Course: MsDS

Data Structures and Algorithms

Module: **5**

Learning Objectives:

1. Understand the fundamental principles and uses of hash tables for effective data storage and retrieval.
2. Understand the differences in the features and applications of HashSets and HashMaps.
3. Understand the fundamental ideas behind Binary Trees and Binary Search Trees, as well as their importance in the representation of hierarchical data.
4. Understand the nuances of height-balanced BSTs, especially AVL Trees, and appreciate how crucial balancing is to achieving peak performance.

Structure:

5.1 HashTables

HashTables, also known as hash maps, are essential data structures that enable efficient storage and retrieval of data. The core component of a HashTable is an array; however, unlike conventional arrays that use indices for value retrieval, HashTables utilise a distinct process referred to as "hashing."

The hashing process entails receiving an input, often referred to as a "key," and generating an output of a certain size. This output is then used as an index to determine the location in the array where the corresponding item is kept. When the need arises to access this information, it may be accomplished by rehashing the key, providing a direct path to the area where the needed data is stored. The use of direct addressing, which is based on the hashed key, enables search operations to have an average complexity of constant time. This characteristic is one of the primary benefits offered by HashTables.

Nevertheless, due to the possibility of various keys being hashed to the same index, HashTables are required to effectively manage collisions. A collision occurs when several keys generate the same hash index. Different techniques, such as chaining (in which each element in the array is transformed into the starting point of a linked list containing keys that hash to the same index) and open addressing (in which we search for the next available slot in the array), may be used to solve these instances of collisions.

HashTables provide a compromise between the efficiency of direct array indexing and the adaptability of linked lists. This feature makes them very advantageous in situations that need prompt data retrievals, such as in databases or caching systems. However, it is crucial to use a proficient hashing function in order to guarantee a consistent distribution and proficiently manage collisions, so ensuring the HashTable retains its effectiveness.

5.2 HashMaps and HashSet

5.2.1 HashMaps

The HashMap is a widely used and adaptable data structure in the field of computer science. It facilitates the association of keys with corresponding values, hence enabling efficient data retrieval. HashMaps, similar to HashTables, use a hashing algorithm to ascertain the appropriate storage location within the underlying array for a given data. Nevertheless, while HashTables are a more versatile data structure, HashMaps are particularly designed to efficiently manage key-value pairs.

The appeal of a HashMap resides in its straightforwardness and effectiveness. Consider the scenario where an individual has an extensive assortment of books and necessitates an expeditious and efficient technique to promptly retrieve any given book by using the author's name as a search criterion. Within the context of this comparison, the appellation of the author serves as the primary identifier, akin to a key, while the book in question represents the corresponding value. The assignment of a book to a certain location or "shelf" inside a library is determined by applying a hashing algorithm to the author's name. When the book is required at a later time, restating the author's name will expedite the process of locating the correct shelf, eliminating the need to individually scan each shelf.

One salient characteristic of HashMaps is their inherent dynamism. In contrast to arrays, which possess a predetermined size upon initialisation, HashMaps possess the ability to dynamically adjust their size as they expand, often doubling in magnitude and afterwards redistributing all preexisting key-value pairs inside the newly allocated memory. The process of dynamic resizing, while it incurs a significant amount of cost during the resizing procedure, guarantees that the HashMap maintains its efficiency in terms of space utilisation and access time when more components are included.

Nevertheless, every data structure is not without its complexities. HashMaps, similar to HashTables, encounter collisions when two distinct keys result in the same array index after hashing. Techniques such as chaining and probing are used to effectively tackle this issue. Furthermore, it is essential to carefully choose a proficient hashing function that is specifically designed for the given data collection in order to guarantee a uniform distribution of data.

HashMaps provide a sophisticated resolution for situations necessitating efficient data retrieval based on distinct keys. The efficiency and versatility of these tools make them very effective in a wide range of applications, spanning from databases to web caches.

5.2.2 HashSet

Within the realm of data structures, the HashSet emerges as a notable and proficient instrument for the purpose of managing collections devoid of any duplicate elements. The HashSet data structure is based on the underlying principles of HashTables and utilises hashing techniques for efficient element management. In contrast, whereas HashMaps establish associations between keys and values, HashSets prioritise the presence of distinct keys or, more accurately, unique components.

The fundamental mechanism of a HashSet involves taking an element, subjecting it to a hashing algorithm, and then storing the resultant value at a specific position defined by the hash. This procedure not only guarantees efficient insertion of items but also facilitates quick lookups and removals. When a query is made about the existence of a certain element inside the set, the hashing function is once again used to promptly ascertain the position of the element, hence bypassing the need for protracted searches.

The distinguishing characteristic of a HashSet is its uniqueness requirement. Consider the scenario when one wants to create a comprehensive inventory of unique names from an extensive register. In this particular instance, a HashSet proves to be really beneficial. The HashSet data structure efficiently identifies and disregards duplicates when new names are appended. The inherent capacity to prevent the existence of duplicates is crucial for activities that involve the management of a collection of distinct things, whether it is in the context of databases, data analytics, or basic list operations.

Although HashSets provide advantages in terms of speed and uniqueness, they are not exempt from encountering problems. The occurrence of collisions, which is a prevalent concept in structures based on hashing, has comparable significance in this context. In situations when two unique items provide identical hash values, it becomes necessary for the system to address this conflict. This is often accomplished by the use of conflict resolution techniques such as chaining.

In conclusion, HashSets have established themselves as effective solutions for the management of unique collections. The simplicity of their design, coupled with the robust functionality of hashing, guarantees their capacity to provide both efficiency and dependability, making them essential in several computing contexts.

5.3 Binary Trees & Binary Search Trees

5.3.1 Binary Trees

Binary trees have a significant role within the realm of data structures. The binary tree is a hierarchical data structure whereby each element, known as a "node," may have a maximum of two offspring, often denoted as the left child and the right child. The first node, which serves as the starting point of the tree, is referred to as the "root".

The appeal of binary trees stems from their adaptability and effectiveness. The inherent design of these structures facilitates efficient operations involving the insertion, deletion, and retrieval of data, therefore establishing a fundamental role in a wide range of applications, including databases and dynamic memory management. Moreover, the recursive characteristic of these data structures facilitates the development of streamlined and sophisticated algorithmic approaches to tasks such as tree traversal, which involves the systematic visitation of each node in a specified sequence.

A significant form of the binary tree is referred to as the "balanced binary tree," which is characterised by minimum differences in depth between the left and right subtrees of all nodes. This characteristic guarantees optimum efficiency for various operations. The depth of a tree may be defined as the count of edges traversed from the root node to the furthest node inside the tree structure. The maintenance of balance is crucial in situations where the entry of data may occur in a sequential or clustered manner since this has the potential to result in imbalanced trees and thus reduce their operational effectiveness.

Nevertheless, it is important to acknowledge that while binary trees include certain benefits, they do not serve as a universal solution for all data management issues. The effectiveness of their design and performance is heavily dependent on their utilisation and the particular contexts in which they are implemented. In some

scenarios that include the requirement for data sorting, the binary search tree (BST), which is a specific variant of binary trees, is seen as more suitable.
Binary trees are widely recognised as a basic and useful technique within the field of computer science. Due to their hierarchical architecture and efficient data management capabilities, they have become a fundamental component of several sophisticated data structures and algorithms, highlighting their profound importance in contemporary computing applications.

5.3.2 Binary Search Trees

Binary Search Trees (BSTs) are a more advanced version of the conventional binary tree, providing improved efficiency and organisation of data. In a binary search tree (BST), each node follows a fundamental property: the value stored in its left child is consistently less than its own value, while the value stored in its right child is consistently larger. The structured arrangement described herein offers a user-friendly approach for storing data in a way that optimises efficiency in search operations, as well as the insertion and deletion of data.
The simplicity of search operations is a key characteristic that contributes to the brilliance of Binary Search Trees (BSTs). When doing a search for a value, the process typically begins at the root node and proceeds by traversing either to the left or right, depending upon whether the desired value is lesser or greater than the value of the current node. The aforementioned procedure is executed iteratively until the desired value is discovered or until a terminal node is encountered, signifying the nonexistence of the value. As a result of the streamlined route, the time complexity of operations in a Binary Search Tree (BST) is often proportional to the height of the tree.
Nevertheless, the beauty of binary search trees (BSTs) is not without its problems. The effectiveness of a binary search tree (BST) may be impaired when it becomes skewed or imbalanced, resulting in one subtree containing a much larger number of nodes than the other. In situations when data is entered into a binary search tree (BST) in a sorted manner, the BST may deteriorate into a linked list, resulting in operations that require linear time complexity rather than logarithmic time

complexity. Therefore, in order to mitigate this vulnerability, balancing approaches such as AVL or Red-Black trees have been devised.

Binary search trees (BSTs) are often used in practical scenarios, particularly in the field of database indexing. The efficiency of finding, adding, or deleting entries becomes a critical factor in managing huge databases, and Binary Search Trees (BSTs), particularly their self-balancing variations, provide useful answers.

Binary Search Trees (BSTs) provide an organised and efficient approach for storing and retrieving data. However, it is crucial to ensure the balance of BSTs in order to fully use their capabilities. The significance of their contribution to improving search performance in many applications underscores their crucial position within the domain of data structures.

5.4 Height-balanced BSTs – AVL Trees

In the domain of data structures, prioritising efficiency is of utmost importance. Although Binary Search Trees (BSTs) provide efficient operations, their performance may be severely affected by the issue of being imbalanced. Height-balanced binary search trees, such as AVL Trees, are a viable solution in this context. They guarantee that the tree maintains its balance after each insertion or deletion, hence preserving logarithmic time complexity for various operations.

The AVL tree, a self-balancing binary search tree, is named after its creators, Adelson-Velsky and Landis. One notable attribute of an AVL tree is its balancing factor, which is determined by the heights of the left and right subtrees of a given node. In the given tree, it is guaranteed that the balancing factor, which represents the difference in height between the left and right subtrees for each node, falls between the range of -1 to 1 in the event that an operation results in the factor exceeding or falling below the specified range, the tree undergoes a rebalancing process to restore its equilibrium.

The process of achieving rebalancing in AVL trees is accomplished by the use of rotations. There are four distinct kinds of rotations that have been defined: single right rotation (LL), single left rotation (RR), right-left rotation (RL), and left-right rotation (LR). The required rotation is calculated based on the identification of the imbalanced kid (either left or right) and the location of the imbalance (either on the left or right child).

AVL Trees have the potential to significantly impact real circumstances. In the context of database systems, the importance of efficient search operations cannot be overstated. The use of an AVL tree guarantees constant efficiency in search operations, regardless of the growth and changes in data, by effectively managing data insertions and deletions.

Although Binary Search Trees (BSTs) provide a basis for performing search operations efficiently, the potential for skewness may be a barrier. AVL Trees, due to their inherent self-balancing mechanism, effectively address this constraint by maintaining optimum balance inside the tree structure. As a result, they provide a well-balanced and efficient approach to data retrieval operations.

5.5 Priority Queues

In typical situations, activities vary in their level of relevance. Certain tasks need urgent attention, whilst others may be deferred. The aforementioned notion finds a fitting illustration in the field of computer science, namely in the use of a priority queue system. In contrast to conventional queues that adhere to a "first in, first out" principle, priority queues work by assigning priority levels to individual elements.

The priority queue is an abstract data structure that facilitates the insertion of items together with their corresponding priorities, as well as the deletion of the element with the greatest priority. An often-seen real-world parallel may be drawn from the context of a hospital emergency department. Patients with higher acuity injuries are prioritised above those with lower acuity injuries, irrespective of their time of arrival.

The implementation of priority queues may be accomplished by many methods, such as using arrays, linked lists, or, with greater efficiency, employing data structures such as heaps. Heaps, particularly binary heaps, facilitate the execution of operations such as insertion and deletion (or enqueue and dequeue) with a temporal complexity of logarithmic order. There are two main categories of binary heaps, namely max-heaps and min-heaps, which determine the order in which elements with the greatest or lowest priority are processed first.

Priority queues have been used in several fields. Computer algorithms rely heavily on the utilisation of data structures, as seen in prominent algorithms such as Dijkstra's shortest route and Huffman encoding. Operating systems play a crucial role in task management by prioritising their execution, so assuring the prompt completion of vital activities. Furthermore, within the context of simulation systems, priority queues play a crucial role in establishing the order of occurrences.

However, it is crucial to meticulously establish the criteria for prioritisation since the effectiveness of operations is directly impacted by the allocation and administration of priorities. Inadequate management may result in instances of famine when some components remain unserved due to their continually low prioritisation.

Priority queues provide an advanced mechanism for organising data according to its relative importance rather than only relying on sequential ordering. The effectiveness of this technology in addressing important concerns swiftly is seen in its applications within the field of computer science and other domains.

5.6 Heaps

Within the domain of data structures, the heap emerges as a distinctive tree-based construct that fulfils a certain criterion, guaranteeing a hierarchical arrangement of its constituent pieces. In contrast to the mental association that the term "heap" may evoke of a haphazard accumulation, within the realm of computer science, a heap is characterised by rigorous organisation, hence facilitating the execution of operations in an efficient manner.

The fundamental principle underlying heaps is the heap property. In a maximum heap, it can be seen that the value of any given node I is larger than or equal to the values of its descendants. On the contrary, in a min heap, the value of I is less than or equal to its offspring. This characteristic guarantees that the heap is capable of efficiently determining the largest or smallest element, which is located at the root.

The effectiveness of heaps in essential tasks is one of the key reasons why they are highly regarded in the field of computer science. The operations of inserting a new element, removing the maximum or minimum, which is responsible for restoring the heap property, all have a temporal complexity of logarithmic order. The use of this efficiency is shown in widely-used algorithms, with heap sort serving as a prominent example.

Heaps are often represented as binary trees. However, they are not necessarily limited to rigidly adhering to the binary structure. Ternary heaps, quaternary heaps, and other variations of heaps are known to exist. Nevertheless, binary heaps continue to be widely favoured in academic and practical contexts owing to their ability to strike an ideal balance between tree height and branching factor.

One prominent use of heaps is seen in the implementation of priority queues. The heap's efficient capability to swiftly access and delete the element with the greatest or lowest priority makes it a suitable selection for such a data structure.

Heaps provide a combination of structural stability and operating effectiveness. The systematic organisation of heaps is ensured by their adherence to the heap property, which also makes them useful for a variety of applications, including sorting algorithms and job scheduling. The persistent use of these concepts in contemporary computer science serves as evidence of their lasting importance.

5.7 Summary

- ❖ HashTables are widely used in various data retrieval structures due to their efficiency. They employ distinct keys to establish a direct correspondence with certain values. HashTables primarily emphasise the association between key-value pairs, but HashMaps expand upon this notion by allowing the inclusion of null values and keys. In contrast, the HashSet data structure also employs a hashing method, but its primary objective is to maintain a collection of distinct values without any related keys. This enables the HashSet to guarantee the uniqueness of its data and provides efficient operations such as adding, removing, and checking for the existence of an item in constant time.
- ❖ Binary trees are data structures that depict hierarchies, where each node may have a maximum of two offspring. Binary Search Trees (BSTs) are an enhanced data structure that guarantees the ordering of nodes, where all nodes in the left subtree have values that are less than the root, and all nodes in the right subtree have values that are larger than the root. In scenarios that require the use of

self-balancing trees, Height-balanced Binary Search Trees (BSTs), particularly AVL Trees, are used. The balancing of the tree is automatically performed after each addition or deletion operation, hence enhancing the efficiency of search operations.

- ❖ Priority queues are a kind of data structure that is designed to handle components by considering their given priority. Efficient access or removal of the element with the greatest or lowest priority may be achieved. Heaps, particularly binary heaps, are often used as the fundamental structure for these queues due to their capacity to uphold the heap property and facilitate efficient operations.
- ❖ The module refers to AVL trees and other advanced data structures. AVL Trees, coined in honour of its creators Adelson-Velsky and Landis, are a kind of self-balancing binary search trees that maintains a maximum height difference ofone between their left and right subtrees for every node. By ensuring logarithmic boundaries on all operations, efficiency is enhanced in circumstances that need frequent updates.

5.8 Keywords

- **HashTables:** Hash-based data structures are used to store key-value pairs by using a hash function to calculate an index inside an array.
- **HashSet:** The set is a data structure that consists of distinct elements without explicit identifiers and is based on the underlying concepts of HashTables.
- **Binary Trees:** Hierarchical structures, characterised by nodes possessing a maximum of two offspring, are often used for the purposes of data storage and retrieval.
- **Binary Search Trees (BSTs):** A distinct variant of binary trees is characterised by the presence of a value assigned to each node, with the property that all nodes in the left subtree possess values that are less than the value of the root node, while all nodes in the right subtree possess values that are larger.
- **AVL Trees:** Self-balancing binary search trees (BSTs) are data structures that preserve balance by guaranteeing that the height difference between the left and right subtrees is no more than one.

- **Heaps:** Tree-based data structures that possess the heap property and are specifically designed to meet the requirements of priority queues.

5.9 Self-Assessment Questions

1. Describe the guiding idea that makes a HashTable effective in retrieving data. What crucial significance does the hash function play in this?
2. Compare the operational variations and use cases of hashmaps and hash sets. Can a HashSet be considered a specific class of HashMap? Explain your response.
3. Compare and contrast the structural features of binary trees and binary search trees. What circumstances favour one over the other?
4. What kind of imbalances may exist in a BST? What methods are used by an AVL Tree to guarantee that these imbalances are fixed?
5. What is the heap property? Explain the differences between max-heaps and min-heaps and how each might be applied to priority queues.

5.10 Case Study

Title: Efficient Management of a Digital Library Database

Introduction:

The increasing prevalence of digital libraries in the era of technology brings to the forefront the notable issue of effectively managing enormous quantities of material. The underlying infrastructure of these libraries encompasses more than mere data storage; it also encompasses the facilitation of efficient data retrieval, changes, and maintenance of data integrity.

Case Study:

The National Digital Library (NDL) has had challenges in its search and retrieval processes, particularly during periods of high demand, resulting in delays. At the outset, they used rudimentary array-based structures; nevertheless, they quickly recognised their constraints as the number of archived records escalated.

Background:

The National Diet Library (NDL) has a vast collection of books, research papers, and articles which have been acquired from various institutions worldwide. Every item in the collection has metadata, which comprises essential information such as the title, author, publication year, and keywords. As the number of users increased, the National Digital Library (NDL) encountered delays in search operations, leading to user dissatisfaction due to the requirement for prompt resource accessibility.

Your Task:

The individual has been appointed in the capacity of a data structure expert with the objective of overhauling the backend architecture of NDL. The aim is to propose and execute the utilisation of data structures that would optimise the velocity and effectiveness of storage and retrieval activities while maintaining the integrity of the data.

Questions to Consider:

1. In this particular situation, it is necessary to choose the most suitable data structure(s) that can provide rapid and effective data retrieval.
2. In what ways might binary search trees or AVL trees be used to enhance search operations inside the library database?
3. Would the use of HashTables or HashMaps provide advantages in the management of metadata pertaining to library resources? What are the reasons for and against it?
4. Which data structures would guarantee the least amount of interruption and the greatest performance, given the frequent updates, additions, and deletions?

Recommendations:

In order to address the difficulties encountered by NDL, it is advisable to use a hybrid approach, including the utilisation of HashTables for the purpose of managing metadata. This approach guarantees an average time complexity of O(1) for retrieval operations. AVL trees may be used to maintain balance and achieve logarithmic time complexity for various operations when dealing with hierarchical or sorted data. Moreover, taking into account the ever-changing characteristics of the library, the use of priority queues might be employed to effectively handle real-time data processing activities.

Conclusion:

The flawless functioning of digital libraries relies heavily on the efficient handling of data. By implementing appropriate data structures that are customised to meet individual requirements, digital platforms such as NDL may optimise data retrieval speed, hence improving user satisfaction and operational effectiveness.

5.11 References

1. Goodrich, M.T., Tamassia, R. and Goldwasser, M.H., 2014. *Data structures and algorithms in Java*. John wiley & sons.
2. Mehlhorn, K., 2013. *Data structures and algorithms 1: Sorting and searching* (Vol. 1). Springer Science & Business Media.
3. Storer, J.A., 2003. *An introduction to data structures and algorithms*. Springer Science & Business Media.
4. Drozdek, A., 2013. *Data Structures and algorithms in C++*. Cengage Learning.
5. Lafore, R., 2017. *Data structures and algorithms in Java*. Sams publishing.