CS 261 – Data Structures

Hash Tables

Open Address Hashing

ADT Dictionaries

computer |kəmˈpyoōtər|

noun

- an electronic device for storing and processing data...

- a person who makes calculations, esp. with a calculating machine.

Dictionaries

computer |kəmˈpyoōtər| **key**

noun

- an electronic device for storing and processing data...

- a person who makes calculations, esp. with a calculating machine.

Dictionaries

computer |kəmˈpyoōtər| **value**

noun

- an electronic device for storing and processing data...
- a person who makes calculations, esp. with a calculating machine.

How to implement dictionaries?

Hash Tables

Similar to dynamic arrays except:

1. Elements can be indexed by their **keys** whose type may differ from integer
2. In general, a single position may hold more than one element

Computing a Hash Table Index: 2 Steps

1. Transform the key to an integer
   - by using the hash function

2. Map the resulting integer to a valid hash table index
   - by using the remainder of dividing the integer with the table size

Example

Say, we're storing names:

Angie

Joe

Abigail

Linda

Mark

Max

Robert

John

0	Angie, Robert
1	Linda
2	Joe, Max, John
3	
4	Abigail, Mark

Example: Computing the Hash Table Index

Storing names:

– Compute an integer from the name

– Map the integer to an index in a table

Hash Function

Hash function maps the keys to integers

Hash Function: Types

Mapping:

Map (a part of) the key into an integer

– Example: a letter to its position in the alphabet

Hash Function: Types

Folding:

Parts of the key combined by operations, such as add, multiply, shift, XOR, etc.

– Example: summing the values of each character in a string

Hash Function: Types

Shifting + Folding:

Shift left the name
to get rid of repeating low-order bits
or
Shift right the name
to multiply by powers of 2

Example: if keys are always even, shift off the low order bit

Hash Function: Combinations

Map, Fold, and Shift combination

Key	Mapped chars	Folded	Shifted and Folded
eat	5 + 1 + 20	26	20 + 2 + 20 = 42
ate	1 + 20 + 5	26	4 + 40 + 5 = 49
tea	20 + 5 + 1	26	80 + 10 + 1 = 91

Hash Function: Types

Casts:

Converting a numeric type into an integer

– Example: casting a character to an integer to get its ASCII value

Hash Functions: Examples

– Key = Character:

char value cast to an int → it's ASCII value

– Key = Date:

value associated with the current time

– Key = Double:

value generated by its bitwise representation

Hash Functions: Examples

– Key = Integer:

the int value itself

– Key = String:

a folded sum of the character values

– Key = URL:

the hash code of the host name

Step 2: Mapping to a Valid Index

- Use modulus operator (%) with table size:
  - Example:

```
idx = hash(val) % size;
```

- Must be sure that the final result is positive
  - Use only positive arithmetic or take absolute value

Step 2: Mapping to a Valid Index

To get a good distribution of indices,

prime numbers make the best table sizes.

– Example: if you have 1000 elements, a table size of 997 or 1009 is preferable

Hash Tables: Ideal Case

1. **Perfect hash function**: each data element hashes to a unique hash index

2. Table size equal to (or slightly larger than) number of elements

Perfect Hashing: Example

- Six friends have a club: Alfred, Alessia, Amina, Amy, Andy, and Anne
- Store member names in a six element array
- Convert 3rd letter of each name to an index:

Alfred	`f =  5 % 6 = 5`
Alessia	`e =  4 % 6 = 4`
Amina	`i =  8 % 6 = 2`
Amy	`y = 24 % 6 = 0`
Andy	`d =  3 % 6 = 3`
Anne	`n = 13 % 6 = 1`

Hash Tables: Collisions

- Unless the data is known in advance, the ideal case is usually not possible
- A ***collision*** is when two or more different keys result in the same hash table index
- How do we deal with collisions?

Indexing: Faster Than Searching

- Can convert a name (e.g., Alessia) into a number (e.g., 4) in constant time
- Faster than searching
- Allows for $O(1)$ time operations

Indexing: Faster Than Searching

Becomes complicated for new elements:

–Alan wants to join the club:

'a' = 0 → same as Amy

–Also:

Al wants to join → no third letter!

Hash Tables: Resolving Collisions

There are two general approaches to resolving collisions:

1. Open address hashing: if a spot is full, probe for next empty spot

2. Chaining (or buckets): keep a collection at each table entry

Open Address Hashing

Open Address Hashing

- All values are stored in an array
- Hash value is used to find initial index to try
- If that position is filled, next position is examined, then next, and so on until an empty position is filled

Open Address Hashing

- The process of looking for an empty position is termed *probing*,
- Specifically, we consider linear probing
- There are other probing algorithms, but we won't consider them

Open Address Hashing: Example

Eight element table using the third-letter hash function:

Already added: Amina, Andy, Alessia, Alfred, and Aspen

Amina			Andy	Alessia	Alfred		Aspen
0	1	2	3	4	5	6	7
aiqy	bjrz	cks	dlt	emu	fnv	gpw	hpq

Open Address Hashing: Adding

Now we need to add: Aimee

Hashes to

Placed here

Amina			Andy	Alessia	Alfred	Aimee	Aspen
0	1	2	3	4	5	6	7
aiqy	bjrz	cks	dlt	emu	fnv	gpw	hpq

The hashed index position (4) is filled by Alessia: so we probe to find next free location

Open Address Hashing: Adding

Suppose Anne wants to join:

Add: Anne Hashes to

???

Amina			Andy	Alessia	Alfred	Aimee	Aspen
0	1	2	3	4	5	6	7
aiqy	bjrz	cks	dlt	emu	fnv	gpw	hpq

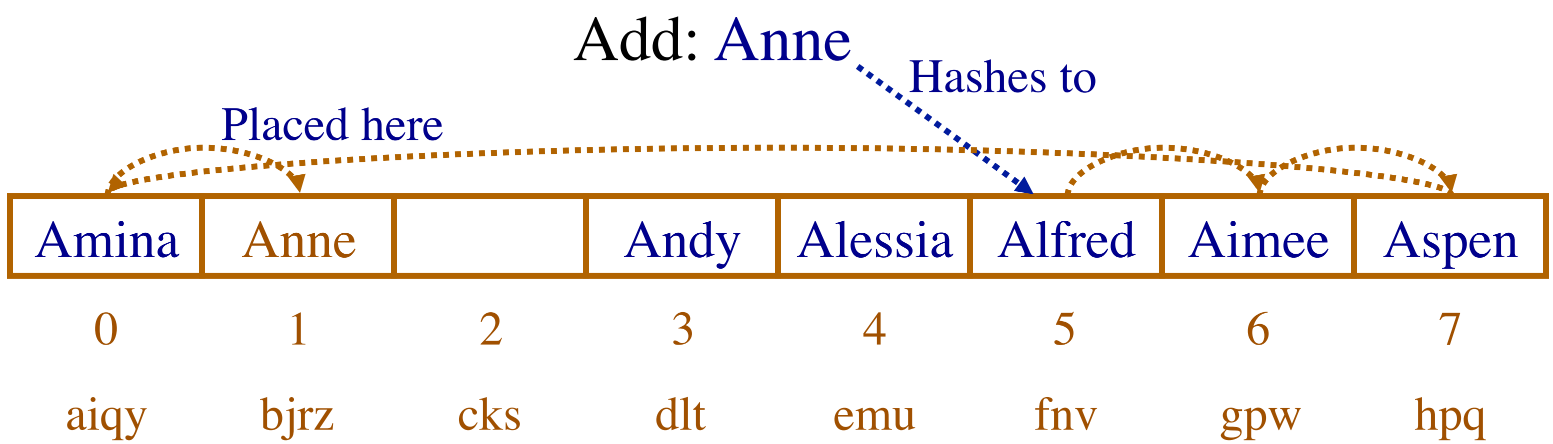
The hashed index position (5) is filled by Alfred:

Probe to find next free location

What happens when we reach the end of the array?

Open Address Hashing: Adding

Suppose Anne wants to join:

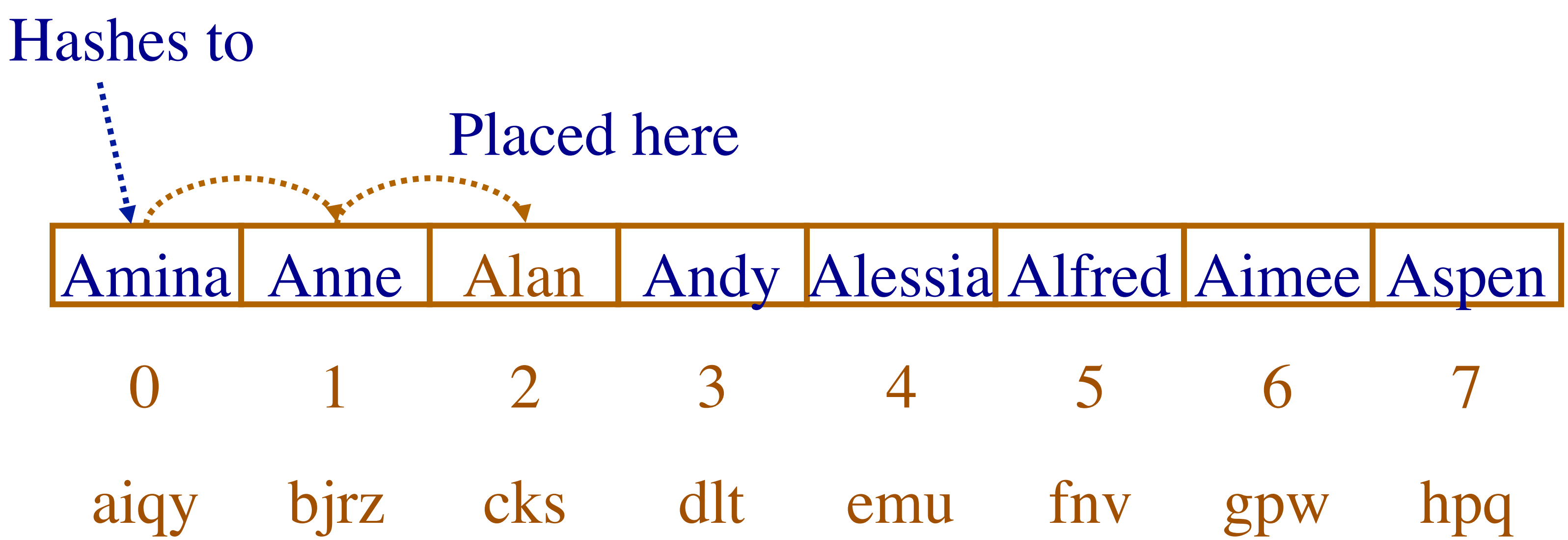


The hashed index position (5) is filled by Alfred:

- –Probe to find next free location
- –When we get to end of array, wrap around to the beginning
- –Eventually, find position at index 1 open

Open Address Hashing: Adding

Finally, Alan wants to join:



The hashed index position (0) is filled by Amina:

– Probing finds last free position (index 2)
– Collection is now completely filled

Open Address Hashing: Contains

- Hash to find initial index, probe forward examining each location until value is found, or *empty location is found*.
- Example, search for: Amina, Aimee, Anne...

Amina	Anne	Alan	Andy	Alessia	Alfred	Aimee	Aspen
0	1	2	3	4	5	6	7
aiqy	bjrz	cks	dlt	emu	fnv	gpw	hpq

- Notice that search time is not uniform

Open Address Hashing: Remove

- Remove is tricky: Can't leave this place empty
- What happens if we delete Anne, then search for Alan?

Remove: Anne

Amina	~~Anne~~	Alan	Andy	Alessia	Alfred	Aimee	Aspen
0-aiqy	1-bjrz	2-cks	3-dlt	4-emu	5-fnv	6-gpw	7-hpq

Find: Alan

Hashes to

Probing finds null entry → Alan not found

Amina		Alan	Andy	Alessia	Alfred	Aimee	Aspen
0-aiqy	1-bjrz	2-cks	3-dlt	4-emu	5-fnv	6-gpw	7-hpq

Open Address Hashing: Handling Remove

- Replace removed item with "tombstone"
  - Special value that marks deleted entry
  - Can be replaced when adding new entry
  - But doesn't halt search during contains (remove)

Find: Alan

Hashes to

Probing skips tombstone → Alan found

Amina	_TS_	Alan	Andy	Alessia	Alfred	Aimee	Aspen
0-aiqy	1-bjrz	2-cks	3-dlt	4-emu	5-fnv	6-gpw	7-hpq

Hash Table Size: Load Factor

Load factor:

# of elements

Load factor $\lambda = n \, / \, m$ Size of table

–Load factor is the average number of elements at each table entry

–For open address hashing, load factor is between 0 and 1 (often somewhere between 0.5 and 0.75)

–For chaining, load factor can be greater than 1

–Want the load factor to remain small

Large Load Factor: What to do?

- Common solution: When load factor becomes too large (say, bigger than 0.75) → Reorganize
- Create new table with twice the number of positions
- Copy each element, rehashing using the new table size, placing elements in new table
- Delete the old table

Hash Tables: Algorithmic Complexity

- Assumptions:
  - Time to compute hash function is constant
  - Worst case analysis → All values hash to same position
  - Best case analysis → Hash function uniformly distributes the values

Hash Tables: Algorithmic Complexity

- Find element operation:
  - –Worst case for open addressing → O(n)
  - –Best case for open addressing → O(1)

Hash Tables: Average Case

- What about average case?
- Turns out, it's

$$1 / (1 - \lambda)$$

- So keeping load factor small is very important

λ	$1 / (1 - \lambda)$
0.25	1.3
0.5	2.0
0.6	2.5
0.75	4.0
0.85	6.6
0.95	19.0

Difficulties with Hash Tables

- Need to find good hash function → uniformly distributes keys to all indices
- Open address hashing:
  - Need to tell if a position is empty or not
  - One solution → store only pointers
- Open address hashing: problem with removal