Key-Value Storage Engine Documentation

Generated by Doxygen 1.9.1

Chapter 1

Class Index

1.1 Class List

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

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

Class Documentation

3.1 BloomFilter Class Reference

Implements a simple Bloom filter for fast key existence checks.

`#include <bloomfilter.h>`

Public Member Functions

- BloomFilter (int _size=10000)

  Constructor to initialize the Bloom filter with a given size.
- void insert (const std::string &_key)

  Inserts a key into the Bloom filter.
- bool contains (const std::string &_key)

  Checks if a key exists in the Bloom filter.
- void remove (const std::string &_key)

  Removes a key from the Bloom filter (Note: Bloom filters generally do not support removals correctly).

3.1.1 Detailed Description

Implements a simple Bloom filter for fast key existence checks.

3.1.2 Constructor & Destructor Documentation

3.1.2.1 BloomFilter()

```
BloomFilter::BloomFilter (
            int _size = 10000 )  [explicit]
```

Constructor to initialize the Bloom filter with a given size.

Parameters

_size	The size of the Bloom filter (default: 10,000).

```
52 : filter(_size, false), filter_size(_size) {}
```

3.1.3 Member Function Documentation

3.1.3.1 contains()

```
bool BloomFilter::contains (
            const std::string & _key )
```

Checks if a key exists in the Bloom filter.

Parameters

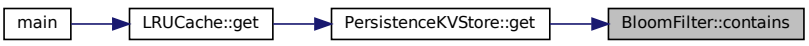
_key	The key to check.

Returns

True if the key is possibly in the filter, false otherwise.

```
60 {
61     return filter[hashKey(_key)];
62 }
```

Here is the caller graph for this function:



3.1.3.2 insert()

```
void BloomFilter::insert (
            const std::string & _key )
```

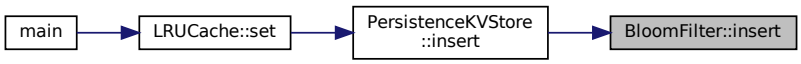
Inserts a key into the Bloom filter.

Parameters

_key	The key to insert.

```
55 {
56     filter[hashKey(_key)] = true;
57 }
```

Here is the caller graph for this function:



3.1.3.3 remove()

```
void BloomFilter::remove (
            const std::string & _key )
```

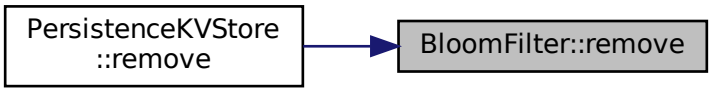
Removes a key from the Bloom filter (Note: Bloom filters generally do not support removals correctly).

Parameters

_key	The key to remove.

```
65 {
66     filter[hashKey(_key)] = false;
67 }
```

Here is the caller graph for this function:



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

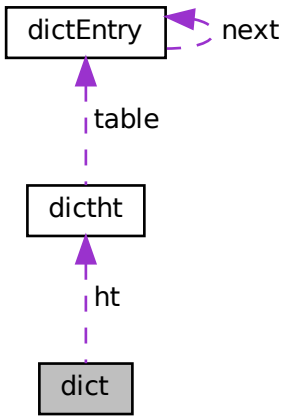
- lib/bloomfilter.h

3.2 dict Struct Reference

The main dictionary structure with two hash tables (for rehashing).

`#include <dict.h>`

Collaboration diagram for dict:



Public Attributes

- dictht ht [2]

  Two hash tables used during rehashing.
- long rehashidx

  Rehashing index (-1 if not rehashing)
- unsigned long iterators

  Number of active iterators.

3.2.1 Detailed Description

The main dictionary structure with two hash tables (for rehashing).

3.2.2 Member Data Documentation

3.2.2.1 ht

`dictht dict::ht[2]`

Two hash tables used during rehashing.

3.2.2.2 iterators

```
unsigned long dict::iterators
```

Number of active iterators.

3.2.2.3 rehashidx

```
long dict::rehashidx
```

Rehashing index (-1 if not rehashing)

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

- lib/dict.h

3.3 Dict Class Reference

A dictionary (hash table) implementation with dynamic resizing and rehashing.

```
#include <dict.h>
```

Public Member Functions

- Dict (std::function< unsigned int(const void *)> hashFunc, std::function< void *(const void *)> keyDup↩ Func, std::function< void *(const void *)> valDupFunc, std::function< int(const void *, const void *)> key↩ CompareFunc, std::function< void(void *)> keyDestructorFunc, std::function< void(void *)> valDestructor↩ Func)

  Constructor for the dictionary.
- ~Dict ()

  Destructor for the dictionary.
- void enableResize (bool enable)

  Enables or disables automatic resizing.
- int add (void *key, void *val)

  Adds a key-value pair to the dictionary with automatic rehashing.
- int replace (void *key, void *val)

  Replaces a key's value in the dictionary.
- int remove (const void *key)

  Removes a key from the dictionary with incremental rehashing.
- void * find (const void *key)

  Finds a key in the dictionary.
- int rehash (int n)

  Performs a rehash operation.
- bool isRehashing ()

  Checks if rehashing is in progress.
- size_t get_size_of_dict ()

  Retrieves the total memory usage of the dictionary for keys, values.
- int size ()

  Retrieves the total no of keys in the dictionary.

3.3.1 Detailed Description

A dictionary (hash table) implementation with dynamic resizing and rehashing.

3.3.2 Constructor & Destructor Documentation

3.3.2.1 Dict()

```
Dict::Dict (
            std::function< unsigned int(const void *)> hashFunc,
            std::function< void *(const void *)> keyDupFunc,
            std::function< void *(const void *)> valDupFunc,
            std::function< int(const void *, const void *)> keyCompareFunc,
            std::function< void(void *)> keyDestructorFunc,
            std::function< void(void *)> valDestructorFunc )  [inline]
```

Constructor for the dictionary.

Parameters

hashFunc	Hash function.
keyDupFunc	Key duplication function.
valDupFunc	Value duplication function.
keyCompareFunc	Key comparison function.
keyDestructorFunc	Key destructor function.
valDestructorFunc	Value destructor function.

```
123         : hashFunction(hashFunc), keyDup(keyDupFunc), valDup(valDupFunc),
124           keyCompare(keyCompareFunc), keyDestructor(keyDestructorFunc),
125           valDestructor(valDestructorFunc)
126     {
127         _dictInit(&d);
128     }
```

3.3.2.2 ~Dict()

```
Dict::~Dict ( )  [inline]
```

Destructor for the dictionary.

```
134     {
135         _dictClear(&d);
136     }
```

3.3.3 Member Function Documentation

3.3.3.1 add()

```
int Dict::add (
            void * key,
            void * val )  [inline]
```

Adds a key-value pair to the dictionary with automatic rehashing.

Parameters

key	Key pointer.
val	Value pointer.

Returns

0 on success, 1 on failure.

```
154     {
155         // Check if resize is needed before adding
156         if (_dictShouldResize())
157         {
158             // Calculate new size based on current usage
159             unsigned long newSize = d.ht[0].used * 2;
160             _dictExpand(&d, newSize);
161         }
162 
163         // Perform a rehash step if rehashing is in progress
164         if (dictIsRehashing(&d))
165             _dictRehashStep();
166 
167         return dictAdd(&d, key, val);
168     }
```

3.3.3.2 enableResize()

```
void Dict::enableResize (
            bool enable )  [inline]
```

Enables or disables automatic resizing.

Parameters

enable	True to enable resizing, false to disable.

```
143     {
144         dict_can_resize = enable;
145     }
```

3.3.3.3 find()

```
void* Dict::find (
            const void * key )  [inline]
```

Finds a key in the dictionary.

Parameters

key	Key pointer.

Returns

Pointer to the value if found, nullptr otherwise.

```
201     {
202         // Perform a rehash step if rehashing is in progress
203         if (dictIsRehashing(&d))
204             _dictRehashStep();
205 
206         dictEntry *he = dictFind(&d, key);
207         return he ? he->v.val : nullptr;
208     }
```

3.3.3.4 get_size_of_dict()

```
size_t Dict::get_size_of_dict ( )  [inline]
```

Retrieves the total memory usage of the dictionary for keys, values.

Returns

size_t The total size of the dictionary in bytes.

```
234     {
235         return total_size_of_dict;
236     }
```

3.3.3.5 isRehashing()

```
bool Dict::isRehashing ( )  [inline]
```

Checks if rehashing is in progress.

Returns

True if rehashing, false otherwise.

```
225     {
226         return dictIsRehashing(&d);
227     }
```

3.3.3.6 rehash()

```
int Dict::rehash (
            int n )  [inline]
```

Performs a rehash operation.

Parameters

n	Number of steps to rehash.

Returns

0 on completion, 1 if rehashing is ongoing.

```
216     {
217         return dictRehash(&d, n);
218     }
```

3.3.3.7 remove()

```
int Dict::remove (
            const void * key )  [inline]
```

Removes a key from the dictionary with incremental rehashing.

Parameters

key	Key pointer.

Returns

0 on success, 1 if key not found.

```
187     {
188         // Perform a rehash step if rehashing is in progress
189         if (dictIsRehashing(&d))
190             _dictRehashStep();
191 
192         return dictDelete(&d, key);
193     }
```

3.3.3.8 replace()

```
int Dict::replace (
            void * key,
            void * val )  [inline]
```

Replaces a key's value in the dictionary.

Parameters

key	Key pointer.
val	Value pointer.

Returns

0 if key already exists and value is replaced, 1 if key is newly added.

```
177     {
178         return dictReplace(&d, key, val);
179     }
```

3.3.3.9 size()

```
int Dict::size ( )  [inline]
```

Retrieves the total no of keys in the dictionary.

Returns

int To total no of keys in the dict

```
244     {
245         return d.ht[0].used + d.ht[1].used;
246     }
```

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

- lib/dict.h

3.4 dictEntry Struct Reference

Represents an entry in the hash table.

`#include <dict.h>`

Collaboration diagram for dictEntry:

Public Attributes

- void * key
  Key pointer.
- union {
  void * val
  Value pointer.
  uint64_t u64
  Unsigned 64-bit integer.
  int64_t s64
  Signed 64-bit integer.
  double d
  Double precision floating point.
  } v
- dictEntry * next
  Pointer to the next entry (for chaining)

3.4.1 Detailed Description

Represents an entry in the hash table.

3.4.2 Member Data Documentation

3.4.2.1 d

```
double dictEntry::d
```

Double precision floating point.

3.4.2.2 key

```
void* dictEntry::key
```

Key pointer.

3.4.2.3 next

```
dictEntry* dictEntry::next
```

Pointer to the next entry (for chaining)

3.4.2.4 s64

```
int64_t dictEntry::s64
```

Signed 64-bit integer.

3.4.2.5 u64

```
uint64_t dictEntry::u64
```

Unsigned 64-bit integer.

3.4.2.6

```
union { ... } dictEntry::v
```

3.4.2.7 val

```
void* dictEntry::val
```

Value pointer.

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

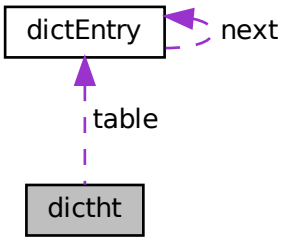
- lib/dict.h

3.5 dictht Struct Reference

Hash table structure containing entries.

```
#include <dict.h>
```

Collaboration diagram for dictht:



Public Attributes

- dictEntry ** table
  Hash table array.
- unsigned long size
  Size of the table.
- unsigned long sizemask
  Mask for indexing.
- unsigned long used
  Number of elements used.

3.5.1 Detailed Description

Hash table structure containing entries.

3.5.2 Member Data Documentation

3.5.2.1 size

```
unsigned long dictht::size
```

Size of the table.

3.5.2.2 sizemask

```
unsigned long dictht::sizemask
```

Mask for indexing.

3.5.2.3 table

```
dictEntry** dictht::table
```

Hash table array.

3.5.2.4 used

```
unsigned long dictht::used
```

Number of elements used.

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

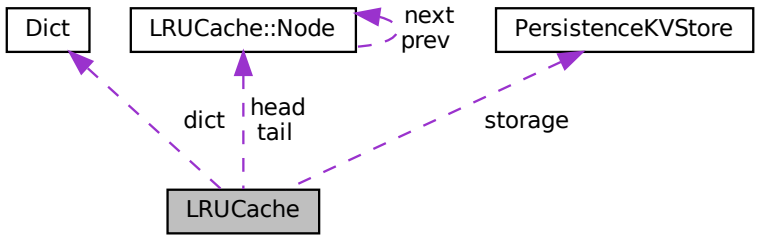
- lib/dict.h

3.6 LRUCache Class Reference

Implements a Least Recently Used (LRU) cache with memory constraints.

`#include <lru_cache.h>`

Collaboration diagram for LRUCache:



Classes

- struct Node

  Represents a node in the doubly linked list for the LRU cache.

Public Member Functions

- LRUCache (size_t max_mem=1024 ∗1024 ∗1024)

  Constructs a new LRU Cache with the specified memory limit.
- ∼LRUCache ()

  Destroys the LRU Cache and frees all allocated memory.
- std::string get (const void ∗key)

  Retrieves the value for a given key.
- void printList ()

  Prints the current state of the cache for debugging.
- void set (void ∗key, void ∗value)

  Adds or updates a key-value pair in the cache.
- int del (const void ∗key)

  Deletes a key-value pair from the cache.
- size_t memory_usage ()

  Gets the current memory usage of the cache.
- size_t max_memory ()

  Gets the maximum memory limit of the cache.
- size_t size ()

  Gets the number of items in the cache.

Public Attributes

- Dict dict
- Node * head
- Node * tail
- size_t current_memory_usage
- size_t max_memory_bytes
- PersistenceKVStore storage
- std::string value

3.6.1 Detailed Description

Implements a Least Recently Used (LRU) cache with memory constraints.

This class provides a memory-constrained LRU cache implementation that evicts least recently used items when memory limits are exceeded. It uses a doubly linked list for tracking usage order and a dictionary for O(1) lookups.

3.6.2 Constructor & Destructor Documentation

3.6.2.1 LRUCache()

```
LRUCache::LRUCache (
            size_t max_mem = 1024 * 1024 * 1024 )  [inline]
```

Constructs a new LRU Cache with the specified memory limit.

Parameters

max_mem	Maximum memory limit in bytes

```
61                                                  : storage("./blink"), dict(stringHash, nullptr,
      nullptr, stringCompare, freeKey, freeValue), current_memory_usage(0)
62      {
63          max_memory_bytes = max_mem;
64          head = new Node(strdup("-1"), strdup("-1"));
65          tail = new Node(strdup("-1"), strdup("-1"));
66          head->next = tail;
67          tail->prev = head;
68      }
```

3.6.2.2 ~LRUCache()

```
LRUCache::~LRUCache ( )  [inline]
```

Destroys the LRU Cache and frees all allocated memory.

```
74      {
75          freeNode(head);
76          freeNode(tail);
77      }
```

3.6.3 Member Function Documentation

3.6.3.1 del()

```
int LRUCache::del (
            const void * key )  [inline]
```

Deletes a key-value pair from the cache.

Parameters

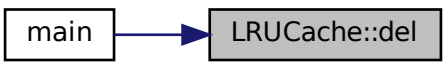
key	The key to delete

Returns

int 0 if successful, 1 if the key was not found

```
174     {
175         Node *retrievedValue = static_cast<Node *>(dict.find(key));
176 
177         if (retrievedValue)
178         {
179             remove(retrievedValue);
180             current_memory_usage -= getSize((char *)retrievedValue->key) + getNodeSize(retrievedValue);
181             dict.remove(retrievedValue->key);
182             return 0;
183         }
184         return 1;
185     }
```

Here is the caller graph for this function:



3.6.3.2 get()

```
std::string LRUCache::get (
            const void * key )  [inline]
```

Retrieves the value for a given key.

If the key exists, it moves the corresponding node to the front of the list to mark it as most recently used.

Parameters

key	The key to look up

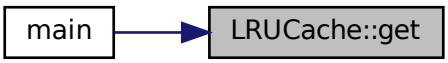
Returns

std::string The value associated with the key, or "-1" if not found

```
89      {
90          Node *retrievedValue = static_cast<Node *>(dict.find(key));
91          if (!retrievedValue){
92              std::string value;
93              if(storage.get(std::string((char *)key), value)){
94                  void *key1 = (void *)key;
95                  void* value1 = (void *) strdup(value.c_str());
96                  Node *node = new Node(key1, value1);
97                  dict.add(key1, node);
98                  add(node);
99                  current_memory_usage += getSize((char *)key) + getNodeSize(node);
100                  return value;
101              };
102              return "-1";
103          }
104
105
106          remove(retrievedValue);
107          add(retrievedValue);
108          return static_cast<char *>(retrievedValue->value);
109      }
```

Here is the call graph for this function:

Here is the caller graph for this function:



3.6.3.3 max_memory()

```
size_t LRUCache::max_memory ( )  [inline]
```

Gets the maximum memory limit of the cache.

Returns

size_t Maximum memory limit in bytes

```
203     {
204         return max_memory_bytes;
205     }
```

3.6.3.4 memory_usage()

```
size_t LRUCache::memory_usage ( )  [inline]
```

Gets the current memory usage of the cache.

Returns

size_t Current memory usage in bytes

```
193     {
194         return current_memory_usage;
195     }
```

3.6.3.5 printList()

```
void LRUCache::printList ( )  [inline]
```

Prints the current state of the cache for debugging.

```
115     {
116         Node *curr = head->next;
117         std::cout « "Cache state: ";
118         while (curr != tail)
119         {
120             std::cout « "[" « static_cast<char *>(curr->key) « ":" « static_cast<char *>(curr->value) «
        "] ";
121             curr = curr->next;
122         }
123         std::cout « std::endl;
124     }
```

3.6.3.6 set()

```
void LRUCache::set (
            void * key,
            void * value )  [inline]
```

Adds or updates a key-value pair in the cache.

If adding the new item exceeds the memory capacity, the least recently used items will be evicted until the memory usage is within limits.

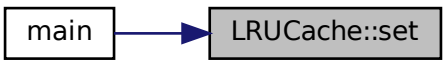
Parameters

key	Pointer to the key
value	Pointer to the value

```
136     {
137         Node *retrievedValue = static_cast<Node *>(dict.find(key));
138         Node *node = new Node(key, value);
139 
140         if (retrievedValue)
141         {
142             current_memory_usage += getNodeSize(node) - getNodeSize(retrievedValue);
143             remove(retrievedValue);
144             dict.replace(key, node);
145             add(node);
146         }
147         else
148         {
149             dict.add(key, node);
150             add(node);
151             current_memory_usage += getSize((char *)key) + getNodeSize(node);
152         }
153 
154         if (current_memory_usage >= max_memory_bytes)
155         {
156             Node *nodeToDelete = tail->prev;
157             storage.insert(std::string(strdup((char *)nodeToDelete->key)), std::string(strdup((char
    *)nodeToDelete->value)));
158             remove(nodeToDelete);
159             std::cout << current_memory_usage << std::endl;
160             current_memory_usage -= getSize((char *)nodeToDelete->key) + getNodeSize(nodeToDelete);
161             dict.remove(nodeToDelete->key);
162             std::cout << "Eviction happen" << std::endl;
163             std::cout << current_memory_usage << std::endl;
164         }
165     }
```

Here is the call graph for this function:

Here is the caller graph for this function:



3.6.3.7 size()

```
size_t LRUCache::size ( )  [inline]
```

Gets the number of items in the cache.

Returns

size_t Number of items in the cache

```
213     {
214         return dict.size();
215     }
```

3.6.4 Member Data Documentation

3.6.4.1 current_memory_usage

```
size_t LRUCache::current_memory_usage
```

Current memory usage in bytes

3.6.4.2 dict

```
Dict LRUCache::dict
```

Dictionary for O(1) lookups

3.6.4.3 head

```
Node* LRUCache::head
```

Head of the doubly linked list

3.6.4.4 max_memory_bytes

```
size_t LRUCache::max_memory_bytes
```

Maximum memory limit in bytes

3.6.4.5 storage

```
PersistenceKVStore LRUCache::storage
```

3.6.4.6 tail

```
Node* LRUCache::tail
```

Tail of the doubly linked list

3.6.4.7 value

```
std::string LRUCache::value
```

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

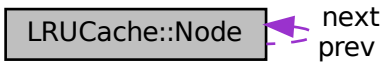
- lib/lru_cache.h

3.7 LRUCache::Node Struct Reference

Represents a node in the doubly linked list for the LRU cache.

```
#include <lru_cache.h>
```

Collaboration diagram for LRUCache::Node:



Public Member Functions

- Node (void *k, void *v)

  Constructs a new Node with the given key and value.

Public Attributes

- void * key
- void * value
- Node * next
- Node * prev

3.7.1 Detailed Description

Represents a node in the doubly linked list for the LRU cache.

Each node contains a key-value pair and pointers to the next and previous nodes.

3.7.2 Constructor & Destructor Documentation

3.7.2.1 Node()

```
LRUCache::Node::Node (
            void * k,
            void * v )  [inline]
```

Constructs a new Node with the given key and value.

Parameters

k	Pointer to the key
v	Pointer to the value

```
40          {
41              key = k;
42              value = v;
43              next = nullptr;
44              prev = nullptr;
45          }
```

3.7.3 Member Data Documentation

3.7.3.1 key

```
void* LRUCache::Node::key
```

Pointer to the key

3.7.3.2 next

```
Node* LRUCache::Node::next
```

Pointer to the next node in the list

3.7.3.3 prev

```
Node* LRUCache::Node::prev
```

Pointer to the previous node in the list

3.7.3.4 value

```
void* LRUCache::Node::value
```

Pointer to the value

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

- lib/lru_cache.h

3.8 PersistenceKVStore Class Reference

A persistent key-value store with background rewriting and indexing.

```
#include <persistence_kv_store.h>
```

Public Member Functions

- PersistenceKVStore (const std::string _dbname, int _bloom_filter_size=10000, int rewrite_interval=5000)

  Constructor: Initializes the key-value store, loads the index, and starts the rewrite scheduler.
- ~PersistenceKVStore ()

  Destructor: Ensures background rewriting stops and closes the file.
- void insert (const std::string &_key, const std::string &_value)

  Inserts a key-value pair into the store.
- bool get (const std::string &_key, std::string &_value)

  Retrieves the value for a given key.
- void remove (const std::string &_key)

  Removes a key from the store.
- void remove_db ()

  Removes the database file.

3.8.1 Detailed Description

A persistent key-value store with background rewriting and indexing.

3.8.2 Constructor & Destructor Documentation

3.8.2.1 PersistenceKVStore()

```
PersistenceKVStore::PersistenceKVStore (
            const std::string _dbname,
            int _bloom_filter_size = 10000,
            int rewrite_interval = 5000 )
```

Constructor: Initializes the key-value store, loads the index, and starts the rewrite scheduler.

Parameters

_dbname	The database name.
_bloom_filter_size	The size of BloomFilter.
_rewrite_interval	Interval for background rewrite in milliseconds (default: 5000).

```
93     : bloomFilter(_bloom_filter_size)
94 {
95     filename = _dbname + ".txt";
96     tempfilename = _dbname + ".temp.txt";
97     rewrite_interval_ms = _rewrite_interval;
98     stopRewrite.store(false);
99     dataFile.open(filename, std::ios::in | std::ios::out | std::ios::binary);
100     index = new Trie();
101 
102     if (!dataFile)
103     {
104         dataFile.open(filename, std::ios::out | std::ios::binary);
105         dataFile.close();
106         dataFile.open(filename, std::ios::in | std::ios::out | std::ios::binary);
107     }
108 
109     syncIndex();
```

```
110     rewriteThread = std::thread(&PersistenceKVStore::startRewriteScheduler, this);
111 }
```

3.8.2.2 ~PersistenceKVStore()

```
PersistenceKVStore::~PersistenceKVStore ( )
```

Destructor: Ensures background rewriting stops and closes the file.

```
114 {
115     dataFile.clear();
116     stopRewrite.store(true);
117     if (rewriteThread.joinable())
118     {
119         rewriteThread.join();
120     }
121     dataFile.close();
122 }
```

3.8.3 Member Function Documentation

3.8.3.1 get()

```
bool PersistenceKVStore::get (
            const std::string & _key,
            std::string & _value )
```

Retrieves the value for a given key.

Parameters

_key	The key to search for.
_value	Reference to store the retrieved value.

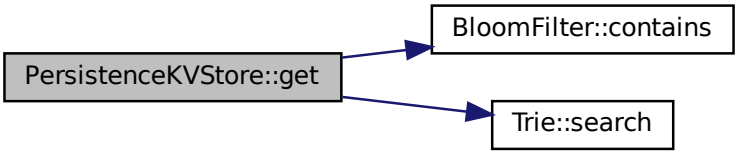
Returns

True if the key exists, false otherwise.

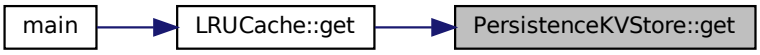
```
143 {
144     std::lock_guard<std::mutex> lock(mtx_index);
145     if (!bloomFilter.contains(_key))
146         return false;
147 
148     long offset = index->search(_key);
149     if (offset == -1)
150         return false;
151 
152     dataFile.clear();
153     dataFile.seekg(offset, std::ios::beg);
154 
155     if (!dataFile)
156         return false;
157 
158     std::string storedKey;
159     dataFile » storedKey;
160 
161     if (storedKey != _key)
162         return false;
```

```
163
164     std::getline(dataFile » std::ws, _value);
165
166     return !_value.empty();
167 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



3.8.3.2 insert()

```
void PersistenceKVStore::insert (
            const std::string & _key,
            const std::string & _value )
```

Inserts a key-value pair into the store.

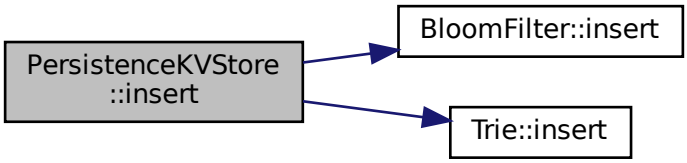
Parameters

_key	The key to insert.
_value	The corresponding value.

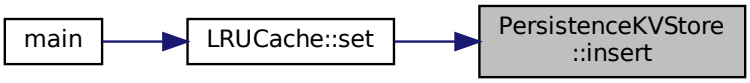
```
125 {
126     dataFile.clear();
127     dataFile.seekp(0, std::ios::end);
128     long offset = dataFile.tellp();
129     if (offset == -1)
130     {
131         std::cerr « "Error: tellp() returned -1" « std::endl;
132         return;
133     }
134
135     dataFile « _key « " " « _value « std::endl;
136     dataFile.flush();
```

```
137     std::lock_guard<std::mutex> lock(mtx_index);
138     index->insert(_key, offset);
139     bloomFilter.insert(_key);
140 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



3.8.3.3 remove()

```
void PersistenceKVStore::remove (
            const std::string & _key )
```

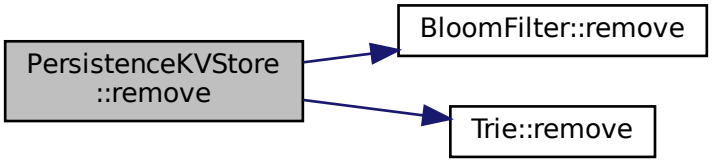
Removes a key from the store.

Parameters

_key	The key to remove.

```
170 {
171     std::lock_guard<std::mutex> lock(mtx_index);
172     index->remove(_key);
173     bloomFilter.remove(_key);
174 }
```

Here is the call graph for this function:



3.8.3.4 remove_db()

```
void PersistenceKVStore::remove_db ( )
```

Removes the database file.

```
177 {
178     std::remove(filename.c_str());
179 }
```

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

- lib/persistence_kv_store.h

3.9 Trie Class Reference

Trie data structure for efficient key lookup.

```
#include <tire.h>
```

Public Member Functions

- Trie ()

  Constructs a new Trie object.
- ~Trie ()

  Destroys the Trie object and deallocates memory.
- void insert (const std::string &key, long offset)

  Inserts a key with a file offset into the Trie.
- long search (const std::string &key)

  Searches for a key in the Trie.
- void remove (const std::string &key)

  Marks a key as deleted in the Trie.
- bool isDeleted (const std::string &key)

  Checks if a key is marked as deleted.

3.9.1 Detailed Description

Trie data structure for efficient key lookup.

3.9.2 Constructor & Destructor Documentation

3.9.2.1 Trie()

```
Trie::Trie ( )  [inline]
```

Constructs a new Trie object.

```
40 { root = new TrieNode(); }
```

3.9.2.2 ~Trie()

```
Trie::~Trie ( )  [inline]
```

Destroys the Trie object and deallocates memory.

```
45 { deleteTrie(root); }
```

3.9.3 Member Function Documentation

3.9.3.1 insert()

```
void Trie::insert (
            const std::string & key,
            long offset )  [inline]
```

Inserts a key with a file offset into the Trie.

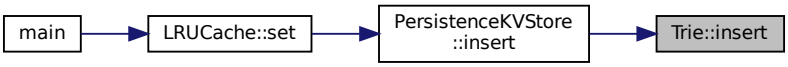
Parameters

key	The key to insert.
offset	The file offset associated with the key.

```
53      {
54          TrieNode *node = root;
55          for (char ch : key)
56          {
57              if (!node->children.count(ch))
58                  node->children[ch] = new TrieNode();
59              node = node->children[ch];
60          }
61          node->file_offset = offset;
62          node->isDeleted = false;
```

```
63      }
```

Here is the caller graph for this function:



3.9.3.2 isDeleted()

```
bool Trie::isDeleted (
            const std::string & key )  [inline]
```

Checks if a key is marked as deleted.

Parameters

key	The key to check.

Returns

True if the key is deleted, false otherwise.

```
104     {
105         TrieNode *node = root;
106         for (char ch : key)
107         {
108             if (!node->children.count(ch))
109                 return false;
110             node = node->children[ch];
111         }
112         return node->isDeleted;
113     }
```

3.9.3.3 remove()

```
void Trie::remove (
            const std::string & key )  [inline]
```

Marks a key as deleted in the Trie.

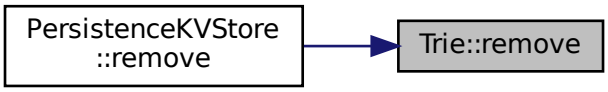
Parameters

key	The key to remove.

```
87      {
88          TrieNode *node = root;
```

```
89          for (char ch : key)
90          {
91              if (!node->children.count(ch))
92                  return;
93              node = node->children[ch];
94          }
95          node->isDeleted = true;
96      }
```

Here is the caller graph for this function:



3.9.3.4 search()

```
long Trie::search (
            const std::string & key )  [inline]
```

Searches for a key in the Trie.

Parameters

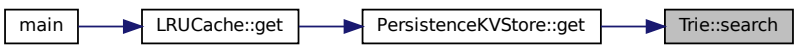
key	The key to search for.

Returns

The file offset if found, otherwise -1.

```
71      {
72          TrieNode *node = root;
73          for (char ch : key)
74          {
75              if (!node->children.count(ch))
76                  return -1;
77              node = node->children[ch];
78          }
79          return node->isDeleted ? -1 : node->file_offset;
80      }
```

Here is the caller graph for this function:



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

- lib/tire.h

3.10 TrieNode Class Reference

Represents a node in the Trie.

`#include <tire.h>`

Public Attributes

- std::unordered_map< char, TrieNode * > children
- long file_offset = -1
- bool isDeleted = false

3.10.1 Detailed Description

Represents a node in the Trie.

3.10.2 Member Data Documentation

3.10.2.1 children

```
std::unordered_map<char, TrieNode *> TrieNode::children
```

Child nodes of the Trie.

3.10.2.2 file_offset

```
long TrieNode::file_offset = -1
```

Offset of the file where the key is stored.

3.10.2.3 isDeleted

```
bool TrieNode::isDeleted = false
```

Flag indicating whether the key is deleted.

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

- lib/tire.h

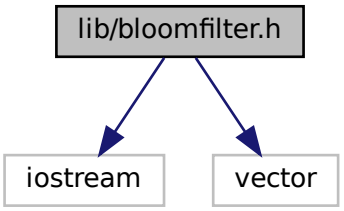
Chapter 4

File Documentation

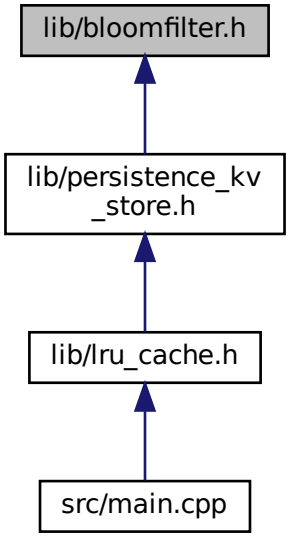
4.1 lib/bloomfilter.h File Reference

```
#include <iostream>
#include <vector>
```

Include dependency graph for bloomfilter.h:



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



Classes

- class BloomFilter

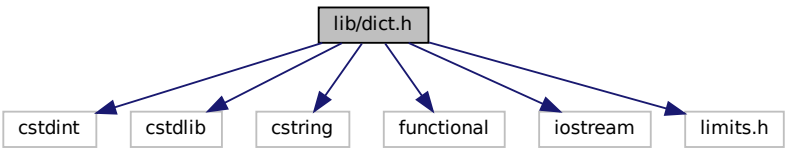
  Implements a simple Bloom filter for fast key existence checks.

4.2 lib/dict.h File Reference

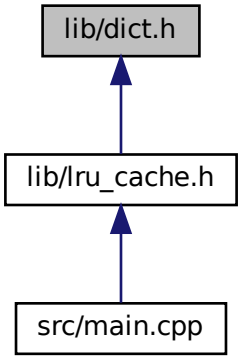
Implementation of a dictionary (hash table) with rehashing support.

```
#include <cstdint>
#include <cstdlib>
#include <cstring>
#include <functional>
#include <iostream>
#include <limits.h>
```

Include dependency graph for dict.h:



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



Classes

- struct dictEntry

  Represents an entry in the hash table.
- struct dictht

  Hash table structure containing entries.
- struct dict

  The main dictionary structure with two hash tables (for rehashing).
- class Dict

  A dictionary (hash table) implementation with dynamic resizing and rehashing.

Functions

- unsigned int stringHash (const void ∗key)

  Hash function for C-style strings.
- int stringCompare (const void ∗key1, const void ∗key2)

  Compares two C-style string keys.
- void ∗ stringDup (const void ∗key)

  Duplicates a C-style string key.
- void freeString (void ∗ptr)

  Frees a dynamically allocated C-style string.

4.2.1 Detailed Description

Implementation of a dictionary (hash table) with rehashing support.

4.2.2 Function Documentation

4.2.2.1 freeString()

```
void freeString (
            void * ptr )
```

Frees a dynamically allocated C-style string.

This function releases the memory allocated for a string key or value.

Parameters

ptr	Pointer to the string to be freed.

```
649 {
650     free(ptr);
651 }
```

4.2.2.2 stringCompare()

```
int stringCompare (
            const void * key1,
            const void * key2 )
```

Compares two C-style string keys.

This function compares two string keys using `strcmp` and returns whether they are equal.

Parameters

key1	Pointer to the first string key.
key2	Pointer to the second string key.

Returns

int Returns 1 if keys are equal, 0 otherwise.

```
623 {
624     return strcmp(static_cast<const char *>(key1), static_cast<const char *>(key2)) == 0;
625 }
```

4.2.2.3 stringDup()

```
void* stringDup (
            const void * key )
```

Duplicates a C-style string key.

This function creates a copy of the given string key using `strdup`. The caller is responsible for freeing the allocated memory.

Parameters

key	Pointer to the original string key.

Returns

void* Pointer to the duplicated string.

```
637 {
638     return strdup(static_cast<const char *>(key));
639 }
```

4.2.2.4 stringHash()

```
unsigned int stringHash (
            const void * key )
```

Hash function for C-style strings.

This function computes a hash value for a given string using the DJB2 algorithm. It iterates through the characters and accumulates a hash value.

Parameters

key	Pointer to the C-style string key.

Returns

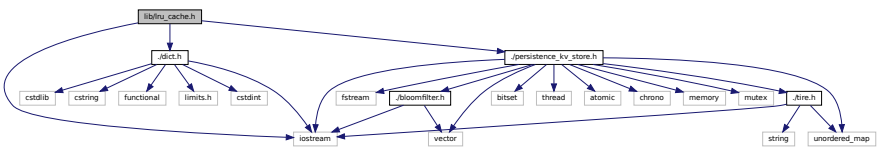
unsigned int The computed hash value.

```
603 {
604     const char *str = static_cast<const char *>(key);
605     unsigned int hash = 5381;
606     int c;
607     while ((c = *str++))
608         hash = ((hash << 5) + hash) + c; // hash * 33 + c
609     return hash;
610 }
```

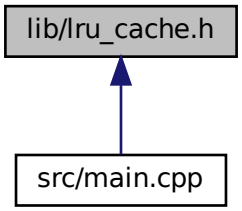
4.3 lib/lru_cache.h File Reference

```
#include <iostream>
#include "./dict.h"
#include "./persistence_kv_store.h"
```

Include dependency graph for lru_cache.h:



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



Classes

- class LRUCache

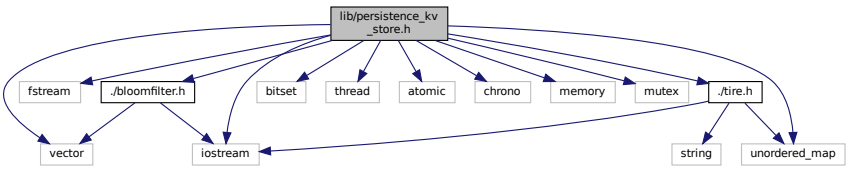
  Implements a Least Recently Used (LRU) cache with memory constraints.
- struct LRUCache::Node

  Represents a node in the doubly linked list for the LRU cache.

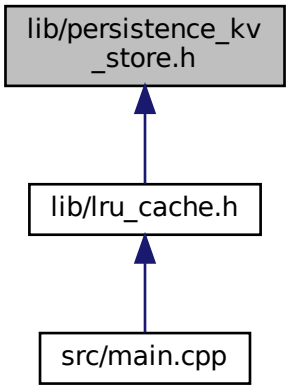
4.4 lib/persistence_kv_store.h File Reference

```
#include <iostream>
#include <fstream>
#include <unordered_map>
#include <vector>
#include <bitset>
#include <thread>
#include <atomic>
#include <chrono>
#include <memory>
#include <mutex>
#include "./bloomfilter.h"
#include "./tire.h"
```

Include dependency graph for persistence_kv_store.h:



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



Classes

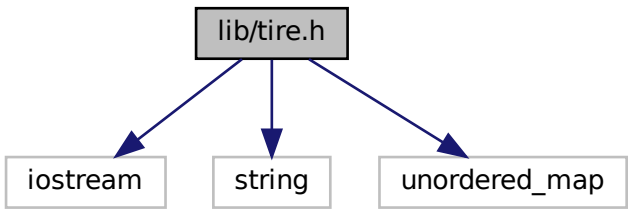
- class PersistenceKVStore

  A persistent key-value store with background rewriting and indexing.

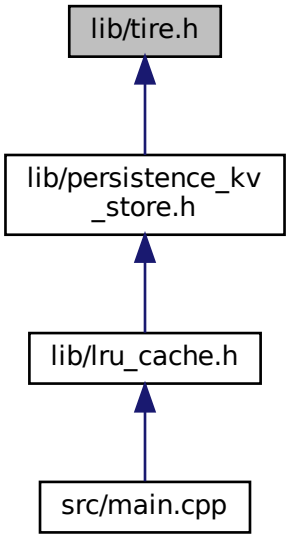
4.5 lib/tire.h File Reference

```
#include <iostream>
#include <string>
#include <unordered_map>
```

Include dependency graph for tire.h:



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



Classes

- class TrieNode

  Represents a node in the Trie.
- class Trie

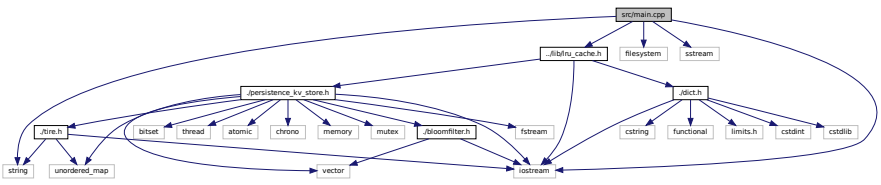
  Trie data structure for efficient key lookup.

4.6 src/main.cpp File Reference

Command-line interface for interacting with the LRU cache database.

```
#include <iostream>
#include <string>
#include <filesystem>
#include <sstream>
#include "../lib/lru_cache.h"
```

Include dependency graph for main.cpp:



Functions

- int main ()

  Main function providing a command-line interface for the LRU cache.

4.6.1 Detailed Description

Command-line interface for interacting with the LRU cache database.

This program provides a simple CLI to interact with an LRUCache-based key-value store. Supported commands:

- SET <key>

: Stores the key-value pair.

- GET <key>: Retrieves the value for a given key.
- DEL <key>: Deletes a key from the cache.
- EXIT: Terminates the program.

4.6.2 Function Documentation

4.6.2.1 main()

```
int main ( )
```

Main function providing a command-line interface for the LRU cache.

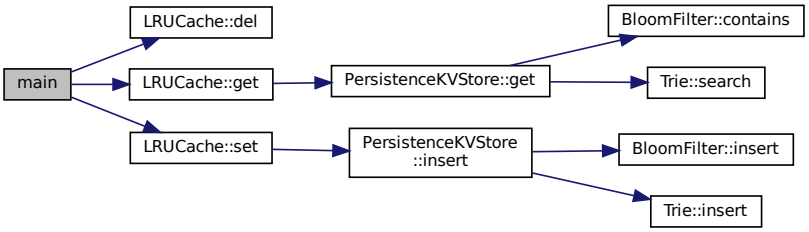
Returns

int Program exit status.

```
26  {
27      LRUCache database;
28      std::cout « "Enter command (SET key value, GET key, DEL key, or EXIT to quit):" « std::endl;
29
30      std::string input, command, key, value;
31      while (true)
32      {
33          std::cout « "> ";
34          std::getline(std::cin, input);
35
36          std::istringstream iss(input);
37          iss » command;
38          if (command == "SET")
39          {
40              iss » key » value;
41              if (key.empty() || value.empty())
42              {
43                  std::cout « "Invalid SET command. Usage: SET <key> <value>" « std::endl;
44                  continue;
45              }
46              database.set(strdup(key.c_str()), strdup(value.c_str()));
47          }
48          else if (command == "GET")
49          {
50              iss » key;
```

```
51              if (key.empty())
52              {
53                  std::cout « "Invalid GET command. Usage: GET <key>" « std::endl;
54                  continue;
55              }
56              std::string value = database.get(key.c_str());
57              std::cout « (value == "-1" ? "NULL" : value) « std::endl;
58          }
59          else if (command == "DEL")
60          {
61              iss » key;
62              if (key.empty())
63              {
64                  std::cout « "Invalid DEL command. Usage: DEL <key>" « std::endl;
65                  continue;
66              }
67              int status = database.del(key.c_str());
68              if (status)
69              {
70                  std::cout « "Does not exist" « std::endl;
71              }
72          }
73          else if (command == "EXIT")
74          {
75              break;
76          }
77          else
78          {
79              std::cout « "Unknown command. Use SET, GET, DEL, or EXIT." « std::endl;
80          }
81      }
82
83      return 0;
84  }
```

Here is the call graph for this function:



Index