B+ Tree Implementation

Generated by Doxygen 1.9.8

1 Class Index 1

1.1 Class List . . . 1

2 File Index 3

2.1 File List . . . 3

3 Class Documentation 5

3.1 BlockBuffer Class Reference . . . 5

3.1.1 Detailed Description . . . 8

3.1.2 Constructor & Destructor Documentation . . . 8

3.1.2.1 BlockBuffer() . . . 8

3.1.3 Member Function Documentation . . . 8

3.1.3.1 clear() . . . 8

3.1.3.2 getCurRBN() . . . 9

3.1.3.3 getLargestKey() . . . 9

3.1.3.4 getNextRBN() . . . 9

3.1.3.5 getNumRecords() . . . 9

3.1.3.6 getPrevRBN() . . . 10

3.1.3.7 isOverFilled() . . . 10

3.1.3.8 isUnderFilled() . . . 10

3.1.3.9 mergeBuffer() . . . 10

3.1.3.10 pack() . . . 11

3.1.3.11 read() . . . 11

3.1.3.12 redistributeBuffer() . . . 11

3.1.3.13 removeRecord() . . . 12

3.1.3.14 setCurRBN() . . . 12

3.1.3.15 setNextRBN() . . . 12

3.1.3.16 setNumRecords() . . . 13

3.1.3.17 setPrevRBN() . . . 13

3.1.3.18 sortBuffer() . . . 13

3.1.3.19 splitBuffer() . . . 13

3.1.3.20 unpack() . . . 14

3.1.3.21 write() . . . 14

3.1.4 Member Data Documentation . . . 15

3.1.4.1 blockSize . . . 15

3.1.4.2 buffer . . . 15

3.1.4.3 curRBN . . . 15

3.1.4.4 minimumBlockCapacity . . . 15

3.1.4.5 nextRBN . . . 15

3.1.4.6 numRecords . . . 15

3.1.4.7 prevRBN . . . 16

3.2 BTreeFile Class Reference . . . 16

3.2.1 Detailed Description . . . 18

3.2.2 Constructor & Destructor Documentation . . . 19
3.2.2.1 BTreeFile() . . . 19
3.2.2.2 ~BTreeFile() . . . 19
3.2.3 Member Function Documentation . . . 19
3.2.3.1 closeFile() . . . 19
3.2.3.2 displayExtrema() . . . 19
3.2.3.3 displayNode() . . . 20
3.2.3.4 displaySequenceSet() . . . 20
3.2.3.5 displayTree() . . . 21
3.2.3.6 findLeafNode() . . . 21
3.2.3.7 findParentNode() . . . 21
3.2.3.8 flushData() . . . 22
3.2.3.9 handleMerge() . . . 22
3.2.3.10 handleNonRootSplit() . . . 22
3.2.3.11 handleRootSplit() . . . 23
3.2.3.12 insert() . . . 23
3.2.3.13 openFile() . . . 23
3.2.3.14 remove() . . . 24
3.2.3.15 search() . . . 24
3.2.4 Member Data Documentation . . . 24
3.2.4.1 file . . . 24
3.2.4.2 filename . . . 25
3.2.4.3 headerBuffer . . . 25
3.2.4.4 height . . . 25
3.2.4.5 order . . . 25
3.2.4.6 root . . . 25
3.3 BTreeIndexBuffer Class Reference . . . 26
3.3.1 Detailed Description . . . 27
3.3.2 Constructor & Destructor Documentation . . . 27
3.3.2.1 BTreeIndexBuffer() . . . 27
3.3.3 Member Function Documentation . . . 27
3.3.3.1 clear() . . . 27
3.3.3.2 pack() . . . 28
3.3.3.3 read() . . . 28
3.3.3.4 unpack() . . . 28
3.3.3.5 write() . . . 29
3.3.4 Member Data Documentation . . . 29
3.3.4.1 blockSize . . . 29
3.3.4.2 buffer . . . 29
3.3.4.3 lengthSeparators . . . 30
3.3.4.4 minimumBlockCapacity . . . 30
3.3.4.5 numSeparators . . . 30

3.4 BTreeNode Class Reference . . . 30
3.4.1 Detailed Description . . . 33
3.4.2 Constructor & Destructor Documentation . . . 33
3.4.2.1 BTreeNode() . . . 33
3.4.3 Member Function Documentation . . . 33
3.4.3.1 getChildren() . . . 33
3.4.3.2 getCurRBN() . . . 34
3.4.3.3 getIsLeaf() . . . 34
3.4.3.4 getLargestKey() . . . 34
3.4.3.5 getNextChild() . . . 34
3.4.3.6 getNextRBN() . . . 35
3.4.3.7 getPrevRBN() . . . 35
3.4.3.8 insertKeyAndChildren() . . . 35
3.4.3.9 insertRecord() . . . 36
3.4.3.10 isOverFilled() . . . 36
3.4.3.11 isUnderFilled() . . . 36
3.4.3.12 merge() . . . 36
3.4.3.13 print() . . . 37
3.4.3.14 read() . . . 37
3.4.3.15 removeKeyAndChildren() . . . 38
3.4.3.16 removeRecord() . . . 38
3.4.3.17 retrieveRecord() . . . 38
3.4.3.18 setCurRBN() . . . 39
3.4.3.19 setIsLeaf() . . . 39
3.4.3.20 setNextRBN() . . . 39
3.4.3.21 setPrevRBN() . . . 40
3.4.3.22 split() . . . 40
3.4.3.23 write() . . . 40
3.4.4 Member Data Documentation . . . 41
3.4.4.1 blockBuffer . . . 41
3.4.4.2 bTreeIndexBuffer . . . 41
3.4.4.3 children . . . 41
3.4.4.4 curRBN . . . 41
3.4.4.5 isLeaf . . . 42
3.4.4.6 keys . . . 42
3.4.4.7 maxKeys . . . 42
3.4.4.8 minKeys . . . 42
3.4.4.9 numKeys . . . 42
3.5 HeaderBuffer Class Reference . . . 43
3.5.1 Detailed Description . . . 44
3.5.2 Constructor & Destructor Documentation . . . 44
3.5.2.1 HeaderBuffer() . . . 44

3.5.3 Member Function Documentation . . . 45
3.5.3.1 readHeader() . . . 45
3.5.3.2 writeHeader() . . . 45
3.5.4 Member Data Documentation . . . 45
3.5.4.1 blockCount . . . 45
3.5.4.2 blockSize . . . 46
3.5.4.3 fileType . . . 46
3.5.4.4 headerRecordSize . . . 46
3.5.4.5 minimumBlockCapacity . . . 46
3.5.4.6 rbnActive . . . 46
3.5.4.7 rbnAvail . . . 46
3.5.4.8 recordCount . . . 47
3.5.4.9 recordFieldCount . . . 47
3.5.4.10 recordFieldsType . . . 47
3.5.4.11 recordFormat . . . 47
3.5.4.12 recordPrimaryKey . . . 47
3.5.4.13 recordSizeDigits . . . 47
3.5.4.14 recordSizeFormat . . . 48
3.5.4.15 stale . . . 48
3.5.4.16 version . . . 48
3.6 Record Struct Reference . . . 48
3.6.1 Detailed Description . . . 49
3.6.2 Constructor & Destructor Documentation . . . 49
3.6.2.1 Record() . . . 49
3.6.3 Member Function Documentation . . . 50
3.6.3.1 display() . . . 50
3.6.4 Member Data Documentation . . . 50
3.6.4.1 County . . . 50
3.6.4.2 Lat . . . 50
3.6.4.3 Long . . . 50
3.6.4.4 PlaceName . . . 51
3.6.4.5 State . . . 51
3.6.4.6 ZipCode . . . 51
3.7 RecordBuffer Class Reference . . . 51
3.7.1 Detailed Description . . . 53
3.7.2 Constructor & Destructor Documentation . . . 53
3.7.2.1 RecordBuffer() . . . 53
3.7.3 Member Function Documentation . . . 53
3.7.3.1 clear() . . . 53
3.7.3.2 getBufferSize() . . . 54
3.7.3.3 getRecordKey() . . . 54
3.7.3.4 pack() . . . 54

3.7.3.5 read() . . . 54
3.7.3.6 unpack() . . . 55
3.7.3.7 write() . . . 55
3.7.4 Member Data Documentation . . . 55
3.7.4.1 buffer . . . 55
3.7.4.2 deliminator . . . 56
3.7.4.3 maxBufferSize . . . 56
3.7.4.4 nextByte . . . 56
3.8 RecordFile Class Reference . . . 56
3.8.1 Detailed Description . . . 58
3.8.2 Constructor & Destructor Documentation . . . 58
3.8.2.1 RecordFile() . . . 58
3.8.2.2 ~RecordFile() . . . 59
3.8.3 Member Function Documentation . . . 59
3.8.3.1 closeFile() . . . 59
3.8.3.2 createLengthIndicatedFile() . . . 59
3.8.3.3 openFile() . . . 59
3.8.3.4 read() . . . 60
3.8.3.5 write() . . . 60
3.8.4 Member Data Documentation . . . 61
3.8.4.1 file . . . 61
3.8.4.2 headerBuffer . . . 61
3.9 StateDatabase Class Reference . . . 61
3.9.1 Detailed Description . . . 62
3.9.2 Member Function Documentation . . . 62
3.9.2.1 printStateInfo() . . . 62
3.9.2.2 processRecord() . . . 62
3.9.3 Member Data Documentation . . . 63
3.9.3.1 stateInfoMap . . . 63
3.10 StateExtrema Struct Reference . . . 63
3.10.1 Detailed Description . . . 64
3.10.2 Member Data Documentation . . . 65
3.10.2.1 eastLong . . . 65
3.10.2.2 eastZip . . . 65
3.10.2.3 northLat . . . 65
3.10.2.4 northZip . . . 65
3.10.2.5 southLat . . . 65
3.10.2.6 southZip . . . 65
3.10.2.7 westLong . . . 66
3.10.2.8 westZip . . . 66

4 File Documentation **67**

4.1 BlockBuffer.cpp File Reference . . . 67
4.1.1 Detailed Description . . . 67
4.2 BlockBuffer.cpp . . . 68
4.3 BlockBuffer.h File Reference . . . 71
4.3.1 Detailed Description . . . 73
4.4 BlockBuffer.h . . . 73
4.5 BTreeFile.cpp File Reference . . . 74
4.6 BTreeFile.cpp . . . 74
4.7 BTreeFile.h File Reference . . . 79
4.7.1 Detailed Description . . . 80
4.8 BTreeFile.h . . . 80
4.9 BTreeIndexBuffer.cpp File Reference . . . 81
4.9.1 Detailed Description . . . 81
4.10 BTreeIndexBuffer.cpp . . . 82
4.11 BTreeIndexBuffer.h File Reference . . . 83
4.11.1 Detailed Description . . . 85
4.12 BTreeIndexBuffer.h . . . 85
4.13 BTreeNode.cpp File Reference . . . 85
4.13.1 Detailed Description . . . 86
4.14 BTreeNode.cpp . . . 86
4.15 BTreeNode.h File Reference . . . 89
4.16 BTreeNode.h . . . 90
4.17 HeaderBuffer.cpp File Reference . . . 91
4.17.1 Detailed Description . . . 92
4.18 HeaderBuffer.cpp . . . 92
4.19 HeaderBuffer.h File Reference . . . 94
4.19.1 Detailed Description . . . 95
4.20 HeaderBuffer.h . . . 96
4.21 main.cpp File Reference . . . 96
4.21.1 Detailed Description . . . 97
4.21.2 Function Documentation . . . 97
4.21.2.1 addRecords() . . . 97
4.21.2.2 deleteRecords() . . . 98
4.21.2.3 main() . . . 98
4.21.2.4 processCommandLine() . . . 98
4.21.2.5 searchIndex() . . . 99
4.22 main.cpp . . . 99
4.23 Record.cpp File Reference . . . 102
4.23.1 Detailed Description . . . 102
4.24 Record.cpp . . . 103
4.25 Record.h File Reference . . . 103
4.25.1 Detailed Description . . . 104

4.26 Record.h . . . 105
4.27 RecordBuffer.cpp File Reference . . . 105
4.27.1 Detailed Description . . . 105
4.28 RecordBuffer.cpp . . . 106
4.29 RecordBuffer.h File Reference . . . 107
4.29.1 Detailed Description . . . 108
4.30 RecordBuffer.h . . . 108
4.31 RecordFile.cpp File Reference . . . 109
4.31.1 Detailed Description . . . 109
4.32 RecordFile.cpp . . . 110
4.33 RecordFile.h File Reference . . . 111
4.33.1 Detailed Description . . . 112
4.34 RecordFile.h . . . 113
4.35 StateDatabase.cpp File Reference . . . 113
4.35.1 Detailed Description . . . 114
4.36 StateDatabase.cpp . . . 114
4.37 StateDatabase.h File Reference . . . 115
4.38 StateDatabase.h . . . 116
4.39 StateExtrema.h File Reference . . . 116
4.39.1 Detailed Description . . . 117
4.40 StateExtrema.h . . . 118

Index **119**

Chapter 1

Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

BlockBuffer
: A class that has functions to read, write, pack, and unpack block files 5

BTreeFile
: A class for building the Btree File 16

BTreeIndexBuffer
: A class for managing a buffer for a files header information 26

BTreeNode 30

HeaderBuffer
: A class for managing a buffer for a files header information 43

Record
: A struct for storing information pertaining to a zip code record 48

RecordBuffer
: A class for managing a recordBuffer 51

RecordFile
: A class for managing the file for a recordBuffer 56

StateDatabase
: A class to store a collection of StateExtrema objects 61

StateExtrema
: A struct for storing information about the extrema of a state 63

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

BlockBuffer.cpp
: Implementation file for the BlockBuffer class . . . 67

BlockBuffer.h
: Header file for the BlockBuffer class . . . 71

BTreeFile.cpp . . . 74

BTreeFile.h
: Header file for the BTreeFile class . . . 79

BTreeIndexBuffer.cpp
: Implementation file for the BTreeIndexBuffer class . . . 81

BTreeIndexBuffer.h
: Header file for the BTreeIndexBuffer class . . . 83

BTreeNode.cpp
: Implementation file for the BTreeNode class . . . 85

BTreeNode.h . . . 89

HeaderBuffer.cpp
: Implementation file for the HeaderBuffer class . . . 91

HeaderBuffer.h
: Header file for the HeaderBuffer class . . . 94

main.cpp
: Main program file for processing CSV data related to ZIP code records using a B+ tree . . . 96

Record.cpp
: Implementation file for the Record struct . . . 102

Record.h
: Header file for the Record struct . . . 103

RecordBuffer.cpp
: Implementation file for the RecordBuffer class . . . 105

RecordBuffer.h
: Header file for the RecordBuffer class . . . 107

RecordFile.cpp
: Implementation file for the RecordFile class . . . 109

RecordFile.h
: Header file for the RecordFile class . . . 111

StateDatabase.cpp
: Implementation file for the StateDatabase class . . . 113

StateDatabase.h . . . 115

StateExtrema.h
: Header file for the StateExtrema struct . . . 116

Chapter 3

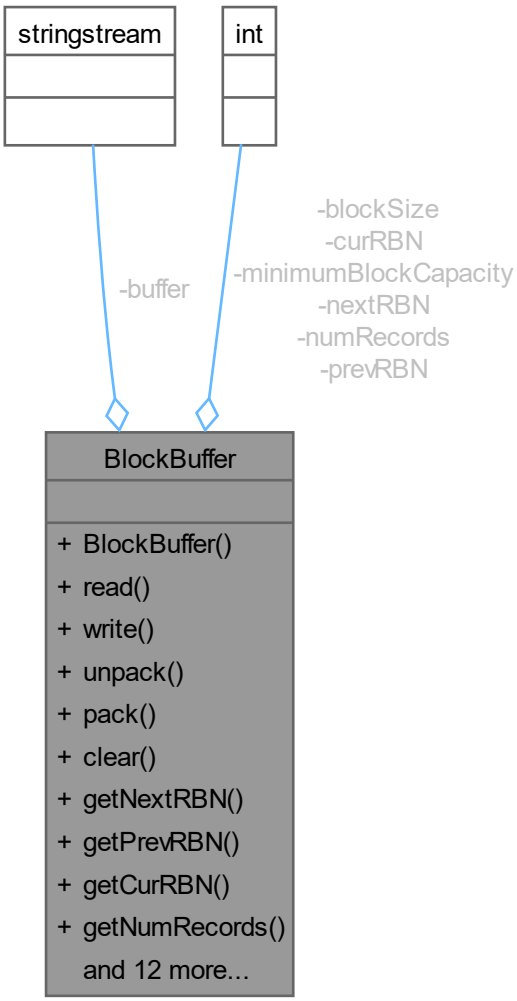
Class Documentation

3.1 BlockBuffer Class Reference

A class that has functions to read, write, pack, and unpack block files.

`#include <BlockBuffer.h>`

Collaboration diagram for BlockBuffer:



Public Member Functions

- BlockBuffer (int blockSz=512, int minCap=256)

  BlockBuffer Constructor.
- int read (std::istream &stream, int headerRecordSize, int blockNumber=-1)

  Read Function, reads one block of data and stores it in the buffer at a time.
- int write (std::ostream &stream, int headerRecordSize, int blockNumber=-1)

  Write Function, writes the data currently stored in buffer.
- int unpack (RecordBuffer &rBuf)

  Unpack Function.
- int pack (RecordBuffer &rBuf)

  Pack Function.
- void clear ()

  Clear Function, clears buffer.

- int getNextRBN ()

  Getter Function for NextRBN variable.
- int getPrevRBN ()

  Getter Function for PrevRBN variable.
- int getCurRBN ()

  Getter Function for curRBN variable.
- int getNumRecords ()

  Getter Function for NumRecords variable.
- void setNextRBN (int rbn)

  Setter Function for NextRBN variable.
- void setPrevRBN (int rbn)

  Setter Function for PrevRBN variable.
- void setCurRBN (int rbn)

  Setter Function for CurRBN variable.
- void setNumRecords (int num)

  Setter Function for NumRecords variable.
- bool isOverFilled ()

  Checks if buffer is too full.
- bool isUnderFilled ()

  Checks if buffer is under minimum requirements.
- void splitBuffer (BlockBuffer &newBlockBuffer)

  Splits the records of the current buffer into the passed in buffer.
- void mergeBuffer (BlockBuffer &newBlockBuffer)

  Merges the records from passed in buffer into the current buffer.
- void redistributeBuffer (BlockBuffer &newBlockBuffer)

  Redistributes the records from passed in buffer into the current buffer until min capacity is reached.
- int getLargestKey ()

  Gets the largest key from the buffer.
- int sortBuffer ()

  Sorts the buffer based on key.
- int removeRecord (int key)

  Removes a record from the buffer.

Private Attributes

- std::stringstream buffer

  Used to help in the pack and unpack functions.
- int blockSize

  Stores the block size as int.
- int minimumBlockCapacity

  Stores the minimum block size as int.
- int numRecords

  Stores the Number of Records as int.
- int prevRBN

  Keeps state of block next block number to read.
- int nextRBN

  Keeps state of block previous block number read.
- int curRBN

  Keeps state of block current block number.

3.1.1 Detailed Description

A class that has functions to read, write, pack, and unpack block files.

: Includes: The ability to read and write blocks one block at a time. Assumes: all references to other buffers to be correct and working.

Definition at line 24 of file BlockBuffer.h.

3.1.2 Constructor & Destructor Documentation

3.1.2.1 BlockBuffer()

```
BlockBuffer::BlockBuffer (
            int blockSz = 512,
            int minCap = 256 )
```

BlockBuffer Constructor.

Parameters

blockSz	Size as an integer.
minCap	minimum block capacity

Postcondition

Class is initialized

Definition at line 13 of file BlockBuffer.cpp.

3.1.3 Member Function Documentation

3.1.3.1 clear()

```
void BlockBuffer::clear ( )
```

Clear Function, clears buffer.

Returns

nothing.

Definition at line 138 of file BlockBuffer.cpp.

3.1.3.2 getCurRBN()

```
int BlockBuffer::getCurRBN ( )
```

Getter Function for curRBN variable.

Returns

integer value of curRBN variable.

Definition at line 151 of file BlockBuffer.cpp.

3.1.3.3 getLargestKey()

```
int BlockBuffer::getLargestKey ( )
```

Gets the largest key from the buffer.

Returns

The int value of the largest key in buffer.

Definition at line 229 of file BlockBuffer.cpp.

3.1.3.4 getNextRBN()

```
int BlockBuffer::getNextRBN ( )
```

Getter Function for NextRBN variable.

Returns

integer value of NextRBN variable.

Definition at line 143 of file BlockBuffer.cpp.

3.1.3.5 getNumRecords()

```
int BlockBuffer::getNumRecords ( )
```

Getter Function for NumRecords variable.

Returns

integer value of NumRecords variable.

Definition at line 155 of file BlockBuffer.cpp.

3.1.3.6 getPrevRBN()

```
int BlockBuffer::getPrevRBN ( )
```

Getter Function for PrevRBN variable.

Returns

integer value of PrevRBN variable.

Definition at line 147 of file BlockBuffer.cpp.

3.1.3.7 isOverFilled()

```
bool BlockBuffer::isOverFilled ( )
```

Checks if buffer is too full.

Returns

true if buffer contains more bytes than specified block size, false otherwise.

Definition at line 175 of file BlockBuffer.cpp.

3.1.3.8 isUnderFilled()

```
bool BlockBuffer::isUnderFilled ( )
```

Checks if buffer is under minimum requirements.

Returns

true if buffer contains less bytes than specified minimum block, false otherwise.

Definition at line 179 of file BlockBuffer.cpp.

3.1.3.9 mergeBuffer()

```
void BlockBuffer::mergeBuffer (
            BlockBuffer & newBlockBuffer )
```

Merges the records from passed in buffer into the current buffer.

Parameters

newBlockBuffer	the new block buffer to get data to merge.

Returns

nothing.

Definition at line 211 of file BlockBuffer.cpp.

3.1.3.10 pack()

```
int BlockBuffer::pack (
            RecordBuffer & rBuf )
```

Pack Function.

Parameters

rBuf	The record buffer to pack data from.

Returns

int value of address of where the record was written.

Definition at line 125 of file BlockBuffer.cpp.

3.1.3.11 read()

```
int BlockBuffer::read (
            std::istream & stream,
            int headerRecordSize,
            int blockNumber = -1 )
```

Read Function, reads one block of data and stores it in the buffer at a time.

Parameters

stream	the input to read from.
headerRecordSize	the size of the header record of file reading from.
blockNumber	the relative blocknumber to read the block from.

Returns

-1 if error, non error the address of of stream location.

Definition at line 22 of file BlockBuffer.cpp.

3.1.3.12 redistributeBuffer()

```
void BlockBuffer::redistributeBuffer (
            BlockBuffer & newBlockBuffer )
```

Redistributes the records from passed in buffer into the current buffer until min capacity is reached.

Parameters

newBlockBuffer	the new block buffer to get data to redistribute.

Returns

nothing.

Definition at line 221 of file BlockBuffer.cpp.

3.1.3.13 removeRecord()

```
int BlockBuffer::removeRecord (
            int key )
```

Removes a record from the buffer.

Parameters

key	The record to remove

Returns

-1 on error, 0 otherwise

Definition at line 270 of file BlockBuffer.cpp.

3.1.3.14 setCurRBN()

```
void BlockBuffer::setCurRBN (
            int rbn )
```

Setter Function for CurRBN variable.

Returns

integer value of CurRBN variable.

Definition at line 167 of file BlockBuffer.cpp.

3.1.3.15 setNextRBN()

```
void BlockBuffer::setNextRBN (
            int rbn )
```

Setter Function for NextRBN variable.

Parameters

integer	value of NextRBN variable.

Definition at line 159 of file BlockBuffer.cpp.

3.1.3.16 setNumRecords()

```
void BlockBuffer::setNumRecords (
            int num )
```

Setter Function for NumRecords variable.

Returns

integer value of NumRecords variable.

Definition at line 171 of file BlockBuffer.cpp.

3.1.3.17 setPrevRBN()

```
void BlockBuffer::setPrevRBN (
            int rbn )
```

Setter Function for PrevRBN variable.

Returns

integer value of PrevRBN variable.

Definition at line 163 of file BlockBuffer.cpp.

3.1.3.18 sortBuffer()

```
int BlockBuffer::sortBuffer ( )
```

Sorts the buffer based on key.

Returns

-1 on error, 0 otherwise

Definition at line 248 of file BlockBuffer.cpp.

3.1.3.19 splitBuffer()

```
void BlockBuffer::splitBuffer (
            BlockBuffer & newBlockBuffer )
```

Splits the records of the current buffer into the passed in buffer.

Parameters

newBlockBuffer	the new block buffer to place half the upper half of records into.

Returns

nothing.

Definition at line 183 of file BlockBuffer.cpp.

3.1.3.20 unpack()

```
int BlockBuffer::unpack (
            RecordBuffer & rBuf )
```

Unpack Function.

Parameters

rBuf	The record buffer to unpack data into.

Returns

int value of address where record is read.

Definition at line 99 of file BlockBuffer.cpp.

3.1.3.21 write()

```
int BlockBuffer::write (
            std::ostream & stream,
            int headerRecordSize,
            int blockNumber = -1 )
```

Write Function, writes the data currently stored in buffer.

Parameters

stream	the output to write to.
headerRecordSize	the size of the header record of file writing to.
blockNumber	the relative blocknumber to write the block to.

Returns

-1 if error, non error the address of of stream location.

Definition at line 68 of file BlockBuffer.cpp.

3.1.4 Member Data Documentation

3.1.4.1 blockSize

```
int BlockBuffer::blockSize  [private]
```

Stores the block size as int.

Definition at line 174 of file BlockBuffer.h.

3.1.4.2 buffer

```
std::stringstream BlockBuffer::buffer  [private]
```

Used to help in the pack and unpack functions.

Definition at line 173 of file BlockBuffer.h.

3.1.4.3 curRBN

```
int BlockBuffer::curRBN  [private]
```

Keeps state of block current block number.

Definition at line 179 of file BlockBuffer.h.

3.1.4.4 minimumBlockCapacity

```
int BlockBuffer::minimumBlockCapacity  [private]
```

Stores the minimum block size as int.

Definition at line 175 of file BlockBuffer.h.

3.1.4.5 nextRBN

```
int BlockBuffer::nextRBN  [private]
```

Keeps state of block previous block number read.

Definition at line 178 of file BlockBuffer.h.

3.1.4.6 numRecords

```
int BlockBuffer::numRecords  [private]
```

Stores the Number of Records as int.

Definition at line 176 of file BlockBuffer.h.

3.1.4.7 prevRBN

```
int BlockBuffer::prevRBN  [private]
```

Keeps state of block next block number to read.

Definition at line 177 of file BlockBuffer.h.

The documentation for this class was generated from the following files:

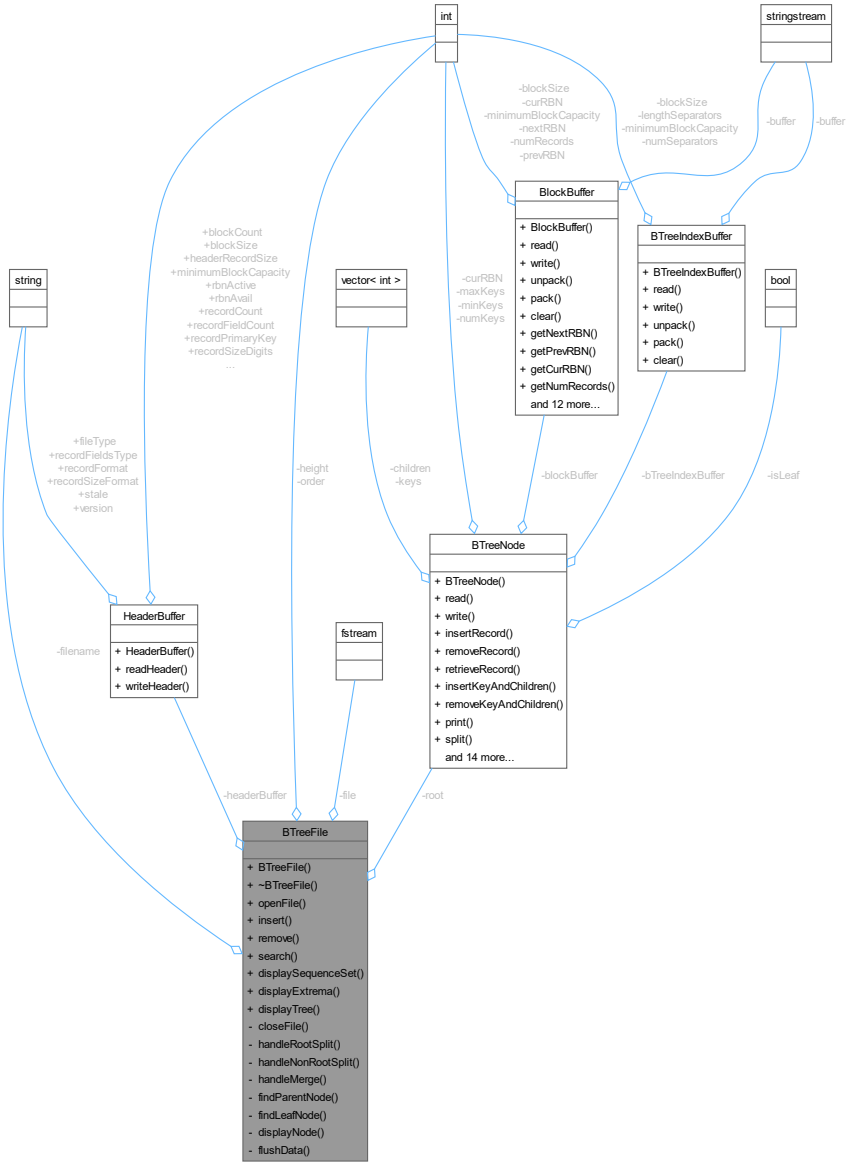
- BlockBuffer.h
- BlockBuffer.cpp

3.2 BTreeFile Class Reference

A class for building the Btree File.

```
#include <BTreeFile.h>
```

Collaboration diagram for BTreeFile:



Public Member Functions

- BTreeFile (HeaderBuffer &hbuf, int order)

  This is the constructor for the BtreeFile, it takes int the header buffer object.
- ~BTreeFile ()

  This is the destructor for the btree object.
- bool openFile (std::string &bTreeFileName)

  This opens the Btree File.
- int insert (RecordBuffer &recordBuffer)

  inserts a record into the btree
- int remove (RecordBuffer &recordBuffer)

  This removes a certain record.
- int search (RecordBuffer &recordBuffer, int key)

Searches the b tree for the record matching the key.

- void displaySequenceSet (std::ostream &ostream)

  Display the tree sequence set.
- void displayExtrema (std::ostream &ostream, std::string state)

  Display the extrema to output stream.
- void displayTree (std::ostream &ostream)

  Display the tree hierarchically.

Private Member Functions

- bool closeFile ()

  This function closes the file.
- void handleRootSplit (int largestKey, BTreeNode ∗leaf, BTreeNode ∗newLeaf)

  This function adds key pairs to reference block, but handles split root.
- void handleNonRootSplit (int largestKey, BTreeNode ∗leaf, BTreeNode ∗newLeaf)

  This function adds key pairs to reference block in cases not involving root node.
- void handleMerge (BTreeNode ∗parent, BTreeNode ∗leaf)

  This function handles the merge between a leaf and another leaf.
- BTreeNode ∗ findParentNode (BTreeNode ∗childNode)

  Find Parent Node Returns parent node if you place in child node.
- BTreeNode ∗ findLeafNode (int key)

  this will find the leaf node based on a key input
- void displayNode (BTreeNode ∗node, std::ostream &ostream, int level, const std::string &prefix)

  Displays the node to the output stream.
- bool flushData ()

  This flushes all the data from the file and places it to btree.

Private Attributes

- HeaderBuffer & headerBuffer

  Stores the reference to the HeaderBuffer object.
- std::fstream file

  Stores the fstream object to the file.
- std::string filename

  Stores the file name for the Btree.
- BTreeNode ∗ root

  This is a pointer the the Btree root Node.
- int order

  This is the order of the btree.
- int height

  This is the height of the btree.

3.2.1 Detailed Description

A class for building the Btree File.

: This class provides methods for working with a file associated with a BlockBuffer. Includes: Methods for opening, closing, reading, creating ,and writing to the file. Assumes:The provided BlockBuffer and HeaderBuffer objects are correctly initialized and valid.

Definition at line 27 of file BTreeFile.h.

3.2.2 Constructor & Destructor Documentation

3.2.2.1 BTreeFile()

```
BTreeFile::BTreeFile (
            HeaderBuffer & hbuf,
            int order )
```

This is the constructor for the BtreeFile, it takes int the header buffer object.

Parameters

HeaderBuffer	Object
order	the order of the b tree

Postcondition

Class is initialized.

Definition at line 12 of file BTreeFile.cpp.

3.2.2.2 ~BTreeFile()

```
BTreeFile::~BTreeFile ( )
```

This is the destructor for the btree object.

Postcondition

Object is properly closed.

Definition at line 18 of file BTreeFile.cpp.

3.2.3 Member Function Documentation

3.2.3.1 closeFile()

```
bool BTreeFile::closeFile ( )  [private]
```

This function closes the file.

Returns

returns false if error closing the file.

Definition at line 59 of file BTreeFile.cpp.

3.2.3.2 displayExtrema()

```
void BTreeFile::displayExtrema (
            std::ostream & ostream,
            std::string state )
```

Display the extrema to output stream.

Parameters

ostream	the stream to display too

Returns

none

Definition at line 157 of file BTreeFile.cpp.

3.2.3.3 displayNode()

```
void BTreeFile::displayNode (
            BTreeNode * node,
            std::ostream & ostream,
            int level,
            const std::string & prefix )  [private]
```

Displays the node to the output stream.

Parameters

node,the	node to display
ostream,the	stream to display too.
level,the	current level the node is on
prefix,the	prefix used to indent node based on level (for tree appearance)

Returns

nothing

Definition at line 343 of file BTreeFile.cpp.

3.2.3.4 displaySequenceSet()

```
void BTreeFile::displaySequenceSet (
            std::ostream & ostream )
```

Display the tree sequence set.

Parameters

ostream	the stream to display too

Returns

none

Definition at line 133 of file BTreeFile.cpp.

3.2.3.5 displayTree()

```
void BTreeFile::displayTree (
            std::ostream & ostream )
```

Display the tree hierarchically.

Parameters

ostream	the stream to display too

Returns

none

Definition at line 180 of file BTreeFile.cpp.

3.2.3.6 findLeafNode()

```
BTreeNode * BTreeFile::findLeafNode (
            int key )  [private]
```

this will find the leaf node based on a key input

Parameters

key	int, This is a zipcode key

Returns

Returns the leaf node based on the key parameter

Definition at line 329 of file BTreeFile.cpp.

3.2.3.7 findParentNode()

```
BTreeNode * BTreeFile::findParentNode (
            BTreeNode * childNode )  [private]
```

Find Parent Node Returns parent node if you place in child node.

Parameters

childNode	the node of which to find parent of

Returns

BtreeNode object, parent node of parameter

Definition at line 309 of file BTreeFile.cpp.

3.2.3.8 flushData()

```
bool BTreeFile::flushData ( )  [private]
```

This flushes all the data from the file and places it to btree.

Returns

Returns False if flush fails, returns True is flush succeeds and file is open

Definition at line 364 of file BTreeFile.cpp.

3.2.3.9 handleMerge()

```
void BTreeFile::handleMerge (
            BTreeNode * parent,
            BTreeNode * leaf )  [private]
```

This function handles the merge between a leaf and another leaf.

Parameters

parent	the parent of leaf to merge
leaf	the leaf to merge with another node

Definition at line 244 of file BTreeFile.cpp.

3.2.3.10 handleNonRootSplit()

```
void BTreeFile::handleNonRootSplit (
            int largestKey,
            BTreeNode * leaf,
            BTreeNode * newLeaf )  [private]
```

This function adds key pairs to reference block in cases not involving root node.

Parameters

largestKey,this	is the largest key in the block,
leaf	cantaining first half of records. Is the original block.
newLeaf	leaf containing other half of records

Definition at line 204 of file BTreeFile.cpp.

3.2.3.11 handleRootSplit()

```
void BTreeFile::handleRootSplit (
            int largestKey,
            BTreeNode * leaf,
            BTreeNode * newLeaf )  [private]
```

This function adds key pairs to reference block, but handles split root.

Parameters

largestKey,this	is the largest key in the block,
leaf	cantaining first half of records. Is the original block.
newLeaf	leaf containing other half of records

Definition at line 185 of file BTreeFile.cpp.

3.2.3.12 insert()

```
int BTreeFile::insert (
            RecordBuffer & recordBuffer )
```

inserts a record into the btree

Parameters

RecordBuffer	record to insert

Returns

returns location of record inserted or -1 if failed

Definition at line 71 of file BTreeFile.cpp.

3.2.3.13 openFile()

```
bool BTreeFile::openFile (
            std::string & bTreeFileName )
```

This opens the Btree File.

Parameters

bTreeFileName	name of b tree file.

Returns

returns int, -1 if error opening the file.

Definition at line 22 of file BTreeFile.cpp.

3.2.3.14 remove()

```
int BTreeFile::remove (
            RecordBuffer & recordBuffer )
```

This removes a certain record.

Parameters

recordBuffer	record to remove

Returns

returns location of record removed, or -1 if failed

Definition at line 93 of file BTreeFile.cpp.

3.2.3.15 search()

```
int BTreeFile::search (
            RecordBuffer & recordBuffer,
            int key )
```

Searches the b tree for the record matching the key.

Parameters

recordBuffer	the object to store record in if found.
key	the zipcode to find.

Returns

-1 on search failed, 0 otherwise

Definition at line 118 of file BTreeFile.cpp.

3.2.4 Member Data Documentation

3.2.4.1 file

```
std::fstream BTreeFile::file  [private]
```

Stores the fstream object to the file.

Definition at line 96 of file BTreeFile.h.

3.2.4.2 filename

```
std::string BTreeFile::filename  [private]
```

Stores the file name for the Btree.

Definition at line 97 of file BTreeFile.h.

3.2.4.3 headerBuffer

```
HeaderBuffer& BTreeFile::headerBuffer  [private]
```

Stores the reference to the HeaderBuffer object.

Definition at line 95 of file BTreeFile.h.

3.2.4.4 height

```
int BTreeFile::height  [private]
```

This is the height of the btree.

Definition at line 100 of file BTreeFile.h.

3.2.4.5 order

```
int BTreeFile::order  [private]
```

This is the order of the btree.

Definition at line 99 of file BTreeFile.h.

3.2.4.6 root

```
BTreeNode* BTreeFile::root  [private]
```

This is a pointer the the Btree root Node.

Definition at line 98 of file BTreeFile.h.

The documentation for this class was generated from the following files:

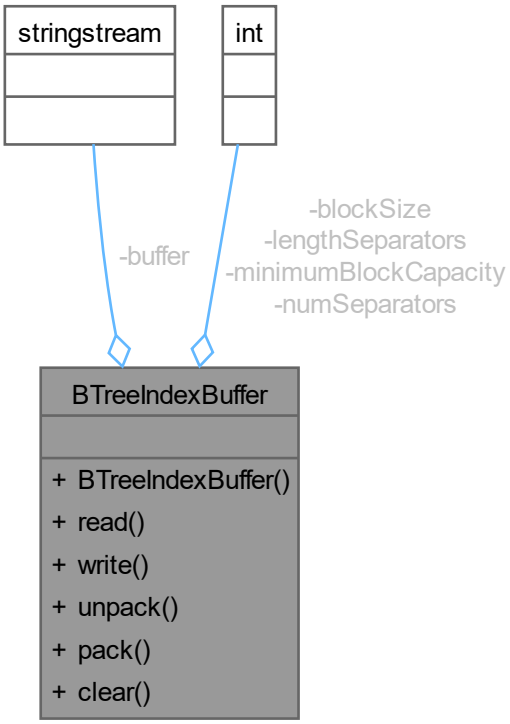
- BTreeFile.h
- BTreeFile.cpp

3.3 BTreeIndexBuffer Class Reference

A class for managing a buffer for a files header information.

`#include <BTreeIndexBuffer.h>`

Collaboration diagram for BTreeIndexBuffer:



Public Member Functions

- BTreeIndexBuffer (int blockSz=512, int minCap=256)

  Constructor for BTreeIndexBuffer.
- int read (std::istream &stream, int headerRecordSize, int blockNumber=-1)

  Reads index data from an input stream.
- int write (std::ostream &stream, int headerRecordSize, int blockNumber=-1)

  Writes index data to an output stream.
- int unpack (std::vector< int > &seperators, std::vector< int > &RBNs)

  Unpacks the buffer content into vectors of separators and RBNs.
- int pack (std::vector< int > seperators, std::vector< int > RBNs)

  Packs vectors of separators and RBNs into the buffer.
- void clear ()

  Clears the buffer.

Private Attributes

- std::stringstream buffer

  Used to help in the pack and unpack functions.
- int blockSize

  Stores the block size as int.
- int minimumBlockCapacity

  Stores the minimum block size as int.
- int numSeparators

  Stores the number of separators.
- int lengthSeparators

  Stores the length of separators.

3.3.1 Detailed Description

A class for managing a buffer for a files header information.

: This class provides methods to read and write header information from/to an input/output stream. Features: Reads and writes header information from/to input/output streams. Assumptions: Assumes the input stream and output stream provided are valid and open.

Definition at line 24 of file BTreeIndexBuffer.h.

3.3.2 Constructor & Destructor Documentation

3.3.2.1 BTreeIndexBuffer()

```
BTreeIndexBuffer::BTreeIndexBuffer (
            int blockSz = 512,
            int minCap = 256 )
```

Constructor for BTreeIndexBuffer.

Parameters

blockSz	The block size.
minCap	The minimum block capacity.

Definition at line 12 of file BTreeIndexBuffer.cpp.

3.3.3 Member Function Documentation

3.3.3.1 clear()

```
void BTreeIndexBuffer::clear ( )
```

Clears the buffer.

Returns

nothing

Definition at line 137 of file BTreeIndexBuffer.cpp.

3.3.3.2 pack()

```
int BTreeIndexBuffer::pack (
            std::vector< int > seperators,
            std::vector< int > RBNs )
```

Packs vectors of separators and RBNs into the buffer.

Parameters

separators	Vector of separators.
RBNs	Vector of RBNs.

Returns

0 on success, -1 if the buffer size exceeds the block size.

Definition at line 107 of file BTreeIndexBuffer.cpp.

3.3.3.3 read()

```
int BTreeIndexBuffer::read (
            std::istream & stream,
            int headerRecordSize,
            int blockNumber = -1 )
```

Reads index data from an input stream.

Parameters

stream	The input stream to read from.
headerRecordSize	The size of the header record.
blockNumber	The block number to read.

Returns

The address where the read operation occurred.

Definition at line 17 of file BTreeIndexBuffer.cpp.

3.3.3.4 unpack()

```
int BTreeIndexBuffer::unpack (
            std::vector< int > & seperators,
            std::vector< int > & RBNs )
```

Unpacks the buffer content into vectors of separators and RBNs.

Parameters

separators	Vector to store the separators.
RBNs	Vector to store the RBNs (relative block numbers).

Returns

0 on success, -1 if the format is incorrect.

Definition at line 80 of file BTreeIndexBuffer.cpp.

3.3.3.5 write()

```
int BTreeIndexBuffer::write (
            std::ostream & stream,
            int headerRecordSize,
            int blockNumber = -1 )
```

Writes index data to an output stream.

Parameters

stream	The output stream to write to.
headerRecordSize	The size of the header record.
blockNumber	The block number to write to.

Returns

The address where the write operation occurred.

Definition at line 54 of file BTreeIndexBuffer.cpp.

3.3.4 Member Data Documentation

3.3.4.1 blockSize

```
int BTreeIndexBuffer::blockSize  [private]
```

Stores the block size as int.

Definition at line 75 of file BTreeIndexBuffer.h.

3.3.4.2 buffer

```
std::stringstream BTreeIndexBuffer::buffer  [private]
```

Used to help in the pack and unpack functions.

Definition at line 74 of file BTreeIndexBuffer.h.

3.3.4.3 lengthSeparators

```
int BTreeIndexBuffer::lengthSeparators  [private]
```

Stores the length of separators.

Definition at line 78 of file BTreeIndexBuffer.h.

3.3.4.4 minimumBlockCapacity

```
int BTreeIndexBuffer::minimumBlockCapacity  [private]
```

Stores the minimum block size as int.

Definition at line 76 of file BTreeIndexBuffer.h.

3.3.4.5 numSeparators

```
int BTreeIndexBuffer::numSeparators  [private]
```

Stores the number of separators.

Definition at line 77 of file BTreeIndexBuffer.h.

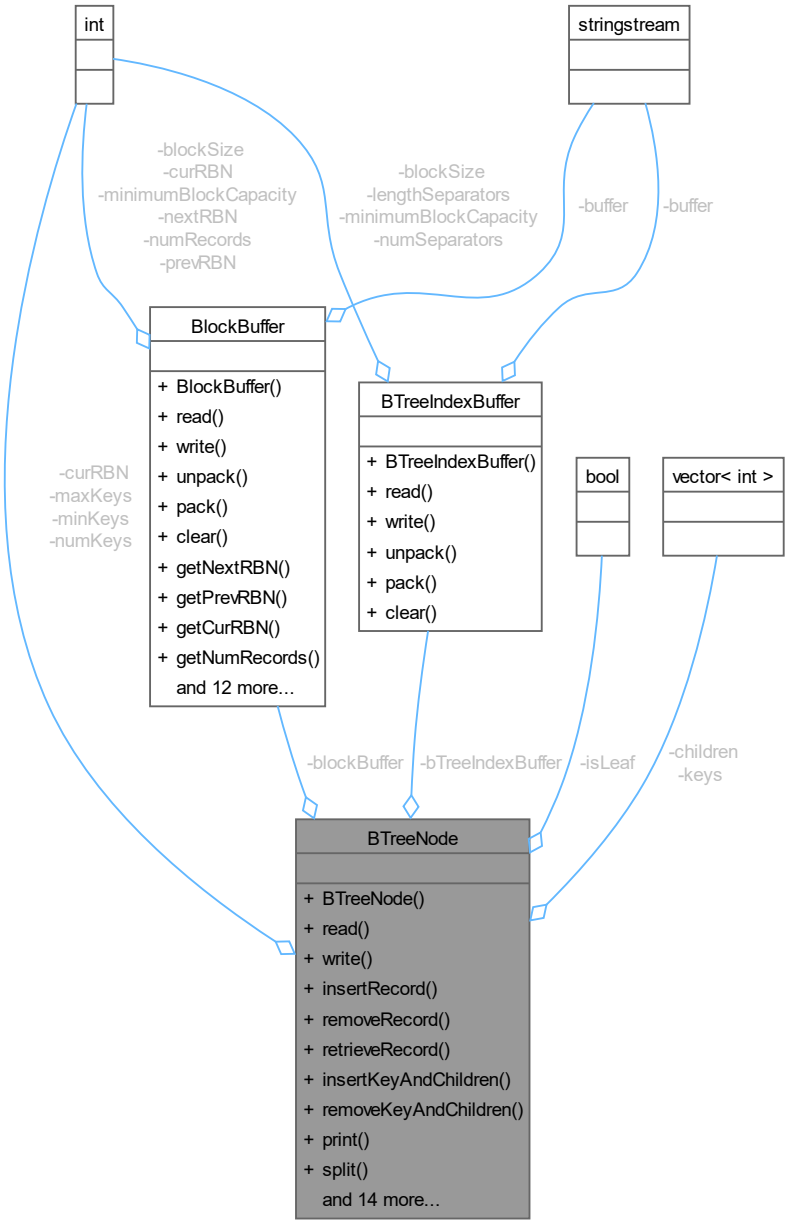
The documentation for this class was generated from the following files:

- BTreeIndexBuffer.h
- BTreeIndexBuffer.cpp

3.4 BTreeNode Class Reference

```
#include <BTreeNode.h>
```

Collaboration diagram for BTreeNode:



Public Member Functions

- BTreeNode (int maxKeys)

  This is the constructor for the BtreeNodes.
- int read (std::istream &stream, int headerRecordSize, int RBN)

  This function reads the node from file.
- int write (std::ostream &stream, int headerRecordSize, int RBN)

  This function writes the node to the file.
- int insertRecord (RecordBuffer &recordBuffer)

This function inserts a record into the B tree.

- int removeRecord (RecordBuffer &recordBuffer)

  This function removes a record from the B tree.
- int retrieveRecord (RecordBuffer &recordBuffer, int key)

  This function retrieves a record from the current buffer.
- int insertKeyAndChildren (int key, int child1, int child2=-1)

  This function inserts a key into the key vector.
- int removeKeyAndChildren (int key, int child)

  This function removes a key into the key vector.
- void print (std::ostream &stream)

  This function prints the node to the output stream.
- int split (BTreeNode *newNode)

  This function shifts a node and it's children toward the root until the B tree becomes balanced.
- int merge (BTreeNode *fromNode)

  This function merges two nodes together.
- int getNextChild (int key)

  This function returns the next node down the tree towards specific key.
- std::vector< int > getChildren ()

  Returns all stored children.
- int getLargestKey ()

  This function returns the largest key in the key vector.
- bool getIsLeaf ()

  This function returns whether a node is a leaf.
- void setIsLeaf (bool isL)

  This function sets a node's isLeaf value to true or false.
- int getCurRBN ()

  This function returns the node's current RBN value.
- void setCurRBN (int rbn)

  Sets the current rbn value.
- int getPrevRBN ()

  This function returns the node's prev RBN value.
- void setPrevRBN (int rbn)

  Sets the previous rbn value.
- int getNextRBN ()

  This function returns the node's next RBN value.
- void setNextRBN (int rbn)

  Sets the next rbn value.
- bool isOverFilled ()

  This function returns whether a node is overfilled.
- bool isUnderFilled ()

  This function returns whether a record is too large.

Private Attributes

- BlockBuffer blockBuffer

  Stores the reference to a block buffer object.
- BTreeIndexBuffer bTreeIndexBuffer

  Stores the reference to a index buffer object.
- int curRBN

  Current RBN in file.

- int maxKeys

  max number of keys to hold
- int minKeys

  min number of keys to hold
- int numKeys

  current number of keys stored
- bool isLeaf

  stores whether node is a leaf or not
- std::vector< int > keys

  Stores the keys of node.
- std::vector< int > children

  Stores the children of node.

3.4.1 Detailed Description

Definition at line 10 of file BTreeNode.h.

3.4.2 Constructor & Destructor Documentation

3.4.2.1 BTreeNode()

```
BTreeNode::BTreeNode (
            int maxKeys )
```

This is the constructor for the BtreeNodes.

Parameters

maxKeys	the maximum number of keys per node

Postcondition

class object is initialized

Definition at line 15 of file BTreeNode.cpp.

3.4.3 Member Function Documentation

3.4.3.1 getChildren()

```
std::vector< int > BTreeNode::getChildren ( )
```

Returns all stored children.

Returns

a vector containing all children

Definition at line 186 of file BTreeNode.cpp.

3.4.3.2 getCurRBN()

```
int BTreeNode::getCurRBN ( )
```

This function returns the node's current RBN value.

Returns

the RBN value as integer

Definition at line 211 of file BTreeNode.cpp.

3.4.3.3 getIsLeaf()

```
bool BTreeNode::getIsLeaf ( )
```

This function returns whether a node is a leaf.

Returns

True is node is leaf false otherwise

Definition at line 163 of file BTreeNode.cpp.

3.4.3.4 getLargestKey()

```
int BTreeNode::getLargestKey ( )
```

This function returns the largest key in the key vector.

Returns

the largest key in node

Definition at line 190 of file BTreeNode.cpp.

3.4.3.5 getNextChild()

```
int BTreeNode::getNextChild (
            int key )
```

This function returns the next node down the tree towards specific key.

Parameters

key	the key to move down the tree towards.

Returns

The next child, an RBN.

Definition at line 171 of file BTreeNode.cpp.

3.4.3.6 getNextRBN()

```
int BTreeNode::getNextRBN ( )
```

This function returns the node's next RBN value.

Returns

the RBN value as integer

Definition at line 231 of file BTreeNode.cpp.

3.4.3.7 getPrevRBN()

```
int BTreeNode::getPrevRBN ( )
```

This function returns the node's prev RBN value.

Returns

the RBN value as integer

Definition at line 219 of file BTreeNode.cpp.

3.4.3.8 insertKeyAndChildren()

```
int BTreeNode::insertKeyAndChildren (
            int key,
            int child1,
            int child2 = -1 )
```

This function inserts a key into the key vector.

Parameters

key	the key to add
child1	the child to add
child2	the child to add

Returns

-1 if failed, 0 otherwise

Definition at line 70 of file BTreeNode.cpp.

3.4.3.9 insertRecord()

```
int BTreeNode::insertRecord (
            RecordBuffer & recordBuffer )
```

This function inserts a record into the B tree.

Parameters

recordBuffer	the record to insert

Returns

-1 on error, 0 otherwise

Definition at line 45 of file BTreeNode.cpp.

3.4.3.10 isOverFilled()

```
bool BTreeNode::isOverFilled ( )
```

This function returns whether a node is overfilled.

Returns

True if its too full, false otherwise

Definition at line 243 of file BTreeNode.cpp.

3.4.3.11 isUnderFilled()

```
bool BTreeNode::isUnderFilled ( )
```

This function returns whether a record is too large.

Returns

true if underfilled, false otherwise

Definition at line 247 of file BTreeNode.cpp.

3.4.3.12 merge()

```
int BTreeNode::merge (
            BTreeNode * fromNode )
```

This function merges two nodes together.

Parameters

fromNode	

Returns

-1 if failed, 0 otherwise

Definition at line 148 of file BTreeNode.cpp.

3.4.3.13 print()

```
void BTreeNode::print (
            std::ostream & stream )
```

This function prints the node to the output stream.

Parameters

stream	the stream to print node to

Returns

nothing

Definition at line 112 of file BTreeNode.cpp.

3.4.3.14 read()

```
int BTreeNode::read (
            std::istream & stream,
            int headerRecordSize,
            int RBN )
```

This function reads the node from file.

Parameters

stream	the stream to read from
headerRecordSize	the size of header record
RBN	the block to read

Returns

Returns -1 if there's an error, otherwise node is filled with data

Definition at line 19 of file BTreeNode.cpp.

3.4.3.15 removeKeyAndChildren()

```
int BTreeNode::removeKeyAndChildren (
            int key,
            int child )
```

This function removes a key into the key vector.

Parameters

key	the key to remove
child	the child to remove

Returns

-1 if failed, 0 otherwise

Definition at line 94 of file BTreeNode.cpp.

3.4.3.16 removeRecord()

```
int BTreeNode::removeRecord (
            RecordBuffer & recordBuffer )
```

This function removes a record from the B tree.

Parameters

recordBuffer	the record to remove

Returns

-1 on error, 0 otherwise

Definition at line 52 of file BTreeNode.cpp.

3.4.3.17 retrieveRecord()

```
int BTreeNode::retrieveRecord (
            RecordBuffer & recordBuffer,
            int key )
```

This function retrieves a record from the current buffer.

Parameters

recordBuffer	is the buffer object to store record in.
key	is the zipcode to search for.

Returns

Returns the status code as an integer

Definition at line 59 of file BTreeNode.cpp.

3.4.3.18 setCurRBN()

```
void BTreeNode::setCurRBN (
            int rbn )
```

Sets the current rbn value.

Parameters

rbn	value to set curRBN to

Returns

nothing

Definition at line 203 of file BTreeNode.cpp.

3.4.3.19 setIsLeaf()

```
void BTreeNode::setIsLeaf (
            bool isL )
```

This function sets a node's isLeaf value to true or false.

Parameters

isL,boolean	value to set isLeaf too

Returns

nothing

Definition at line 167 of file BTreeNode.cpp.

3.4.3.20 setNextRBN()

```
void BTreeNode::setNextRBN (
            int rbn )
```

Sets the next rbn value.

Parameters

rbn	value to set nextRBN to

Returns

nothing

Definition at line 237 of file BTreeNode.cpp.

3.4.3.21 setPrevRBN()

```
void BTreeNode::setPrevRBN (
            int rbn )
```

Sets the previous rbn value.

Parameters

rbn	value to set prevRBN to

Returns

nothing

Definition at line 225 of file BTreeNode.cpp.

3.4.3.22 split()

```
int BTreeNode::split (
            BTreeNode * newNode )
```

This function shifts a node and it's children toward the root until the B tree becomes balanced.

Parameters

newNode	the node to place half of data into

Returns

-1 if failed, 0 otherwise

Definition at line 124 of file BTreeNode.cpp.

3.4.3.23 write()

```
int BTreeNode::write (
            std::ostream & stream,
```

```
            int headerRecordSize,
            int RBN )
```

This function writes the node to the file.

Parameters

stream	the stream to write to
headerRecordSize	the size of header record
RBN	the block to write to

Returns

Returns -1 if there's an error, otherwise node is written to stream

Definition at line 33 of file BTreeNode.cpp.

3.4.4 Member Data Documentation

3.4.4.1 blockBuffer

```
BlockBuffer BTreeNode::blockBuffer  [private]
```

Stores the reference to a block buffer object.

Definition at line 181 of file BTreeNode.h.

3.4.4.2 bTreeIndexBuffer

```
BTreeIndexBuffer BTreeNode::bTreeIndexBuffer  [private]
```

Stores the reference to a index buffer object.

Definition at line 182 of file BTreeNode.h.

3.4.4.3 children

```
std::vector<int> BTreeNode::children  [private]
```

Stores the children of node.

Definition at line 189 of file BTreeNode.h.

3.4.4.4 curRBN

```
int BTreeNode::curRBN  [private]
```

Current RBN in file.

Definition at line 183 of file BTreeNode.h.

3.4.4.5 isLeaf

```
bool BTreeNode::isLeaf  [private]
```

stores whether node is a leaf or not

Definition at line 187 of file BTreeNode.h.

3.4.4.6 keys

```
std::vector<int> BTreeNode::keys  [private]
```

Stores the keys of node.

Definition at line 188 of file BTreeNode.h.

3.4.4.7 maxKeys

```
int BTreeNode::maxKeys  [private]
```

max number of keys to hold

Definition at line 184 of file BTreeNode.h.

3.4.4.8 minKeys

```
int BTreeNode::minKeys  [private]
```

min number of keys to hold

Definition at line 185 of file BTreeNode.h.

3.4.4.9 numKeys

```
int BTreeNode::numKeys  [private]
```

current number of keys stored

Definition at line 186 of file BTreeNode.h.

The documentation for this class was generated from the following files:

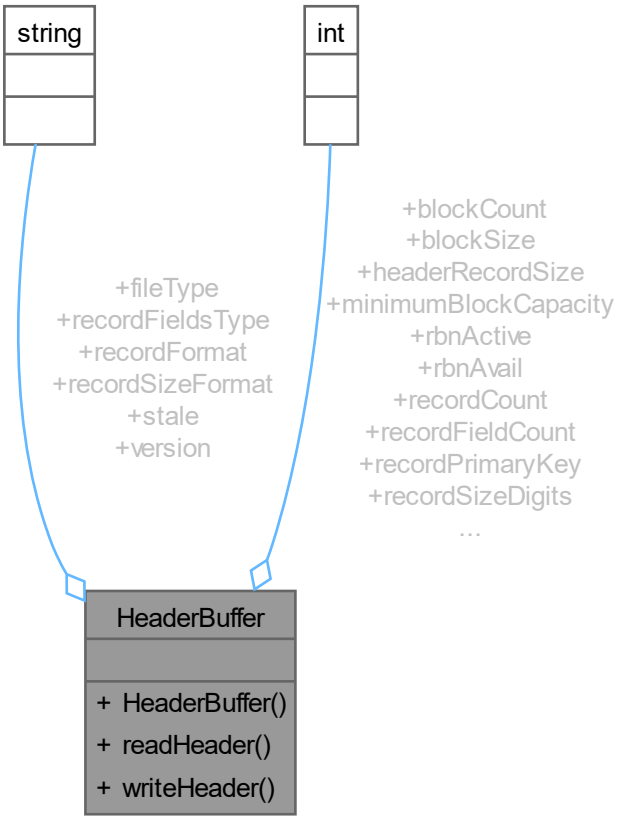
- BTreeNode.h
- BTreeNode.cpp

3.5 HeaderBuffer Class Reference

A class for managing a buffer for a files header information.

`#include <HeaderBuffer.h>`

Collaboration diagram for HeaderBuffer:



Public Member Functions

- HeaderBuffer ()

  Initialize the header buffer with default values.
- int readHeader (std::istream &stream)

  Read the header information from the given input stream.
- int writeHeader (std::ostream &stream) const

  Write the header information to the given output stream.

Public Attributes

- std::string fileType
  The structure of the file.
- std::string version
  The version of the file format.
- int headerRecordSize
  The size of header record.
- int recordSizeDigits
  The number of digits for the record size.
- std::string recordSizeFormat
  The format of the record size.
- int blockSize
  The size of the blocks in bytes.
- int minimumBlockCapacity
  The minimum number of bytes in block.
- int recordCount
  The total count of records.
- int blockCount
  The total count of blocks.
- int recordFieldCount
  The number of fields in each record.
- std::string recordFieldsType
  The data type of record fields.
- std::string recordFormat
  The format of the record fields.
- int recordPrimaryKey
  The primary key of the records.
- int rbnAvail
  Link to beginning of available sequence set.
- int rbnActive
  Link to beginning of active sequence set.
- std::string stale
  Indicates if data is stale.

3.5.1 Detailed Description

A class for managing a buffer for a files header information.

: This class provides methods to read and write header information from/to an input/output stream. Features↩ : Reads and writes header information from/to input/output streams. Assumptions:Assumes the input stream and output stream provided are valid and open.

Definition at line 22 of file HeaderBuffer.h.

3.5.2 Constructor & Destructor Documentation

3.5.2.1 HeaderBuffer()

```
HeaderBuffer::HeaderBuffer ( )
```

Initialize the header buffer with default values.

Definition at line 14 of file HeaderBuffer.cpp.

3.5.3 Member Function Documentation

3.5.3.1 readHeader()

```
int HeaderBuffer::readHeader (
            std::istream & stream )
```

Read the header information from the given input stream.

Parameters

stream	The input stream from which to read the header.

Returns

0 if successful, -1 if an error occurs.

Definition at line 33 of file HeaderBuffer.cpp.

3.5.3.2 writeHeader()

```
int HeaderBuffer::writeHeader (
            std::ostream & stream ) const
```

Write the header information to the given output stream.

Parameters

stream	The output stream to which to write the header.

Returns

0 if successful, -1 if an error occurs.

Definition at line 125 of file HeaderBuffer.cpp.

3.5.4 Member Data Documentation

3.5.4.1 blockCount

```
int HeaderBuffer::blockCount
```

The total count of blocks.

Definition at line 53 of file HeaderBuffer.h.

3.5.4.2 blockSize

```
int HeaderBuffer::blockSize
```

The size of the blocks in bytes.

Definition at line 50 of file HeaderBuffer.h.

3.5.4.3 fileType

```
std::string HeaderBuffer::fileType
```

The structure of the file.

Definition at line 45 of file HeaderBuffer.h.

3.5.4.4 headerRecordSize

```
int HeaderBuffer::headerRecordSize
```

The size of header record.

Definition at line 47 of file HeaderBuffer.h.

3.5.4.5 minimumBlockCapacity

```
int HeaderBuffer::minimumBlockCapacity
```

The minimum number of bytes in block.

Definition at line 51 of file HeaderBuffer.h.

3.5.4.6 rbnActive

```
int HeaderBuffer::rbnActive
```

Link to beginning of active sequence set.

Definition at line 59 of file HeaderBuffer.h.

3.5.4.7 rbnAvail

```
int HeaderBuffer::rbnAvail
```

Link to beginning of available sequence set.

Definition at line 58 of file HeaderBuffer.h.

3.5.4.8 recordCount

```
int HeaderBuffer::recordCount
```

The total count of records.

Definition at line 52 of file HeaderBuffer.h.

3.5.4.9 recordFieldCount

```
int HeaderBuffer::recordFieldCount
```

The number of fields in each record.

Definition at line 54 of file HeaderBuffer.h.

3.5.4.10 recordFieldsType

```
std::string HeaderBuffer::recordFieldsType
```

The data type of record fields.

Definition at line 55 of file HeaderBuffer.h.

3.5.4.11 recordFormat

```
std::string HeaderBuffer::recordFormat
```

The format of the record fields.

Definition at line 56 of file HeaderBuffer.h.

3.5.4.12 recordPrimaryKey

```
int HeaderBuffer::recordPrimaryKey
```

The primary key of the records.

Definition at line 57 of file HeaderBuffer.h.

3.5.4.13 recordSizeDigits

```
int HeaderBuffer::recordSizeDigits
```

The number of digits for the record size.

Definition at line 48 of file HeaderBuffer.h.

3.5.4.14 recordSizeFormat

`std::string HeaderBuffer::recordSizeFormat`

The format of the record size.

Definition at line 49 of file HeaderBuffer.h.

3.5.4.15 stale

`std::string HeaderBuffer::stale`

Indicates if data is stale.

Definition at line 60 of file HeaderBuffer.h.

3.5.4.16 version

`std::string HeaderBuffer::version`

The version of the file format.

Definition at line 46 of file HeaderBuffer.h.

The documentation for this class was generated from the following files:

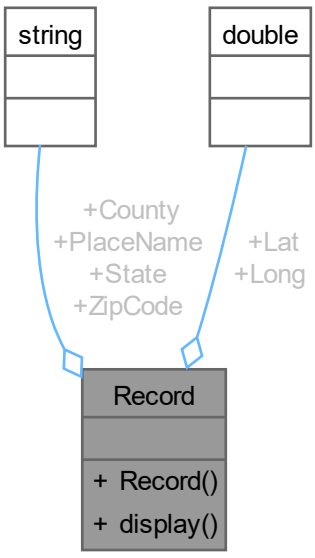
- HeaderBuffer.h
- HeaderBuffer.cpp

3.6 Record Struct Reference

A struct for storing information pertaining to a zip code record.

`#include <Record.h>`

Collaboration diagram for Record:



Public Member Functions

- Record (RecordBuffer &buffer)
  Default constructor.
- void display ()
  Display Function.

Public Attributes

- std::string ZipCode
  ZipCode of record.
- std::string PlaceName
  Place Name of record.
- std::string State
  State of record.
- std::string County
  County of record.
- double Lat
  latitude of record
- double Long
  Longitude of record.

3.6.1 Detailed Description

A struct for storing information pertaining to a zip code record.

Record Struct: Data structure for storing information about a zip code record. Includes: Zipcode, PlaceName, State, County, Latitude, and Longitude. Assumes: This struct constructor assumes that the input will always have six fields

Definition at line 22 of file Record.h.

3.6.2 Constructor & Destructor Documentation

3.6.2.1 Record()

```
Record::Record (
            RecordBuffer & buffer )
```

Default constructor.

Parameters

in	*buffer*	A recordBuffer containing the record information, length indicated and seperated by commas.

Precondition

recordBuffer has six comma seperated fields.

Postcondition

recordBuffer has been parsed and struct fields have all been filled with data.

Definition at line 15 of file Record.cpp.

3.6.3 Member Function Documentation

3.6.3.1 display()

```
void Record::display ( )
```

Display Function.

Precondition

recordBuffer has six comma seperated fields.

Postcondition

The recordBuffer's fields have been outputted to the terminal.

Definition at line 27 of file Record.cpp.

3.6.4 Member Data Documentation

3.6.4.1 County

```
std::string Record::County
```

County of record.

Definition at line 42 of file Record.h.

3.6.4.2 Lat

```
double Record::Lat
```

latitude of record

Definition at line 43 of file Record.h.

3.6.4.3 Long

```
double Record::Long
```

Longitude of record.

Definition at line 44 of file Record.h.

3.6.4.4 PlaceName

```
std::string Record::PlaceName
```

Place Name of record.

Definition at line 40 of file Record.h.

3.6.4.5 State

```
std::string Record::State
```

State of record.

Definition at line 41 of file Record.h.

3.6.4.6 ZipCode

```
std::string Record::ZipCode
```

ZipCode of record.

Definition at line 39 of file Record.h.

The documentation for this struct was generated from the following files:

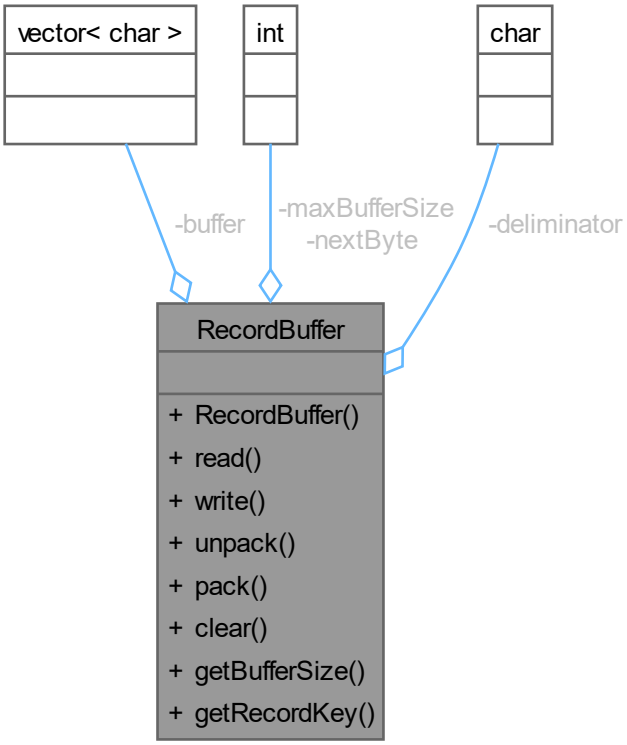
- Record.h
- Record.cpp

3.7 RecordBuffer Class Reference

A class for managing a recordBuffer.

```
#include <RecordBuffer.h>
```

Collaboration diagram for RecordBuffer:



Public Member Functions

- RecordBuffer (int maxSize=1000)

  Constructor for RecordBuffer class.
- int read (std::istream &stream, int recaddr=-1)

  Read an entire record into the buffer.
- int write (std::ostream &stream, int recaddr=-1)

  Write an entire record into the buffer.
- int unpack (std::string &field)

  Read one field from the buffer.
- int pack (std::string field)

  Write one field to the buffer.
- void clear ()

  Clear all buffer data.
- int getBufferSize ()

  Gets the buffer size.
- int getRecordKey ()

  Gets the record key (zipcode).

Private Attributes

- std::vector< char > buffer

  The buffer object.
- int maxBufferSize

  The max buffer size.
- int nextByte

  The next byte location to read.
- char deliminator

  The delimiting character.

3.7.1 Detailed Description

A class for managing a recordBuffer.

RecordBuffer class: Data structure for managing a header buffer. Includes: File type, version, data types, record format, index file and more.

Definition at line 23 of file RecordBuffer.h.

3.7.2 Constructor & Destructor Documentation

3.7.2.1 RecordBuffer()

```
RecordBuffer::RecordBuffer (
            int maxSize = 1000 )
```

Constructor for RecordBuffer class.

Parameters

maxSize	The maximum size of the buffer.

Definition at line 10 of file RecordBuffer.cpp.

3.7.3 Member Function Documentation

3.7.3.1 clear()

```
void RecordBuffer::clear ( )
```

Clear all buffer data.

Returns

nothing.

Definition at line 115 of file RecordBuffer.cpp.

3.7.3.2 getBufferSize()

```
int RecordBuffer::getBufferSize ( )
```

Gets the buffer size.

Returns

buffer size as an integer.

Definition at line 120 of file RecordBuffer.cpp.

3.7.3.3 getRecordKey()

```
int RecordBuffer::getRecordKey ( )
```

Gets the record key (zipcode).

Returns

key as an integer.

Definition at line 124 of file RecordBuffer.cpp.

3.7.3.4 pack()

```
int RecordBuffer::pack (
            std::string field )
```

Write one field to the buffer.

Parameters

field	the field value to write.

Returns

-1 if error or the number of bytes written.

Definition at line 102 of file RecordBuffer.cpp.

3.7.3.5 read()

```
int RecordBuffer::read (
            std::istream & stream,
            int recaddr = -1 )
```

Read an entire record into the buffer.

Parameters

stream	the input file stream.

Returns

-1 if error or the current byte address.

Definition at line 16 of file RecordBuffer.cpp.

3.7.3.6 unpack()

```
int RecordBuffer::unpack (
            std::string & field )
```

Read one field from the buffer.

Parameters

field	the field value to read.

Returns

-1 if error or the current byte address.

Definition at line 86 of file RecordBuffer.cpp.

3.7.3.7 write()

```
int RecordBuffer::write (
            std::ostream & stream,
            int recaddr = -1 )
```

Write an entire record into the buffer.

Parameters

stream	the outout file stream.

Returns

-1 if error or the number of bytes written.

Definition at line 58 of file RecordBuffer.cpp.

3.7.4 Member Data Documentation

3.7.4.1 buffer

```
std::vector<char> RecordBuffer::buffer  [private]
```

The buffer object.

Definition at line 78 of file RecordBuffer.h.

3.7.4.2 deliminator

```
char RecordBuffer::deliminator  [private]
```

The delimiting character.

Definition at line 81 of file RecordBuffer.h.

3.7.4.3 maxBufferSize

```
int RecordBuffer::maxBufferSize  [private]
```

The max buffer size.

Definition at line 79 of file RecordBuffer.h.

3.7.4.4 nextByte

```
int RecordBuffer::nextByte  [private]
```

The next byte location to read.

Definition at line 80 of file RecordBuffer.h.

The documentation for this class was generated from the following files:

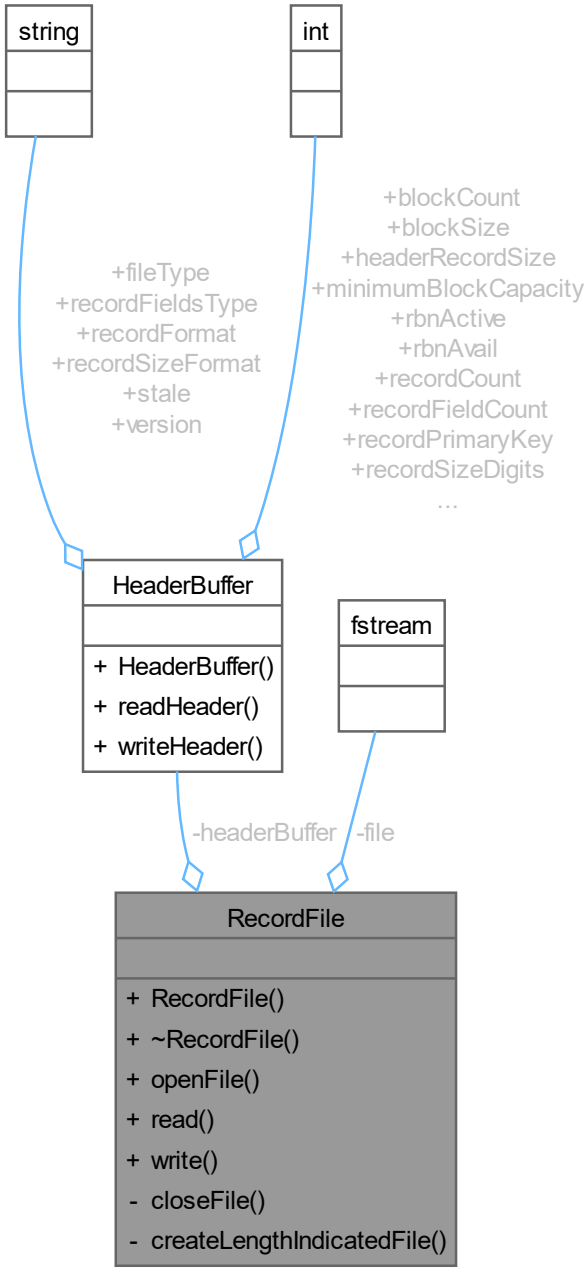
- RecordBuffer.h
- RecordBuffer.cpp

3.8 RecordFile Class Reference

A class for managing the file for a recordBuffer.

```
#include <RecordFile.h>
```

Collaboration diagram for RecordFile:



Public Member Functions

- RecordFile (HeaderBuffer &hbuf)

  Constructor for RecordFile class.
- ~RecordFile ()

  This is the destructor for the record file object.
- bool openFile (const std::string &dataFile, std::string &recordFile)

Opens the file with the given dataFile.

- int read (RecordBuffer &recordBuffer, int recaddr=-1)

  Reads a record from the file at the specified address.

- int write (RecordBuffer &recordBuffer, int recaddr=-1)

  Writes a record to the file at the specified address.

Private Member Functions

- bool closeFile ()

  Closes the currently open file.

- int createLengthIndicatedFile (const std::string &inputCsvFile)

  Creates a length-indicated file from the given CSV input file.

Private Attributes

- HeaderBuffer & headerBuffer

  The HeaderBuffer containing file header information.

- std::fstream file

  The file stream for the record file.

3.8.1 Detailed Description

A class for managing the file for a recordBuffer.

: This class provides methods for working with a file associated with a RecordBuffer. Includes: Methods for opening, closing, reading, and writing to the file. Assumes:The provided RecordBuffer and HeaderBuffer objects are correctly initialized and valid.

Definition at line 23 of file RecordFile.h.

3.8.2 Constructor & Destructor Documentation

3.8.2.1 RecordFile()

```
RecordFile::RecordFile (
            HeaderBuffer & hbuf )
```

Constructor for RecordFile class.

Parameters

rbuf	The RecordBuffer associated with the file.
hbuf	The HeaderBuffer containing file header information.

Definition at line 14 of file RecordFile.cpp.

3.8.2.2 ~RecordFile()

```
RecordFile::~RecordFile ( )
```

This is the destructor for the record file object.

Postcondition

Object is properly closed.

Definition at line 18 of file RecordFile.cpp.

3.8.3 Member Function Documentation

3.8.3.1 closeFile()

```
bool RecordFile::closeFile ( )  [private]
```

Closes the currently open file.

Returns

True if the file is closed successfully, false otherwise.

Definition at line 58 of file RecordFile.cpp.

3.8.3.2 createLengthIndicatedFile()

```
int RecordFile::createLengthIndicatedFile (
            const std::string & inputCsvFile )  [private]
```

Creates a length-indicated file from the given CSV input file.

Parameters

inputCsvFile	The name of the input CSV file.

Returns

number of records, or -1 on error.

Definition at line 76 of file RecordFile.cpp.

3.8.3.3 openFile()

```
bool RecordFile::openFile (
            const std::string & dataFile,
            std::string & recordFile )
```

Opens the file with the given dataFile.

Parameters

dataFile	The name of the file to open.

Returns

True if the file is opened successfully, false otherwise.

Definition at line 22 of file RecordFile.cpp.

3.8.3.4 read()

```
int RecordFile::read (
            RecordBuffer & recordBuffer,
            int recaddr = -1 )
```

Reads a record from the file at the specified address.

Parameters

recordBuffer	The address of the record to read (default: -1).
recaddr	The record address in the file (default: -1).

Returns

0 if successful, -1 if an error occurs.

Definition at line 68 of file RecordFile.cpp.

3.8.3.5 write()

```
int RecordFile::write (
            RecordBuffer & recordBuffer,
            int recaddr = -1 )
```

Writes a record to the file at the specified address.

Parameters

recaddr	The address of the record to write (default: -1).
recaddr	The record address in the file (default: -1).

Returns

0 if successful, -1 if an error occurs.

Definition at line 72 of file RecordFile.cpp.

3.8.4 Member Data Documentation

3.8.4.1 file

```
std::fstream RecordFile::file  [private]
```

The file stream for the record file.

Definition at line 64 of file RecordFile.h.

3.8.4.2 headerBuffer

```
HeaderBuffer& RecordFile::headerBuffer  [private]
```

The HeaderBuffer containing file header information.

Definition at line 63 of file RecordFile.h.

The documentation for this class was generated from the following files:

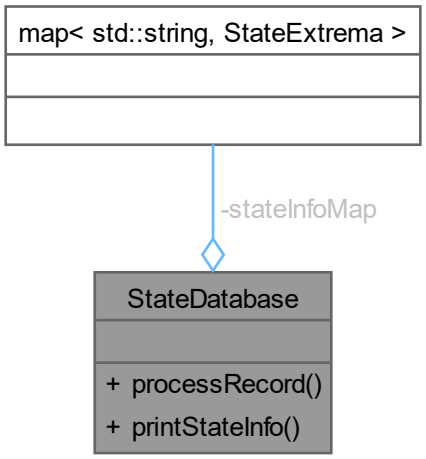
- RecordFile.h
- RecordFile.cpp

3.9 StateDatabase Class Reference

A class to store a collection of StateExtrema objects.

```
#include <StateDatabase.h>
```

Collaboration diagram for StateDatabase:



Public Member Functions

- void processRecord (const Record &record)

  Processes a record and updates state information.
- void printStateInfo (std::string state) const

  Prints the state information.

Private Attributes

- std::map< std::string, StateExtrema > stateInfoMap

  Map that links state ID to StateExtrema object.

3.9.1 Detailed Description

A class to store a collection of StateExtrema objects.

StateDatabase class: A database class to store a collection of state extrema objects utilizing a map which links the State IF (Ex: "MN") to the associated StateExtrema object. Features: Provides methods to process a record and display all currently stored StateExtrema objects.

Definition at line 25 of file StateDatabase.h.

3.9.2 Member Function Documentation

3.9.2.1 printStateInfo()

```
void StateDatabase::printStateInfo (
            std::string state ) const
```

Prints the state information.

Precondition

None.

Postcondition

All objects in stateInfoMap are displayed by field.

Definition at line 54 of file StateDatabase.cpp.

3.9.2.2 processRecord()

```
void StateDatabase::processRecord (
            const Record & record )
```

Processes a record and updates state information.

Parameters

in	*record*	- The Record object to process.

Precondition

record is a fully initialized object with all fields filled.

Postcondition

StateExtrema object linked by state ID in record object is updated, or new state is added to map, and StateExtrema object initialized with record data.

Definition at line 15 of file StateDatabase.cpp.

3.9.3 Member Data Documentation

3.9.3.1 stateInfoMap

```
std::map<std::string, StateExtrema> StateDatabase::stateInfoMap  [private]
```

Map that links state ID to StateExtrema object.

Definition at line 47 of file StateDatabase.h.

The documentation for this class was generated from the following files:

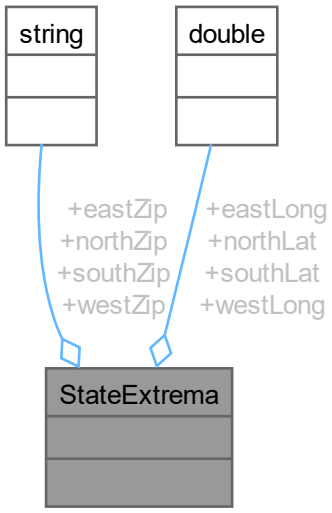
- StateDatabase.h
- StateDatabase.cpp

3.10 StateExtrema Struct Reference

A struct for storing information about the extrema of a state.

```
#include <StateExtrema.h>
```

Collaboration diagram for StateExtrema:



Public Attributes

- std::string eastZip

  Easternmost zip code of state.
- double eastLong

  Longitude of easternmost zipcode.
- std::string westZip

  Westernmost zip code of state.
- double westLong

  Longitude of westernmost zipcode.
- std::string northZip

  Northernmost zip code of state.
- double northLat

  Latitude of northernmost zipcode.
- std::string southZip

  Southernmost zip code of state.
- double southLat

  Latitude of southernmost zipcode.

3.10.1 Detailed Description

A struct for storing information about the extrema of a state.

StateExtrema Struct: Data structure for storing the zipcode and location (latitude or longitude) of the Northernmost, Easternmost, Southernmost, and Westernmost locations in a state.

Definition at line 19 of file StateExtrema.h.

3.10.2 Member Data Documentation

3.10.2.1 eastLong

```
double StateExtrema::eastLong
```

Longitude of easternmost zipcode.

Definition at line 21 of file StateExtrema.h.

3.10.2.2 eastZip

```
std::string StateExtrema::eastZip
```

Easternmost zip code of state.

Definition at line 20 of file StateExtrema.h.

3.10.2.3 northLat

```
double StateExtrema::northLat
```

Latitude of northernmost zipcode.

Definition at line 27 of file StateExtrema.h.

3.10.2.4 northZip

```
std::string StateExtrema::northZip
```

Northernmost zip code of state.

Definition at line 26 of file StateExtrema.h.

3.10.2.5 southLat

```
double StateExtrema::southLat
```

Latitude of southernmost zipcode.

Definition at line 30 of file StateExtrema.h.

3.10.2.6 southZip

```
std::string StateExtrema::southZip
```

Southernmost zip code of state.

Definition at line 29 of file StateExtrema.h.

3.10.2.7 westLong

```
double StateExtrema::westLong
```

Longitude of westernmost zipcode.

Definition at line 24 of file StateExtrema.h.

3.10.2.8 westZip

```
std::string StateExtrema::westZip
```

Westernmost zip code of state.

Definition at line 23 of file StateExtrema.h.

The documentation for this struct was generated from the following file:

- StateExtrema.h

Chapter 4

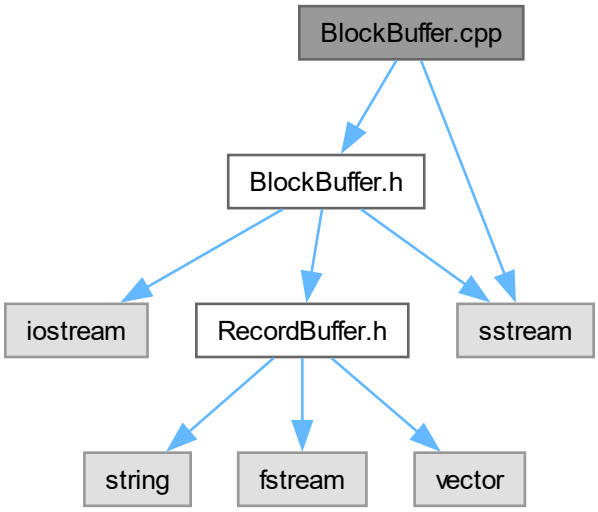
File Documentation

4.1 BlockBuffer.cpp File Reference

Implementation file for the BlockBuffer class.

```
#include "BlockBuffer.h"
#include <sstream>
```

Include dependency graph for BlockBuffer.cpp:



4.1.1 Detailed Description

Implementation file for the BlockBuffer class.

Author

Team 5

Date

November 18, 2023

Version

2.0

Definition in file BlockBuffer.cpp.

4.2 BlockBuffer.cpp

Go to the documentation of this file.

```
00001
00009 #include "BlockBuffer.h"
00010 #include <sstream>
00011 using namespace std;
00012
00013 BlockBuffer::BlockBuffer(int blockSz, int minCap) {
00014     blockSize = blockSz;
00015     minimumBlockCapacity = minCap;
00016     prevRBN = 0;
00017     nextRBN = 0;
00018     curRBN = 0;
00019     numRecords = 0;
00020 }
00021
00022 int BlockBuffer::read(std::istream &stream, int headerRecordSize, int blockNumber) {
00023     // Move to location if needed
00024     if (blockNumber != -1) {
00025         stream.clear();
00026         stream.seekg((blockNumber-1) * blockSize + headerRecordSize);
00027     }
00028
00029     // check input stream
00030     if (!stream) return -1;
00031
00032     // Get current location in stream
00033     int addr = stream.tellg();
00034     curRBN = (addr + blockSize - headerRecordSize) / blockSize;
00035
00036     // Clear current buffer
00037     clear();
00038
00039     // Read block into buffer
00040     vector<char> buf(blockSize);
00041     stream.read(buf.data(), blockSize);
00042     // Easy way to check if block is completely empty
00043     if (buf[0] == '\0') return -1;
00044     // Remove end spaces as they are rewritten in write function
00045     auto pos = std::find_if_not(buf.rbegin(), buf.rend(), [](char c) { return std::isspace(c) || c ==
      '\n'; }).base();
00046     // copy data into internal buffer
00047     buffer.write(buf.data(), std::distance(buf.begin(), pos)+1);
00048
00049     // Get metadata
00050     string metadata;
00051     getline(buffer, metadata, ',');
00052     numRecords = stoi(metadata);
00053     std::getline(buffer, metadata, ',');
00054     prevRBN = stoi(metadata);
00055     std::getline(buffer, metadata);
00056     nextRBN = std::stoi(metadata);
00057     buffer.seekg(0);
00058
00059     // check stream
00060     if (stream.bad()) {
00061         stream.clear();
00062         return -1;
```

```
00063     }
00064
00065     return addr;
00066 }
00067
00068 int BlockBuffer::write(std::ostream &stream, int headerRecordSize, int blockNumber) {
00069
00070     // Move to location if needed
00071     if (blockNumber != -1) {
00072         stream.seekp((blockNumber-1) * blockSize + headerRecordSize);
00073     }
00074
00075     // Get output location
00076     int addr = stream.tellp();
00077     curRBN = (addr + blockSize - headerRecordSize) / blockSize;
00078
00079     // Write the buffer - always rewrite metadata
00080     string str = buffer.str();
00081     int pos = str.find_first_of('\n');
00082     if (pos > 0) str = str.substr(pos+1);
00083     str = to_string(numRecords) + "," + to_string(prevRBN) + "," + to_string(nextRBN) + "\n" + str;
00084
00085
00086     int sz = str.length();
00087     int remainingSpace = blockSize - sz;
00088     if (remainingSpace > 0) {
00089         str += string(remainingSpace-1, ' '); str += '\n';
00090     }
00091     stream.write(str.c_str(), blockSize);
00092
00093     // check stream
00094     if (!stream) return -1;
00095
00096     return addr;
00097 }
00098
00099 int BlockBuffer::unpack(RecordBuffer &rBuf) {
00100     numRecords--;
00101     if (numRecords < 0) {
00102         numRecords = 0;
00103         return -1;
00104     }
00105
00106     // Ignore block metadata
00107     string buf = buffer.str();
00108     int pos = buf.find_first_of('\n');
00109     if (pos > 0) buf = buf.substr(pos+1);
00110     buffer.seekg(pos+1);
00111
00112     // Read the record into record buffer, than remove from block buffer
00113     int recordAddr = rBuf.read(buffer);
00114     int curAddr = buffer.tellg();
00115     buf.erase(0, curAddr - recordAddr);
00116     clear();
00117
00118     // Rewrite new block buffer
00119     buffer « numRecords « "," « prevRBN « "," « nextRBN « endl;
00120     buffer « buf;
00121
00122     return recordAddr;
00123 }
00124
00125 int BlockBuffer::pack(RecordBuffer &rBuf) {
00126     numRecords++;
00127     string buf = buffer.str();
00128     // Get the buffer without the current block metadata as it will be rewritten
00129     int pos = buf.find_first_of('\n');
00130     if (pos > 0) buf = buf.substr(pos+1);
00131     clear();
00132     buffer « numRecords « "," « prevRBN « "," « nextRBN « endl;
00133     buffer « buf;
00134
00135     return rBuf.write(buffer);
00136 }
00137
00138 void BlockBuffer::clear() {
00139     buffer.clear();
00140     buffer.str("");
00141 }
00142
00143 int BlockBuffer::getNextRBN() {
00144     return nextRBN;
00145 }
00146
00147 int BlockBuffer::getPrevRBN() {
00148     return prevRBN;
00149 }
```

```
00150
00151 int BlockBuffer::getCurRBN() {
00152     return curRBN;
00153 }
00154
00155 int BlockBuffer::getNumRecords() {
00156     return numRecords;
00157 }
00158
00159 void BlockBuffer::setNextRBN(int rbn) {
00160     nextRBN = rbn;
00161 }
00162
00163 void BlockBuffer::setPrevRBN(int rbn) {
00164     prevRBN = rbn;
00165 }
00166
00167 void BlockBuffer::setCurRBN(int rbn) {
00168     curRBN = rbn;
00169 }
00170
00171 void BlockBuffer::setNumRecords(int num) {
00172     numRecords = num;
00173 }
00174
00175 bool BlockBuffer::isOverFilled() {
00176     return buffer.str().length() > blockSize;
00177 }
00178
00179 bool BlockBuffer::isUnderFilled() {
00180     return buffer.str().length() < minimumBlockCapacity;
00181 }
00182
00183 void BlockBuffer::splitBuffer(BlockBuffer &newBlockBuffer) {
00184     RecordBuffer rBuf;
00185     BlockBuffer tmpBuffer1;
00186     BlockBuffer tmpBuffer2;
00187
00188     // Assuming splitting the records evenly between the current and new block
00189     int recordsToMove = numRecords / 2.0;
00190
00191     // Move the records to the new block buffer
00192     int num = numRecords;
00193     for (int i = 0; i < num; i++) {
00194         unpack(rBuf);
00195         if (i < recordsToMove) tmpBuffer1.pack(rBuf);
00196         else tmpBuffer2.pack(rBuf);
00197     }
00198
00199     clear();
00200
00201     while(tmpBuffer1.unpack(rBuf) != -1) {
00202         pack(rBuf);
00203     }
00204
00205     while(tmpBuffer2.unpack(rBuf) != -1) {
00206         newBlockBuffer.pack(rBuf);
00207     }
00208
00209 }
00210
00211 void BlockBuffer::mergeBuffer(BlockBuffer &newBlockBuffer) {
00212     // Move the records to the new block buffer
00213     int count = newBlockBuffer.getNumRecords();
00214     for (int i = 0; i < count; ++i) {
00215         RecordBuffer rBuf;
00216         newBlockBuffer.unpack(rBuf);
00217         pack(rBuf);
00218     }
00219 }
00220
00221 void BlockBuffer::redistributeBuffer(BlockBuffer &newBlockBuffer) {
00222     while (isUnderFilled()) {
00223         RecordBuffer rBuf;
00224         newBlockBuffer.unpack(rBuf);
00225         pack(rBuf);
00226     }
00227 }
00228
00229 int BlockBuffer::getLargestKey() {
00230     // Create a copy of the current buffer
00231     std::stringstream bufferCopy(this->buffer.str());
00232
00233     RecordBuffer rbuf;
00234     int largestKey = -1;
00235     BlockBuffer tmpBuffer;
00236     tmpBuffer.read(bufferCopy, 0);
```

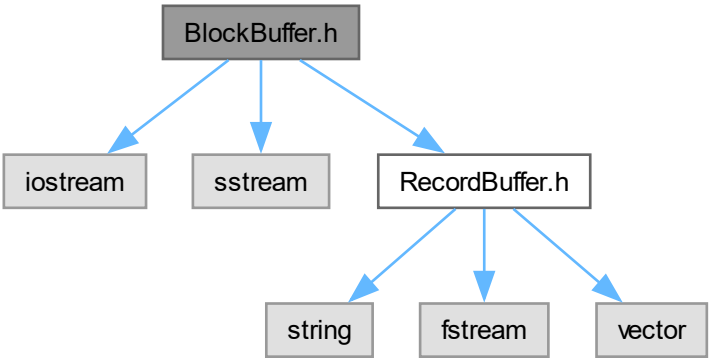
```
00237
00238     while (tmpBuffer.unpack(rbuf) != -1) {
00239         int key = rbuf.getRecordKey();
00240         if (key > largestKey) {
00241             largestKey = key;
00242         }
00243     }
00244
00245     return largestKey;
00246 }
00247
00248 int BlockBuffer::sortBuffer() {
00249     vector<RecordBuffer> recordBuffers;
00250     RecordBuffer recordBuffer;
00251
00252     while(unpack(recordBuffer) != -1) {
00253         recordBuffers.push_back(recordBuffer);
00254     }
00255
00256     std::sort(recordBuffers.begin(), recordBuffers.end(),
00257               [](RecordBuffer& a, RecordBuffer& b) {
00258                   return a.getRecordKey() < b.getRecordKey();
00259               });
00260
00261     clear();
00262
00263     for (auto rBuf : recordBuffers) {
00264         pack(rBuf);
00265     }
00266
00267     return 0;
00268 }
00269
00270 int BlockBuffer::removeRecord(int key) {
00271     vector<RecordBuffer> recordBuffers;
00272     RecordBuffer recordBuffer;
00273
00274     while(unpack(recordBuffer) != -1) {
00275         if (recordBuffer.getRecordKey() != key) {
00276             recordBuffers.push_back(recordBuffer);
00277         }
00278     }
00279
00280     clear();
00281
00282     for (auto rBuf : recordBuffers) {
00283         pack(rBuf);
00284     }
00285 }
```

4.3 BlockBuffer.h File Reference

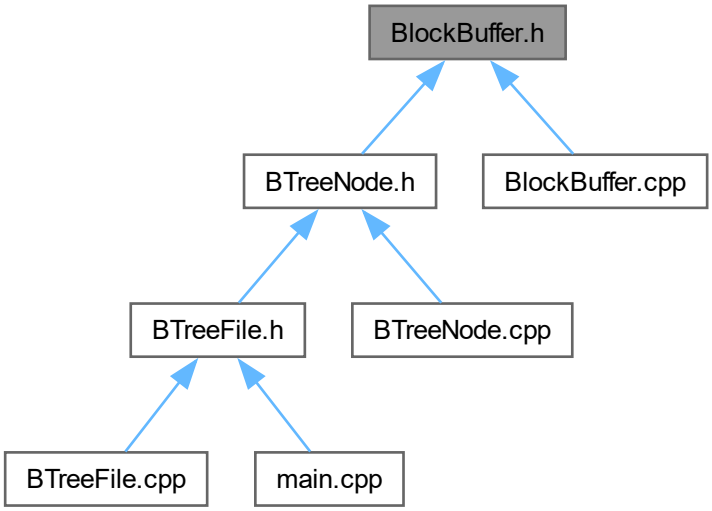
Header file for the BlockBuffer class.

```
#include <iostream>
#include <sstream>
#include "RecordBuffer.h"
```

Include dependency graph for BlockBuffer.h:



This graph shows which files directly or indirectly include this file:



Classes

- class BlockBuffer

  A class that has functions to read, write, pack, and unpack block files.

4.3.1 Detailed Description

Header file for the BlockBuffer class.

Author

Team 5

Date

November 18, 2023

Version

1.0

Definition in file BlockBuffer.h.

4.4 BlockBuffer.h

Go to the documentation of this file.

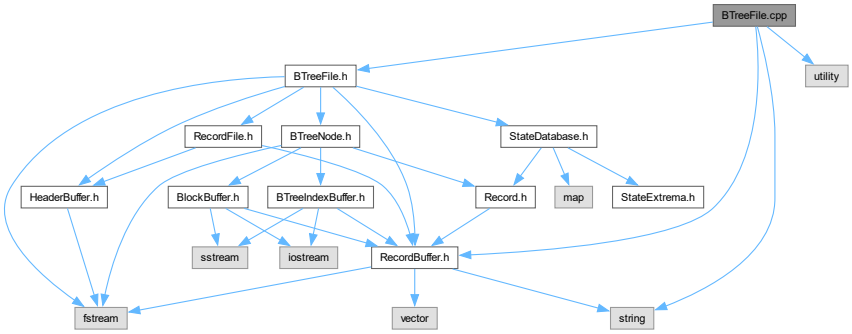
```
00001
00017 #ifndef CSCI331_PROJECT2_P2_BLOCKBUFFER_H
00018 #define CSCI331_PROJECT2_P2_BLOCKBUFFER_H
00019
00020 #include <iostream>
00021 #include <sstream>
00022 #include "RecordBuffer.h"
00023
00024 class BlockBuffer {
00025 public:
00032     BlockBuffer(int blockSz = 512, int minCap = 256);
00033
00041     int read(std::istream & stream, int headerRecordSize, int blockNumber = -1);
00042
00050     int write(std::ostream & stream, int headerRecordSize, int blockNumber = -1);
00051
00057     int unpack(RecordBuffer &rBuf);
00058
00064     int pack(RecordBuffer &rBuf);
00065
00070     void clear();
00071
00076     int getNextRBN();
00077
00082     int getPrevRBN();
00083
00088     int getCurRBN();
00089
00094     int getNumRecords();
00095
00100     void setNextRBN(int rbn);
00101
00106     void setPrevRBN(int rbn);
00107
00112     void setCurRBN(int rbn);
00113
00118     void setNumRecords(int num);
00119
00124     bool isOverFilled();
00125
00130     bool isUnderFilled();
00131
00137     void splitBuffer(BlockBuffer &newBlockBuffer);
00138
00144     void mergeBuffer(BlockBuffer &newBlockBuffer);
00145
00151     void redistributeBuffer(BlockBuffer &newBlockBuffer);
00152
```

```
00157     int getLargestKey();
00158
00163     int sortBuffer();
00164
00170     int removeRecord(int key);
00171
00172 private:
00173     std::stringstream buffer;
00174     int blockSize;
00175     int minimumBlockCapacity;
00176     int numRecords;
00177     int prevRBN;
00178     int nextRBN;
00179     int curRBN;
00180 };
00181
00182
00183 #endif //CSCI331_PROJECT2_P2_BLOCKBUFFER_H
```

4.5 BTreeFile.cpp File Reference

```
#include "BTreeFile.h"
#include "RecordBuffer.h"
#include <string>
#include <utility>
```

Include dependency graph for BTreeFile.cpp:



4.6 BTreeFile.cpp

Go to the documentation of this file.

```
00001
00006 #include "BTreeFile.h"
00007 #include "RecordBuffer.h"
00008 #include <string>
00009 #include <utility>
00010 using namespace std;
00011
00012 BTreeFile::BTreeFile(HeaderBuffer &hbuf, int order) : headerBuffer(hbuf), root(nullptr) {
00013     this->order = order;
00014     this->height = 1;
00015     this->root = new BTreeNode(order);
00016 }
00017
00018 BTreeFile::~BTreeFile() {
00019     closeFile();
00020 }
00021
00022 bool BTreeFile::openFile(std::string &bTreeFileName) {
00023     // Attempt to open in read/write
00024     filename = bTreeFileName;
00025     file.open(filename.c_str(), std::ios::in | std::ios::out | std::ios::binary);
```

```
00026
00027     // If that fails, most likely means file does not exist, so create it and write header.
00028     if (!file.is_open()) {
00029         file.open(filename.c_str(), std::ios::out | std::ios::binary);
00030         headerBuffer.fileType = "blocked sequence set with index";
00031         headerBuffer.writeHeader(file);
00032         flushData();
00033     } else {
00034         headerBuffer.readHeader(file);
00035     }
00036
00037     // Move past the header
00038     file.seekg(headerBuffer.headerRecordSize);
00039     file.seekp(headerBuffer.headerRecordSize);
00040
00041     // Check stream status
00042     if (!file) {
00043         return false;
00044     }
00045
00046     // Read root into memory
00047     if (root->read(file, headerBuffer.headerRecordSize, 1) == -1) {
00048         flushData();
00049         // If read fails assume empty file, so init with root
00050         if (root->write(file, headerBuffer.headerRecordSize, 1) == -1)
00051         {
00052             return false;
00053         }
00054     }
00055
00056     return true;
00057 }
00058
00059 bool BTreeFile::closeFile() {
00060     if (file.is_open()) {
00061         file.seekg(0, std::ios::end);
00062         headerBuffer.blockCount = file.tellg() / headerBuffer.blockSize;
00063         headerBuffer.stale = "false";
00064         headerBuffer.writeHeader(file);
00065         file.close();
00066         return true;
00067     }
00068     return false;
00069 }
00070
00071 int BTreeFile::insert(RecordBuffer& recordBuffer) {
00072     int key = recordBuffer.getRecordKey();
00073     BTreeNode* leaf = findLeafNode(key);
00074
00075     // If the leaf is too full
00076     if (leaf->insertRecord(recordBuffer) == -1) {
00077         BTreeNode* newLeaf = new BTreeNode(order);
00078         leaf->split(newLeaf);
00079         int largestKey = leaf->getLargestKey();
00080
00081         if (leaf == root) {
00082             handleRootSplit(largestKey, leaf, newLeaf);
00083         } else {
00084             handleNonRootSplit(largestKey, leaf, newLeaf);
00085         }
00086     } else {
00087         leaf->write(file, headerBuffer.headerRecordSize, leaf->getCurRBN());
00088     }
00089
00090     return flushData();
00091 }
00092
00093 int BTreeFile::remove(RecordBuffer& recordBuffer) {
00094     int key = recordBuffer.getRecordKey();
00095     BTreeNode* leaf = findLeafNode(key);
00096     BTreeNode* parent = findParentNode(leaf);
00097
00098     int largestKey = leaf->getLargestKey();
00099     int status = leaf->removeRecord(recordBuffer);
00100
00101     // Update parent if largest key changed
00102     int newLargestKey = leaf->getLargestKey();
00103     if (newLargestKey != largestKey) {
00104         parent->removeKeyAndChildren(largestKey, leaf->getCurRBN());
00105         parent->insertKeyAndChildren(newLargestKey, leaf->getCurRBN());
00106     }
00107
00108     // If the leaf is under limit
00109     if (status == -1) {
00110         handleMerge(parent, leaf);
00111     } else {
00112         leaf->write(file, headerBuffer.headerRecordSize, leaf->getCurRBN());
```

```
00113     }
00114
00115     return flushData();
00116 }
00117
00118 int BTreeFile::search(RecordBuffer& recordBuffer, int key) {
00119     BTreeNode * node = findLeafNode(key);
00120
00121     if (node == nullptr) {
00122         return -1;
00123     } else {
00124         node->retrieveRecord(recordBuffer, key);
00125         if (recordBuffer.getBufferSize() <= headerBuffer.recordSizeDigits) {
00126             return -1;
00127         }
00128     }
00129
00130     return 0;
00131 }
00132
00133 void BTreeFile::displaySequenceSet(std::ostream &ostream) {
00134     BlockBuffer blockBuffer;
00135     RecordBuffer recordBuffer;
00136     stringstream ss;
00137
00138     // Get leftmost node
00139     BTreeNode * node = findLeafNode(0);
00140     int RBN = node->getCurRBN();
00141
00142     while(RBN != 0) {
00143         blockBuffer.read(file, headerBuffer.headerRecordSize, RBN);
00144
00145         int currentBlock = blockBuffer.getCurRBN();
00146         cout << "RELATIVE BLOCK NUMBER: " << currentBlock << endl;
00147
00148         blockBuffer.write(ss, 0);
00149         cout << ss.str();
00150
00151         ss.clear();
00152         ss.str("");
00153         RBN = blockBuffer.getNextRBN();
00154     }
00155 }
00156
00157 void BTreeFile::displayExtrema(ostream &ostream, std::string state) {
00158     BlockBuffer blockBuffer;
00159     RecordBuffer recordBuffer;
00160     StateDatabase stateDb;
00161
00162     // Get leftmost node
00163     BTreeNode * node = findLeafNode(0);
00164     int RBN = node->getCurRBN();
00165
00166     while(RBN != 0) {
00167         blockBuffer.read(file, headerBuffer.headerRecordSize, RBN);
00168
00169         while(blockBuffer.unpack(recordBuffer) != -1) {
00170             Record record(recordBuffer);
00171             stateDb.processRecord(record);
00172         }
00173
00174         RBN = blockBuffer.getNextRBN();
00175     }
00176
00177     stateDb.printStateInfo(std::move(state));
00178 }
00179
00180 void BTreeFile::displayTree(ostream &ostream) {
00181     displayNode(root, ostream, 0, "");
00182 }
00183
00184
00185 void BTreeFile::handleRootSplit(int largestKey, BTreeNode* leaf, BTreeNode* newLeaf) {
00186     // If the split node was the root, we need to create a new root
00187     BTreeNode* newRoot = new BTreeNode(order);
00188     newRoot->insertKeyAndChildren(largestKey, headerBuffer.rbnAvail, headerBuffer.rbnAvail+1);
00189
00190     leaf->setNextRBN(headerBuffer.rbnAvail + 1);
00191     leaf->write(file, headerBuffer.headerRecordSize, headerBuffer.rbnAvail);
00192     headerBuffer.rbnAvail++;
00193
00194     newLeaf->setPrevRBN(leaf->getCurRBN());
00195     newLeaf->write(file, headerBuffer.headerRecordSize, headerBuffer.rbnAvail);
00196     headerBuffer.rbnAvail++;
00197
00198     root = newRoot;
00199     root->setIsLeaf(false);
```

```
00200     root->write(file, headerBuffer.headerRecordSize, 1);
00201     height++;
00202 }
00203
00204 void BTreeFile::handleNonRootSplit(int largestKey, BTreeNode* leaf, BTreeNode* newLeaf) {
00205     BTreeNode* parent = findParentNode(leaf);
00206     if (parent == nullptr) {
00207         handleRootSplit(largestKey, leaf, newLeaf);
00208         return;
00209     }
00210
00211     // Write both child nodes and insert key pair into parent
00212     parent->insertKeyAndChildren(largestKey, headerBuffer.rbnAvail);
00213
00214     // Write the split leaf
00215     newLeaf->setNextRBN(leaf->getNextRBN());
00216     leaf->setNextRBN(headerBuffer.rbnAvail);
00217     leaf->write(file, headerBuffer.headerRecordSize, leaf->getCurRBN());
00218
00219     // Write the new leaf
00220     newLeaf->setPrevRBN(leaf->getCurRBN());
00221     newLeaf->write(file, headerBuffer.headerRecordSize, headerBuffer.rbnAvail);
00222     headerBuffer.rbnAvail++;
00223
00224     // Set the prev RBN of the next node of new node
00225     if (newLeaf->getNextRBN() != 0) {
00226         BTreeNode * nextOfNewLeaf = new BTreeNode(order);
00227         nextOfNewLeaf->read(file, headerBuffer.headerRecordSize, newLeaf->getNextRBN());
00228         nextOfNewLeaf->setPrevRBN(newLeaf->getCurRBN());
00229         nextOfNewLeaf->write(file, headerBuffer.headerRecordSize, nextOfNewLeaf->getCurRBN());
00230     }
00231
00232     // Check if the parent is overfull and handle splitting recursively
00233     if (parent->isOverFilled()) {
00234         BTreeNode* newParentNode = new BTreeNode(order);
00235         newParentNode->setIsLeaf(false);
00236         int parentSplitKey = parent->split(newParentNode);
00237
00238         handleNonRootSplit(parentSplitKey, parent, newParentNode);
00239     } else {
00240         parent->write(file, headerBuffer.headerRecordSize, parent->getCurRBN());
00241     }
00242 }
00243
00244 void BTreeFile::handleMerge(BTreeNode* parent, BTreeNode *leaf) {
00245     if (parent == nullptr) {
00246         return;
00247     }
00248
00249     int prevRBN = leaf->getPrevRBN();
00250     int nextRBN = leaf->getNextRBN();
00251     int curRBN = leaf->getCurRBN();
00252     BTreeNode * prevLeaf = new BTreeNode(order);
00253     BTreeNode * nextLeaf = new BTreeNode(order);
00254
00255     if (prevRBN != 0) {
00256         prevLeaf->read(file, headerBuffer.headerRecordSize, prevRBN);
00257     }
00258
00259     if (nextRBN != 0) {
00260         nextLeaf->read(file, headerBuffer.headerRecordSize, nextRBN);
00261     }
00262
00263     if (prevRBN != 0 && prevLeaf->isUnderFilled()) {
00264         int largestKey = prevLeaf->getLargestKey();
00265         parent->removeKeyAndChildren(largestKey, leaf->getCurRBN());
00266         prevLeaf->merge(leaf);
00267
00268         // Update metadata
00269         nextLeaf->setNextRBN(leaf->getNextRBN());
00270         leaf->setPrevRBN(0);
00271         leaf->setNextRBN(headerBuffer.rbnAvail);
00272         headerBuffer.rbnAvail = leaf->getCurRBN();
00273
00274         // Write old block (now empty)
00275         leaf->write(file, headerBuffer.headerRecordSize, curRBN);
00276         // Write merged block
00277         prevLeaf->write(file, headerBuffer.headerRecordSize, prevRBN);
00278         // Write parent
00279         parent->write(file, headerBuffer.headerRecordSize, parent->getCurRBN());
00280
00281     } else if (nextRBN != 0 && nextLeaf->isUnderFilled()) {
00282         int largestKey = leaf->getLargestKey();
00283         parent->removeKeyAndChildren(largestKey, nextLeaf->getCurRBN());
00284         leaf->merge(nextLeaf);
00285
00286         // Update metadata
```

```
00287         leaf->setNextRBN(nextLeaf->getNextRBN());
00288         nextLeaf->setPrevRBN(0);
00289         nextLeaf->setNextRBN(headerBuffer.rbnAvail);
00290         headerBuffer.rbnAvail = nextLeaf->getCurRBN();
00291 
00292         // Write merged block
00293         leaf->write(file, headerBuffer.headerRecordSize, curRBN);
00294         // Write old block (now empty)
00295         nextLeaf->write(file, headerBuffer.headerRecordSize, nextRBN);
00296         // Write parent
00297         parent->write(file, headerBuffer.headerRecordSize, parent->getCurRBN());
00298     } else {
00299         leaf->write(file, headerBuffer.headerRecordSize, curRBN);
00300         parent->write(file, headerBuffer.headerRecordSize, parent->getCurRBN());
00301     }
00302 
00303     if (parent->isUnderFilled()) {
00304         BTreeNode *newParent = findParentNode(parent);
00305         handleMerge(newParent, parent);
00306     }
00307 }
00308 
00309 BTreeNode* BTreeFile::findParentNode(BTreeNode* childNode) {
00310     BTreeNode* currentNode = this->root;
00311     BTreeNode* parentNode = nullptr;
00312 
00313     while (currentNode && !currentNode->getIsLeaf()) {
00314         int childRBN = currentNode->getNextChild(childNode->getLargestKey());
00315 
00316         if (childRBN == childNode->getCurRBN()) {
00317             if (parentNode == nullptr) return root;
00318             else return currentNode;
00319         }
00320 
00321         parentNode = currentNode;
00322         currentNode = new BTreeNode(order);
00323         currentNode->read(file, headerBuffer.headerRecordSize, childRBN);
00324     }
00325 
00326     return nullptr;
00327 }
00328 
00329 BTreeNode* BTreeFile::findLeafNode(int key) {
00330     BTreeNode * currentNode = this->root;
00331 
00332     while (!currentNode->getIsLeaf()) {
00333         int childKey = currentNode->getNextChild(key);
00334         // read node from child key
00335         BTreeNode * newNode = new BTreeNode(order);
00336         newNode->read(file, headerBuffer.headerRecordSize, childKey);
00337         currentNode = newNode;
00338     }
00339 
00340     return currentNode;
00341 }
00342 
00343 void BTreeFile::displayNode(BTreeNode* node, ostream& ostream, int level, const string& prefix) {
00344     if (node == nullptr) return;
00345 
00346     ostream « prefix;
00347     if (level > 0) {
00348         ostream « "|-- ";
00349     }
00350 
00351     node->print(ostream);
00352 
00353     if (!node->getIsLeaf()) {
00354         vector<int> children = node->getChildren();
00355         for (int i = 0; i < children.size(); i++) {
00356             BTreeNode * childNode = new BTreeNode(order);
00357             childNode->read(file, headerBuffer.headerRecordSize, children[i]);
00358             string newPrefix = prefix + (i < children.size() - 1 ? "|   " : "    ");
00359             displayNode(childNode, ostream, level + 1, newPrefix);
00360         }
00361     }
00362 }
00363 
00364 bool BTreeFile::flushData() {
00365     if (file.is_open()) {
00366         file.flush();
00367         file.close();
00368     } else return false;
00369 
00370     file.open(filename.c_str(), std::ios::in | std::ios::out | std::ios::binary);
00371     return file.is_open();
00372 }
00373 
```

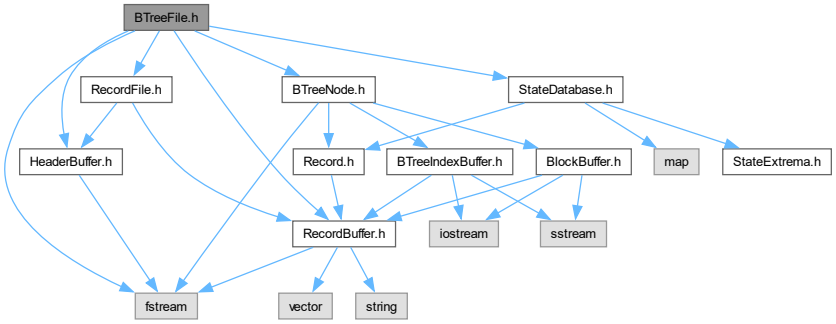
00374

4.7 BTreeFile.h File Reference

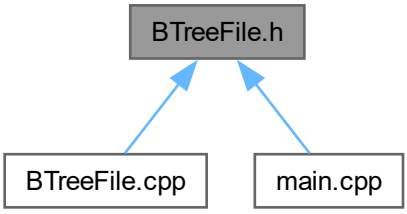
Header file for the BTreeFile class.

```
#include "RecordBuffer.h"
#include "HeaderBuffer.h"
#include "BTreeNode.h"
#include "RecordFile.h"
#include "StateDatabase.h"
#include <fstream>
```

Include dependency graph for BTreeFile.h:



This graph shows which files directly or indirectly include this file:



Classes

- class BTreeFile

  A class for building the Btree File.

4.7.1 Detailed Description

Header file for the BTreeFile class.

Author

Team 5

Date

December 9, 2023

Version

1.0

Definition in file BTreeFile.h.

4.8 BTreeFile.h

Go to the documentation of this file.

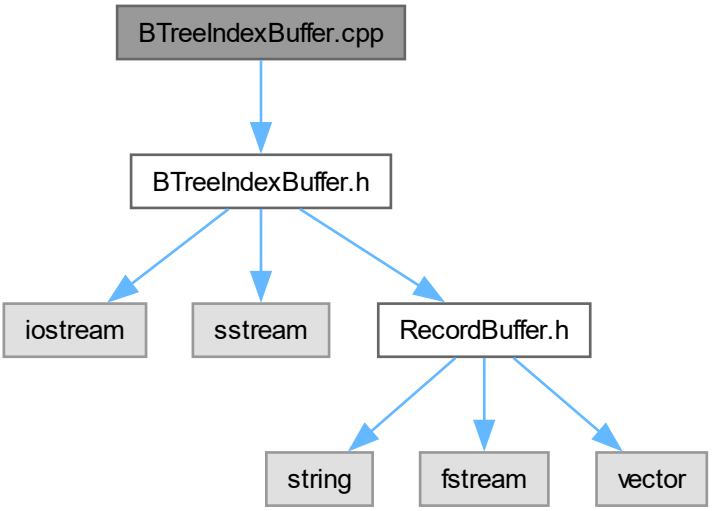
```
00001
00017 #ifndef PROJECT2_BTREEFILE_H
00018 #define PROJECT2_BTREEFILE_H
00019
00020 #include "RecordBuffer.h"
00021 #include "HeaderBuffer.h"
00022 #include "BTreeNode.h"
00023 #include "RecordFile.h"
00024 #include "StateDatabase.h"
00025 #include <fstream>
00026
00027 class BTreeFile
00028 {
00029 public:
00036     BTreeFile(HeaderBuffer &hbuf, int order);
00037
00042     ~BTreeFile();
00043
00049     bool openFile(std::string &bTreeFileName);
00050
00056     int insert(RecordBuffer& recordBuffer);
00057
00063     int remove(RecordBuffer& recordBuffer);
00064
00071     int search(RecordBuffer& recordBuffer, int key);
00072
00078     void displaySequenceSet(std::ostream& ostream);
00079
00085     void displayExtrema(std::ostream& ostream, std::string state);
00086
00092     void displayTree(std::ostream& ostream);
00093
00094 private:
00095     HeaderBuffer &headerBuffer;
00096     std::fstream file;
00097     std::string filename;
00098     BTreeNode* root;
00099     int order;
00100     int height;
00106     bool closeFile();
00107
00114     void handleRootSplit(int largestKey, BTreeNode* leaf, BTreeNode* newLeaf);
00115
00122     void handleNonRootSplit(int largestKey, BTreeNode* leaf, BTreeNode* newLeaf);
00123
00129     void handleMerge(BTreeNode *parent, BTreeNode *leaf);
00130
00136     BTreeNode* findParentNode(BTreeNode* childNode);
00137
00143     BTreeNode* findLeafNode(int key);
00144
00153     void displayNode(BTreeNode* node, std::ostream& ostream, int level, const std::string& prefix);
00154
00159     bool flushData();
00160
00161 };
00162
00163 #endif // PROJECT2_BTREEFILE_H
```

4.9 BTreeIndexBuffer.cpp File Reference

Implementation file for the BTreeIndexBuffer class.

```
#include "BTreeIndexBuffer.h"
```

Include dependency graph for BTreeIndexBuffer.cpp:



4.9.1 Detailed Description

Implementation file for the BTreeIndexBuffer class.

Author

Team 5

Date

December 9, 2023

Version

2.0

Definition in file BTreeIndexBuffer.cpp.

4.10 BTreeIndexBuffer.cpp

Go to the documentation of this file.

```
00001
00009 #include "BTreeIndexBuffer.h"
00010 using namespace std;
00011
00012 BTreeIndexBuffer::BTreeIndexBuffer(int blockSz, int minCap) {
00013     blockSize = blockSz;
00014     minimumBlockCapacity = minCap;
00015 }
00016
00017 int BTreeIndexBuffer::read(std::istream &stream, int headerRecordSize, int blockNumber) {
00018     // Move to location if needed
00019     if (blockNumber != -1) {
00020         stream.clear();
00021         stream.seekg((blockNumber-1) * blockSize + headerRecordSize);
00022     }
00023
00024     // check input stream
00025     if (!stream) return -1;
00026
00027     // Get current location in stream
00028     int addr = stream.tellg();
00029
00030     // Clear current buffer
00031     clear();
00032
00033     // Read block into buffer
00034     vector<char> buf(blockSize);
00035     stream.read(buf.data(), blockSize);
00036
00037     // Used to mark index nodes
00038     if (buf[0] != 'I') {
00039         return -1;
00040     }
00041     // Remove end spaces as they are rewritten in write function
00042     auto pos = std::find_if_not(buf.rbegin(), buf.rend(), [](char c) { return std::isspace(c) || c ==
      '\n'; }).base();
00043     buffer.write(buf.data(), std::distance(buf.begin(), pos)+1);
00044
00045     // check stream
00046     if (stream.bad()) {
00047         stream.clear();
00048         return -1;
00049     }
00050
00051     return addr;
00052 }
00053
00054 int BTreeIndexBuffer::write(std::ostream &stream, int headerRecordSize, int blockNumber) {
00055     // Move to location if needed
00056     if (blockNumber != -1) {
00057         stream.seekp((blockNumber-1) * blockSize + headerRecordSize);
00058     }
00059
00060     // Get output location
00061     int addr = stream.tellp();
00062
00063     // Write the buffer
00064     string metadata = "I\n";
00065     string str = metadata + buffer.str();
00066
00067     int sz = str.length();
00068     int remainingSpace = blockSize - sz;
00069     if (remainingSpace > 0) {
00070         str += string(remainingSpace-1, ' '); str += '\n';
00071     }
00072     stream.write(str.c_str(), blockSize);
00073
00074     // check stream
00075     if (!stream) return -1;
00076
00077     return addr;
00078 }
00079
00080 int BTreeIndexBuffer::unpack(std::vector<int>& separators, std::vector<int>& RBNs) {
00081     std::string buf = buffer.str();
00082
00083     size_t semicolonPos = buf.find(';');
00084     if (semicolonPos == std::string::npos) {
00085         return -1;
00086     }
00087
00088     std::string keysStr = buf.substr(2, semicolonPos);
```

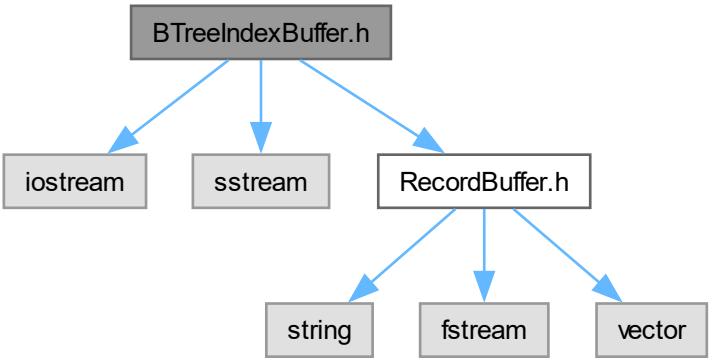
```
00089     std::string RBNsStr = buf.substr(semicolonPos + 1);
00090
00091     auto splitAndConvertToInt = [](const std::string& str, char delimiter) {
00092         std::vector<int> result;
00093         std::stringstream ss(str);
00094         std::string item;
00095         while (std::getline(ss, item, delimiter)) {
00096             result.push_back(std::stoi(item));
00097         }
00098         return result;
00099     };
00100
00101     separators = splitAndConvertToInt(keysStr, ',');
00102     RBNs = splitAndConvertToInt(RBNsStr, ',');
00103
00104     return 0;
00105 }
00106
00107 int BTreeIndexBuffer::pack(std::vector<int> seperators, std::vector<int> RBNs) {
00108
00109     string buf;
00110
00111     for (int i = 0; i < seperators.size(); i++) {
00112         buf += to_string(seperators[i]);
00113         if (i < seperators.size()-1) {
00114             buf += ",";
00115         }
00116     }
00117
00118     buf += ";";
00119
00120     for (int i = 0; i < RBNs.size(); i++) {
00121         buf += to_string(RBNs[i]);
00122         if (i < RBNs.size()-1) {
00123             buf += ",";
00124         }
00125     }
00126
00127     buf += "\n";
00128
00129     if (buffer.str().size() +  buf.size() <= blockSize) {
00130         buffer « buf;
00131         return 0;
00132     }
00133
00134     return -1;
00135 }
00136
00137 void BTreeIndexBuffer::clear() {
00138     buffer.clear();
00139     buffer.str("");
00140 }
```

4.11 BTreeIndexBuffer.h File Reference

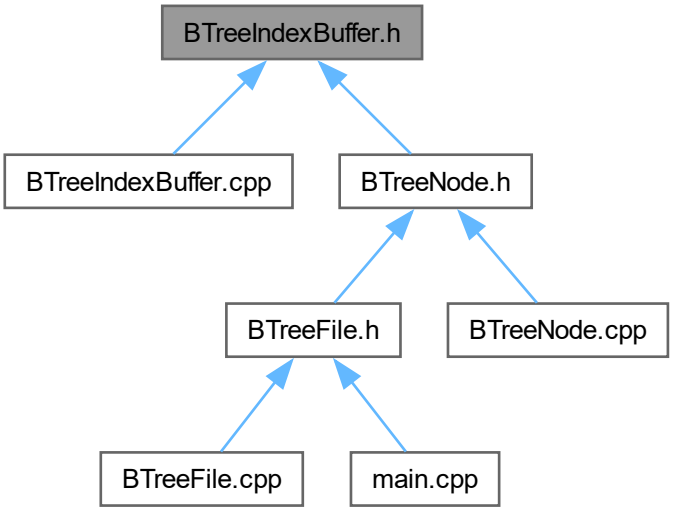
Header file for the BTreeIndexBuffer class.

```
#include <iostream>
#include <sstream>
#include "RecordBuffer.h"
```

Include dependency graph for BTreeIndexBuffer.h:



This graph shows which files directly or indirectly include this file:



Classes

- class BTreeIndexBuffer

  A class for managing a buffer for a files header information.

4.11.1 Detailed Description

Header file for the BTreeIndexBuffer class.

Author

Team 5

Date

December 9, 2023

Version

2.0

Definition in file BTreeIndexBuffer.h.

4.12 BTreeIndexBuffer.h

Go to the documentation of this file.

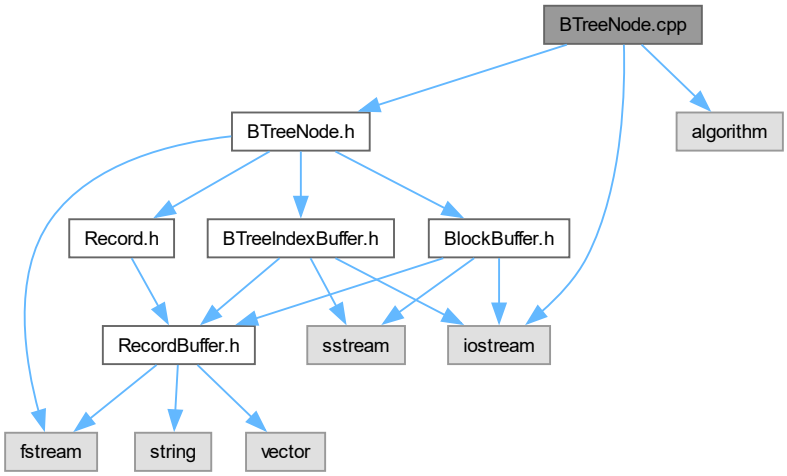
```
00001
00017 #ifndef CSCI331_PROJECT4_BTREEINDEXBUFFER
00018 #define CSCI331_PROJECT4_BTREEINDEXBUFFER
00019
00020 #include <iostream>
00021 #include <sstream>
00022 #include "RecordBuffer.h"
00023
00024 class BTreeIndexBuffer {
00025 public:
00031     BTreeIndexBuffer(int blockSz = 512, int minCap = 256);
00032
00040     int read(std::istream & stream, int headerRecordSize, int blockNumber = -1);
00041
00049     int write(std::ostream & stream, int headerRecordSize, int blockNumber = -1);
00050
00057     int unpack(std::vector<int>& seperators, std::vector<int>& RBNs);
00058
00065     int pack(std::vector<int> seperators, std::vector<int> RBNs);
00066
00071     void clear();
00072
00073 private:
00074     std::stringstream buffer;
00075     int blockSize;
00076     int minimumBlockCapacity;
00077     int numSeparators;
00078     int lengthSeparators;
00079 };
00080
00081 #endif //CSCI331_PROJECT4_BTREEINDEXBUFFER
```

4.13 BTreeNode.cpp File Reference

Implementation file for the BTreeNode class.

```
#include "BTreeNode.h"
#include <iostream>
```

`#include <algorithm>`
Include dependency graph for BTreeNode.cpp:



4.13.1 Detailed Description

Implementation file for the BTreeNode class.

Author

Team 5

Date

December 9, 2023

Version

2.0

Definition in file BTreeNode.cpp.

4.14 BTreeNode.cpp

Go to the documentation of this file.

```
00001
00009 #include "BTreeNode.h"
00010 #include <iostream>
00011 #include <algorithm>
00012
00013 using namespace std;
00014
00015 BTreeNode::BTreeNode(int maxKeys) : maxKeys(maxKeys), minKeys(maxKeys / 2) {
00016     isLeaf = true;
00017 }
```

```
00018
00019 int BTreeNode::read(std::istream& stream, int headerRecordSize, int RBN) {
00020     int addr = bTreeIndexBuffer.read(stream, headerRecordSize, RBN);
00021     if (addr != -1) {
00022         curRBN = addr / headerRecordSize;
00023         bTreeIndexBuffer.unpack(keys, children);
00024         numKeys = keys.size();
00025         isLeaf = false;
00026     } else {
00027         return blockBuffer.read(stream, headerRecordSize, RBN);
00028     }
00029
00030     return addr;
00031 }
00032
00033 int BTreeNode::write(std::ostream& stream, int headerRecordSize, int RBN) {
00034     if (isLeaf) {
00035         return blockBuffer.write(stream, headerRecordSize, RBN);
00036     } else {
00037         bTreeIndexBuffer.clear();
00038         bTreeIndexBuffer.pack(keys, children);
00039         int addr = bTreeIndexBuffer.write(stream, headerRecordSize, RBN);
00040         curRBN = addr / headerRecordSize;
00041         return addr;
00042     }
00043 }
00044
00045 int BTreeNode::insertRecord(RecordBuffer& recordBuffer) {
00046     if (!isLeaf || blockBuffer.pack(recordBuffer) == -1 || blockBuffer.sortBuffer() == -1 ||
      blockBuffer.isOverFilled()) {
00047         return -1;
00048     }
00049     return 0;
00050 }
00051
00052 int BTreeNode::removeRecord(RecordBuffer& recordBuffer) {
00053     if (!isLeaf || blockBuffer.removeRecord(recordBuffer.getRecordKey()) == -1 ||
      blockBuffer.isUnderFilled()) {
00054         return -1;
00055     }
00056     return 0;
00057 }
00058
00059 int BTreeNode::retrieveRecord(RecordBuffer& recordBuffer, int key) {
00060     while(blockBuffer.unpack(recordBuffer) != -1) {
00061         if (recordBuffer.getRecordKey() == key) {
00062             return 0;
00063         }
00064     }
00065
00066     recordBuffer.clear();
00067     return -1;
00068 }
00069
00070 int BTreeNode::insertKeyAndChildren(int key, int child1, int child2) {
00071     // Find the position to insert the key
00072     auto keyIt = std::lower_bound(keys.begin(), keys.end(), key);
00073     int keyIndex = std::distance(keys.begin(), keyIt);
00074
00075     // Insert the key
00076     keys.insert(keyIt, key);
00077
00078     // When inserting a new root, the children are the original root and the new node created from a
      split.
00079     // child1 should be at the current index (as children are one more than keys) and child2 right
      after it.
00080     if (child2 != -1) {
00081         children.insert(children.begin() + keyIndex, child1);
00082         children.insert(children.begin() + keyIndex + 1, child2);
00083     } else {
00084         // For other cases, insert the child to the right of the key
00085         children.insert(children.begin() + keyIndex + 1, child1);
00086     }
00087
00088     numKeys++;
00089     return 0;
00090 }
00091
00092
00093
00094 int BTreeNode::removeKeyAndChildren(int key, int child) {
00095     auto it = std::find(keys.begin(), keys.end(), key);
00096     if (it == keys.end()) {
00097         return -1;
00098     }
00099
00100     keys.erase(it);
```

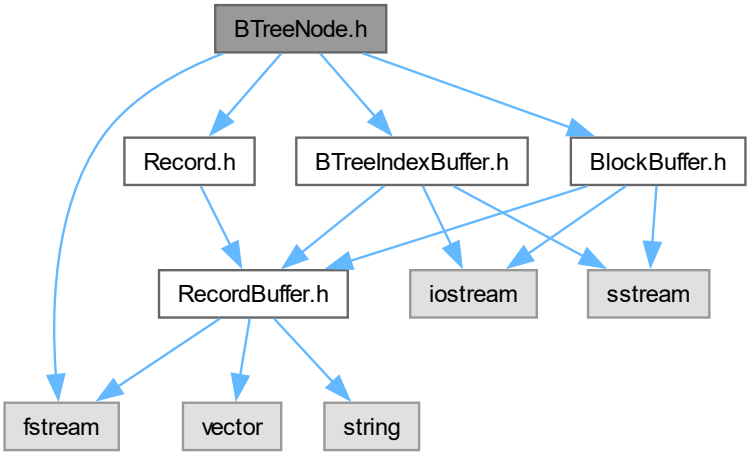
```
00101     numKeys--;
00102
00103     it = std::find(children.begin(), children.end(), child);
00104     if (it == children.end()) {
00105         return -1;
00106     }
00107
00108     children.erase(it);
00109     return 0;
00110 }
00111
00112 void BTreeNode::print(std::ostream &stream) {
00113     if (isLeaf) {
00114         cout « "LEAF NODE: LARGEST KEY = " « getLargestKey() « endl;
00115     } else {
00116         cout « "INDEX NODE: RBN = " « getCurRBN() « ", KEYS = ";
00117         for (int key : keys) {
00118             stream « key « " ";
00119         }
00120         stream « endl;
00121     }
00122 }
00123
00124 int BTreeNode::split(BTreeNode *newNode) {
00125     if (isLeaf) {
00126         blockBuffer.splitBuffer(newNode->blockBuffer);
00127     } else {
00128         int midIndex = keys.size() / 2;
00129         int splitKey = keys[midIndex];  // The key to move up to the parent
00130
00131         // Move the right half of the keys to the new node
00132         for (int i = midIndex + 1; i < keys.size(); i++) {
00133             newNode->keys.push_back(keys[i]);
00134             newNode->children.push_back(children[i]);
00135         }
00136         newNode->children.push_back(children[keys.size()]);
00137
00138         // Adjust the original node
00139         keys.erase(keys.begin() + midIndex + 1, keys.end());
00140         children.erase(children.begin() + midIndex + 1, children.end());
00141
00142         return splitKey;
00143     }
00144
00145     return -1;
00146 }
00147
00148 int BTreeNode::merge(BTreeNode *fromNode) {
00149     if (isLeaf) {
00150         blockBuffer.mergeBuffer(fromNode->blockBuffer);
00151     } else {
00152         for (auto key : fromNode->keys) {
00153             keys.push_back(key);
00154         }
00155         for (auto child : fromNode->children) {
00156             keys.push_back(child);
00157         }
00158     }
00159
00160     return 0;
00161 }
00162
00163 bool BTreeNode::getIsLeaf() {
00164     return isLeaf;
00165 }
00166
00167 void BTreeNode::setIsLeaf(bool isL) {
00168     isLeaf = isL;
00169 }
00170
00171 int BTreeNode::getNextChild(int key) {
00172     if (isLeaf) {
00173         return -1;
00174     }
00175
00176     int numChildren = children.size();
00177     for (int i = 0; i < numChildren - 1; i++) {
00178         if (key <= keys[i]) {
00179             return children[i];
00180         }
00181     }
00182
00183     return children[numChildren - 1];
00184 }
00185
00186 std::vector<int> BTreeNode::getChildren() {
00187     return children;
```

```
00188 }
00189
00190 int BTreeNode::getLargestKey() {
00191     if (isLeaf) {
00192         return blockBuffer.getLargestKey();
00193     }
00194
00195     if (keys.empty()) {
00196         return -1;
00197     }
00198
00199     return keys[keys.size() -1];
00200 }
00201
00202
00203 void BTreeNode::setCurRBN(int rbn) {
00204     if (isLeaf) {
00205         blockBuffer.setCurRBN(rbn);
00206     } else {
00207         curRBN = rbn;
00208     }
00209 }
00210
00211 int BTreeNode::getCurRBN() {
00212     if (isLeaf) {
00213         return blockBuffer.getCurRBN();
00214     }
00215
00216     return curRBN;
00217 }
00218
00219 int BTreeNode::getPrevRBN() {
00220     if (isLeaf) {
00221         return blockBuffer.getPrevRBN();
00222     } else return -1;
00223 }
00224
00225 void BTreeNode::setPrevRBN(int rbn) {
00226     if (isLeaf) {
00227         blockBuffer.setPrevRBN(rbn);
00228     }
00229 }
00230
00231 int BTreeNode::getNextRBN() {
00232     if (isLeaf) {
00233         return blockBuffer.getNextRBN();
00234     } else return -1;
00235 }
00236
00237 void BTreeNode::setNextRBN(int rbn) {
00238     if (isLeaf) {
00239         blockBuffer.setNextRBN(rbn);
00240     }
00241 }
00242
00243 bool BTreeNode::isOverFilled() {
00244     return numKeys > maxKeys;
00245 }
00246
00247 bool BTreeNode::isUnderFilled() {
00248     if (isLeaf) {
00249         return blockBuffer.isUnderFilled();
00250     } else {
00251         return numKeys < minKeys;
00252     }
00253 }
```

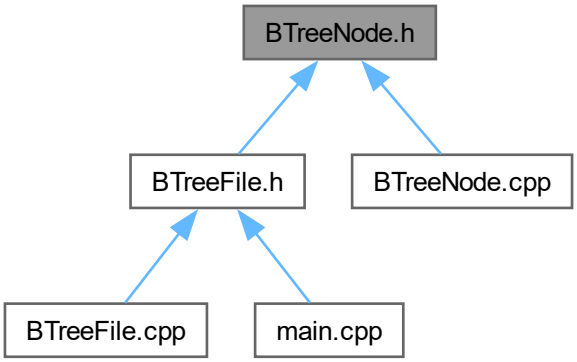
4.15 BTreeNode.h File Reference

```
#include <fstream>
#include "BTreeIndexBuffer.h"
#include "BlockBuffer.h"
#include "Record.h"
```

Include dependency graph for BTreeNode.h:



This graph shows which files directly or indirectly include this file:



Classes

- class BTreeNode

4.16 BTreeNode.h

Go to the documentation of this file.

```
00001
```

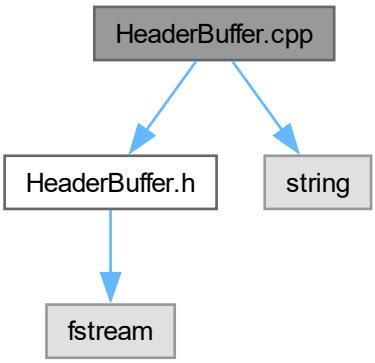
```
00002 #ifndef CSCI331_PROJECT3_BTREENODE_H
00003 #define CSCI331_PROJECT3_BTREENODE_H
00004
00005 #include <fstream>
00006 #include "BTreeIndexBuffer.h"
00007 #include "BlockBuffer.h"
00008 #include "Record.h"
00009
00010 class BTreeNode {
00011 public:
00017     BTreeNode(int maxKeys);
00018
00026     int read(std::istream& stream, int headerRecordSize, int RBN);
00027
00035     int write(std::ostream& stream, int headerRecordSize, int RBN);
00036
00042     int insertRecord(RecordBuffer& recordBuffer);
00043
00049     int removeRecord(RecordBuffer& recordBuffer);
00050
00057     int retrieveRecord(RecordBuffer& recordBuffer, int key);
00058
00066     int insertKeyAndChildren(int key, int child1, int child2 = -1);
00067
00074     int removeKeyAndChildren(int key, int child);
00075
00081     void print(std::ostream & stream);
00082
00088     int split(BTreeNode * newNode);
00089
00095     int merge(BTreeNode * fromNode);
00096
00102     int getNextChild(int key);
00103
00108     std::vector<int> getChildren();
00109
00114     int getLargestKey();
00115
00120     bool getIsLeaf();
00121
00127     void setIsLeaf(bool isL);
00128
00133     int getCurRBN();
00134
00140     void setCurRBN(int rbn);
00141
00146     int getPrevRBN();
00147
00153     void setPrevRBN(int rbn);
00154
00159     int getNextRBN();
00160
00166     void setNextRBN(int rbn);
00167
00172     bool isOverFilled();
00173
00178     bool isUnderFilled();
00179
00180 private:
00181     BlockBuffer blockBuffer;
00182     BTreeIndexBuffer bTreeIndexBuffer;
00183     int curRBN;
00184     int maxKeys;
00185     int minKeys;
00186     int numKeys;
00187     bool isLeaf;
00188     std::vector<int> keys;
00189     std::vector<int> children;
00190 };
00191
00192 #endif //CSCI331_PROJECT3_BTREENODE_H
00193
```

4.17 HeaderBuffer.cpp File Reference

Implementation file for the HeaderBuffer class.

```
#include "HeaderBuffer.h"
#include <string>
```

Include dependency graph for HeaderBuffer.cpp:



4.17.1 Detailed Description

Implementation file for the HeaderBuffer class.

Author

Team 5

Date

November 18, 2023

Version

2.0

Definition in file HeaderBuffer.cpp.

4.18 HeaderBuffer.cpp

Go to the documentation of this file.

```
00001
00009 #include "HeaderBuffer.h"
00010 #include <string>
00011
00012 using namespace std;
00013
00014 HeaderBuffer::HeaderBuffer() {
00015     this->fileType = "";
00016     this->version = "2.0";
00017     this->headerRecordSize = 512;
00018     this->recordSizeDigits = 2;
00019     this->recordSizeFormat = "ASCII";
00020     this->blockSize = 512;
00021     this->minimumBlockCapacity = 256;
```

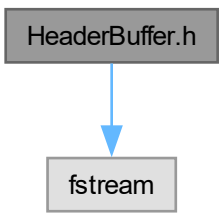
```
00022     this->recordCount = -1;
00023     this->blockCount = -1;
00024     this->recordFieldCount = 6;
00025     this->recordFieldsType = "STRING";
00026     this->recordFormat = "ZipCode,PlaceName,State,County,Lat,Long";
00027     this->recordPrimaryKey = 1;
00028     this->rbnAvail = 2;
00029     this->rbnActive = 1;
00030     this->stale = "true";
00031 }
00032
00033 int HeaderBuffer::readHeader(std::istream &stream)
00034 {
00035     string line;
00036
00037     while (std::getline(stream, line) && line != "END")
00038     {
00039
00040         int equalPos = line.find('=');
00041         if (equalPos != std::string::npos)
00042         {
00043             string key = line.substr(0, equalPos);
00044             string value = line.substr(equalPos + 1);
00045
00046             if (key == "FILE_TYPE")
00047             {
00048                 fileType = value;
00049             }
00050             else if (key == "VERSION")
00051             {
00052                 version = value;
00053             }
00054             else if (key == "HEADER_RECORD_SIZE")
00055             {
00056                 headerRecordSize = stoi(value);
00057             }
00058             else if (key == "RECORD_SIZE_DIGITS")
00059             {
00060                 recordSizeDigits = stoi(value);
00061             }
00062             else if (key == "RECORD_SIZE_FORMAT")
00063             {
00064                 recordSizeFormat = value;
00065             }
00066             else if (key == "BLOCK_SIZE")
00067             {
00068                 blockSize = stoi(value);
00069             }
00070             else if (key == "MINIMUM_BLOCK_CAPACITY")
00071             {
00072                 minimumBlockCapacity = stoi(value);
00073             }
00074             else if (key == "RECORD_COUNT")
00075             {
00076                 recordCount = stoi(value);
00077             }
00078             else if (key == "BLOCK_COUNT")
00079             {
00080                 blockCount = stoi(value);
00081             }
00082             else if (key == "RECORD_FIELD_COUNT")
00083             {
00084                 recordFieldCount = stoi(value);
00085             }
00086             else if (key == "RECORD_FIELDS_TYPE")
00087             {
00088                 recordFieldsType = value;
00089             }
00090             else if (key == "RECORD_FORMAT")
00091             {
00092                 recordFormat = value;
00093             }
00094             else if (key == "RECORD_PRIMARY_KEY")
00095             {
00096                 recordPrimaryKey = stoi(value);
00097             }
00098             else if (key == "RBN_AVAIL")
00099             {
00100                 rbnAvail = stoi(value);
00101             }
00102             else if (key == "RBN_ACTIVE")
00103             {
00104                 rbnActive = stoi(value);
00105             }
00106             else if (key == "STALE")
00107             {
00108                 stale = value;
```

```
00109             }
00110             else
00111             {
00112                 return -1;
00113             }
00114         }
00115         else
00116         {
00117             return -1;
00118         }
00119     }
00120
00121     // Return the stream position after reading the header
00122     return stream.tellg();
00123 }
00124
00125 int HeaderBuffer::writeHeader(std::ostream &stream) const
00126 {
00127     string buffer;
00128     stream.seekp(0, std::ios::beg);
00129
00130     buffer += "FILE_TYPE="; buffer += fileType; buffer += '\n';
00131     buffer += "VERSION="; buffer += version; buffer += '\n';
00132     buffer += "HEADER_RECORD_SIZE="; buffer += to_string(headerRecordSize); buffer += '\n';
00133     buffer += "RECORD_SIZE_DIGITS="; buffer += to_string(recordSizeDigits); buffer += '\n';
00134     buffer += "RECORD_SIZE_FORMAT="; buffer += recordSizeFormat; buffer += '\n';
00135     buffer += "BLOCK_SIZE="; buffer += to_string(blockSize); buffer += '\n';
00136     buffer += "MINIMUM_BLOCK_CAPACITY="; buffer += to_string(minimumBlockCapacity); buffer += '\n';
00137     buffer += "RECORD_COUNT="; buffer += to_string(recordCount); buffer += '\n';
00138     buffer += "BLOCK_COUNT="; buffer += to_string(blockCount); buffer += '\n';
00139     buffer += "RECORD_FIELD_COUNT="; buffer += to_string(recordFieldCount); buffer += '\n';
00140     buffer += "RECORD_FIELDS_TYPE="; buffer += recordFieldsType; buffer += '\n';
00141     buffer += "RECORD_FORMAT="; buffer += recordFormat; buffer += '\n';
00142     buffer += "RECORD_PRIMARY_KEY="; buffer += to_string(recordPrimaryKey); buffer += '\n';
00143     buffer += "RBN_AVAIL="; buffer += to_string(rbnAvail); buffer += '\n';
00144     buffer += "RBN_ACTIVE="; buffer += to_string(rbnActive); buffer += '\n';
00145     buffer += "STALE="; buffer += stale; buffer += '\n';
00146     buffer += "END"; buffer += '\n';
00147
00148     int remainingSpace = headerRecordSize - buffer.length() - 1;
00149     buffer += string(remainingSpace, ' '); buffer += '\n';
00150
00151     // Write the header to file
00152     stream << buffer;
00153
00154     // Return header size
00155     return headerRecordSize;
00156 }
```

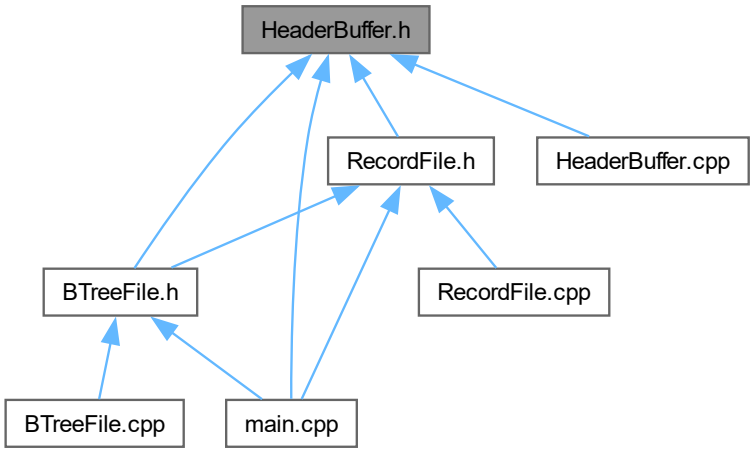
4.19 HeaderBuffer.h File Reference

Header file for the HeaderBuffer class.

`#include "fstream"`
Include dependency graph for HeaderBuffer.h:



This graph shows which files directly or indirectly include this file:



Classes

- class HeaderBuffer

  A class for managing a buffer for a files header information.

4.19.1 Detailed Description

Header file for the HeaderBuffer class.

Author

Team 5

Date

November 18, 2023

Version

2.0

Definition in file HeaderBuffer.h.

4.20 HeaderBuffer.h

Go to the documentation of this file.

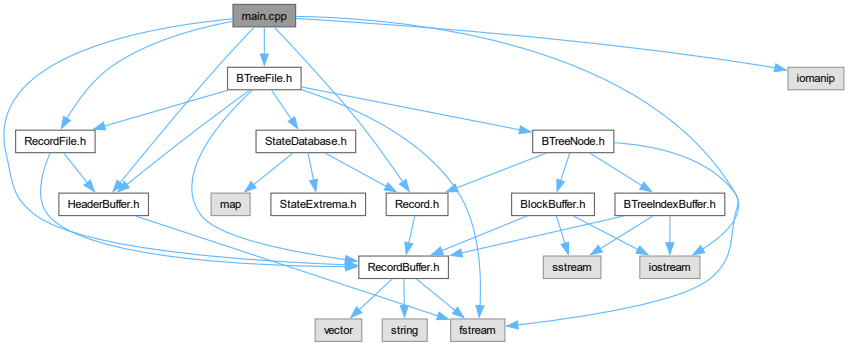
```
00001
00017 #ifndef PROJECT2_PART1_HEADERBUFFER_H
00018 #define PROJECT2_PART1_HEADERBUFFER_H
00019
00020 #include "fstream"
00021
00022 class HeaderBuffer
00023 {
00024
00025 public:
00029     HeaderBuffer();
00030
00036     int readHeader(std::istream &stream);
00037
00043     int writeHeader(std::ostream &stream) const;
00044
00045     std::string fileType;
00046     std::string version;
00047     int headerRecordSize;
00048     int recordSizeDigits;
00049     std::string recordSizeFormat;
00050     int blockSize;
00051     int minimumBlockCapacity;
00052     int recordCount;
00053     int blockCount;
00054     int recordFieldCount;
00055     std::string recordFieldsType;
00056     std::string recordFormat;
00057     int recordPrimaryKey;
00058     int rbnAvail;
00059     int rbnActive;
00060     std::string stale;
00061 };
00062
00063 #endif // PROJECT2_PART1_HEADERBUFFER_H
```

4.21 main.cpp File Reference

Main program file for processing CSV data related to ZIP code records using a B+ tree.

```
#include <iostream>
#include <iomanip>
#include "RecordBuffer.h"
#include "HeaderBuffer.h"
#include "Record.h"
#include "RecordFile.h"
#include "BTreeFile.h"
```

Include dependency graph for main.cpp:



Functions

- bool processCommandLine (int argc, char ∗argv[ ], HeaderBuffer &headerBuffer, vector< vector< string > > &actions)

  Processes command line arguments to configure operations such as adding or deleting records, and adjusting B+ tree properties.
- void addRecords (BTreeFile &bTreeFile, HeaderBuffer &headerBuffer, const string &fileName)

  Adds records to the B+ tree from a specified file.
- void deleteRecords (BTreeFile &bTreeFile, HeaderBuffer &headerBuffer, const string &fileName)

  Deletes records from the B+ tree based on the contents of a specified file.
- void searchIndex (BTreeFile &bTreeFile, vector< string > zipcodes)

  Searches the B+ tree for records matching the specified ZIP codes and displays them.
- int main (int argc, char ∗argv[ ])

  Main function which serves as the entry point for the program.

4.21.1 Detailed Description

Main program file for processing CSV data related to ZIP code records using a B+ tree.

This program supports adding, deleting, and searching records, as well as displaying various views of the data, such as extrema, sequence sets, and the tree structure itself.

Definition in file main.cpp.

4.21.2 Function Documentation

4.21.2.1 addRecords()

```
void addRecords (
            BTreeFile & bTreeFile,
            HeaderBuffer & headerBuffer,
            const string & fileName )
```

Adds records to the B+ tree from a specified file.

This function opens the file containing new records, reads each record into a RecordBuffer object, and then inserts each record into the B+ tree. If the specified file contains records in a format not compatible with direct insertion into the B+ tree, a pre-processing step might be necessary to convert the file into a suitable format.

Parameters

bTreeFile	Reference to the BTreeFile object to perform operations on the B+ tree.
headerBuffer	Reference to the HeaderBuffer object for managing file headers, used here for configuring the RecordFile object for reading the new records.
fileName	Name of the file containing the new records to add. The file format is expected to be compatible with the RecordBuffer and RecordFile specifications for direct reading and insertion into the B+ tree.

Definition at line 163 of file main.cpp.

4.21.2.2 deleteRecords()

```
void deleteRecords (
            BTreeFile & bTreeFile,
            HeaderBuffer & headerBuffer,
            const string & fileName )
```

Deletes records from the B+ tree based on the contents of a specified file.

This function iterates through each record in the file, constructs a RecordBuffer object for it, and then attempts to remove it from the B+ tree.

Parameters

bTreeFile	Reference to the BTreeFile object to operate on the B+ tree.
headerBuffer	Reference to the HeaderBuffer object for managing file headers.
fileName	Name of the file containing records to delete.

Definition at line 196 of file main.cpp.

4.21.2.3 main()

```
int main (
            int argc,
            char * argv[] )
```

Main function which serves as the entry point for the program.

It processes command line arguments to perform various operations on ZIP code records using a B+ tree structure.

Parameters

argc	Number of command line arguments.
argv	Array of command line arguments.

Returns

int Program exit status.

Definition at line 32 of file main.cpp.

4.21.2.4 processCommandLine()

```
bool processCommandLine (
            int argc,
            char * argv[],
            HeaderBuffer & headerBuffer,
            vector< vector< string > > & actions )
```

Processes command line arguments to configure operations such as adding or deleting records, and adjusting B+ tree properties.

It parses the arguments to determine the actions to be taken, configuring parameters like block size and the minimum block capacity, or scheduling tasks like adding records, deleting records, displaying various data views, or searching within the B+ tree.

Parameters

argc	The number of command line arguments.
argv	The array containing the command line arguments.
headerBuffer	A reference to the HeaderBuffer object for managing file header settings.
actions	A reference to a vector that will store parsed actions and their parameters derived from the arguments.

Returns

A boolean indicating if the command line processing was successful.

Definition at line 87 of file main.cpp.

4.21.2.5 searchIndex()

```
void searchIndex (
            BTreeFile & bTreeFile,
            vector< string > zipcodes )
```

Searches the B+ tree for records matching the specified ZIP codes and displays them.

Each ZIP code provided in the zipcodes vector is converted to an integer and searched within the B+ tree. If a record is found, it is displayed; otherwise, a not found message is shown.

Parameters

bTreeFile	Reference to the BTreeFile object for B+ tree operations.
zipcodes	A vector of strings representing the ZIP codes to search for.

Definition at line 226 of file main.cpp.

4.22 main.cpp

Go to the documentation of this file.

```
00001
00008 #include <iostream>
00009 #include <iomanip>
00010 #include "RecordBuffer.h"
00011 #include "HeaderBuffer.h"
00012 #include "Record.h"
00013 #include "RecordFile.h"
00014 #include "BTreeFile.h"
00015
00016 using namespace std;
00017
00018 // Function prototypes
00019 bool processCommandLine(int argc, char* argv[], HeaderBuffer &headerBuffer, vector<vector<string»
      &actions);
00020 void addRecords(BTreeFile &bTreeFile, HeaderBuffer &headerBuffer, const string& fileName);
```

```
00021 void deleteRecords(BTreeFile &bTreeFile, HeaderBuffer &headerBuffer, const string& fileName);
00022 void searchIndex(BTreeFile &bTreeFile, vector<string> zipcodes);
00023
00032 int main(int argc, char* argv[]) {
00033     HeaderBuffer headerBuffer; // Object to manage file header operations.
00034
00035     string bTreeFileName; // Name of the B+ tree file.
00036     vector<vector<string> actions; // Stores parsed command line actions.
00037
00038     // Process command line arguments and exit if failed.
00039     if (!processCommandLine(argc, argv, headerBuffer, actions)) {
00040         return -1;
00041     } else {
00042         bTreeFileName = argv[argc - 1]; // Last argument is the B+ tree file name.
00043     }
00044
00045     BTreeFile bTreeFile(headerBuffer, 10); // Initialize B+ tree with default order 10.
00046     // Attempt to open the B+ tree file and exit if failed.
00047     if (!bTreeFile.openFile(bTreeFileName)) {
00048         cout « "Failed to open " « bTreeFileName « "!" « endl;
00049         return -1;
00050     }
00051
00052     // Iterate through actions derived from command line arguments and perform them.
00053     for (int i = 0; i < actions.size(); i++) {
00054         string action = actions[i][0]; // Action type (e.g., -ADD_RECORDS, -SEARCH).
00055         // Call specific function based on the action.
00056         if (action == "-ADD_RECORDS") {
00057             addRecords(bTreeFile, headerBuffer, actions[i][1]);
00058         } else if (action == "-DELETE_RECORDS") {
00059             deleteRecords(bTreeFile, headerBuffer, actions[i][1]);
00060         } else if (action == "-DISPLAY_EXTREMA") {
00061             bTreeFile.displayExtrema(cout, actions[i][1]);
00062         } else if (action == "-DISPLAY_SEQUENCE_SET") {
00063             bTreeFile.displaySequenceSet(cout);
00064         } else if (action == "-DUMP_TREE") {
00065             bTreeFile.displayTree(cout);
00066         } else if (action == "-SEARCH") {
00067             searchIndex(bTreeFile, actions[i]);
00068         }
00069     }
00070
00071     return 0;
00072 }
00073
00074
00087 bool processCommandLine(int argc, char* argv[], HeaderBuffer &headerBuffer, vector<vector<string»
      &actions) {
00088     // Ensure at least the filename is provided.
00089     if (argc < 2) {
00090         cout « "Error: Filename parameter is required." « endl;
00091         return false;
00092     }
00093
00094     // Iterate through all arguments to configure settings or schedule tasks.
00095     for (int i = 1; i < argc - 1; i++) {
00096         string arg = argv[i]; // Current argument being processed.
00097
00098         if (arg == "-BLOCK_SIZE") {
00099             if (i + 1 < argc) {
00100                 headerBuffer.blockSize = stoi(argv[++i]); // Parse and set block size, advance to next
      argument.
00101             } else {
00102                 cout « "Error: -BLOCK_SIZE flag requires a numerical value." « endl;
00103                 return false;
00104             }
00105         } else if (arg == "-MINIMUM_BLOCK_CAPACITY") {
00106             if (i + 1 < argc) {
00107                 headerBuffer.minimumBlockCapacity = stoi(argv[++i]); // Parse and set minimum block
      capacity, advance.
00108             } else {
00109                 cout « "Error: -MINIMUM_BLOCK_CAPACITY flag requires a numerical value." « endl;
00110                 return false;
00111             }
00112         } else if (arg == "-ADD_RECORDS") {
00113             if (i + 1 < argc) {
00114                 actions.push_back({arg, argv[++i]}); // Schedule addition of records, move past
      filename.
00115             } else {
00116                 cout « "Error: -ADD_RECORDS flag requires a filename." « endl;
00117                 return false;
00118             }
00119         } else if (arg == "-DELETE_RECORDS") {
00120             if (i + 1 < argc) {
00121                 actions.push_back({arg, argv[++i]}); // Schedule deletion of records, move past
      filename.
00122             } else {
```

```
00123                 cout « "Error: -DELETE_RECORDS flag requires a filename." « endl;
00124                 return false;
00125             }
00126         } else if (arg == "-DISPLAY_EXTREMA") {
00127             vector<string> tmp = {arg};
00128             if (i + 1 < argc && argv[i + 1][0] != '-') {
00129                 tmp.push_back(argv[++i]); // Add state argument if present, then advance.
00130             }
00131             actions.push_back(tmp);
00132         } else if (arg == "-DISPLAY_SEQUENCE_SET") {
00133             actions.push_back({arg}); // Schedule display of the sequence set.
00134         } else if (arg == "-DUMP_TREE") {
00135             actions.push_back({arg}); // Schedule display of the B+ tree structure.
00136         } else if (arg == "-SEARCH") {
00137             vector<string> tmp = {arg};
00138             // Accumulate all zip codes until another flag or the end of arguments.
00139             while (i + 1 < argc && argv[i + 1][0] != '-') {
00140                 tmp.push_back(argv[++i]);
00141             }
00142             actions.push_back(tmp);
00143         }
00144     }
00145
00146     return true; // Command line arguments processed successfully.
00147 }
00148
00149
00163 void addRecords(BTreeFile &bTreeFile, HeaderBuffer &headerBuffer, const string& fileName) {
00164     RecordBuffer recordBuffer; // Buffer for individual records to be added.
00165     RecordFile newRecordsFile(headerBuffer); // File handler for the new records, configured with the
      header buffer.
00166
00167     string lengthIndicatedFile; // Filename for a temporary, length-indicated version of the records
      file.
00168     // Attempt to open the file containing new records. If unsuccessful, print an error message and
      exit this function.
00169     if (!newRecordsFile.openFile(fileName, lengthIndicatedFile)) {
00170         cout « "Failed to open " « fileName « " for adding records." « endl;
00171         return;
00172     }
00173
00174     // Read each record from the file into the record buffer, then insert it into the B+ tree.
00175     while (newRecordsFile.read(recordBuffer) != -1) {
00176         // Insert the current record into the B+ tree.
00177         bTreeFile.insert(recordBuffer);
00178     }
00179
00180     // Clean up: Attempt to delete the temporary file used for processing.
00181     if (std::remove(lengthIndicatedFile.c_str()) != 0) {
00182         cout « "Failed to delete temporary file: " « lengthIndicatedFile « endl;
00183     }
00184 }
00185
00186
00196 void deleteRecords(BTreeFile &bTreeFile, HeaderBuffer &headerBuffer, const string& fileName) {
00197     RecordBuffer recordBuffer; // Buffer for individual records to be deleted.
00198     RecordFile deleteRecordsFile(headerBuffer); // Handles file operations for records to delete.
00199
00200     string lengthIndicatedFile; // Temporary file for length-indicated processing.
00201     if (!deleteRecordsFile.openFile(fileName, lengthIndicatedFile)) {
00202         cout « "Failed to open " « fileName « " for deletion!" « endl;
00203         return;
00204     }
00205
00206     // Loop to read each record from the file and attempt to delete from the B+ tree.
00207     while (deleteRecordsFile.read(recordBuffer) != -1) {
00208         bTreeFile.remove(recordBuffer);
00209     }
00210
00211     // Clean up: Attempt to delete the temporary file used for processing.
00212     if (std::remove(lengthIndicatedFile.c_str()) != 0) {
00213         cout « "Failed to delete temporary file: " « lengthIndicatedFile « endl;
00214     }
00215 }
00216
00217
00226 void searchIndex(BTreeFile &bTreeFile, vector<string> zipcodes) {
00227     RecordBuffer recordBuffer; // Buffer to hold the search result record.
00228
00229     // Skip the first element (action command) and start from the first ZIP code.
00230     for (int i = 1; i < zipcodes.size(); i++) {
00231         int zipCode = stoi(zipcodes[i]); // Convert ZIP code string to integer.
00232         if (bTreeFile.search(recordBuffer, zipCode) != -1) {
00233             // If found, convert the record buffer back to a Record object and display.
00234             Record record = Record(recordBuffer);
00235             record.display();
00236         } else {
```

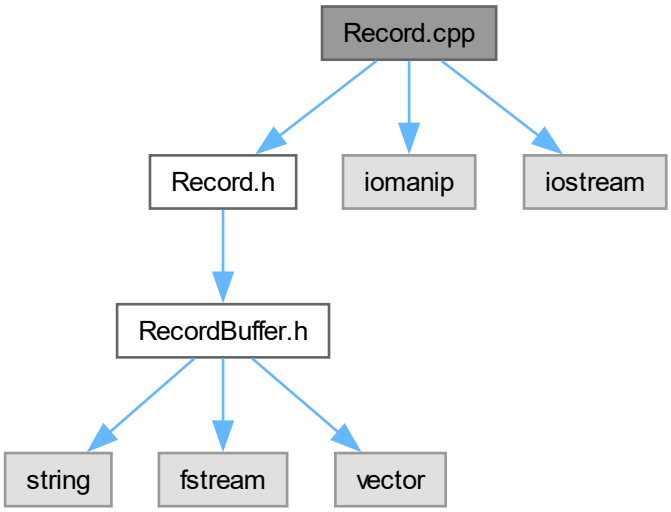
```
00237             // If not found, display a message indicating the ZIP code was not found.
00238             cout « "ZIP Code " « zipcodes[i] « " not found in the B+ tree." « endl;
00239         }
00240     }
00241 }
```

4.23 Record.cpp File Reference

Implementation file for the Record struct.

```
#include "Record.h"
#include <iomanip>
#include <iostream>
```

Include dependency graph for Record.cpp:



4.23.1 Detailed Description

Implementation file for the Record struct.

Author

Team 5

Date

November 18, 2023

Version

2.0

Definition in file Record.cpp.

4.24 Record.cpp

Go to the documentation of this file.

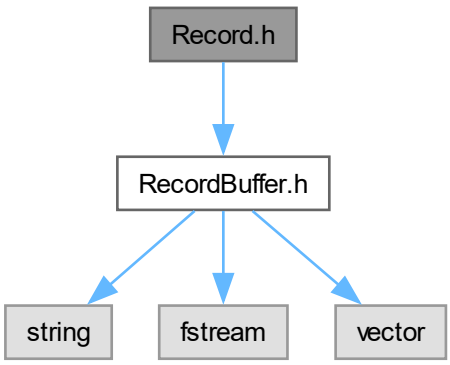
```
00001
00009 #include "Record.h"
00010 #include <iomanip>
00011 #include <iostream>
00012
00013 using namespace std;
00014
00015 Record::Record(RecordBuffer &buffer) {
00016     string lat, lon;
00017     buffer.unpack(ZipCode);
00018     buffer.unpack(PlaceName);
00019     buffer.unpack(State);
00020     buffer.unpack(County);
00021     buffer.unpack(lat);
00022     buffer.unpack(lon);
00023     Lat = stod(lat);
00024     Long = stod(lon);
00025 }
00026
00027 void Record::display() {
00028     cout « left « setw(10) « ZipCode « setw(15) « PlaceName « setw(15) « State « setw(15) « County «
      setw(10) « Lat « setw(10) « Long « endl;
00029 }
```

4.25 Record.h File Reference

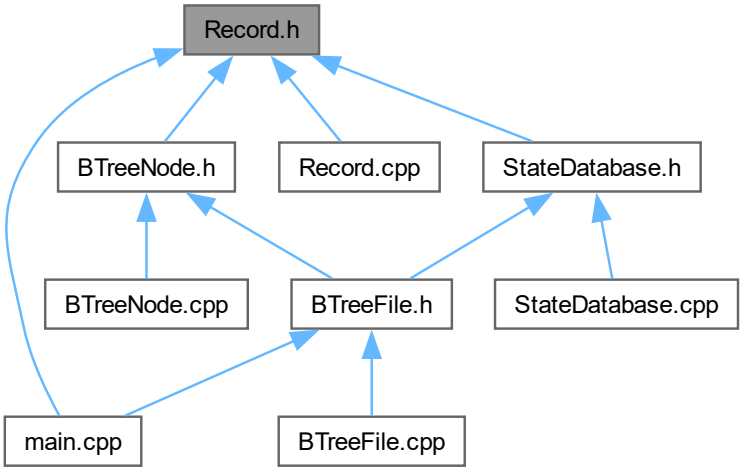
Header file for the Record struct.

`#include "RecordBuffer.h"`

Include dependency graph for Record.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct Record

  A struct for storing information pertaining to a zip code record.

4.25.1 Detailed Description

Header file for the Record struct.

Author

Team 5

Date

November 18, 2023

Version

2.0

Definition in file Record.h.

4.26 Record.h

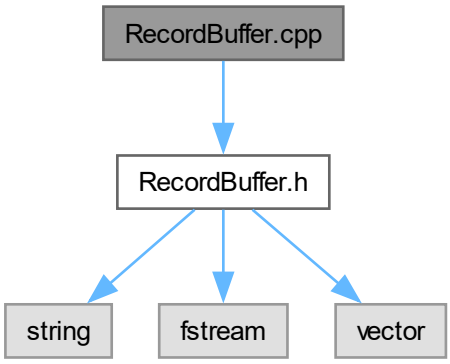
Go to the documentation of this file.

```
00001
00017 #ifndef ZIPCODE_RECORD_H
00018 #define ZIPCODE_RECORD_H
00019
00020 #include "RecordBuffer.h"
00021
00022 struct Record {
00029     Record(RecordBuffer &buffer);
00030
00031
00037     void display();
00038
00039     std::string ZipCode;
00040     std::string PlaceName;
00041     std::string State;
00042     std::string County;
00043     double Lat;
00044     double Long;
00045 };
00046
00047 #endif //ZIPCODE_RECORD_H
00048
```

4.27 RecordBuffer.cpp File Reference

Implementation file for the RecordBuffer class.

`#include "RecordBuffer.h"`
Include dependency graph for RecordBuffer.cpp:



4.27.1 Detailed Description

Implementation file for the RecordBuffer class.

Definition in file RecordBuffer.cpp.

4.28 RecordBuffer.cpp

Go to the documentation of this file.

```
00001
00006 #include "RecordBuffer.h"
00007
00008 using namespace std;
00009
00010 RecordBuffer::RecordBuffer(int maxSize) {
00011     deliminator = ',';
00012     maxBufferSize = maxSize;
00013     nextByte = 0;
00014 }
00015
00016 int RecordBuffer::read(istream &stream, int recaddr) {
00017     // Move to location if needed
00018     if (recaddr != -1) {
00019         stream.clear();
00020         stream.seekg(recaddr, ios::beg);
00021         if (stream.tellg() != recaddr) return -1;
00022     }
00023
00024     // check stream
00025     if (!stream) return -1;
00026
00027     // Get current location in stream
00028     int addr = stream.tellg();
00029
00030     clear();
00031
00032     char rsz[3]; // Make sure it can hold 2 characters and a null character
00033     stream.read(rsz, 2);
00034     rsz[2] = '\0'; // Null-terminate the string
00035     int recordSize = 0;
00036     try {
00037         recordSize = std::stoi(rsz);
00038     }
00039     catch (...) {
00040         return -1;
00041     }
00042
00043     // Make sure record can fit in recordBuffer
00044     if (recordSize > maxBufferSize) return -1;
00045     // read the record into recordBuffer
00046     buffer.resize(recordSize);
00047     stream.read(buffer.data(), recordSize);
00048
00049     // check stream
00050     if (stream.bad()) {
00051         stream.clear();
00052         return -1;
00053     }
00054
00055     return addr;
00056 }
00057
00058 int RecordBuffer::write(ostream &stream, int recaddr) {
00059
00060     // Move to location if needed
00061     if (recaddr != -1) {
00062         stream.seekp(recaddr);
00063         if (stream.tellp() != recaddr) return -1;
00064     }
00065
00066     // Get output location
00067     int addr = stream.tellp();
00068
00069     // Write the recordBuffer (record) size
00070     int recordSize;
00071     recordSize = buffer.size();
00072     stream « to_string(recordSize);
00073
00074     // check stream
00075     if (!stream) return -1;
00076
00077     // Write the recordBuffer
00078     stream.write(buffer.data(), buffer.size());
00079
00080     // check stream
00081     if (!stream) return -1;
00082
00083     return addr;
00084 }
00085
00086 int RecordBuffer::unpack(string &field) {
```

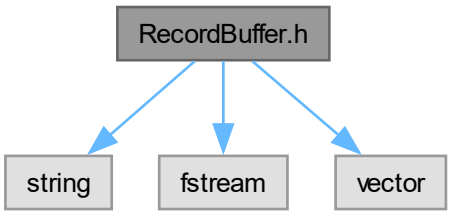
```
00087
00088     int bufferIndex = nextByte;
00089
00090     while (bufferIndex < buffer.size() && buffer[bufferIndex] != deliminator) {
00091         field.push_back(buffer[bufferIndex]);
00092         bufferIndex++;
00093     }
00094
00095     // deliminator not found
00096     if (bufferIndex == buffer.size()) return -1;
00097     nextByte = bufferIndex+1;
00098
00099     return (int)field.length();
00100 }
00101
00102 int RecordBuffer::pack(string field) {
00103     // ensure recordBuffer capacity
00104     if (buffer.size() + field.length() > maxBufferSize) return -1;
00105
00106     // Add field to recordBuffer
00107     buffer.insert(buffer.end(), field.begin(), field.end());
00108
00109     // Add deliminator
00110     buffer.push_back(deliminator);
00111
00112     return (int)field.length();
00113 }
00114
00115 void RecordBuffer::clear() {
00116     buffer.clear();
00117     nextByte = 0;
00118 }
00119
00120 int RecordBuffer::getBufferSize() {
00121     return buffer.size() + to_string(buffer.size()).length();
00122 }
00123
00124 int RecordBuffer::getRecordKey() {
00125     string key;
00126     int i = 0;
00127
00128     while (i < buffer.size() && buffer[i] != deliminator) {
00129         key.push_back(buffer[i]);
00130         i++;
00131     }
00132
00133     return stoi(key);
00134 }
00135
00136
```

4.29 RecordBuffer.h File Reference

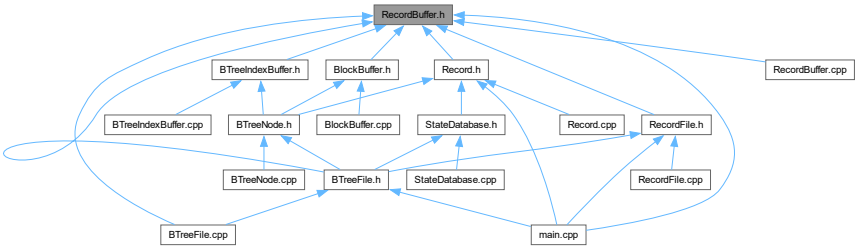
Header file for the RecordBuffer class.

```
#include <string>
#include <fstream>
#include <vector>
```

Include dependency graph for RecordBuffer.h:



This graph shows which files directly or indirectly include this file:



Classes

- class RecordBuffer

  A class for managing a recordBuffer.

4.29.1 Detailed Description

Header file for the RecordBuffer class.

Author

Team 5

Date

November 18, 2023

Version

2.0

Definition in file RecordBuffer.h.

4.30 RecordBuffer.h

Go to the documentation of this file.

```
00001
00016 #ifndef ZIPCODE_BUFFER_H
00017 #define ZIPCODE_BUFFER_H
00018
00019 #include <string>
00020 #include <fstream>
00021 #include <vector>
00022
00023 class RecordBuffer {
00024 public:
00029     RecordBuffer(int maxSize = 1000);
00030
00036     int read(std::istream & stream, int recaddr = -1);
00037
00043     int write(std::ostream & stream, int recaddr = -1);
00044
```

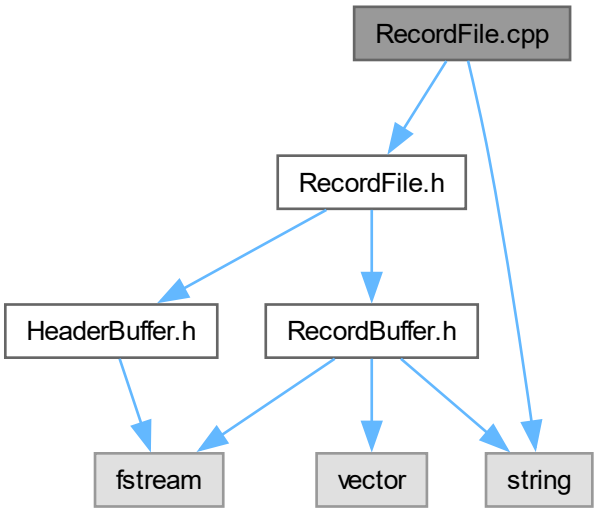
```
00050     int unpack(std::string &field);
00051
00057     int pack(std::string field);
00058
00063     void clear();
00064
00069     int getBufferSize();
00070
00075     int getRecordKey();
00076
00077 private:
00078     std::vector<char> buffer;
00079     int maxBufferSize;
00080     int nextByte;
00081     char deliminator;
00082 };
00083
00084 #endif //ZIPCODE_BUFFER_H
```

4.31 RecordFile.cpp File Reference

Implementation file for the RecordFile class.

```
#include "RecordFile.h"
#include <string>
```

Include dependency graph for RecordFile.cpp:



4.31.1 Detailed Description

Implementation file for the RecordFile class.

Author

Team 5

Date

November 18, 2023

Version

2.0

Definition in file RecordFile.cpp.

4.32 RecordFile.cpp

Go to the documentation of this file.

```
00001
00009 #include "RecordFile.h"
00010 #include <string>
00011
00012 using namespace std;
00013
00014 RecordFile::RecordFile(HeaderBuffer &hbuf) : headerBuffer(hbuf) {
00015
00016 }
00017
00018 RecordFile::~RecordFile() {
00019     closeFile();
00020 }
00021
00022 bool RecordFile::openFile(const std::string& dataFile, std::string &recordFile) {
00023     // Open the length indicated file for writing
00024     string baseFileName = dataFile.substr(0, dataFile.find_last_of('.'));
00025     recordFile = baseFileName + "-LENGTH_INDICATED_RECORDS.txt";
00026     file.open(recordFile.c_str(), std::ios::out | std::ios::binary);
00027     if (!file.is_open()) {
00028         return false;
00029     }
00030
00031     // Add default header to file, then create the data for the file
00032     int headerSize = headerBuffer.writeHeader(file);
00033     if (headerSize == -1) return false;
00034     int count = createLengthIndicatedFile(dataFile);
00035     if (count == -1) return false;
00036
00037     // Rewrite the header with relevant info
00038     headerBuffer.fileType = "length indicated records";
00039     headerBuffer.recordCount = count;
00040     headerSize = headerBuffer.writeHeader(file);
00041
00042     // Close the file for writing and open for reading
00043     file.clear();
00044     file.close();
00045     file.open(recordFile.c_str(), std::ios::in | std::ios::binary);
00046
00047     // Move past the header
00048     file.seekg(headerSize);
00049     file.seekp(headerSize);
00050
00051     if (!file) {
00052         return false;
00053     }
00054
00055     return true;
00056 }
00057
00058 bool RecordFile::closeFile() {
00059     if (file.is_open()) {
00060         headerBuffer.stale = "false";
00061         headerBuffer.writeHeader(file);
00062         file.close();
00063         return true;
00064     }
00065     return false;
00066 }
00067
00068 int RecordFile::read(RecordBuffer &recordBuffer, int recaddr) {
00069     return recordBuffer.read(file, recaddr);
00070 }
```

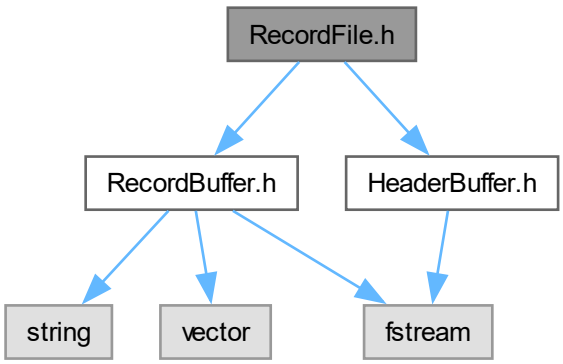
```
00071
00072 int RecordFile::write(RecordBuffer &recordBuffer, int recaddr) {
00073     return recordBuffer.write(file, recaddr);
00074 }
00075
00076 int RecordFile::createLengthIndicatedFile(const string &inputCsvFile) {
00077     string line;
00078     int count = 0;
00079
00080     ifstream infile(inputCsvFile);
00081     if (!infile) {
00082         return -1;
00083     }
00084
00085     // Handle header
00086     if (!infile.eof()) {
00087         getline(infile, line);
00088     }
00089
00090     // Make records length indicated
00091     while (getline(infile, line)) {
00092         // Write the length of the record followed by the record itself
00093         file « line.size() + 1 « line;
00094         count++;
00095
00096         // Check if it's not the last line, then add a newline
00097         if (!infile.eof()) {
00098             file « '\n';
00099         }
00100     }
00101
00102     infile.close();
00103
00104     return count;
00105 }
```

4.33 RecordFile.h File Reference

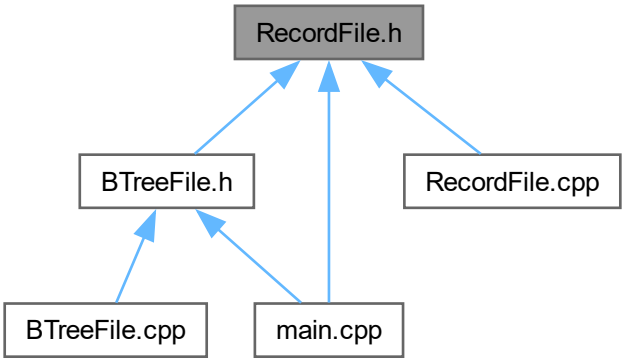
Header file for the RecordFile class.

```
#include "RecordBuffer.h"
#include "HeaderBuffer.h"
```

Include dependency graph for RecordFile.h:



This graph shows which files directly or indirectly include this file:



Classes

- class RecordFile

  A class for managing the file for a recordBuffer.

4.33.1 Detailed Description

Header file for the RecordFile class.

Author

Team 5

Date

November 18, 2023

Version

2.0

Definition in file RecordFile.h.

4.34 RecordFile.h

Go to the documentation of this file.

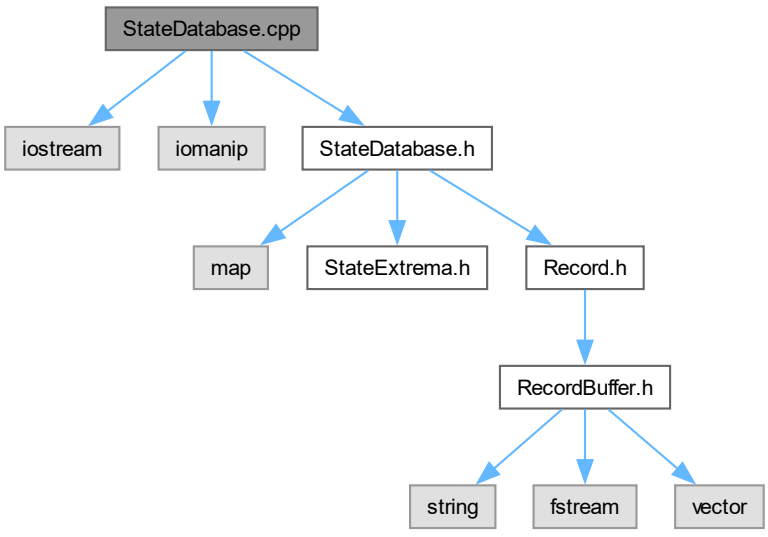
```
00001
00017 #ifndef PROJECT2_PART1_RECORDFILE_H
00018 #define PROJECT2_PART1_RECORDFILE_H
00019
00020 #include "RecordBuffer.h"
00021 #include "HeaderBuffer.h"
00022
00023 class RecordFile
00024 {
00025 public:
00031     RecordFile(HeaderBuffer &hbuf);
00032
00037     ~RecordFile();
00038
00044     bool openFile(const std::string &dataFile, std::string &recordFile);
00045
00052     int read(RecordBuffer &recordBuffer, int recaddr = -1);
00053
00060     int write(RecordBuffer &recordBuffer, int recaddr = -1);
00061
00062 private:
00063     HeaderBuffer &headerBuffer;
00064     std::fstream file;
00065
00070     bool closeFile();
00071
00077     int createLengthIndicatedFile(const std::string &inputCsvFile);
00078 };
00079
00080 #endif // PROJECT2_PART1_RECORDFILE_H
```

4.35 StateDatabase.cpp File Reference

Implementation file for the StateDatabase class.

```
#include <iostream>
#include <iomanip>
#include "StateDatabase.h"
```

Include dependency graph for StateDatabase.cpp:



4.35.1 Detailed Description

Implementation file for the StateDatabase class.

Header file for the StateDatabase class.

Author

Team 5

Date

November 18, 2023

Version

2.0

Definition in file StateDatabase.cpp.

4.36 StateDatabase.cpp

Go to the documentation of this file.

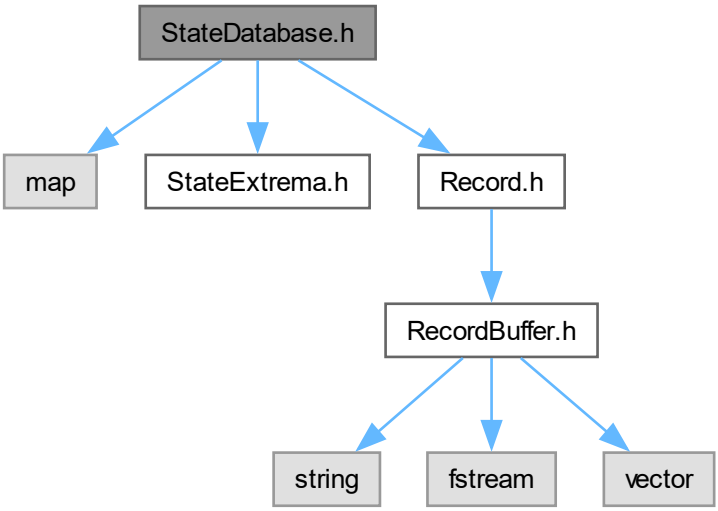
```
00001
00009 #include <iostream>
00010 #include <iomanip>
00011 #include "StateDatabase.h"
00012
00013 using namespace std;
00014
00015 void StateDatabase::processRecord(const Record &record) {
00016     auto it = stateInfoMap.find(record.State);
00017
00018     if (it != stateInfoMap.end()) {
00019         StateExtrema &extrema = it->second;
00020
00021         if (record.Long < extrema.eastLong) {
00022             extrema.eastZip = record.ZipCode;
00023             extrema.eastLong = record.Long;
00024         }
00025
00026         if (record.Long > extrema.westLong) {
00027             extrema.westZip = record.ZipCode;
00028             extrema.westLong = record.Long;
00029         }
00030
00031         if (record.Lat > extrema.northLat) {
00032             extrema.northZip = record.ZipCode;
00033             extrema.northLat = record.Lat;
00034         }
00035
00036         if (record.Lat < extrema.southLat) {
00037             extrema.southZip = record.ZipCode;
00038             extrema.southLat = record.Lat;
00039         }
00040     } else {
00041         StateExtrema extrema;
00042         extrema.eastZip = record.ZipCode;
00043         extrema.eastLong = record.Long;
00044         extrema.westZip = record.ZipCode;
00045         extrema.westLong = record.Long;
00046         extrema.northZip = record.ZipCode;
00047         extrema.northLat = record.Lat;
00048         extrema.southZip = record.ZipCode;
00049         extrema.southLat = record.Lat;
00050         stateInfoMap[record.State] = extrema;
00051     }
00052 }
```

```
00053
00054 void StateDatabase::printStateInfo(std::string state) const {
00055     if (state == "*") {
00056         cout « left « setw(5) « "State " « setw(15) « "Easternmost" « setw(15)
00057              « "Westernmost" « setw(15) « "Northernmost" « setw(15) « "Southernmost" « endl;
00058
00059         for (auto state: stateInfoMap) {
00060             cout « left « setw(6) « state.first
00061                  « setw(15) « state.second.eastZip
00062                  « setw(15) « state.second.westZip
00063                  « setw(15) « state.second.northZip
00064                  « setw(15) « state.second.southZip « endl;
00065         }
00066     } else {
00067         auto it = stateInfoMap.find(state);
00068         if (it != stateInfoMap.end()) {
00069             cout « left « setw(5) « "State " « setw(15) « "Easternmost" « setw(15)
00070                  « "Westernmost" « setw(15) « "Northernmost" « setw(15) « "Southernmost" « endl;
00071
00072             StateExtrema extrema = it->second;
00073             cout « left « setw(6) « it->first
00074                  « setw(15) « extrema.eastZip
00075                  « setw(15) « extrema.westZip
00076                  « setw(15) « extrema.northZip
00077                  « setw(15) « extrema.southZip « endl;
00078         } else {
00079             cout « "STATE: " « state « " not found in database!" « endl;
00080         }
00081     }
00082 }
```

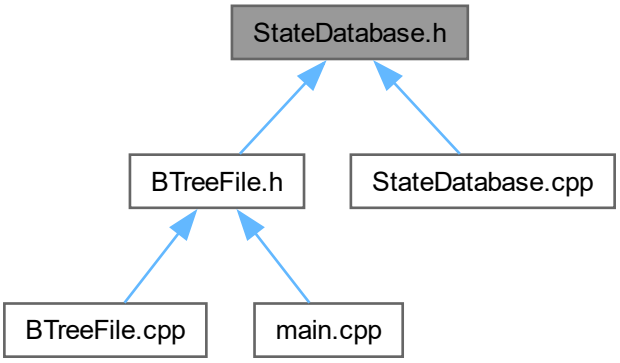
4.37 StateDatabase.h File Reference

```
#include <map>
#include "StateExtrema.h"
#include "Record.h"
```

Include dependency graph for StateDatabase.h:



This graph shows which files directly or indirectly include this file:



Classes

- class StateDatabase

  A class to store a collection of StateExtrema objects.

4.38 StateDatabase.h

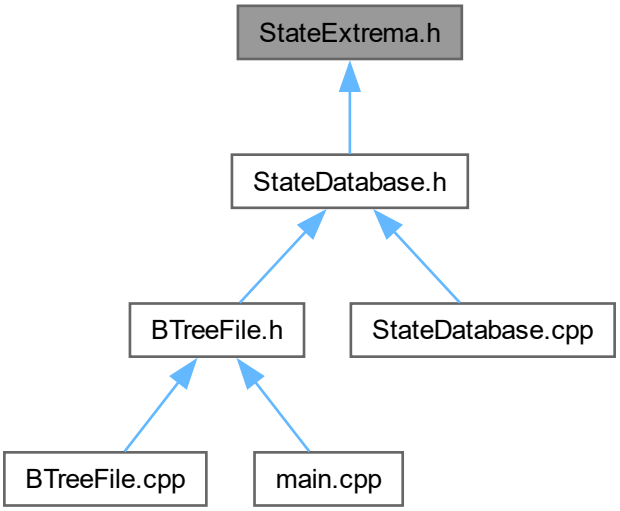
Go to the documentation of this file.

```
00001
00018 #ifndef ZIPCODE_STATEDATABASE_H
00019 #define ZIPCODE_STATEDATABASE_H
00020
00021 #include <map>
00022 #include "StateExtrema.h"
00023 #include "Record.h"
00024
00025 class StateDatabase {
00026 public:
00036     void processRecord(const Record &record);
00037
00044     void printStateInfo(std::string state) const;
00045
00046 private:
00047     std::map<std::string, StateExtrema> stateInfoMap;
00048 };
00049
00050 #endif //ZIPCODE_STATEDATABASE_H
```

4.39 StateExtrema.h File Reference

Header file for the StateExtrema struct.

This graph shows which files directly or indirectly include this file:



Classes

- struct StateExtrema

  A struct for storing information about the extrema of a state.

4.39.1 Detailed Description

Header file for the StateExtrema struct.

Author

Team 5

Date

November 18, 2023

Version

2.0

Definition in file StateExtrema.h.

4.40 StateExtrema.h

Go to the documentation of this file.

```
00001
00016 #ifndef CSCI331_ZIPCODE_STATEEXTREMA_H
00017 #define CSCI331_ZIPCODE_STATEEXTREMA_H
00018
00019 struct StateExtrema {
00020     std::string eastZip;
00021     double eastLong;
00023     std::string westZip;
00024     double westLong;
00026     std::string northZip;
00027     double northLat;
00029     std::string southZip;
00030     double southLat;
00031 };
00032
00033 #endif //CSCI331_ZIPCODE_STATEEXTREMA_H
```

Index

~BTreeFile
    BTreeFile, 19
~RecordFile
    RecordFile, 58

addRecords
    main.cpp, 97

BlockBuffer, 5
    BlockBuffer, 8
    blockSize, 15
    buffer, 15
    clear, 8
    curRBN, 15
    getCurRBN, 8
    getLargestKey, 9
    getNextRBN, 9
    getNumRecords, 9
    getPrevRBN, 9
    isOverFilled, 10
    isUnderFilled, 10
    mergeBuffer, 10
    minimumBlockCapacity, 15
    nextRBN, 15
    numRecords, 15
    pack, 11
    prevRBN, 15
    read, 11
    redistributeBuffer, 11
    removeRecord, 12
    setCurRBN, 12
    setNextRBN, 12
    setNumRecords, 13
    setPrevRBN, 13
    sortBuffer, 13
    splitBuffer, 13
    unpack, 14
    write, 14
blockBuffer
    BTreeNode, 41
BlockBuffer.cpp, 67, 68
BlockBuffer.h, 71, 73
blockCount
    HeaderBuffer, 45
blockSize
    BlockBuffer, 15
    BTreeIndexBuffer, 29
    HeaderBuffer, 45
BTreeFile, 16
    ~BTreeFile, 19
    BTreeFile, 19
    closeFile, 19
    displayExtrema, 19
    displayNode, 20
    displaySequenceSet, 20
    displayTree, 20
    file, 24
    filename, 24
    findLeafNode, 21
    findParentNode, 21
    flushData, 22
    handleMerge, 22
    handleNonRootSplit, 22
    handleRootSplit, 22
    headerBuffer, 25
    height, 25
    insert, 23
    openFile, 23
    order, 25
    remove, 23
    root, 25
    search, 24
BTreeFile.cpp, 74
BTreeFile.h, 79, 80
BTreeIndexBuffer, 26
    blockSize, 29
    BTreeIndexBuffer, 27
    buffer, 29
    clear, 27
    lengthSeparators, 29
    minimumBlockCapacity, 30
    numSeparators, 30
    pack, 28
    read, 28
    unpack, 28
    write, 29
bTreeIndexBuffer
    BTreeNode, 41
BTreeIndexBuffer.cpp, 81, 82
BTreeIndexBuffer.h, 83, 85
BTreeNode, 30
    blockBuffer, 41
    bTreeIndexBuffer, 41
    BTreeNode, 33
    children, 41
    curRBN, 41
    getChildren, 33
    getCurRBN, 33
    getIsLeaf, 34

getLargestKey, 34
getNextChild, 34
getNextRBN, 35
getPrevRBN, 35
insertKeyAndChildren, 35
insertRecord, 35
isLeaf, 41
isOverFilled, 36
isUnderFilled, 36
keys, 42
maxKeys, 42
merge, 36
minKeys, 42
numKeys, 42
print, 37
read, 37
removeKeyAndChildren, 37
removeRecord, 38
retrieveRecord, 38
setCurRBN, 39
setIsLeaf, 39
setNextRBN, 39
setPrevRBN, 40
split, 40
write, 40
BTreeNode.cpp, 85, 86
BTreeNode.h, 89, 90
buffer
BlockBuffer, 15
BTreeIndexBuffer, 29
RecordBuffer, 55

children
BTreeNode, 41
clear
BlockBuffer, 8
BTreeIndexBuffer, 27
RecordBuffer, 53
closeFile
BTreeFile, 19
RecordFile, 59
County
Record, 50
createLengthIndicatedFile
RecordFile, 59
curRBN
BlockBuffer, 15
BTreeNode, 41

deleteRecords
main.cpp, 97
deliminator
RecordBuffer, 56
display
Record, 50
displayExtrema
BTreeFile, 19
displayNode
BTreeFile, 20
displaySequenceSet
BTreeFile, 20
displayTree
BTreeFile, 20

eastLong
StateExtrema, 65
eastZip
StateExtrema, 65

file
BTreeFile, 24
RecordFile, 61
filename
BTreeFile, 24
fileType
HeaderBuffer, 46
findLeafNode
BTreeFile, 21
findParentNode
BTreeFile, 21
flushData
BTreeFile, 22

getBufferSize
RecordBuffer, 53
getChildren
BTreeNode, 33
getCurRBN
BlockBuffer, 8
BTreeNode, 33
getIsLeaf
BTreeNode, 34
getLargestKey
BlockBuffer, 9
BTreeNode, 34
getNextChild
BTreeNode, 34
getNextRBN
BlockBuffer, 9
BTreeNode, 35
getNumRecords
BlockBuffer, 9
getPrevRBN
BlockBuffer, 9
BTreeNode, 35
getRecordKey
RecordBuffer, 54

handleMerge
BTreeFile, 22
handleNonRootSplit
BTreeFile, 22
handleRootSplit
BTreeFile, 22
HeaderBuffer, 43
blockCount, 45
blockSize, 45
fileType, 46

HeaderBuffer, 44
headerRecordSize, 46
minimumBlockCapacity, 46
rbnActive, 46
rbnAvail, 46
readHeader, 45
recordCount, 46
recordFieldCount, 47
recordFieldsType, 47
recordFormat, 47
recordPrimaryKey, 47
recordSizeDigits, 47
recordSizeFormat, 47
stale, 48
version, 48
writeHeader, 45
headerBuffer
BTreeFile, 25
RecordFile, 61
HeaderBuffer.cpp, 91, 92
HeaderBuffer.h, 94, 96
headerRecordSize
HeaderBuffer, 46
height
BTreeFile, 25

insert
BTreeFile, 23
insertKeyAndChildren
BTreeNode, 35
insertRecord
BTreeNode, 35
isLeaf
BTreeNode, 41
isOverFilled
BlockBuffer, 10
BTreeNode, 36
isUnderFilled
BlockBuffer, 10
BTreeNode, 36

keys
BTreeNode, 42

Lat
Record, 50
lengthSeparators
BTreeIndexBuffer, 29
Long
Record, 50

main
main.cpp, 98
main.cpp, 96, 99
addRecords, 97
deleteRecords, 97
main, 98
processCommandLine, 98
searchIndex, 99
maxBufferSize
RecordBuffer, 56
maxKeys
BTreeNode, 42
merge
BTreeNode, 36
mergeBuffer
BlockBuffer, 10
minimumBlockCapacity
BlockBuffer, 15
BTreeIndexBuffer, 30
HeaderBuffer, 46
minKeys
BTreeNode, 42

nextByte
RecordBuffer, 56
nextRBN
BlockBuffer, 15
northLat
StateExtrema, 65
northZip
StateExtrema, 65
numKeys
BTreeNode, 42
numRecords
BlockBuffer, 15
numSeparators
BTreeIndexBuffer, 30

openFile
BTreeFile, 23
RecordFile, 59
order
BTreeFile, 25

pack
BlockBuffer, 11
BTreeIndexBuffer, 28
RecordBuffer, 54
PlaceName
Record, 50
prevRBN
BlockBuffer, 15
print
BTreeNode, 37
printStateInfo
StateDatabase, 62
processCommandLine
main.cpp, 98
processRecord
StateDatabase, 62

rbnActive
HeaderBuffer, 46
rbnAvail
HeaderBuffer, 46
read
BlockBuffer, 11

BTreeIndexBuffer, 28
BTreeNode, 37
RecordBuffer, 54
RecordFile, 60
readHeader
HeaderBuffer, 45
Record, 48
County, 50
display, 50
Lat, 50
Long, 50
PlaceName, 50
Record, 49
State, 51
ZipCode, 51
Record.cpp, 102, 103
Record.h, 103, 105
RecordBuffer, 51
buffer, 55
clear, 53
deliminator, 56
getBufferSize, 53
getRecordKey, 54
maxBufferSize, 56
nextByte, 56
pack, 54
read, 54
RecordBuffer, 53
unpack, 55
write, 55
RecordBuffer.cpp, 105, 106
RecordBuffer.h, 107, 108
recordCount
HeaderBuffer, 46
recordFieldCount
HeaderBuffer, 47
recordFieldsType
HeaderBuffer, 47
RecordFile, 56
∼RecordFile, 58
closeFile, 59
createLengthIndicatedFile, 59
file, 61
headerBuffer, 61
openFile, 59
read, 60
RecordFile, 58
write, 60
RecordFile.cpp, 109, 110
RecordFile.h, 111, 113
recordFormat
HeaderBuffer, 47
recordPrimaryKey
HeaderBuffer, 47
recordSizeDigits
HeaderBuffer, 47
recordSizeFormat
HeaderBuffer, 47
redistributeBuffer
BlockBuffer, 11
remove
BTreeFile, 23
removeKeyAndChildren
BTreeNode, 37
removeRecord
BlockBuffer, 12
BTreeNode, 38
retrieveRecord
BTreeNode, 38
root
BTreeFile, 25

search
BTreeFile, 24
searchIndex
main.cpp, 99
setCurRBN
BlockBuffer, 12
BTreeNode, 39
setIsLeaf
BTreeNode, 39
setNextRBN
BlockBuffer, 12
BTreeNode, 39
setNumRecords
BlockBuffer, 13
setPrevRBN
BlockBuffer, 13
BTreeNode, 40
sortBuffer
BlockBuffer, 13
southLat
StateExtrema, 65
southZip
StateExtrema, 65
split
BTreeNode, 40
splitBuffer
BlockBuffer, 13
stale
HeaderBuffer, 48
State
Record, 51
StateDatabase, 61
printStateInfo, 62
processRecord, 62
stateInfoMap, 63
StateDatabase.cpp, 113, 114
StateDatabase.h, 115, 116
StateExtrema, 63
eastLong, 65
eastZip, 65
northLat, 65
northZip, 65
southLat, 65
southZip, 65
westLong, 65

westZip, 66
StateExtrema.h, 116, 118
stateInfoMap
StateDatabase, 63

unpack
BlockBuffer, 14
BTreeIndexBuffer, 28
RecordBuffer, 55

version
HeaderBuffer, 48

westLong
StateExtrema, 65
westZip
StateExtrema, 66
write
BlockBuffer, 14
BTreeIndexBuffer, 29
BTreeNode, 40
RecordBuffer, 55
RecordFile, 60
writeHeader
HeaderBuffer, 45

ZipCode
Record, 51