Hash Table

1. Introduction

Hash Table is a data structure to map key to values (also called Table or Map Abstract Data Type/ADT). It uses a **hash function** to map large or even non-Integer keys into a small range of Integer indices (typically [0..hash_table_size-1]).

The probability of two distinct keys colliding into the same index is relatively high and each of this potential collision needs to be resolved to maintain data integrity.

There are several collision resolution strategies that will be highlighted in this visualization: Open Addressing (Linear Probing, Quadratic Probing, and Double Hashing) and Closed Addressing (Separate Chaining). Try clicking Search(8) for a sample animation of searching a value in a Hash Table using Separate Chaining technique.

2. Motivation

Hashing is an algorithm (via a hash function) that maps large data sets of variable length, called keys, not necessarily Integers, into smaller Integer data sets of a fixed length.

A Hash Table is a data structure that uses a hash function to efficiently map keys to values (Table or Map ADT), for efficient search/retrieval, insertion, and/or removals.

Hash Table is widely used in many kinds of computer software, particularly for associative arrays, database indexing, caches, and sets.

In this e-Lecture, we will digress to Table ADT, the basic ideas of Hashing, the discussion of Hash Functions before going into the details of Hash Table data structure itself.

2-1. Table ADT

A Table ADT must support **at least** the following three operations as efficient as possible:

1. Search(v) — determine if **v** exists in the ADT or not,
2. Insert(v) — insert **v** into the ADT,

3. Remove(v) — remove **v** from the ADT.

Hash Table is one possible good implementation for this Table ADT (the other one is this).

PS1: For two weaker implementations of Table ADT, you can click the respective link: unsorted array or a sorted array to read the detailed discussions.

PS2: In live class, you may want to compare the requirements of Table ADT vs List ADT.

2-2. Direct Addressing Table (DAT)

When the range of the **Integer** keys is **small**, e.g., [0..**M**-1], we can use an initially empty (Boolean) array **A** of size **M** and implement the following Table ADT operations **directly**:

1. Search(v): Check if **A[v]** is true (filled) or false (empty),
2. Insert(v): Set **A[v]** to be true (filled),
3. Remove(v): Set **A[v]** to be false (empty).

That's it, we use the small Integer key itself to determine the address in array **A**, hence the name **Direct Addressing**. It is clear that all three major Table ADT operations are O(**1**).

PS: This idea is also used elsewhere, e.g., in Counting Sort.

2-3. Example of DAT

In Singapore (as of Sep 2021), bus routes are numbered from [2..991].

Not all integers between [2..991] are currently used, e.g., there is no bus route 989 — Search(989) should return false. A new bus route **x** may be introduced, i.e., Insert(**x**) or an existing bus route **y** may be discontinued, i.e., Remove(**y**).

As the range of possible bus routes is **small**, to record the data whether a bus route number exists or not, we can use a DAT with a Boolean array of size 1000.

Discussion: In real life class, we may discuss on why we use 1000 instead of 991 (or 992).

2-4. Example of DAT with Satellite Data

Notice that we can always add **satellite data** instead of just using a Boolean array to record the existence of the keys.

For example, we can use an **associative** String array **A** instead to map a bus route number to its operator name, e.g.,

```
A[2] = "Go-Ahead Singapore",
A[10] = "SBS Transit",
A[183] = "Tower Transit Singapore",
A[188] = "SMRT Buses", etc.
```

Discussion: Can you think of a few other real-life DAT examples?

2-5. The Answer

[This is a hidden slide]

2-6. DAT Limitations

The keys must be (or can be easily mapped to) **non-negative Integer** values. Note that basic DAT has problem in the full version of the example in the previous few slides as there are actually variations of bus route numbers in Singapore, e.g., 96B, 151A, NR10, etc.

The range of keys must be **small**.
The memory usage will be (insanely) large if we have (insanely) large range.

The keys must be dense, i.e., not many gaps in the key values.
DAT will contain too many empty (and wasted) cells otherwise.

We will overcome these restrictions with hashing.

3. Hashing: Ideas

Using hashing, we can:

1. Map (some) **non-Integer** keys (e.g., Strings) to Integers keys,
2. Map **large** Integers to **smaller** Integers.

3-1. Phone Numbers Example

For example, we have **N** = 400 Singapore phone numbers (Singapore phone number has 8 digits, so there are up to 10^8 = 100M possible phone numbers in Singapore).

Instead of using a DAT and use a **gigantic** array up to size **M** = 100 Million, we can use the following simple hash function **h(v) = v%997**.

This way, we map 8 digits phone numbers **6675 2378** and **6874 4483** into up to 3 digits **h(6675 2378) = 237** and **h(6874 4483) = 336**, respectively. Therefore, we only need to prepare array of size **M** = 997 (or just simplify to 1000) instead of **M** = 100 Million.

3-2. Hash Table Preview

With hashing, we can now implement the following Table ADT operations using Integer array (instead of Boolean array) as follows:

1. Search(v): Check if **A[h(v)] != -1** (we use -1 for an empty cell assuming **v ≥ 0**),
2. Insert(v): Set **A[h(v)] = v** (we hash **v** into **h(v)** so we need to somehow record key **v**),
3. Remove(v): Set **A[h(v)] = -1** — to be elaborated further.

3-3. Hash Table with Satellite Data

If we have keys that map to satellite data and we want to record the original keys too, we can implement the Hash Table using pair of (Integer, satellite-data-type) array as follows:

1. Search(v): Return **A[h(v)]**, which is a **pair (v, satellite-data)**, possibly empty,
2. Insert(v, satellite-data): Set **A[h(v)] = pair(v, satellite-data)**,
3. Remove(v): Set **A[h(v)] = (empty pair)** — to be elaborated further.

However, by now you should notice that something is incomplete...

3-4. Collision

A hash function may, and quite likely, map **different keys (Integer or not)** into the **same Integer slot**, i.e., a **many-to-one** mapping instead of **one-to-one** mapping.

For example, **h(6675 2378) = 237** from three slides earlier and if we want to insert another phone number **6675 4372**, we will have a problem as **h(6675 4372) = 237** too.

This situation is called a **collision**, i.e., two (or more) keys have the **same hash value**.

3-5. Probability of Collision

The Birthday (von Mises) Paradox asks: 'How many people (number of keys) must be in a room (Hash Table) of size 365 seats (cells) before the probability that some person **share a birthday** (collision, two keys are hashed to the same cell), ignoring the leap years (i.e., all years have 365 days), becomes > 50 percent (i.e., more likely than not)?'

The answer, which maybe surprising for some of us, is Reveal.

Let's do some calculation.

3-6. The Calculation

Let **Q(n)** be the probability of **unique** birthday for **n** people in a room.
$Q(n) = 365/365 \times 364/365 \times 363/365 \times ... \times (365-n+1)/365$,
i.e., the first person's birthday can be any of the 365 days, the second person's birthday can be any of the 365 days except the first person's birthday, and so on.

Let **P(n)** be the probability of **same birthday** (collision) for **n** people in a room.
$P(n) = 1-Q(n)$.

We compute that **$P(23) = 0.507$** > **0.5 (50%)**.

Thus, we only need **23 people** (a small amount of keys) in the room (Hash Table) of size 365 seats (cells) for a (more than) 50% chance collision to happen (the birthday of two different people in that room is one of 365 days/slots).

3-7. Two Important Issues

Issue 1: We have seen a simple hash function like the **h(v) = v%997** used in Phone Numbers example that maps large range of Integer keys into a smaller range of Integer keys, but how about non Integer keys? How to do such hashing efficiently?

Issue 2: We have seen that by hashing, or mapping, large range into smaller range, there will very likely be a collision. How to deal with them?

4. Hash Functions

How to create a good hash function with these desirable properties?

1. Fast to compute, i.e., in O(**1**),
2. Uses as minimum slots/Hash Table size **M** as possible,
3. Scatter the keys into different base addresses as uniformly as possible ∈ [0..**M**-1],
4. Experience as minimum collisions as possible.

4-1. Preliminaries

Suppose we have a hash table of size **M** where keys are used to identify the satellite-data and a specific hash function is used to compute a hash value.

A **hash value/hash code** of key **v** is computed from the key **v** with the use of a hash function to get an Integer in the range 0 to **M**-1. This hash value is used as the base/home index/address of the Hash Table entry for the satellite-data.

4-2. Example of a Bad Hash Function

Using the Phone Numbers example, if we we define **h(v) = floor(v/1 000 000)**, i.e. we select the first two digits a phone number.

```
h(66 75 2378) = 66
h(68 74 4483) = 68
```

Discuss: What happen when you use that hash function? Hint: See this.

4-3. The Answer

[This is a hidden slide]

4-4. Perfect Hash Function

Before discussing the reality, let's discuss the ideal case: **perfect hash functions**.

A perfect hash function is a **one-to-one** mapping between keys and hash values, i.e., no collision at all. It is possible if all keys are known beforehand. For example, a compiler/interpreter search for reserved keywords. However, such cases are rare.

A minimal perfect hash function is achieved when the table size is the same as the number of keywords supplied. This case is even rarer.

If you are interested, you can explore GNU gperf, a freely available perfect hash function generator written in C++ that automatically constructs perfect functions (a C++ program) from a user supplied list of keywords.

4-5. Hashing Integer - Best Practice

People has tried various ways to hash a large range of Integers into a smaller range of Integers as uniformly as possible. In this e-Lecture, we jump directly to one of the best and most popular version: **h(v) = v%M**, i.e., map **v** into Hash Table of size **M** slots. The (%) is a modulo operator that gives the remainder after division. This is clearly fast, i.e., O(**1**) assuming that **v** does not exceed the natural Integer data type limit.

The Hash Table size **M** is set to be a reasonably large prime not near a power of 2, about 2+ times larger than the expected number of keys **N** that will ever be used in the Hash Table. This way, the load factor α = N/M < 0.5 — we shall see later that having low load factor, thereby sacrificing empty spaces, help improving Hash Table performance.

Discuss: What if we set **M** to be a power of 10 (decimal) or power of 2 (binary)?

4-6. The Answer

[This is a hidden slide]

4-7. Hashing String - Best Practice

People has also tried various ways to hash Strings into a small range of Integers as uniformly as possible. In this e-Lecture, we jump directly to one of the best and most popular version, shown below:

```
int hash_function(string v) { // assumption 1: v uses ['A'..'Z'] only
  int sum = 0;                // assumption 2: v is a short string
  for (auto& c : v) // for each character c in v
    sum = ((sum*26)%M + (c-'A'+1))%M; // M is table size
  return sum;
}
```

Discussion: In real life class, discuss the components of the hash function above, e.g., why loop through all characters?, will that be slower than O(**1**)?, why multiply with 26?, what if the string v uses more than just UPPERCASE chars?, etc.

4-8. The Answer

[This is a hidden slide]

5. Collision Resolution

There are two major ideas: **Open Addressing** versus **Closed Addressing** method.

In Open Addressing, all hashed keys are located in a single array. The hash code of a key gives its base address. Collision is resolved by checking/probing multiple alternative addresses (hence the name **open**) in the table based on a certain rule.

In Closed Addressing, the Hash Table looks like an Adjacency List (a graph data structure). The hash code of a key gives its fixed/**closed** base address. Collision is resolved by appending the collided keys inside a (Doubly) Linked List identified by the base address.

5-1. Open Addressing (OA)

There are three Open Addressing (OA) collision resolution techniques discussed in this visualization: Linear Probing (LP), Quadratic Probing (QP), and Double Hashing (DH).

To switch between the three modes, please click on the respective header.

Let:
M = HT.length = the current hash table size,
base = (key%HT.length),
step = the current probing step,
secondary = smaller_prime - key%smaller_prime (to avoid zero — elaborated soon)

We will soon see that the probing sequences of the three modes are:
Linear Probing: i=(base+step*1) % M,
Quadratic Probing: i=(base+step*step) % M, and
Double Hashing: i=(base+step*secondary) % M.

5-2. Separate Chaining (SC)

Separate Chaining (SC) collision resolution technique is simple. We use **M** copies of auxiliary data structures, usually Doubly Linked Lists. If two keys **a** and **b** both have the same hash value **i**, both will be appended to the (front/back) of Doubly Linked List **i** (in this visualization, we append to the back in O(**1**) with help of tail pointer). That's it, where the keys will be slotted in is completely dependent on the hash function itself, hence we also call Separate Chaining as Closed Addressing collision resolution technique.

If we use Separate Chaining, the load factor α = **N/M** describes the average length of the **M** lists and it will determine the performance of Search(v) as we may have to explore α elements on average. As Remove(v) — also requires Search(v), its performance will be similar as Search(v). Insert(v) is clearly O(**1**).

If we can bound α to be a small constant (true if we know the expected largest **N** in our Hash Table application so that we can set up **M** accordingly), then all Search(v), Insert(v), and Remove(v) operations using Separate Chaining will be O(**1**).

6. Visualisation

View the visualisation of Hash Table above.

In this visualization, we prevent insertion of duplicate keys.

Due to limited screen space, we limit the maximum Hash Table size to be **M = 19**.

The Hash Table is visualized horizontally like an array where index 0 is placed leftmost and index **M**-1 is placed rightmost but the details are different when we are visualizing Open Addressing versus Separate Chaining collision resolution techniques.

Discuss: What to modify in order to allow duplicate keys?

6-1. Open Addressing Version

There are three Open Addressing collision resolution techniques discussed in this visualization: Linear Probing (LP), Quadratic Probing (QP), and Double Hashing (DH).

For all three techniques, each Hash Table cell is displayed as a vertex with cell value of [0..99] displayed as the vertex label. Without loss of generality, we do not show any satellite data in this visualization as we concentrate only on the arrangement of the keys. We reserve value -1 to indicate an 'EMPTY cell' (visualized as a blank vertex) and -2 to indicate a 'DELETED cell' (visualized as a vertex with abbreviated label "DEL"). The cell indices ranging from [0..**M**-1] are shown as red label below each vertex.

6-2. Separate Chaining Version

For Separate Chaining (SC) collision resolution technique, the first row contains the **M** "H" (Head) pointers of **M** Doubly Linked Lists.

Then, each Doubly Linked List **i** contains all keys that are hashed into **i** in arbitrary order. Mathematically, all keys that can be expressed as **i** (mod **M**) are hashed into DLL **i**. Again, we do not store any satellite data in this visualization.

7. Linear Probing (LP)

In **Linear Probing** collision resolution technique, we scan forwards one index at a time for the **next empty/deleted slot** (wrapping around when we have reached the last slot) whenever there is a collision.

For example, let's assume we start with an empty Hash Table **HT** with table size **M = HT.length = 7** as shown above that uses index 0 to **M**-1 = 7-1 = 6. Notice that 7 is a prime number. The (primary) hash function is simple, **h(v) = v%M**.

This walk-through will show you the steps taken by Insert(v), Search(v), and Remove(v) operations when using Linear Probing as collision resolution technique.

7-1. Insert([18, 14, 21)

Now click Insert([18,14,21]) — three individual insertions in one command.

Recap (to be shown after you click the button above).

Formally, we describe Linear Probing index **i** as **i = (base+step*1) % M** where **base** is the (primary) hash value of key **v**, i.e., **h(v)** and **step** is the Linear Probing step starting from 1.

Tips: To do a quick mental calculation of a (small) Integer **V** modulo **M**, we simply subtract **V** with the largest multiple of **M** ≤ **V**, e.g., 18%7 = 18-14 = 4, as 14 is the largest multiple of 7 that is ≤ 18.

7-2. Insert([1, 35])

Now click Insert([1,35]) (on top of the first three values inserted in the previous slide).

Recap (to be shown after you click the button above)

7-3. Search(35) and Search(8)

Now we illustrate Search(v) operation while using Linear Probing as collision resolution technique. The steps taken are very similar as with Insert(v) operation, i.e., we start from the (primary) hash key value and check if we have found **v**, otherwise we move one index forward at a time (wrapping around if necessary) and recheck on whether we have found **v**. We stop when we encounter an empty cell which implies that **v** is not in Hash Table at all (as earlier Insert(v) operation would have placed **v** there otherwise).

Now click Search(35) — you should see probing sequence [0,1,2,3 (key 35 found)].

Now click Search(8) — [1,2,3,4, 5 (empty cell, so key 8 is not found in the Hash Table)].

7-4. Remove(v) - Preliminary

Now let's discuss Remove(v) operation.

If we just set **HT[i] = EMPTY** cell straightaway where **i** is the index that contains **v** (after linear probing if necessary), do you realize that we will cause a problem? Why?

Hint: Review the past three slides on how Insert(v) and Search(v) behave.

7-5. The Answer

[This is a hidden slide]

7-6. Remove(21)

Now let's see the complete Remove(v). If we find **v** at index **i** (after Linear Probing if necessary), we have to set **HT[i] = DELETED** (abbreviated as **DEL** in this visualization) where **DEL** is a special symbol (generally you should only use a symbol that is **not** used in your application) to indicate that cell can be by-passed if necessary by future Search(v), but can be overwritten by future Insert(w). This strategy is called **Lazy Deletion**.

Now click Remove(21) — [0,1 (key 21 found and we set **H[1] = DEL**)].

Afterwards, please continue the discussion in the next slide.

7-7. Search(35) Again

Now click Search(35) — [0,1 (bypassing that DELETED cell), 2,3 (found key 35)].

Imagine what would have happened if we *wrongly* set **H[1] = EMPTY**.

7-8. Insert(28) - Overwriting DEL

Now click Insert(28) — you should see probing sequence [0,1 (found a cell with DEL symbol)], so it is actually can be overwritten with a new value without affecting the correctness of future Search(v). Therefore, we put 28 in index 1.

7-9. Primary Clustering, Part 1

Although we can resolve collision with Linear Probing, it is not the most effective way.

We define a **cluster** to be a collection of consecutive occupied slots. A cluster that covers the base address of a key is called the **primary cluster** of the key.

Now notice that Linear Probing can create large primary clusters that will increase the running time of Search(v)/Insert(v)/Remove(v) operations beyond the advertised O(**1**).

See an example above with **M** = 31 and we have inserted 15 keys [1..15] so that they occupy cells [1..15]. Now see how 'slow' Insert(32) (the 16th key) is.

7-10. Linear Probing Sequence

The probe sequence of Linear Probing can be formally described as follows:

```
 h(v) // base address
(h(v) + 1*1) % M // 1st probing step if there is a collision
(h(v) + 2*1) % M // 2nd probing step if there is still a collision
(h(v) + 3*1) % M // 3rd probing step if there is still a collision
...
(h(v) + k*1) % M // k-th probing step, etc...
```

During Insert(v), if there is a collision but there is an empty (or DEL) slot remains in the Hash Table, we are sure to find it after at most **M** Linear Probing steps, i.e., in O(**M**). And when we do, the collision will be resolved, but the primary cluster of the key **v** is expanded as a result and future Hash Table operations will get slower too. Try the slow Search(32) on the same Hash Table as in the previous slide but with many DEL markers (suppose {4, 5, 8, 9, 10, 12, 14} have just been deleted).

7-11. Primary Clustering, Part 2

In the previous slide (Primary Clustering, Part 1), we break the assumption that the hash function should uniformly distribute keys around [0..**M**-1]. In the next example, we will show that the problem of primary clustering can still happen even if the hash function distribute the keys into several relatively short primary clusters around [0..**M**-1].

On screen, you see **M** = 31 from the previous slide, but only cells [1..6], [8..13], and [15..20] are filled. If we then insert these next 3 keys {37, 44, 63}, these first two keys will initially collide with the cells that already contain {6, 13}, have "short" one-step probes, then by doing so "plug" the empty cells and accidentally annex (or combine) those neighboring (but previously disjointed) clusters into a (very) long primary cluster. So the next insertion of a key {63} that lands at (the beginning of) this long primary cluster will end up performing almost O(M) probing steps just to find an empty cell. Try Insert([37,44,63]).

8. Quadratic Probing (QP)

To reduce primary clustering, we can modify the probe sequence to:

```
 h(v) // base address
(h(v) + 1*1) % M // 1st probing step if there is a collision
(h(v) + 2*2) % M // 2nd probing step if there is still a collision
(h(v) + 3*3) % M // 3rd probing step if there is still a collision
...
(h(v) + k*k) % M // k-th probing step, etc...
```

That's it, the probe jumps quadratically, wrapping around the Hash Table as necessary.

A very common mistake as this is a different kind of Quadratic Probing:
Doing h(v), (h(v)+1) % M, (h(v)+1+4) % M, (h(v)+1+4+9) % M, ...

8-1. Insert(38)

Assume that we have called Insert(18) and Insert(10) into an initially empty Hash Table of size **M = HT.length = 7**. As 18%7 = 4 and 10%7 = 3, 18 and 3 do not collide and both reside in index 4 and 3 respectively as shown above.

Now, let's click Insert(38).

Recap (to be shown after you click the button above).

8-2. Remove(18) and Search(38) Again

Remove(x) and Search(y) operations are defined similarly. Just that this time we use Quadratic Probing instead of Linear Probing.

For example, assume that we have called Remove(18) after the previous slide and we mark **HT[4] = DEL**. If we then call Search(38), we will use the same Quadratic Probing sequence as with previous slide, but passing through **HT[4]** which marked as DELETED.

8-3. Better than Linear Probing?

In a glance, Quadratic Probing that jumps +1, +4, +9, +16, ... quadratically seems able to solve the primary clustering issue that we have with Linear Probing earlier, but is it the perfect collision resolution technique?

Try Insert([12,17]).

Do you realized what has just happened?

8-4. The Details

We can insert 12 easily as h(12) = 12%7 = 5 was empty previously (see above).

However we have major problem inserting key 17 even if we still have 3 empty slots as:
h(17) = 17%7 = 3 is already occupied by key 10,
(3+1*1) % 7 = 4 is already occupied by key 18,
(3+2*2) % 7 = 0 is already occupied by key 38,
(3+3*3) % 7 = 5 is already occupied by key 12,
(3+4*4) % 7 = 5 again is already occupied by key 12,
(3+5*5) % 7 = 0 again is already occupied by key 38,
(3+6*6) % 7 = 4 again is already occupied by key 18,
(3+7*7) % 7 = 3 again is already occupied by key 10,
it will **cycle forever** if we continue the Quadratic Probing...

Although we still have a few (3) empty cells, we are unable to insert this new value 17 into the Hash Table...

8-5. A Theorem

If $\alpha < 0.5$ and **M** is a prime (> 3), then we can always find an empty slot using Quadratic Probing. Recall: α is the load factor and **M** is the Hash Table size (HT.length).

If the two requirements above are satisfied, we can prove that the first **M**/2 Quadratic Probing indices, including the base address `h(v)` are all distinct and unique.

But there is no such guarantee beyond that. Hence if we want to use Quadratic Probing, we need to ensure that α < 0.5 (not enforced in this visualization but we do break the loop after **M** steps to prevent infinite loop).

8-6. A Proof

We will use proof by contradiction. We first assume that two Quadratic Probing steps: x and y, x != y (let's say x < y), can yield the same address modulo **M**.

```
h(v) + x*x = h(v) + y*y (mod M)
x*x = y*y (mod M) // strike out h(v) from both sides
x*x - y*y = 0 (mod M) // move y*y to LHS
(x-y)*(x+y) = 0 (mod M) // rearrange the formula
```

Now, either `(x-y)` or `(x+y)` has to be equal to zero.
As our assumption says `x != y`, then `(x-y)` cannot be 0.
As `0 ≤ x < y ≤ (M/2)` and `M` is a prime > 3 (an odd Integer),
then `(x+y)` also cannot be 0 modulo `M`.

Contradiction!

So the first `M/2` Quadratic Probing steps cannot yield the same address modulo `M` (if we set **M** to be a prime number greater than 3).

Discussion: Can we make Quadratic Probing able to use the other ~50% of the table cells?

8-7. Better Quadratic Probing

[This is a hidden slide]

8-8. Secondary Clustering

In Quadratic Probing, clusters are formed along the path of probing, instead of around the base address like in Linear Probing. These clusters are called **Secondary Clusters** and it is 'less visible' compared to the Primary Clusters that plagued the Linear Probing.

Secondary clusters are formed as a result of using the same pattern in probing by colliding keys, i.e., if two distinct keys have the same base address, their Quadratic Probing sequences are going to be the same.

To illustrate this, see the screen with **M** = 31. We have populated this Hash Table with only 10 keys (so load factor α = 10/31 ≤ 0.5) and the Hash Table looks 'sparse enough' (no visibly big primary cluster). However, if we then insert Insert(62,93), despite the fact that there are many (31-10 = 21) empty cells and 62 != 93 (different keys that ends up hashed into index 0), we end up doing 10 probing steps along this 'less visible' secondary cluster (notice that both {62, 93} follow similar Quadratic Probing sequences).

Secondary clustering in Quadratic Probing is not as bad as primary clustering in Linear Probing as a good hash function should theoretically disperse the keys into different base addresses ∈ [0..**M**-1] in the first place.

9. Double Hashing (DH)

To reduce primary and secondary clustering, we can modify the probe sequence to:

```
 h(v) // base address
(h(v) + 1*h2(v)) % M // 1st probing step if there is a collision
(h(v) + 2*h2(v)) % M // 2nd probing step if there is still a collision
(h(v) + 3*h2(v)) % M // 3rd probing step if there is still a collision
...
(h(v) + k*h2(v)) % M // k-th probing step, etc...
```

That's it, the probe jumps according to the value of the **second hash function** `h2(v)`, wrapping around the Hash Table as necessary.

9-1. Secondary Hash Function h2(v)

If `h2(v) = 1`, then Double Hashing works exactly the same as Linear Probing.
So we generally wants `h2(v) > 1` to avoid primary clustering.

If `h2(v) = 0`, then Double Hashing does not work for an obvious reason as any probing step multiplied by 0 remains 0, i.e. we stay at the base address forever during a collision. We need to avoid this.

Usually (for Integer keys), `h2(v) = M' - v%M'` where `M'` is a smaller prime than `M`.
This makes `h2(v)` ∈ [1..**M'**], which is diverse enough to avoid secondary clustering.

The usage of the secondary hash function makes it theoretically hard to have either primary or secondary clustering issue.

9-2. Insert([35, 42])

Click Insert([35,42]) to insert 35 and then 42 to the current Hash Table above.

Recap (to be shown after you click the button above).

9-3. Remove(17) and Search(35) Again

Remove(x) and Search(y) operations are defined similarly. Just that this time we use Double Hashing instead of Linear Probing or Quadratic Probing.

For example, assume that we have called Remove(17) after the previous slide and we mark **HT[3] = DEL**. If we then call Search(35), we will use the same Double Hashing sequence as with previous slide, but passing through **HT[3]** which marked as DELETED.

9-4. Good OA Collision Resolution Technique

In summary, a good Open Addressing collision resolution technique needs to:

1. Always find an empty slot if it exists,
2. Minimize clustering (of any kind),
3. Give different probe sequences when 2 different keys collide,
4. Fast, O(**1**).

Now, let's see the same test case that plagues Quadratic Probing earlier. Now try Insert(62,93) again. Although h(62) = h(93) = 0 and their collide with 31 that already occupy index 0, their probing steps are not the same: h2(62) = 29-62%29 = 25 is not the same as h2(93) = 29-93%29 = 23.

Discussion: Double Hashing seems to fit the bill. But... Is Double Hashing strategy flexible enough to be used as the default library implementation of a Hash Table? Let's see...

10. Separate Chaining (SC)

Try Insert([9,16,23,30,37,44]) to see how Insert(v) operation works if we use Separate Chaining as collision resolution technique. On such random insertions, the performance is good and each insertion is clearly O(**1**).

However if we try Insert([68,90]), notice that all Integers {68,90} are 2 (modulo 11) so all of them will be appended into the (back of) Doubly Linked List 2. We will have a long chain in that list. Note that due to the screen limitation, we limit the length of each Doubly Linked List to be at maximum 6.

10-1. Search(35) and Remove(35)

Try Search(35) to see that Search(v) can be made to run in O(**1+α**).

Try Remove(35) to see that Remove(v) can be made to run in O(**1+α**) too.

If **α** is large, Separate Chaining performance is not really O(**1**). However, if we roughly know the potential maximum number of keys **N** that our application will ever use, then we can set table size **M** accordingly such that **α = N/M** is a very low positive (floating-point) number, thereby making Separate Chaining performances to be expected O(**1**).

10-2. Open Addressing vs Separate Chaining?

Discussion: After all these explanations, which of the two collision resolution technique is the better one?

10-3. The (Current) Answer

[This is a hidden slide]

11. Extras

You have reached the end of the basic stuffs of this Hash Table data structure and we encourage you to explore further in the **Exploration Mode**.

However, we still have a few more interesting Hash Table challenges for you that are outlined in this section.

11-1. Rehash

The performance of Hash Table degrades when the load factor α gets higher. For (standard) Quadratic Probing collision resolution technique, insertions might fail when the Hash Table has α > 0.5.

If that happens, we can **rehash**. We build another Hash Table about twice as big with a new hash function. We go through all keys in the original Hash Table, recompute the new hash values, and re-insert the keys (with their satellite-data) into the new, bigger Hash Table, before finally we delete the older, smaller Hash Table.

A rule of thumb is to rehash when α ≥ 0.5 if using Open Addressing and when α > small constant (close to 1.0, as per requirement) if using Separate Chaining.

If we know the maximum number of total possible keys, we can always influence α to be a low number.

11-2. Hash Table Implementation

However, if you ever need to implement a Hash Table in C++, Python, or Java, and your keys are either Integers or Strings, you can use the built-in C++ STL, Python standard library, or Java API, respectively. They already have good built-in implementation of default hash functions for Integers or Strings.

See C++ STL std::unordered_map, std::unordered_set, Python dict, set, or Java HashMap, HashSet.

For C++, note that the std::multimap/std::multiset implementations are also exist where duplicate keys are allowed.

For OCaml, we can use Hashtbl.

However, here is our take of a simple Separate Chaining implementation: HashTableDemo.cpp | py | java.

11-3. Data Structure Combo?

[This is a hidden slide]

11-4. Alternative Data Structure for Table ADT

Hash Table is an extremely good data structure to implement Table ADT if the (Integer or String) keys only need to be mapped to satellite-data, with O(**1**) performance for Search(v), Insert(v), and Remove(v) operations if the Hash Table is set up properly.

However, if we need to do much more with the keys, we may need to use an alternative data structure.

11-5. Online Quiz

For a few more interesting questions about this data structure, please practice on Hash Table training module (no login is required, but short and of medium difficulty setting only).

However, for registered users, you should login and then go to the Main Training Page to officially clear this module and such achievement will be recorded in your user account.

11-6. Online Judge Exercises

Try to solve a few basic programming problems that somewhat requires the usage of Hash Table (especially if the input size is much larger):

1. Kattis - cd (the inputs are already sorted so alternative, non Hash Table solution exists; if the inputs are not sorted, this set intersection problem is best solved with help of a Hash Table),
2. Kattis - oddmanout (we can map large invitation codes into smaller range of integers; this is a practice of hashing (large range) of integers),
3. Kattis - whatdoesthefoxsay (we put sounds that are not fox into an unordered set; this is a practice of hashing strings).