black.
3. **Red Node Property**: Red nodes must have black parent and black children nodes (i.e., no two red nodes can be adjacent).
4. **Black Height Property**: Every path from a node (including root) to any of its descendant NULL nodes must have the same number of black nodes. This consistent number is called the black height of the node.
5. **Path Property**: For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

These properties ensure that the longest path from the root to a leaf is no more than twice as long as the shortest path, meaning the tree remains approximately balanced.

Operations Complexity

Due to the self-balancing nature of Red-Black Trees, operations such as insertion, deletion, and lookup can be performed in $\mathcal{O}(\log n)$ time complexity, where n is the number of nodes in the tree.

Hash Tables

Hashtables, also known as hash tables, are a type of data structure that implements an associative array, a structure that can map keys to values. A hashtable uses a hash function to compute an index into an array of slots, from which the desired value can be found. This approach enables efficient data retrieval, insertion, and deletion operations.

Key Components

- **Hash Function**: A function that computes an index (the hash) into an array of buckets or slots, from which the desired value can be found. The efficiency of a hash table depends significantly on the hash function it uses.
- **Buckets or Slots**: The array where the actual data is stored. Each slot can store one or more key-value pairs.
- **Collision Resolution**: Since a hash function may assign the same hash for two different keys (a collision), mechanisms such as chaining (linked lists) or open addressing (probing) are used to resolve collisions.

Operations

1. **Insertion**: Add a new key-value pair to the table.
2. **Deletion**: Remove a key-value pair from the table.
3. **Lookup**: Retrieve the value associated with a given key.

The average-case time complexity for these operations is $\mathcal{O}(1)$, assuming a good hash function and low load factor, making hashtables one of the most efficient data structure for these types of operations.

Universal Hash Functions

Universal hash functions are designed to minimize the probability of collisions between keys being hashed into the same slot or bucket. A family of hash functions $\mathcal{H}$ is considered universal if, for any two distinct keys k and l, the probability of a collision is at most $1/m$, where m is the number of slots in the hashtable.

Properties

- **Reduced Collisions**: By randomly choosing a hash function from a universal family at the beginning of execution, universal hash functions help distribute keys more uniformly across the buckets, reducing the chance of collisions.
- **Performance**: Universal hash functions ensure that the average-case time complexity of hashtable operations remains constant, $\mathcal{O}(1)$.

Runtime Complexity

The efficiency of hash table operations—namely insertion, deletion, and lookup—is a critical aspect of their performance. These operations ideally have a time complexity of $\mathcal{O}(1)$ on average. However, this efficiency can be affected by several factors, including the choice of hash function, the method of collision resolution, and the load factor of the hash table. Here we explore these factors in detail.

Factors Affecting Runtime Complexity

1. **Hash Function**: The hash function's role is to distribute keys uniformly across the buckets. A poor hash function can lead to clustering, where many keys hash to the same index, increasing the likelihood of collisions and thereby the time complexity of operations.
2. **Collision Resolution Strategy**: How collisions are handled significantly impacts performance. Two primary methods are:
    - **Chaining**: Each bucket at a given index stores a linked list of all elements that hash to the same index. While insertion remains $\mathcal{O}(1)$, deletion and lookup operations can degrade to $\mathcal{O}(n)$ in the worst-case scenario when all elements collide at the same index.
    - **Open Addressing**: All elements are stored within the array itself. If a collision occurs, the hash table probes for the next available slot according to a probing sequence. The worst-case time complexity can also degrade to $\mathcal{O}(n)$, particularly in a highly congested table.
3. **Load Factor** (α): Defined as $\alpha = \frac{n}{m}$, where n is the number of entries and m is the number of buckets. A higher load factor indicates a more filled table, increasing the likelihood of collisions and the costs associated with collision resolution, thus affecting the average-case complexity.

Average-case Time Complexity

Assuming a well-designed hash function and a low to moderate load factor, the average-case time complexity for insertion, deletion, and lookup operations in a hash table is $\mathcal{O}(1)$. This assumes that the collision resolution process is efficient and that the hash function distributes keys uniformly.

Worst-case Time Complexity

In the worst-case scenario, particularly with a poor hash function or high load factor, all keys could hash to the same index, leading to a situation where each operation takes $\mathcal{O}(n)$ time. This scenario is typically mitigated by ensuring a good hash function and keeping the load factor manageable through techniques like dynamic resizing.

Mitigation Strategies

- **Dynamic Resizing**: To maintain a low load factor, hash tables often increase their size and re-hash all entries when a certain load threshold is exceeded.
- **Choosing an Efficient Collision Resolution Method**: Depending on the use case, selecting between chaining and open addressing can optimize performance.
- **Using Universal Hash Functions**: These functions reduce the probability of collisions and help maintain the $\mathcal{O}(1)$ average-case time complexity.