Introduction

A database is a collection of information that is organized so that it can be easily accessed, managed and updated. A structured set of data is organized into rows, columns and tables, and it is indexed to make it easier to find relevant information. Data gets updated, expanded and deleted as new information is added. Databases process workloads to create and update themselves, querying the data they contain and running applications against it.

A B-tree is a self-balancing tree data structure that maintains sorted data and allows searches, sequential access, insertions, and deletions in logarithmic time. The B-tree is a generalization of a binary search tree in that a node can have more than two children. Each node can have up to m children, where m is called the tree's "order". To keep the tree mostly balanced, we also say nodes have to have at least m/2 children (rounded up). Unlike self-balancing binary search trees, the B-tree is well suited for storage systems that read and write relatively large blocks of data, such as discs.

Using B-tree as data structure for designing database is good as

- Searching for a particular value is fast (logarithmic time)
- Inserting / deleting a value you've already found is fast (constant time to rebalance)
- Traversing a range of values is fast (unlike a hash map)

About the project

- Implementing a Database Using B Tree.
- Using B Tree Node as Table's data.
- Creating our own Language for creating, storing and retrieving data.
- Read and write the B Tree structure in a file.

B Tree Structure

```
struct btnode {
        int degree;
        struct row key_data[2 * DEG - 1];
        fpos_t childs[2 * DEG];
        int curr_nodes;
        int is_leaf;
};

struct table {
        char table_name[20];
        int number_of_column;
        char columns[10][15];
        int data_type[10];
        int is_root;
        fpos_t rootpos;
};
```

```
union type {
        int int_data;
        char char_data[30];
        float float_data;
};


struct row {
        int index;
        fpos_t data;
};
```

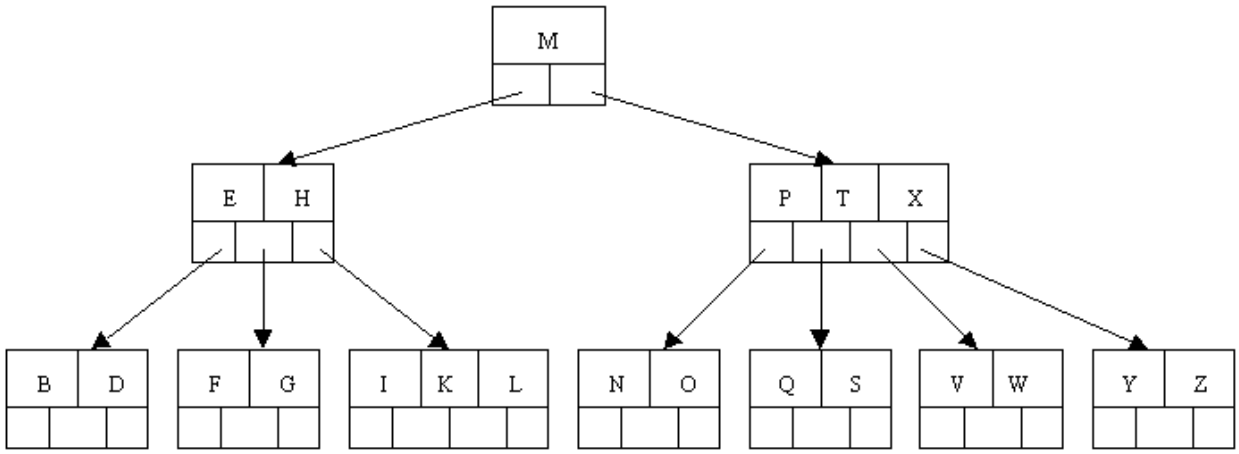


Fig: B Tree representation

Implementation

We have implemented a simple database which stores single table using B Tree. 3 data types are supported for column data i.e. integer, string & float. Also there can be a maximum of 10 columns in the table. The first column is treated as the primary key which should always be of integer type. We have implemented the functions like Create table, insert in table, get all table data and get specific data based on primary key.

The programs starts by first reading a file named **data.bin** which may or may not be present. In case if it is not present we create a new file else we read the data which is already present. Suppose the file is not present (user is using for the first time) the user will create a new table, insert the data, etc. For this we are storing a structure named **table** at the start of the file. This structure is used as meta-data where all the attributes regarding the table are saved, e.g. table name, number of columns, data types of columns and also the root of B Tree.

Initially the root is empty but as soon as first data is inserted we create a B tree node named as **BTnode**, save it in file and assign its position to the root position. This B tree node contains keys along with row data and position of child nodes in the file. Row data is stored as a structure named as **row** which consists of an array of union. Here union is used for different data types. The position of children is saved using the **fpos_t** data structure which is the type of an object that can encode information about the file position of a stream.

The below are the functionalities with syntax which have been implemented.

1. **CREATE TABLE {*TABLE_NAME*} ({*Column_1*} INTEGER, {*Column_2*} {*DATA_TYPE*}, {*Columns…*})**

   This is the syntax to create a new table. We have to provide the table name, and columns with their data type. The first column should always be of **INTEGER** type which will be the primary key of the table and also will be the key for searching in the B tree. There can be a maximum of 10 columns (including the primary column) in a table.
   Note: If there is a table already present then executing this statement will clear the previous table and its data.

   Example: CREATE TABLE Records (RollNo INTEGER, Name STRING, Marks FLOAT)

2. **INSERT {*TABLE_NAME*} ({*COLUMN_DATA_1*}, {*COLUMN_DATA_2*}, …)**

This is the insert statement used to insert data into the table. Here we have to provide the row data in the order the columns of the table were created.

Note:

- If the table is not created and the user tries to insert then the system will show an error message as ***Table not present***.
- If the primary already exists then the system will show an error message as ***The primary key already exists***.

When this insert statement is executed in the background we are inserting the data at proper position after traversing the B tree from root (this includes traversing the child nodes and splitting of nodes if required). If the root is empty means this is the first insert statement then a new B tree node is created and assigned as the root of B tree.

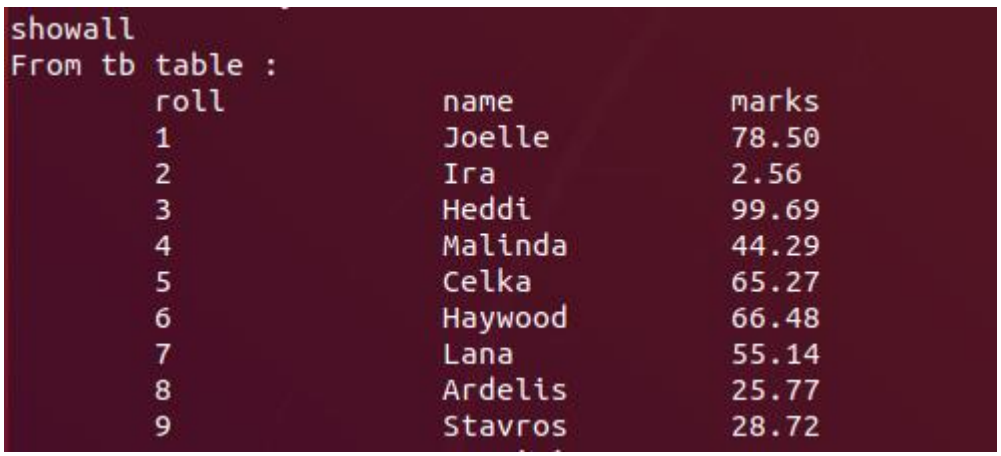
Example: INSERT Records (1, Joelle, 78.50)

3. **SHOWALL**

The SHOWALL statement lists out all the entries present in the table.

Note: If there are no entries in the table or the table is not present then an error message will be displayed as ***No records found.***

Example: SHOWALL



4. **SHOW WHERE {*PRIMARY_KEY*}**

This statement is used to get the data specific to a particular primary key.

Note: If the primary key is not found then the system will show an error message as ***Primary key not found***.

Example: SHOW WHERE 5

```
show where 5
From tb table where primary key = 5
        roll            name            marks
        5               Celka           65.27
```

5. **UPDATE {*PRIMARY_KEY*} ({*Column_data_2*}, {*Column_data_3*}, …)**

The above statement is used update the row data based on primary key. The user has to provide all the columns data except the primary key. The primary key will remain same, rest of the column values will get modified.

Note:

- Executing this statement will modify the data only if the primary key passed is present and will show the message ***Updated successfully***.
- If the primary key is not found then the system will show ***Primary key not found***.

Example: UPDATE 20 (chester, 70.11)

```
show where 5
From tb table where primary key = 5
        roll            name            marks
        5               Celka           65.27
update 5 (chester, 70.11)
Updated successfully!!
show where 5
From tb table where primary key = 5
        roll            name            marks
        5               chester         70.11
```

References

Literature Review from

- Introduction to Algorithms, Third Edition by *Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein*.
- Data Structures and Algorithms by *Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman*

Online Web References

- https://en.cppreference.com/w/c/io
- https://stackoverflow.com