CSE 100: UNION-FIND HASH
If we have no rule in place for performing the union of two up trees, what is the worst case run time of find and union operations in terms of the number of elements $N$?

A. find: $O(N)$, union: $O(1)$
B. find: $O(1)$, union: $O(N)$
C. find: $O(\log_2 N)$, union: $O(1)$
D. find: $O(1)$, union: $O(\log_2 N)$
Improving Union operations

- In the Union operation, a tree becomes like a linked list if, when joining two trees, the root of the smaller tree (e.g. a single node) always becomes the root of the new tree.
- We can avoid this by making sure that in a Union operation, the larger of the two trees’ roots becomes the root of the new tree (ties are broken arbitrarily).

Henry Kautz, U. Washington
Smarter Union operations

• Avoid this problem by having the *larger* of the 2 trees become the root
  • **Union-by-size**
    • Each root stores the size (# nodes) of its respective tree
    • The root with the larger size becomes the parent
    • Update its size = sum of its former size and the size of its new child
    • Break ties arbitrarily

```
  a
 / \
 b   c
 |   |
 d   f
 |   |
 e
```
Disjoint subsets using trees
(Union-by-size and simple Find)

Start with 4 items: 0, 1, 2, 3

Assume that \textbf{Union}(i,j) makes the root of the smaller tree a child of the root of the larger tree.

Perform these operations:

- \textbf{Union}(2,3)
- \textbf{Union}(1,2)
- \textbf{Find}(0) =
- \textbf{Find}(3) =
- \textbf{Find}(1) =
- \textbf{Find}(0) =
Smarter Union operations

- Avoid this problem by having the *larger* of the 2 trees become the root
  - **Union-by-size**
  - **Union-by-height (also called union by rank)**
    - Each root stores the height of its respective tree
    - If one root has a greater height than the other, it becomes the parent. Its stored height doesn’t need to be updated
    - If the roots show equal height, pick either one as the parent
    - Its stored height should be increased by one
    - Break ties arbitrarily
Disjoint subsets using trees
(Union-by-height and simple Find)

Start with 4 items: 0, 1, 2, 3

Assume that \texttt{Union(i,j)} makes the root of the shorter tree a child of the root of the taller tree.

Perform these operations:

\begin{itemize}
  \item \texttt{Union}(2,3)
  \item \texttt{Union}(1,2)
  \item \texttt{Find}(0) =
  \item \texttt{Find}(3) =
  \item \texttt{Find}(0) =
  \item \texttt{Union}(0,2)
  \item \texttt{Find}(1) =
  \item \texttt{Find}(0) =
\end{itemize}
Either union-by-size or union-by-height will guarantee that the height of any tree is no more than $\log_2 N$, where $N$ is the total number of nodes in all trees.

Q: What is the worst case run time of find and union operations in terms of the number of elements $N$?

A. find: $O(N)$, union: $O(1)$
B. find: $O(1)$, union: $O(N)$
C. find: $O(\log_2 N)$, union: $O(1)$
D. find: $O(1)$, union: $O(\log_2 N)$

Solo vote!
Bounding the height of the up-tree using union by size.

- Initially all the nodes are in singleton trees (with height 1)
- Take the perspective of a single node.
- The only time the height of the tree which the node is part of increases by 1 is when the node joins a larger group i.e. if the height of the node’s uptree increases by 1, the number of nodes in that tree would have at least doubled
- The maximum number of nodes in any tree is N, so the height of the resulting tree can be at most log N
Q: What is the worst case run time of find and union operations in terms of the number of elements N?

- Therefore, doing N-1 union operations (the maximum possible) and M find operations takes time $O(N + M \log_2 N)$ worst case

- With simple unions the complexity was:

- This is a big improvement; but we can do still better, by a slight change to the Find operation: adding *path compression*

Discuss and revote

Cost of disjoint subsets operations with smarter Union and simple Find

- A. find: $O(N)$, union: $O(1)$
- B. find: $O(1)$, union: $O(N)$
- C. find: $O(\log_2 N)$, union: $O(1)$
- D. find: $O(1)$, union: $O(\log_2 N)$
Kruskal’s algorithm run time with smart union path compression find:

1. Sort edges in increasing order of cost

2. Set of edges in MST, $T = \{}$

3. For $i = 1$ to $|E|$
   
   if $(\text{find}(u) \neq \text{find}(v))$  //If $T \cup \{e_i = u,v\}$ has no cycles
   
   Add $e_i$ to $T$

   $\text{union}(\text{find}(u), \text{find}(v))$

What is the improvement from simple union find?

Ref: Tim Roughgarden (stanford)
• Using the array representation for disjoint subsets, the code for implementing the Disjoint Subset ADT’s methods is compact

```cpp
class DisjSets{
    int *array;

    /**
     * Construct the disjoint sets object
     * numElements is the initial number of disjoint sets
     */
    DisjSets( int numElements )
        { 
            array = new int [ numElements ];
            for( int i = 0; i < numElements; i++ )
                array[ i ] = -1;
        }
}
```
/**
 * Union two disjoint sets using the height heuristic.
 * For simplicity, we assume root1 and root2 are distinct
 * and represent set labels.
 * root1 is the root of set 1
 * root2 is root of set 2
 * returns the root of the union
 */
int union ( int root1, int root2 )
{
if( array[ root2 ] < array[ root1 ] ) {
    // root2 is higher
    array[ root1 ] = root2; // Make root2 new root
    return root2;
} else {
    if( array[ root1 ] == array[ root2 ] )
    {
        array[ root1 ]--; // Update height if same
        array[ root2 ] = root1; // Make root1 new root
    }
    return root1;
}
}
Simple Find

• In a disjoint subsets structure using parent-pointer trees, the basic Find operation is implemented as:
  • Go to the node corresponding to the item you want to Find the equivalence class for
  • Traverse parent pointers from that node to the root of its tree
  • Return the label of the root
  • This has worst-case time cost $O(\log_2 N)$
  • The time cost of doing another Find operation on the same item is the same
Find with Path-compression

- The path-compression Find operation is implemented as:
  - Go to the node corresponding to the item you want to Find the equivalence class for
  - Traverse parent pointers from that node to the root of its tree
  - Return the label of the root
  - But ….

… as part of the traversal to the root, change the parent pointers of every node visited, to point to the root of the tree (all become children of the root)

worst-case time cost is still $O(\log N)$
Example of path compression

\[ \text{find(e)} \]

… as part of the traversal to the root, change the parent pointers of every node visited, to point to the root of the tree (all become children of the root)
worst-case time cost is still \( O(\log N) \)

Henry Kautz
Cost of find with path compression

What is the time cost of doing another Find operation on the same item, or on any item that was on the path to the root?

A. O(log N)  
B. O(N)  
C. O(1)
Disjoint subsets using trees
(Union-by-height and path-compression Find)

• Start with 4 items: 0, 1, 2, 3

• \textbf{Union}(i,j) makes the root of the shorter tree a child of the root of the taller tree

• We perform path compression

• If an array element contains a negative \textit{int}, then that element represents a tree root, and the value stored there is -1 times (the height of the tree plus 1)

Perform these operations:
  \texttt{Union(2,3)}
  \texttt{Union(0,1)}
  \texttt{Find(0) =}
  \texttt{Find(3) =}
  \texttt{Find(3) =}
  \texttt{Union(0,2)}
  \texttt{Find(1) =}
  \texttt{Find(3) =}
  \texttt{Find(3) =}
Find with path compression

/**
 * Perform a find with path compression
 * Error checks omitted again for simplicity
 * @param x the label of the element being searched for
 * @return the label of the set containing x
 */
int find( int x ) {
    if( array[ x ] < 0 )
        return x;
    else
        return array[ x ] = find( array[ x ] );
}

• Note that this path-compression find method does not update the disjoint subset tree heights; so the stored heights (called “ranks”) will overestimate of the true height
• A problem for the cost analysis of the union-by-height method (which now is properly called union-by-rank)
Self-adjusting data structures

- Path-compression Find for disjoint subset structures is an example of a *self-adjusting* structure.

- Other examples of self-adjusting data structures are splay trees, self-adjusting lists, skew heaps, etc.

- In a self-adjusting structure, a find operation occasionally incurs high cost because it does extra work to modify (adjust) the data structure, with the hope of making subsequent operations much more efficient.

- Does this strategy pay off? *Amortized cost analysis* is the key to answering that question...
Amortized cost analysis

- Amortization corresponds to spreading the cost of an infrequent expensive item (car, house) over the use period of the item.
- The amortized cost should be comparable to alternatives such as renting the item or taking a loan.
- Amortized analysis of a data structure considers the average cost over many actions.
• It can be shown (Tarjan, 1984) that with Union-by-size or Union-by-height, using path-compression, Find makes any combination of up to N-1 Union operations and M Find operations have a worst-case time cost of $O(N + M \log^* N)$.

• This is very good: it is almost constant time per operation, when amortized over the N-1 + M operations!
Amortized cost analysis results for path compression

Find

- $\log^* N = \text{“log star of } N\text{”} = \text{smallest } k \text{ such that } \log^{(k)} n \leq 1$ or
  \# times you can take the log base-2 of $N$, before we get a number $\leq 1$
- Also known as the “single variable inverse Ackerman function”

\[
\begin{align*}
\log^* 2 &= 1 \\
\log^* 4 &= 2 \\
\log^* 16 &= 3 \\
\log^* 65536 &= 4 \\
\log^* 2^{65536} &= 5 \\
4 &= (2)^2 \\
16 &= (2)^4 \\
65536 &= (2)^{16} \\
2^{65536} &= \text{a huge number of 20000 digits}
\end{align*}
\]

- $\log^* N$ grows extremely slowly as a function of $N$
- It is not constant, but for all practical purposes, $\log^* N$ is never more than 5
Running Time of Kruskal’s algorithm with union find data structure:

1. Sort edges in increasing order of cost

2. Set of edges in MST, $T=\{\}$

3. For $i=1$ to $|E|$

   $S_1=\text{find}(u)$; $S_2=\text{find}(v)$;

   if ($S_1! = S_2$) //If $T \cup \{e_i=u,v\}$ has no cycles

   Add $e_i$ to $T$

   $\text{union}(S_1, S_2)$

   }

Ref: Tim Roughgarden (stanford)
Dijkstra’s Algorithm: Run time

- Initialize the graph: Give all vertices a dist of INFINITY, set all “done” flags to false
- Start at s; give s dist = 0 and set prev field to -1
- Enqueue (s, 0) into a priority queue. This queue contain pairs (v, cost) where cost is the best cost path found so far from s to v. It will be ordered by cost, with smallest cost at the head.

- While the priority queue is not empty:
  - Dequeue the pair (v, c) from the head of the queue.
  - If v’s “done” is true, continue
  - Else set v’s “done” to true. We have found the shortest path to v. (It’s prev and dist field are already correct).
  - For each of v’s adjacent nodes, w:
    - Calculate the best path cost, c, to w via v by adding the edge cost for (v, w) to v’s “dist”.
    - If c is less than w’s “dist”, replace w’s “dist” with c, replace its prev by v and enqueue (w, c)

What is the running time of this algorithm in terms of |V| and |E|? (More than one might be correct—which is tighter?)

A. O(|V|^2)
B. O(|E| + |V|)
C. O(|E| log(|V|))
D. O(|E| log(|E|) + |V|)
E. O(|E|*|V|)
1. Create an empty graph T. Initialize the vertex vector for the graph. Set all “done” fields to false. Pick an arbitrary start vertex $s$. Set its “done” field to true. Iterate through the adjacency list of $s$, and put those edges in the priority queue.

2. While the priority queue is not empty:
   - Remove from the priority queue the edge $(v, w, \text{cost})$ with the smallest cost.
   - Is the “done” field of the vertex $w$ marked true?
     - If Yes: this edge connects two vertices already connected in the spanning tree, and we cannot use it. Go to 2.
     - Else accept the edge:
       - Mark the “done” field of vertex $w$ true, and add the edge $(v, w)$ to the spanning tree $T$.
       - Iterate through $w$’s adjacency list, putting each edge in the priority queue.

What is the running time of this algorithm in terms of $|V|$ and $|E|$? (More than one might be correct—which is tighter?)

A. $O(|V|^2)$
B. $O(|E| + |V|)$
C. $O(|E| \log(|V|))$
D. $O(|E| \log(|E|) + |V|)$
E. $O(|E|*|V|)$
Where we’ve been and where we are going...

Our goal so far: We want to store and retrieve data (keys) fast

Tree structures
- **BSTs**: simple, fast in the average case, but can perform poorly with ordered data (you implemented)
- **AVL trees**: Guaranteed fast, but tricky to implement (you will not implement)
- **RSTs (built on treaps)**: Simpler than AVL trees, usually fast, even with ordered data (you will not implement)
- **Red-black trees**: Guaranteed fast, built in to C++ (coming soon… but you will not implement)

Hash tables (today thru Friday):
- Very fast in the average case
- No sorted access to data
- Can be tricky to do well (you are using an implementation built into C++ in PA3)

Note: you are responsible for knowing how to work with all of these structures as we did in class/in the reading
Fast Lookup: Hash tables

• So far, almost all of our data structures have had performance $O(\log N)$ for insert and find.

• Operations supported by hash tables:
  • Find (key based look up)
  • Insert
  • Delete

• Average run times:

• No worst case guarantees
Hash table Motivation: Two sum problem

- Consider the 2-sum problem: Given an unsorted array \( (A) \) of \( N \) integers between 0 and 1000000, find all pairs of elements that sum to a given number \( T \)

A

- Method 1: Exhaustive search
  
  \[
  \text{for } (i = 0; i < N; i++)\{
    \text{for } (j = i+1; j < N; j++) \{
      \text{if } ((A[i]+A[j])==T) \{
        \text{store } (A[i], A[j]);
      } \}
  \}
  \]

Q: What is the worst case run time of this method? Assume the store method (inserts the elements into an unsorted linked list)

A. \( O(N) \)
B. \( O(N\log N) \)
C. \( O(N^2) \)
Hash table Motivation: Two sum problem

• Consider the 2-sum problem: Given an unsorted array (A) of N integers between 0 and 1000,000, find all pairs of elements that sum to a given number T

```java
sort (A);
for (i = 0; i<N; i++){
    j=binary_search(T-A[i]);
    store (A[i], A[j]);
}
```

Q: What is the worst case run time of this method? Assume the store method (inserts the elements into an unsorted linked list)

A.  O(N)
B.  O(NlogN)
C.  O(N^2)
De-tour: Finding data fast

• Recall:

The data you will store will always be integers between 0 and 1,000,000. You decide to use an array with 1,000,000 Boolean values to store your data. An entry will be true if the item is in your structure, and false otherwise.

• What is the (Big-O) running time to insert an item into this proposed structure?

A. O(N)
B. O(logN)
C. O(1)

The Big O running time of a find will be:
Can we use this results to improve the performance of the two-sum problem
Hash table Motivation: Two sum problem

Consider the 2-sum problem: Given an unsorted array \( A \) of \( N \) integers between 0 and 1000,000, find all pairs of elements that sum to a given number \( T \).

Method 3: Use an array based lookup table

```c
bool Hashtable[ARRAY_SIZE]={false};
for (i = 0; i<N; i++){
    insert(Hashtable, A[i]);
}
for (i = 0; i<N; i++){
    if (find(Hashtable, T-A[i]) == true)
        store (A[i], A[j]);
}
```

Q: What is the worst case run time of this method? Assume the store method (inserts the elements into an unsorted linked list)

A. \( O(N) \)
B. \( O(N\log N) \)
C. \( O(N^2) \)

What is the problem with this method?
Fast Lookup: Hash functions

- Setup for hashing:
  - Universe of possible keys $U$
  - Keep track of evolving set $S$
  - $|S|$ is approximately known
Hashing

- Let’s modify our array-based look up table
- Need a hash-function $h(x)$: takes in a key, returns an index in the array
- Gold standard: random hash function

Hash table (array)

Size is proportional to # of keys (not value of keys)

• Hash function
  - (must be fast)

Key

Hash Code (HC)

(index) data

(might be collisions)
Probability of Collisions

- Suppose you have a hash table that can hold 100 elements. It currently stores 9 elements (in 9 different locations in the hash table). What is the probability that your next insert will cause a collision (assuming a totally random hash function)?

A. 0.09
B. 0.10
C. 0.37
D. 0.90
E. 1.00
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)

A. .09  
B. .30  
C. .52  
D. .74  
E. .90
Probability of Collisions

- If you have a hash table with $M$ slots and $N$ keys to insert in it, then the probability of at least 1 collision is:

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

$$= 1 - \prod_{i=1}^{N} P_{N,M}(ith\ key\ no\ collision)$$
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 $N = 10$, $\text{prob}_{N,M}(\text{collision}) = 12\%$
  - For $N = 20$, $\text{prob}_{N,M}(\text{collision}) = 41\%$
  - For $N = 30$, $\text{prob}_{N,M}(\text{collision}) = 71\%$
  - For $N = 40$, $\text{prob}_{N,M}(\text{collision}) = 89\%$
  - For $N = 50$, $\text{prob}_{N,M}(\text{collision}) = 97\%$
  - For $N = 60$, $\text{prob}_{N,M}(\text{collision}) = 99+\%$

- 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 $\sqrt{2 \oplus 365 \, H_{27}}$ 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
Making hashing work

• Important issues in implementing hashing are:
  • Deciding on the **hash function**
  • Deciding on the **size of the hash table**
  • Deciding on the **collision resolution strategy**

• With a good hashtable design, O(1) average-case insert and find operation time costs can be achieved, with O(1) space cost per key stored

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

Important considerations in designing a hash function for use with a hash table
- It is fast to compute (must be $O(1)$)
- It distributes keys evenly
- It is consistent with the equality testing function (i.e. two keys that are equal will have the same hash value)

Designing a good hash function is not easy!
We’ll look at a few, but there’s much more to explore.
Simple (and effective?) hash function for integers: $H(K) = K \text{ mod } M$

- When is the function $H(K) = K \text{ mod } M$ where $K$ is the key and $M$ is the table size a good hash function for integer keys?
  A. Always
  B. When $M$ is prime
  C. When values of $K$ are evenly distributed
  D. In the case of either B or C
  E. Never
Simple (and effective?) hash function for integers: \( H(K) = K \mod M \)

- When is the function \( H(K) = K \mod M \) where \( K \) is the key and \( M \) is the table size a good hash function for integer keys?

  When \( M \) is prime OR When values of \( K \) are evenly distributed

Think of an example of a non-prime \( M \) and a distribution of keys that would cause poor behavior from this function.
Simple (and effective?) hash function for integers: $H(K) = K \ mod \ M$

* When is the function $H(K) = K \ mod \ M$ where $K$ is the key and $M$ is the table size a good hash function for integer keys?

  - When $M$ is prime OR When values of $K$ are evenly distributed

Because you can never count on evenly distributed keys, always use prime-size table with this hash function.
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 (especially with the simple hash function we looked at).

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.)
Important issues in implementing hashing are:

- Deciding on the **hash function**
- Deciding on the **size of the hash table**
- Deciding on the **collision resolution strategy**

With a good hashtable design, $O(1)$ average-case insert and find operation time costs can be achieved, with $O(1)$ space cost per key stored.

This makes hashtables a very useful and very commonly used data structure.
Simple (and effective?) hash function for integers: $H(K) = K \mod M$

- When is the function $H(K) = K \mod M$ where $K$ is the key and $M$ is the table size a good hash function for integer keys?

  When $M$ is prime OR When values of $K$ are evenly distributed

Because you can never count on evenly distributed keys, always use prime-size table with this hash function
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 (especially with the simple hash function we looked at).

- 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.)
Hash functions for strings

• It is common to want to use string-valued keys in hash tables

• What is a good hash function for strings?

• The basic approach is to use the characters in the string to compute an integer, and then take the integer mod the size of the table

• How to compute an integer from a string?
  • You could just take the last two 16-bit chars (or last four 8-bit characters) of the string and form a 32-bit int
  • But then all strings ending in the same 2 (or 4) chars would hash to the same location; this could be very bad
  • It would be better to have the hash function depend on all the chars in the string

• There is no recognized single "best" hash function for strings. Let’s look at some possible ones
String hash function #1

• This hash function adds up the integer values of the chars in the string (then need to take the result mod the size of the table):

```cpp
int hash(std::string const & key) {
    int hashVal = 0,
    len=key.length();
    for(int i=0; i<len; i++) {
        hashVal += key[i];
    }
    return hashVal;
}
```

What is wrong with this hash function for storing words of 8-characters?
A. Nothing
B. It is too complex (takes too long) to compute
C. It will lead to collisions between words with similar endings
D. It will not distribute keys well in a large table
E. It will never distribute keys well in any table
String hash function #1

• This hash function adds up the integer values of the chars in the string (then need to take the result mod the size of the table):

```cpp
int hash(std::string const & key) {
    int hashVal = 0, len = key.length();
    for(int i=0; i<len; i++) {
        hashVal += key[i];
    }
    return hashVal;
}
```

• This function is simple to compute, but it often doesn’t work very well in practice:

• Suppose the keys are strings of 8 ASCII capital letters and spaces

• There are $27^8$ possible keys; however, ASCII codes for these characters are in the range 65-95, and so the sums of 8 char values will be in the range 520 - 760

• In a large table ($M>1000$), only a small fraction of the slots would ever be mapped to by this hash function! For a small table ($M<100$), it may be okay
String hash function #2: Java code

- java.lang.String’s hashCode() method uses a polynomial with $x=31$ (though it goes through the String’s chars in reverse order), and uses Horner’s rule to compute it:

```java
class String implements java.io.Serializable, Comparable {
    /** The value is used for character storage. */
    private char value[];
    /** The offset is the first index of the storage that is used. */
    private int offset;
    /** The count is the number of characters in the String. */
    private int count;

    public int hashCode() {
        int h = 0;
        int off = offset;
        char val[] = value;
        int len = count;

        for (int i = 0; i < len; i++)
            h = 31*h + val[off++];

        return h;
    }
}
```

$$
H(s) = \sum_{i=0}^{n} s.charAt(i) \times x^i
$$
Collisions

• Since the hash function is $O(1)$, a hash table has the potential for very fast find performance (the best possible!), but...

• ... since the hash function is mapping from a large set (the set of all possible keys) to a smaller set (the set of hash table locations) there is the possibility of *collisions*: two different keys wanting to be at the same table location
Collision resolution strategies

• Unless we are doing "perfect hashing" we have to have a collision resolution strategy, to deal with collisions in the table.

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

• Collision resolution strategies we will look at are:
  • Linear probing (from your reading)
  • Random hashing
  • Separate chaining (from your reading)
Linear probing: inserting a key

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

1. Set $\text{indx} = H(K)$
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 $\text{indx} = (\text{indx} + 1) \mod M$.
5. If $\text{indx} == H(K)$, table is full! (Throw an exception, or enlarge table.) Else go to 2.

$M = 7, \ H(K) = K \mod M$
insert these keys 701 (1), 145 (5), 217 (0), 19 (5), 13 (6), 749 (0)
in this table, using linear probing:
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 $indx = H(K)$
  2. If table location $indx$ contains the key, return FOUND.
  3. Else if table location $indx$ is empty, return NOT FOUND.
  4. Else set $indx = (indx + 1) \mod M$.
  5. If $indx == H(K)$, 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?
Random 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. There is another method called double hashing that works just as well.
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></th>
<th>217</th>
<th>701</th>
<th></th>
<th></th>
<th>145</th>
<th></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>217</td>
<td>701</td>
<td></td>
<td></td>
<td>145</td>
<td></td>
</tr>
</tbody>
</table>
Open addressing vs. separate chaining

• Linear probing 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)
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:

<table>
<thead>
<tr>
<th>index: 0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
</table>

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
D. Both B and C
Analysis of open-addressing hashing

- What is the worst-case time to find a single element in a hash table with N elements in it?
  A. $O(1)$
  B. $O(\log(N))$
  C. $O(N)$
  D. $O(N \log(N))$
  E. $O(N^2)$
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

  We will not derive these, but rather look at a graph that plots the relationship between $\alpha$ and expected number of probes.
Dependence of average performance on load

![Graph showing expected probes vs. load factor](image)

Take away 1: performance depends on load factor, not directly on number of elements or size of table.
Take away 2: things start to go badly when the load factor exceeds 70%
Average case costs with separate chaining

What you need to know:
• Separate chaining performance also depends only on load factor, and not the number of elements or size of table
• In practice it performs extremely well, even with relatively high loads