Kouretas Panagiotis - Theodoros

AM: E19072

Data Processing Techniques: Implementation of range query algorithm and Best First algorithm on a B-Tree structure.

Technical report

The **BTree** class is the main class that represents the B-tree data structure. It contains methods for inserting, searching, and removing elements from the tree. The algorithms used in this class are based on the B-tree data structure theory.

The **insert** method is used to insert a new key-value pair into the tree. It first checks if the current node is a leaf node, and if so, it adds the key-value pair to the node and rebuilds the node if necessary. If the current node is not a leaf node, it finds the subtree where the key-value pair should be inserted and recursively calls the insert method on that subtree. This is the standard B-tree insertion algorithm.

The **search** method is used to find the value associated with a specific key. It starts at the root node and compares the key to the keys in the current node. If the key is less than or equal to a key in the node, it recursively searches the corresponding child subtree. If the key is greater than all the keys in the node, it recursively searches the last child subtree. If the current node is a leaf node, it searches through the keys in the node to find the key-value pair with the matching key. If it is found, it returns the value, otherwise it returns null. This is the standard B-tree search algorithm.

The **remove** method is used to remove a key-value pair from the tree. It first finds the node containing the key and then checks if the node has more than the minimum number of keys required to keep the tree balanced. If it does, it simply deletes the key-value pair from the node. If it does not, it needs to borrow a key from a neighbouring node or merge with a neighbouring node to keep the tree balanced. This is the standard B-tree deletion algorithm.

The **Node** class is used to represent a node in the B-tree. It contains the keys and children of the node and methods for managing them, such as adding and deleting keys, as well as getting information about the node such as whether it is a leaf node or not.

The **Pair** class is used to store key-value pairs in the tree. It contains a key and a value and methods for getting and setting them.

The **Main** class is used to instantiate and test the BTree class. The **getT()** method is used to set the order of the B-tree, which determines the maximum number of children per node and is used to keep the tree balanced.

The **rangeQuery** method is used to find all the key-value pairs in the B-tree within a specific range of keys. It starts at the root node and recursively searches the left subtrees of all nodes whose keys are greater than the upper bound of the range. It also recursively searches the right subtrees of all nodes whose keys are less than the lower bound of the range. For all other nodes, it adds their keys to a list of key-value pairs that are within the range. The method returns the list of key-value pairs as the result of the range query. This algorithm is based on the property of B-tree that all the keys to the left of a node are less than or equal to the node's keys and all the keys to the right of a node are greater than the node's keys. This allows the algorithm to efficiently search the tree by only considering nodes and subtrees that have a chance of containing keys within the range. The **rangeQuery** method is a variation of a standard B-tree traversal algorithm, where instead of visiting all the nodes it visits only the nodes whose keys are in the range.

The **Best First** algorithm begins by starting at the root node of the Btree and then traversing down the tree until a leaf node is reached. At this point, the algorithm will iterate through all of the key-value pairs in the leaf node, searching for the key that was passed as a parameter to the **findKey()** method. If the key is found, it is returned as a Pair object. If the key is not found, the algorithm returns null. The **findSubtree()** method is used to traverse down the tree, and it utilizes a simple linear search to determine which child node to visit next. This method takes in a current node and a Pair object containing the key to be searched for. It iterates through the keys in the current node, comparing the key of the Pair object passed in to the key of each pair in the node. If the key of the Pair object is less than or equal to the key of the current pair, the method returns the corresponding child node. If the key of the Pair object is greater than the key of the current pair, the method continues iterating through the keys in the node until a pair is found with a key that is greater than or equal to the key of the Pair object. In the Main class, an instance of the BTree class is created and a few key-value pairs are inserted into the tree. The **findKey()** method is then called, passing in the key to be searched for. If the key is found, the key is printed to the console. If the key is not found, the message "Key not found" is printed to the console.

Developer guide

In my main class in the 31-52 I have in comments the range query algorithm example on the BTree. In the lines 54-74 you can see the example of the Best First algorithm on the BTree. To perform the example of the Best First you just run the code as it is. To perform the example of the range query you have to comment in the lines 54-74 and to uncomment the lines 31-52.

The following points in my code can be changed based on what we want to achieve:

Instantiate the BTree class:

BTree bTree = new BTree();

This will create a new B-tree with the default order (which is set in the Main class's **getT()** method).

Insert key-value pairs into the B-tree:

Pair key = new Pair(5, "value5");

bTree.insert(key);

This will insert a new key-value pair with key 5 and value "value5" into the B-tree.

Search for a specific key in the B-tree:

Pair result = bTree.search(5);

This will search for the key-value pair with key 5 in the B-tree and return the pair. If the key is not found, it will return null.

Remove a key-value pair from the B-tree:

bTree.remove(5);

This will remove the key-value pair with key 5 from the B-tree.

Perform range query on the B-tree:

List<Pair> result = bTree.rangeQuery(2, 8);

This will return a list of key-value pairs with keys between 2 and 8 inclusive Change the order of the B-tree:

bTree.T = 10;

This will change the order of the B-tree to 10, which means that each node can have up to 10 children.

Perform Best First on the B-tree:

```
Pair result = tree.findKey(40);
```

This will return the key if it is found, otherwise it will return the message "Key not found".