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 a number of 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.

Trees are a powerful tool that can be used to store and manipulate data. By understanding how trees work, you can write more efficient and effective code.

Here are some examples of how trees can be used:

- 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.

Trees are a versatile data structure that can be used to store a variety of data. By understanding how trees work and using them correctly, you can write more efficient and effective code.

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.

Trees are a powerful tool that can be used to store and manipulate data. By understanding how trees work and using them correctly, you can write more efficient and effective code.

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 by key.
- Trees are very scalable.
- Trees are very versatile.

Overall, trees are a powerful tool that can be used to store and manipulate data. By understanding the limitations and advantages of trees, you can use them effectively in your Java programs.

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.

Each type of tree has its own advantages and disadvantages. By understanding the different types of trees, you can choose the right type of tree for your specific needs.

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 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.

Operations on Binary Trees

There are many different operations that can be performed on binary trees. Some of the most common operations include:

- Searching is the process of finding a data item in a binary tree.
- Inserting is the process of adding a new data item to a binary tree.
- Deleting is the process of removing a data item from a binary tree.
- Traversing is the process of visiting all of the nodes in a binary tree.

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.

Conclusion

Binary trees are a powerful data structure that can be used to represent a variety of data structures and algorithms. They are a versatile and efficient data structure that can be used in a wide variety of applications.

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.

BSTs can also be used to represent a variety of other data structures, such as sorted lists, sets, and maps.

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

```
      10

      / \

    5    15

   / \   / \

  2  7  12  20

 / \   / \

1  3  11  25
```

In this tree, the 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

The following are some of the most common operations that can be performed on binary search trees:

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.

Applications

BSTs are used in a wide variety of applications, including:

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

Conclusion

BSTs are a powerful data structure that can be used to represent a variety of data structures and algorithms. They are a versatile and efficient data structure that can be used in a wide variety of applications.

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 differ 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 differ 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.

Operations on Balanced Search Trees

The following are some of the most common operations that can be performed on balanced search trees:

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 balanced 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 the node, then the right child of the node, and then the node itself.

Applications

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

- Operating systems use balanced search trees to represent file systems.
- Compilers use balanced search trees to represent syntax trees.

- Databases use balanced search trees to represent indexes.
- Web browsers use balanced search trees to represent the structure of web pages.

Conclusion

Balanced search trees are a powerful data structure that can be used to represent a variety of data structures and algorithms. They are a versatile and efficient data structure that can be used in a wide variety of applications.

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".

How AVL Trees Work

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.

Operations on AVL Trees

The following are some of the most common operations that can be performed on AVL trees:

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 an AVL 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 the node, then the right child of the node, and then the node itself.

Applications

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

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

Conclusion

AVL trees are a powerful data structure that can be used to represent a variety of data structures and algorithms. They are a versatile and efficient data structure that can be used in a wide variety of applications.

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".

How B-Trees Work

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.

Operations on B-Trees

The following are some of the most common operations that can be performed on B-trees:

Searching is the process of finding a key in a B-tree. The search algorithm starts at the root node and compares the key to the keys in the root node. If the key is found in the root node, the algorithm returns the node. If the key is not found in the root node, the algorithm recursively searches the children of the root node. The algorithm continues to recursively search the tree until the key is found or until the entire tree has been searched.

Inserting is the process of adding a new key to a B-tree. The insertion algorithm starts at the root node and compares the key to the keys in the root node. If the key is smaller than the smallest key in the root node, the algorithm recursively inserts the key into the left child of the root node. If the key is larger than the largest key in the root node, the algorithm recursively inserts the key into the right child of the root node. If the key is already in the tree, the algorithm does nothing.

Deleting is the process of removing a key from a B-tree. The deletion algorithm starts at the root node and compares the key to the keys in the root node. If the key is found in the root node, the algorithm recursively deletes the key from the root node. If the key is not found in the root node, the algorithm recursively searches the children of the root node. The algorithm continues to recursively search the tree until the key is found or until the entire tree has been searched.

Traversing is the process of visiting all the keys in a B-tree. There are three different ways to traverse a B-tree:

- Inorder traversal visits the keys in the left child of a node, then the key in the node itself, and then the keys in the right child of the node.
- Preorder traversal visits the key in the node itself, then the keys in the left child of the node, and then the keys in the right child of the node.
- Postorder traversal visits the keys in the left child of a node, then the keys in the right child of the node, and then the key in the node itself.

Applications

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

- Databases use B-trees to store indexes.

- File systems use B-trees to store directories.
- Operating systems use B-trees to store file tables.
- Compilers use B-trees to store symbol tables.

Conclusion

B-trees are a powerful data structure that can be used to store large amounts of data efficiently. B-trees are a versatile and efficient data structure that can be used in a wide variety of applications.

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".

How Red-Black Trees Work

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.

Operations on Red-Black Trees

The following are some of the most common operations that can be performed on red-black trees:

Searching is the process of finding a key in a red-black tree. The search algorithm starts at the root node and compares the key to the keys in the root node. If the key is found in the root node, the algorithm returns the node. If the key is not found in the root node, the algorithm recursively searches the children of the root node. The algorithm continues to recursively search the tree until the key is found or until the entire tree has been searched.

Inserting is the process of adding a new key to a red-black tree. The insertion algorithm starts at the root node and compares the key to the keys in the root node. If the key is smaller than the smallest key in the root node, the algorithm recursively inserts the key into the left child of the root node. If the key is larger than the largest key in the root node, the algorithm recursively inserts the key into the right child of the root node. If the key is already in the tree, the algorithm does nothing.

Deleting is the process of removing a key from a red-black tree. The deletion algorithm starts at the root node and compares the key to the keys in the root node. If the key is found in the root node, the algorithm recursively deletes the key from the root node. If the key is not found in the root node, the algorithm recursively searches the children of the root node. The algorithm continues to recursively search the tree until the key is found or until the entire tree has been searched.

Traversing is the process of visiting all the keys in a red-black tree. There are three different ways to traverse a red-black tree:

- Inorder traversal visits the keys in the left child of a node, then the key in the node itself, and then the keys in the right child of the node.
- Preorder traversal visits the key in the node itself, then the keys in the left child of the node, and then the keys in the right child of the node.

- Postorder traversal visits the keys in the left child of a node, then the keys in the right child of the node, and then the key in the node itself.

Applications

Red-black trees are used in a wide variety of applications, including:

- Databases use red-black trees to store indexes.
- File systems use red-black trees to store directories.
- Operating systems use red-black trees to store file tables.
- Compilers use red-black trees to store symbol tables.

Conclusion

Red-black trees are a powerful data structure that can be used to store large amounts of data efficiently. Red-black trees are a versatile and efficient data structure that can be

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.

How Splay Trees Work

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.

Operations on Splay Trees

The following are some of the most common operations that can be performed on splay trees:

Searching is the process of finding a key in a splay tree. The search algorithm starts at the root node and compares the key to the keys in the root node. If the key is found in the root node, the algorithm returns the node. If the key is not found in the root node, the algorithm recursively searches the children of the root node. The algorithm continues to recursively search the tree until the key is found or until the entire tree has been searched.

Inserting is the process of adding a new key to a splay tree. The insertion algorithm starts at the root node and compares the key to the keys in the root node. If the key is smaller than the smallest key in the root node, the algorithm recursively inserts the key into the left child of the root node. If the key is larger than the largest key in the root node, the algorithm recursively inserts the key into the right child of the root node. If the key is already in the tree, the algorithm does nothing.

Deleting is the process of removing a key from a splay tree. The deletion algorithm starts at the root node and compares the key to the keys in the root node. If the key is found in the root node, the algorithm recursively deletes the key from the root node. If the key is not found in the root node, the algorithm recursively searches the children of the root node. The algorithm continues to recursively search the tree until the key is found or until the entire tree has been searched.

Traversing is the process of visiting all the keys in a splay tree. There are three different ways to traverse a splay tree:

- Inorder traversal visits the keys in the left child of a node, then the key in the node itself, and then the keys in the right child of the node.
- Preorder traversal visits the key in the node itself, then the keys in the left child of the node, and then the keys in the right child of the node.
- Postorder traversal visits the keys in the left child of a node, then the keys in the right child of the node, and then the key in the node itself.

Applications

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

- Databases use splay trees to store indexes.

- File systems use splay trees to store directories.
- Operating systems use splay trees to store file tables.
- Compilers use splay trees to store symbol tables.

Conclusion

Splay trees are a powerful data structure that can be used to store large amounts of data efficiently. Splay trees are a versatile and efficient data structure that can be used in a wide variety of applications.