TREES

DATA STRUCTURES IN JAVA

Sercan Külcü | Data Structures In Java | 10.05.2023

Contents

Trees

A tree is a data structure that can be used to store a collection of elements. Trees are often used to store data that is hierarchical, such as a list of files in a directory.

In Java, trees are implemented using the Tree class. The Tree class defines several methods for adding, removing, and accessing elements in a tree.

Some of the most common methods in the Tree class include:

- add(): Adds an element to the tree.
- remove(): Removes an element from the tree.
- get(): Gets the element at a specific location in the tree.
- size(): Gets the number of elements in the tree.

Here are some examples of how trees can be used: Trees are a versatile data structure that can be used to store a variety of data.

- A tree can be used to store a list of files in a directory.
- A tree can be used to store a list of tasks to be completed.
- A tree can be used to store a list of websites to visit.
- A tree can be used to store a list of friends on social media.

Here are some additional details about trees:

- Trees are implemented using linked lists, which are data structures that store data in a linked list.
- Trees are not synchronized, which means that multiple threads can access the tree at the same time. This can lead to race conditions, which are errors that can occur when multiple threads are trying to access the same data at the same time.
- Trees are a good choice for storing data that is hierarchical, such as a list of files in a directory.

- Trees are a good choice for storing data that needs to be frequently searched.

Here are some tips for using trees:

- Use trees when you need to store a collection of elements that is hierarchical.
- Use trees when you need to be able to add and remove elements from the collection frequently.
- Use trees when you need to be able to search the collection for specific elements.

Here are some of the limitations of trees:

- Trees can be slow if the tree is unbalanced.
- Trees can be slow if the tree is very large.
- Trees are not thread-safe by default.

Here are some of the advantages of trees:

- Trees are very efficient for storing and retrieving data with a key.
- Trees are very scalable.

Here are some of the most common types of trees:

- Binary trees: A binary tree is a tree where each node has at most two children.
- Binary search trees: A binary search tree is a binary tree where the elements are sorted in ascending order.
- AVL trees: An AVL tree is a self-balancing binary search tree.
- Red-black trees: A red-black tree is a self-balancing binary search tree.

Binary Trees

A binary tree is a tree data structure in which each node has at most two children. Each node of a binary tree consists of three items:

- Data item
- Address of left child
- Address of the right child

The data item is the value stored at the node. The left child and right child are pointers to the nodes that are the children of the current node.

Binary trees can be used to represent a variety of data structures, such as sorted lists, sets, and maps. They are also used in many algorithms, such as binary search and binary tree traversal.

Types of Binary Trees

There are many different types of binary trees, each with its own advantages and disadvantages. Some of the most common types of binary trees include:

- Binary search trees are a type of binary tree in which the data items are sorted in increasing order. This makes binary search trees very efficient for searching for data items.
- Heaps are a type of binary tree in which the data items are arranged in a special way so that the largest (or smallest) data item is always at the top of the tree. This makes heaps very efficient for operations such as finding the maximum (or minimum) data item in the tree.
- Balanced binary trees are a type of binary tree in which the height of the tree is always logarithmic in the number of data items. This makes balanced binary trees very efficient for operations such as searching, inserting, and deleting data items.

Applications of Binary Trees

Binary trees are used in a wide variety of applications, including:

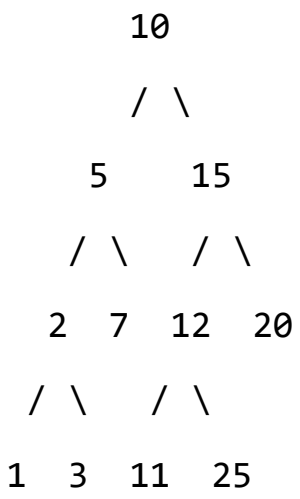
- Operating systems use binary trees to represent file systems.
- Compilers use binary trees to represent syntax trees.
- Databases use binary trees to represent indexes.
- Web browsers use binary trees to represent the structure of web pages.

Binary Search Trees

A binary search tree (BST) is a tree data structure in which each node has at most two children. The left child of a node contains all the nodes with values less than the value of the current node, and the right child contains all the nodes with values greater than the value of the current node. This property is known as the binary search property.

BSTs are a very efficient data structure for searching, inserting, and deleting elements. The time complexity of these operations is O(log n), where n is the number of nodes in the tree. This is because the search algorithm can quickly narrow down the search space by comparing the value of the target element to the values of the nodes in the tree.

Example: Here is an example of a binary search tree:



In this tree, value 10 is at the root of the tree. The left child of the root node contains all the nodes with values less than 10, and the right child contains all the nodes with values greater than 10. The left child of the left child contains all the nodes with values less than 5, and so on.

Operations

Searching is the process of finding a node with a given value in the tree. The search algorithm starts at the root of the tree and compares the value of the target node to the value of the root node. If the target node has a smaller value, the search algorithm recursively searches the left child of the root node. If the target node has a larger value, the search algorithm recursively searches the right child of the root node. The search algorithm continues to recursively search the tree until the target node is found or until the entire tree has been searched.

Inserting is the process of adding a new node to the tree. The insertion algorithm starts at the root of the tree and compares the value of the new node to the value of the root node. If the new node has a smaller value, the insertion algorithm recursively inserts the new node into the left child of the root node. If the new node has a larger value, the insertion algorithm recursively inserts the new node into the right child of the root node. The insertion algorithm continues to recursively insert the new node into the tree until the new node can be inserted into a leaf node.

Deleting is the process of removing a node from the tree. The deletion algorithm starts at the root of the tree and compares the value of the node to be deleted to the value of the root node. If the node to be deleted has a smaller value, the deletion algorithm recursively deletes the node from the left child of the root node. If the node to be deleted has a larger value, the deletion algorithm recursively deletes the node from the right child of the root node. The deletion algorithm continues to recursively delete the node from the tree until the node to be deleted is found.

Traversing is the process of visiting all the nodes in the tree. There are three different ways to traverse a binary search tree:

Inorder traversal visits the left child of a node, then the node itself, and then the right child of the node.

Preorder traversal visits the node itself, then the left child of the node, and then the right child of the node.

Postorder traversal visits the left child of a node, then the right child of the node, and then the node itself.

Balanced Search Trees

A balanced search tree is a binary search tree in which the height of the left and right subtrees of any node differs by at most 1. This ensures that the time complexity of all operations on the tree is O(log n), where n is the number of nodes in the tree.

There are many different types of balanced search trees, each with its own advantages and disadvantages. Some of the most common types of balanced search trees include:

- AVL trees are a type of balanced search tree in which the height of the left and right subtrees of any node differs by at most 1. AVL trees are very efficient for operations such as searching, inserting, and deleting data items.
- Red-black trees are a type of balanced search tree in which the nodes are colored red or black. Red-black trees are very efficient for operations such as searching, inserting, and deleting data items, and they are also very easy to implement.
- B-trees are a type of balanced search tree in which the nodes can store multiple data items. B-trees are very efficient for operations such as searching, inserting, and deleting large amounts of data.

AVL Trees

AVL trees are a type of self-balancing binary search tree. This means that the height of the tree is always logarithmic in the number of nodes, which ensures that all operations on the tree can be performed in O(log n) time.

AVL trees are named after their inventors, Adelson-Velsky and Landis. They were first published in their 1962 paper "An algorithm for the organization of information".

AVL trees work by keeping the height of the tree balanced. This is done by rotating the tree whenever a node is inserted or deleted. Rotations are operations that move nodes around in the tree in order to maintain the balance.

There are four types of rotations in AVL trees:

- Left rotation rotates a node around its left child.
- Right rotation rotates a node around its right child.
- Left-right rotation rotates a node around its left child, and then rotates the resulting node around its right child.
- Right-left rotation rotates a node around its right child, and then rotates the resulting node around its left child.

B-Trees

B-trees are a type of tree data structure that is used to store large amounts of data efficiently. B-trees are a self-balancing tree, which means that the height of the tree is always logarithmic in the number of nodes. This ensures that all operations on the tree, such as searching, inserting, and deleting, can be performed in O(log n) time.

B-trees are named after their inventors, Bayer and McCreight. They were first published in their 1972 paper "Organization and maintenance of large, ordered indices".

B-trees work by storing data in nodes. Each node can store a certain number of keys and pointers to other nodes. The number of keys that can be stored in a node is called the order of the B-tree.

The root node of a B-tree can store up to m-1 keys, where m is the order of the B-tree. The children of the root node can store up to m keys each, and so on. The leaf nodes of the B-tree can store up to m-1 keys each.

When a new key is inserted into a B-tree, the tree is rebalanced. Rebalancing is done by splitting nodes that are full and merging nodes that are underfull.

Red-Black Trees

Red-black trees are a type of self-balancing binary search tree. This means that the height of the tree is always logarithmic in the number of nodes, which ensures that all operations on the tree, such as searching, inserting, and deleting, can be performed in O(log n) time.

Red-black trees are named after their inventors, Bayer and McCreight. They were first published in their 1972 paper "Organization and maintenance of large, ordered indices".

Red-black trees work by storing data in nodes. Each node can store a certain number of keys and pointers to other nodes. The number of keys that can be stored in a node is called the order of the red-black tree.

The root node of a red-black tree can store up to m-1 keys, where m is the order of the red-black tree. The children of the root node can store up to m keys each, and so on. The leaf nodes of the red-black tree can store up to m-1 keys each.

Each node in a red-black tree has a color, which can be either red or black. The following rules must be obeyed in a red-black tree:

- Every node is either red or black.
- The root node is black.
- Every leaf node is black.
- If a node is red, then both its children must be black.
- Every path from the root node to a leaf node contains the same number of black nodes.

When a new key is inserted into a red-black tree, the tree is rebalanced. Rebalancing is done by rotating nodes that are out of balance.

Splay Trees

Splay trees are a type of binary search tree that are dynamically balanced. This means that the height of the tree is always logarithmic in the number of nodes, which ensures that all operations on the tree, such as searching, inserting, and deleting, can be performed in O(log n) time.

Splay trees are named after the splaying operation, which is a restructuring operation that moves the accessed node to the root of the tree. Splaying ensures that the accessed node is always near the root of the tree, which makes subsequent access to the node faster.

Splay trees work by storing data in nodes. Each node can store a key and a pointer to its left and right child nodes. The root node of a splay tree is the node with the smallest key.

When a new key is inserted into a splay tree, the splaying operation is performed on the new node. The splaying operation moves the new node to the root of the tree.

When a key is deleted from a splay tree, the splaying operation is performed on the node that was deleted. The splaying operation moves the deleted node to the root of the tree.

Efficient insertion and search in logarithmic time

To support both efficient insertion and efficient search in logarithmic time, we need to consider a data structure that can allow us to take advantage of the fast search time of a sorted list (like in binary search), while minimizing the cost of insertion (which requires shifting elements in a sorted array).

Search in log n time: This is achievable with a sorted list or tree, where we can use binary search.

Insertion in log n time: This is tricky because, in an array-based list, inserting an element into the sorted list requires shifting elements to maintain the sorted order, which takes O(n) time. A tree-like structure can help us avoid this problem by allowing us to insert and keep the list sorted with fewer movements.

Possible Solution: Balanced Search Tree (BST)

One approach to efficiently support both insertion and search in logarithmic time is to use a self-balancing binary search tree (BST), such as: AVL Tree, Red-Black Tree, B-tree (for larger datasets)

1. Self-Balancing Binary Search Tree (BST)

A binary search tree is a tree where each node contains a key, and for any node, all keys in its left subtree are smaller, and all keys in its right subtree are larger. Insertion and search in a balanced binary search tree can be done in O(log n) time.

Why?

Insertion: When inserting a new element, the tree will rebalance itself after insertion to ensure that the height remains logarithmic. This guarantees that no operation in the tree exceeds O(log n) time.

Search: Binary search on a balanced tree works by comparing the target value to the value at the current node and recursively searching in the left or right subtree, which takes O(log n) time.

2. B-tree (or B+ tree for database indexing)

A B-tree is a self-balancing search tree that maintains sorted data and allows for efficient insertion, deletion, and search in O(log n) time. It is particularly useful for storing large datasets that are too big to fit into memory (e.g., databases).

A B+ tree, a variant of the B-tree, is optimized for systems that read and write large blocks of data and is widely used for indexing in databases.

B-trees are more efficient in situations where there are frequent insertions and deletions because they store multiple keys per node, making them less likely to require rebalancing compared to a traditional binary search tree. They support log n search time and log n insertions due to their balanced structure.