Hash Tables

Hash Table Overview

- Purpose: Maintain a set of stuff and provide extremely fast lookup
- Supported Operations: insert, delete, lookup - using a "key" all in O(1) time
- AKA Dictionary (note however, hash tables do not maintain an ordering of it's elements)

Caveats

- Hash Tables are easy to implement badly, violating the 'all operations in O(1) time' constraint
- No guarantee of worst case behavior. Constant time constraint will not exist over all possible data set. Constant time operation is valid for 'non-pathological' data sets (if properly implemented)

De-duplication

- Given: a stream of objects (i.e. records from a file, data passing over a network connection, etc.)
- Goal: remove duplicates (retain only unique items)
- Solution: look up each object in hash table. If it exists, it's a duplicate, otherwise add it. Hash table will contain set of unique items

2-sum problem

- Input: unsorted int array A, target sum t
- Naive solution: Test all combinations of 2, runs in $O(n^2)$ time
- Better solution: sort array A, for each element, determine value t-x. Using binary search, test if t-x exists in A. Runs in O(n log n) time
- Even better solution: insert each element of array A into hash table. For each item x, check if t-x exists in hash. Runs in O(n) time.

Other Applications

- Symbol tables in compilers
- Filter router traffic by blacklist
- Search optimization, i.e. game tree exploration. Use hash table to avoid exploring any configuration more than once

Pre-implementation

- First, identify the universe (U) of elements to be stored (generally very large)
- Goal: keep track of an evolving subset (S) of the universe, where S <= U (generally of reasonable size)

Solutions

Naive Solutions

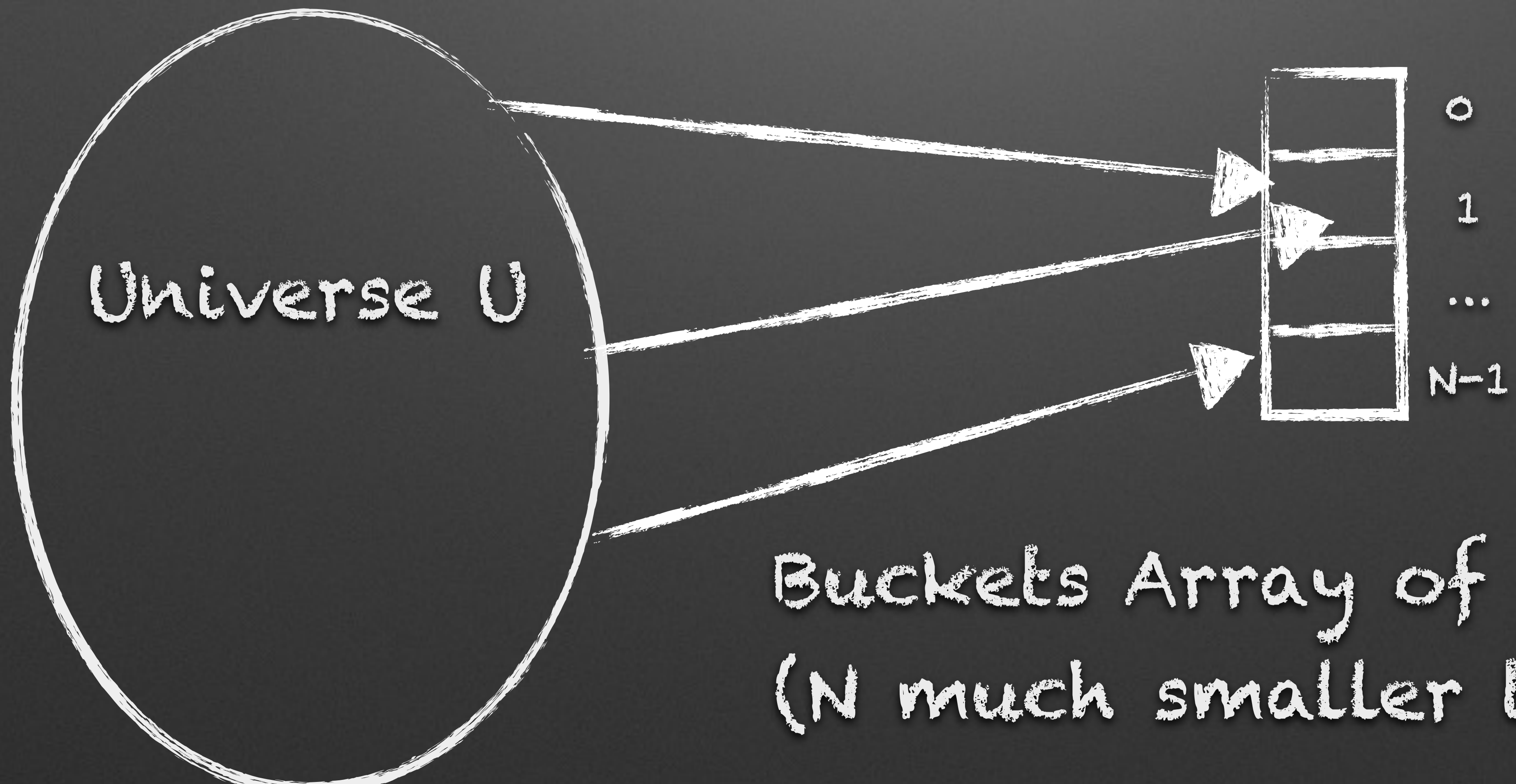
- array indexed by U O(1) operations, but requires O({U}) space - infeasible unless U is very small
- List based solution: requires O({S}) space, but lookup requires O({S}) time (sequencial)

Better Solution

- Select N, where N='number of buckets,' which should be approximately equal to the expected size of {S}
- Choose a hash function h:U -> which maps an element to an index position
- Store element x in array, i.e. Array[hash(x)] = x

Hash Collision

- Collision: hash(A) = hash(B) when A != B
- Collisions are inevitable



Buckets Array of size N,
(N much smaller than U)

Collision Resolution

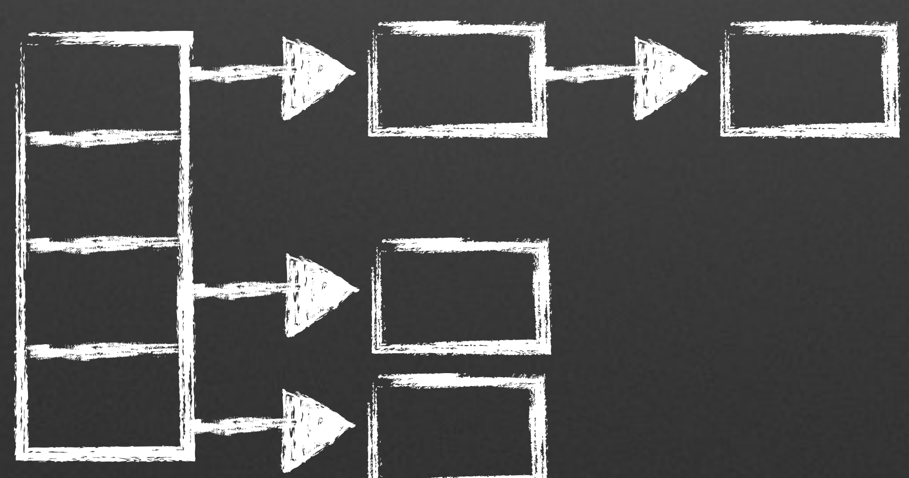
Two Options

- Chaining, (AKA separate chaining): keep linked list in each bucket. Insert is O(1), and deletion and lookup are O(list length).
- Open Addressing: Maintains one object per bucket. Hash function specifies probe sequence. I.e. if collision, re-hash and check next address, until no collision

Chaining

- Each element in the hash table contains a linked list, in the event of collision, colliding items are added to the list

- Better option than open addressing if hash table needs to support delete operation.



Open Addressing

- Linear Probing: If collision, increment hash result until an empty bucket is found.
- Double Hashing: Two hash functions. First result is initial hash. Second hash is used as an additive shift to derive alternate locations in case of collision
- Which method to use? If space is at a premium, prefer open addressing. If deletion is important, may prefer chaining - deletion is 'tricky' with open addressing.

Hash Function

- Hash table performance depends on the choice of the hash function.

Properties of a 'good' hash function

- Should lead to good performance by minimizing collisions, i.e. "spread the data out" (Gold standard, completely random hashing)

- Should be easy to store and very fast to evaluate

Bad Hash Functions

- Example: keys = 10 digit phone numbers. Terrible hash function would be: first three digits
- Mediocre: Take object's in memory address and

Quick & Dirty Hash Function

- Get numeric value equal to or derived from the object key. For string keys, generate number by summing the ascii code for each character. For example:

```
int sascii(String x, int M) {
  char ch[];
  ch = x.toCharArray();
  int xlength = x.length();

  int i, sum;
  for (sum=0, i=0; i < x.length(); i++)
    sum += ch[i];
  return sum % M;
}
```

- Choose n, the size of your hash array, where n should be a prime number within a constant factor of the number of objects in the table
- Apply compression function, to the numeric key value, to obtain hash table index. Modulus function:

```
int h(int x) {
  return x % intSize;
}
```

where intSize is 16 or 32, etc. corresponding to the size of your key.

Resources

- Probability Calculations in Hashing (https://math.dartmouth.edu/archive/m19w03/public_html/Section6-5.pdf) for details
- Sample Hash Functions: http://algoviz.org/OpenDSA/Books/OpenDSA/html/HashFuncExamp.html