B-Trees

Lawrence Cabbabe, Aaron Levesque, Ben Uthoff, Sam Zhong

4 December, 2023

Table of Contents:

Introduction To B-Trees:

B-Trees are a widely used data structure that is designed to efficiently organize and store large amounts of data while facilitating quick search, insertion, and deletion operations. B-tree was introduced by Rudolf Bayer and Edward M. McCreight in a 1970 paper they co-authored while they were working for Boeing Research Labs. They were attempting to create a more efficient way to organize index pages for large random-access files. In their paper about large ordered indices, they state they have been under the impression that indices could contain more data so that less could be stored directly in the main memory. Because of this, B-Trees are well-suited for managing data in storage systems such as databases and file systems.

When naming the new tree system, the chosen name was B-Trees, with no public confirmation on what ‘B’ stood for. When McCreight was asked about its meaning at the 24th Annual Symposium on Combinatorial Pattern Matching in 2013, he answered "you just have no idea what a lunchtime conversation can turn into." Paraphrasing this, he stated that the B doesn't have a specific meaning, B could mean Boeing (where they worked), Bayer (senior author), Balance (tree's job), but to McCreight, he jokingly believes "the more you think about what the B in B-trees means, the better you understand B-trees."

The main feature of a B-tree is its balance, which ensures that the tree remains symmetrical and even, and prevents nodes going down the same side. As a result, B-tree offers a more reliable and consistent performance since even when the size grows, it stays relatively balanced. The primary advantage of B-tree is its ability to optimize disk I/O operations; it minimizes the number of disk accesses required for search operations which makes it well-suited for instances where data is stored on external storage devices. The efficiency provided by B-tree is a big advantage for accessing a large amount of data while maintaining the performance.

To conclude, B-tree is a fundamental data structure that plays a crucial role in the design and implementation of high-performance storage systems, databases, and file systems. The balancing and disk access aspects make B-tree very efficient for managing large datasets and supporting rapid data retrieval and modification.

Introduction To Our Project:

This project is the implementation of a B-Tree into C++, specifically the implementation of a string based B-Tree with insert and search. A B-Tree, as mentioned above, is a data structure primarily used in databases for quicker results. This implementation was made with the intent to be a faster dictionary, where words (as strings) could be inserted, searched, and found, as well as removed. This dictionary could be populated with a variety of different types of information such as words, numbers, IP addresses, Personal Identifiable Information, and so on.

A harder structure to implement compared to Binary Search Trees or Red Black Trees, B-Trees run faster in almost every case because of the unique way they store information in nodes, not being limited in the same way binary storage would be. We understood the challenges ahead of us and attempted to exhibit ambition, going to implement a delete function on top of the requirements.

We specifically chose this project because of our combined interest in data storage, their algorithms, and their different results. As Computer Science students, we have an innate interest in all things technology. Talking specifically about us, we all learned a lot about how storage is done on the operating systems we grew up with, whether it be MacOS or Windows XP, 7, 8 or 10. Getting into modding video games or searching through program files to find the save folder for cache data to simply delete it, these “adventures” excited us. Now as college students and colleagues, we were able to bond over our upbringings, came together as a team, and chose B-Tree implementation

as our project. With the experience of creating Binary Search Trees and Red Black Trees already, we believed the next step was to get down to the basics of B-Trees as they are used all around us today, with D.N.S, secondary storage devices, and other forms of data storage and retrieval. Additionally, B-Trees have an extensive web of structures built off them such as B+ Trees, Maple Trees, and their ability to be run paralleled, all allow for future understanding of B-Trees and would be an incredible basis for our understanding.

Methods:

As mentioned in the introduction, a B-tree is commonly used in databases and file systems to store and manage large amounts of data; it is designed to provide efficient searching, insertion, and deletion operations. Its ability to maintain its balance with larger datasets make it efficient. The structure of a B-tree consists of nodes, and each node has multiple keys and pointers to child nodes; the number of keys in each node is between a minimum and a maximum value, which ensures the balance in the tree. The order of a B-tree is the maximum number of children each node can have, and a B-tree of order **t** will have at most **2t - 1** keys and at least **t - 1** keys in each non-root node.

The structure starts with a root node which may have as few as one key or may have multiple keys. To insert a key into the B-tree, it starts at the root and traverses the tree to find the appropriate position for the key, if the node is not full, insert the key into the node in its correct position, and if the node is full, split the node into two, and promote the middle key to the parent node, and repeat the process with parent node if it is needed. When a node is split, it involves redistributing the keys and pointers to form two nodes; the middle key is moved up to the parent node, the keys to the left of the middle key stay in the original node, and the keys to the right of the middle key moves to the new node; the pointers adjusts accordingly. When a key is promoted to the parent, it is inserted into the parent node, and if the parent node is already full, it is split similarly, and the process may propagate up the tree; If the root node is split, a new root is create, and the height of the tree increases by 1. There are a lot of steps when inserting

a value into the B-tree, because there are many factors to be considered to keep the tree balanced.

Another key feature in B-tree is deletion, which is used to remove a key from the tree. To delete a key, it starts at the root and traverses the tree to find the key; when the key is found, if the key is a non-leaf node, it finds its predecessor or successor in the tree, and replaces the key with its predecessor or successor and then deletes the predecessor or successor.

As we see with deletion and insertion, there are a lot of steps that could cause a split or a merge; when a split or a merge happens, the pointers also need to be adjusted accordingly to maintain the tree structure. Maintaining the balance property of the B-tree is crucial because it ensures that the depth of the tree remains relatively controlled.

Implementations:

In our implementation, we included some functions that are essential to the structure of B-tree. Let's first start with the header file and then go on the cpp file. The header file for BTreeNode includes a bool for if it's a leaf, a vector of strings for the keys, and a vector of nodes for the children, and other function prototypes for the required functions.

```
class BTreeNode {
private:
    bool is_leaf;
    std::vector<std::string> keys;
    std::vector<BTreeNode*> children;
```

The BTree class contains the root of the tree, and the degree of the tree, and the rest are function prototypes.

```
class BTree {
private:
    BTreeNode* root;
    int degree;
```

Now we look at the cpp file. We first started with the constructor of the B-tree structure, which initializes the where the root of the tree is, and initializes the degree of the tree, which is entered by the user.

```
BTree::BTree(int _degree){
    this->root = nullptr;
    this->degree = _degree;
}
```

After the construction of the Btree, we also incorporated a constructor for if the node is a leaf which is important for keeping B-trees balanced structure.

After finishing the constructors, we moved to the insert function, but when working on the insert function, we ran into the problem of needing other functions to work with the insert function. Let's first start with the insert function.

```
void BTree::insert(std::string& key) {
    // If there is no root...
    if (root == nullptr) {
        root = new BTreeNode(true);
        root->keys.push_back(key);
        return;
    }
    // If the root node is reaches max...
    if (root->keys.size() == (2 * degree - 1)) {
```

The **insert** function is the entry point for inserting a key into the B-tree. There are two conditions to be found in this function; If the tree is empty, which is defined by if root == nullptr, it creates a new root node and inserts the key. Otherwise it will first check the size of the node and make sure it doesn't exceed **2t - 1**, and if it does, we call the **splitChild** function to balance the tree structure, and if it doesn't, we will call the **insertNonFull** function to handle the insertion in the non-empty tree. As we see in the insert function, it works with the **splitChild** function and **insertNonFull** function inorder to keep its B-tree structure. Next we will look at the **splitChild** function.

```
void BTree::splitChild(BTreeNode* parent, int index) {
    BTreeNode* new_child = new BTreeNode();
    BTreeNode* old_child = parent->children[index];
    parent->keys.insert(parent->keys.begin() + index, old_child->keys[degree - 1]);
    parent->children.insert(parent->children.begin() + index + 1, new_child);

    new_child->keys.assign(old_child->keys.begin() + degree, old_child->keys.end());
    old_child->keys.resize(degree - 1);

    if (!old_child->is_leaf) {
        new_child->children.assign(old_child->children.begin() + degree, old_child->children.end());
        old_child->children.resize(degree);
    }

    parent->is_leaf = false;
    if (new_child->children.size()) new_child->is_leaf = false;

}
```

As mentioned earlier, the **splitChild** function is called when a non-leaf node's child becomes full during an insertion operation; the main purpose of this function is to split the full child into two nodes. The parameters, "BTreeNode* parent" is the parent node of the full child, and "int index" is the index of the full child in the parent's children vector. The function creates a new child node, "new_child", which is another node in the

structure. Then it takes the middle key from the full child, “old_child”, and inserts it into the parent node. Lastly, it adjusts the keys and children vectors of bothe the old child and the new child to reflect the split, and update the if the node is a leaf in the node creates, “new_child”. Next let’s look at the **insertNonFull** function.

```
void BTree::insertNonFull(BTreeNode* node, std::string& key) {
    // Start at the end of the node...
    int i = node->keys.size() - 1;
    // Go through each key until you find a place to put the new one...
    while (i >= 0 && key < node->keys[i]) {
        i--;
    }
    i++;

    // If the node is a leaf...
    if (node->is_leaf) {
        // Add the key to the node.
        node->keys.insert(node->keys.begin() + i, key);

        // The node is not a leaf...
    } else {
        if (node->children[i]->keys.size() == (2 * degree - 1)) {
            splitChild(node, i);
            if (key > node->keys[i])
                i++;
        }
        insertNonFull(node->children[i], key);
    }
}
```

The **insertNonFull** function is a recursive function with the purpose of inserting a key into a B-tree node that is not full, and for instances where the node is full, it will call the

splitChild function addressed earlier. The parameters “BTreeNode* node” is the current node being processed, and “std::string& key” is the key to be inserted. In detail, the function first finds the correct position for the key in the sorted list of keys in the node, and if the node is a leaf, it recursively calls itself on the appropriate child of the current node; if the child is full, it triggers a split operation using the **splitChild** function. After completing all functions that get insert working, we tested the code, but realized we had no way to visualize the tree, so the next function was a **generateDotFile** function for visualizations of the tree; it is a function that generates a DOT file that lets us see the tree after we run our code.

```
void BTree::generateDotFile(std::string filename) {

    std::ofstream dotFile(filename);
    if (!dotFile.is_open()) {
        std::cerr << "Error opening DOT file." << std::endl;
        return;
    }

    dotFile << "digraph BTree {\n";
    dotFile << "node [shape=record];\n";

    std::string nodes;
    std::string arrows;

    // Recursively draw each node.
    int node_count = 0;
    dotFileHelper(root, node_count, nodes, arrows);

    dotFile << nodes << arrows;

    dotFile << "}\n";
    dotFile.close();
```

The main purpose of the **generateDotFile** function is to generate a DOT file representation of the B-tree and create a file and write to it. The function takes one parameter “std::string filename”, and it is the name of the DOT file to be generated. In detail, the function opens the output file with error checking, and it writes the initial lines for the DOT file which includes the graph name and node shape. Then it calls the recursive helper function, **dotFileHelper**, to traverse the tree and generate node and arrow information, and then closes the DOT file. Now let’s look at the **dotFileHelper** function.

```
void BTree::dotFileHelper(BTreeNode* node, int& node_count, std::string& nodes, std::string& arrows) {
    // Add Node to string.
    nodes = nodes + "    node" + std::to_string(node_count) + " [label=\"";
    for (int i=0; i < node->keys.size(); i++) {
        nodes = nodes + node->keys[i] + " | ";
    }
    nodes = nodes.substr(0, nodes.size() - 3);
    nodes = nodes + "\"";
    if (node->is_leaf) nodes = nodes + " color=\"#62b362\"";
    nodes = nodes + "];\n";

    // Add child nodes recursively and their children.
    int parent = node_count;
    for (int i=0; i < node->children.size(); i++) {
        node_count++;
        arrows = arrows + "    node" + std::to_string(parent) + " -> node" + std::to_string(node_count) + ";\n";
        dotFileHelper(node->children[i], node_count, nodes, arrows);
    }

}
```

The **dotFileHelper** function is a recursive function that traverses the B-tree and generates information for each node and arrow in the DOT file. The function takes in four parameters; the “BTreeNode* node” is the current node being processed, the int& node_count is a reference to the count of nodes encountered so far, the std::string& nodes is a reference to a string that accumulates node information, and the

std::string& arrows is a reference to a string that accumulates arrow information. In detail, the function first adds information for the current node to the “nodes” string, and if it’s a leaf, we include its label, and some color. Then it recursively calls itself for each child of the current node, updating the “node_count” and adding arrow information to the “arrow” string; the label of each node is created by concatenating the kays of the node with the separators “|” to match the syntax of the DOT file. After seeing a visual of the B-tree, the next function we decided to work on is the **search** and **searchHelper** functions.

```
int BTree::search(std::string& key) {
    int count = 0;
    searchHelper(root, key, count);
    return count;
}
```

The **search** function really isn’t a lot, it just initializes the count at zero, and calls the helper function which is where all of the search happens.

```
void BTree::searchHelper(BTreeNode* node, std::string& key, int& count) {
    // Look through each key in node.
    for (int i=0; i < node->keys.size(); i++) {
        if (key < node->keys[i]) break;
        if (key == node->keys[i]) count=count+1;
    }

    // Check if there are no children.
    if (node->is_leaf == true) return;

    // Place before the first child.
    if (key <= node->keys[0])
        searchHelper(node->children[0], key, count);
    // Place inbetween two keys in the node.
    for (int i=1; i < node->children.size()-1; i++) {
        if (key >= node->keys[i-1] && key <= node->keys[i]) {
            searchHelper(node->children[i], key, count);
        }
    }
    // place after the last child
    if (key >= node->keys[node->keys.size()-1])
        searchHelper(node->children[node->children.size()-1], key, count);

}
```

The **searchHelper** function recursively traverses the B-tree to search for a ekyt and counts its occurrences; It iterates through each key in the current node, and if teh key is less than the current key, it breaks the loop, and if the key matches the current key, it increments the count by 1. Then it checks if the node is a leaf, if it is, the function returns, as there are no further children to explore; if the key is less than or equal to the first key in the node, it recursively calls itself with the first child of the node,

iterates through the remaining keys in the node and recursively calls itself for the appropriate child based on the key's position; if the key is greater than or equal to the last key in the node, it recursively calls itself with the last child of the node.

With the above functions, we have covered the basic functions required to build a B-tree, insert any element inside of the B-tree, generate a DOT file to visualize the B-tree, and search how many of a specific element are inside of a B-tree. Next, we've decided to explore another key function of the B-tree structure which is the **remove** function. We first had a **removeStarter** function which initiates the removal process by calling the recursive **remove** function with the root of the B-tree and the key to be removed.

```
void BTree::removeStarter(std::string& key) {

    // Run the actual remove function
    remove(root, key);
    // Check if the root is out of keys, then set the root to the child.
    if (root->keys.empty() && !root->is_leaf) {
        BTreeNode* newRoot = root->children[0];
        delete root;
        root = newRoot;
    }
}
```

Next, we implemented the **remove** function which recursively removes the specified key from the B-tree. The function finds the index of the key in the current node, and if the key is found in a leaf node, it calls the **removeFromLeaf** function to the delete the key from the leaf, and if the key is not in a leaf node, it proceeds to handle the removal in a

non-leaf node, and considers cases like merging and borrowing keys from siblings. It also handles the special case where the root becomes empty after removal.

```
void BTree::remove(BTreeNode* node, std::string& key) {
    // set the index.
    int index = node->findKeyIndex(key);

    // The key exists in this node AND the current node is a Leaf.
    if (index < node->keys.size() && key == node->keys[index]) {
        removeFromLeaf(node, index);
    } else {
        // The Key was not found in this node; continue...

        // Check if the current node is root or not a leaf then go to the correct child node.
        if (!node->is_leaf) {
            bool lastChild = (index == node->keys.size());
            BTreeNode* child = node->children[index];

            // The child does not have the minimum keys.  Merge with a sibling to resolve.
            if (child->keys.size() < degree) {
                becomeOneWithChild(node, index);
            }

            // The child was merged, go to the child.
            if (lastChild && index > node->keys.size()) {
                index--;
            }

            // Remove the child.
            remove(node->children[index], key);
        } else {
            // Not found
            //std::cout<<"NOTHING FOUND<<std::endl;
        }
    }
}
```

The **removeFromLeaf** function is just a simple function that removes a key from a leaf node.

```
void BTree::removeFromLeaf(BTreeNode* node, int index) {
    //deletes the keys to remove from the leaf.
    node->keys.erase(node->keys.begin() + index);
}
```

The next function to look at is the **becomeOneWithChild** function, which is called in the **remove** function when after removing, the node contains less keys that it requires by the degree.

```
void BTree::becomeOneWithChild(BTreeNode* parent, int childIndex) {
    BTreeNode* sibling;
    if (childIndex > 0) {
        sibling = parent->children[childIndex - 1];
    } else {
        sibling = nullptr;
    }

    // Take from left.
    if (degree <= sibling->keys.size() && sibling) {
        borrowLeft(parent, childIndex);
    } else if (childIndex < parent->keys.size() && parent->children[childIndex + 1]->keys.size() >= degree) {

        borrowRight(parent, childIndex);
    } else if (sibling) { // combine left.
        mergeLeft(parent, childIndex);
    } else { // combine right.
        mergeRight(parent, childIndex);
    }
}
```

The main purpose of this function is to determine whether to borrow, merge with the left or right sibling, or do nothing based on the status of the child node. The function checks if borrowing from the left or right sibling is possible, and if so, calls the appropriate borrowing function, and if borrowing is not possible, it merges with the left or right sibling or does nothing based on the sibling's status. Now let’s look at the borrowing functions, which includes a **borrowRight** and a **borrowLeft**.

```
void BTree::borrowRight(BTreeNode* parent, int childIndex) {
    BTreeNode* child = parent->children[childIndex];
    BTreeNode* rightSibling = parent->children[childIndex + 1];

    // Parent obtains key from right child sibling.
    child->keys.push_back(parent->keys[childIndex]);
    parent->keys[childIndex] = rightSibling->keys.front();
    rightSibling->keys.erase(rightSibling->keys.begin());

    // Child is not leaf, take the pointer away from the right child sibling.
    if (!child->is_leaf) {
        child->children.push_back(rightSibling->children.front());
        rightSibling->children.erase(rightSibling->children.begin());
    }
}
```

These two functions have the same features, but one works on the left and the other works on the right. The main purpose of the burrow functions is to borrow a key from their respective side of a child node. The function moves a key from the parent to the child, and adjusts the keys and children of the child and the respective siblings. Another two functions that's required when reconstructing the tree after the removal are the merge functions.

```
void BTree::mergeLeft(BTreeNode* parent, int childIndex) {
    BTreeNode *child = parent->children[childIndex];
    BTreeNode *leftSibling = parent->children[childIndex - 1];

    // key moves from parent to left sibling.
    leftSibling->keys.push_back(parent->keys[childIndex - 1]);

    // All keys and points are shifted to the left.
    leftSibling->keys.insert(leftSibling->keys.end(), child->keys.begin(), child->keys.end());
    leftSibling->children.insert(leftSibling->children.end(), child->children.begin(), child->children.end());

    // Delete the key and child pointer from the parent.
    parent->keys.erase(parent->keys.begin() + childIndex - 1);
    parent->children.erase(parent->children.begin() + childIndex);

    // delete the child.
    delete child;
}
```

```
void BTree::mergeRight(BTreeNode* parent, int childIndex) {
    BTreeNode *child = parent->children[childIndex];
    BTreeNode *rightSibling = parent->children[childIndex + 1];

    // Key moves from the child back to the parent.
    child->keys.push_back(parent->keys[childIndex]);

    // All keys and pointers move to the child.
    child->keys.insert(child->keys.end(), rightSibling->keys.begin(), rightSibling->keys.end());
    child->children.insert(child->children.end(), rightSibling->children.begin(), rightSibling->children.end());

    // Delete the key and child pointer.
    parent->keys.erase(parent->keys.begin() + childIndex);
    parent->children.erase(parent->children.begin() + childIndex + 1);

    // delete the child right sibling
    delete rightSibling;
}
```

The main purpose of the merge functions is to merge the child node with its respective siblings in the parent. The function moves a key from the parent to their respective siblings and merges the keys and children of the respective siblings and childs, and then deletes the child. The two functions are required to keep the structure balanced, and reconstruct the tree when out of place.

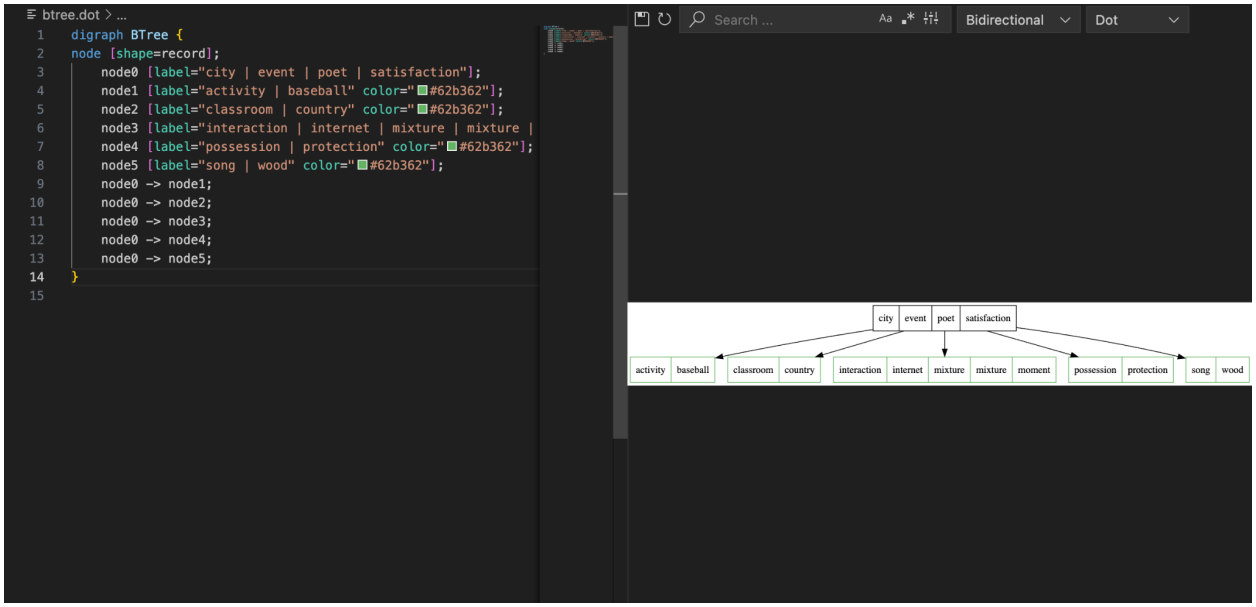
After giving the remove function some test cases, we saw some great results in that it was working to remove the keys we want, and it is also reconstructing the tree at the same time. It works for most cases, but we found that for some obscured cases, it will crash. Our belief is that the program does not move back through the nodes, so it will get stuck on some child or sibling node and can no longer search and remove other words.

The last thing we did with our implementation was we incorporated error detection and user interaction in our main.cpp. We gave the user some prompts to answer, and when

the user answers, our program will open the corresponding files and generate the B-tree with the degree and file they decide to use.

```
"Hello!  Welcome to the B-Tree program."<<std::endl;
"What would you like to do?"<<std::endl;
"(Enter the number of the action would like to perform)"
"(1) Search for a word."<<std::endl;
"(2) Insert a word."<<std::endl;
"(3) Delete a word. (Beta/Unstable) "<<std::endl;
"(4) Generate a .DOT file"<<std::endl;
"(0) Close the program"<<std::endl;
```

The user can keep using the program and when they want to quit, they can type zero to close the program. Error detection is also added for when the user tries to open a file that doesn't exist, or type in a non integer for the degree, or type in an invalid number when they are given the options. There is also an visualize extension which allows the user to see the dot file within the idle.



Contribution:

Group Member	Contribution
Lawrence Cabbabe	● Work done on slideshow presentation and research paper. ● Research on History of B-Trees. ● Implementation of Menu and creating test cases. ● Attempted Implementation of Delete Functions. ● Archived attempt at Implementation of Insert Functions. ● Archived attempt at utilizing templates for B-Tree implementation.
Aaron Levesque	
Ben Uthoff	
Sam Zhong	

Conclusion:

In conclusion, our exploration of B-tree exposed to us its significance as a fundamental data structure in the aspects of high-performance storage systems, databases, and file systems. Introduced by Rudolf Bayer and Edward M. McCreight in 1972, B-tree offered a balanced and efficient solution for managing large datasets, ensuring reliable search, insertion and deletion operations.

The key feature of B-tree lies in its ability to maintain balance, which prevents nodes from leaning towards one side, which is a key problem in binary search trees, which motivated B-tree; its ability to maintain balance also ensures consistency in performance even as the dataset size grows. This balancing feature is achieved through a hierarchical structure where each node contains a specific number of keys and pointers to its children, arranged in ascending order; the order of the B-tree determines the maximum number of children each node can have. Depending on the degree **t**, all nodes may contain at most **2t – 1** keys, and the number of children of a node is equal to the number of keys in it plus 1.

Our project delves into the functions of implementing and utilizing B-trees, particularly in databases and file systems. We mainly focused on the steps of the insert and search functions within the B-tree structure, emphasizing the importance of maintaining balance during these operations. With every insert, the balance of the B-tree may change, so our splitChild function plays a crucial role in maintaining the balancing structure of a B-tree. The dot file function is also important for the visualization of B-tree

after it is inserted; with the dot file, we discovered an extension named graphviz preview which allowed us to visualize the B-tree as the dot file is generated.

Through our research on B-tree, we recognized the importance of B-trees towards some pivotal real life applications. In the realm of data storage and retrieval, B-trees play a pivotal role in file systems, ensuring efficient organization and quick access to files and directories; database management systems leverage B-trees for indexing, facilitating rapid searches and retrieval of records. Search engines are another real life application of the B-tree data structure, managing and indexing vast amounts of web page data, which enables obtaining search results at a faster speed. In networking, B-trees contribute to the organization of routing tables, which optimizes the path determination for network traffic. Geographic Information System (GIS), which is a system that creates, manages, analyzes, and maps all types of data also rely on B-trees for spatial indexing, which allows efficient retrieval of geographic data in mapping systems. From compilers organizing symbol tables to operating systems structuring file systems, B-trees showcase its versatility and efficiency across a wide spectrum of real-life applications.

In essence, the B-tree's impact on computer science is lily to endure as long as efficient data organization, storage, and retrieval remains as a challenge in the industry. B-tree's ability to stay balanced which ensures efficiency make them a versatile solution for various applications, and like how Binary Search Tree motivated B-tree, the concepts

and principles of a B-tree may motivate the development of new data structures to adapt to evolving needs of computer science in the future.

References:

"B-Tree Indexes." Www.ibm.com, 20 July 2022, www.ibm.com/docs/en/informix-servers/14.10?topic=indexes-b-tree

"B*-Trees Implementation in C++." GeeksforGeeks, 30 July 2019, www.geeksforgeeks.org/b-trees-implementation-in-c/#

"B Tree in Data Structure: Learn Working of B Trees in Data Structures." EDUCBA, 14 June 2023, www.educba.com/b-tree-in-data-structure/

Bayer, R., and E. M. McCreight. "Organization and Maintenance of Large Ordered Indexes." Acta Informatica, vol. 1, no. 3, 1972, pp. 173–189, infolab.usc.edu/csci585/Spring2010/den_ar/indexing.pdf

"Deep Understanding of B-TREE Indexing." Www.linkedin.com, www.linkedin.com/pulse/deep-understanding-b-tree-indexing-sohel-rana/

"Google Code Archive - Long-Term Storage for Google Code Project Hosting." code.google.com/archive/p/cpp-btree/

"Introducing Maple Trees [LWN.net]." Lwn.net, lwn.net/Articles/845507/.

"Speedb | LSM vs B-Tree." Www.speedb.io, www.speedb.io/blog-posts/02-lsm-vs-b-tree-v2.