Project Phase 3 - B+Tree Indexing

Deadline - 11:59 PM, 31st October 2023

The objective of Phase 1 is to implement a B+Tree index. The disk-based index is built on top of an unordered heap file organization. In this project we assume records are unique 1-sized tuples consisting of non-negative integers (records consist of only one attribute which is a key attribute). This phase consists of 3 parts -

- Implement B+ Tree insertion
- Implement B+ Tree deletion
- Perform analysis

The Codebase for this project is different. Accept the group assignment on GitHub Classroom . The project is built using CMake (version3.20).

We strongly suggest you to read: Section 17.3 (Dynamic Multilevel Indexes Using B-Trees and B+ Trees) from Pg617 - Pg624, Chapter 17 (Indexing Structures for Files and Physical Database Design),Fundamentals of Database Systems, 7th Edition, R Elmasri, Shamkant B Navathe, R Elmasri, SB Navathe . This document does not explain B+Tree indexing in detail, it is recommended you read the text book before starting on this project phase.

Queries

The provided codebase implements a query execution engine that processes the following 6 commands

- INSERT, DELETE, RANGE, SOURCE, EXPORT and QUIT.

Syntax	Description
INSERT `<key>`	Inserts `key` into heap and B+Tree
DELETE `<key>`	Deletes `key` from heap and B+Tree
RANGE `<key1>`` ``<key2>`	Performs a range query to retrieve keys within the range `[key1, key2]`. The results are written in file `data/range_[key1_key2].txt`. This command also prints the number of block accesses needed to perform this query using the B+Tree and just using the heap
SOURCE `<file>`	Executes all commands in file `data/<file>`
EXPORT	Exports heap into the file `heap.md` and B+Tree into the file `data/bptree.md`
QUIT	Quits the command line interpreter

Unordered Heap (class Unordered Heap)

We use an unordered heap to organize the data on the disk (class UnorderedHeap). The heap consists of a linked list of data blocks (class Block). The heap object contains a pointer that points to the first data block in the list (UnorderedHeap::first_block_ptr). Each data block can have at most BLOCK_SIZE (4) records. Similar to the SimpleRA codebase, each data block is stored as a file in the temp directory. All the functions for the unordered heap have already been implemented. When a new record is inserted, we start searching for an "empty space" in a block starting from the first block. If no empty space exists, we insert the record in a new block. An "empty space" is generated when a deletion occurs. In the code, an empty space is denoted using the DELETE_MARKER (-1). You can experiment with the unordered heap by running the list of commands in file data/heap. It generates the following linked list.

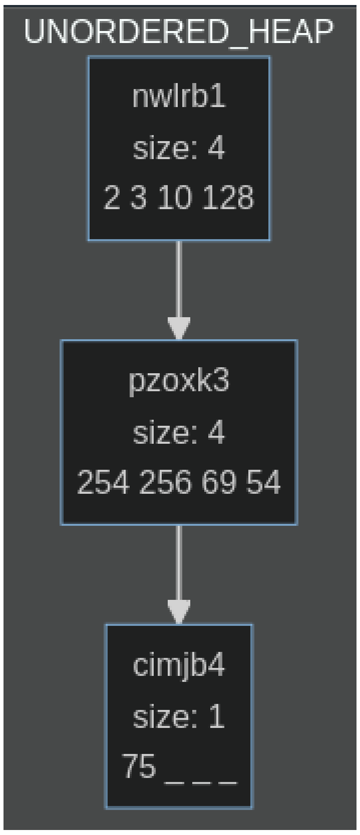

Figure 1: heap

The name of the blocks nwlrb1, pzoxk3 and cimjb4 are the names of the files in which the first, second and third data blocks are stored respectively.

B+Tree Indexing (class BPTree)

A B+Tree (class BPTree) index is a tree structure consisting of tree nodes (class TreeNode) - internal nodes (class InternalNode) and leaf nodes (class LeafNode) satisfying the following properties. An internal node consists of keys and tree pointers.

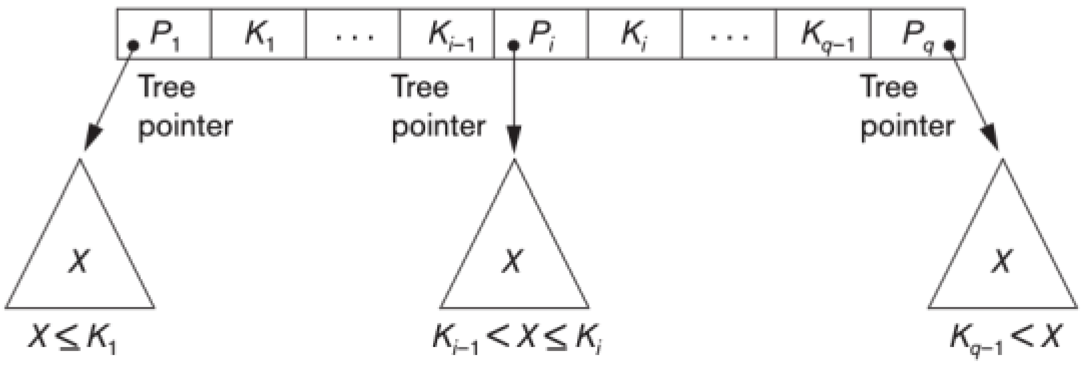

Figure 2: internal node structure

All the elements in the subtree rooted at P_i are greater than K_i – 1 and less than or equal to K_i. All the keys in the internal node are sorted ($K_1 < K_2 < ... < K_{q-1}$). Every internal node, except the root node, should have at least ⌈f/2⌉ and at most f children (tree pointers), where f is the fanout of the B+Tree (FANOUT = 3).

A leaf consists of <K, data_pointer> pairs, where K is the key or value of the indexing field, and data_pointer points to the location of the record on the disk. Data pointers could be block or record pointers - in this project, we use record pointers (class RecordPtr). Each record pointer holds a pointer to the block and the position of the record within the block.

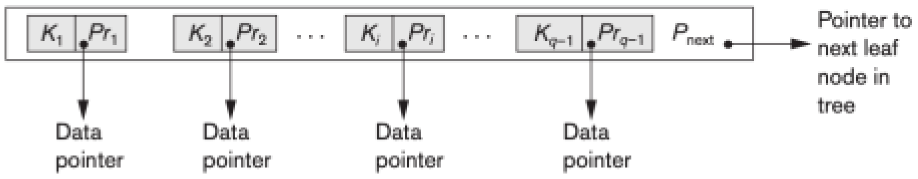

Figure 3: leaf node structure

Every leaf points to the next leaf. Every leaf node (except the root node) should have at least ⌈f/2⌉ and at most f children, represented as pairs of <key, record pointers>, where f is the fanout of the B+Tree.

```
The insertion and deletion operators have not been implemented in the code
base. Implementing B+Tree insertion is part (a) and deletion is part (b) of
this project.
```

Chart

A fun aspect of this project is the dynamic generation of a visual mermaid chart stored in the file data/chart.md. This chart visually represents the B+Tree and heap. The insert and delete functions for the B+Tree are not implemented; therefore, the boilerplate code will generate a chart of the heap only. To view this chart, open the data/chart.md file in a Markdown editor that supports rendering mermaid charts.

```
CLion is capable of rendering this chart, the Mermaid option has to be
enabled in the settings. VSCode is also capable of rendering mermaid charts
```

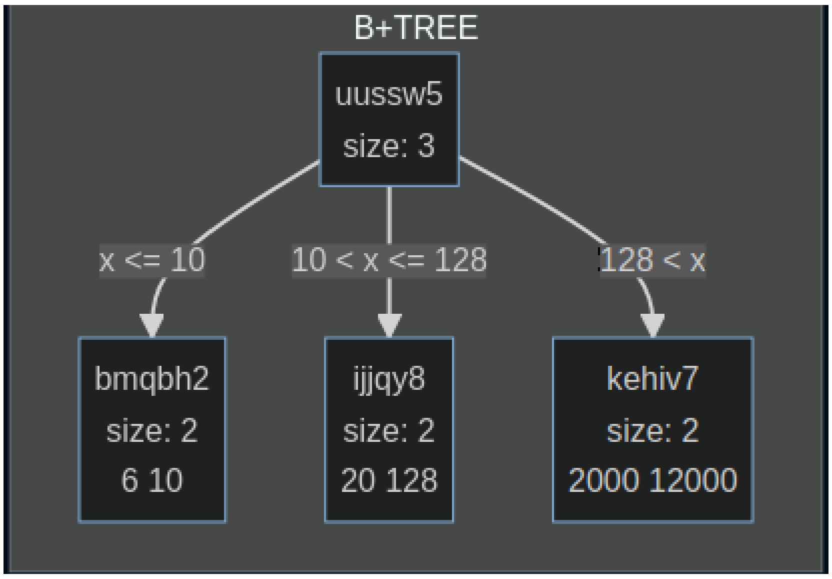

Figure 4: chart

Code Structure

The code base is structured as follows - the unordered heap (global objectheap) consists of a linked list of multiple data blocks (calledBlock) and the BPTree (global objectbptree) points to the root tree node. Any tree node (calledTreeNode) is either an internal node (InternalNode) or a leaf node (LeafNode). An internal node consists of many children which point to tree nodes (internal or leaf). A leaf node contains many<key, RecordPtr>pairs.

A given query is run both on the B+Tree and unordered heap. When insert is called on the heap, the heap returns a record pointer which points to the position of the inserted key on the disk. This key and returned record pointer should be inserted into the B+Tree. The delete function is called on both the heap and the B+Tree. As a part of this project, you will have to fill in theinsertanddelete functions in theLeafNodeandInternalNodeclasses. The rest of the code has been written for you. It is recommended you follow the order of the functions we ask you to implement

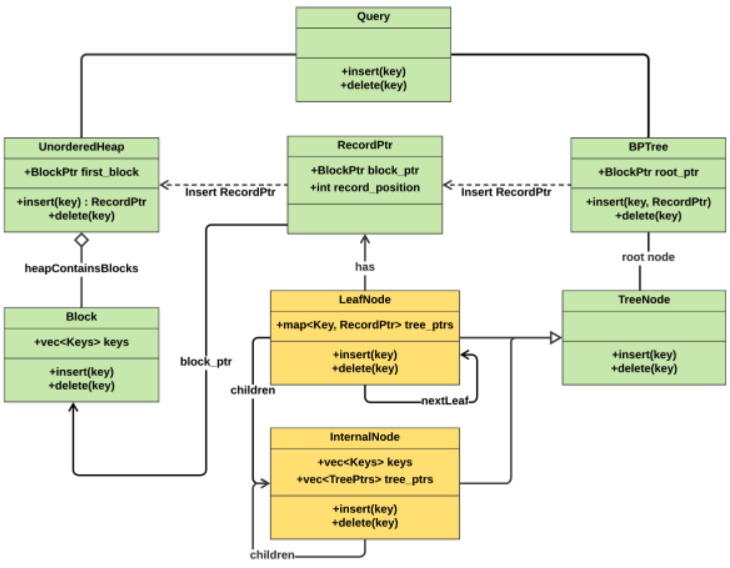

Figure 5: class diagram

Part (a) - Insertion

In this part, you have to complete two function definitions. The first is performing insert in leaf nodes.

```
TreePtr LeafNode::insert_key(Key key, RecordPtr record_ptr)
```

Implementing insertion in the leaf node is straight forward - if there is space in the leaf, we insert the <key, record_ptr> pair and write the leaf back to the disk. If there is no space in the leaf (overflow), split the leaf into two and return a tree pointer pointing to the new split leaf.

```
When a leaf node n is split to create a new leaf node split_node, node n
retains the smallest(key) ⌈f/2⌉ <key, RecordPtr>pairs and the rest are
inserted into split_node.
```

The case where the root node is split is taken care of within the BPTree class. To test this function, run the set of commands from leaf_insert. The expected output is

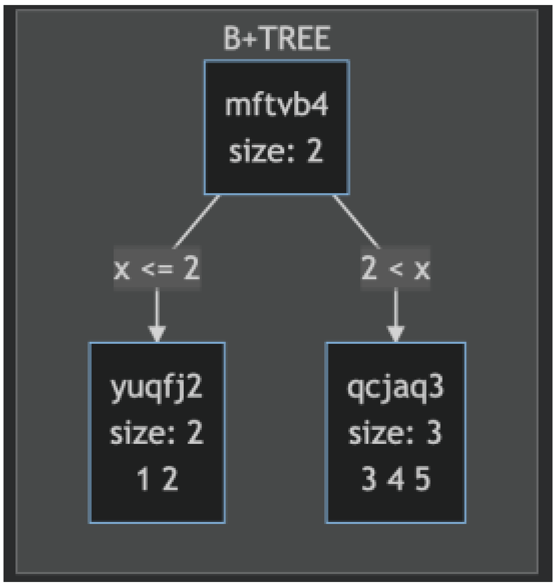

Figure 6: leaf insert

The second function to be implemented is insertion within internal nodes

```
TreePtr InternalNode::insert_key(Key key, RecordPtr record_ptr)
```

This insert function should provide two capabilities - (1) It first finds the appropriate child to insert the <key, record_ptr> into and (2) If the chosen child returns a split node, then the insert function should include this new node as a new child; if the inclusion of the new child causes a split then the new split node should be returned.

```
When internal node n is split to create a new node split_node, node n
retains the first ⌈f/2⌉ are inserted into split_node.
```

To test this function, run the set of commands from internal_insert. The expected output is

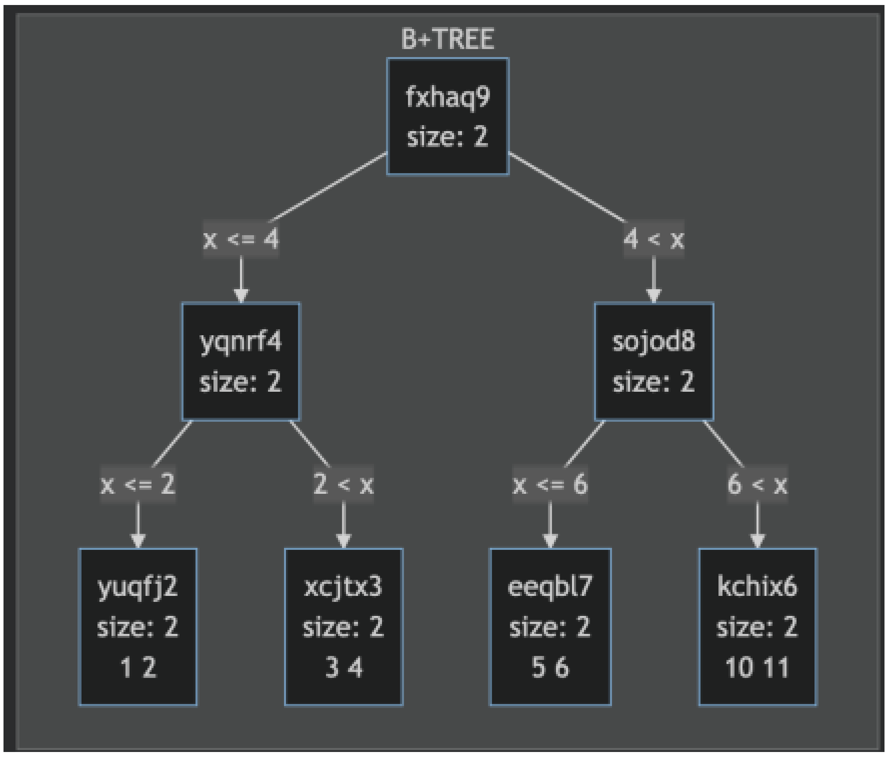

Figure 7: internal insert

Part (b) - Deletion

Deletion is a little more complicated than insertion. Again there are two functions to be implemented - deletion in the leaf nodes and internal nodes. When inserting, an overflow can occur whereas when deleting an underflow can occur. To handle underflows that violate the properties of B+Trees there are broadly two options - (1) redistribute the data in the node with a sibling so they both have at least ⌈*f/2*⌉ children or (2) merge with a sibling such that together they have at most *f* children.

```
In this project strictly follow this preference order

1. If left sibling exists, and redistribution can occur - perform
redistribution with left sibling else
2. If left sibling exists, and merging can occur - perform merge with left
sibling else
3. If right sibling exists, and redistribution can occur - perform
redistribution with right sibling else
4. If right sibling exists, and merging can occur - perform merge with
right sibling

When node n underflows and is redistributed with sibling node sib_node,n
takes just enough children from sib_node to have exactly ⌈f/2⌉    children.
```

```
TreePtr LeafNode::delete_key(Key key)
```

Merging and redistribution will happen through the parent node - therefore, deletion in the leaf node is simple - if key exists, delete <key, RecordPtr> pair. To test use leaf_delete

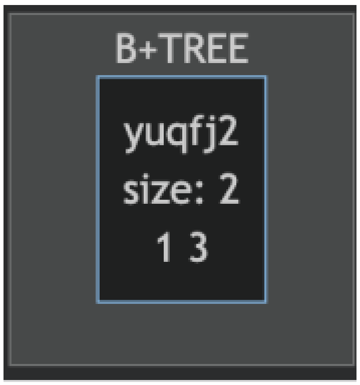

Figure 8: leaf_delete

Deletion in the internal nodes is a little complex, again there are two tasks - (1) find the appropriate child to delete the key from and (2) check if the chosen child now underflows, if it underflows then perform the appropriate redistribute/merge operations as per the preference detailed above.

```
TreePtr InternalNode::delete_key(Key key)
```

To test your code use the following files -internal_delete1,internal_delete2,internal_delete ,internal_delete4. This list is not exhaustive, you are encouraged to write your own cases.

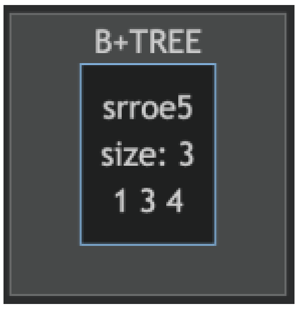

Figure 9: internal_delete

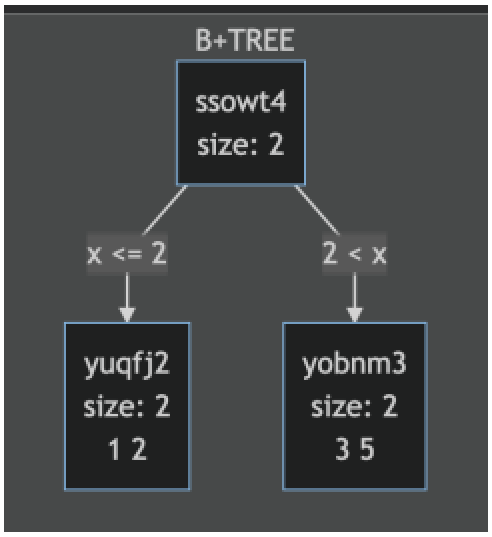

Figure 10: internal_delete

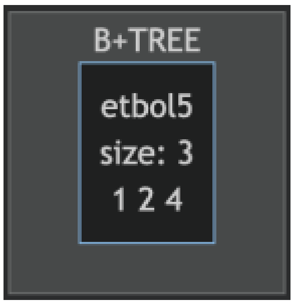

Figure 11: internal_delete

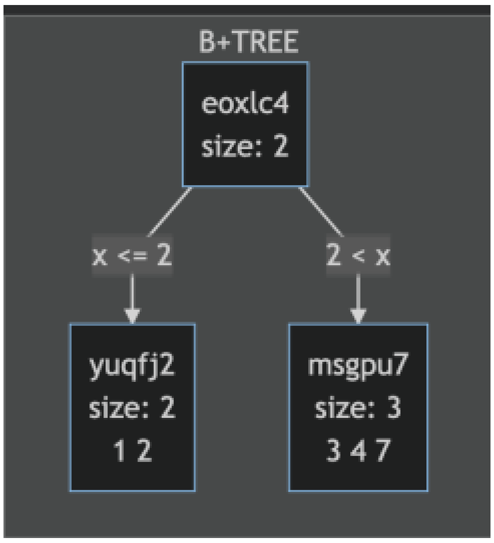

Figure 12: internal_delete

Part (c) - Analysis

The RANGE command has been implemented for you. When RANGE is called, it prints the number of node accesses needed when the B+Tree is used and when it is not as space-seperated integers. In this section you have to

1. Generate a source file where you insert numbers 1-100 in some random order

2. Plot the distribution of block accesses needed with and without the B+Tree when calling RANGE i i+1 on every valid i as two seperate traces (with and without)

Along the x-axis - number of block accesses and along y - either the count or the percentage. Submit this plot as a pdf in the data folder of your repository

Provide this graph for any 3 different values of FANOUT.

Submission Instructions

All your code should be pushed to your GitHub repository. At the deadline, your codebase will be automatically downloaded.

For any doubts please use the doubts document exclusively.

This course is intolerant of plagiarism. Any plagiarism will lead to an F in the course.

References

B+Tree deletion
Virtual Functions