

CompSci 261P Project #1: Hashing Algorithms

Program: Master of Computer Science Name: Yi-Dar Liu ID#: 32845675
--

CONTENT

A. INTRODUCTION	1
B. DESCRIPTION OF HASHING ALGORITHMS	2
C. THEORETICAL ANALYSIS OF RUNNING TIMES	10
D. EXPERIMENTS	12
E. COMPARATIVE EXPERIMENTAL ANALYSIS	15
F. CONCLUSIONS	31
G. REMARKS	32
H. REFERENCES	32

A. INTRODUCTION

This report is organized as follows: Section B describes the four hashing algorithms studied in this project. Section C analyzes their running times. Section D details the experiments conducted in this project. Section E shows and explains the empirical results. Section F concludes the insights gained from Section E. Section G supplements additional remarks about this project. Section H lists the references cited in this report.

General Terminology and Notation:

- [i] means i^{th} reference in the “H. REFERENCES” section.

B. DESCRIPTION OF HASHING ALGORITHMS

This report explores four hashing algorithms: Chained Hashing, Linear Hashing, Cuckoo Hashing, and Coalesced Hashing. Here is some terminology and notation used in this report:

Terminology:

- For convenience, we call Chained Hashing, Linear Hashing, and Cuckoo Hashing “basic hashing algorithms”.
- An “entry” is a pair of $\langle \text{key}, \text{value} \rangle$. Because key is the primary part of the entry, sometimes we use “entry” and “key” interchangeably. For example, inserting/deleting an entry is equivalent to inserting/deleting a key.
- An “element” is a node in a linked list or a cell in a hash table which can contain or not contain an entry. Sometimes we also call it a “slot” when it comes to an empty element. An element in a linked list of Chained Hashing or in a hash table of Linear/Cuckoo/Coalesced Hashing has the following structure:
 - key: an integer variable used to store the key, must be unique within the hash table
 - value: an integer variable used to store the value
 - valid: a bool variable which is 1 if the element is nonempty and 0 if empty
 - prev: a pointer to the previous entry in the chain, which is null if the element is the head; only applicable for Coalesced Hashing
 - next: a pointer to the next entry in the chain, which is null if the element is the tail; only applicable for Coalesced Hashing

Notation:

- k: key
- v: value
- n: number of entries in the hash table
- N: hash table size
- H: hash table
- H1: the second hash table of Cuckoo Hashing
- $h(x)$: hash function
- $h1(x)$: the second hash function of Cuckoo Hashing

Chained Hashing

The basic idea of Chained Hashing is to create an array of pointers, each of which points to a linked list which stores entries hashed to the pointer. Pseudocode of the basic operations is shown below:

Set:

First find the pointer to the linked list, and then walk through the list to check if the key already exists. If not, store the key at the end of the linked list.

```
def set(k, v):  
    i ← h(k)  
    if linked list pointed by H[i] contains a key = k: update the value  
    else: store(k, v) in the end of the linked list
```

Search:

First find the pointer to the linked list, and then walk through the list to find the key.

```
def search(k):  
    i ← h(k)  
    if linked list pointed by H[i] contains a key = k: return the value  
    else: throw exception
```

Delete:

First find the pointer to the linked list, and then walk through the list to find the key. If it is found, delete the key from the linked list.

```
def delete(k):  
    i ← h(k)  
    if linked list pointed by H[i] contains a key = k: delete the entry from the linked list  
    else: throw exception
```

Linear Hashing

The basic idea of Linear Hashing is to create an array of elements which store entries with matching hash values. Pseudocode of the basic operations is shown below. It is very similar to the pseudocode in the Lecture Slide. However, when the hash table is full, it is likely that the while loop never terminates, so we add termination conditions (highlighted in yellow) to the Lecture Slide version.

Set:

Look up the slot with matching hash value first. If there is a collision, then search forward for the closest empty slot to store the entry.

```
def set(k, v):
    if n = N: resize H
    i ← h(k)
    while H[i].valid = 1 and H[i].key ≠ k:
        i ← (i+1) mod N
    if H[i].valid = 0: store (k, v) in H[i]
    else: H[i].value ← v
```

Search:

Lookup of a key k is done by scanning the hash table from h(k) until either k or an empty element is found.

```
def search(k):
    i ← h(k)
    counter ← 1
    while H[i].valid = 1 and H[i].key ≠ k and counter ≤ N:
        i ← (i+1) mod N
        increment counter
    if H[i].valid = 1 and counter ≤ N: return H[i].value
    else: throw exception
```

Delete:

We use “eager delete” to enhance search efficiency and maintain hash table capacity. After a key is deleted, some keys may have to be moved back in order to fill the hole.

```
def delete(k):
    i ← h(k)
    counter ← 1
    while H[i].valid = 1 and H[i].key ≠ k and counter ≤ N:
        i ← (i+1) mod N
        increment counter
    if H[i].valid = 0 or counter > N: throw exception
    j ← (i+1) mod N
    i_1 ← i //back up hash value
    while H[j] is nonempty and j ≠ i_1:
        if h(H[j].key) is not in the (circular) range [i+1...j]:
```

```

        H[i] ← H[j]
        i ← j
    j ← (j+1) mod N
clear H[i]

```

Cuckoo Hashing

The basic idea of Cuckoo Hashing is to create two hash tables, both of which are arrays of elements, with two different hash functions. Store a new entry in the first hash table first. If the slot in the first hash table has been occupied, then kick the incumbent off to the second hash table with the second hash function and then occupy the slot. Pseudocode of the basic operations is shown below:

Set:

When inserting a new key, first we check if it is already in the hash tables. If not, then we insert it in the first hash table and move around other keys if necessary. It is likely that there is a closed loop and the rearrangement never stops. We use the answer to Homework 2 Required Problem 2 to find the cycle. That is, we count the number of times of moving the starting key. If the number reaches 3, it means that the key will be pushed back to the original cell and there is an infinite loop. The advantage of this method is that when the loop is detected, the hash tables restore to the original state as if the new key were not yet inserted. In this case, we resize the hash tables and insert the key again.

```

def set(k, v):
    //first check if key has already been in the hash table...
    i ← h(k)
    if H[i].valid = 1 and H[i].key = k:
        H[i].value ← v
    i ← h1(k)
    if H1[i].valid = 1 and H1[i].key = k:
        H1[i].value ← v

    //if key is not in the hash table...
    rc ← set_1(k, v) //set_1() is defined below
    while rc < 0: //if there is a loop => resize the hash tables and try again
        resize H and H1
        rc ← set_1(k, v)

```

```

def set_1(k, v): //return -1 if there is a closed loop, return 0 if success
    k_o ← k //back up the starting key
    counter ← 0
    t ← 0
    while true:
        if k = k_o: increment the counter
        if counter = 3: return -1 //there is a closed loop
        if t = 0:
            i ← h(k)
            if H[i].valid = 0:
                store(k, v) in H[i]

```

```

        return 0
    else:
        k_1 ← H[i].key
        v_1 ← H[i].value
        store(k, v) in H[i]
        t ← 1
        k ← k_1
        v ← v_1
    else:
        i ← h1(k)
        if H1[i].valid = 0:
            store(k, v) in H1[i]
            return 0
        else:
            k_1 ← H1[i].key
            v_1 ← H1[i].value
            store(k, v) in H1[i]
            t ← 0
            k ← k_1
            v ← v_1

```

Search:

Searching a key in the Cuckoo hash tables is straightforward. It just needs to check two places.

```

def search(k):
    i ← h(k)
    if H[i].valid = 1 and H[i].key = k: return H[i].value
    i ← h1(k)
    if H1[i].valid = 1 and H1[i].key = k: return H1[i].value
    throw exception

```

Delete:

Similar to search.

```

def delete(k):
    i ← h(k)
    if H[i].valid = 1 and H[i].key = k: clear H[i]
    i ← h1(k)
    if H1[i].valid = 1 and H1[i].key = k: clear H1[i]
    throw exception

```

Coalesced Hashing

The basic idea of Coalesced Hashing is to create an array of elements, which are divided into two parts: one part is the main hash table, and the other is used to store entries which collide with those in the main hash table, called the “cellar”. Entries with the same hash values are connected into a linked list via pointers. If the cellar is full, another linked list with a different hash value may have to be “coalesced” with the linked list.

The reason why we choose this as our additional algorithm is that it combines the advantages of the three basic hashing algorithms:

1. It has the benefit of Chained Hashing: It uses pointers to connect keys with the same hash values, so lookup will be faster than Linear Hashing.
2. It has the benefit of Linear Hashing: It allocates a continuous memory block to the hash table, so the caching performance is better than Chained Hashing. Moreover, it resolves collisions by placing the new key in a specific empty slot. As long as the hash table is not yet full, the collision can be always resolved. On the contrary, Cuckoo Hashing may have to be rehashed to resolve the collision before the hash table becomes full.
3. It has the benefit of Cuckoo Hashing: Like Cuckoo Hashing, it divides the memory into two parts – the main hash table and the cellar. If the slot in the main hash table has been occupied, we can place the key in the cellar. The main hash table will not be messed up until the cellar is full.

Therefore, we expect Coalesced Hashing to perform better than all of the three basic hashing algorithms.

The biggest weakness of Coalesced Hashing is deletion. If we delete an entry and try to fill the hole, we may find it necessary to un-coalesce the chains, which could be more cumbersome than Linear Hashing’s eager delete. Therefore, unlike Linear Hashing, we implement “lazy delete” for Coalesced Hashing instead. When an entry is deleted, if it is at the end of the linked list, then clearing the entry will not affect other entries in the hash table, so we just clear it. Otherwise, we put a deletion marker in the slot, called “tombstone”. In our implementation, a tombstone is characterized by having $H[j].valid = 0$ and $H[j].next \geq 0$, which indicates that it is still within some linked list and cannot be completely erased. When inserting a new key, we will try to fill tombstones first (if the key and the tombstone are hashed to the same linked list) before utilizing any empty slot.

Set:

First, we check if the key already exists in the hash table. It is involved with walking through the linked list with the matching hash value. If we encounter a tombstone during the process, then we record it for future use. If the key does not exist, then we insert the key, whose location is determined based on the following priority:

1. If the element with the matching hash value is empty, just insert there.
2. If there is a tombstone in the linked list, put the entry into the first one.
3. Otherwise, insert the entry into a specific empty slot and connect it with the last element of the linked list. In our implementation, we maintain a pointer to the highest-indexed empty slot in the hash table. If the hash table size is M , then the pointer is initialized to be $M-1$. Every modification of the hash table may also involve updating the pointer. If the pointer is -1 , it means that the hash table has been full and needs to be resized before any key is inserted.

```

def set(k, v):
    //check if key already exists
    //if there is "tombstone" in the chain, write down for future use
    tombstone ← -1
    do:
        if H[i].valid = 1 and H[i].key = k:
            H[i].value = v
            return
        else:
            if H[i].valid = 0 and H[i].next ≥ 0 and tombstone = -1: tombstone ← i
            prev ← i
            i ← H[i].next
    while i ≥ 0 //break loop if i < 0, i.e., reach the end of the linked list
    i ← h(k)
    if H[i].valid = 0 and H[i].next < 0 and i ≠ empty_slot: //the top place to insert key
        store(k, v) in H[i]
    else if tombstone ≥ 0: //if there is tombstone => use tombstone first
        store(k, v) in H[tombstone]
    else: //otherwise use empty slot
        if empty_slot < 0: //hash table has been full
            resize H
            H[prev].next ← empty_slot
            store(k, v) in H[empty_slot]
            //reset empty_slot
            if H has been full (including tombstones): empty_slot ← -1
        else: //empty_slot ≥ 0
            j ← empty_slot - 1
            while j ≥ 0:
                if H[j].valid = 0 and H[j].next = null: //avoid tombstone
                    empty_slot ← j
                    break the loop
                decrement j
            if j < 0: throw exception //means that H has been full

```

Search:

Like Chained Hashing, search is just walking through the linked list.

```

def search(k):
    i ← h(k)
    while H[i].valid ≠ 1 or H[i].key ≠ k:
        if H[i].next is not null:
            i ← H[i].next
        else: throw exception
    return H[i].value

```


Delete:

As we mentioned earlier, if the deleted entry is at the end of a linked list, then clearing the entry will not affect other entries in the hash table, so we just clear it and update the empty slot pointer if the deleted entry happens to have a higher index. Otherwise, we clear everything but prev and next pointers to create a tombstone.

```
def delete(k):
    i ← h(k)
    while H[i].valid ≠ 1 or H[i].key ≠ k:
        if H[i].next is not null:
            prev ← i
            i ← H[i].next
        else: throw exception

    next ← H[i].next
    if next < 0: //if the item to be deleted is at the end of the linked list
        clear H[i]
        empty_slot ← max(empty_slot, i);
        //check if the previous elements are also tombstones; if yes, convert them to empty slots
        while prev ≥ 0:
            set H[prev].next to be null
            if H[prev].valid = 0:
                empty_slot ← max(empty_slot, prev)
                prev ← H[prev].prev
            else: prev = -1
    else: //if the item is in the middle of the linked list => create tombstone
        clear H[i]
        H[i].prev ← prev
        H[i].next ← next
```

C. THEORETICAL ANALYSIS OF RUNNING TIMES

Below is the summary of running times of each hashing algorithm. Please refer to References for their complete proofs.

Notation:

- n : number of entries in the hash table
- N : hash table size
- $\alpha = \frac{n}{N}$: load factor
- Worst: “typical” worst-case time per operation (assume that hash values are uniformly distributed and mutually independent)
- Average: expected time per operation (assume that hash values are uniformly distributed and mutually independent)
- Amortized: amortized time, i.e., worst average time per operation

	Successful Search	Unsuccessful Search	Set	Delete
Chained Hashing	Worst: $\Theta(\frac{\log n}{\log \log n})$ Average: $1+\alpha$ Amortized: $O(1)$	Average: $1+\alpha$ Amortized: $O(1)$	Average: $2+\alpha$ Amortized: $O(1)$	Average: $2+\alpha$ Amortized: $O(1)$
Linear Hashing	Worst: $\Omega(\log n)$ Average: $\frac{1}{2}(1 + \frac{1}{1-\alpha})$ Amortized: $O(1)$	Average: $\frac{1}{2}(1 + \frac{1}{(1-\alpha)^2})$ Amortized: $O(1)$	Amortized: $O(1)$	Amortized: $O(1)$
Cuckoo Hashing	Worst: 2 Amortized: $O(1)$	Amortized: $O(1)$	Amortized: $O(1)$	Amortized: $O(1)$
Coalesced Hashing	Average: $1 + \frac{1}{8\alpha}(e^{2\alpha} - 1 - 2\alpha) + \frac{\alpha}{4}$ Amortized: $O(1)$	Average: $1 + \frac{1}{4}(e^{2\alpha} - 1 - 2\alpha)$ Amortized: $O(1)$	Amortized: $O(1)$	Amortized: $O(1)$

References: [1], [2], [3], Lecture Slides, Assigned Readings

Our project defines running time as the “number of elements accessed”. Any element access regarding lookup and modification increments the running time counter by 1. Using the results above, we predict the worst and average running times of the hashing algorithms as follows:

Chained Hashing

- **Successful Search:** A successful search is involved with a memory access to the pointer and another to the entry itself, so search cost is at least 2. As load factor becomes larger, the average length of linked lists should become longer and average search cost should also increase. When the number of entries increases, the length of the longest linked list should become longer and worst search cost should increase as well. In particular, it should be proportional to $\frac{\log n}{\log \log n}$.
- **Unsuccessful Search:** Since the unsuccessful search needs to walk through the whole linked list, the average cost should be $1+\alpha$.
- **Set:** Besides unsuccessful search, set requires one more memory access to an empty slot, so set cost should be one more memory access than unsuccessful search cost.
- **Delete:** Besides successful search, delete requires one more memory access to the previous entry in the chain in order to update the next pointer, so delete cost should be one more memory access than successful search cost. We expect this to be the most expensive operation of Chained Hashing.

Linear Hashing

- **Successful Search:** As load factor increases, the hash table becomes fuller, so it should walk through more entries to find the key. We expect the average cost to be $\frac{1}{2}(1 + \frac{1}{1-\alpha})$, which increases exponentially with α . As sample size increases, so does the longest continuous block, so the worst search cost should also rise. In particular, it should be proportional to $\log n$, which is larger than Chained Hashing's $\frac{\log n}{\log \log n}$ and Cuckoo Hashing's 2, so we expect Linear Hashing to have the worst "worst-case" search time among the three basic hashing algorithms.
- **Unsuccessful Search:** By the same token, we expect the average cost to be $\frac{1}{2}(1 + \frac{1}{(1-\alpha)^2})$, which increase more drastically with α than Successful Search. In particular, as the load factor becomes 1, it will need to walk through the whole hash table to confirm if a key exists. We expect the worst cost to be the same as the table size in this case.
- **Set:** As load factor increases, keys are more likely to be clustered. It implies that it has to walk through more elements to find an empty slot, so the set cost will rise.
- **Delete:** Deleting a key may involve moving other keys around so that subsequent search can succeed. We expect this to be the most expensive operation of Linear Hashing.

Cuckoo Hashing

- **Successful Search:** Since it only needs to check two places, its worst and average cost should be no more than 2 memory accesses. However, as the first hash table approaches full, more keys will be placed in the second hash table and thus accessing two elements in a lookup will become more frequent. Therefore, Cuckoo Hashing's average search cost will still increase with load factor.
- **Unsuccessful Search:** Same as above.
- **Set:** Before inserting a key, it needs to check two locations to confirm if the key already exists, so the set cost is at least $2 + 1(\text{insertion per se}) = 3$. Moreover, there is a risk of infinite loop, which will result in resizing the hash tables and can be time-consuming. We expect this to be the most expensive operation of Cuckoo Hashing.
- **Delete:** Same as search, should be straightforward and its cost should be independent of sample size and load factor.








Coalesced Hashing

- **Successful Search:** When load factor increases, coalescence will happen more frequently and hurt Coalesced Hashing's performance. We expect the average cost to be $1 + \frac{1}{8\alpha}(e^{2\alpha} - 1 - 2\alpha) + \frac{\alpha}{4}$. Please note that $e^{2\alpha}$ increases faster than 8α , so the cost should increase with α .
- **Unsuccessful Search:** Unlike Linear Hashing, thanks to null pointer, even if load factor is 1, Coalesced Hashing does not need to walk through the whole hash table to check if a key exists. We expect the average cost to be $1 + \frac{1}{4}(e^{2\alpha} - 1 - 2\alpha)$. Since the load factor of Coalesced Hashing is between 0 and 1, as per calculation the average search cost is bounded by 2, a very desirable property like Cuckoo Hashing.
- **Set:** Besides checking if a key already exists, sometimes it also needs to update the empty slot pointer, so set cost would be higher than Chained Hashing. If the cellar is not yet full and no delete was done, updating the empty slot pointer is just decrementing the pointer. However, if empty slots are sparsely distributed throughout the hash table, updating the empty slot pointer will be costly. Therefore, we expect this to be the most expensive operation of Coalesced Hashing.
- **Delete:** Because we implement lazy delete, delete should be straightforward and cheap.

D. EXPERIMENTS

We use C++ to implement our code. The code is organized as follows:

- main.cpp: used to execute complete test
- hashing.h and hashing.cpp: implement hashing algorithms
- unit_test.h: perform simple tests to debug hashing.cpp
- complete_test.h: define the experiment settings and perform the actual experiments which will be explained later
- prime.h and prime.cpp: can force hash function's parameter N to be a prime number, but not used in our experiments

 complete_test	2018/4/29 上午 01:15	C Header File	14 KB
 hashing	2018/4/28 下午 06:19	C Header File	3 KB
 prime	2018/4/26 下午 03:44	C Header File	1 KB
 unit_test	2018/4/28 下午 04:21	C Header File	4 KB
 hashing	2018/4/28 下午 10:12	C++ Source File	14 KB
 main	2018/4/29 上午 01:14	C++ Source File	1 KB
 prime	2018/4/26 下午 03:44	C++ Source File	1 KB

We imagine creating a hash table for CS261P to record UCI MCS students' mid-term exam scores. Students are identified by 8-digit student IDs, ranging from 00000000 to 99999999, which serves as the key k . We assume that student IDs are mutually independent and (1) uniformly distributed within the range or (2) normally distributed with mean = 50000000 and standard deviation = 25000000. We would like to save students' CS261P mid-term exam scores, which are the values v associated with the keys. Assume that the mid-term exam's full mark is 100 points, so score ranges from 0 (assume no extra point loss for wrong answers). We also assume that MCS students are quite smart, so scores are normally distributed with mean = 75 and standard deviation = 12.5. We use C++ `<random>` header file to generate these random numbers. The following is an example of how we generate student IDs (keys) and scores (values):

```
#include <random>    // std::default_random_engine
std::default_random_engine generator;
std::uniform_int_distribution<int> k_distribution(0, 99999999);
std::normal_distribution<double> v_distribution(75, 12.5);
k = k_distribution(generator); //generate a key
v = value(v_distribution(generator), d->vmin, d->vmax); //generate a value, value() is a function to
transform a double number to an integer between specified minimum (d->vmin) and maximum (d->vmax)
```

We generate test cases of different class sizes n from $2^4 = 16$ to $2^{13} = 8192$ (assume that students can take the program online, so the class size can be expanded without limits). The files are named as "test_d_i.txt" where:

- d: 1 if keys are uniformly distributed, 2 if keys are normally distributed
- i = log n

For example, "test_1_4.txt" is the test case with $n = 2^4$ uniformly distributed keys. The first line of the files is sample size n , followed by n pairs of "key, value", which are separated by semicolons. The following screenshot shows all the data used for test cases:

test_1_4	2018/4/28 下午 10:11	文字文件	1 KB
test_1_5	2018/4/28 下午 10:11	文字文件	1 KB
test_1_6	2018/4/28 下午 10:11	文字文件	1 KB
test_1_7	2018/4/28 下午 10:11	文字文件	2 KB
test_1_8	2018/4/28 下午 10:11	文字文件	3 KB
test_1_9	2018/4/28 下午 10:11	文字文件	6 KB
test_1_10	2018/4/28 下午 10:11	文字文件	12 KB
test_1_11	2018/4/28 下午 10:11	文字文件	24 KB
test_1_12	2018/4/28 下午 10:11	文字文件	48 KB
test_1_13	2018/4/28 下午 10:11	文字文件	96 KB
test_2_4	2018/4/30 上午 11:27	文字文件	1 KB
test_2_5	2018/4/30 上午 11:27	文字文件	1 KB
test_2_6	2018/4/30 上午 11:27	文字文件	1 KB
test_2_7	2018/4/30 上午 11:27	文字文件	2 KB
test_2_8	2018/4/30 上午 11:27	文字文件	3 KB
test_2_9	2018/4/30 上午 11:27	文字文件	6 KB
test_2_10	2018/4/30 上午 11:27	文字文件	12 KB
test_2_11	2018/4/30 上午 11:27	文字文件	24 KB
test_2_12	2018/4/30 上午 11:27	文字文件	48 KB
test_2_13	2018/4/30 上午 11:27	文字文件	96 KB

We try different load factors α from 0.1 to 1 with step size = 0.1 and from 1 to 4 with step size = 0.5. Dividing class size by load factor leads to the hash table size N , i.e., $N = \frac{n}{\alpha}$. For fairness, we compare hashing algorithms with the same hash table size. Because Cuckoo Hashing has two hash tables, its hash table is half of that of Linear Hashing. As for Coalesced Hashing, [2] advises that the optimal ratio of main hash table to whole hash table is 0.86, so we allocate $0.86N$ to the main hash table and the remaining to the cellar. Hash function used in our experiments is key mod some number which corresponds to the hash table size. The following table summarizes the hashing methods' memory usage and hash functions:

	Memory Used	Hash Function
Chained Hashing	N linked list pointers + size of all linked lists	key mod N
Linear Hashing	N elements	key mod N
Cuckoo Hashing	N elements, which are divided into two hash tables of size $\frac{N}{2}$	h1: key mod $(N/2)$ h2: (key/($N/2$)) mod $(N/2)$
Coalesced Hashing	N elements, which are divided into a main hash table of size $0.86N$ and a cellar of size $0.14N$	key mod $0.86N$

We consult [1] and perform the following experiments:

Dictionaries of Stable Size

Insert n keys, and then repeat $3n$ times the following four operations:

- 1) Unsuccessful Search: Randomly search a key which does not exist in the hash table
- 2) Successful Search: Randomly search a key which exists in the hash table
- 3) Delete: Randomly delete a key which exists in the hash table
- 4) Set: Randomly set a key which does not exist in the hash table

In order to tell if a key exists in the hash table and at the same verify the correctness of our code, we use C++ `<set>` class to store the keys. If we would like to find a key which does not exist in the hash table, we randomly generate a number k until `set.find(k)` returns `end()`. If we would like to find a key which exists in the hash table, we randomly generate a number h between 0 and $n-1$ and find the h^{th} key in the set. After performing a sequence of the four operations, we restore the hash table to match the key set.

As we mentioned in “C. THEORETICAL ANALYSIS OF RUNNING TIMES”, our performance measure is the number of elements accessed because it is the primary cost in any hash table operations. As long as an instruction is involved with reading or modifying an element in the hash table, the cost counter will be incremented by one. Each hash table operation function returns the number of elements accessed, and we average/take the maximum of the numbers to obtain the average/worst-case cost.

When Cuckoo Hashing encounters an infinite loop, we resolve the issue by doubling the hash table size. In order to put the negative effect of infinite loops into consideration, we add the resizing cost to Cuckoo Hashing's set cost. However, because we care about the “average” cost of an operation, resizing cost of Coalesced Hashing is not counted. The long sequence of operations will unavoidably create tombstones in Coalesced Hashing's hash table and may trigger resizing. However, any single operation per se will not cause resizing unless the hash table has been full.

Growing and Shrinking Dictionaries

After doing the experiment above, we delete all the keys in the hash table. We calculate the worst-case and average cost of a set/delete operation during the process of building/destroying a hash table containing n keys. When load factor is more than 1, Linear Hashing and Coalesced Hashing must be resized during the building process. We add the resizing cost to their set costs.

Other implementation details include:

- Chained Hashing: We use C++ `<vector>` class to implement linked list.
- Linear Hashing: As we mentioned in “B. DESCRIPTION OF HASHING ALGORITHMS”, we use eager delete.
- Coalesced Hashing: As we mentioned in “B. DESCRIPTION OF HASHING ALGORITHMS”, we use lazy delete, i.e., put “tombstones” in deleted slots.

Our experiment settings are summarized below:

- key (k): UCI MCS student ID, (1) uniformly distributed between 00000000 and 99999999 or (2) normally distributed with mean = 50000000 and standard deviation = 25000000
- value (v): CS261P mid-term exam score, normally distributed with mean = 75 and standard deviation = 12.5
- sample size (n): 2^i , $i = 4, 5, \dots, 13$
- load factor (α): $0.1j$, $j = 1, 2, \dots, 10$, and $1+0.5k$, $k = 1, 2, \dots, 6$
- hash table size/hash function parameter (N) = $\frac{n}{\alpha}$

E. COMPARATIVE EXPERIMENTAL ANALYSIS












To run our program in Linux, just put the code in a folder and use “make” command to compile the code. A program called “hashing” will be generated in the folder. It takes two arguments:

- argument 1: used to annotate the test case files and the result file in case that several versions of complete tests are done
- argument 2: 1 if we would like to generate test cases, 0 if they are already available (files names must be in the form of “test_[argument 1]_i.txt”, i = 4, 5, ... , 13)




Please note that by default, the keys produced in the test case files are uniformly distributed between 00000000 and 99999999. Conducting the experiments using normal distribution assumption only requires some small revisions to “complete_test.h” (specifically, functions test_generator() and single_test()). The following screenshot shows a trial run of our program in the UCI ICS openlab:

```
vidarl@andromeda-20 14:32:08 ~/cs261p/project1
$ make
g++ -g -Wall -MMD -std=c++11 -pthread -c -o prime.o prime.cpp
g++ -g -Wall -MMD -std=c++11 -pthread -c -o main.o main.cpp
g++ -g -Wall -MMD -std=c++11 -pthread -c -o hashing.o hashing.cpp
g++ -g -Wall -MMD -std=c++11 -pthread prime.o main.o hashing.o -o hashing
vidarl@andromeda-20 14:32:15 ~/cs261p/project1
$ ./hashing 3 1
vidarl@andromeda-20 14:42:09 ~/cs261p/project1
$
```

The program will generate test cases based on the aforementioned experiment settings (if argument 2 is 1), import them, perform the experiments, and export the test results to a text file called “result_[argument 1].txt”. As can be seen from the screenshot, the execution time of the program takes about 10 minutes. The following files are the outputs of the previous trial run:

 result_3.txt	59 KB	2018/5/1 下午 02:42:09
 test_3_13.txt	96 KB	2018/5/1 下午 02:33:06
 test_3_12.txt	48 KB	2018/5/1 下午 02:33:06
 test_3_11.txt	24 KB	2018/5/1 下午 02:33:06
 test_3_10.txt	12 KB	2018/5/1 下午 02:33:06
 test_3_9.txt	6 KB	2018/5/1 下午 02:33:06
 test_3_8.txt	3 KB	2018/5/1 下午 02:33:06
 test_3_7.txt	2 KB	2018/5/1 下午 02:33:06
 test_3_6.txt	1 KB	2018/5/1 下午 02:33:06
 test_3_5.txt	1 KB	2018/5/1 下午 02:33:06
 test_3_4.txt	1 KB	2018/5/1 下午 02:33:06

In our case, argument 1 is 1 (keys are uniformly distributed) and 2 (keys are normally distributed). We imported the resulting “result_1.txt” and “result_2.txt” into spreadsheets and drew plots with Excel built-in tools. The following screenshot shows the program outputs and our spreadsheets used to generate the plots in this section.

 result_1	2018/4/29 上午 01:20	文字文件	59 KB
 result_1	2018/4/30 下午 12:05	Microsoft Excel 工作...	356 KB
 result_2	2018/4/30 上午 11:32	文字文件	59 KB
 result_2	2018/4/30 上午 11:59	Microsoft Excel 工作...	357 KB

[1] mentions that the typical load factor of a hash table is $\frac{1}{3}$. Therefore, when we show a plot of costs against varying sample sizes, the load factor is fixed at 0.3. On the other hand, when we show a plot of costs against varying load factors, the sample size is fixed at $2^{10} = 1024$ because we think the size is large enough to represent a practical case (normally a class in a university program would not exceed 1024 students), and it also allows clear demonstration of the hashing algorithms' differences. However, different combinations of load factor and sample size are also possible given the data we have.

For hashing algorithms with static hash table size, namely Linear Hashing, Cuckoo Hashing, and Coalesced Hashing, we also tried $\alpha > 1$ and measured their resizing costs. However, after reviewing the results, we do not think the numbers are quite meaningful. Anyway, we can allocate a sufficiently large hash table to contain the entries, or we can resize the hash table as long as its load factor reaches a threshold. Unlike Cuckoo Hashing's infinite loop problem, this kind of resizing is predictable and its cost can be detached from a single operation (for example, auto-resize the hash table when we do not perform hash table operations). Therefore, only Chained Hashing's results are shown for $\alpha > 1$.

We find that there is no significant difference in results between uniform distribution assumption and normal distribution assumption. It is possibly because our key domain is quite large compared with sample size and the standard deviation is not small. Consequently, even though the keys are normally distributed, their hash values do not exhibit apparent clustering behavior. Therefore, only uniform distribution assumption results are shown here. Interested readers can refer to "result_2.xlsx" for normal distribution assumption results.

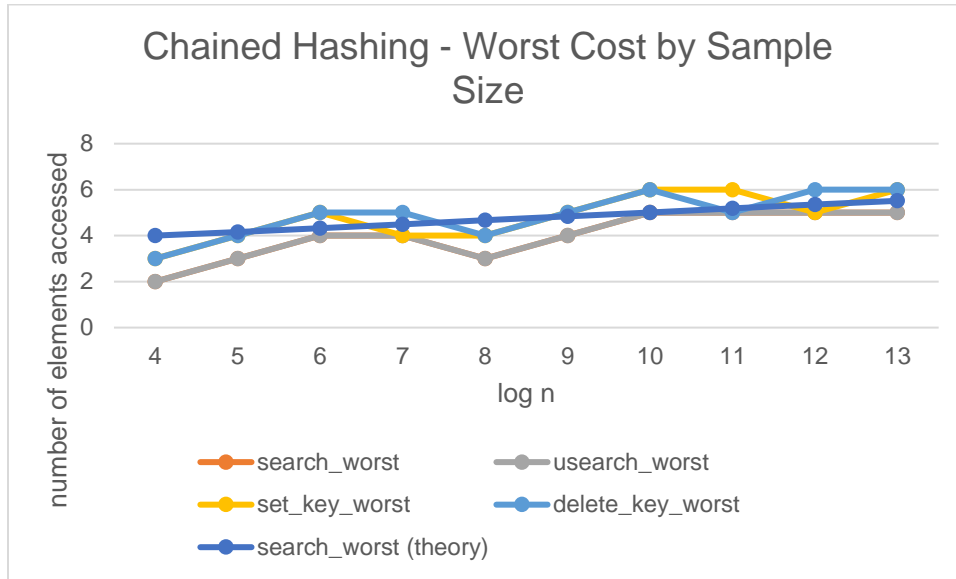
Individual Results

Below we show the results by individual hashing algorithm. According to our analysis in Section C, worst-case time is more related to sample size, while average time is more correlated with load factor. Therefore, only worst-case time by sample size and average time by load factor plots are shown for each hashing algorithm. The primary goal here is to investigate whether the results are consistent with our predictions in Section C.

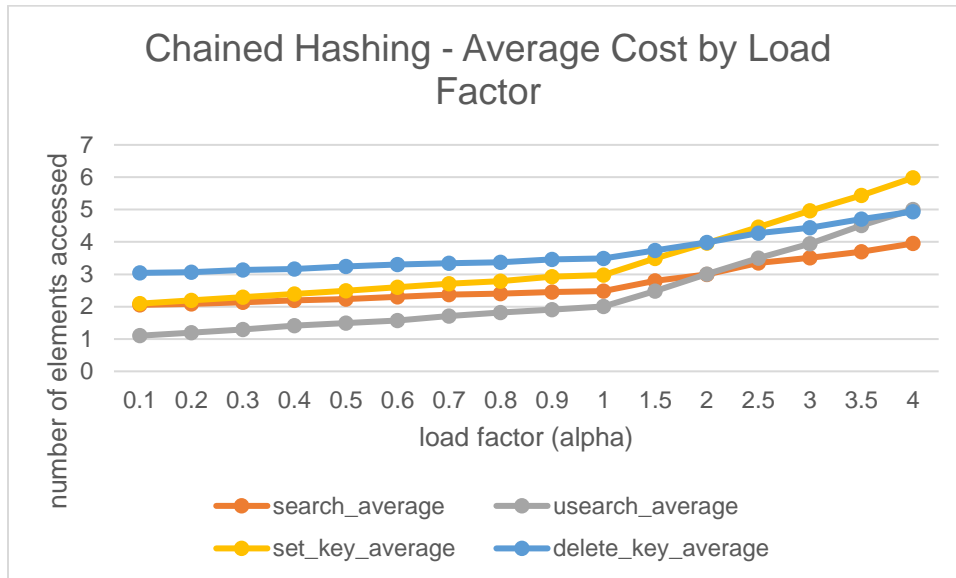
Notation:

- search_average: average cost of a successful search
- usearch_average: average cost of an unsuccessful search
- set_key_average: average cost of setting a key
- delete_key_average: average cost of deleting a key
- search_worst: worst cost of a successful search
- usearch_worst: worst cost of an unsuccessful search
- set_key_worst: worst cost of setting a key
- delete_key_worst: worst cost of deleting a key

Chained Hashing: (load factor = 0.3, n = 1024)

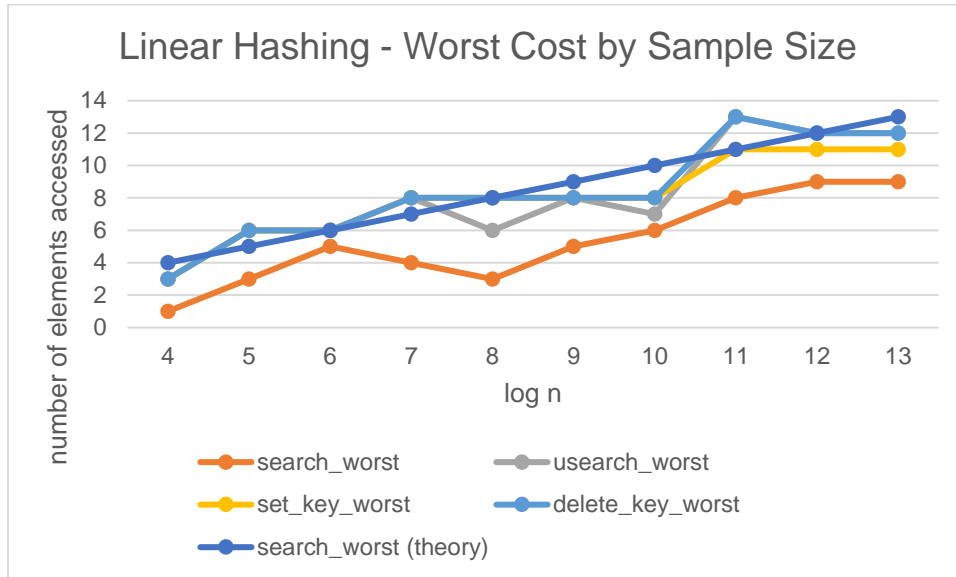


Generally, all operations see their costs rise with sample size. The “search_worst” line is invisible because it is completely the same as the “usearch_worst” line. It is worth noticing that worst search time is below the theoretical bound $2 + \frac{\log n}{\log \log n} = \Theta\left(\frac{\log n}{\log \log n}\right)$.

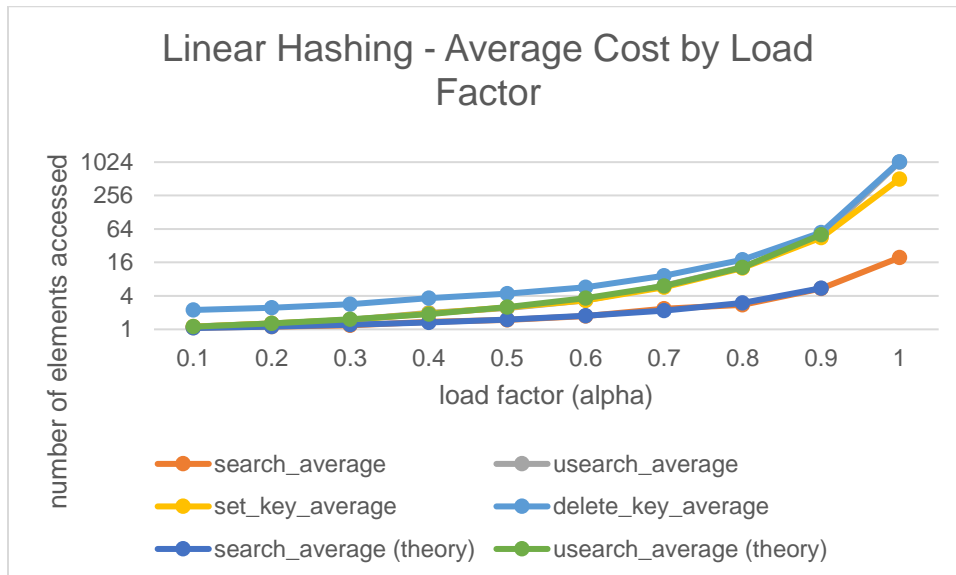


As we expected, average unsuccessful search cost is equal to $1 + \alpha$, and successful search cost is at least 2. Average unsuccessful search cost and average set cost differs by 1 memory access, and average successful search cost and average delete cost differs by 1 memory access as well. Please note that the former pair of lines are almost parallel, and the latter pair of lines are almost parallel too. The lines completely coincide with our predictions.

Linear Hashing: (load factor = 0.3, n = 1024)

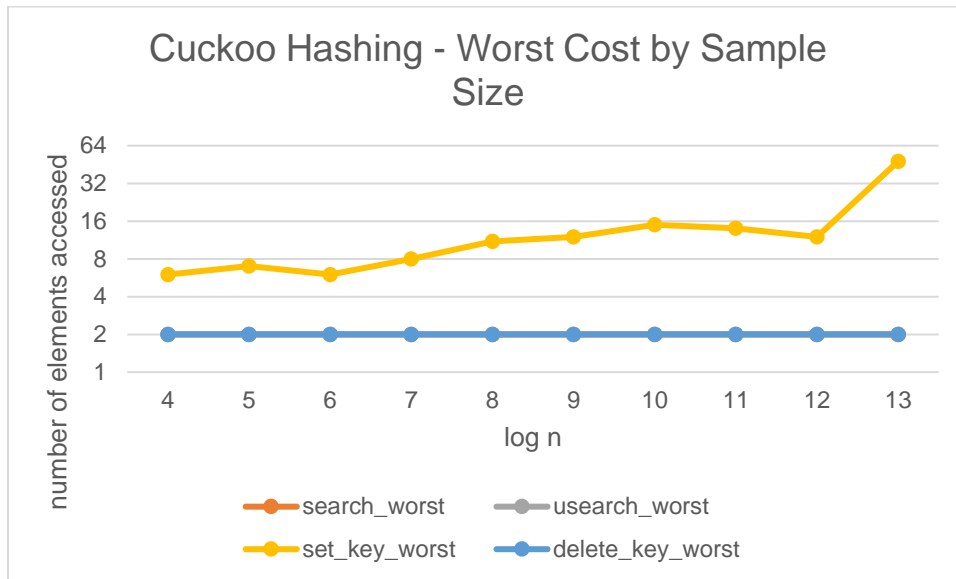


Generally, all operations see their costs rise with sample size. It is worth noticing that worst search time is below the theoretical bound of $\log n$. The “set_key_worst” line is invisible between $\log n = 4$ and 10 because it is completely the same as the “delete_key_worst” line. It makes sense. Setting a key hashed to the left end of the longest continuous block requires walking through this block. Deleting a key at the left end of the longest continuous block also requires walking through this block to check if any key needs to be moved back.

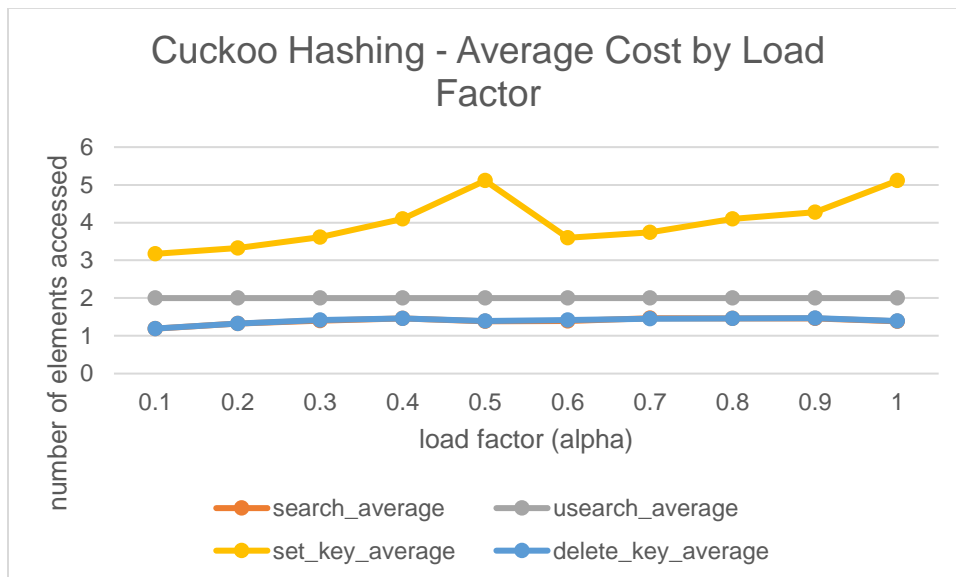


As we expected, delete is the most expensive operation. Besides, all operations see their costs rise with load factor. In particular, successful search cost matches its theoretical cost $\frac{1}{2}(1 + \frac{1}{1-\alpha})$ very well. So does unsuccessful search cost, whose line in the graph is completely covered by its theoretical cost line, $\frac{1}{2}(1 + \frac{1}{(1-\alpha)^2})$.

Cuckoo Hashing: (load factor = 0.3, n = 1024)

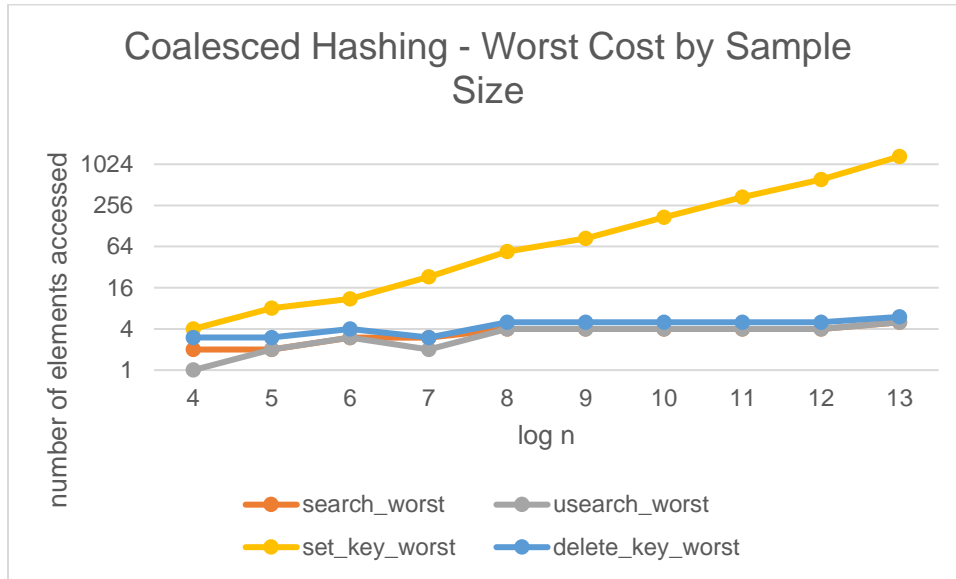


As we expected, all operations but set have worst cost no more than 2.

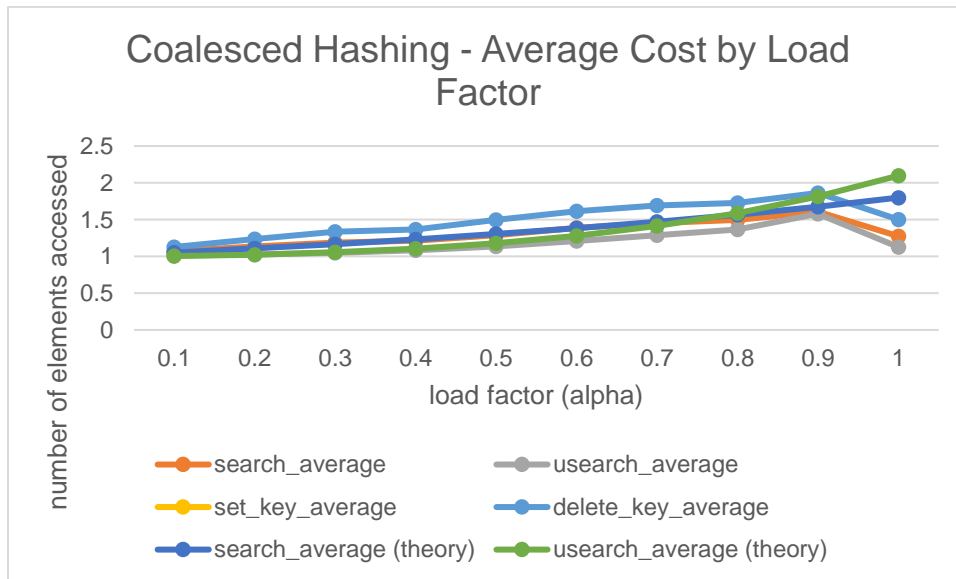


As we expected, setting a key not only costs at least 3 memory accesses but also is the most expensive operation of Cuckoo Hashing. The “search_average” line is invisible in this graph because it is almost the same as the “delete_key_average” line. We can see that increasing load factor seems to raise search cost, but to a very mild extent. It is worth noticing that the average set cost peaks at $\alpha = 0.5$ and $\alpha = 1$. Since our sample sizes are powers of 2, when $\alpha = 0.5$ and $\alpha = 1$, the parameter N in the hash function is also a power of 2. In this case, Cuckoo Hashing is particularly subject to collisions and hence table resizing, which seriously hurts its set operation’s performance.

Coalesced Hashing: (load factor = 0.3, n = 1024)



Our experiment is involved with a long sequence of deletions and insertions, which create sparse holes across the hash table. As we expected, in such an adverse situation, Coalesced Hashing performs poorly in set operation, whose cost grows linearly with sample size. When load factor is 0.3, on average it needs to walk through at most 15% of table size to find the next highest empty slot. Other than that, other operations' worst-case costs seem to be bounded by a constant and independent of sample size.

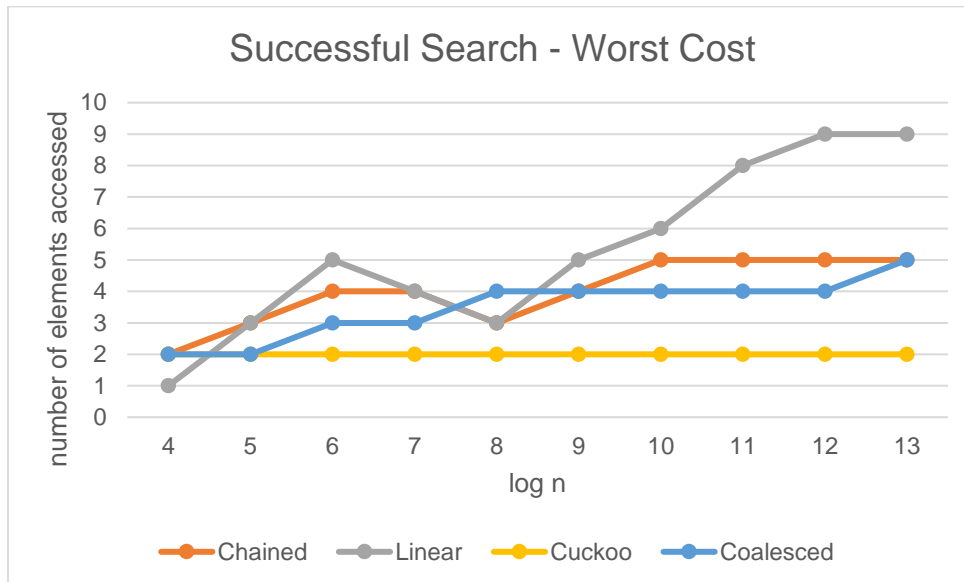


Since set operation has a far higher cost than other operations, we remove the “set_key_average” line from the figure to make other lines look clearer. As we expected, other operations' average costs slowly increase with load factor and are capped by 2. In particular, successful search cost matches its theoretical cost $1 + \frac{1}{8\alpha}(e^{2\alpha} - 1 - 2\alpha) + \frac{\alpha}{4}$ pretty well. Unsuccessful search cost is also quite close to its theoretical cost $1 + \frac{1}{4}(e^{2\alpha} - 1 - 2\alpha)$. It may look strange that when $\alpha = 1$, the actual average costs fall. It is because the sequence of deletions and insertions create tombstones and trigger table resizing, which reduces the load factor to 0.5. That is why the costs at $\alpha = 1$ plunge and are similar to those at $\alpha = 0.5$.

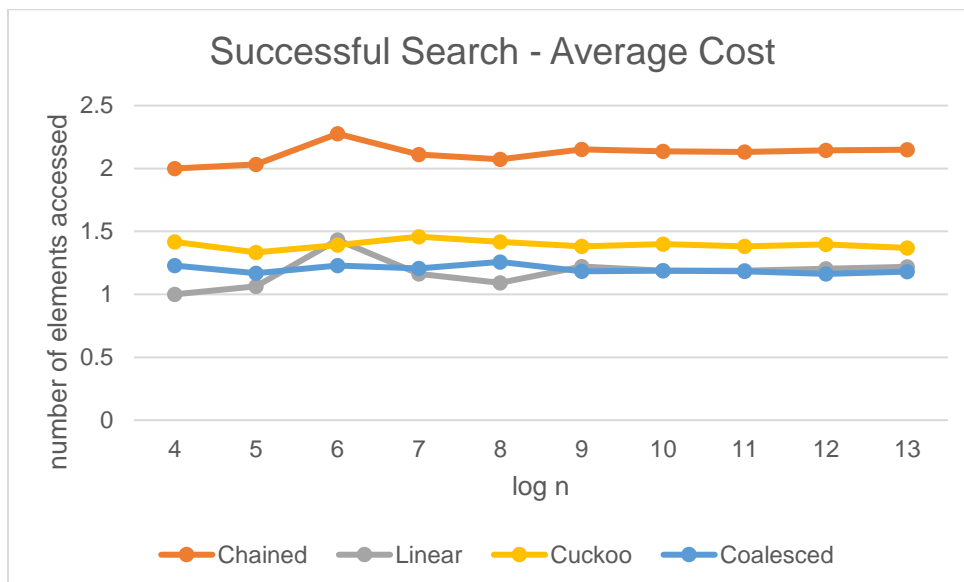
Dictionary of Stable Size

In subsequent parts we show the results of the hashing algorithms in the same plots. Our focus is on their relative performances as well as similarities/differences.

Successful Search: (load factor = 0.3)

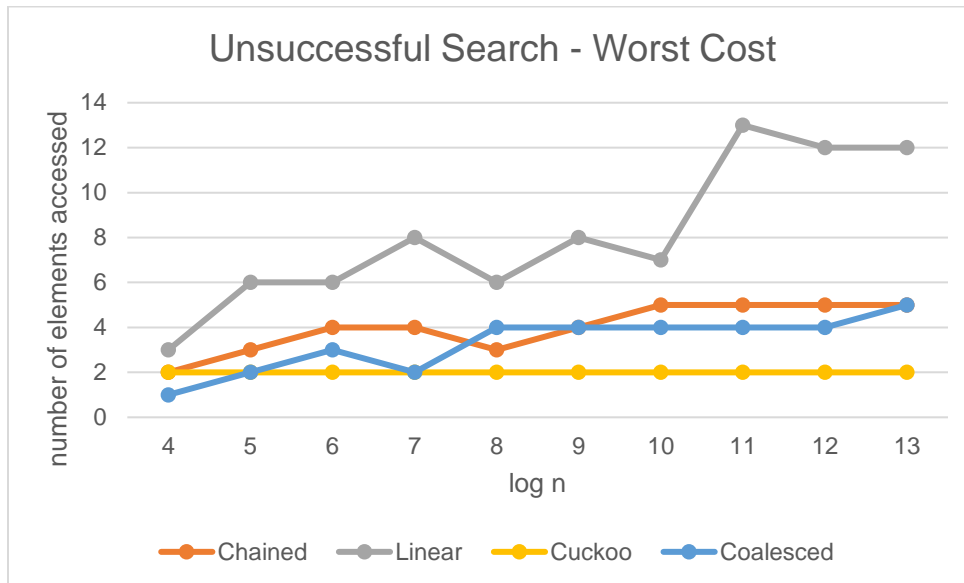


As we expected, generally Linear Hashing has the worst “worst-case” time, and only Cuckoo Hashing has constant worst-case time. Chained Hashing and Coalesced Hashing are in the middle, but the latter performs better than the former in most cases.

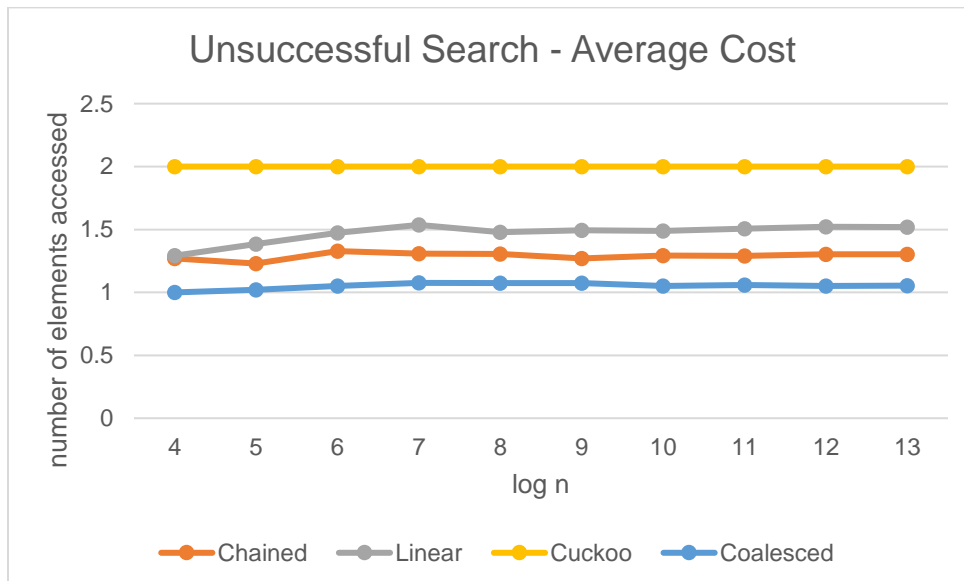


As expected, all hashing algorithms have constant average time. Chained Hashing is slightly over 2 because it needs at least 2 element accesses – one for the pointer to the linked list, the other for the entry per se. Other hashing methods have average search time between 1 and 1.5.

Unsuccessful Search: (load factor = 0.3)

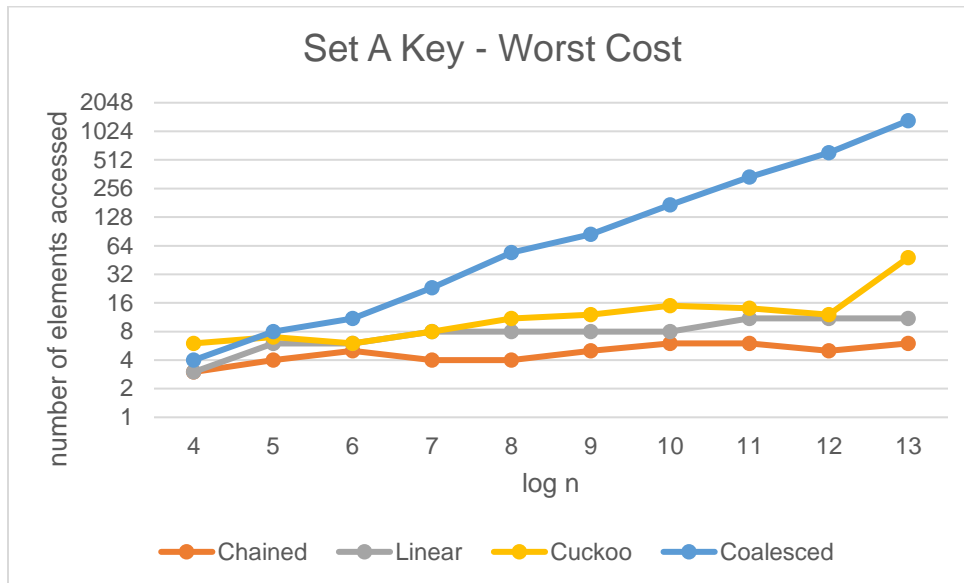


Still, Linear Hashing has the worst “worst-case” time. As expected, Cuckoo Hashing has constant worst-case time. Chained Hashing and Coalesced Hashing are in the middle, but the latter performs better than the former in most cases.

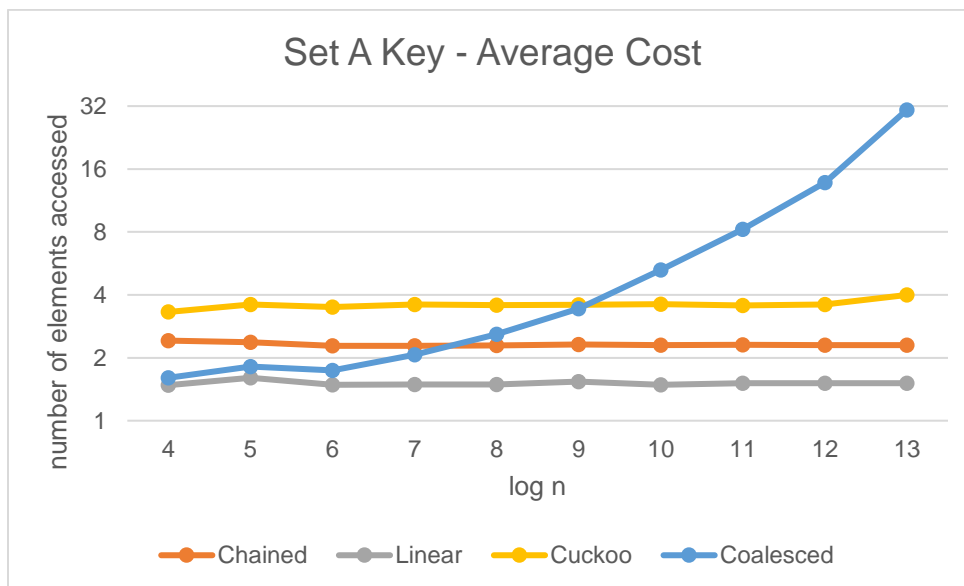


In contrast to successful search, Cuckoo Hashing performs worst in unsuccessful search. That is because in any cases, Cuckoo Hashing needs to check two elements before answering if a key exists. Other hashing methods have constant average time between 1 and 1.5. Interestingly, the hashing methods’ ranking in performance, Coalesced Hashing > Chained Hashing > Linear Hashing > Cuckoo Hashing, is very stable throughout the sample size axis. Coalesced Hashing wins this contest possibly because it stores the entry instead of the pointer (beat Chained Hashing) and it uses the null pointer instead of empty element as the termination condition of search (beat Linear Hashing).

Set A Key: (load factor = 0.3)

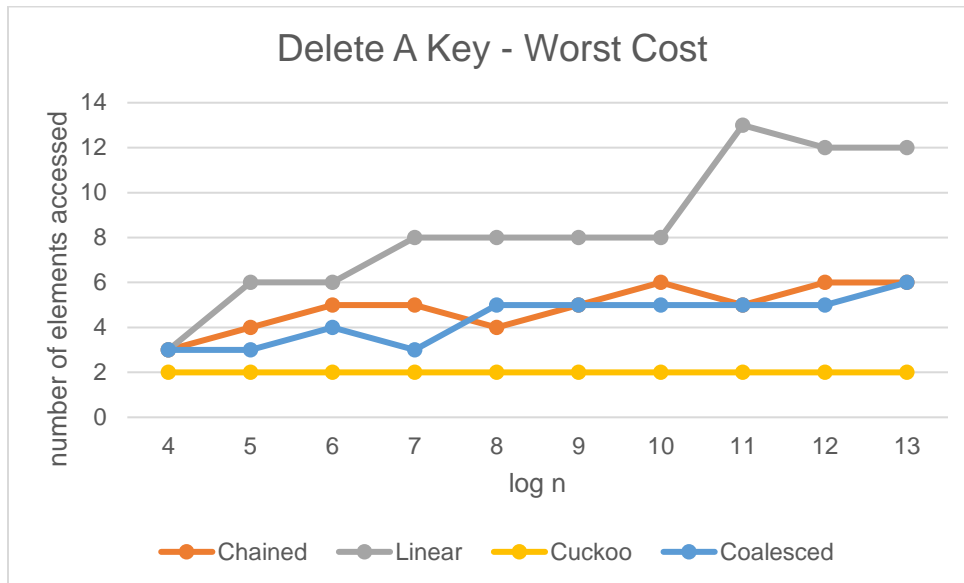


Coalesced Hashing performs astonishingly bad in this item. It is primarily due to the cost of updating the empty slot pointer. After a number of random deletions, empty slots become sparse and it may take some time to find the next highest-indexed empty slot. Cuckoo Hashing performs the second worst because it may need to resize hash table when there is a loop. Resizing hash tables costs a lot of element accesses. Chained Hashing is a big winner because it never worries about the two issues mentioned above.

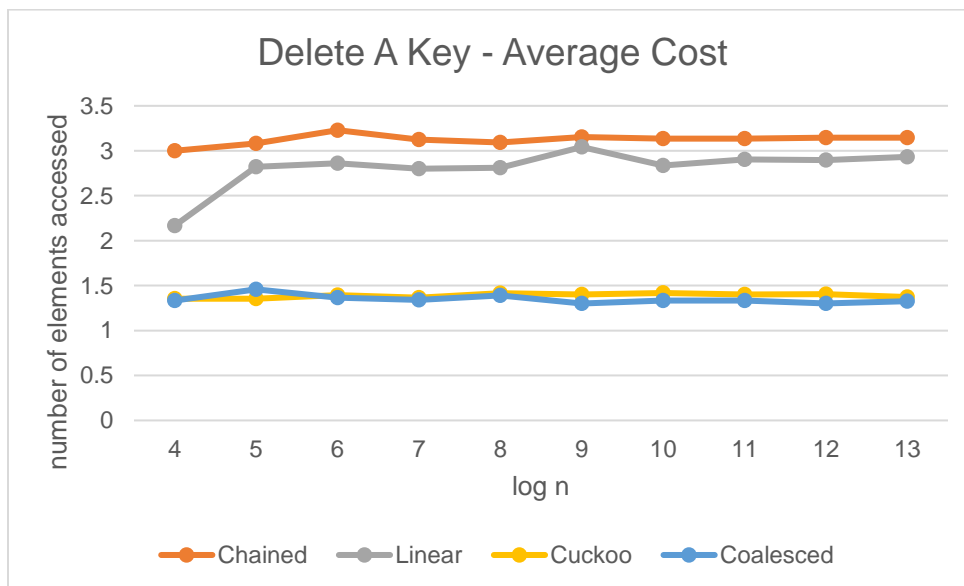


Similarly, Coalesced Hashing is the worst performer in this item. As expected, other hashing methods have constant average set time. When load factor is only 0.3, there is less clustering effect and Linear Hashing can easily find an empty slot to insert the new key. It is also an easy task for Chained Hashing, except that it has an extra memory access to the linked list pointer. Cuckoo Hashing lags behind because it needs to check two places before inserting a new key.

Delete A Key: (load factor = 0.3)



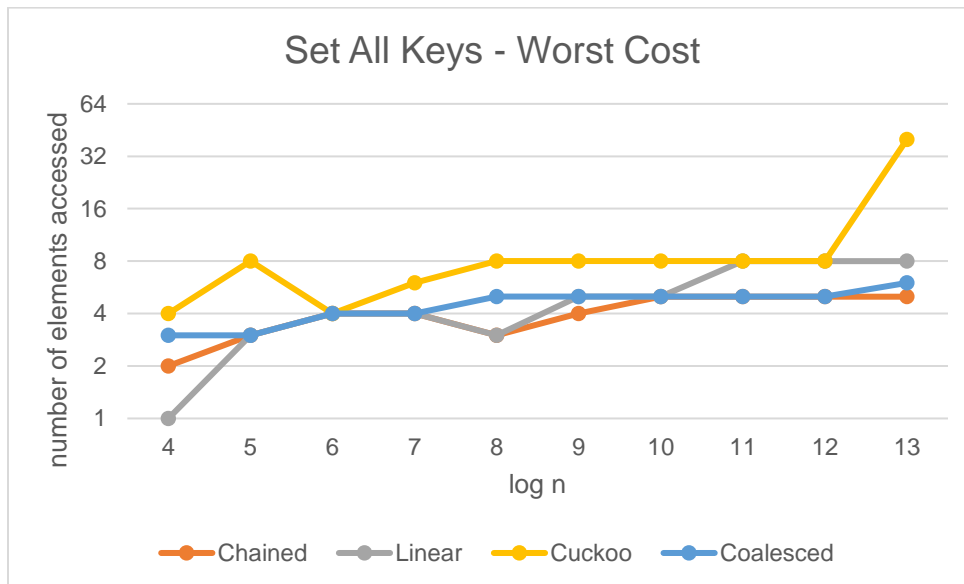
As expected, Linear Hashing has the worst “worst-case” delete time because after it deletes a key, it needs to move other keys around to maintain the invariant that keys hashed to the same value are connected. Cuckoo Hashing is a big winner in this contest because in any case, it only needs to check two elements. Chained Hashing and Coalesced Hashing also perform well because both use linked lists and only need constant time to delete a key. (Please note that Coalesced Hashing uses lazy delete in our experiment. If it used eager delete, it could be the worst performer.)



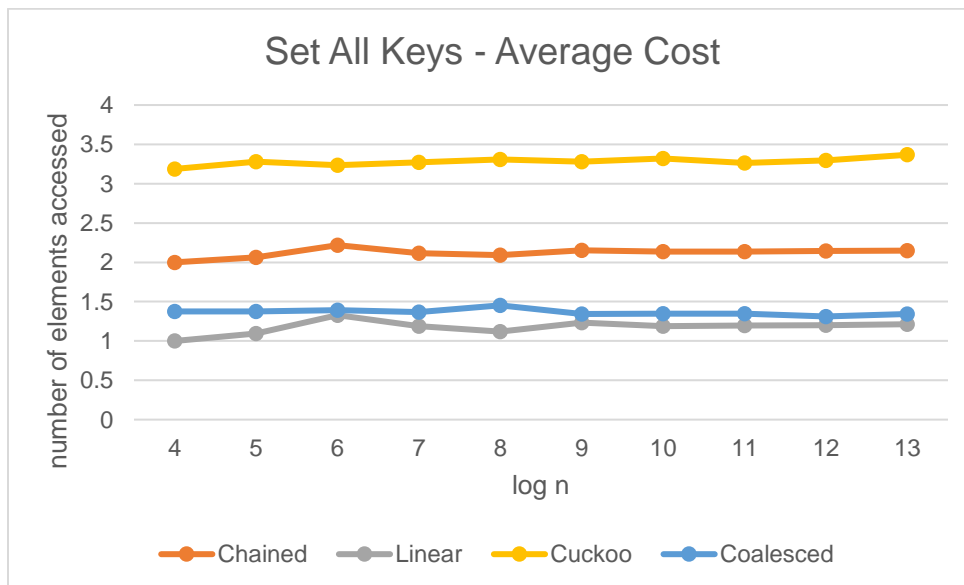
Two groups are discriminated in this graph: Cuckoo Hashing and Coalesced Hashing are on the top, while Chained Hashing and Linear Hashing are on the bottom. Surprisingly, Chained Hashing has worst average delete time even though Linear Hashing has far worse “worst-case” time. Why? We could still blame the extra memory access to the pointer. In any case, it takes Chained Hashing a memory access to the pointer, another to the key per se, and another to the previous element for updating the next pointer, which results in at least 3 memory accesses. As expected, all hashing methods have constant average delete time.

Growing and Shrinking Dictionaries

Set All Keys: (load factor = 0.3)

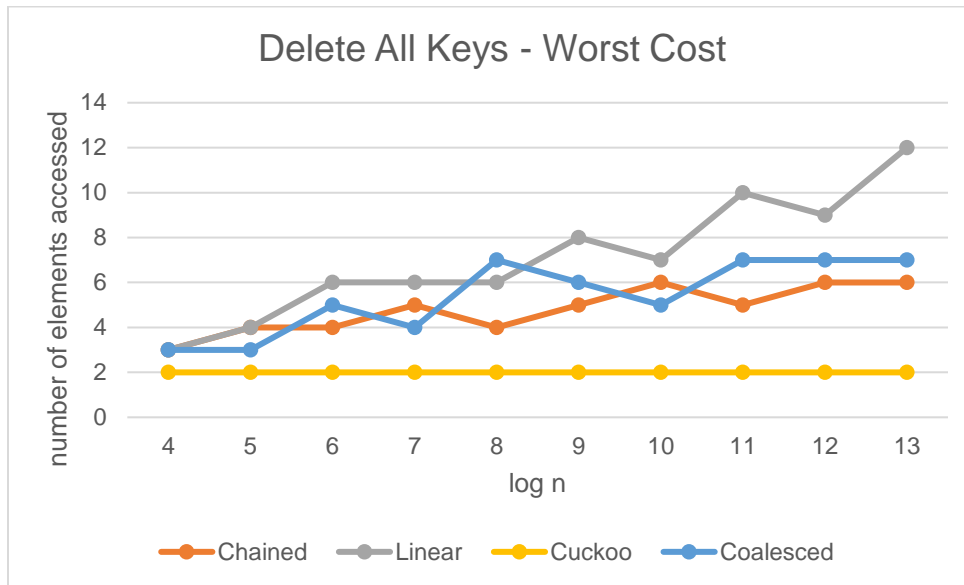


In this item, all hashing methods are entwined. However, it is still clear that Cuckoo Hashing is a loser due to the possibility of resizing to resolve a closed loop.

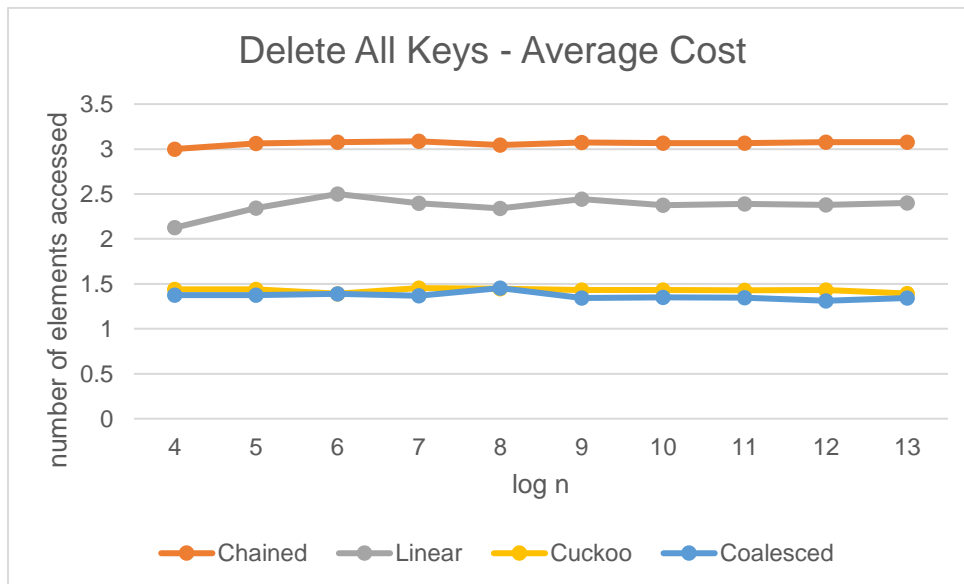


In this item, the ranking in performance is black and white clear: Linear Hashing > Coalesced Hashing > Chained Hashing > Cuckoo Hashing. Linear Hashing wins thanks to the low load factor. It takes Coalesced Hashing slightly more time because it needs to update the empty slot pointer. Without the memory access to the pointer, Chained Hashing's average set time would be very similar to that of Linear Hashing. Besides being haunted by infinite loops, Cuckoo Hashing also suffers from the fact that it needs to check two places before inserting a new key, so its set cost is at least 3 memory accesses.

Delete All Keys: (load factor = 0.3)



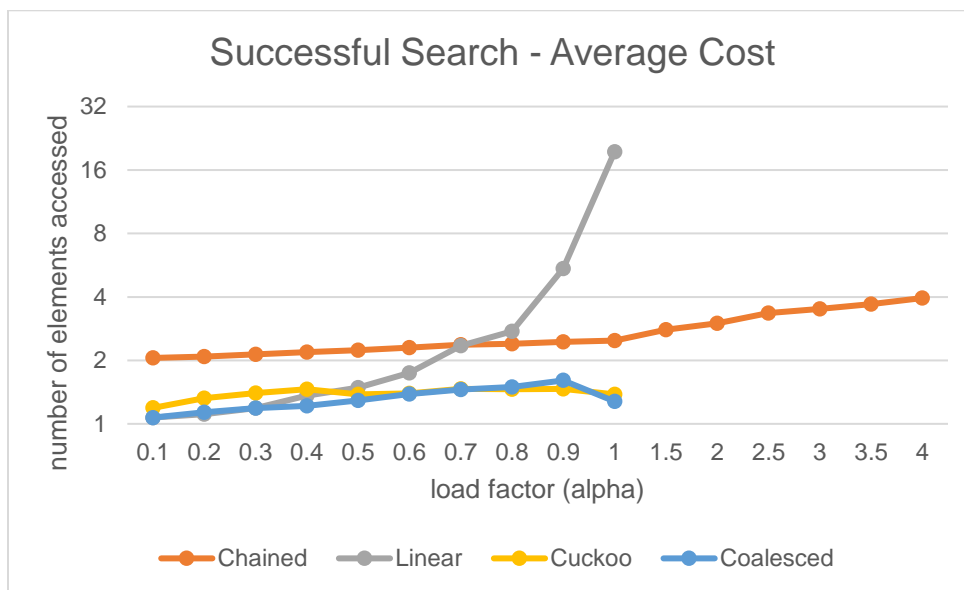
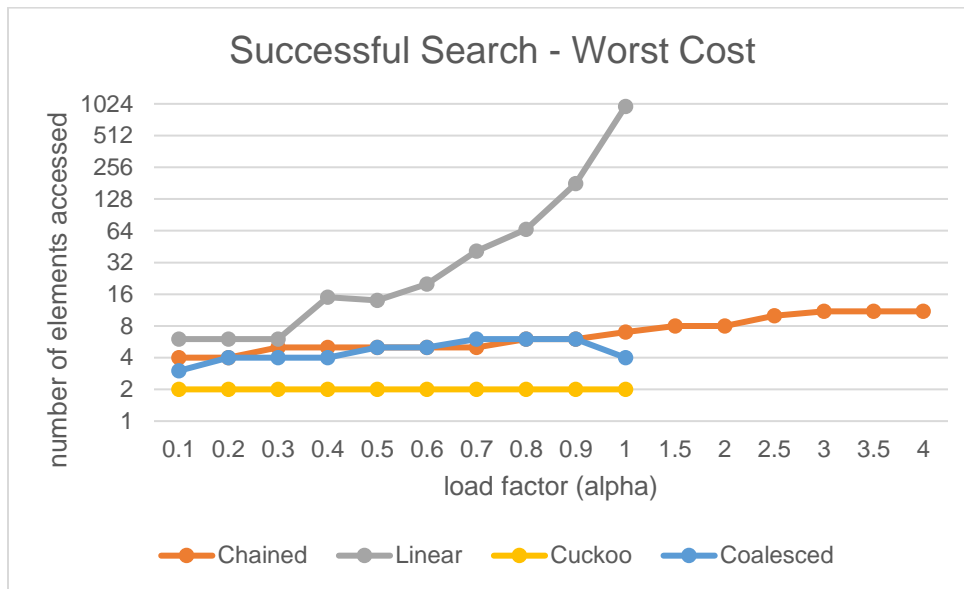
As with the case of deleting a single key, Linear Hashing has the worst “worst-case” time in this item. Although Coalesced Hashing uses lazy delete, it still has to walk through tombstones to find the key to delete. Deletion for Chained Hashing and Cuckoo Hashing is quite straightforward. They just need to find the key and delete it. Chained Hashing needs to walk through the linked list, while Cuckoo Hashing only needs to check two places, which helps Cuckoo Hashing win the contest.



In this item, Cuckoo Hashing and Coalesced Hashing are almost equally competitive. Since most of the keys are in the first hash table, Cuckoo Hashing usually can delete a key within one memory access. Coalesced Hashing only needs to walk through the linked list and place a deletion marker, so it is fast too. As we mentioned, it is awkward for Linear Hashing to delete a key, but as more keys are deleted, load factor becomes lower, which helps Linear Hashing escape the bottom. Unfortunately, Chained Hashing needs at least 3 memory accesses to delete a key regardless of load factor, so it is at the bottom. It is worth noticing that Coalesced Hashing is within the top in both building and destroying a hash table.

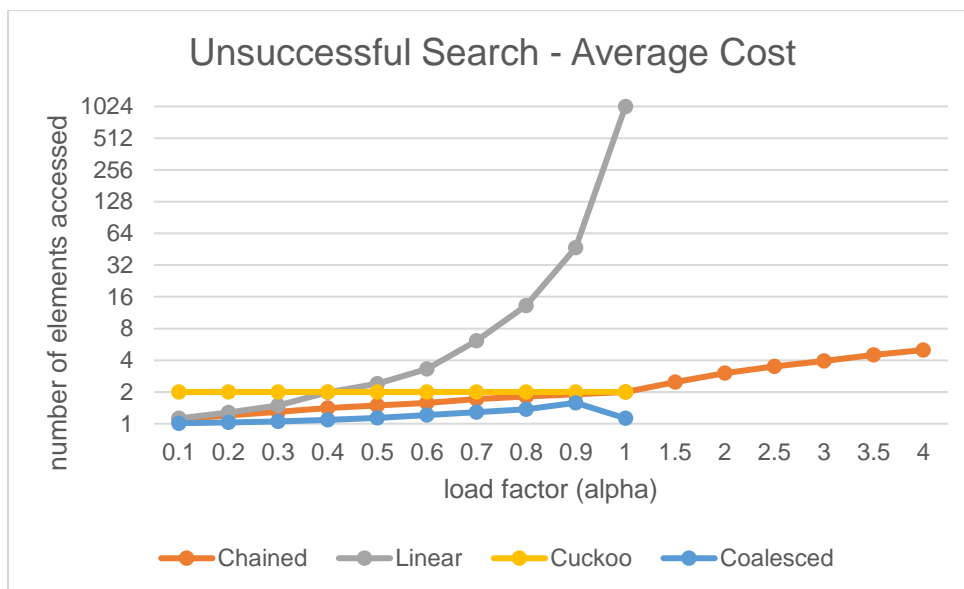
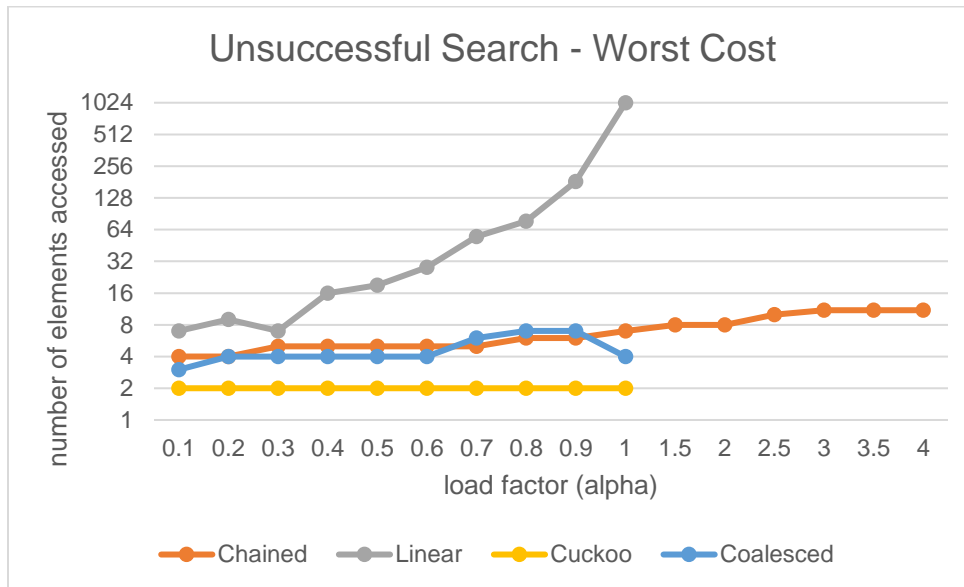
Running Time by Load Factor

Successful Search by Load Factor: (n = 1024)



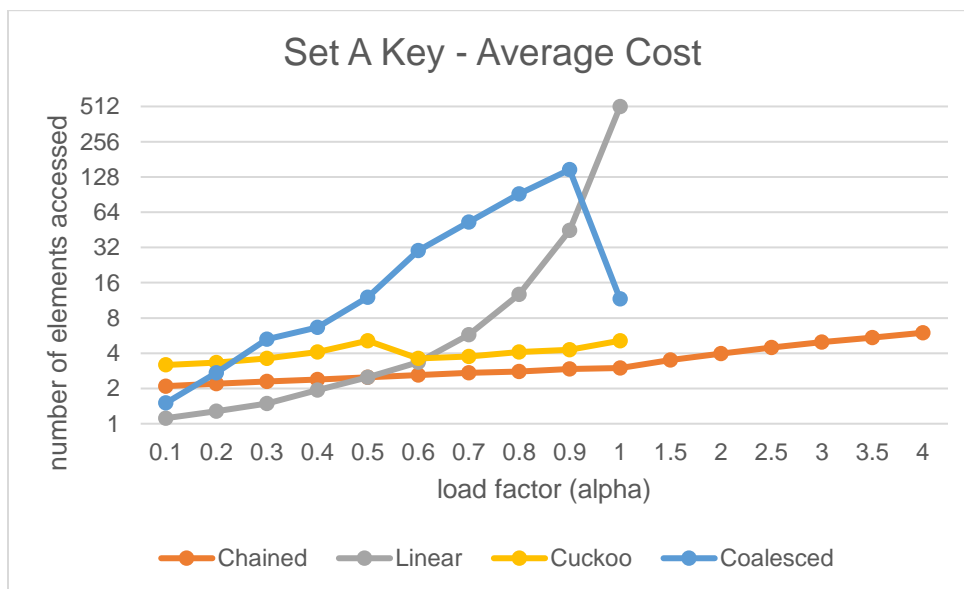
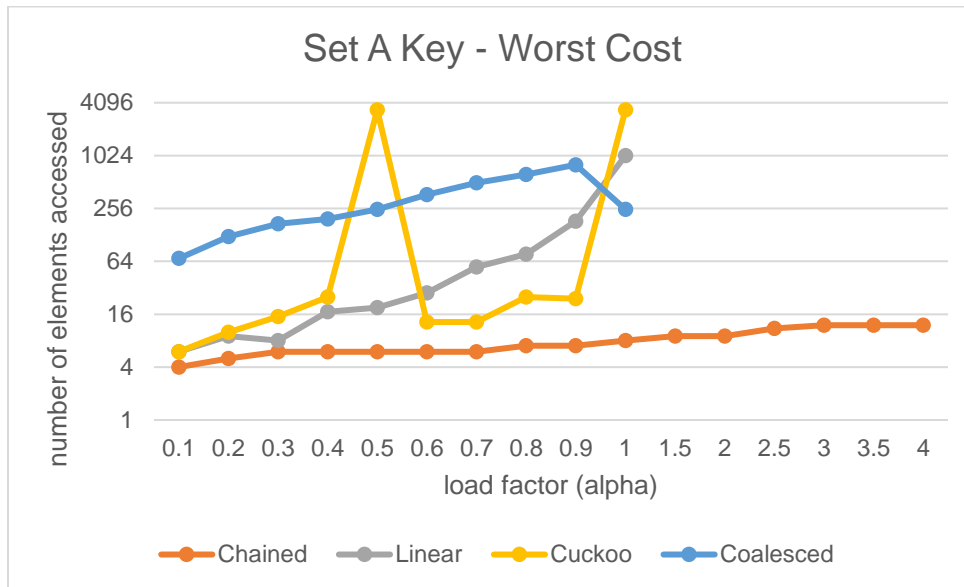
As expected, search cost for Linear Hashing increases exponentially after load factor exceeds 0.5. On the contrary, other hashing methods' search costs grow linearly and smoothly with load factor. It is worth noticing that Coalesced Hashing's average search cost is lower than both Chained Hashing and Linear Hashing under all kinds of load factors. It proves its superiority over these two basic hashing algorithms.

Unsuccessful Search by Load Factor: (n = 1024)



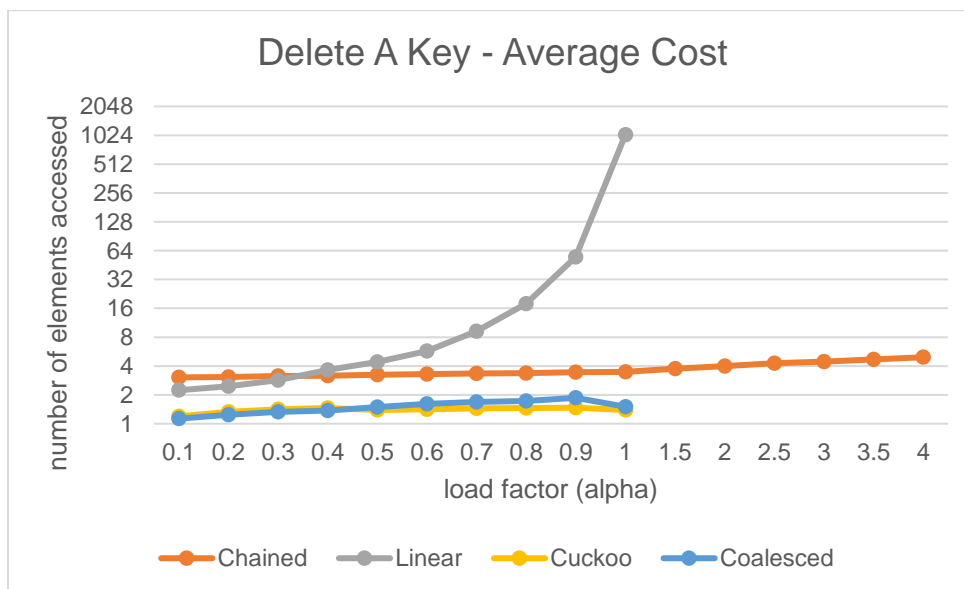
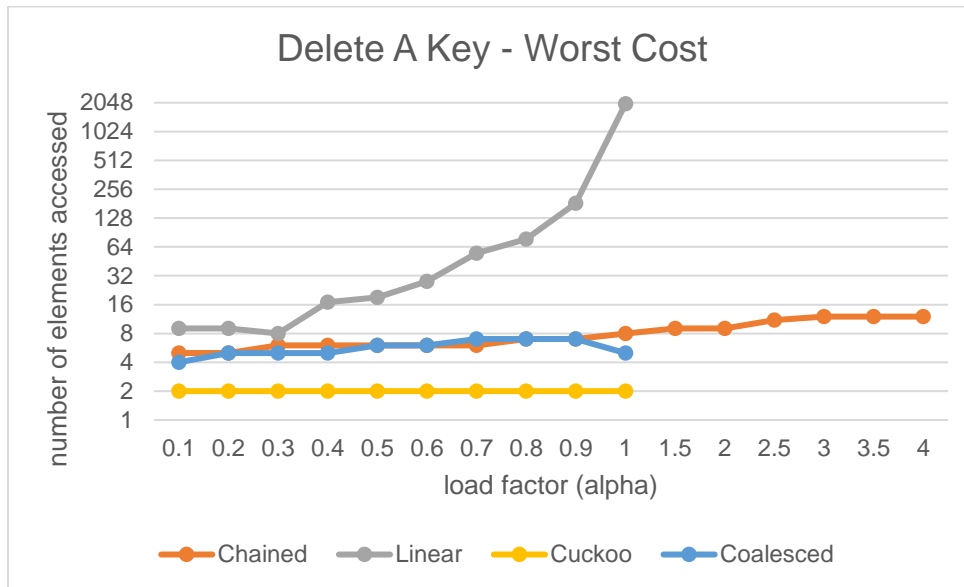
By the same token, Linear Hashing's performance deteriorates quickly as load factor approaches 1. In particular, when $\alpha = 1$, all entries are connected, so Linear Hashing has to scan the whole hash table to see if a key exists. In any case, Cuckoo Hashing needs two memory accesses to check the existence of a key, so its unsuccessful search cost is flat with regards to load factor. Although Chained Hashing and Coalesced Hashing's costs increase with load factor, they are still lower than that of Cuckoo Hashing when $\alpha < 1$.

Set A Key by Load Factor: (n = 1024)



For Chained Hashing and Linear Hashing, the plots are very similar to the previous ones. However, Cuckoo Hashing's worst-case time is quite volatile, while Coalesced Hashing has the worst average set time between 0.1 and 0.9. The former should blame the infinite loops, and the latter should impute the extra efforts to update the empty slot pointer.

Delete A Key by Load Factor: (n = 1024)



Linear Hashing's delete cost is also sensitive to load factor. As with the case of unsuccessful search, when $\alpha = 1$, after deleting a key, Linear Hashing needs to scan the whole hash table to check if any key has to be moved back. Other three hashing methods have relatively stable delete costs as load factor increases.

F. CONCLUSIONS

The following table summarizes the key cost features of each hashing algorithm:

	The Most Expensive Operation	Worst Search Cost	Average Search Cost
Chained Hashing	Delete	Slowly increase with n	Increase linearly with α
Linear Hashing	Delete	Increase with n	Increase exponentially with α
Cuckoo Hashing	Set	At most 2	Increase with α , but bounded by 2
Coalesced Hashing	Set	Seemingly bounded and insensitive to n	Increase with α , but bounded by 2

The following table summarizes our findings from the comparative empirical analysis:

	Advantages	Disadvantages	Suitable Situation
Chained Hashing	<ul style="list-style-type: none"> Can expand load factor more than 1 without resizing the hash table Do not have to spend time in finding an empty slot (in fact, the work is done by memory allocator) 	<ul style="list-style-type: none"> Need one extra memory access to the linked list pointer Need extra space to save the entries 	Number of keys is unknown and scalability of hash table is desired
Linear Hashing	<ul style="list-style-type: none"> Fastest key inserter when load factor is low 	<ul style="list-style-type: none"> Performance degrades exponentially with load factor When the hash table is full, unsuccessful search and deletion take $O(n)$ time 	Low load factor
Cuckoo Hashing	<ul style="list-style-type: none"> Worst search cost is constant 	<ul style="list-style-type: none"> Infinite loops result in table resizing before the hash tables are full It needs at least two memory accesses to confirm if a key exists in the hash tables 	Spontaneous lookup time is desired
Coalesced Hashing	<ul style="list-style-type: none"> Constant worst search cost regardless of load factor and sample size like Cuckoo Hashing while having no infinite loop problem 	<ul style="list-style-type: none"> Frequent random deletions create tombstones and sparse empty slots, which make the hash table under-loaded and updating the empty slot pointer expensive 	Random deletion is rare

Section B mentioned that Coalesced Hashing combines the benefits of the three basic hashing algorithms and should outperform all of them. Our empirical results corroborate this argument; thus, we think it is a good substitute for the three basic hashing algorithms.

- Compared with Chained Hashing: Its hash table stores entries instead of pointers, so on average it has one less memory access than Chained Hashing.
- Compared with Linear Hashing: Its performance is quite stable under different load factors and sample sizes, as opposed to Linear Hashing.
- Compared with Cuckoo Hashing: On average it only needs one memory access to confirm if a key already exists, while Cuckoo Hashing needs at least two. In addition, it is free of the infinite loop problem.

Nevertheless, Coalesced Hashing still has some weaknesses:

- Deletion is hard: Eager delete, if implemented, would be awkward and expensive, while lazy delete will generate tombstones and reduce the capacity of the hash table.
- Updating the empty slot pointer may be costly: If random deletion is frequent, then empty slots could be sparsely distributed throughout the hash table and difficult to locate.

Therefore, if random deletion is quite rare, like our case in which the Professor would like to preserve all students' records (unless a student drops out after the mid-term exam), Coalesced Hashing is a good choice. We would recommend using Coalesced Hashing to store UCI MCS students' CS261P mid-term exam scores (and final exam scores as well).

G. REMARKS

Some points not considered in this project include:

- Other factors which may affect running times: Our performance measure only considers the number of elements accessed. Other factors may also have influence on operation speed:
 - Caching effect: If the whole hash table can be placed in the cache, then the efficiency will improve.
 - Sequential access: If the elements are accessed in a sequential manner, it will be faster than random memory access.
 - Memory allocation: In Chained Hashing, every time it inserts a new key, a new memory space needs to be allocated, as opposed to static hash tables like Linear Hashing, Cuckoo Hashing, and Coalesced Hashing. The extra time is not included in our cost calculation.

Linear Hashing has strengths in all of the three aspects, so our results may underestimate its performance. Conversely, Chained Hashing performs poorly in the three aspects, so our results may overestimate its performance.

- Other input variables which may affect the number of elements accessed: For example, prime versus non-prime hash table sizes. Although we do not conduct a formal study of how prime versus non-prime hash table sizes affect the number of elements accessed, according to Cuckoo Hashing's results at $\alpha = 0.5$ and 1, we find that when N (size of an individual hash table) is a power of 2, Cuckoo Hashing is more likely to have infinite loops and need rehashing.

H. REFERENCES

1. Rasmus Pagh and Flemming Friche Rodler: "Cuckoo Hashing"
2. https://en.wikipedia.org/wiki/Coalesced_hashing
3. Mikhail J. Atallah and Marina Blanton: "Algorithms and Theory of Computation Handbook"
4. Lecture Slides and Assigned Readings