Fast Lookup: Hash tables

• Operations:
  – Find (key based look up)
  – Insert
  – Delete

• Consider the 2-sum problem: Given an unsorted array of N integers, find all pairs of elements that sum to a given number T.

1. Exhaustive Search: $O(N^2)$
2. Sort first: $O(N \log N)$. For each $x$, look for $(T-x)$ in the sorted array. Search for $(T-x) = O(N \log N)$
3. Hash Table: Insert $N$ elements into a hashtable $O(N)$. For each $x$ look up $(T-x) = O(N)$
Under the hood of hash tables

- Array based solution: In the 2-sum problem suppose that all the integers were positive and bounded by N, how could you implement a “look up table” using arrays.

Use an array of size \((N+1)\) indexed by the `key` in our original array.

Problem: The size of our hash table would be very large if the key values become very large. This results in a lot of unnecessary space wastage. Although inserting & finding keys would be \(O(1)\)
Fast Lookup: Hash functions

• Setup for hashing:
  – Universe of possible keys $U$ (Very large)
  – Keep track of evolving set $S \subseteq U$
  – $|S|$ is approximately known

Diagram:
- $U$ is the universe of keys.
- $h(x) = \text{index in the array}$ is the hash function that maps keys to indices in an array.
- The hash function distributes keys uniformly across the array's buckets.
- A random hash function is considered the gold standard because it distributes keys uniformly.

Notes:
- $|S|$ is approximately known, and $S \subseteq U$.
- The hash function is $h(x)$.
Probability of Collisions

• Suppose you have a hash table that can hold 100 elements. It currently stores 30 elements (in one of 30 possible different locations in the hash table). What is the probability that your next two inserts will cause at least one collision (assuming a totally random hash function)? (Choose the closest match)

\[ 1 - \text{Pr( no collisions) } \]

\[ = 1 - \frac{30 \times 69}{100 \times 100} \]

\[ = 0.52 \]

A. .09
B. .30
C. .52
D. .74
E. .90
Hashtable collisions and the "birthday paradox"

• Suppose there are 365 slots in the hash table: M=365
• What is the probability that there will be a collision when inserting N keys?
  • For N = 10, prob\text{\textsubscript{N,M}}(\text{collision}) = 12%
  • For N = 20, prob\text{\textsubscript{N,M}}(\text{collision}) = 41%
  • For N = 30, prob\text{\textsubscript{N,M}}(\text{collision}) = 71%
  • For N = 40, prob\text{\textsubscript{N,M}}(\text{collision}) = 89%
  • For N = 50, prob\text{\textsubscript{N,M}}(\text{collision}) = 97%
  • For N = 60, prob\text{\textsubscript{N,M}}(\text{collision}) = 99+% 

• So, among 60 randomly selected people, it is almost certain that at least one pair of them have the same birthday
• On average one pair of people will share a birthday in a group of about \( \sqrt{2 \cdot 365} \approx 27 \) people
• In general: collisions are likely to happen, unless the hash table is quite sparsely filled
• So, if you want to use hashing, can’t use perfect hashing because you don’t know the keys in advance, and don’t want to waste huge amounts of storage space, you have to have a strategy for dealing with collisions
The birthday collision "paradox"

<table>
<thead>
<tr>
<th>Jan</th>
<th>Feb</th>
<th>Mar</th>
<th>Apr</th>
<th>May</th>
<th>Jun</th>
<th>Jul</th>
<th>Aug</th>
<th>Sep</th>
<th>Oct</th>
<th>Nov</th>
<th>Dec</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Probability of Collisions

- If you have an empty hash table with $M$ slots, the probability that there is at least 1 collision when $N$ keys are inserted is:

$$P_{N,M}(\text{collision}) = 1 - P_{N,M}(\text{no collision})$$

$$= 1 - \prod_{i=1}^{N} P_{N,M}(\text{ith key no collision})$$

$$= 1 - \left(\frac{1}{M} \cdot \frac{(M-1)}{M} \cdot \frac{(M-2)}{M} \cdots \frac{m-(n-1)}{M}\right)$$
Making hashing work

• Important decisions when implementing hashing:
  • Hash function → \( H(x) \) → very bad
  • Size of the hash table → \( n \) → 1.3 \times \text{no. of keys that you expect}
  • Collision resolution strategy

• With a good hashtable design
  • Average-case insert and find operation time costs: \( O(1) \)
  • Average-case space cost per key stored: \( O(1) \)

• This makes hashtables a very useful and very commonly used data structure
Hash functions: desiderata

- Fast to compute
  
  \[ h(x) = O(1) \]
  
- Distribute keys as uniformly as possible in the hash table
  
  Avoid collisions
  
  Don't want \( h(x) \) that hashes to only a subset of indices in the hashtable

- A hash function should be consistent with the equality testing function
  
  - If two keys are equal, the hash function should map them to the same table location
    
    \[ x = y \quad h(x) = h(y) \]
Hash functions: desiderata

- Fast to compute

- Distribute keys as uniformly as possible in the hash table, to avoid collisions as much as possible
  - For example, you don’t want a hash function that will map the set of keys to only a subset of the locations in the table!
  - The hash function should "look like" it is distributing keys randomly to locations

- A hash function should be consistent with the equality testing function
  - If two keys are equal, the hash function should map them to the same table location
  - Otherwise, the fundamental hash table operations will not work correctly

- A good choice of hash function can depend on the type of keys, the distribution of key insert requests, and the size of the table; and it is difficult to design a good one
Quick and Dirty Hash functions for Integers: \( H(K) = K \mod M \)

- When is the function \( H(K) = K \mod M \) a good hash function for integer keys?
  A. Always
  B. When \( M \) is prime
  C. When \( M \) is even
  D. Never

\( M \) is the size of the hashtable:

- By "size" of the hash table we mean how many slots or buckets it has

If \( K \) and \( M \) share a common factor, \( K \mod M \) is always going to be a multiple of \( c \):

\[
K \mod M = w \times c
\]

For example:

\[
1000 \mod 100 = 0
\]

This index is inserted in the hash table:
Hash functions for integers: $H(K) = K \mod M$

- For general integer keys choose a table of size $M$ (prime):
  - a good fast general purpose hash function is $H(K) = K \mod M$

- If table size is not a prime, this may not work well for some key distributions
  - for example, suppose table size is an even number; and keys happen to be all even, or all odd. Then only half the table locations will be hit by this hash function

- So, use prime-sized tables with $H(K) = K \mod M$
Hash table size

- Factors that affect our choice of the hash table size
  - Expected number of keys $|S|$
  - The hash function
  - The collision resolution strategy and load factor

The load factor is computed as:

$$\alpha = \frac{N}{M}$$

where $M = \text{size of the table}$
$N = \text{number of keys that have been inserted in the table}$

If $\alpha \uparrow$, rehash with a larger $M$
Hash table size

Q: If you know you will have a maximum of 100 elements, which is the best choice of hash table size of the following?

A. 100  
   
B. 128  
   
C. 131  
   
D. 200  
   
   1.3x 100

   small

   not prime

   not prime
Hash table size

• By "size" of the hash table we mean how many slots or buckets it has

• Choice of hash table size depends in part on choice of hash function, and collision resolution strategy

• But a good general “rule of thumb” is:
  • The hash table should be an array with length about 1.3 times the maximum number of keys that will actually be in the table, and
  • Size of hash table array should be a prime number

• So, let M = the next prime larger than 1.3 times the number of keys you will want to store in the table, and create the table as an array of length M

• (If you underestimate the number of keys, you may have to create a larger table and rehash the entries when it gets too full; if you overestimate the number of keys, you will be wasting some space)
Using a hash function

- using the hash function $H(K) = K \mod M$, insert these integer keys:
  
  $701 \ (1), \ 145 \ (5), \ 217 \ (0), \ 19 \ (5), \ 13 \ (6), \ 749 \ (0)$

  in this table:

<table>
<thead>
<tr>
<th>index:</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>217</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>701</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>145</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Collison
Resolving Collisions: Separate Chaining

• using the hash function $H(K) = K \mod M$, insert these integer keys:

$701 \ (1), \ 145 \ (5), \ 217 \ (0), \ 19 \ (5), \ 13 \ (6), \ 749 \ (0)$

in this table:

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

index: 0 1 2 3 4 5 6

19

Is there an upper bound to the number of elements that we can insert into a hashtable with separate chaining?

A. Yes, because of space constraints in the array
B. No, but inserting too many elements can affect the run time of insert
C. No, but inserting too many elements can affect the run time of find
Resolving Collisions: Linear Probing

- using the hash function \( H(K) = K \mod M \), insert these integer keys:

  701 (1), 145 (5), 217 (0), 19 (5), 13 (6), 749 (0)

in this table:

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>217</td>
<td>701</td>
<td>13</td>
<td>749</td>
<td>145</td>
<td>19</td>
<td></td>
</tr>
</tbody>
</table>

Find: 19, 16

Is linear probing more space efficient than separate chaining?
A. Yes  
B. No
Resolving Collisions: Double hashing

- A sequence of possible positions to insert an element are produced using two hash functions
- \( h_1(x) \): to determine the position to insert in the array, \( h_2(x) \): the offset from that position
- 701 (1, 2), 145 (5, 4), 217 (0, 3), 19 (5, 3), 13 (6, 2), 749 (0, 2)

In this table:

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>217</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>19</td>
<td>145</td>
<td></td>
</tr>
<tr>
<td>701</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ h_1(x), h_1(x) + h_2(x), h_1(x) + 2h_2(x) \]

If \( h_1(x) \) is pseudorandom, \( h_2(x) \) is seed.
Resolving Collisions: Randomhashing

• Same as double hashing but the two hash values are produced by a pseudo random number generator

• \( h_1(x) \): to determine the position to insert in the array, \( h_2(x) \): the offset from that position

701 (1, 2), 145 (5,4), 217 (0,3), 19 (5, 3), 13 (6,2), 749 (0,2)

in this table:

<table>
<thead>
<tr>
<th></th>
<th>217</th>
<th>701</th>
<th></th>
<th></th>
<th></th>
<th>145</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
</tr>
</tbody>
</table>
Collision resolution strategies

• The strategy has to permit find, insert, and delete operations that work correctly!

• Collision resolution strategies we will look at are:
  • Linear probing
  • Double hashing
  • Random hashing
  • Separate chaining

open addressing
Linear probing: inserting a key

• When inserting a key K in a table of size M, with hash function H(K)

  1. Set indx = H(K)
  2. If table location indx already contains the key, no need to insert it. Done!
  3. Else if table location indx is empty, insert key there. Done!
  4. Else collision. Set indx = (indx + 1) mod M.
  5. If indx == H(K), table is full! (Throw an exception, or enlarge table.) Else go to 2.

• So, linear probing basically does a linear search for an empty slot when there is a collision

• Advantages: easy to implement; always finds a location if there is one; very good average-case performance when the table is not very full

• Disadvantages: “clusters” or "clumps" of keys form in adjacent slots in the table; when these clusters fill most of the array, performance deteriorates badly as the probe sequence performs what is essentially an exhaustive search of most of the array
Linear probing: searching for a key

• If keys are inserted in the table using linear probing, linear probing will find them!

• When searching for a key $K$ in a table of size $N$, with hash function $H(K)$:

1. Set $\text{indx} = H(K)$
2. If table location $\text{indx}$ contains the key, return FOUND.
3. Else if table location $\text{indx}$ is empty, return NOT FOUND.
4. Else set $\text{indx} = (\text{indx} + 1) \mod M$.
5. If $\text{indx} = \equiv H(K)$, return NOT FOUND. Else go to 2.

• Question: How to delete a key from a table that is using linear probing?
• Could you do "lazy deletion", and just mark the deleted key’s slot as empty? Why or why not?
Double hashing

• Linear probing collision resolution leads to clusters in the table, because if two keys collide, the next position probed will be the same for both of them.

• The idea of double hashing: Make the offset to the next position probed depend on the key value, so it can be different for different keys; this can reduce clustering
  • Need to introduce a second hash function $H_2(K)$, which is used as the offset in the probe sequence (think of linear probing as double hashing with $H_2(K) == 1$)
  • For a hash table of size $M$, $H_2(K)$ should have values in the range 1 through $M-1$; if $M$ is prime, one common choice is $H2(K) = 1 + ( (K/M) \mod (M-1) )$

• The insert algorithm for double hashing is then:

  1. Set $indx = H(K); offset = H_2(K)$
  2. If table location $indx$ already contains the key, no need to insert it. Done!
  3. Else if table location $indx$ is empty, insert key there. Done!
  4. Else collision. Set $indx = (indx + offset) \mod M$.
  5. If $indx == H(K)$, table is full! (Throw an exception, or enlarge table.) Else go to 2.

• With prime table size, double hashing works very well in practice
Random hashing

• As with double hashing, random hashing avoids clustering by making the probe sequence depend on the key

• With random hashing, the probe sequence is generated by the output of a pseudorandom number generator seeded by the key (possibly together with another seed component that is the same for every key, but is different for different tables)

• The insert algorithm for random hashing is then:

  1. Create RNG seeded with K. Set \( \text{indx} = \text{RNG.next()} \mod M \).
  2. If table location \( \text{indx} \) already contains the key, no need to insert it. Done!
  3. Else if table location \( \text{indx} \) is empty, insert key there. Done!
  4. Else collision. Set \( \text{indx} = \text{RNG.next()} \mod M \).
  5. If all \( M \) locations have been probed, give up. Else, go to 2.

• Random hashing is easy to analyze, but because of the "expense" of random number generation, it is not often used; double hashing works about as well
Open addressing vs. separate chaining

• Linear probing, double and random hashing are appropriate if the keys are kept as entries in the hashtable itself...
  • doing that is called "open addressing"
  • it is also called "closed hashing"

• Another idea: Entries in the hashtable are just pointers to the head of a linked list ("chain"); elements of the linked list contain the keys...
  • this is called "separate chaining"
  • it is also called "open hashing"

• Collision resolution becomes easy with separate chaining: just insert a key in its linked list if it is not already there
  • (It is possible to use fancier data structures than linked lists for this; but linked lists work very well in the average case, as we will see)

• Next time: we’ll look at analyzing the time costs of these strategies
Analysis of open-addressing hashing

• A useful parameter when analyzing hash table Find or Insert performance is the load factor

\[ \alpha = \frac{N}{M} \]

where \( M \) = size of the table  
\( N \) = number of keys that have been inserted in the table

• The load factor is a measure of how full the table is

• Given a load factor \( \alpha \), we would like to know the time costs, in the best, average, and worst case of
  • new-key insert and unsuccessful find (these are the same)
  • successful find

• The best case is \( O(1) \) and worst case is \( O(N) \) for all of these... so let’s analyze the average case

• We will give results for random hashing and linear probing. In practice, double hashing is similar to random hashing
Average case unsuccessful find / insertion cost

- Assume $\alpha = N/M$
- Consider random hashing, so clustering is not a problem; each probe location is generated randomly, and independently
- With each probe, the probability of finding an empty location is $(1 - \alpha)$. Finding an empty location stops the find or insertion
- This is a Bernoulli process, with probability of success $(1 - \alpha)$. The expected first-order interarrival time of such a process is $1/(1 - \alpha)$. So:
- The average number of probes for insert or unsuccessful find with random hashing is

  $$U_{\alpha} = \frac{1}{1 - \alpha}$$

- With linear probing, probe locations are not independent; clusters form, which leads to long probe sequences when load factor is high. It can be shown that the average number of probes for insert or unsuccessful find with linear probing is approximately

  $$U_{\alpha} \approx \frac{1}{2} \left( 1 + \frac{1}{(1 - \alpha)^2} \right)$$

- These average case time costs are bad, bounded only by $M$, when $\alpha$ is close to 1; but are not too bad (4 and 8.5 respectively) when $\alpha \leq 0.75$, independent of $M$
Average case successful find cost

- Assume a table with load factor $\alpha = N/M$. Consider random hashing, so clustering is not a problem; each probe location is generated randomly, and independently.
- For a key in the table, the number of probes required to successfully find it is equal to the number of probes taken when it was inserted in the table.
- The insertion of each new key increases the load factor, starting from 0 and going to $\alpha$.
- Therefore, the average number of probes for successful find with random hashing is

$$S_{\alpha} = \frac{1}{N} \sum_{i=0}^{N-1} \frac{1}{1 - i / M} \approx \frac{1}{\alpha} \int_{0}^{\alpha} \frac{1}{1 - x} dx = \frac{1}{\alpha} \ln \frac{1}{1 - \alpha}$$

- With linear probing, clusters form, which leads to longer probe sequences. It can be shown that the average number of probes for successful find with linear probing is

$$U_{\alpha} \approx \frac{1}{2} \left( 1 + \frac{1}{(1 - \alpha)} \right)$$

- These average case time costs are bad, bounded only by $M$, when $\alpha$ is close to 1; but are good (1.8 and 2.5 respectively) when $\alpha \leq 0.75$, independent of $M$. 

\[ Average\ case\ successful\ find\ cost\]

\[ S_{\alpha} = \frac{1}{N} \sum_{i=0}^{N-1} \frac{1}{1 - i / M} \approx \frac{1}{\alpha} \int_{0}^{\alpha} \frac{1}{1 - x} dx = \frac{1}{\alpha} ln \frac{1}{1 - \alpha} \]

\[ U_{\alpha} \approx \frac{1}{2} \left( 1 + \frac{1}{(1 - \alpha)} \right) \]

\[ These\ average\ case\ time\ costs\ are\ bad,\ bounded\ only\ by\ M,\ when\ \alpha\ is\ close\ to\ 1;\ but\ are\ good\ (1.8\ and\ 2.5\ respectively)\ when\ \alpha \leq 0.75,\ independent\ of\ M. \]
Average case successful find cost

- With linear probing, clusters form, which leads to longer probe sequences. It can be shown that the average number of probes for successful find with linear probing is

\[ U_\alpha \approx \frac{1}{2} \left( 1 + \frac{1}{(1 - \alpha)} \right) \]

- These average case time costs are bad, bounded only by M, when \( \alpha \) is close to 1; but are good (1.8 and 2.5 respectively) when \( \alpha \leq 0.75 \), independent of M.
Dependence of average performance on load
Analysis of separate-chaining hashing

- Keep in mind the load factor measure of how full the table is:

\[ \alpha = \frac{N}{M} \]

where \( M \) is the size of the table, and \( N \) is the number of keys that have been inserted in the table.

- With separate chaining, it is possible to have \( \alpha > 1 \)

- Given a load factor \( \alpha \), we would like to know the time costs, in the best, average, and worst case of
  - new-key insert and unsuccessful find (these are the same)
  - successful find

- The best case is \( O(1) \) and worst case is \( O(N) \) for all of these... let’s analyze the average case.
Average case costs with separate chaining

- Assume a table with load factor $\alpha = \frac{N}{M}$
- There are $N$ items total distributed over $M$ linked lists (some of which may be empty), so the average number of items per linked list is:
  
  A. $N$
  B. $M$
  C. $\frac{N}{M}$
  D. $N^2$
  E. Other
Average case costs with separate chaining

- Assume a table with load factor $\alpha = N/M$
- There are $N$ items total distributed over $M$ linked lists (some of which may be empty), so the average number of items per linked list is:

- In unsuccessful find/insert, one of the linked lists in the table must be exhaustively searched, and the average length of a linked list in the table is $\alpha$. So the average number of comparisons for insert or unsuccessful find with separate chaining is

  $$ U_\alpha = \alpha $$

- In successful find, the linked list containing the target key will be searched. There are on average $(N-1)/M$ keys in that list besides the target key; on average half of them will be searched before finding the target. So the average number of comparisons for successful find with separate chaining is

  $$ S_\alpha = 1 + \frac{(N-1)}{2M} \approx 1 + \frac{\alpha}{2} $$

- These are less than 1 and 1.5 respectively, when $\alpha < 1$
- And these remain $O(1)$, independent of $N$, even when $\alpha$ exceeds 1.
Hashtables vs. balanced search trees

- Hashtables and balanced search trees can both be used in applications that need fast insert and find

- What are advantages and disadvantages of each?
Hashtables vs. balanced search trees

- Hashtables and balanced search trees can both be used in applications that need fast insert and find

- What are advantages and disadvantages of each?

  - Balanced search trees guarantee worst-case performance $O(\log N)$, which is quite good
  - A well-designed hash table has typical performance $O(1)$, which is excellent; but worst-case is $O(N)$, which is bad

- Search trees require that keys be well-ordered: For any keys $K1$, $K2$, either $K1<K2$, $K1=K2$, or $K1> K2$
  - So, there needs to be an ordering function that can be applied to the keys
  - Hashtables only require that keys be testable for equality, and that a hash function be computable for them
  - So, there need to be an equality test function and a hash function that can be applied to the keys
Hashtables vs. balanced search trees, cont’d

• A search tree can easily be used to return keys close in value to a given key, or to return the smallest key in the tree, or to output the keys in sorted order
• A hashtable does not normally deal with ordering information efficiently

• In a balanced search tree, delete is as efficient as insert
• In a hashtable that uses open addressing, delete can be inefficient, and somewhat tricky to implement (easy with separate chaining though)

• Overall, balanced search trees are rather difficult to implement correctly
• Hash tables are relatively easy to implement, though they depend on a good hash function for good performance
More advanced topics on hashing

• We know that pathological data exists for any given hash function
• If the hash function is too simple, it’s susceptible to intruder attacks
• To deal with intruder attacks more advanced hash functions are
  – Cryptographic hashing
  – Universal hashing
Final Exam...

- 4 Parts
- Part 1: Basic knowledge of data structures and C++
  - 20% to 30% of final score
  - Multiple choice
- Part 2: Application, Comparison and Implementation of the data structures we have covered
  - 20% to 30% of final score
  - Short answers
- Part 3: Simulating algorithms and run time analysis
  - 20% to 30% of final score
  - Short answers
- Part 4: C++ and programming assignments