Introduction to Algorithms and Data Structures

Lecture 8: Sets, dictionaries and hashing

John Longley

School of Informatics
University of Edinburgh

12 October 2023

Sets and dictionaries

Two important datatypes ...

- (Finite) sets of items of a given type X. E.g. $\{3, 5\}$

$$\begin{array}{rl}
\textbf{contains} & : X \to \texttt{bool} \\
\textbf{insert} & : X \to \texttt{void} \\
\textbf{delete} & : X \to \texttt{void} \\
\textbf{isEmpty} & : \texttt{void} \to \texttt{bool}
\end{array}$$

Sets and dictionaries

Two important datatypes ...

- (Finite) sets of items of a given type X. E.g. $\{3,5\} = \{5,3\}$

$$\begin{array}{rl} \textbf{contains} & : X \to \texttt{bool} \\ \textbf{insert} & : X \to \texttt{void} \\ \textbf{delete} & : X \to \texttt{void} \\ \textbf{isEmpty} & : \texttt{void} \to \texttt{bool} \end{array}$$

Sets and dictionaries

Two important datatypes ...

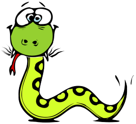
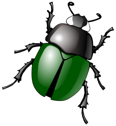
- (Finite) sets of items of a given type X. E.g. $\{3,5\} = \{5,3\}$

$$\begin{array}{rl} \textbf{contains} & : X \to \texttt{bool} \\ \textbf{insert} & : X \to \texttt{void} \\ \textbf{delete} & : X \to \texttt{void} \\ \textbf{isEmpty} & : \texttt{void} \to \texttt{bool} \end{array}$$

- Dictionaries (i.e. lookup tables) mapping keys of type X to values of type Y.

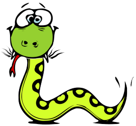
$$\begin{array}{rl} \textbf{lookup} & : X \to Y \\ \textbf{insert} & : X * Y \to \texttt{void} \\ \textbf{delete} & : X \to \texttt{void} \\ \textbf{isEmpty} & : \texttt{void} \to \texttt{bool} \end{array}$$

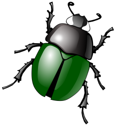
Sets and dictionaries in Python

```
Beatles = {'John', 'Paul', 'George', 'Ringo'}
'George' in Beatles                    # returns True
```

Sets and dictionaries in Python





```
Beatles = {'John', 'Paul', 'George', 'Ringo'}
'George' in Beatles                # returns True

BeatlesYearsOfBirth =
   {'John':1940, 'Paul':1942, 'George':1943, 'Ringo':1940}
BeatlesYearsOfBirth['George']      # returns 1943
```

Sets and dictionaries via sorted arrays

Could implement sets/dictionaries via (any impl of) lists:

```
Beatles_Rep = ['John', 'Paul', 'George', 'Ringo']
BeatlesYearsOfBirth_Rep = [('John',1940), ('Paul',1942), ....]
```

Sets and dictionaries via sorted arrays

Could implement sets/dictionaries via (any impl of) lists:

```
Beatles_Rep = ['John', 'Paul', 'George', 'Ringo']
BeatlesYearsOfBirth_Rep = [('John',1940), ('Paul',1942), ....]
```

But average-case time for **contains/lookup** will be $\Theta(n)$ (terrible!)

Sets and dictionaries via sorted arrays

Could implement sets/dictionaries via (any impl of) lists:

```
Beatles_Rep = ['John', 'Paul', 'George', 'Ringo']
BeatlesYearsOfBirth_Rep = [('John',1940), ('Paul',1942), ....]
```

But average-case time for **contains/lookup** will be $\Theta(n)$ (terrible!)

Much better if arrays are sorted (by key).
Can then use binary search.

Sets and dictionaries via sorted arrays

Could implement sets/dictionaries via (any impl of) lists:

```
Beatles_Rep = ['John', 'Paul', 'George', 'Ringo']
BeatlesYearsOfBirth_Rep = [('John',1940), ('Paul',1942), ....]
```

But average-case time for **contains/lookup** will be $\Theta(n)$ (terrible!)

Much better if arrays are sorted (by key).
Can then use binary search. E.g. for dictionaries:

```
binarySearch(A,key,i,j):      # searches A[i], ..., A[j−1]
    if j−1 = i
        if A[i].key = key then return A[i].value else FAIL
    else
        k = ⌊ i+j/2 ⌋
        if key < A[k].key then return binarySearch(A,key,i,k)
        else return binarySearch(A,key,k,j)
```

Sets and dictionaries via sorted arrays

Could implement sets/dictionaries via (any impl of) lists:

```
Beatles_Rep = ['John', 'Paul', 'George', 'Ringo']
BeatlesYearsOfBirth_Rep = [('John',1940), ('Paul',1942), ....]
```

But average-case time for **contains/lookup** will be $\Theta(n)$ (terrible!)

Much better if arrays are sorted (by key).
Can then use binary search. E.g. for dictionaries:

```
binarySearch(A,key,i,j):      # searches A[i], ..., A[j−1]
    if j−1 = i
        if A[i].key = key then return A[i].value else FAIL
    else
        k = ⌊ i+j/2 ⌋
        if key < A[k].key then return binarySearch(A,key,i,k)
        else return binarySearch(A,key,k,j)
```

Using this, **contains/lookup** have worst-case time $\Theta(\lg n)$.

Sets and dictionaries via sorted arrays

Could implement sets/dictionaries via (any impl of) lists:

```
Beatles_Rep = ['John', 'Paul', 'George', 'Ringo']
BeatlesYearsOfBirth_Rep = [('John',1940), ('Paul',1942), ....]
```

But average-case time for **contains/lookup** will be $\Theta(n)$ (terrible!)

Much better if arrays are sorted (by key).
Can then use binary search. E.g. for dictionaries:

binarySearch(A,key,i,j): # searches A[i], ..., A[j−1]
  if j−1 = i
    if A[i].key = key then return A[i].value else FAIL
  else
    k = ⌊ i+j/2 ⌋
    if key < A[k].key then return **binarySearch**(A,key,i,k)
    else return **binarySearch**(A,key,k,j)

Using this, **contains/lookup** have worst-case time $\Theta(\lg n)$.
But **insert/delete** still costly. Can we do better?

Hash tables

Suppose our keys are strings (e.g. people's names). Number K of potential keys is vast — number n of actual keys 'currently in use' much smaller.

Hash tables

Suppose our keys are strings (e.g. people's names). Number K of potential keys is vast — number n of actual keys 'currently in use' much smaller.

Really silly idea: Give a way of converting strings s to integers $\imath(s)$ (E.g. treat ASCII characters as digits to base 128). Then store value associated with s in a big array at position $\imath(s)$.

Hash tables

Suppose our keys are strings (e.g. people's names). Number K of potential keys is vast — number n of actual keys 'currently in use' much smaller.

Really silly idea: Give a way of converting strings s to integers $\imath(s)$ (E.g. treat ASCII characters as digits to base 128). Then store value associated with s in a big array at position $\imath(s)$.

Impractical: K normally far too large, and most of the array would be unused.

Hash tables

Suppose our keys are strings (e.g. people's names). Number K of potential keys is vast — number n of actual keys 'currently in use' much smaller.

Really silly idea: Give a way of converting strings s to integers $\imath(s)$ (E.g. treat ASCII characters as digits to base 128). Then store value associated with s in a big array at position $\imath(s)$.

Impractical: K normally far too large, and most of the array would be unused.

More sensible idea: Choose some hash function $\#$ mapping potential keys s to integers $0, \ldots, m-1$ (hash codes), where $m \sim n$. Want $\#$ to be easy to compute. E.g. we might define:

$$\#(s) = \imath(s) \bmod m$$

Hash tables

Suppose our keys are strings (e.g. people's names). Number K of potential keys is vast — number n of actual keys 'currently in use' much smaller.

Really silly idea: Give a way of converting strings s to integers $\imath(s)$ (E.g. treat ASCII characters as digits to base 128). Then store value associated with s in a big array at position $\imath(s)$.

Impractical: K normally far too large, and most of the array would be unused.

More sensible idea: Choose some hash function $\#$ mapping potential keys s to integers $0, \ldots, m-1$ (hash codes), where $m \sim n$.
Want $\#$ to be easy to compute. E.g. we might define:

$$\#(s) = \imath(s) \bmod m$$

Then try to use an array A of size m, storing the entry for key s at position $\#(s)$ in A.

Hashes and clashes

Problem: What if $\#(s) = \#(t)$ for two keys s, t?

Hashes and clashes

Problem: What if $\#(s) = \#(t)$ for two keys s, t?

How likely are clashes to arise? E.g. if we took e.g. $m \sim 5n$ (and accepted the space wastage), would clashes be improbable?

Hashes and clashes

Problem: What if $\#(s) = \#(t)$ for two keys s, t?

How likely are clashes to arise? E.g. if we took e.g. $m \sim 5n$ (and accepted the space wastage), would clashes be improbable?

Example: Keys are people, $m = 366$, $\#(p) =$ birthday of p.

How many people must there be for probability of shared birthday to be $> 1/2$? (Assume uniform distrib.)

Hashes and clashes

Problem: What if $\#(s) = \#(t)$ for two keys s, t?

How likely are clashes to arise? E.g. if we took e.g. $m \sim 5n$ (and accepted the space wastage), would clashes be improbable?

Example: Keys are people, $m = 366$, $\#(p) =$ birthday of p.

How many people must there be for probability of shared birthday to be $> 1/2$? (Assume uniform distrib.)

Answer: Just 23! (Sometimes called the birthday paradox.)

See CLRS 5.4.1 for analysis (if you're interested).

Hashes and clashes

Problem: What if $\#(s) = \#(t)$ for two keys s, t?

How likely are clashes to arise? E.g. if we took e.g. $m \sim 5n$ (and accepted the space wastage), would clashes be improbable?

Example: Keys are people, $m = 366$, $\#(p) =$ birthday of p.

How many people must there be for probability of shared birthday to be $> 1/2$? (Assume uniform distrib.)

Answer: Just 23! (Sometimes called the birthday paradox.)

See CLRS 5.4.1 for analysis (if you're interested).

Question: In a class of 347 (assuming uniform distrib), what would be the probability of a birthday shared by 2 people? By 3 people? By 4, 5, 6, 7, ...?

Hashes and clashes

Problem: What if $\#(s) = \#(t)$ for two keys s, t?

How likely are clashes to arise? E.g. if we took e.g. $m \sim 5n$ (and accepted the space wastage), would clashes be improbable?

Example: Keys are people, $m = 366$, $\#(p) =$ birthday of p.

How many people must there be for probability of shared birthday to be $> 1/2$? (Assume uniform distrib.)

Answer: Just 23! (Sometimes called the birthday paradox.)

See CLRS 5.4.1 for analysis (if you're interested).

Question: In a class of 347 (assuming uniform distrib), what would be the probability of a birthday shared by 2 people? By 3 people? By 4, 5, 6, 7, ...?

2	3	4	5	6	7
$> (100 - 10^{-123})\%$	$> 99.9999\%$	$> 99.8\%$	66%	15%	2%

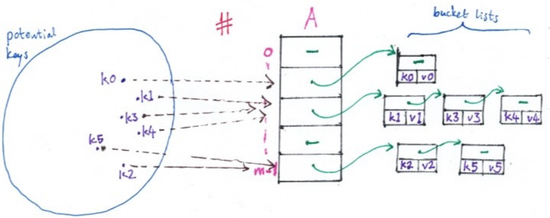
Dealing with clashes

So we must accept clashes (a.k.a. collisions) as a fact of life.

Dealing with clashes

So we must accept clashes (a.k.a. collisions) as a fact of life.

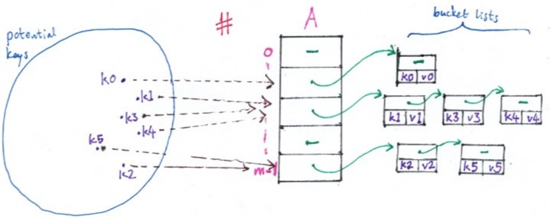
Solution 1: Store a list of entries (or bucket) for each hash value.



Dealing with clashes

So we must accept clashes (a.k.a. collisions) as a fact of life.

Solution 1: Store a list of entries (or bucket) for each hash value.

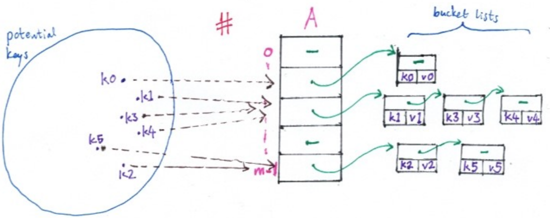


(Omit value components if it's just a set.)

Dealing with clashes

So we must accept clashes (a.k.a. collisions) as a fact of life.

Solution 1: Store a list of entries (or bucket) for each hash value.



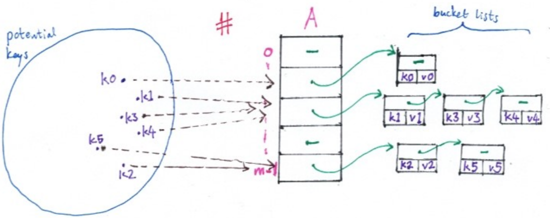
(Omit value components if it's just a set.)

Write n for number of entries, m for array size.
The ratio $\alpha = n/m$ is called the load on the hash table:
may be ≤ 1 or > 1.

Dealing with clashes

So we must accept clashes (a.k.a. collisions) as a fact of life.

Solution 1: Store a list of entries (or bucket) for each hash value.



(Omit value components if it's just a set.)

Write n for number of entries, m for array size.
The ratio $\alpha = n/m$ is called the load on the hash table:
may be ≤ 1 or > 1.

If we've decided on a desired load α, can 'expand-and-rehash' any time n gets too large (amortized cost is reasonable).

Bucket-list hash tables: some analysis

Recall: n table entries, m hash codes, $\alpha = n/m$.
Write b_i for number of entries in ith bucket.

Bucket-list hash tables: some analysis

Recall: n table entries, m hash codes, $\alpha = n/m$.
Write b_i for number of entries in ith bucket.

Let's analyse average time for an unsuccessful lookup.
Assume that for k not in the table, $\#(k)$ equally likely to be any of the m hash codes.

Bucket-list hash tables: some analysis

Recall: n table entries, m hash codes, $\alpha = n/m$.
Write b_i for number of entries in ith bucket.

Let's analyse average time for an unsuccessful lookup.
Assume that for k not in the table, $\#(k)$ equally likely to be any of the m hash codes.

If $\#(k) = i$, lookup will do b_i key comparisons if unsuccessful.

So average number of key comparisons is

$$\frac{1}{m}\sum_{i=0}^{m-1} b_i \;=\; n/m \;=\; \alpha$$

Bucket-list hash tables: some analysis

Recall: n table entries, m hash codes, $\alpha = n/m$.
Write b_i for number of entries in ith bucket.

Let's analyse average time for an unsuccessful lookup.
Assume that for k not in the table, $\#(k)$ equally likely to be any of the m hash codes.

If $\#(k) = i$, lookup will do b_i key comparisons if unsuccessful.

So average number of key comparisons is

$$\frac{1}{m}\sum_{i=0}^{m-1} b_i \;=\; n/m \;=\; \alpha$$

If computing $\#(k)$ itself takes $O(1)$ time, conclude that average time for unsuccessful lookup is $\Theta(\alpha)$. (Thinking of $\alpha \to \infty$.)

Bucket-list hash tables: some analysis

Recall: n table entries, m hash codes, $\alpha = n/m$.
Write b_i for number of entries in ith bucket.

Let's analyse average time for an unsuccessful lookup.
Assume that for k not in the table, $\#(k)$ equally likely to be any of the m hash codes.

If $\#(k) = i$, lookup will do b_i key comparisons if unsuccessful.

So average number of key comparisons is

$$\frac{1}{m}\sum_{i=0}^{m-1} b_i \;=\; n/m \;=\; \alpha$$

If computing $\#(k)$ itself takes $O(1)$ time, conclude that average time for unsuccessful lookup is $\Theta(\alpha)$. (Thinking of $\alpha \to \infty$.)

Can also show the same for successful lookup, assuming all keys present in table are equally likely. See CLRS 11.2.

Making a *proper* hash of it

Rarely true that all *potential* keys (e.g. strings) 'equally probable'. But in the interests of 'balancing' our hash table, we'd like the hash codes $0, \ldots, m-1$ to be all equally likely.

Making a *proper* hash of it

Rarely true that all *potential* keys (e.g. strings) 'equally probable'. But in the interests of 'balancing' our hash table, we'd like the hash codes $0, \ldots, m-1$ to be all equally likely.

Bad choice: $\#(s) = \imath(s)$ mod 128. Effectively just last character of s. So avoid powers of two!

Making a *proper* hash of it

Rarely true that all *potential* keys (e.g. strings) 'equally probable'. But in the interests of 'balancing' our hash table, we'd like the hash codes $0, \ldots, m-1$ to be all equally likely.

Bad choice: $\#(s) = \imath(s) \bmod 128$. Effectively just last character of s. So avoid powers of two!

Also not great: $\#(s) = \imath(s) \bmod 127$. Gives $\#(s) = \#(t)$ whenever s, t are anagrams. So $\#(\text{`algorithms'}) = \#(\text{`logarithms'})$.

Making a *proper* hash of it

Rarely true that all *potential* keys (e.g. strings) 'equally probable'. But in the interests of 'balancing' our hash table, we'd like the hash codes $0, \ldots, m-1$ to be all equally likely.

Bad choice: $\#(s) = \imath(s) \bmod 128$. Effectively just last character of s. So avoid powers of two!

Also not great: $\#(s) = \imath(s) \bmod 127$. Gives $\#(s) = \#(t)$ whenever s, t are anagrams. So $\#(\text{'algorithms'}) = \#(\text{'logarithms'})$.

Better: $\#(s) = \imath(s) \bmod 97$. Primes not too close to powers of two are reasonable.

Making a *proper* hash of it

Rarely true that all *potential* keys (e.g. strings) 'equally probable'. But in the interests of 'balancing' our hash table, we'd like the hash codes $0, \ldots, m-1$ to be all equally likely.

Bad choice: $\#(s) = \imath(s)$ mod 128. Effectively just last character of s. So avoid powers of two!

Also not great: $\#(s) = \imath(s)$ mod 127. Gives $\#(s) = \#(t)$ whenever s, t are anagrams. So $\#(\text{'algorithms'}) = \#(\text{'logarithms'})$.

Better: $\#(s) = \imath(s)$ mod 97. Primes not too close to powers of two are reasonable.

Just the start of the delicate art of hash function design...

Making a *proper* hash of it

Rarely true that all *potential* keys (e.g. strings) 'equally probable'. But in the interests of 'balancing' our hash table, we'd like the hash codes $0, \ldots, m-1$ to be all equally likely.

Bad choice: $\#(s) = \imath(s) \bmod 128$. Effectively just last character of s. So avoid powers of two!

Also not great: $\#(s) = \imath(s) \bmod 127$. Gives $\#(s) = \#(t)$ whenever s, t are anagrams. So $\#(\text{'algorithms'}) = \#(\text{'logarithms'})$.

Better: $\#(s) = \imath(s) \bmod 97$. Primes not too close to powers of two are reasonable.

Just the start of the delicate art of hash function design...

But whatever we do, worst case (all keys hashing to same code) is always terrible. A malicious user who knew your hash function could force this to happen ...

Open addressing and probing

Solution 2: Rather than keeping bucket lists outside the hash table, store all items within the table itself (open addressing).

Open addressing and probing

Solution 2: Rather than keeping bucket lists outside the hash table, store all items within the table itself (open addressing).

To deal with clashes, we use not just a simple hash function $\#(k)$, but a function $\#(k, i)$ where $0 \leq i < m$. For a key k:

- $\#(k, 0)$ is our first choice of hash value,
- $\#(k, 1)$ is our second choice, etc.

Open addressing and probing

Solution 2: Rather than keeping bucket lists outside the hash table, store all items within the table itself (open addressing).

To deal with clashes, we use not just a simple hash function $\#(k)$, but a function $\#(k, i)$ where $0 \leq i < m$. For a key k:

- $\#(k, 0)$ is our first choice of hash value,
- $\#(k, 1)$ is our second choice, etc.

so that $\#(k, 0), \#(k, 1), \ldots, \#(k, m-1)$ is a permutation of $0, \ldots, m-1$. (Ideally, for a randomly chosen k, all $m!$ permutations should be equally likely.)

Open addressing and probing

Solution 2: Rather than keeping bucket lists outside the hash table, store all items within the table itself (open addressing).

To deal with clashes, we use not just a simple hash function $\#(k)$, but a function $\#(k, i)$ where $0 \leq i < m$. For a key k:

- $\#(k, 0)$ is our first choice of hash value,
- $\#(k, 1)$ is our second choice, etc.

so that $\#(k, 0), \#(k, 1), \ldots, \#(k, m-1)$ is a permutation of $0, \ldots, m-1$. (Ideally, for a randomly chosen k, all $m!$ permutations should be equally likely.)

To **insert** an item e with key k, probe $A[\#(k, 0)], A[\#(k, 1)], \ldots$ until we find a free slot $A[\#(k, i)]$, then put e there.

Open addressing and probing

Solution 2: Rather than keeping bucket lists outside the hash table, store all items within the table itself (open addressing).

To deal with clashes, we use not just a simple hash function $\#(k)$, but a function $\#(k, i)$ where $0 \leq i < m$. For a key k:

- $\#(k, 0)$ is our first choice of hash value,
- $\#(k, 1)$ is our second choice, etc.

so that $\#(k, 0), \#(k, 1), \ldots, \#(k, m-1)$ is a permutation of $0, \ldots, m-1$. (Ideally, for a randomly chosen k, all $m!$ permutations should be equally likely.)

To **insert** an item e with key k, probe $A[\#(k, 0)], A[\#(k, 1)], \ldots$ until we find a free slot $A[\#(k, i)]$, then put e there.

To **lookup** an item with key k, probe $A[\#(k, 0)], A[\#(k, 1)], \ldots$ until we find either an item with key k, or free cell (lookup failed).

Probing: example

Let's use an array A of size $m = 10$ to store a set of integers.

0	1	2	3	4	5	6	7	8	9

Probe function: $\#(k, i) = (k + i) \bmod 10$.

Probing: example

Let's use an array A of size $m = 10$ to store a set of integers.

0	1	2	3	4	5	6	7	8	9

Probe function: $\#(k, i) = (k + i) \bmod 10$.

insert(49).

Probing: example

Let's use an array A of size $m = 10$ to store a set of integers.

0	1	2	3	4	5	6	7	8	9

Probe function: $\#(k, i) = (k + i) \bmod 10$.

insert(49). $\#(49, 0) = 9$:

Probing: example

Let's use an array A of size $m = 10$ to store a set of integers.

0	1	2	3	4	5	6	7	8	9
									49

Probe function: $\#(k, i) = (k + i) \bmod 10$.

insert(49). $\#(49, 0) = 9$: free.

Probing: example

Let's use an array A of size $m = 10$ to store a set of integers.

0	1	2	3	4	5	6	7	8	9
									49

Probe function: $\#(k, i) = (k + i) \bmod 10$.

insert(49). $\#(49, 0) = 9$: free.

insert(28).

Probing: example

Let's use an array A of size $m = 10$ to store a set of integers.

0	1	2	3	4	5	6	7	8	9
									49

Probe function: $\#(k, i) = (k + i) \bmod 10$.

insert(49). $\#(49, 0) = 9$: free.

insert(28). $\#(28, 0) = 8$:

Probing: example

Let's use an array A of size $m = 10$ to store a set of integers.

0	1	2	3	4	5	6	7	8	9
								28	49

Probe function: $\#(k, i) = (k + i) \bmod 10$.

insert(49). $\#(49, 0) = 9$: free.

insert(28). $\#(28, 0) = 8$: free.

Probing: example

Let's use an array A of size $m = 10$ to store a set of integers.

0	1	2	3	4	5	6	7	8	9
								28	49

Probe function: $\#(k, i) = (k + i) \bmod 10$.

insert(49). $\#(49, 0) = 9$: free.

insert(28). $\#(28, 0) = 8$: free.

insert(58).

Probing: example

Let's use an array A of size $m = 10$ to store a set of integers.

0	1	2	3	4	5	6	7	8	9
								28	49

Probe function: $\#(k, i) = (k + i) \bmod 10$.

insert(49). $\#(49, 0) = 9$: free.

insert(28). $\#(28, 0) = 8$: free.

insert(58). $\#(58, 0) = 8$: taken.

Probing: example

Let's use an array A of size $m = 10$ to store a set of integers.

0	1	2	3	4	5	6	7	8	9
								28	49

Probe function: $\#(k, i) = (k + i) \bmod 10$.

insert(49). $\#(49, 0) = 9$: free.

insert(28). $\#(28, 0) = 8$: free.

insert(58). $\#(58, 0) = 8$: taken. $\#(58, 1) = 9$: taken.

Probing: example

Let's use an array A of size $m = 10$ to store a set of integers.

0	1	2	3	4	5	6	7	8	9
58								28	49

Probe function: $\#(k, i) = (k + i) \bmod 10$.

insert(49). $\#(49, 0) = 9$: free.

insert(28). $\#(28, 0) = 8$: free.

insert(58). $\#(58, 0) = 8$: taken. $\#(58, 1) = 9$: taken.
$\#(58, 2) = 0$: free.

Probing: example

Let's use an array A of size $m = 10$ to store a set of integers.

0	1	2	3	4	5	6	7	8	9
58								28	49

Probe function: $\#(k, i) = (k + i) \bmod 10$.

insert(49). $\#(49, 0) = 9$: free.

insert(28). $\#(28, 0) = 8$: free.

insert(58). $\#(58, 0) = 8$: taken. $\#(58, 1) = 9$: taken.
$\#(58, 2) = 0$: free.

contains(28).

Probing: example

Let's use an array A of size $m = 10$ to store a set of integers.

0	1	2	3	4	5	6	7	8	9
58								28	49

Probe function: $\#(k, i) = (k + i) \bmod 10$.

insert(49). $\#(49, 0) = 9$: free.

insert(28). $\#(28, 0) = 8$: free.

insert(58). $\#(58, 0) = 8$: taken. $\#(58, 1) = 9$: taken.
$\#(58, 2) = 0$: free.

contains(28). $\#(28, 0) = 8$, $A[8] = 28$.

Probing: example

Let's use an array A of size $m = 10$ to store a set of integers.

0	1	2	3	4	5	6	7	8	9
58								28	49

Probe function: $\#(k, i) = (k + i) \bmod 10$.

insert(49). $\#(49, 0) = 9$: free.

insert(28). $\#(28, 0) = 8$: free.

insert(58). $\#(58, 0) = 8$: taken. $\#(58, 1) = 9$: taken.
$\#(58, 2) = 0$: free.

contains(28). $\#(28, 0) = 8$, $A[8] = 28$. So true.

Probing: example

Let's use an array A of size $m = 10$ to store a set of integers.

0	1	2	3	4	5	6	7	8	9
58								28	49

Probe function: $\#(k, i) = (k + i) \bmod 10$.

insert(49). $\#(49, 0) = 9$: free.

insert(28). $\#(28, 0) = 8$: free.

insert(58). $\#(58, 0) = 8$: taken. $\#(58, 1) = 9$: taken.
$\#(58, 2) = 0$: free.

contains(28). $\#(28, 0) = 8$, $A[8] = 28$. So true.

contains(58).

Probing: example

Let's use an array A of size $m = 10$ to store a set of integers.

0	1	2	3	4	5	6	7	8	9
58								28	49

Probe function: $\#(k, i) = (k + i) \bmod 10$.

insert(49). $\#(49, 0) = 9$: free.

insert(28). $\#(28, 0) = 8$: free.

insert(58). $\#(58, 0) = 8$: taken. $\#(58, 1) = 9$: taken.
$\#(58, 2) = 0$: free.

contains(28). $\#(28, 0) = 8$, $A[8] = 28$. So true.

contains(58). $\#(58, 0) = 8$, $A[8] = 28 \neq 58$.

Probing: example

Let's use an array A of size $m = 10$ to store a set of integers.

0	1	2	3	4	5	6	7	8	9
58								28	49

Probe function: $\#(k, i) = (k + i) \bmod 10$.

insert(49). $\#(49, 0) = 9$: free.

insert(28). $\#(28, 0) = 8$: free.

insert(58). $\#(58, 0) = 8$: taken. $\#(58, 1) = 9$: taken.
$\#(58, 2) = 0$: free.

contains(28). $\#(28, 0) = 8$, $A[8] = 28$. So true.

contains(58). $\#(58, 0) = 8$, $A[8] = 28 \neq 58$.
$\#(58, 1) = 9$, $A[9] = 49 \neq 58$.

Probing: example

Let's use an array A of size $m = 10$ to store a set of integers.

0	1	2	3	4	5	6	7	8	9
58								28	49

Probe function: $\#(k, i) = (k + i) \bmod 10$.

insert(49). $\#(49, 0) = 9$: free.

insert(28). $\#(28, 0) = 8$: free.

insert(58). $\#(58, 0) = 8$: taken. $\#(58, 1) = 9$: taken.
$\#(58, 2) = 0$: free.

contains(28). $\#(28, 0) = 8$, $A[8] = 28$. So true.

contains(58). $\#(58, 0) = 8$, $A[8] = 28 \neq 58$.
$\#(58, 1) = 9$, $A[9] = 49 \neq 58$.
$\#(58, 2) = 0$: $A[0] = 58$. So true.

Probing: example

Let's use an array A of size $m = 10$ to store a set of integers.

0	1	2	3	4	5	6	7	8	9
58								28	49

Probe function: $\#(k, i) = (k + i) \bmod 10$.

insert(49). $\#(49, 0) = 9$: free.

insert(28). $\#(28, 0) = 8$: free.

insert(58). $\#(58, 0) = 8$: taken. $\#(58, 1) = 9$: taken.
$\#(58, 2) = 0$: free.

contains(28). $\#(28, 0) = 8$, $A[8] = 28$. So true.

contains(58). $\#(58, 0) = 8$, $A[8] = 28 \neq 58$.
$\#(58, 1) = 9$, $A[9] = 49 \neq 58$.
$\#(58, 2) = 0$: $A[0] = 58$. So true.

contains(39).

Probing: example

Let's use an array A of size $m = 10$ to store a set of integers.

0	1	2	3	4	5	6	7	8	9
58								28	49

Probe function: $\#(k, i) = (k + i) \bmod 10$.

insert(49). $\#(49, 0) = 9$: free.

insert(28). $\#(28, 0) = 8$: free.

insert(58). $\#(58, 0) = 8$: taken. $\#(58, 1) = 9$: taken.
$\#(58, 2) = 0$: free.

contains(28). $\#(28, 0) = 8$, $A[8] = 28$. So true.

contains(58). $\#(58, 0) = 8$, $A[8] = 28 \neq 58$.
$\#(58, 1) = 9$, $A[9] = 49 \neq 58$.
$\#(58, 2) = 0$: $A[0] = 58$. So true.

contains(39). $\#(39, 0) = 9$, $A[9] = 49 \neq 39$.

Probing: example

Let's use an array A of size $m = 10$ to store a set of integers.

0	1	2	3	4	5	6	7	8	9
58								28	49

Probe function: $\#(k, i) = (k + i) \bmod 10$.

insert(49). $\#(49, 0) = 9$: free.

insert(28). $\#(28, 0) = 8$: free.

insert(58). $\#(58, 0) = 8$: taken. $\#(58, 1) = 9$: taken.
$\#(58, 2) = 0$: free.

contains(28). $\#(28, 0) = 8$, $A[8] = 28$. So true.

contains(58). $\#(58, 0) = 8$, $A[8] = 28 \neq 58$.
$\#(58, 1) = 9$, $A[9] = 49 \neq 58$.
$\#(58, 2) = 0$: $A[0] = 58$. So true.

contains(39). $\#(39, 0) = 9$, $A[9] = 49 \neq 39$.
$\#(39, 1) = 0$, $A[0] = 58 \neq 39$.

Probing: example

Let's use an array A of size $m = 10$ to store a set of integers.

0	1	2	3	4	5	6	7	8	9
58								28	49

Probe function: $\#(k, i) = (k + i) \bmod 10$.

insert(49). $\#(49, 0) = 9$: free.

insert(28). $\#(28, 0) = 8$: free.

insert(58). $\#(58, 0) = 8$: taken. $\#(58, 1) = 9$: taken.
$\#(58, 2) = 0$: free.

contains(28). $\#(28, 0) = 8$, $A[8] = 28$. So true.

contains(58). $\#(58, 0) = 8$, $A[8] = 28 \neq 58$.
$\#(58, 1) = 9$, $A[9] = 49 \neq 58$.
$\#(58, 2) = 0$: $A[0] = 58$. So true.

contains(39). $\#(39, 0) = 9$, $A[9] = 49 \neq 39$.
$\#(39, 1) = 0$, $A[0] = 58 \neq 39$.
$\#(39, 2) = 1$, $A[1]$ free. So false.

Probing: pros and cons

- Expected number of probes for **insert** (and hence for **lookup**) stays low until table is nearly full. (Can show it's $1/(1-\alpha)$ for unsuccessful lookup; less for successful one.)

Probing: pros and cons

- Expected number of probes for **insert** (and hence for **lookup**) stays low until table is nearly full. (Can show it's $1/(1-\alpha)$ for unsuccessful lookup; less for successful one.)
- No need for pointers. The memory this saves can be 'spent' on increasing table size *m* and so decreasing load α ...
  So compared to bucket lists, get faster **lookup** for same amount of memory.

Probing: pros and cons

- Expected number of probes for **insert** (and hence for **lookup**) stays low until table is nearly full. (Can show it's $1/(1-\alpha)$ for unsuccessful lookup; less for successful one.)
- No need for pointers. The memory this saves can be 'spent' on increasing table size m and so decreasing load $\alpha \ldots$
  So compared to bucket lists, get faster **lookup** for same amount of memory.
- However, **delete** is a pain for the probing approach.

Probing: pros and cons

- Expected number of probes for **insert** (and hence for **lookup**) stays low until table is nearly full. (Can show it's $1/(1-\alpha)$ for unsuccessful lookup; less for successful one.)
- No need for pointers. The memory this saves can be 'spent' on increasing table size *m* and so decreasing load α ...
  So compared to bucket lists, get faster **lookup** for same amount of memory.
- However, **delete** is a pain for the probing approach.
- Design of probing functions is again a delicate art (*linear probing*, *quadratic probing*, *double hashing*, ...).

Probing: pros and cons

- Expected number of probes for **insert** (and hence for **lookup**) stays low until table is nearly full. (Can show it's $1/(1-\alpha)$ for unsuccessful lookup; less for successful one.)
- No need for pointers. The memory this saves can be 'spent' on increasing table size m and so decreasing load α ...
  So compared to bucket lists, get faster **lookup** for same amount of memory.
- However, **delete** is a pain for the probing approach.
- Design of probing functions is again a delicate art (*linear probing*, *quadratic probing*, *double hashing*, ...).

See CLRS 11.4 for more details.

Radical alternative: Perfect hashing

- *All* the approaches we've mentioned are bad in the worst case: size of bucket/sequence of probes can be of length n.

Radical alternative: Perfect hashing

- *All* the approaches we've mentioned are bad in the worst case: size of bucket/sequence of probes can be of length n.
- Even in typical cases, probably *some* buckets will be large relative to α. (Birthday paradox!)

Radical alternative: Perfect hashing

- *All* the approaches we've mentioned are bad in the worst case: size of bucket/sequence of probes can be of length n.
- Even in typical cases, probably *some* buckets will be large relative to α. (Birthday paradox!)

If we could avoid clashes altogether, these problems would vanish! Would get worst-case $\Theta(1)$ lookup.

Radical alternative: Perfect hashing

- *All* the approaches we've mentioned are bad in the worst case: size of bucket/sequence of probes can be of length n.
- Even in typical cases, probably *some* buckets will be large relative to α. (Birthday paradox!)

If we could avoid clashes altogether, these problems would vanish! Would get worst-case $\Theta(1)$ lookup.

If set of keys is static (no **insert/delete** required), may be worth finding a perfect hash function (no clashes) for this set of keys.

Radical alternative: Perfect hashing

- *All* the approaches we've mentioned are bad in the worst case: size of bucket/sequence of probes can be of length n.
- Even in typical cases, probably *some* buckets will be large relative to α. (Birthday paradox!)

If we could avoid clashes altogether, these problems would vanish! Would get worst-case $\Theta(1)$ lookup.

If set of keys is static (no **insert/delete** required), may be worth finding a perfect hash function (no clashes) for this set of keys.

As part of Coursework 1, we'll explore a modern approach to perfect hashing.

Reading:

Roughgarden 12.1-12.4 (good!)
CLRS Chapter 11, omitting theorems and their proofs, except for Theorem 11.1 which corresponds to slide 8.