**B-Tree Initial Class Design**

Capstone Project CS 4488 – Andreas Kramer

* **BTree Class**

This class represents the B-Tree itself.

Attributes:

* ***Root*** 
  + A reference to the root node of the B-Tree. (LeafNode)
* ***M***
  + The degree of the B-Tree. Determines the maximum and minimum number of keys/entries stored in each node. Specifically, each node (except possibly the root) must have at least M-1 and at most 2\*M-1 keys.

Methods:

* **insert(key, [data])**
  + Inserts a new key (and associated data into the tree). Finds the appropriate leaf node for the new key and inserts it there, potentially causing a split if the node gets full.
* **delete(key)** 
  + Removes a key (and its associated data) from the tree. This method involves finding the key, removing it, and then handling underflow in nodes, which might involve merging or redistributing keys.
* **search(key)** 
  + Searches for a key in the tree and returns the node containing it, if it exists. This is typically a traversal from the root down to the appropriate leaf. If the key is found, it returns the node containing the key and the position of the key in the node; otherwise, it returns null.
* **traverse()**
  + Performs an in-order traversal of the tree and returns a sorted list of keys. This method should be used for debugging or displaying the tree's contents, as long as we do not have a visual representation.
* **BTreeNode Class (Abstract/Base Class)**

This is an abstract class that serves as the base for the different types of nodes in the B-Tree.

Defines the common structure and behavior for the nodes in the B-Tree. Might provide interfaces for operations but does not implement them.

Attributes:

* + **keys**
    - An array of keys stored in the node. The keys are stored in a sorted order to facilitate efficient search, insert, and delete operations.
  + **parent**
    - Reference to the node’s immediate parent node, facilitating upward traversal and backtracking.
  + **leaf**
    - A boolean flag indicating whether the node is a leaf. Would help in determining the node's behavior during insert and delete operations. (optional)
  + **numKeys**
    - An integer representing the current number of valid keys in the node. This count is essential for understanding the state of the node, particularly during insertions and deletions.

Methods (not implementing since abstract):

* + **searchKey(key)** Searches for a given key within the node. The implementation of this method will differ in leaf and non-leaf nodes (use override).
  + **split()** 
    - Splits the node when it’s full. Will be implemented differently in leaf and non-leaf nodes.
  + **isFull()** 
    - Checks if the node is full (more than maximum key capacity)
  + **IsUnderflow()**
    - Checks if the node has too few keys. (less than minimum key capacity)
  + **insert(key, [data])** 
    - inserts a key into a node that is not full. Is used during the insert operation of the BTree class. Needs to check if the node needs to be split after insertion. Will need different implementations in leaf nodes and non-leaf nodes.
  + **deleteKey(key)** 
    - deletes a key from the node. Is used during the delete operation of the BTree class. Needs to check if redistribution or merging is needed. If a node has too few keys. Implemented differently in different nodes.
* **LeafNode Class (extends BTreeNode)**

This class represents the leaf nodes in the B-Tree. It is inherited from the abstract BTreeNode so needs to implement all its methods and attributes. (In B-Trees leaf nodes are NOT linked as linked lists)

Attributes:

Should already automatically have the following attributes inherited from the abstract class:

* + **keys**
    - An array of keys stored in the node. The keys are stored in a sorted order to facilitate efficient search, insert, and delete operations.
  + **parent**
    - Reference to the node’s immediate parent node, facilitating upward traversal and backtracking.
  + **leaf**
    - A boolean flag indicating whether the node is a leaf. Would help in determining the node's behavior during insert and delete operations. (**optional**)
  + **numKeys**
    - An integer representing the current number of valid keys in the node. This count is essential for understanding the state of the node, particularly during insertions and deletions.
  + **data**
    - An array of data entry objects, that come with an associated key.

Methods

Needs to implement all methods that are mentioned in the abstract node class including:

* **insertKey(key, [data]):** 
  + Inserts a key into the leaf node. If the node is full, it triggers a split.
* **deleteKey(key):** 
  + Deletes a key from the leaf node. Also deletes the associated data entry. Handles the deletion and any necessary rebalancing (or redistributing) of the tree due to underflow. (too few keys).
* **mergeWith(LeafNode leaf)**
  + Combines two leaf nodes into one leaf node. Correctly handles redistribution of attributes, and keeps keys sorted. (deletes the empty node). Is called during the deletion process, if nodes suffer from underflow.
* **split()**
  + splits a leaf node into two leaf nodes in case the maximum number of keys is reached/overstepped. Correctly distributes half the entries and keys into the new node and keeps the other half in the original. Successfully links the nodes into the appropriate place in the tree.
* **NonLeafNode Class (extends BTreeNode)**

This class represents the non-leaf (internal) nodes in the B-Tree.

Attributes:

Should already automatically have the following attributes inherited from the abstract class:

* + **keys**
    - An array of keys stored in the node. The keys are stored in a sorted order to facilitate efficient search, insert, and delete operations.
  + **parent**
    - Reference to the node’s immediate parent node, facilitating upward traversal and backtracking.
  + **leaf**
    - A boolean flag indicating whether the node is a leaf. Would help in determining the node's behavior during insert and delete operations. (**optional**)
  + **numKeys**
    - An integer representing the current number of valid keys in the node. This count is essential for understanding the state of the node, particularly during insertions and deletions.

Since non-leaf nodes (internal nodes) have pointers to child nodes it also needs an attribute for it:

* + **children**
    - An array of pointers/references to child nodes. The size of this array is usually one more than the number of keys, to point to all the child nodes that fall between and outside the key values. (on second thought do not implement this since leaf nodes do not have children)

Methods:

Needs to implement all methods that are mentioned in the abstract node class including:

* **insertKey(key, [data]):** 
  + Inserts a key and the associated data entry into the leaf node. If the node is full, it triggers a split.
* **deleteKey(key):** 
  + Deletes a key from the leaf node. Handles the deletion and any necessary rebalancing (or redistributing) of the tree due to underflow. (too few keys).
* **mergeWith(LeafNode leaf)**
  + Combines two non-leaf nodes into one non-leaf node. Correctly handles redistribution of attributes, and keeps keys sorted. (deletes the empty node). Is called during the deletion process, if nodes suffer from underflow.
* **split()**
  + splits a leaf node into two leaf nodes in case the maximum number of keys is reached/overstepped. Correctly distributes half the entries and keys into the new node and keeps the other half in the original. Successfully links the nodes into the appropriate place in the tree.
* **insertChild(key, childNode)**
  + Inserts a new child node into the appropriate position after a split in a child node. This method updates keys and children arrays of the non-leaf node as well.
* **deleteChild(key/or index)** 
  + Deletes a reference to a child node based on the given key(index). This method is involved in handling deletions in the tree and may require merging or redistributing keys and children if underflow occurs.
* **Data Class**

This class stores data associated with keys. This can be any object. A data entry object typically encapsulates the actual information stored in the B-Tree.

Attributes:

* + i.e. Person with ID (used as the key), Name, Age, Email, Profession, etc…)

Methods:

* + Basic methods like toString() for debugging purposes and displaying a data entry in a single line.

**Remarks and Order of Implementation:**

1. **Define a Data class with relevant attributes and methods.**

* B-Tree stores the data objects and sorts them based on their key (integer).

1. **Implement the BTreeNode Class:**

* Start with the basic structure: attributes for keys, children, leaf status, and the number of keys.
* Implement methods common to both leaf and non-leaf nodes, such as isFull, and placeholders for abstract methods like insertKey and deleteKey.

1. **Extend the BTreeNode class to implement these classes.**

* For LeafNode, implement specific methods like insertKey and deleteKey.
* For NonLeafNode, focus on methods that manage children, like insertChild, deleteChild, and split

1. **Implement the BTree Class:**

* Define attributes like the root and the minimum degree (T).
* Implement the search method to validate the tree structure.
* Implement the insert method, starting with cases that don't require splitting nodes.
* Add the logic to handle splitting nodes during insertion.
* Make sure insertion works 100% correctly before moving on.
* Implement the delete method, starting with simpler cases (e.g., deleting from a leaf).
* Progressively add complexity to the delete method, handling cases that require merging or redistributing keys.

1. **Implement Utility Functions: (can also be implemented earlier for debugging)**

* Functions for tree traversal, printing, and debugging.
* Essential for testing and verifying the correctness of the implementation.

1. **Testing and Debugging:**

* Throughout the development process, write tests for each functionality as we implement it.
* Start with unit tests for individual methods and progress to integration tests that verify the interaction between different components of the B-Tree

1. **Optimization:**

* Once the basic functionality is working, optimize the implementation for efficiency.
* Refine the code for readability, maintainability, and adherence to coding standards.