**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**

We have implemented this by first creating sequence set of blocks, then going through the file and adding an index node

**BPTree**

This B+ tree class can take a blocked sequence set file and create a B+ tree index. This index can be used for searching and displaying Location objects.

Class BPTree

**Public**

BPTree() - default constructor for BPTree

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

void writeIndex(string file) – This writes the indexnodes to the blocked sequence file

bool lookUpKey(string) – In this a key is looked up in the blocked/index file

void display(IndexNode \*) – This prints the tree

indexNode \*getRoot() - Accessor for the root node

void insert(Pair) – This Inserts the pair to the B+ Tree

**Class 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

**Public**

block(const string &filename) – this is a default constructor

blockNode<datatype> readBlock(const int &pos) – This reads block at a position

void readFile(const string &filename) – This reads block from a file

int getBlockCount() const – This counts the number of nodes

bool addData(const datatype &record) – This adds new record to the block

bool removeData(const string &key) – This deletes a record from the block

int str2int(const string &s) const – This function is to convert string to int

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

void sortRecords() – This sorts records

bool findRecord(const string &keyStr) – This searches for a record

void logicalDump() – This dumps method showing logical ordering

void physicalDump() – This dumps method showing physical ordering

**Class IndexBuffer**

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

Class IndexBuffer

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