

# B+ TREE

CSL303 : Database Management System

---

Linux API's Manual

---

**BpTree** – B+ Tree disk-based index for database management systems

## SYNOPSIS

---

```
void init(void);
int writeData(int key, unsigned char* data);
unsigned char* readData(int key);
int deleteData(int key);
unsigned char** readRangeData(int lowerKey, int upperKey, int* n);
void closeIndex(void);
```

## DESCRIPTION

---

The **BpTree** library provides a disk-based B+ tree index implementation for database management systems. It supports efficient insertion, deletion, point queries, and range queries on integer keys with fixed-size (100-byte) data tuples.

The index is persisted to disk using memory-mapped I/O (`mmap`), providing zero-copy access to pages. The implementation uses a page size of 4096 bytes and supports automatic file expansion.

# API REFERENCE

---

## Write Operations

### writeData()

#### Synopsis:

```
int writeData(int key, unsigned char* data);
```

#### Description:

Inserts a key-value pair into the B+ tree index. If the key already exists, the insertion fails and the existing data is not modified. The data must be exactly 100 bytes.

The function automatically handles page splits when leaf nodes become full, propagating splits up the tree as necessary.

#### Parameters:

- **key** – Integer key for indexing (must be unique)
- **data** – Pointer to 100-byte tuple data (must not be NULL)

#### Returns:

- 1 – Insertion successful
- 0 – Insertion failed (duplicate key or internal error)

#### Errors:

- Returns 0 if key already exists (no duplicate keys allowed)
- Returns 0 if memory allocation fails during page split
- Returns 0 if data pointer is NULL

#### Performance:

Average case:  $O(\log n)$

Worst case:  $O(\log n)$  with page splits

#### Example:

```
unsigned char tuple[100];
memset(tuple, 0, 100);
strcpy((char*)tuple, "Sample data");

int result = writeData(42, tuple);
if (result == 1) {
    printf("Insert successful\n");
} else {
    printf("Insert failed\n");
}
```

## Read Operations

readData()

### Synopsis:

```
unsigned char* readData(int key);
```

### Description:

Searches for a key in the B+ tree index and returns a copy of the associated tuple data. The function traverses the tree from root to leaf, performing binary search at each level.

### Parameters:

- key – Integer key to search for

### Returns:

- Pointer to 100-byte tuple data if key exists
- NULL if key does not exist in the index

### Memory Management:

The returned pointer is dynamically allocated and must be freed by the caller using `delete[]` (C++) or `free()` (C).

### Performance:

Average case:  $O(\log n)$

Worst case:  $O(\log n)$

### Thread Safety:

This function is thread-safe for concurrent reads.

### Example:

```
unsigned char* result = readData(42);
if (result != NULL) {
    printf("Found data: %s\n", (char*)result);
    delete[] result; // Don't forget to free!
} else {
    printf("Key not found\n");
}
```

## Delete Operations

```
deleteData()
```

### Synopsis:

```
int deleteData(int key);
```

### Description:

Removes a key and its associated tuple from the B+ tree index. This implementation uses lazy deletion – it removes the entry from the leaf node.

The deleted space will be reclaimed when the page is split or when new entries are inserted.

### Parameters:

- **key** – Integer key to delete

### Returns:

- 1 – Deletion successful
- 0 – Key does not exist in the index

### Side Effects:

- Marks the leaf page as dirty
- Does not perform immediate disk write (uses batch flushing)

### Performance:

Average case:  $O(\log n)$

Worst case:  $O(\log n)$

### Note:

Repeated deletions may lead to low page utilization. Consider rebuilding the index periodically for optimal performance.

### Example:

```
int result = deleteData(42);
if (result == 1) {
    printf("Delete successful\n");
} else {
    printf("Key not found\n");
}
```

## readRangeData()

### Synopsis:

```
unsigned char** readRangeData(int lowerKey,  
                             int upperKey,  
                             int* n);
```

### Description:

Retrieves all tuples with keys in the range [lowerKey, upperKey] (inclusive). The function performs an efficient range scan by traversing the linked leaf nodes.

Results are returned in ascending key order. The count of returned tuples is stored in the integer pointed to by n.

### Parameters:

- lowerKey – Starting key of the range (inclusive)
- upperKey – Ending key of the range (inclusive)
- n – Pointer to integer that will store the count of returned tuples (must not be NULL)

### Returns:

- Array of pointers to 100-byte tuples if keys exist in range
- NULL if no keys exist in the specified range
- Sets \*n to 0 if no keys found

### Memory Management:

Both the array and each tuple pointer must be freed by the caller:

```
for (int i = 0; i < count; i++) {  
    delete[] results[i];  
}  
delete[] results;
```

### Performance:

O(log n + k) where k is the number of keys in range

### Example:

```
int count = 0;  
unsigned char** results = readRangeData(10, 100, &count);  
  
if (results != NULL) {  
    printf("Found %d tuples\n", count);  
    for (int i = 0; i < count; i++) {  
        printf("Tuple %d: %s\n", i, (char*)results[i]);  
        delete[] results[i];  
    }  
    delete[] results;  
} else {  
    printf("No keys in range\n");  
}
```

# CONFIGURATION

---

## Constants (defined in source)

```
const char* DB_FILE = "index.bin";           // Index file name
const int PAGE_SIZE = 4096;                  // Page size in bytes
const int TUPLE_SIZE = 100;                   // Fixed tuple size
const long long MAX_DB_SIZE = 256L*1024*1024;
const INVALID_PAGE_ID = -1;
```

## Modifying Configuration

To change configuration parameters, edit the constants in the source file and recompile.

# TROUBLESHOOTING

---

## Common Issues

### Problem:

Compilation fails with “mmap not found”

### Solution:

Ensure you're on a POSIX-compliant system with proper headers

### Problem:

Segmentation fault on startup

### Solution:

Check file permissions in current directory and ensure sufficient disk space

### Problem:

“Disk Full” error

### Solution:

Index has exceeded the defined limit, increase MAX\_DB\_SIZE constant

### Problem:

Corrupted index file or incorrect root page

### Solution:

Delete `index.bin` and restart with fresh database. The meta page should restore properly on next init.

### Problem:

Linking errors with multiple definition

### Solution:

Ensure proper header guards in all .h files and use `extern "C"` for API functions

## Debugging

Enable debug output by compiling with -g flag and using gdb:

```
$ g++ -std=c++11 -g -o db_engine \
BPlusTree.cpp c_api.cpp DiskManager.cpp driver.cpp
$ gdb ./db_engine
```

## COPYRIGHT

---

This implementation is provided for DBMS assignment by Arpit Kumar, Arpit Bhomia, Ashish Ranjan , Sarthak Gopal Sharma and Abhigyan Sharma.