CECS 274: Data Structures
Hash Tables

Oscar Morales Ponce

California State University Long Beach

Example

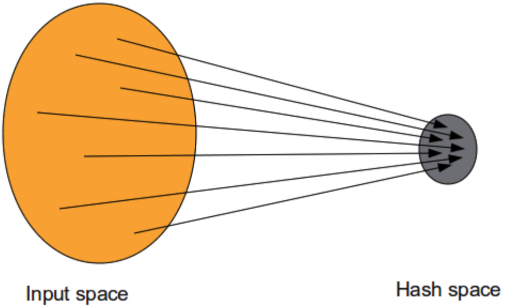
- We can check if the following expression is valid using a Stack (Array-based, Linked-List based)

$$((a + b) * (c * d) + (a + (b * (c/d)))$$

  - How can we replace the terms with the actual values efficiently:
    $a = 5413.13, b = 243.12, c = 4212.12, d = 421.33$
- Suppose you have a list of several millions of unique items. How to search efficiently when the name of the item is known.

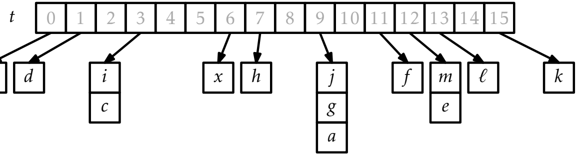
Hash Table

- Hash tables are an efficient method of storing a small number, n, of integers from a large range $U = \{0, \ldots, 2^w - 1\}$.
- The term *hash table* includes a broad range of data structures.
- The *hash value* of a data item x, denoted $\mathrm{hash}(x)$ is a value in the range $\{0, \ldots, 2^d - 1\}$ for some $d \geq 0$



ChainedHashTable: The Basics

- ChainedHashTable
  - State variables:
    - n: Number of elements in a.
    - d: Determines the size of the array
    - t: Array of lists of length 2^d
    - z: A random odd integer
  - Invariant: $n \leq length(t) = 2^d < 3n$ and
    For a pair $(key, value)$, $value$ is uniquely stored in the list at $t[hash(key)]$



ChainedHashTable: $initialize()$

- $initialize()$: Initialize the state variables

initialize()
$d = 1$
$t \leftarrow \text{alloc\_table}(2^d)$
$z \leftarrow \text{random\_odd\_int}()$
$n \leftarrow 0$

ChainedHashTable: $add(key, value)$

- $add(x, v)$: Add the tuple (x, v) in the table if it does not exist

$\mathrm{add}(key, value)$
Check Preconditions
Check that the invariant holds
Insert $Node(key, value)$ in the list $t[hash(key)]$
Increment n by one

ChainedHashTable: $add(key, value)$

$\text{add}(key, value)$
    if $\text{find}(key) \neq nil$ **then return** $false$
    if $n = \text{length}(t)$ **then** $\text{resize}()$
    $t[\text{hash}(key)].\text{add}(0, Node(key, value))$
    $n \leftarrow n + 1$
    return $true$

- The cost of growing is only constant when amortized over a sequence of insertions
- Appending $key, value$ to the list $t[\text{hash}(key)]$ takes only constant time.

ChainedHashTable: $remove(key)$

- $remove(key)$: Remove the element $(key, \cdot)$ from the table if it exists

remove(key)
    Check Preconditions
    Check if the value key exists in $t[hash(key)]$
    Check that the Invariant holds

ChainedHashTable: $remove(key)$

remove(key)
    for i **in** $0, 1, .., t[hash(key)].size() - 1$
        if $t[hash(key)].get(i).key = key$ **then**
            $t[hash(key)].remove(i)$
            $n \leftarrow n - 1$
            if $\text{length}(a) \geq 3 \cdot n$ **then** $\text{resize}()$
            return $true$
    return $false$

- This takes $O(n_{\text{hash}(x)})$ time, where n_i denotes the length of the list stored at $t[hash(key)]$.

ChainedHashTable: $resize()$ (Discussion Activity Write the Steps)

- $resize()$: Resize the table t such that $n \leq length(t) < 3n$

resize()

ChainedHashTable: $resize()$

resize()
    if $n = length(t)$ **then** $d + 1$
    else $d - 1$
    $a \leftarrow \text{alloc\_table}(2^d)$
    for j **in** $0, 1, .., length(t) - 1$
        for i **in** $0, 1, .., t[j].size() - 1$
            $a[hash(t[j].get(i).key)].add(t[j].get(i))$
    $t = a$

ChainedHashTable: $find(key)$

- $find(key)$: Returns the node if key exists and None otherwise

```
find(key)
    Check Preconditions
    Check if key exist in t[hash(key)] and return the node.
    return nil
```

ChainedHashTable: $find(key)$

$\text{find}(key)$
for i **in** $0, 1, .., t[hash(key)].size() - 1$
if $t[hash(key)].get(i).key = key$ **then**
return $t[hash(key)].get(i)$
return nil

- This takes time proportional to the length of the list $t[\text{hash}(x)]$.

ChainedHashTable: Performance

- The performance of a hash table depends critically on the choice of the hash function.
- A good hash function will spread the elements evenly among the $\mathrm{length}(t)$ lists, so that the expected size of the list $t[\mathrm{hash}(x)]$ is $O(n/\mathrm{length}(t)) = O(1)$.
- On the other hand, a bad hash function will hash all values to the same table location, in which case the size of the list $t[\mathrm{hash}(key)]$ will be n.

Multiplicative Hashing

- It uses the div operator, which calculates the integral part of a quotient, while discarding the remainder.
- Formally, for any integers $a \geq 0$ and $b \geq 1$, $a \; div \; b = \lfloor a/b \rfloor$.
- We use a hash table of size 2^d for some integer d (called the *dimension*).
- The formula for hashing an integer $key \in \{0, \ldots, 2^w - 1\}$ is

$$hash(key) = ((z \cdot hashCode(key) \bmod 2^w) \; div \; 2^{w-d}$$

- Here, z is a randomly chosen *odd* integer in $\{1, \ldots, 2^w - 1\}$.
- By default operations on integers are already done modulo 2^w where w is the number of bits in an integer

Multiplicative Hashing

- Integer division by 2^{w-d} is equivalent to dropping the rightmost $w - d$ bits in a binary representation
- It is implemented by shifting the bits right by $w - d$ using the $>>$ operator

$$
\begin{array}{l}
\mathrm{hash}(key) \\
\quad \textbf{return } ((z \cdot hashCode(key)) \bmod 2^{w}) >> (w - d)
\end{array}
$$

Multiplicative Hashing

- Python

```
def _hash(self, x):
     return ((self.z * hash(x)) % (1<<w)) >> (w-self.d)
```

- Java

```
int hash(Object x) {
   return (z * x.hashCode()) >>> (w-d);
 }
```

- C++

```
int hash(T x) {
   return ((unsigned)(z * std::hash<std::string>{}(x)))
                 >> (w-d);
}
```

Universal Hashing

Lemma

Let x and y be any two values in $\{0, \ldots, 2^w - 1\}$ with $x \neq y$. Then $\Pr\{\mathrm{hash}(x) = \mathrm{hash}(y)\} \leq 2/2^d$.

Proof

Observe that the highest-order d bits of $zx \bmod 2^w anddbitsofz$ are the same. Therefore, the highest-order d bits in the binary representation of $z(x - y) \bmod 2^w$ are either all 0's or all 1's. That is,

$$z(x - y) \bmod 2^w = (\underbrace{0, \ldots, 0}_{d}, \underbrace{\star, \ldots, \star}_{w-d})_2 \tag{1}$$

when $zx \bmod 2^w > zy \bmod 2^w$ or

$$z(x - y) \bmod 2^w = (\underbrace{1, \ldots, 1}_{d}, \underbrace{\star, \ldots, \star}_{w-d})_2 \quad . \tag{2}$$

when $zx \bmod 2^w < zy \bmod 2^w$.

Therefore, we only have to bound the probability that $z(x-y) \bmod 2^w$ looks like (2) or (3).
Let q be the unique odd integer such that $(x-y) \bmod 2^w = q2^r$ for some integer $r \geq 0$. The binary representation of $zq \bmod 2^w$ has $w-1$ random bits, followed by a 1:

$$zq \bmod 2^w = (\underbrace{b_{w-1}, \ldots, b_1}_{w-1}, 1)_2$$

Therefore, the binary representation of $z(x-y) \bmod 2^w = zq2^r \bmod 2^w$ has $w-r-1$ random bits, followed by a 1, followed by r 0's:

$$z(x-y) \bmod 2^w = zq2^r \bmod 2^w = (\underbrace{b_{w-r-1}, \ldots, b_1}_{w-r-1}, 1, \underbrace{0, 0, \ldots, 0}_{r})_2$$

Universal Hashing (Proof 3/3)

- If $r > w - d$, then the d higher order bits of $z(x-y) \bmod 2^w$ contain both 0's and 1's, so the probability that $z(x-y) \bmod 2^w$ looks like (2) or (3) is 0.
- If $r = w - d$, then the probability of looking like (0) is 0, but the probability of looking like (1) is $1/2^{d-1} = 2/2^d$ (since we must have $b_1, \ldots, b_{d-1} = 1, \ldots, 1$).
- If $r < w - d$, then we must have $b_{w-r-1}, \ldots, b_{w-r-d} = 0, \ldots, 0$ or $b_{w-r-1}, \ldots, b_{w-r-d} = 1, \ldots, 1$. The probability of each of these cases is $1/2^d$ and they are mutually exclusive, so the probability of either of these cases is $2/2^d$.

Expected Value and Probability

- For a discrete random variable X taking on values in some countable universe U, the expected value of X, denoted by $E[X]$, is given by

$$\mathrm{E}[X] = \sum_{x \in U} x \cdot \Pr\{X = x\}$$

  Where $\Pr\{\mathcal{E}\}$ denotes the probability that the event $\mathcal{E}$ occurs.

- Linearity of expectation. For any two random variables X and Y,

$$\mathrm{E}[X + Y] = \mathrm{E}[X] + \mathrm{E}[Y]$$

  More generally, for any random variables $X_1, \ldots, X_k$,

$$\mathrm{E}\left[\sum_{i=1}^{k} X_i\right] = \sum_{i=1}^{k} \mathrm{E}[X_i]$$

Hash Analysis

Lemma

For any unique key x, the expected length of the list $t[\mathrm{hash}(x)]$ is at most 2.

Proof Let S be the set of elements stored in the hash table that are not equal to x. For an element $y \in S$, define the indicator variable

$$I_y = \begin{cases} 1 & \text{if } \mathrm{hash}(x) = \mathrm{hash}(y) \\ 0 & \text{otherwise} \end{cases}$$

Proof

By the previous lemma, $E[I_y] \leq 2/2^d = 2/length(t)$.
The expected length of the list $t[hash(x)]$ is given by

$$\begin{aligned}
\mathrm{E}\left[t[hash(x)].size()\right] &= \mathrm{E}\left[\sum_{y \in S} I_y\right] \\
&\leq \sum_{y \in S} 2/length(t) \\
&\leq \sum_{y \in S} 2/n \\
&= 2
\end{aligned}$$

Summary of ChainedHashTable

Theorem

The expected running time of $add(x)$, $remove(x)$ *and* $find(x)$ *of ChainHashTable is* $O(1)$ *.*