**Project 4 Design Document**

**Group 1**

Myint Aung

Justine Canlas

Mariah Coughlin

Jordan Hebler

Anuja Modi

Requirements

Generate a B+tree as a file composed of (doubly) linked, fixed-size blocks, of variable-sized records.  
(based on Programming Exercises 10.19 in the Folk textbook).  
  
You will design your B+tree using the specifications in Folk Section 9.10, with the sequence set of pages/blocks (containing records) being doubly linked, as you produced in the final Sequence Set project.  
The B-Tree pages/blocks, growing up from the sequence set leaves, will contain only lists of ordered pairs of the largest keys of each child page/block with the relative block number of that child, sorted by key.  
The B-tree index (growing up from the sequence set leaves) will replace the simple index file (from the previous project), with the crucial factors being that the B-tree index will use the same page/block size as the sequence set, and the B-tree index pages/blocks will be interleaved with the sequence set pages/blocks in the same file.  
  
Teams will each implement a dynamic B+tree Class that generates and uses a paged/blocked B+tree file.  
Only the minimum set of block and record objects will be kept in RAM at any one time.  
  
The same US Postal Code data file will be used (from the previous project),  
however, you should test the flexibility of your design by successfully processing a column-reordered version of the data file, and one with fewer fields.  
  
Consider developing and testing, first, a static version of the B+tree Class which can generate a B+tree file,  
then verify that the file can be used as a sequence set and as an index to access any record through the root via the record's primary key.  
Then, add dynamic functionality to be able to insert and delete records.

You should have a method of printing (dumping) the contents and shape of your B-tree file so that you can determine whether your implementations for building, insertion and deletion are functioning properly. This dump should be sufficiently readable to help facilitate debugging. Write a test driver that demonstrates that your B+tree Class and file work properly and robustly.

**Implementation**

According to the requirements (objectives) for this Project 4, we have successfully implemented a B+ Tree that generates and uses a page/blocked B+ tree file.

As mentioned in the requirements (objectives) of the project, we have implemented the B+ Tree functionalities in a way that it will go through the program by starting with getting the tail block number and traverse through the block file as it adds the highest key to the B+ Tree index. Then, it will build and update the header file to include the root for it to have access to the root. For it to have access to any record through the root, it will go through the keys again to find the corresponding zip code via the record’s primary key.

Dynamically, we can add records into and remove records from the B+ tree. If we do add values into the B+ tree, it is not required for the B+ tree to be updated (splitting) as the program will determine if the added values are the highest key or not. If it is not the highest key, there is no need to add/change the index. On the other hand, if overflow occurs to one of the blocks, we will create a new block, add the new block, and update the two indexes. In the same way, if we do add records to the block file, since it is the same as adding indexes to the B+ Tree, the node will split if it becomes full. For the remove function, we will pass a record to the function and if that record is found from the file, it will be removed from the file.

As required, we have included a function that will print (dump) the contents and shape of our B+ tree file so that the other functionalities such as building, insertion and deletion can be demonstrated clearly. All the source files are tagged extensively to ensure Doxygen will work better with the files. We will be creating the user guide and finalize the design document after we have completed the program. After that we will make sure to create a test document and conduct a couple of tests to ensure that the program is working as intended. The script file is also included with all the test runs.

Below are all the files and functions that we are going to be implementing in this program:

**BPTree**

This B+ Tree class can take a blocked sequence set file and create a B+ Tree index. This class can be used for searching and displaying Location objects, sorting Location objects, and adding/deleting records from the blocked/index file.

**class IndexNode**

**public:**

IndexNode() - constructor

**class BPTree**

**public:**

BPTree() - default constructor for BPTree

IndexNode \*getRoot() - Create a B+ Tree index from the Blocked sequence set file

void createIndex(string file) – Create a B+ Tree index from the Blocked sequence set file

void lookUpKey(string) – A key is looked up in the blocked/index file and outputted if found

void sortRecords() - Sort records

void display(IndexNode \*) – Outputs the B+ Tree

void add(Location) - Add new record to the blocked/index file

void remove(string) – Delete a record from the blocked/index file

**private:**

IndexNode \*root - Root Node of the Tree

void clear() - Clear function to reset the Tree

void insertInternal(Pair, IndexNode \*, IndexNode \*) - Helper function for internal insertion

IndexNode \*findParent(IndexNode \*, IndexNode \*) - The parent of the indexnode is found

void writeIndex(string file) - Writes the indexnodes to the blocked sequence file

void insert(Pair) - Inserting the pair to the B+ Tree

void writeNodes(IndexNode \*, int, int, int, ofstream &) - Helper function to write the IndexNodes to the blocked sequence file

Pair \*findBlock(int) - Find block that the key passed in may be contained in

**Block**

A blocked sequence set file object can read from a file, sort records, find a specific record, and add/or delete records from a block.

**class Block**

**private:**

void readHeader(istream &infile) - Read header

void writeHeader(ostream &outfile) - Write header

void writeBlockInfo(BlockNode<dataType> curBlock, ostream &outfile) - Write block information

int getHeadBlockNumber() const - Retrieve headBlockNumber

int findDesiredBlock(const string &key) - Find fit block

void updateIndexFile() - Update the index file

void redistribution(BlockNode<dataType> &block1, BlockNode<dataType> &block2) - Merge 2 block to prevent underflow

**public:**

Block(const string &filename) - Default constructor

BlockNode<dataType> readBlock(const int &pos) - Read block at a position

void readFile(const string &filename) - Read block from file

int getBlockCount() const - Count the number of nodes

bool addData(const dataType &record, BlockNode<dataType>) - Add new record to a the block

bool removeData(const string &key, BlockNode<dataType>) - Delete a record from the block

int str2int (const string &s) const - string to int converter

void updateBlockFile(const BlockNode<dataType> &mainBlock, const BlockNode<dataType> &newBlock, const BlockNode<dataType> &sMainBlock) - Update the block file with new information

void sortRecords() - Sort records

bool findRecord(const string &keyStr, BlockNode<dataType> b) - Search for a record

void logicalDump() - Dump method showing logical ordering

void physicalDump() - Dump method showing physical ordering

**BlockNode**

A block node contains the data in each block and has 2 pointers which point to its adjacent blocks.

**class BlockNode**

private:

int str2int (const string &s) const - string to int converter

public:

BlockNode() – Default constructor

BlockNode(const int &bNum) - Constructor with block number

BlockNode(const int &bNum, const int &sBlock, const int &pBlock) – Constructor

int getBlockNumber() const - Get block number

int getDataSize() const - Get block's size

int getNumRecs() const - Get number of records

dataType getData(const int num) const - Get i-th data

bool addData(const dataType &anEntry) - Add new data entry

bool removeData(const int &position) - Remove a data entry by position

int getSBlockNumber() const - Get succeeded block

int getPBlockNumber() const - Get preceded block

void setSBlock(const int &val) - Set succeeded block

void setPBlock(const int &val) - Set preceded block

**DelimBuffer**

This class reads in each field from the provided file or instream. This class reads in each block header and its corresponding indexes. This class is used to pack and unpack from the record blocks.

**class DelimBuffer**

**private:**

void setNumRecs(const int &val) - Set the current block's size to a certain value

void setBlockNumber(const int &val) - Set the current block's number to a certain value

**public:**

DelimBuffer(char ind = ‘,’) – This is the default constructor

bool read(istream& infile) – This reads from the file stream block by block

bool unpackField(string &aStr) – This gets the next field from read file stream

int getBlockNumber() const – This gets the current block’s number

int getNumRecs() const – This gets the current block’s size

int getSBlockNumber() const – This gets the succeeded block’s number

int getPBlockNumber() const – This gets the preceded block’s number

void increment() – This increments the nextChar index

**Location**

The Location class describes a record type that stores a Zipcode, City, County, and Coordinates.

**class Location**

**public:**

Location() - Default constructor

Location(const Location &loc) - Copy constructor

Location(const string &s) – Constructor

string getKey() const - Access primary key

string getZipCode() const - Access zip code

void setZipCode(const string &val) - Set zip code

string getPlace() const - Access place name

void setPlace(const string &val) - Set place name

string getState() const - Access state

void setState(const string &val) - Set state

string getCounty() const - Access county

void setCounty(const string &val) - Set county

float getLat() const - Access latitude

void setLat(const float &val) - Set latitude

float getLong() const - Access longitude

void setLong(const float &val) - Set longitude

bool unpack(DelimBuffer &buffer) This method unpacks data from a buffer and pass it into the Location object

void operator= (const Location &loc) - Overloaded assignment operator for a location object

bool operator< (const Location &loc) const - Less than comparison operator overloaded

bool operator> (const Location &loc) const - Greater than comparison operator overloaded

friend ostream& operator<< (ostream& out, const Location &loc) - outstream operator overloaded

getWeight() const - This method get the weight of the Location object

**IndexBuffer**

This class reads in each field from the provided file or instream. This class reads in each block header and its corresponding indexes. This class packs and unpack from the index blocks.

**class IndexBuffer**

**private:**

void setNumIndexes(const int &val) - Set the current block's size to a certain value

void setBlockNumber(const int &val) - Set the current block's number to a certain value

**public:**

IndexBuffer(char ind = ‘,’) – This is the default constructor

bool read(istream& infile) – This reads from the file stream block by block

bool unpackField(string &aStr) – This gets the next field from read file stream

int getBlockNumber() const – This gets the current block’s number

int getNumIndexes() const – This gets the current block’s size

int getSBlockNumber() const – This gets the succeeded block’s number

int getPBlockNumber() const – This gets the preceded block’s number

void increment() – This increments the nextChar index