CMSC 341 - Project 4: The File System - Fall 2021

Due: Tuesday, Dec 7, before 9:00 pm

Addenda

- 12/01/2021 - clarifications:
  - When rehashing, 25% transfer refers to 25% of m_size. We need to scan the table for the live data and transfer as many as required, once a live data is transferred it should be tagged as deleted in the old table.
  - The 25% of data is an integer number, we use the floor value of the result.
  - Reminder, m_size includes live data and deleted data.
  - Live data is the data node which is available to be used.
  - A deleted node is not available to be used.
  - An occupied bucket is a bucket which can contain either a live data node or a deleted node.
- 11/23/2021 - modifications: The files `hash.h` and `hash.cpp` have been modified. The function `File HashTable::getFile(string name, int diskBlock)` is changed to `File HashTable::getFile(string name, unsigned int diskBlock)`. Please download the latest version from the link on this page.

Objectives

- To implement a hash table.
- To implement quadratic probing to manage hash collisions.
- To implement an incremental hashing algorithm.
- To practice writing unit tests.

Introduction

We are implementing a new operating system. A part of the operating system is a file system to manage the files on the disk. The users can create a new file, delete an existing file, and search for a file. Then, the main operations in the file system are insert, find, and remove. Since the efficiency of operations is extremely important, we have decided to use a hash table to manage accessing the files on the disk. Your task is to develop this part of the operating system. The keys are files names, we know that this can cause collisions and clustering in a hash table, since there are many common names to be used. To manage the collision cases, we use the quadratic probing for the collision handling policy. The file system can change the table size based on some specific criteria in order to use the memory more efficiently. When a change is required, we need to rehash the entire table. However, the rehash operation happens incrementally.

Hash Table

A hash table is a data structure which implements the map ADT. It stores the pairs of key-value. The key determines the index number of an array cell (bucket) in which the value will be stored. A hash function calculates the index number. If there is more than one value for a key in the data set, the hash table uses a collision handling scheme to find another cell for storing the new value.

Quadratic Probing

When we try to find an index in the hash table for a key and we find out that the determined bucket is taken, we need to find another bucket to store the key. In a linear probing we store the information in the next available bucket. This can cause clustering, i.e. having many data points in a row. Clustering reduces the serach efficiency. Using quadratic probing instead of storing data in the next available bucket, we calculate a jump step to find the next bucket for storage. In this project we use the following equation to find the next bucket, in which i = 0,1,2,3 ....

index = ((Hash(key) % TableSize) + (i x i)) % TableSize

Assignment

Your assignment is to complete the class HashTable and the appropriate test functions.

The application starts with a hash table of size MINPRIME. After certain criteria appearing it will switch to another table and it transfers all data nodes from the current table to the new one incrementally. Once the switching happens it transfers 25% of the nodes in the old table and at every consecutive operation (insert/remove) It continues to transfer 25% more of the data from the old to table to the new

table until all data is transferred. We do not transfer deleted buckets to the new table.

After an insertion, if the load factor becomes greater than 0.5, we need to rehash to a new hash table. The capacity of the new table would be the smallest prime number greater than 4 times the current number of data points. The current number of data points is total number of occupied buckets minus total number of deleted buckets.

After a deletion, if the number of deleted buckets is more than 80 percent of the total number of occupied buckets, we need to rehash to a new table. The capacity of the new table would be the smallest prime number greater than 4 times the current number of data points. The current number of data points is total number of occupied buckets minus total number of deleted buckets.

During a rehashing process the deleted buckets will be removed from the system permanently. They will not be transferred to the new table.

For this project, you are provided with the following files:

- hash.h – header file for the HashTable class.
- hash.cpp – the template file for the HashTable class. Complete your HashTable implementation in this file.
- file.h – header file for the File class.
- file.cpp – implementation of the File class. You should not need to modify this class.
- driver.cpp – A sample driver program.
- driver.txt – A sample output produced by driver.cpp.

Specifications

HashTable Class

The HashTable class uses the File class. It has a member variable to store a pointer to a hash function. It also has two member variables to store pointers to two arrays of File objects. These arrays are the hash tables 1 and 2, and the name of the File object is used as the key for hashing purposes. A File object has another member variable which stores a pointer to the disk block. This variable holds a unique number for every File object.

HashTable::HashTable(unsigned size, hash_fn hash)	The alternative constructor, size is an unsigned integer to specify the length of hash table, and hash is a function pointer to a hash function. The type of hash is defined in hash.h. The table size must be a prime number between MINPRIME and MAXPRIME. If the user passes a size less than MINPRIME, the capacity must be set to MINPRIME. If the user passes a size larger than MAXPRIME, the capacity must be set to MAXPRIME. If the user passes a non-prime number the capacity must be set to the smallest prime number greater than user's value. Moreover, the constructor creates memory for table 1 and initializes all member variables. And, it initializes m_newTable to TABLE1.
HashTable::~HashTable()	Destructor, deallocates the memory for both table 1 and table 2.
bool HashTable::insert(File file)	This function inserts an object into the hash table. The insertion index is determined by applying the hash function m_hash that is set in the HashTable constructor call and then reducing the output of the hash function modulo the table size. A hash function is provided in the sample driver.cpp file to be used in this project. Hash collisions should be resolved using the quadratic probing policy. We insert into the table indicated by m_newTable. After every insertion we need to check for the proper criteria, and if it is required, we need to rehash the entire table incrementally into a new table. The incremental transfer proceeds with 25% of the nodes at a time. Once we transferred 25% of the nodes for the first time, the second 25% will be transferred at the next operation (insertion or removal). Once all data is transferred to the new table, the old table will be removed, and its memory will be deallocated. If the "file" object is inserted, the function returns true, otherwise it returns false. A File object can only be inserted once. The hash table does not contain duplicate objects.

	Moreover, the disk block value should be a valid one falling in the range [DISKMIN-DISKMAX]. Every File object is a unique object carrying the file's name and the disk block number. The file's name is the key which is used for hashing.
bool HashTable::remove(File file)	This function removes a data point from the hash table. In a hash table we do not empty the bucket, we only tag it as deleted. To tag a removed bucket we assign the DELETED object to the bucket. The DELETED object is defined in hash.h. To find the bucket of the object we should use the quadratic probing policy. After every deletion we need to check for the proper criteria, and if it is required, we need to rehash the entire table incrementally into a new table. The incremental transfer proceeds with 25% of the nodes at a time. Once we transferred 25% of the nodes for the first time, the second 25% will be transferred at the next operation (insertion or removal). Once all data is transferred to the new table, the old table will be removed, and its memory will be deallocated. If the "file" object is found and is deleted, the function returns true, otherwise it returns false.
File HashTable::getFile(string name, unsigned int diskBlock)	This function looks for the File object with name and diskBlock in the hash table, if the object is found the function returns it, otherwise the function returns empty object. If there are two hash tables at the time, the function needs to look into both tables.
unsigned HashTable::tableSize(TABLENAME tableName) const	This function returns the size of the hash table specified by the parameter tableName.
unsigned HashTable::numEntries(TABLENAME tableName) const	This function returns the number of elements in the hash table specified by the parameter tableName.
float HashTable::lambda(TABLENAME tablename) const	This function returns the load factor of the hash table. The load factor is the ratio of occupied buckets to the table capacity. The number of occupied buckets is the total of available buckets and deleted buckets. The parameter tablename specifies the table that should be used for the calculation.
float HashTable::deletedRatio(TABLENAME tablename) const	This function returns the ratio of the deleted buckets to the total number of occupied buckets . The parameter tablename specifies the table that should be used for the calculation.
void HashTable::dump()	This function dumps the contents of the hash table in index order. It prints the contents of the hash table in array-index order. **Note:** The implementation of this function is provided.
int HashTable::findNextPrime(int current)	This function returns the smallest prime number greater than the passed argument "current". If "current" is less than or equal to MINPRIME, the function returns MINPRIME. If "current" is greater than or equal to MAXPRIME, the function returns MAXPRIME. In a hash table we'd like to use a table with prime size. Then, everytime we need to determine the size for a new table we use this function. **Note:** The implementation of this function is provided.
bool HashTable::isPrime(int number)	This function returns true if the passed argument "number" is a prime number, otherwise it returns false. **Note:** The implementation of this function is provided.

Additional Requirements

Requirement: Private helper functions may be added to the HashTable class; however, they must be declared in the private section of the class declaration.

Requirement: Hash collisions must be resolved by quadratic probing depending on the current probing policy.
The equation for quadratic probing is ***index = ((Hash(key) % TableSize) + (i x i)) % TableSize***.

Requirement: The allocated memory to the hash table must be dynamically managed at execution time when there is rehashing.

Requirement: The capacity of the new table is determined by the information from the old table. It would be the smallest prime number greater than ((m_size - m_numDeleted)*4) from the old table.

Requirement: When rehashing, the deleted buckets will be removed from the system. No deleted bucket will be transferred to the new table.

Requirement: Here are the rules for lazy deletion,

- Treat deleted element as empty when inserting.
- Treat deleted element as occupied when searching.

Requirement: Here are the rules for rehashing criteria,

- rehash once the load factor exceeds 50%.
- rehash once the deleted ratio exceeds 80%.

Requirement: The load factor is the number of occupied buckets divided by the table size. The number of occupied buckets is the total of available data and deleted data.

Requirement: The deleted ratio is the number of deleted buckets divided by the number of occupied buckets.

Requirement: No STL containers or additional libraries may be used in the HashTable class. STL containers can be used in mytest.cpp for testing purposes.

Testing

You must write your own, extensive test driver called mytest.cpp; this file contains your Tester class, all test functions, all test cases, hash function, and the main function. You can copy the hash function, and the random data generator code from the sample driver.cpp to your mytest.cpp. Your test driver program must compile with your HashTable class and it must run to completion.

Following is a non-exhaustive list of tests to perform on your implementation. Please check the testing guidelines, it helps you to write effecive test cases.

Testing HashTable Class

- Test the insertion operation in the hash table. The following presents a sample algorithm to test the normal insertion operation:
  - There is some non-colliding data points in the hash table.
  - Insert multiple non-colliding keys.
  - Check whether they are inserted in the correct bucket (correct index).
  - Check whether the data size changes correctly.
- Test the find operation (getFile() function) with a few non-colliding keys. This also tests whether the insertion works correctly.
- Test the find operation (getFile() function) with a number of colliding keys without triggering a rehash. This also tests whether the insertion works correctly.
- Test the remove operation with a few non-colliding keys.
- Test the remove operation with a number of colliding keys without triggering a rehash.
- Test the rehashing is triggered after a descent number of colliding keys insertion.
- Test the rehashing is triggered after a descent number of colliding keys removal.

Random Numbers for Testing

For testing purposes, we need data. Data can be written as fixed values or can be generated randomly. Writing fixed data values might be a tedious work since we need a large amount of data points. The approach for creating data will be your decision.

In the file driver.cpp there is the class Random which generates pseudorandom numbers to be used as Disk Block Pointers. Since this code is using a fixed seed value, on the same machine it always generates the same sequence of numbers. That is why the numbers are called pseudorandom numbers, they are not real random numbers. Please note, the numbers are machine dependent, therefore, the numbers you see in the sample file driver.txt might be different from the numbers your machine generates.

Memory leaks and errors

- Run your test program in valgrind; check that there are no memory leaks or errors.
  Note: If valgrind finds memory errors, compile your code with the -g option to enable debugging support and then re-run valgrind with the -s and --track-origins=yes options. valgrind will show you the lines numbers where the errors are detected and can usually tell you which line is causing the error.
- Never ignore warnings. They are a major source of errors in a program.

What to Submit

You must submit the following files to the proj4 directory.

- hash.h
- hash.cpp
- file.h
- file.cpp
- mytest.cpp

If you followed the instructions in the Project Submission page to set up your directories, you can submit your code using the following command:

```
cp hash.h hash.cpp file.h file.cpp mytest.cpp ~/cs341proj/proj4/
```

Grading Rubric

The following presents a course rubric. It shows how a submitted project might lose points.

- Conforming to coding standards make about 10% of the grade.
- Correctness and completeness of your test cases (mytest.cpp) make about 15% of the grade.
- Passing tests make about 30% of the grade.

If the submitted project is in a state that receives the deduction for all above items, it will be graded for efforts. The grade will depend on the required efforts to complete such a work.
