SC3020 Database System Principles

Project 1 Report

Group Members:

Name	Matriculation Number	Contribution
Jadhav Chaitanya Dhananjay	U2121503D	Design and Implementation of Storage, Records and B+ Tree Experiments 1-5 Report Writing
Sethi Aryan	U2123583E	Design and Implementation of Storage, Records and B+ Tree Experiments 1-5 Report Writing
Gupta Tushar Sandeep	U2123144E	B+ Tree Deletion Report Writing
Lin Run Yu	U2020490F	B+ Tree Deletion Report Writing

Installation Instructions can be found in README.md in the code folder

Github Repository: https://github.com/indicium15/sc3020-project-1

Record Component Design

As we were presented with the task of storing a relatively large data set, we explored various means by which we could reduce the size of a Record stored on a disk block. A smaller Record structure enables more nodes to be stored on a disk block, minimizing the potential wastage of memory space in our unspanned storage system. Having more records per block also reduces the number of disk I/O reads for ranged queries in our contiguous storage solution.

The tradeoff for optimizing Record storage is that more computational work is required to convert data stored on a disk to a usable structure. Our team however believes that the benefits outweigh the costs in this case. With that in mind, the following are the steps we took to optimize the storage of a Record:

1. **Storing FG_PCT_home, FT_PCT_home, and FG3_PCT_home as float**

As we only need to store values up to three decimal places, we use the smallest data type available in C++ for doing so.

2. **Storing the number of days between 01/01/200 and GAME_DATE as int**

Storing GAME_DATE as a string or date object in C++ would take up a lot of memory. We decided therefore to write a helper function dateToOffset() available in the Record class to return the number of days from 01/01/2000 to the GAME_DATE value read from each record.

```cpp
/**
 * @brief Function that converts date string found in data to an integer offset from 01/01/2000 for storage
 *
 * @param dateString Input string for conversion
 * @return int Date offset from 01/01/2000
 */
int Record::dateToOffset(const string &dateString)
{
    // Set starting date 01/01/2000
    chrono::system_clock::time_point startingDate = chrono::system_clock::from_time_t(946684800); // UNIX timestamp for January 1, 2000

    // Parse input date string with the format given in the data file
    tm tmDate = {};
    istringstream dateStream(dateString);
    dateStream >> get_time(&tmDate, "%d/%m/%Y");
    chrono::system_clock::time_point inputDate = chrono::system_clock::from_time_t(mktime(&tmDate));

    // Calculate the date offset from starting date
    days dateOffset = chrono::duration_cast<days>(inputDate - startingDate);
    int daysSinceEpoch = dateOffset.count();
    return daysSinceEpoch;
}
```

Consequently, we provide a helper function offsetToDate() that calculates and returns the original string value stored in games.txt. This function can be used when displaying information about records or for further computation.

```
/**
 * @brief Function to convert an offset to a date string for displaying information
 *
 * @param offsetInDays Integer value stored in database
 * @return string Date string used for display
 */
string Record::offsetToDate(int offsetInDays)
{
    // Set starting date 01/01/2000
    chrono::system_clock::time_point startingDate = chrono::system_clock::from_time_t(946684800); // UNIX timestamp for January 1, 2000

    // Calculate date from offset in database
    chrono::system_clock::time_point calculatedDate = startingDate + days(offsetInDays);

    // Convert date to a string with same format as input file
    time_t calculatedTime = chrono::system_clock::to_time_t(calculatedDate);
    tm tmDate = *localtime(&calculatedTime);
    ostringstream dateStream;
    dateStream << put_time(&tmDate, "%d/%m/%Y");
    return dateStream.str();
}
```

3. **Storing the offset of TEAM_ID as uint8_t**

Further analysis of our data revealed that the range of values for TEAM_ID is between 1610612737 and 1610612766. By calculating the offset of the TEAM_ID from the minimum value, we can save three bytes of memory per record by storing the offset as a uint8_t (1 byte) instead of an integer (4 bytes).

The function teamIDToOffset() in the Record class helps to perform this computation. The value 1610612736 is used for subtraction to ensure that teams with TEAM_ID 161061237 have an offset of 1.

```
/**
 * @brief Function that converts team id to a offset from the lowest team ID found in the dataset for storage
 *
 * @param teamID read from file
 * @return uint8_t
 */
uint8_t Record::teamIDToOffset(int fileTeamID)
{
    int difference = fileTeamID - 1610612736;
    uint8_t offset = difference;
    return (offset);
}
```

Similarly, the Record class also provides a function to convert the offset stored in the database to a readable TEAM_ID for displaying information.

```cpp
/**
 * @brief Function that converts team id offset to integer value for displaying
 *
 * @param offset read from database
 * @return int
 */
int Record::offsetToTeamID(uint8_t offset)
{
    int difference = static_cast<int>(offset);
    return (1610612736 + difference);
}
```

4. **Storing PTS_home, AST_home and REB_home as uint8_t**

Similar analysis on these columns revealed that the range of values of these columns in games.txt correspond to the range of the uint8_t data type. This enables us to save 9 bytes of memory space in total across these three fields.

Other variables stored in a Record include:

1. **blockAddress** - base address of the database block that a Record is stored in. This is assigned when a Record is pushed to the database.
2. **offset** - offset of the record from blockAddress. Similarly assigned when a Record is pushed to the database.
3. **homeTeamWins** - corresponds to the value of HOME_TEAM_WINS read from games.txt

The resulting variables, data types, size and byte offset of the Record structure can be found in Table 1 below. The original size of the Record is 35 bytes. The C++ compiler adds 5 bytes for each defined Record structure to maintain memory padding.

Variable	Data Type	Size	Offset
fgPct	float	4 bytes	0
ftPct	float	4 bytes	4
fg3Pct	float	4 bytes	8
gameDate	int	4 bytes	12
blockAddress	uchar *	8 bytes	16
offset	int	4 bytes	24
recordID	unsigned short int	2 bytes	28
teamID	uint8_t	1 byte	30

pts	uint8_t	1 byte	31
ast	uint8_t	1 byte	32
reb	uint8_t	1 byte	33
homeTeamWins	bool	1 byte	34
C++ Compiler Padding	-	5 bytes	-
Total Record Size:		40 bytes	

The following Record constructor is used when reading values from the database:

```
Record::Record(float fgPercentage, float ftPercentage, float fg3Percentage, int date, uchar* blockAddress, int offset,
    unsigned short int recordID, uint8_t teamID, uint8_t points, uint8_t assists, uint8_t rebounds, bool homeTeamWins)
    : recordID(recordID), gameDate(date), teamID(teamID), pts(points), ast(assists), reb(rebounds),
      fgPct(fgPercentage), ftPct(ftPercentage), fg3Pct(fg3Percentage), homeTeamWins(homeTeamWins), offset(offset),
      blockAddress(blockAddress){};
```

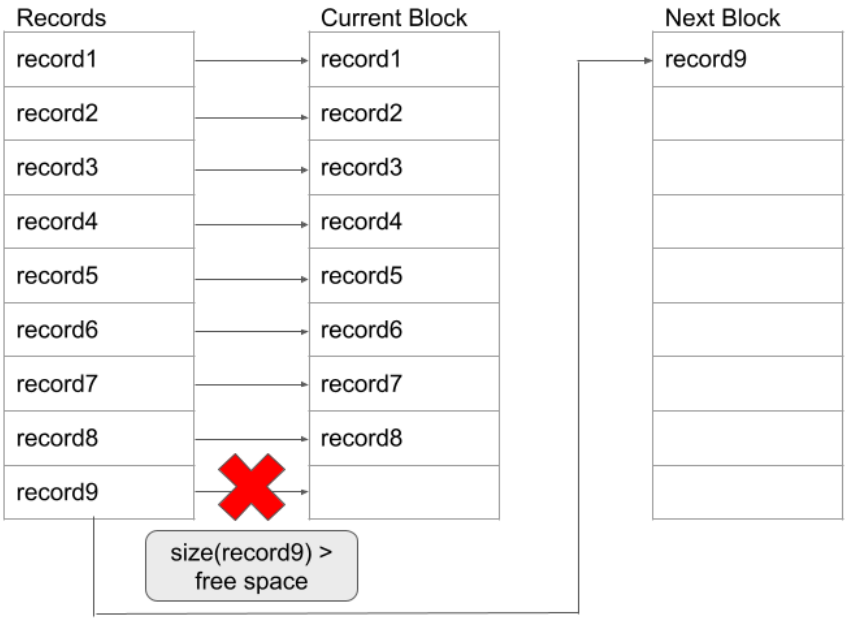
The following Record constructor is used while reading values from games.txt, using the helper functions defined earlier to instantiate the object into a data type ready for storing to the database.

```
Record::Record(unsigned short int recordID, string gameDateStr, uint8_t teamID, uint8_t pts, uint8_t reb, uint8_t ast,
float fgPct, float ftPct, float fg3Pct, int homeTeamWins, uchar* blockAddress, unsigned short int offset)
{
    this->fgPct = fgPct;
    this->ftPct = ftPct;
    this->fg3Pct = fg3Pct;
    int gameDateOffset = dateToOffset(gameDateStr);
    this->gameDate = gameDateOffset;
    this->blockAddress = blockAddress;
    this->recordID = recordID;
    this->offset = offset;
    this->teamID = teamID;
    this->pts = pts;
    this->reb = reb;
    this->ast = ast;
    bool homeTeamWinsBoolean = winsToBool(homeTeamWins);
    this->homeTeamWins = homeTeamWinsBoolean;
}
```

Storage Component Design

Unspanned Record System

For the purposes of this project, we chose to implement an unspanned record storage system. Therefore, our database allocates a new block for storing a record if there is insufficient space in the current block, as illustrated in the figure below.



Traditionally, the unspanned record system comes with the downside of memory wastage in each block. However, as we have already taken measures to optimize the storage of a Record, this tradeoff is easier to make.

We focus instead on the benefits of unspanned implementation - each record will always take only one disk I/O to read from the block. By eliminating the case where a record is spanned between two blocks, we reduce the operational complexity and ensure a more consistent level of performance. Given that disk I/O is the bottleneck for most systems and a historical dataset such as this would mostly be used for querying data, unspanned storage is a good choice for our requirements.

Contiguous Data Storage

In order to speed up insertion, deletion and retrieval queries in the experiments for this project, we sorted records based on ascending order of FG_PCT_home before inserting them into the database. Maintaining a contiguous set of data also enables us to perform bulk loading when constructing the B+ tree on our records, minimizing wastage of space in leaf nodes.

Achieving One-Pass Deletion and Insertion

In addition, we also use some optimization when reading, inserting or deleting records from a disk by storing a map of the blockID and offsets to read for all the records we want to read. By doing pre-processing to ensure that we know all the offsets from one blockID, we avoid having to read the block multiple times from disk if we do not have enough memory space. All operations can be carried out therefore in O(n) time complexity.

Storage Class Attributes

Variable Name	Data Type	Description
diskCapacity	uint	Capacity of database in bytes
blockSize	uint	Size of a block in bytes (initialized to 400 for this project)
currentBlock	int	Internal variable that keeps track of the current block that records are being inserted into.
currentBlockSize	int	Internal variable that keeps track of the size of the current block that is being inserted into. Used during insertion to implement our unspanned storage system.
availableBlocks	int	Initialized to the floor division of diskCapacity and blockSize. Used during insertion to check if there is space for new records.
blockRecords	unordered_map<int,int>	Tracks the number of records stored in each block

Storage Class Methods

Storage():

```
Storage::Storage(uint diskCapacity, uint blockSize)
{
    this->diskCapacity = diskCapacity;
    this->blockSize = blockSize;
    this->currentBlock = 0;
    this->currentBlockSize = 0;
    this->availableBlocks = diskCapacity / blockSize;
    this->baseAddress = static_cast<uchar *>(malloc(size: diskCapacity));
    this->databaseCursor = baseAddress;
    this->blockRecords[0] = 0;
    this->recordsStored = 0;
}
```

a) The Storage class constructor allocates memory for the database based on diskCapacity using malloc.
b) It also assigns initial values for the class explained above.

deleteBlock():

```
bool Storage::deleteBlock(int blockID)
{
    uchar *cursor = baseAddress;
    // Navigate to the first memory address of a block
    cursor += (blockID * blockSize);
    uchar zeroBytes[blockSize];              // Create an array of 400 null bytes
    memset(b: zeroBytes, c: 0, len: sizeof(zeroBytes)); // Initialize it to zeros
    if (memcpy(dst: cursor, src: zeroBytes, n: sizeof(zeroBytes)))
    {
        this->recordsStored -= (currentBlockSize / sizeof(Record));
        this->availableBlocks++;
        return true; // Copy to the memory pointed by cursor
    }
    return false;
}
```

a) This function is used to delete a specific block within the database
b) It creates an array of null bytes of the current blockSize.
c) It calculates the memory location of the block to delete and uses ‘memset’ and ‘memcpy’ to overwrite existing data.
d) It updates internal variables recordsStored and availableBlocks accordingly.

deleteRecord():

```
bool Storage::deleteRecord(int blockID, int offset)
{
    uchar *cursor = baseAddress;
    cursor += (blockID * blockSize) + offset;
    uchar zeroBytes[sizeof(Record)];         // Create an array of 40 null bytes
    memset(zeroBytes, 0, sizeof(zeroBytes)); // Initialize it to zeros
    if (memcpy(cursor, zeroBytes, sizeof(zeroBytes))){
        // Decrement the tracker of number of records stored
        this->recordsStored--;
        // If there is only one record on a block, increment the number of available blocks
        if(this->recordsInBlock(blockID) == 1){
            this->availableBlocks++;
        }
        // Update the blockRecords data structure
        this->setRecordsInBlock(blockID, this->recordsInBlock(blockID)-1);
        return true;
    }
    return false;
}
```

a) This function is used to delete a specific record within a block.
b) It creates an array of null bytes corresponding to the size of the Record structure.
c) It calculates the memory location of the record to delete and uses memset and memcpy to erase the data.
d) It updates internal variable recordsStored and availableBlocks if the record deleted was the last record in a block.
e) It returns true upon successful deletion.

allocateRecord():

```
bool Storage::allocateRecord(Record record)
{
    // Error case: no blocks are available
    if (availableBlocks == 0)
    {
        return false;
    }
    // Try to find an available block and get the address of it
    uchar *blockAddress = findAvailableBlock(sizeof(Record));
    if (!blockAddress)
    {
        cerr << "Failed to find an available block" << endl;
        return false;
    }
    // Update record header
    record.setBlockAddress(this->getBlockAddress(this->currentBlock));
    int offset = currentBlockSize;
    record.setOffset(offset);
    record.print();
    // Copy the record data to the memory address
    memcpy(blockAddress, &record, sizeof(Record));
    // cout << "Current block size " << currentBlockSize << endl;
    currentBlockSize += sizeof(Record);
    // Update blockRecords value to keep track
    auto find = blockRecords.find(currentBlock);
    if (find != blockRecords.end())
    {
        find->second += 1;
    }
    recordsStored++;
    // Move the cursor forward by the amount of memory allocated
    databaseCursor = (blockAddress + sizeof(Record));
    return true;
}
```

a) This function allocates memory to store a ‘Record’ object in the database
b) It checks for available blocks and memory space to allocate the record.
c) It updates the record's block address, offset, etc and copies the record data into the allocated memory.
d) It returns true if allocation is successful and false if there is an error in allocation.

findAvailableBlock():

```
uchar *Storage::findAvailableBlock(int recordSize)
{
    // Error case: no blocks are available
    if (currentBlockSize + recordSize > blockSize && availableBlocks == 0)
    {
        return nullptr;
    }
    // Case 1: The new record can fit in an existing block
    if ((currentBlockSize + recordSize) <= blockSize)
    {
        return databaseCursor;
    }
    // Case 2: A new block has to be allocated for the new record
    if (currentBlockSize + recordSize > blockSize && availableBlocks > 0)
    {
        // Unspanned implementation, go past the remaining fields
        databaseCursor += (blockSize - currentBlockSize);
        // Change internal variables to acknowledge the new block
        availableBlocks--;
        currentBlock++;
        currentBlockSize = 0;
        blockRecords[currentBlock] = 0;
        return databaseCursor;
    }
}
```

a) It is a helper function that is used to scan through the database and return the first available memory address for inserting a record. It explores three cases:

b) pointer. If there is enough space:

   i) If there are no available blocks and the new record cannot fit in the current block, a null pointer will be returned to indicate that no block is available.

   ii) If there are available blocks, our method checks if the new record can fit in the current block. If the new record can fit in the current block, our method returns the memory address of the current block.

   iii) If the new record cannot fit in the current block, our method allocates a new block for the new record. Our method then updates the ‘databaseCursor’ pointer to point to the next available memory address in the new block, and updates the `availableBlocks`, `currentBlock`, and `currentBlockSize` attributes to reflect the new block allocation. Our method then returns the memory address of the new block.

readBlock():

```
uchar *Storage::readBlock(int blockID)
{
    // Calculate starting memory address of block from base address of database and block size
    uchar *blockCursor = baseAddress + (blockID * blockSize);
    uchar *copy = new uchar[400];
    memcpy(dst: copy, src: blockCursor, n: blockSize);
    return copy;
}
```

a) It is a function that copies a block to “RAM” and returns the memory of the copied area.
b) It returns the memory address of the copied data.

printBlockRecords():

```
void Storage::printBlockRecords()
{
    for (const auto &pair: const value_type & : blockRecords)
    {
        cout << "Block ID: " << pair.first << ", Num Records: " << pair.second << endl;
    }
}
```

a) It is a function that prints the contents of the blockRecords attribute in the database.

readAllRecords():

```
vector<Record> Storage::readAllRecords()
{
    vector<Record> records;
    for (int blockID = 0; blockID <= currentBlock; ++blockID)
    {
        vector<Record> blockRecords = readRecordsFromBlock(blockID);
        records.insert(position: records.end(), first: blockRecords.begin(), last: blockRecords.end());
    }
    return records;
}
```

a) It is a function that reads all records in the database object and returns them as a vector.

readRecordsFromBlock():

```
vector<Record> Storage::readRecordsFromBlock(int blockID)
{
    vector<Record> records;
    uchar nullBytes[sizeof(Record)];
    memset(b: nullBytes, c: 0, len: sizeof(Record));
    uchar *blockCursor = baseAddress + (blockID * blockSize); // starting memory address of the particular block
    int numRecordsInBlock = recordsInBlock(blockID);
    if (numRecordsInBlock == 0)
    {
        return records;
    }
    for (int i = 0; i < numRecordsInBlock; i++)
    {
        int offset = i * sizeof(Record);
        if (memcmp(s1: blockCursor + offset + sizeof(Record), s2: nullBytes, n: sizeof(Record)) == 0)
        {
            continue;
        }
        else
        {
            float fgPct = *reinterpret_cast<float *>(blockCursor + offset + offsetof(Record, fgPct));
            float ftPct = *reinterpret_cast<float *>(blockCursor + offset + offsetof(Record, ftPct));
            float fg3Pct = *reinterpret_cast<float *>(blockCursor + offset + offsetof(Record, fg3Pct));
            int gameDate = *reinterpret_cast<int *>(blockCursor + offset + offsetof(Record, gameDate));
            uchar *blockAddress = *reinterpret_cast<uchar **>(blockCursor + offset + offsetof(Record, blockAddress));
            int recordOffset = *reinterpret_cast<int *>(blockCursor + offset + offsetof(Record, offset));
            unsigned short int recordID = *reinterpret_cast<unsigned short int *>(blockCursor + offset + offsetof(Record, recordID));
            uint8_t teamID = *reinterpret_cast<uint8_t *>(blockCursor + offset + offsetof(Record, teamID));
            uint8_t pts = *reinterpret_cast<uint8_t *>(blockCursor + offset + offsetof(Record, pts));
            uint8_t ast = *reinterpret_cast<uint8_t *>(blockCursor + offset + offsetof(Record, ast));
            uint8_t reb = *reinterpret_cast<uint8_t *>(blockCursor + offset + offsetof(Record, reb));
            bool homeTeamWins = *reinterpret_cast<bool *>(blockCursor + offset + offsetof(Record, homeTeamWins));
            Record record = Record(fgPercentage: fgPct, ftPercentage: ftPct, fg3Percentage: fg3Pct, date: gameDate, blockAddress,
            offset: recordOffset, recordID, teamID, points: pts, assists: ast, rebounds: reb, homeTeamWins);      You, 1 second ago • Uncommitted changes
            records.push_back(x: record);
        }
    }
    return records;
}
```

a) This function simulates block level access on our storage system by always reading through all bytes for the blockSize.
b) It calculates the memory address of the start of the specified block based on the “baseAddress” and “blockSize”.
c) If there are no records in the specified block, an empty records vector is returned.
d) The function uses memcmp to check if the cursor points to a record that has not been deleted. Using information about the offsets of the record and data types, the function is able to call the record constructor with the corresponding byte data.
e) It returns the ‘records’ vector that contains all the records stored in a block.

recordsInBlock():

```
int Storage::recordsInBlock(int blockID)
{
    auto find: iterator = blockRecords.find(k: blockID);
    if (find != blockRecords.end())
    {
        return find->second;
    }
    return 0;
}
```

a) It is a function that returns the number of records stored in a block by looking up the blockRecords map.
b) It returns the number of records as an integer value.

setRecordsInBlock():

```
void Storage::setRecordsInBlock(int blockID, int value)
{
    auto find: iterator = blockRecords.find(k: blockID);
    if (find != blockRecords.end())
    {
        find->second = value;
    }
    else
    {
        blockRecords[blockID] = value;
    }
}
```

a) It is a function that sets the number of records stored in a specific block in the blockRecords map.
b) If the block already has an entry in the blockRecords map, it updates the value associated with it, else if there's no existing entry for the block it creates a new entry with the provided value.

readRecordsfromAddresses():

```
vector<Record> Storage::readRecordsfromAddresses(vector<Address> addresses)
{
    // Initialize a map to store the indexes we want to read for each blockID
    int blockAccessCount = 0;
    map<int, vector<int>> indexMap;
    for (int i = 0; i < addresses.size(); i++)
    {
        int blockID = getBlockID(blockAddress: addresses[i].blockAddress);
        int index = addresses[i].offset / sizeof(Record);
        if (indexMap.find(k: blockID) != indexMap.end())
        {
            // Push index to existing vector if the key already exists
            indexMap[blockID].push_back(x: index);
        }
        else
        {
            // Create and push a vector with index value if the key does not exist
            indexMap[blockID] = vector<int>{index};
            // indexMap.insert(blockID, vector<int>{index});
        }
    }
    // Value to return, vector of Records
    vector<Record> results;
    // Iterate through all keys of hashmap (block ID we want to read)
    for (const auto &pair: const value_type & : indexMap)
    {
        // Perform a unit reading of each block
        vector<Record> recordsFromBlock = readRecordsFromBlock(blockID: pair.first);
        blockAccessCount++;
        // TODO: Make sure to write in the report how we're reading each block only once to optimise the "I/O operations"
        // Get the indexes we want to read from each block
        vector<int> indexes = pair.second;
        for (int index : indexes)
        {
            // Push each index into the results
            results.push_back(x: recordsFromBlock.at(n: index));
        }
    }
    cout << "Number of Data Blocks Accessed: " << blockAccessCount << endl;
    return results;
}
```

a) This function reads records from specific memory addresses based on the list of the addresses provided in the “addresses” vector.
b) It initializes a map to store the indexes we want to read for each BlockID.
c) The code uses a for loop to iterate through the elements in the addresses vector.
d) After processing all block IDs and indexes, the function prints the number of data blocks accessed (blockAccessCount) to the console.
e) Finally, it returns the results vector, which contains the records that were read based on the provided addresses.

readRecordsfromNestedAddresses():

```
vector<Record> Storage::readRecordsfromNestedAddresses(vector<vector<Address>> addresses)
{
    int blockAccessCount = 0;
    map<int, vector<int>> indexMap;
    for (int i = 0; i < addresses.size(); i++)
    {
        vector<Address> cursor = addresses[i];
        for (int j = 0; j < cursor.size(); j++)
        {
            int blockID = getBlockID(blockAddress: cursor[j].blockAddress);
            int index = cursor[j].offset / sizeof(Record);
            if (indexMap.find(k: blockID) != indexMap.end())
            {
                // Push index to existing vector if the key already exists
                indexMap[blockID].push_back(x: index);
            }
            else
            {
                // Create and push a vector with index value if the key does not exist
                indexMap[blockID] = vector<int>{index};
                // indexMap.insert(blockID, vector<int>{index});
            }
        }
    }
    // Value to return, vector of Records
    vector<Record> results;
    // Iterate through all keys of hashmap (block ID we want to read)
    for (const auto &pair: const value_type & : indexMap)
    {
        // Perform a unit reading of each block
        vector<Record> recordsFromBlock = readRecordsFromBlock(blockID: pair.first);
        blockAccessCount++;
        // TODO: Make sure to write in the report how we're reading each block only once to optimise the "I/O operations"
        // Get the indexes we want to read from each block
        vector<int> indexes = pair.second;
        for (int index : indexes)
        {
            // Push each index into the results
            results.push_back(x: recordsFromBlock.at(n: index));
        }
    }
    cout << "Number of Data Blocks Accessed: " << blockAccessCount << endl;
    return results;
}
```

a) This function is used to read the records from the results of a ranged query from our B+ tree structure. Therefore, it reads records from a nested vector of addresses, where each inner vector represents a set of addresses for a B+ tree value.
b) It initializes a map to store the indexes we want to read for each blockID.
c) Pre-processing is carried out to group all the offsets that are required to be read from one block. This allows us to achieve one pass query.
d) After processing all records, the function prints the number of data blocks accessed (blockAccessCount) to the console.
e) Finally, it returns the results vector, which contains the records that were read from the database.

removeRecordsfromNestedAddresses():

```
int Storage::removeRecordsfromNestedAddresses(vector<vector<Address>> addresses)
{
    int blockAccessCount = 0;
    map<int, vector<int>> indexMap;
    for (int i = 0; i < addresses.size(); i++)
    {
        vector<Address> cursor = addresses[i];
        for (int j = 0; j < cursor.size(); j++)
        {
            int blockID = getBlockID(cursor[j].blockAddress);
            int index = cursor[j].offset / sizeof(Record);
            if (indexMap.find(blockID) != indexMap.end())
            {
                // Push index to existing vector if the key already exists
                indexMap[blockID].push_back(index);
            }
            else
            {
                // Create and push a vector with index value if the key does not exist
                indexMap[blockID] = vector<int>{index};
                // indexMap.insert(blockID, vector<int>{index});
            }
        }
    }
    cout << "Printing blocks and indexes to be deleted" << endl;
    for (const auto &pair : indexMap)
    {
        cout << "Block Number: " << pair.first << ", Indexes : ";
        vector<int> indexes = pair.second;
        // Check if the length is the full block, then just delete the whole block
        if (indexes.size() == (blockSize / sizeof(Record)))
        {
            cout << "Trying to delete block" << endl;
            if (deleteBlock(pair.first) == 1)
            {
                cout << "Block ID " << pair.first << "sucessfully deleted" << endl;
            }
        }
        for (int index : indexes)
        {
            // Push each index into the results
            if(deleteRecord(pair.first,index*sizeof(Record)) == 1){
                cout << "Record deleted" << endl;
            };
        }
        cout << endl;
    }
    return 1;
}
```

a) This function is used to remove the records from the results of a ranged query from our B+ tree structure. Therefore, it reads records from a nested vector of addresses, where each inner vector represents a set of addresses for a B+ tree value.
b) It initializes a map to store the block IDs and indexes of the records to be deleted.
c) It iterates over each vector of Address objects in the input vector. For each Address object, it calculates the block ID and index of the corresponding record, then checks if the block ID already exists in the map.
d) For each block ID in the map, the method checks if the vector of indexes is the full block size.
e) If the vector of indexes is the full block size, the method deletes the entire block.
f) Otherwise, the individual

getBlockAddress():

```
387  uchar *Storage::getBlockAddress(int blockID)
388  {
389      return (baseAddress + (blockID * blockSize));
390  }
391
```

a) This function calculates the memory address of a block based on the given blockID. It uses the formula: baseAddress + (blockID * blockSize).

getBlockID():

```
392  int Storage::getBlockID(void *blockAddress)
393  {
394      return (static_cast<uchar *>(blockAddress) - baseAddress) / blockSize;
395  }
```

a) This function calculates the block ID corresponding to the given memory blockAddress.

getRecordsStored():

```
399  int Storage::getRecordsStored()
400  {
401      return recordsStored;
402  }
```

a) It returns the total number of records stored in the storage.

getBlockUsed():

```
404    int Storage::getBlocksUsed()
405    {
406        return (currentBlock + 1);
407    }
```

a) It returns the number of blocks currently used in the storage.

getAvailableBlocks():

```
409    int Storage::getAvailableBlocks()
410    {
411        return availableBlocks;
412    }
```

a) It returns the total number of available blocks in the storage.

getBlockSize():

```
414    int Storage::getBlockSize()
415    {
416        return blockSize;
417    }
```

a) It returns the size of each block in the storage.

~Storage():

```
419    Storage::~Storage()
420    {
421        free(baseAddress);
```

a) It is called when an instance of the ‘Storage’ class is destroyed.
b) It is used to release the dynamically allocated memory for ‘baseAddress’ using the free function.

Usage of Storage Class in Experiment 1

To store the data about NBA games on the disk, the getRecordsStored() function is used to store records from Record class, which was created after reading every line from our .txt file.

Usage of Storage Class in Experiments 2 and 5

For experiment 2, to insert records into the database, we used findAvailableBlock() function to find the available spaces to store the record. It ensures that the new record can fit in the available block, and allocates a new block if needed.

For experiment 5, to delete records from the database, deleteRecord() function is utilized. The function first reads the block from the database using readBlock(), then deletes the record at the specified offset. The remaining records are then shifted in the block to fill the gap left by the deleted record. It then updates the block headers to reflect the new number of records in the block.

Linear Search Optimization in Experiment 5

Breaking the linear search early helps reduce the computational time and resource usage, especially in cases where the dataset is large and the condition being checked is near the beginning or end of the sorted data. The primary reason for doing this is to reduce unnecessary work and improve the efficiency of the search process when certain conditions are met.

In Experiment 5, the linear search is looking for records with fgPct values less than or equal to 0.35. Since the data blocks are sorted, once a record is found with an fgPct value greater than 0.35, there's no need to continue searching, as all the subsequent records will also have fgPct values greater than 0.35. Thus, breaking the loop in this case optimizes the search process.

B+ Tree Design

In our database management system, we have implemented a B+ Tree structure which provides efficient indexing and retrieval capabilities for large datasets like our NBA games database. This meets the need for a robust and efficient data structure to manage and retrieve games by FG_PCT_home (Field Goal Percentage).

Implementation Details:

To uniquely identify each record in the storage, we created a new structure *Address* which contains the following variables:

- **void *blockAddress**: Address of the first byte of the block
- **int offset**: Offset from the record from the block address

Using this structure helped us in accessing the records from the tree efficiently.

```
struct Address
{
  void *blockAddress;
  int offset;

//Constructor
  Address(void* blockAddress, int offset) : blockAddress(blockAddress), offset(offset) {}
};
```

Node Structure:
We followed the basic B+ tree structure where we have two different types of nodes, internal and leaf nodes. The leaf nodes store actual addresses of the records whereas the internal nodes store keys and pointers to the child nodes.

The Node class consists of following attributes:

- **float *keys**: Array of keys
- **uint8_t numKeys**: Number of keys stored in a node
- **uint8_t maxKeys**: Maximum number of keys that can be stored in the node
- **vector<vector<Address>> children**: Vector of children
- **bool isLeaf**: Boolean to check if node is leaf
- **BPlusTree**: Friend class to allow BPlusTree to access private members of the node

```
class Node{
    private:
        float *keys;
        uint8_t numKeys;
        uint8_t maxKeys;
        vector<vector<Address>> children;
        bool isLeaf;
        friend class BPlusTree;
```

In addition, the Node class consists of following functions to construct and destruct Node objects, and to get and set Node attributes:

```
    Node(int maxKeys, bool isLeaf);
    ~Node();

    float getKey(int index){
        return keys[index];
    };
    int getNumKeys(){
        return static_cast<int>(numKeys);
    };
    int getMaxKeys(){
        return static_cast<int>(maxKeys);
    };
    vector<Address> getChildren(int index){
        return children[index];
    };
    int getIsLeaf(){
        return isLeaf;
    };
    void setKey(int index, float key){
        keys[index] = key;
    };
    void setNumKeys(int numKeys){
        this->numKeys = numKeys;
    };
    void setChildren(int index, vector<Address> children){
        this->children[index] = children;
    };
    void setIsLeaf(bool isLeaf){
        this->isLeaf = isLeaf;
    };
```

The parameter ‘n’ or maximum keys allowed in a node is an important parameter. We have specified it as *maxKeys*. The parameter significantly influences the structure and performance of the tree. If the parameter is too small, the tree may become excessively tall, leading to slower searches. Conversely, if the parameter is too large, it may result in wasted memory space and slower reactions. Our logic behind finding the best appropriate size is to use the number of records to be stored in the tree at runtime to dynamically compute the *maxKeys*.

```
while ((count + sizeof(Address) + sizeof(float)) <= (nodeSize - 2 * sizeof(uint8_t) - sizeof(bool)))
{
    count += sizeof(Address) + sizeof(float);
    calculatedCapacity++;
}
if (calculatedCapacity == 0)
{
    cerr << "Error: block size is too small" << endl;
}
while ((numberOfRecords % calculatedCapacity) < ((calculatedCapacity + 1) / 2))
{
    calculatedCapacity--;
}
this->maxKeys = calculatedCapacity;
```

BPlusTree Class Structure

```
class BPlusTree{
    public:
        Node* rootNode;
        Node* index;
        int keysStored;
        int maxKeys;
        int levels;
        int nodesStored;
        int nodeSize;
        int blockSize;
```

Our BPlusTree class consists of following attributes:

- **Node* rootNode**: Memory address of the root node of the tree
- **Node* index**: Index node of the tree
- **int keysStored**: Number of keys stored in the tree
- **int maxKeys**: Maximum number of keys that can be stored in a node
- **int levels**: Number of levels in the tree
- **int nodesStored**: Number of nodes stored in the tree
- **int nodeSize**: Size of a node
- **int blockSize**: Size of a block

Dealing with Duplicate Keys

To address the issue of duplicate keys, we pre-processed our records to generate a vector of all the Addresses sharing the same key. This vector has further been utilized to optimize the number of insertion calls made on the B+ tree - which will now be only the set of unique values that we have. Most of the games with the same key are also stored on the same block, which reduces the number of disk I/O when reading the records. We used a record map with definition

map<float, vector<Address>> recordMap to store all the records with the same key, and used this map to populate our B+ tree.

```
for (const auto &pair : recordMap)
{
    count++;
    cout << "Count: " << count << endl;
    std::cout << "Key: " << pair.first << ", Value: " << &pair.second << endl;
    tree.insert(pair.first, pair.second);
    tree.displayTree(tree.rootNode, 1);
    cout << "Number of nodes: " << tree.nodesStored << endl;
    cout << "Number of keys: " << tree.keysStored << endl;
    cout << "Number of levels: " << tree.levels << endl;
}
```

B+ Tree Search

We implemented 2 search functions for our B+ Tree:

searchKey():

The below code is meant for performing a search operation on B+ Tree to find a specific key. It was used for Experiment 3.

```
vector<Address> BPlusTree::searchKey(float key)
{
    if (this->rootNode == nullptr)
    {
        // Error case: tree is empty
        cerr << "Tree is empty" << endl;
    }
    Node *cursor = this->rootNode;
    int indexNodesAccessed = 1; // Tracking the total number of index nodes accessed
    displayNode(node: cursor);
```

a) The B+ tree search function firstly checks whether the root node of the tree is null.
b) The searchKey function takes a float key as input and returns a vector of Address objects. This vector represents the addresses of data items associated with the given key.
c) Initially, it checks whether the rootNode of the B+ tree is nullptr, which means the tree is empty. In such a case, it prints an error message and returns an empty vector.
d) If the tree is not empty, it initializes a cursor to the root node of the B+ tree and sets indexNodesAccessed to 1. This variable is used to keep track of the total number of index nodes accessed during the search.

```
while (!cursor->getIsLeaf())
{
    for (int i = 0; i < cursor->getNumKeys(); i++)
    {
        // Find the key in the current node
        if (key < cursor->getKey(index: i))
        {
            cursor = static_cast<Node *>(cursor->getChild(index: i, childIndex: 0).blockAddress);
            displayNode(node: cursor);
            indexNodesAccessed++;
            break;
        }
        // If we cannot find in any of the current keys, go to the last pointer
        if (i == cursor->getNumKeys() - 1)
        {
            cursor = static_cast<Node *>(cursor->getChild(index: i + 1, childIndex: 0).blockAddress);
            displayNode(node: cursor);
            indexNodesAccessed++;
            break;
        }
    }
}
```

e) It enters a while loop which iterates until the cursor reaches a leaf node.Within the loop, it iterates through the keys in the current node. For each key, it compares the key with the target key being searched for.

f) If the key is less than the current key, it implies that the desired key might be present in the subtree represented by the child at that index. So, it chooses the appropriate child node and updates the cursor to that child node. This is done in order to navigate deeper into the tree.

g) If it cannot find the key in any of the current keys, it chooses the last child pointer to move to the next node. This happens when the key is greater than or equal to the last key in the current node.

```
// Found a leaf node,traverse through it to find the key
for (int i = 0; i < cursor->getNumKeys(); i++)
{
    if (cursor->getKey(index: i) == key)
    {
        cout << ": " << indexNodesAccessed << endl;
        return cursor->getChildren(index: i);
    }
}
return vector<Address>{};
}
```

h) It iterates through the keys in the leaf node to find the exact match for the search key. If it finds a match, it prints the number of index nodes accessed during the search and returns the vector of addresses associated with the respective key.

i) If no match is found in the leaf node, it returns an empty vector which indicates that the search key was not found in the B+ tree.

searchRange():

The below code is meant for performing a search operation on B+ tree to find a range of keys. It was used for Experiment 4.

```
vector<vector<Address>> BPlusTree::searchRange(float low, float high)
{
    if (this->rootNode == nullptr)
    {
        // Error case: tree is empty
        cerr << "Tree is empty" << endl;
    }
    vector<vector<Address>> results;
    Node *cursor = this->rootNode;
    int indexNodesAccessed = 1; // Tracking the total number of index nodes accessed
```

a) It has two float parameters, namely low and high, which respectively represent the inclusive lower and upper bounds of the range being searched.
b) Initially, it starts by checking if the rootNode of the B+ tree is nullptr. If the tree is empty, it prints an error message and returns an empty vector of vectors.
c) It initializes an empty vector called results to store the results of the search which contains the vectors of Address objects, where each inner vector represents the addresses associated with keys in the specified range.
d) It initializes a cursor as a pointer to the root node of the B+ tree and sets indexNodesAccessed to 1 to track the total number of index nodes accessed during the search.

```
    while (!cursor->getIsLeaf())
    {
        for (int i = 0; i < cursor->getNumKeys(); i++)
        {
            // Find the key in the current node
            if (low < cursor->getKey(index: i))
            {
                cursor = static_cast<Node *>(cursor->getChild(index: i, childIndex: 0).blockAddress);
                // displayNode(cursor);
                indexNodesAccessed++;
                break;
            }
            // If we cannot find in any of the current keys, go to the last pointer
            if (i == cursor->getNumKeys() - 1)
            {
                cursor = static_cast<Node *>(cursor->getChild(index: i + 1, childIndex: 0).blockAddress);
                // displayNode(cursor);
                indexNodesAccessed++;
                break;      indicium15, 2 days ago • finished experiment 3 …
            }
        }
    }
```

e) The code then enters a loop that continues until the cursor reaches a leaf node. This is used to traverse the B+ tree from the root to the appropriate leaf node based on the low value.
f) Within the loop, it iterates through the keys in the current node. For each key, it checks if low is less than the key. If it is, it chooses the appropriate child pointer and updates the cursor to that child node.
g) If the low value is greater than or equal to the last key in the current node, it means that the range might start in the next sibling node. In this case, it chooses the last child pointer in order to move to the next node.

```
    // Found a leaf node,traverse through it to find the range
    bool flag = false;
    Node *temp;
    while (cursor != nullptr && flag != true)
    {
        for (int i = 0; i < cursor->getNumKeys(); i++)
        {
            if (low <= cursor->getKey(index: i) && cursor->getKey(index: i) <= high)
            {
                results.push_back(x: cursor->getChildren(index: i));
            }
            if (cursor->getKey(index: i) > high)
            {
                flag = true;
            }
        }
        temp = static_cast<Node *>(cursor->getChild(index: cursor->getNumKeys(), childIndex: 0).blockAddress);
        cursor = temp;
    }
    cout << "Number of Index Nodes Accessed in B+ Tree Search: " << indexNodesAccessed << endl;
    return results;
}
```

h) The code enters another loop, where it iterates through the keys in the leaf nodes and checks if each key falls within the specified range [low, high].
i) If a key is within the range, it adds the vector of Address objects to the results vector.
j) If it comes across a key greater than the high value, it sets the flag variable to true, indicating that it has passed the end of the search range.
k) After it finishes processing all the keys in the leaf node, it moves to the next leaf node using the last child pointer of the current leaf node, and the loop iterates until it reaches the end of the B+ tree.
l) The function prints the number of index nodes accessed during the search and returns the results vector.

B+Tree Insertion:

Bulk Loading

We have populated our B+ tree following the bulk loading approach. Our reason for doing so was that Bulk loading is faster than inserting records individually as it minimizes overhead associated with insertion. It reduces the number of I/O operations as data is loaded in large batches, thus improving execution time. It also optimizes memory and disk usage as the database can more efficiently allocate resources when working with large batches of data.

insert():

```
122  int BPlusTree::insert(float key, const vector<Address> value)
123  {
124      // No root node exists, create one
125      cout << "Inserting key: " << key << endl;
126      cout << "Inserting value: " << value[0].blockAddress << endl;
127      if (this->rootNode == nullptr)
128      {
129          Node *root = new Node(maxKeys: this->maxKeys, isLeaf: true);
130          root->setKey(index: 0, key);
131          cout << "KEY STORED: " << root->getKey(index: root->numKeys) << endl;
132          root->setChildren(index: 0, children: value);
133          cout << "ADDRESS STORED: " << root->getChildren(index: root->numKeys)[0].blockAddress << endl;
134          root->setNumKeys(root->getNumKeys() + 1);
135          this->rootNode = root;
136          this->nodesStored++;
137          this->keysStored++;
138          this->levels++;
139          return 1;
140      }
```

a) Initially it checks whether B+ tree has a root node or not.
b) If not, it creates a new root node, inserts the provided key and value into it. It also sets the root node to this newly created node. This is the case when the tree is initially empty.

```
141      else
142      {
143          cout << "Root node exists" << endl;
144          Node *cursor = rootNode;
145          Node *parent = nullptr;
146          Node *oldRootNode = rootNode;
147          while (cursor->getIsLeaf() == false)
148          {
149              parent = cursor;
150              cout << "Cursor Address: " << cursor << endl;
151              cout << "CURSOR GET LEAF: " << cursor->getIsLeaf() << endl;
152              bool found = false;
153              // cout << "Num Keys: " << cursor->getNumKeys() << endl;
154              // cout << "First key of cursor: " << cursor->getKey(0) << endl;
155              //  cout << "Keys in the cursor: " << cursor->numKeys << endl;
156              for (int i = 0; i < cursor->getNumKeys(); i++)
157              {
158                  cout << "Exploring Key " << cursor->getKey(index: i) << endl;
159                  if (key < cursor->getKey(index: i))
160                  {
161                      cout << "Expanding key " << cursor->getKey(index: i) << endl;
162                      cursor = static_cast<Node *>(cursor->getChild(index: 0, childIndex: 0).blockAddress);
163                      found = true;
164                      break;
165                  }
166              }
167              if (!found)
168              {
169                  // cout << "CURSOR -> GETNUMKEYS() " << cursor->getNumKeys() << endl;
170                  // cout << "Expanding last key " << cursor->getKey(cursor->getNumKeys()) << endl;
171                  // cout << "Memory address being accessed: " << cursor->getChild(cursor->getNumKeys(),0).blockAddress << endl;
172                  cursor = static_cast<Node *>(cursor->children[cursor->getNumKeys()][0].blockAddress);
173                  // cout << "Cursor Address: " << cursor << endl;
174                  // cout << "CURSOR GET LEAF IN IF !FOUND: " << cursor->getIsLeaf() << endl;
175                  // cout << "CURSOR KEYS =======" << cursor->getNumKeys() << endl;
176              }
177          }
178          // At this point we should have reached a leaf node
```

c) If the tree already has a root node, it traverses down the tree from the root node to find the appropriate leaf node where the key should be inserted.

d) It does this by iteratively following child pointers in non-leaf nodes until it reaches a leaf node. During this traversal, it prints debugging information to the console, such as the addresses of nodes, whether a node is a leaf or not, and the keys being explored.

```
//  Reaching a leaf node and it has space to add more nodes
if (cursor->getNumKeys() < this->maxKeys)
{
    // cout << "Reached a leaf node : " << (cursor->getIsLeaf()) << endl;
    int i = 0;
    // Find the first last stored key smaller than the current key to insert
    while (key > cursor->getKey(index: i) && i < cursor->getNumKeys())
        i++;
    // cout << "Position for Insertion: " << i << endl;
    //  Swap all keys after this point to make space for the new key
    for (int j = cursor->getNumKeys(); j > i; j--)
    {
        // cout << "Swapping j " << j << " with " << j - 1 << endl;
        cursor->setKey(index: j, key: cursor->getKey(index: j - 1));
        cursor->setChildren(index: j, children: cursor->getChildren(index: j - 1));
    }
    // Insert the keys into the correct point i
    cout << "Inserting key into position " << i << endl;
    cursor->setKey(index: i, key);
    cout << "Inserting children into position " << i << endl;
    cursor->setChildren(index: i, children: value);
    // cout << "Incrementing numKeys. New Num keys = " << cursor->getNumKeys() << endl;
    cursor->setNumKeys(cursor->getNumKeys() + 1);
    // cout << "Incrementing keysStored" << endl;
    this->keysStored++;
    return 1;
}
```

e) Once it reaches a leaf node, it checks if there is enough space in the leaf node to insert the new key and value.

f) If there is enough space, it inserts the key and value into the leaf node while maintaining the order of keys.

```
// If there is no more space, create a new node
else if (cursor->numKeys >= maxKeys)
{
    cout << "Maximum keys exceeded in current node, creating new node" << endl;
    // Initialize new node
    Node *newNode = new Node(maxKeys, isLeaf: true);
    // Link the last pointer of the old node to the pointer of the new leaf node
    cursor->setChildren(cursor->getNumKeys(), vector<Address>{Address(newNode, 0)});
    newNode->setIsLeaf(true);
    // cout << "NEWNODE IS A LEAF: " << newNode->getIsLeaf() << endl;
    // Move cursor to next node after linking the two leaf nodes together
    cursor = newNode;
    cursor->setKey(index: 0, key);
    cursor->setNumKeys(cursor->getNumKeys() + 1);
    cursor->setChildren(index: 0, children: value);
    this->nodesStored++;
    this->keysStored++;
    cout << "NODES STORED AFTER CREATING NEW NODE: " << this->nodesStored << endl;
    // cout << "PARENT == ROOTNODE: " << (parent == rootNode) << endl;
    // cout << "PARENT MEMORY ADDRESS: " << parent << endl;
    // cout << "ROOTNODE MEMORY ADDRESS: " << rootNode << endl;

```

g) If there is not enough space, it creates a new leaf node, and adjusts the pointers to link the old and new leaf nodes. Thus, it inserts the key and value into the new leaf node.

```
228             if (oldRootNode == rootNode && nodesStored == 2)
229             {
230                 cout << "Manually insert new root node when there are two leaf nodes" << endl;
231                 Node *newParentNode = new Node(maxKeys, isLeaf: false);
232                 // Address *newChildNodeAddress = new Address(&newNode, 0);
233                 // newParentNode->keys[0] = newNode->keys[0]; //0.477
234                 // First value in parent is the left bound of the right node
235                 newParentNode->setKey(index: 0, key: newNode->getKey(index: 0));
236                 cout << "First value in parent [left bound of the right node]? : " << (newParentNode->getKey(index: 0) == newNode->getKey(index: 0)) << endl;
237                 // Second pointer in parent is the memory address of the new node
238                 newParentNode->setChild(index: 0, child: Address(blockAddress: rootNode, offset: 0)); // All values less than 0.477
239                 newParentNode->setChild(index: 1, child: Address(blockAddress: newNode, offset: 0));  // All values greater than or equal to 0.477
240                 // cout << "Actual pointer of root node in memory " << (rootNode) << endl;
241                 // cout << "Actual pointer of new node in memory " << (newNode) << endl;
242                 // cout << "First pointer in parent: " << newParentNode->getChild(0,0).blockAddress << endl;
243                 // cout << "Second pointer in parent: " << newParentNode->getChild(1,0).blockAddress << endl;
244                 //  Set first value of children to be the old root node
245                 newParentNode->setNumKeys(1);
246                 newParentNode->setIsLeaf(false);
247                 // Update old root node to now be a leaf node
248                 // cout << "Address of old root node " << static_cast<Node *>(this->rootNode) << endl;
249                 // Update root node variable to the new parent node
250                 parent = newParentNode;
251                 this->rootNode = newParentNode;
252                 cout << "Address of new root node " << static_cast<Node *>(this->rootNode) << endl;
253                 this->nodesStored++;
254                 this->levels++;
255                 return 1;
256             }
257             else
258             {
259                 cout << "Calling insertInternal on parent address " << parent << endl;
260                 cout << "Storing child address in parent:  " << newNode << endl;
261                 insertInternal(key, parent, child: newNode);
262                 return 1;
263             }
264         }
265     }
266     return 0;
267 }
```

h) If the creation of a new leaf node is necessary and the old root node is the only node in the tree (indicating that the tree has only one level), it manually creates a new root node that acts as the parent of both the old root node and the new leaf node. This step is crucial for tree growth.

i) Finally, the code returns 1 if the insertion was successful.

insertInternal():

```
int BPlusTree::insertInternal(float key, Node *parent, Node *child)
{
    Node *cursor = parent;
    cout << "Cursor has " << cursor->getNumKeys() << " keys" << endl;
    cout << "Cursor has " << maxKeys << " max keys" << endl;
    // Check if the value can be inserted in the existing parent
    if ((cursor->getNumKeys() + 1) <= maxKeys)
    {
        // Find the correct position for inserting
        cout << "Space in parent, trying to find the location" << endl;
        cursor->setKey(index: cursor->getNumKeys(), key);
        cursor->setNumKeys(cursor->getNumKeys() + 1);
        cursor->setChildren(cursor->getNumKeys(), vector<Address>{Address(child, 0)});
        // //Keep incrementing i until you find the index where the key is smaller than current key stored
        // while(key > cursor->getKey(i) && i < cursor->getNumKeys()){
        //     i++;
        // }
        // cout << "Inserting in Parent Node at " << i << endl;
        // //Shift all keys by one step to create space for the change
        // for(int j = cursor->getNumKeys(); j>i; j--){
        //     cursor->setKey(j,cursor->getKey(j-1));
        // }
        // //Shift all pointers by one step to create space for the change
        // for(int j = cursor->getNumKeys()+1; j>i; j--){
        //     cursor->setChildren(j,cursor->getChildren(j-1));
        // }
        // Insert new key and pointer to parent
    }
```

The main purpose of using the insertInternal() function is to update the non leaf nodes according to the new values inserted in the leaf nodes and maintain a valid B+ tree data structure.

a) It initializes a cursor node with the parent node and prints debugging information about the number of keys and the maximum number of keys allowed in the cursor node.
b) It also checks if there is enough space in the cursor node to insert the new key and child pointer. If there is enough space:
    i) It inserts the key into the cursor node at the end of the keys list.
    ii) It increments the number of keys in the cursor node.
    iii) It inserts the child pointer into the cursor node at the corresponding position.

```
298     else
299     {
300         Node *newSibling = new Node(maxKeys, isLeaf: false);
301         newSibling->setIsLeaf(false);
302         // Temp list of keys and addresses to insert into the split nodes
303         float tempKeysList[maxKeys + 1];
304         vector<Address> tempAddressList;
305         //(maxKeys+2) = {nullptr, 0};
306         // Copy all keys into temp list
307         for (int i = 0; i < maxKeys; i++)
308         {
309             tempKeysList[i] = cursor->getKey(index: i);
310         }
311         tempKeysList[maxKeys] = key;
312         // Copy all addresses into temp list
313         for (int i = 0; i < maxKeys + 1; i++)
314         {
315             tempAddressList.push_back(x: cursor->getChild(index: i, childIndex: 0));
316         }
317         tempAddressList.push_back(Address{child, 0});
```

c) If there is not enough space in the cursor node to insert the new key and child pointer, it means that a node split is necessary:
   i) It creates a new sibling node (newSibling) with the same maximum number of keys and sets it as a non-leaf node.
   ii) It creates temporary lists to hold keys and child pointers, including the new key and child pointer to be inserted.
   iii) It copies the necessary keys and child pointers from the cursor node into the temporary lists.

```
318         // Split the nodes into two
319         cursor->setNumKeys((maxKeys + 1) / 2); //2
320         newSibling->setNumKeys((maxKeys - ((maxKeys + 1) / 2))); //2
321         // Reassign keys and pointers to cursor
322         // i = 0-1
323         for (int i = 0; i < cursor->getNumKeys(); i++)
324         {
325             //0 and 1 here
326             cursor->setKey(index: i, key: tempKeysList[i]);
327         }
328 
329         // Assign keys and pointers to newSibling
330         // i = 0-1, j = 2-3
331         for (int i = 0, j = cursor->getNumKeys()+1; i < newSibling->getNumKeys(); i++, j++)
332         {
333             newSibling->setKey(index: i, key: tempKeysList[j]);
334         }
335 
336         //TODO: change made here
337         // newSibling->setKey(newSibling->getNumKeys(), tempKeysList[maxKeys]);
338 
339         // Assign pointers into the new parent node
340         //TODO: change made here
341         for (int i = 0, j = cursor->getNumKeys()+1; i < newSibling->getNumKeys() + 1; i++, j++)
342         {
343             newSibling->setChild(index: i, child: tempAddressList[j]);
344         }
345 
346         // Remove remaining cursor keys from cursor
347         // float newRootValue = cursor->getKey(cursor->getNumKeys());
348         for (int i = cursor->getNumKeys(); i < cursor->getMaxKeys(); i++)
349         {
350             cursor->setKey(index: i, key: float());
351         }
352 
353         // Remove remaining cursor pointers from cursor
354         for (int i = cursor->getNumKeys() + 1; i < cursor->getMaxKeys() + 1; i++)
355         {
356             Address nullAddress{nullptr, 0};
357             cursor->setChild(index: i, child: nullAddress);
358         }
359 
360         // Assign new child to original parent
361         // TODO: change made here
362         newSibling->setChild(newSibling->getNumKeys(), Address{child, 0});
363 
```

iv) It splits the keys and child pointers between the cursor node and the new sibling node, to ensure that both the nodes have roughly equal numbers of keys.

v) It clears the remaining keys and child pointers in the cursor node.

vi) It sets the new child pointer to the child node in the cursor node.

```
364             if (cursor == this->rootNode)
365             {
366                 Node *newRoot = new Node(maxKeys, isLeaf: false);
367                 newRoot->setIsLeaf(false);
368                 // Set first key of new root node to be the rightmost key of original parent
369                 newRoot->setKey(index: 0, key: cursor->getKey(index: cursor->getNumKeys()));
370                 newRoot->setNumKeys(newRoot->getNumKeys() + 1);
371                 // Assign pointers to new root node
372                 newRoot->setChild(0, Address{parent, 0});
373                 newRoot->setChild(1, Address{newSibling, 0});
374                 // Update class variables
375                 this->rootNode = newRoot;
376                 this->nodesStored++;
377                 this->levels++;
378             }
```

d) If the cursor node is also the root node, it means that a new root node needs to be created:
   i) It creates a new root node with the same maximum number of keys and sets it as a non-leaf node.
   ii) It sets the first key of the new root node to be the rightmost key of the original parent node.
   iii) It assigns the appropriate child pointers to the new root node.
   iv) It updates the class variables to reflect the new root node, incrementing the number of nodes stored and the tree's level.

```
379         else
380         {
381             Node *parentNode = findParent(rootNode: this->rootNode, childNode: parent, lowerBoundKey: cursor->getKey(index: 0));
382             // insertInternal(cursor->numKeys, parent of cursor, newSibling);
383             insertInternal(key: tempKeysList[cursor->getNumKeys()], parent: parentNode, child: newSibling);
384         }
385     }
386     return 1;
387 }
```

e) If the cursor node is not the root node, it means that the parent node might also be required to be split:
   i) It finds the parent node of the cursor node by using the findParent function, passing in the root node, the cursor's parent node, and the key of the cursor's first key.
   ii) It recursively calls insertInternal with the new key, the parent node, and the new sibling node as parameters to continue the propagation of the split operation up the tree.

f) The function returns 1 if it has been successful.

B+ Tree Deletion

deleteNode():

```
527  int BPlusTree::deleteNode(float key)
528  {
529      // Check if tree is empty
530      if (this->rootNode == nullptr)
531      {
532          cerr << "Tree is empty" << endl;
533      }
534      const int minimumKeysForLeafNode = (maxKeys + 1) / 2;
535      Node *cursor = this->rootNode;
536      Node *parent;
537      int leftSibling, rightSibling; // Index of the left and the right leaf node that we need to borrow from
538
539      // Finding the leaf node with the key to delete
540      while (!cursor->getIsLeaf())
541      {
542          parent = cursor;
543
544          // Look for all the keys in the current node
545          for (int i = 0; i < cursor->getNumKeys(); i++)
546          {
547              leftSibling = i - 1;
548              rightSibling = i + 1;
549
550              // Find the key that we have to delete
551              if (key < cursor->getKey(index: i))
552              {
553                  cursor = static_cast<Node *>(cursor->getChild(index: i, childIndex: 0).blockAddress);
554                  break;
555              }
556              // If we cannot find, go to the last pointer in the node
557              if (i == cursor->getNumKeys() - 1)
558              {
559                  leftSibling = i;
560                  rightSibling = i + 2;
561                  cursor = static_cast<Node *>(cursor->getChild(index: i + 1, childIndex: 0).blockAddress);
562                  break;
563              }
564          }
565      }
566
```

a) It checks whether the tree is empty. If the tree is empty, it prints an error message and returns. It also calculates minimumKeysForLeafNode, which represents the minimum number of keys a leaf node should have to maintain tree balance.
b) It also initializes a cursor node to start traversal from the root node and defines variables for parent, leftSibling, and rightSibling indices, which will be used for potential node borrowing during the delete process.
c) The code then enters a loop that traverses the tree to find the leaf node containing the specified key. During this traversal:
    i) It keeps track of the leftSibling and rightSibling indices for potential key borrowing.
    ii) It also compares the key with the keys in the current node to determine the path to follow.
    iii) If the key is not found in the current node, it updates the leftSibling and rightSibling and continues performing the traversal.

```
567     // Cursor now points to the leaf node that contains the key that needs to be deleted
568
569     bool found = false;
570     int indexToBeDeleted;
571
572     // Traversing the leaf node to find the index of the specified key
573     for (int i = 0; i < cursor->getNumKeys(); i++)
574     {
575         if (cursor->getKey(index: i) == key)
576         {
577             found = true;
578             indexToBeDeleted = i;
579             break;
580         }
581     }
582
583     if (found == false)
584     {
585         cerr << "Can't find the key in the tree!" << endl;
586     }
587
588     // Removing the specified key and its pointer in the memory from the node
589     for (int i = indexToBeDeleted; i < cursor->getNumKeys(); i++)
590     {
591         cursor->setKey(index: i, key: cursor->getKey(index: i + 1));
592         cursor->setChildren(index: i, children: cursor->getChildren(index: i + 1));
593     }
594     // Decrement number of keys stored in the current node
595     cursor->setNumKeys(cursor->getNumKeys() - 1);
596     // Manually set the last node since it is not covered by for loop range
597     cursor->setChildren(index: cursor->getNumKeys(), children: cursor->getChildren(index: cursor->getNumKeys() + 1));
598
599     for (int i = cursor->getNumKeys() + 1; i < maxKeys + 1; i++)
600     {
601         cursor->setChildren(i, vector<Address>{Address{nullptr, 0}});
602     }
603
```

d) Once the cursor points to the leaf node containing the key, it searches for the index of the key within that leaf node.
e) If the key is not found in the leaf node, it prints an error message and returns.
f) If the key is found in the leaf node, it proceeds to delete the key and its associated pointer from the node. This is done by shifting all keys and pointers after the target key to the left by one position and decrementing the number of keys in the node.
g) After removing the key and pointer, the code ensures that the last pointer in the leaf node is set correctly.

```
604     // Deleting a key from the root
605     if (cursor == this->rootNode)
606     {
607         if (cursor->getNumKeys() == 0)
608         {
609             cout << "Entire index has been deleted" << endl;
610             rootNode = nullptr;
611         }
612         cout << "Successfully deleted " << key << endl;
613         return 1;
614     }
615 
616     // If we do not need to borrow from other nodes, end function
617     if (cursor->getNumKeys() >= minimumKeysForLeafNode)
618     {
619         cout << "Successfully deleted " << key << endl;
620         return 1;
621     }
```

h) If the node being deleted is the root node:
    i) It checks if the root node has no keys left. If it has no keys, the entire index has been deleted, and it sets rootNode to nullptr.
    ii) It prints a success deletion message and returns.
i) If the node being deleted is not the root node and it still has enough keys to maintain balance (equal to or greater than minimumKeysForLeafNode), it prints a successful deletion message and returns.

```
623     // If we don't have enough keys for a balanced tree, we try to take a key from the left sibling
624     if (leftSibling >= 0)
625     {
626         Node *leftNode = static_cast<Node *>(parent->getChild(index: leftSibling, childIndex: 0).blockAddress);
627 
628         // We check if leftNode has enough keys to lend one and still be a valid leaf node
629         if (leftNode->getNumKeys() >= minimumKeysForLeafNode + 1)
630         {
631 
632             // Shift the last pointer by one
633             cursor->setChildren(index: cursor->getNumKeys() + 1, children: cursor->getChildren(index: cursor->getNumKeys()));
634 
635             // Shift all the remaining keys and pointer back by one
636             for (int i = cursor->getNumKeys(); i > 0; i--)
637             {
638                 cursor->setKey(index: i, key: cursor->getKey(index: i - 1));
639                 cursor->setChildren(index: i, children: cursor->getChildren(index: i - 1));
640             }
641 
642             // Transfer the borrowed key from leftNode to current Node
643             cursor->setKey(index: 0, key: leftNode->getKey(index: leftNode->getNumKeys() - 1));
644             cursor->setChildren(index: 0, children: leftNode->getChildren(index: leftNode->getNumKeys() - 1));
645             cursor->setNumKeys(cursor->getNumKeys() + 1);     // to account for the increase in key
646             leftNode->setNumKeys(leftNode->getNumKeys() - 1); // to account for the decrease in key
647 
648             // Move the last pointer in the left node back by one to account for the deletion
649             leftNode->setChildren(index: cursor->getNumKeys(), children: cursor->getChildren(index: cursor->getNumKeys() + 1));
650             leftNode->setChildren(cursor->getNumKeys() + 1, vector<Address>{Address{nullptr, 0}});
651 
652             // Updating parent with the new key at the beginning of current node
653             parent->setKey(index: leftSibling, key: cursor->getKey(index: 0));
654 
655             if (cursor->getNumKeys() >= minimumKeysForLeafNode)
656             {
657                 cout << "Successfully deleted " << key << endl;
658             }
659 
660             return 1;
661         }
662     }
```

j) If the node being deleted is not the root node and it does not have enough keys to maintain balance, the code considers three cases:
    i) Try to borrow a key from the left sibling (if it exists): If the left sibling has enough keys to lend one, it borrows a key from the left sibling, adjusts the keys and pointers in the nodes accordingly, and updates the parent node.

```
664     // If we can't take a key from left sibling, we check the right sibling
665     if (rightSibling <= parent->getNumKeys())
666     {
667         Node *rightNode = static_cast<Node *>(parent->getChild(index: rightSibling, childIndex: 0).blockAddress);
668 
669         // We check if rightNode has enough keys to lend one and still be a valid leaf node
670         if (rightNode->getNumKeys() >= minimumKeysForLeafNode + 1)
671         {
672 
673             // Shifting the last pointer by one to make space to add new key and pointer
674             // Ensure that the last pointer still points to the right sibling
675             cursor->setChildren(index: cursor->getNumKeys() + 1, children: cursor->getChildren(index: cursor->getNumKeys()));
676 
677             // Transfer borrowed key and pointer
678             cursor->setKey(index: cursor->getNumKeys(), key: rightNode->getKey(index: 0));
679             cursor->setChildren(index: cursor->getNumKeys(), children: rightNode->getChildren(index: 0));
680             cursor->setNumKeys(cursor->getNumKeys() + 1);
681             rightNode->setNumKeys(rightNode->getNumKeys() - 1);
682 
683             // Shift all the keys and pointers in rightNode left by one
684             for (int i = 0; i < rightNode->getNumKeys(); i++)
685             {
686                 rightNode->setKey(index: i, key: rightNode->getKey(index: i + 1));
687                 rightNode->setChildren(index: i, children: rightNode->getChildren(index: i + 1));
688             }
689 
690             // Shift right sibling's last pointer left by one
691 
692             rightNode->setChildren(index: cursor->getNumKeys(), children: rightNode->getChildren(index: cursor->getNumKeys() + 1));
693             rightNode->setChildren(cursor->getNumKeys() + 1, vector<Address>{Address{nullptr, 0}});
694 
695             // Updating the parent node with the new key at the beginning of the right sibling
696             parent->setKey(index: rightSibling - 1, key: rightNode->getKey(index: 0));
697 
698             if (cursor->getNumKeys() >= minimumKeysForLeafNode)
699             {
700                 cout << "Successfully deleted " << key << endl;
701             }
702 
703             return 1;
704         }
705     }
```

ii) Try to borrow a key from the right sibling (if it exists): If the right sibling has enough keys to lend one, it borrows a key from the right sibling, adjusts the keys and pointers, and updates the parent node.

```
    // Reaching this point means that we cannot borrow keys from the siblings
    // In this case, we must merge nodes

    // If current node has a left sibling, merge with it
    if (leftSibling >= 0)
    {

        // Left sibling
        Node *leftNode = static_cast<Node *>(parent->getChild(index: leftSibling, childIndex: 0).blockAddress);

        // Transfering all keys and pointers from current node to left sibling
        for (int i = leftNode->getNumKeys(), j = 0; j < cursor->getNumKeys(); i++, j++)
        {
            leftNode->setKey(index: i, key: cursor->getKey(index: j));
            leftNode->setChildren(index: i, children: cursor->getChildren(index: j));
        }

        // Updating numKeys and making sure that the last pointer in the left node points to the node after the current node
        leftNode->setNumKeys(leftNode->getNumKeys() + cursor->getNumKeys());
        leftNode->setChildren(index: leftNode->getNumKeys(), children: cursor->getChildren(index: cursor->getNumKeys()));


        deleteInternal(key: parent->getKey(index: leftSibling), parent, child: cursor);
    }
    // If merge with left sibling does not work, we try merging with the right sibling
    else if (rightSibling <= parent->getNumKeys())
    {
        // Right sibling
        Node *rightNode = static_cast<Node *>(parent->getChild(index: rightSibling, childIndex: 0).blockAddress);

        // Transfering all keys and pointers from right node to current node
        for (int i = cursor->getNumKeys(), j = 0; j < rightNode->getNumKeys(); i++, j++)
        {
            cursor->setKey(index: i, key: rightNode->getKey(index: j));
            cursor->setChildren(index: i, children: rightNode->getChildren(index: j));
        }

        // Updating numkeys and making sure that the last pointer in the current node points to the node after the right node
        cursor->setNumKeys(cursor->getNumKeys() + rightNode->getNumKeys());
        cursor->setChildren(index: cursor->getNumKeys(), children: rightNode->getChildren(index: rightNode->getNumKeys()));

        if(key == static_cast<float>(0.29)){
            bool i = true;
        }
        // We need to update the parent in order to fully remove the right node.
        deleteInternal(key: parent->getKey(index: rightSibling - 1), parent, child: rightNode);
    }
    this->nodesStored--;
    return 1;
}
```

iii) Merge nodes: If neither borrowing from the left nor right sibling is possible, the code merges the current node with either the left or right sibling, transferring keys and pointers as needed.

k) The code also updates the number of stored nodes and recursively calls the ‘deleteInternal’ function to handle parent node adjustments in the case of node merges.

deleteInternal():

The main purpose of using the deleteInternal() function is to update the non leaf nodes according to the new values deleted in the leaf nodes and maintain a valid B+ tree data structure.

```
758  int BPlusTree::deleteInternal(float key, Node *parent, Node *child)
759  {
760      Node *cursor = parent;
761
762      // If current parent is root
763      if (cursor == this->rootNode)
764      {
765
766          // If we have to remove all keys in root (parent), we change the new root to its child
767          if (cursor->getNumKeys() == 1)
768          {
769
770              // If the second pointer contains the child to delete, we make the first pointer the new root
771              if (cursor->getChild(index: 1, childIndex: 0).blockAddress == child)
772              {
773                  this->rootNode = static_cast<Node *>(cursor->getChild(index: 0, childIndex: 0).blockAddress);
774                  cout << "Root node changed!" << endl;
775
776                  return 1;
777              }
778              // if the first pointer contains the child to delete, we make the second pointer the new root
779              else if (cursor->getChild(index: 0, childIndex: 0).blockAddress == child)
780              {
781                  this->rootNode = static_cast<Node *>(cursor->getChild(index: 1, childIndex: 0).blockAddress);
782                  cout << "Root node changed!" << endl;
783                  return 1;
784              }
785          }
786      }
```

a) It starts by setting the cursor node to the parent node provided as an argument.
b) If the current parent node is the root node, it handles special cases for root node deletion:
c) If there's only one key left in the root node, it checks which child (left or right) corresponds to the child node being deleted. Then, it promotes the other child as the new root node and prints a message indicating the change.
d) It then starts searching for the position of the key to delete within the parent node based on the first key of the child node.
e) It then removes the key from the parent node by shifting all keys to the left after the position of the key being deleted.

```
788     int positionToDelete; // position of the internal node to be deleted
789
790     // Search for key to delete in parent based on the first key of the child to delete
791     for (int i = 0; i < cursor->getNumKeys(); i++)
792     {
793         if (cursor->getKey(index: i) == key)
794         {
795             positionToDelete = i;
796             break;
797         }
798     }
799
800     // Deleting the key from the root
801     for (int i = positionToDelete; i < cursor->getNumKeys(); i++)
802     {
803         cursor->setKey(index: i, key: cursor->getKey(index: i + 1));
804     }
805
806     // Search for pointer to delete in parent based on the key
807     for (int i = 0; i < cursor->getNumKeys() + 1; i++)
808     {
809         if (cursor->getChild(index: i, childIndex: 0).blockAddress == child)
810         {
811             positionToDelete = i;
812             break;
813         }
814     }
815 // TODO: EVERYTHING IS CORRECT TILL THIS POINT!
816     // Deleting the pointer from the root
817     for (int i = positionToDelete; i < cursor->getNumKeys() + 1; i++)
818     {
819         cursor->setChild(index: i, child: cursor->getChild(index: i + 1, childIndex: 0));
820     }
821
822     // Updating the number of keys in the parent
823     cursor->setNumKeys(cursor->getNumKeys() - 1);
824
825     // Check if the node size is valid according to the requirements
826     // TODO: Original Line
827     // if (cursor->getNumKeys() >= ((maxKeys + 1) / 2 - 1))
828     if (cursor->getNumKeys() >= ((maxKeys) / 2))
829     {
830         return 1;
831     }
832
```

f) The code searches for the position of the child node to delete within the parent node based on the key.
g) It deletes the corresponding pointer (child) from the parent node by shifting all pointers to the left after the position of the pointer being deleted.
h) The number of keys in the parent node is updated by decrementing it.
i) The code checks if the number of keys in the parent node still satisfies the minimum requirement for an internal node, which is (maxKeys + 1) / 2 - 1. If it does, the function returns with 1 since it has been successful.

```
    // Satisfying this condition means that the parent is a root node and it does not matter if it doesn't satisfy the minimum keys condition for a non leaf node
    if (parent == this->rootNode)
    {
        return 1;
    }

    // We find the parent of the parent node to find the parent's siblings
    // TODO: Original Line
    // Node *grandParent = findParent(this->rootNode, parent, parent->getKey(0));
    Node *grandParent = findParent(rootNode: this->rootNode, childNode: cursor, lowerBoundKey: cursor->getKey(index: 0));
    int leftSibling, rightSibling;

    // Find the left and right sibling of the parent
    for (int i = 0; i < grandParent->getNumKeys() + 1; i++)
    {
        if (grandParent->getChild(index: i, childIndex: 0).blockAddress == parent)
        {
            leftSibling = i - 1;
            rightSibling = i + 1;
            positionToDelete = i;
            break;
        }
    }
```

j) If the parent node is also the root node, it means it's acceptable for the root node to have fewer keys. In this case, it returns with 1.

k) If the parent node doesn't satisfy the minimum key condition and is not the root node, it then proceeds to find the grandParent of the parent node. This is required in order to locate the left and right siblings of the parent node.

l) It then searches for the indices of the left and right siblings of the parent node within the grandParent node.

```
    // Check if left sibling exists
    if (leftSibling >= 0)
    {
        Node *leftNode = static_cast<Node *>(grandParent->getChild(index: leftSibling, childIndex: 0).blockAddress);
        // Check if it is possible to take a key from the left sibling
        if (leftNode->getNumKeys() >= ((maxKeys) / 2))
        {
            // Making space at the beginning of the current parent to fit the incoming key from left sibling
            for (int i = cursor->getNumKeys(); i > 0; i--)
            {
                cursor->setKey(index: i, key: cursor->getKey(index: i - 1));
            }
            // Transfer borrowed key and cursor to pointer from left node
            cursor->setKey(index: 0, key: grandParent->getKey(index: leftSibling));
            // Node* leftSibling = static_cast<Node*>(grandParent->getChild(leftSibling,0).blockAddress);
            grandParent->setKey(index: leftSibling, key: leftNode->getKey(index: leftNode->getNumKeys() - 1));

            // Move all pointers back in the cursor to fit a pointer
            for (int i = cursor->getNumKeys() + 1; i > 0; i--)
            {
                cursor->setChild(index: i, child: cursor->getChild(index: i - 1, childIndex: 0));
            }

            cursor->setChild(index: 0, child: leftNode->getChild(index: leftNode->getNumKeys(), childIndex: 0));

            // Updating the number of keys in the node
            cursor->setNumKeys(cursor->getNumKeys() + 1);
            leftNode->setNumKeys(leftNode->getNumKeys() - 1);

            // Shift left sibling's last pointer left by one
            leftNode->setChild(index: cursor->getNumKeys(), child: leftNode->getChild(index: cursor->getNumKeys() + 1, childIndex: 0));

            return 1;
        }
    }
```

m) If a left sibling exists and has enough keys to lend, it transfers a key from the left sibling to the parent node to maintain balance. It shifts the keys and pointers in the parent and left sibling accordingly and updates the grandParent node with the new key value.

```
893     // Check if right sibling exists
894     if (rightSibling <= grandParent->getNumKeys())
895     {
896 
897         Node *rightNode = static_cast<Node *>(grandParent->getChild(index: rightSibling, childIndex: 0).blockAddress);
898 
899         // Check if it is possible to take a key from the right sibling
900         if (rightNode->getNumKeys() >= ((maxKeys) / 2))
901         {
902 
903             // Transfer leftmost key and pointer from right node to current node
904             // TODO: come back here for pos later
905             cursor->setKey(index: cursor->getNumKeys(), key: grandParent->getKey(index: positionToDelete));
906             grandParent->setKey(index: positionToDelete, key: rightNode->getKey(index: 0));
907 
908             // Shift the keys in the right node left by one
909             for (int i = 0; i < rightNode->getNumKeys() - 1; i++)
910             {
911                 rightNode->setKey(index: i, key: rightNode->getKey(index: i + 1));
912             }
913 
914             // Transfer first pointer from right node to cursor
915             cursor->setChild(index: cursor->getNumKeys() + 1, child: rightNode->getChild(index: 0, childIndex: 0));
916 
917             // Shift pointers left for right node and delete the first pointer
918             for (int i = 0; i < rightNode->getNumKeys(); ++i)
919             {
920                 rightNode->setChild(index: i, child: rightNode->getChild(index: i + 1, childIndex: 0));
921             }
922 
923             // Updating the number of keys inside the nodes
924             cursor->setNumKeys(cursor->getNumKeys() + 1);
925             rightNode->setNumKeys(rightNode->getNumKeys() - 1);
926 
927             return 1;
928         }
929     }
```

n) If a right sibling exists and has enough keys to lend, it transfers a key from the right sibling to the parent node in a similar manner as with the left sibling.

```
931     // Case 3 : Reaching here means that we cannot borrow keys from siblings so we must merge nodes
932     // If left sibling exists we merge with it
933     if (leftSibling >= 0)
934     {
935         Node *leftNode = static_cast<Node *>(grandParent->getChild(index: leftSibling, childIndex: 0).blockAddress);
936 
937         // Making left node's upper bound to be cursor's lower bound
938         leftNode->setKey(index: leftNode->getNumKeys(), key: grandParent->getKey(index: leftSibling));
939 
940         // Transfer all keys from current node to left node
941         for (int i = leftNode->getNumKeys() + 1, j = 0; j < cursor->getNumKeys(); j++)
942         {
943             leftNode->setKey(index: i, key: cursor->getKey(index: j));
944         }
945 
946         // Transfer all pointers from current node to left node
947         for (int i = leftNode->numKeys + 1, j = 0; j < cursor->numKeys + 1; j++)
948         {
949             leftNode->setChild(index: i, child: cursor->getChild(index: j, childIndex: 0));
950             cursor->setChild(j, Address{nullptr, 0});
951         }
952 
953         // Update numkeys
954         leftNode->setNumKeys(leftNode->getNumKeys() + cursor->getNumKeys() + 1);
955         cursor->setNumKeys(0);
956         this->nodesStored--;
957 
958         // We need to update the parent in order to fully remove the current node
959         deleteInternal(key: grandParent->getKey(index: leftSibling), parent: grandParent, child: parent);
960     }
```

o) If neither borrowing from the left nor right sibling is possible, it means the nodes must be merged. If a left sibling exists, the parent node is merged into the left sibling, transferring keys and pointers. The parent node is then deleted.
p) If no left sibling exists, and a right sibling does, the parent node is merged into the right sibling in a similar fashion. The parent node is deleted after the merge.

Usage of B+ Tree in Experiments

Experiment 2:

a) To build a B+ tree on the attribute "FG_PCT_home", the insert() function of BPlusTree class is used.
b) It uses the Node class to traverse the tree and find the appropriate leaf node for the key.
c) Once the appropriate leaf node is found, the insert() function inserts the record into the node.
d) If the leaf node is full after the insertion, the function splits the node into two nodes.
e) It then creates a new node and moves half of the keys and values from the full node to the new node.
f) If the parent node of the leaf node is full after the insertion, the function recursively splits the parent node and updates the tree structure accordingly.

Experiment 3:

a) To retrieve movies with certain “FG_PCT_home” values, the searchKey() function is used.
b) It uses the Node class to traverse the tree to find the appropriate leaf node for the key.
c) It initializes a cursor variable to point to the root node.
d) It then enters a loop that continues until the cursor variable points to a leaf node and in each loop iteration, the searchKey() function iterates over the keys in the current node to find the appropriate child node for the key.
e) If the key is less than the current key, the searchKey() function sets the cursor to point to the child node to the left of the current key. Else, it sets the cursor to point to the right. Once the appropriate leaf node is found, the searchKey() function iterates over the keys in the node to find the key that matches with its input key.

Experiment 4:

a) To retrieve movies with a certain range of “FG_PCT_home” values, the searchRange() function is used.
b) Unlike the searchKey() function, it takes two keys as input and returns a vector of Address objects that correspond to the keys within the specified range.
c) After finding the value of the lower bound key, it proceeds to iterate over the remaining keys in the node to find all keys that are less than or equal to the upper bound key.
d) The function repeats this process for all subsequent leaf nodes until all keys within the specified range have been found.

Experiment 5:

a) To delete those movies that are below a certain “FG_PCT_home” value, the deleteNode() function is used.
b) The function first initializes a cursor variable to point to the root node of the tree.
c) It then enters a loop that continues until the cursor variable points to a leaf node and in each iteration of the loop, the function iterates over the keys in the current node to find the appropriate child node for the key.
d) If the key is less than the current key, the function sets the cursor variable to point to the child node to the left of the current key. Else, it sets the cursor variable to point to the child node to the right of the last key.
e) Once the appropriate leaf node is found, the function iterates over the keys in the node to find the key that matches the input parameter. If the key is found, it deletes the corresponding key-value pair from the node.
f) If the node becomes less than half full after the deletion, the function checks if it can borrow keys from its left or right sibling nodes. If borrowing is not possible, it will merge the node with its left or right sibling node.

Experiment Results

Experiment 1

```
Experiment 1
Number of Records: 26651
Size of a Record: 40 bytes
Number of Records in a Block : 10
Number of Blocks for Storing Data : 2666
```

Number of records	26651
Size of a record	40
Number of records stored in a block	10
Number of blocks for storing the data	2666

Experiment 2

```
----------------------------------------
Experiment 2
Parameter 'n' of B+ Tree: 16
Number of nodes of B+ Tree: 25
Number of levels of B+ Tree: 3
Content of Root Node:
| 0.425 | x | x | x | x | x | x | x | x | x | x | x | x | x | x |
----------------------------------------
```

Parameter n of the B+ tree	16
Number of nodes of the B+ tree	25
Number of levels of the B+ tree	3
Content of the root node (only the keys)	| 0.425 | x | x | x | x | x | x | x | x | x | x | x | x | x | x |

Experiment 3

```
13  ----------------------------------------
14  Experiment 3
15  Index Nodes Accessed while Searching B+ Key: 3
16  Number of Data Blocks Accessed for Reading Records from Database: 86
17  Number of Records with FG_PCT 0.5: 848
18  Linear Search for Records with FG_PCT 0.5
19  Average of FG3_PCT_HOME for Records Returned: 0.391201
20  Number of Records with FG_PCT 0.5 with Linear Search: 848
21  Number of Data Blocks accessed for Query with Linear Search: 2060
22  Runtime of Search and Retrieval: 786125 nanoseconds
23  Runtime of Linear Search of Data Blocks: 5620584 nanoseconds
24
```

Number of index nodes the process accesses	3
Number of data blocks the process accesses	86
Average of “FG3_PCT_home” of the records that are returned	0.391201
Running time of the retrieval process using B+ Tree	786125 nanoseconds
Number of data blocks that would be accessed by a brute-force linear scan method	2060
Running time for brute-force linear scan	5620584 nanoseconds

Our linear search stops when we encounter the first record that is greater than 0.5, because we know that the data is structured in ascending order. So we can stop the search when we find the first key that violates the upper bound.

Experiment 4

```
----------------------------------------
Experiment 4
Number of Index Nodes Accessed in B+ Tree Search: 3
Number of Data Blocks Accessed: 24
Average of FG3_PCT_HOME for Records Returned: 0.502329
Number of Records with FG_PCT between 0.6 and 1 (inclusive) with Linear Search: 237
Number of Data Blocks accessed for Ranged Query with Linear Search: 2666
Runtime of Search and Retrieval: 274791 nanoseconds
Runtime of Linear Search of Data Blocks: 7301875 nanoseconds
```

Number of index nodes the process accesses	3
Number of data blocks the process accesses	24
Average of “FG3_PCT_home” of the records that are returned	0.502329
Running time of the retrieval process using B+ Tree	274791 nanoseconds
Number of data blocks that would be accessed by a brute-force linear scan method	2666
Running time for brute-force linear scan	7301875 nanoseconds

We had to iterate through all the nodes as we have no way of knowing the minimum value in a simple linear search. So we had to start from the 0th index and go till the end.

```
Level 1: | 0.425 | x | x | x | x | x | x | x | x | x | x | x | x | x | x |
    Level 2: | 0.29 | 0.306 | 0.322 | 0.343 | 0.359 | 0.375 | 0.391 | 0.409 | x | x | x | x | x | x | x | x |
        Level 3: | 0 | 0.25 | 0.257 | 0.266 | 0.269 | 0.274 | 0.275 | 0.277 | 0.278 | 0.279 | 0.282 | 0.283 | 0.284 | 0.286 | 0.288 | 0.289 |
        Level 3: | 0.29 | 0.291 | 0.292 | 0.293 | 0.294 | 0.295 | 0.296 | 0.297 | 0.298 | 0.299 | 0.3 | 0.301 | 0.302 | 0.303 | 0.304 | 0.305 |
        Level 3: | 0.306 | 0.307 | 0.308 | 0.309 | 0.31 | 0.311 | 0.312 | 0.313 | 0.314 | 0.315 | 0.316 | 0.317 | 0.318 | 0.319 | 0.32 | 0.321 |
        Level 3: | 0.322 | 0.323 | 0.324 | 0.325 | 0.326 | 0.327 | 0.328 | 0.329 | 0.33 | 0.333 | 0.337 | 0.338 | 0.339 | 0.34 | 0.341 | 0.342 |
        Level 3: | 0.343 | 0.344 | 0.345 | 0.346 | 0.347 | 0.348 | 0.349 | 0.35 | 0.351 | 0.352 | 0.353 | 0.354 | 0.355 | 0.356 | 0.357 | 0.358 |
        Level 3: | 0.359 | 0.36 | 0.361 | 0.362 | 0.363 | 0.364 | 0.365 | 0.366 | 0.367 | 0.368 | 0.369 | 0.37 | 0.371 | 0.372 | 0.373 | 0.374 |
        Level 3: | 0.375 | 0.376 | 0.377 | 0.378 | 0.379 | 0.38 | 0.381 | 0.382 | 0.383 | 0.384 | 0.385 | 0.386 | 0.387 | 0.388 | 0.389 | 0.39 |
        Level 3: | 0.391 | 0.392 | 0.393 | 0.394 | 0.395 | 0.396 | 0.397 | 0.398 | 0.4 | 0.402 | 0.403 | 0.404 | 0.405 | 0.406 | 0.407 | 0.408 |
        Level 3: | 0.409 | 0.41 | 0.411 | 0.412 | 0.413 | 0.414 | 0.415 | 0.416 | 0.417 | 0.418 | 0.419 | 0.42 | 0.421 | 0.422 | 0.423 | 0.424 |
    Level 2: | 0.442 | 0.458 | 0.474 | 0.49 | 0.514 | 0.53 | 0.546 | 0.563 | 0.58 | 0.596 | 0.614 | 0.63 | 0.65 | x | x | x |
        Level 3: | 0.425 | 0.426 | 0.427 | 0.429 | 0.43 | 0.431 | 0.432 | 0.433 | 0.434 | 0.435 | 0.436 | 0.437 | 0.438 | 0.439 | 0.44 | 0.441 |
        Level 3: | 0.442 | 0.443 | 0.444 | 0.445 | 0.446 | 0.447 | 0.448 | 0.449 | 0.45 | 0.451 | 0.452 | 0.453 | 0.454 | 0.455 | 0.456 | 0.457 |
        Level 3: | 0.458 | 0.459 | 0.46 | 0.461 | 0.462 | 0.463 | 0.464 | 0.465 | 0.466 | 0.467 | 0.468 | 0.469 | 0.47 | 0.471 | 0.472 | 0.473 |
        Level 3: | 0.474 | 0.475 | 0.476 | 0.477 | 0.478 | 0.479 | 0.48 | 0.481 | 0.482 | 0.483 | 0.484 | 0.485 | 0.486 | 0.487 | 0.488 | 0.489 |
        Level 3: | 0.49 | 0.492 | 0.493 | 0.494 | 0.495 | 0.496 | 0.5 | 0.505 | 0.506 | 0.507 | 0.508 | 0.509 | 0.51 | 0.511 | 0.512 | 0.513 |
        Level 3: | 0.514 | 0.515 | 0.516 | 0.517 | 0.518 | 0.519 | 0.52 | 0.521 | 0.522 | 0.523 | 0.524 | 0.525 | 0.526 | 0.527 | 0.528 | 0.529 |
        Level 3: | 0.53 | 0.531 | 0.532 | 0.533 | 0.534 | 0.535 | 0.536 | 0.537 | 0.538 | 0.539 | 0.54 | 0.541 | 0.542 | 0.543 | 0.544 | 0.545 |
        Level 3: | 0.546 | 0.547 | 0.548 | 0.549 | 0.55 | 0.551 | 0.552 | 0.553 | 0.554 | 0.556 | 0.557 | 0.558 | 0.559 | 0.56 | 0.561 | 0.562 |
        Level 3: | 0.563 | 0.564 | 0.565 | 0.566 | 0.567 | 0.568 | 0.569 | 0.57 | 0.571 | 0.573 | 0.574 | 0.575 | 0.576 | 0.577 | 0.578 | 0.579 |
        Level 3: | 0.58 | 0.581 | 0.582 | 0.583 | 0.584 | 0.585 | 0.586 | 0.587 | 0.588 | 0.589 | 0.59 | 0.591 | 0.592 | 0.593 | 0.594 | 0.595 |
        Level 3: | 0.596 | 0.597 | 0.598 | 0.6 | 0.602 | 0.603 | 0.604 | 0.605 | 0.606 | 0.607 | 0.608 | 0.609 | 0.61 | 0.611 | 0.612 | 0.613 |
        Level 3: | 0.614 | 0.615 | 0.616 | 0.617 | 0.618 | 0.619 | 0.62 | 0.621 | 0.622 | 0.623 | 0.624 | 0.625 | 0.626 | 0.627 | 0.628 | 0.629 |
        Level 3: | 0.63 | 0.631 | 0.632 | 0.633 | 0.634 | 0.635 | 0.636 | 0.639 | 0.64 | 0.641 | 0.642 | 0.643 | 0.644 | 0.645 | 0.646 | 0.648 |
        Level 3: | 0.65 | 0.651 | 0.652 | 0.653 | 0.654 | 0.655 | 0.658 | 0.662 | 0.667 | 0.671 | 0.675 | x | x | x | x | x |
----------------------------------------
```

Experiment 5

```
----------------------------------------
Experiment 5
Number of nodes of the updated B+ tree: 21
Number of levels of the updated B+ tree: 3
Content of the Root Node:
| 0.49 | x | x | x | x | x | x | x | x | x | x | x | x | x | x |
Runtime of Linear Search Query for Keys Between 0 and 0.35: 6888291 nanoseconds
Number of Data Blocks Accessed for Linear Search: 2666
Runtime of B+ Tree Query for Keys Between 0 and 0.35: 40708 nanoseconds
Runtime of B+ Tree Deletion of Keys Between 0 and 0.35: 1152708 nanoseconds
Runtime of Database Record Deletion of Keys Between 0 and 0.35: 2138958 nanoseconds
```

Number of nodes of the updated B+ tree	21
Number of levels of the updated B+ tree	3
Content of the root node of the updated B+ tree	| 0.425 | x | x | x | x | x | x | x | x | x | x | x | x | x |
Running time of the process using B+ Tree	3332374 nanoseconds
Number of data blocks that would be accessed by a brute-force linear scan method	2666
Running time for brute-force linear scan	9027249 nanoseconds

Resulting B+ Tree:

```
Level 1: | 0.49 | x | x | x | x | x | x | x | x | x | x | x | x | x |
   Level 2: | 0.359 | 0.375 | 0.391 | 0.409 | 0.425 | 0.442 | 0.458 | 0.474 | x | x | x | x | x | x | x | x |
      Level 3: | 0.351 | 0.352 | 0.353 | 0.354 | 0.355 | 0.356 | 0.357 | 0.358 | x | x | x | x | x | x | x | x |
      Level 3: | 0.359 | 0.36 | 0.361 | 0.362 | 0.363 | 0.364 | 0.365 | 0.366 | 0.367 | 0.368 | 0.369 | 0.37 | 0.371 | 0.372 | 0.373 | 0.374 |
      Level 3: | 0.375 | 0.376 | 0.377 | 0.378 | 0.379 | 0.38 | 0.381 | 0.382 | 0.383 | 0.384 | 0.385 | 0.386 | 0.387 | 0.388 | 0.389 | 0.39 |
      Level 3: | 0.391 | 0.392 | 0.393 | 0.394 | 0.395 | 0.396 | 0.397 | 0.398 | 0.4 | 0.402 | 0.403 | 0.404 | 0.405 | 0.406 | 0.407 | 0.408 |
      Level 3: | 0.409 | 0.41 | 0.411 | 0.412 | 0.413 | 0.414 | 0.415 | 0.416 | 0.417 | 0.418 | 0.419 | 0.42 | 0.421 | 0.422 | 0.423 | 0.424 |
      Level 3: | 0.425 | 0.426 | 0.427 | 0.429 | 0.43 | 0.431 | 0.432 | 0.433 | 0.434 | 0.435 | 0.436 | 0.437 | 0.438 | 0.439 | 0.44 | 0.441 |
      Level 3: | 0.442 | 0.443 | 0.444 | 0.445 | 0.446 | 0.447 | 0.448 | 0.449 | 0.45 | 0.451 | 0.452 | 0.453 | 0.454 | 0.455 | 0.456 | 0.457 |
      Level 3: | 0.458 | 0.459 | 0.46 | 0.461 | 0.462 | 0.463 | 0.464 | 0.465 | 0.466 | 0.467 | 0.468 | 0.469 | 0.47 | 0.471 | 0.472 | 0.473 |
      Level 3: | 0.474 | 0.475 | 0.476 | 0.477 | 0.478 | 0.479 | 0.48 | 0.481 | 0.482 | 0.483 | 0.484 | 0.485 | 0.486 | 0.487 | 0.488 | 0.489 |
   Level 2: | 0.514 | 0.53 | 0.546 | 0.563 | 0.58 | 0.596 | 0.614 | 0.63 | 0.65 | x | x | x | x | x | x | x |
      Level 3: | 0.49 | 0.492 | 0.493 | 0.494 | 0.495 | 0.496 | 0.5 | 0.505 | 0.506 | 0.507 | 0.508 | 0.509 | 0.51 | 0.511 | 0.512 | 0.513 |
      Level 3: | 0.514 | 0.515 | 0.516 | 0.517 | 0.518 | 0.519 | 0.52 | 0.521 | 0.522 | 0.523 | 0.524 | 0.525 | 0.526 | 0.527 | 0.528 | 0.529 |
      Level 3: | 0.53 | 0.531 | 0.532 | 0.533 | 0.534 | 0.535 | 0.536 | 0.537 | 0.538 | 0.539 | 0.54 | 0.541 | 0.542 | 0.543 | 0.544 | 0.545 |
      Level 3: | 0.546 | 0.547 | 0.548 | 0.549 | 0.55 | 0.551 | 0.552 | 0.553 | 0.554 | 0.556 | 0.557 | 0.558 | 0.559 | 0.56 | 0.561 | 0.562 |
      Level 3: | 0.563 | 0.564 | 0.565 | 0.566 | 0.567 | 0.568 | 0.569 | 0.57 | 0.571 | 0.573 | 0.574 | 0.575 | 0.576 | 0.577 | 0.578 | 0.579 |
      Level 3: | 0.58 | 0.581 | 0.582 | 0.583 | 0.584 | 0.585 | 0.586 | 0.587 | 0.588 | 0.589 | 0.59 | 0.591 | 0.592 | 0.593 | 0.594 | 0.595 |
      Level 3: | 0.596 | 0.597 | 0.598 | 0.6 | 0.602 | 0.603 | 0.604 | 0.605 | 0.606 | 0.607 | 0.608 | 0.609 | 0.61 | 0.611 | 0.612 | 0.613 |
      Level 3: | 0.614 | 0.615 | 0.616 | 0.617 | 0.618 | 0.619 | 0.62 | 0.621 | 0.622 | 0.623 | 0.624 | 0.625 | 0.626 | 0.627 | 0.628 | 0.629 |
      Level 3: | 0.63 | 0.631 | 0.632 | 0.633 | 0.634 | 0.635 | 0.636 | 0.639 | 0.64 | 0.641 | 0.642 | 0.643 | 0.644 | 0.645 | 0.646 | 0.648 |
      Level 3: | 0.65 | 0.651 | 0.652 | 0.653 | 0.654 | 0.655 | 0.658 | 0.662 | 0.667 | 0.671 | 0.675 | x | x | x | x | x |
```

As seen from the tables for Experiments 3, 4 & 5, B+ Tree search is significantly faster than brute-force linear search due to its lower running time. This is because for simple linear search, it has to iterate through all the nodes as we have no way of knowing the minimum value. So we had to start from the 0th index and go till the end. On the other hand, B+ tree uses a divide and conquer strategy by dividing search space and eliminating values after the target value is

compared to the current value in the node. This reduces the number of key comparisons, thus reducing the running time.