Hashing Part 1

Ira A. Fulton Schools of Engineering
Arizona State University

Why Hashing?

- Many applications require a dynamic set that supports the operations INSERT, SEARCH and DELETE.
- O(1) time insertion and searching in best case (many cases in practice)
- O(n) time insertion and searching in worst case.

Direct-Address Tables

- **Universe U={0, 1, …, m-1} of keys, where m is not too large. No two elements have the same key.**
- **Use an array T[0..m-1] as the direct-address table.**
  - Direct-Address-Search(T, k)

    return T[k]
  - Direct-Address-Insert(T, x)

    T[key(x)]:=x
  - Direct-Address-Delete(T, x)

    T[key(x)]:=nil

Properties of Direct-Address Tables

- O(1) time searching
- O(1) time insertion
- O(1) time deletion
- Table size cannot be very large. Why?
- Many slots are wasted. Why?

Hash Tables

- **With direct addressing, an element with key *k* is stored in slot *k*.**

- **With hashing, this element is stored in slot h(k), where h is a *hash function*.**

  - h: $U \rightarrow \{0, 1, \ldots, m-1\}$, where $m \ll |U|$.

Collision and Collision Resolution

- *Collision*: two keys being hashed to the same slot!
- There are methods to *resolve* collision:
  - Chaining
  - Open addressing

Hashing with Chaining

- **Maintain a list at each hash slot**
- **Chained-Hash-Insert(T, x)**
  - Insert x at the head of list T[h(key(x))]
- **Chained-Hash-Search(T, k)**
  - Search for an element with key k in list T[h(k)]
- **Chained-Hash-Delete(T, x)**
  - Delete x from the list T[h(key(x))]
- **See page 278 in textbook…**

Hashing with Chaining

- A hash table of m slots with n elements
- load factor is $\alpha=n/m$.
- In the worst-case, insertion takes $O(1)$ time.
- In the worst-case, searching takes $O(n)$ time.
- In the worst-case, deletion takes $O(n)$ time.

- On average, searching and deletion takes $\Theta(1+\alpha)$ time.

Hashing Part 2

Ira A. Fulton Schools of Engineering
Arizona State University

Hashing with Open addressing

- In open addressing, all elements are stored in the hash table itself.
- In other words, there is no chain. A slot either contains an element or nil.
- There is a systematic way of searching the slots (in both searching and insertion)
- Find the next open address.

Probing

- $h: U \times \{0, 1, \ldots, m-1\} \rightarrow \{0, 1, \ldots, m-1\}$
- probe sequence is $\langle h(k, 0), h(k, 1), \ldots h(k, m-1) \rangle$. Should be a permutation of $\langle 0, 1, \ldots, m-1 \rangle$.

- Example: $h(k, j) = (h'(k) + j) \bmod m$, $j=0, 1, \ldots, m-1$.
- Here $h'()$ is a hash function itself.

Hash-Insert(T, k): p. 294

```
i := 0
repeat
        q := h(k, i)
        if T[q] == NIL then
                T[q] := k
                return q
        else    i := i+1
until i == m
error "hash table overflow"
```

Hash-Insert(T, k): p. 294

```
i := 0
repeat
        q := h(k, i)
        if T[q] == k then
                return q
        else    i := i+1
until T[q]==NIL or i==m
return NIL
```

Example: Insertion and Searching

- Assume: $h'(k) = k \bmod 13$
  $h(k, i) = (h'(k) + i) \bmod 13$

j	T[j]
0	NIL
1	NIL
2	NIL
3	NIL
4	NIL
5	NIL
6	NIL
7	NIL
8	NIL
9	NIL
10	NIL
11	NIL
12	NIL

Example: Insertion and Searching

Hash-Search(T, 50)

i ← 0

q ←h(50, 0) = 11

T[q] == NIL

return NIL

j	T[j]
0	NIL
1	NIL
2	NIL
3	NIL
4	NIL
5	NIL
6	NIL
7	NIL
8	NIL
9	NIL
10	NIL
11	NIL
12	NIL

Example: Insertion and Searching

Hash-Insert(T, 50)

i ← 0

q ←h(50, 0) = 11

T[q] == NIL

T[q] ← 50

return 11

j	T[j]
0	NIL
1	NIL
2	NIL
3	NIL
4	NIL
5	NIL
6	NIL
7	NIL
8	NIL
9	NIL
10	NIL
11	NIL
12	NIL

Example: Insertion and Searching

Hash-Insert(T, 50)

i ← 0

q ←h(50, 0) = 11

T[q] == NIL

T[q] ← 50

return 11

j	T[j]
0	NIL
1	NIL
2	NIL
3	NIL
4	NIL
5	NIL
6	NIL
7	NIL
8	NIL
9	NIL
10	NIL
11	50
12	NIL

Example: Insertion and Searching

Hash-Insert(T, 11)

i ← 0

q ←h(11, 0) = 11

T[q] == 50

i ← 1

q ← h(11, 1) = 12

T[q] == NIL

T[q] ← 11

return 12

j	T[j]
0	NIL
1	NIL
2	NIL
3	NIL
4	NIL
5	NIL
6	NIL
7	NIL
8	NIL
9	NIL
10	NIL
11	50
12	NIL

Example: Insertion and Searching

Hash-Insert(T, 11)

i ← 0

q ←h(11, 0) = 11

T[q] == 50

i ← 1

q ← h(11, 1) = 12

T[q] == NIL

T[q] ← 11

return 12

j	T[j]
0	NIL
1	NIL
2	NIL
3	NIL
4	NIL
5	NIL
6	NIL
7	NIL
8	NIL
9	NIL
10	NIL
11	50
12	11

Example: Insertion and Searching

Hash-Search(T, 11)

i ← 0

q ←h(11, 0) = 11

T[q] == 50 != 11

i ← 1

q ← h(11, 1) = 12

T[q] == 11

return 12

j	T[j]
0	NIL
1	NIL
2	NIL
3	NIL
4	NIL
5	NIL
6	NIL
7	NIL
8	NIL
9	NIL
10	NIL
11	50
12	11

Example: Insertion and Searching

Hash-Search(T, 24)

i ← 0

q ←h(24, 0) = 11

T[q] == 50 != 24

i ← 1

q ← h(24, 1) = 12

T[q] == 11 != 24

i ← 2

q ← h(24, 2) = 0

T[q] == NIL

return NIL

j	T[j]
0	NIL
1	NIL
2	NIL
3	NIL
4	NIL
5	NIL
6	NIL
7	NIL
8	NIL
9	NIL
10	NIL
11	50
12	11

Hashing Part 3

Ira A. Fulton Schools of Engineering
Arizona State University

Deletion

- Can we just delete a key from the hash table?
- Can we change the table content to NIL (call it Hash-Delete-1)?
- What is the effect on Hash-Search?
- How about Hash-Insert?

Hash-Delete-1(T, k): A bad approach

```
if Hash-Search(T, k) != NIL then
        q := Hash-Search(T, k)
        T[q] := NIL
return
```

Example: First attempt of deletion

Hash-Delete-1(T, 50)

q ←Hash-Search(T, 50) = 11

T[11] == NIL

j	T[j]
0	NIL
1	NIL
2	NIL
3	NIL
4	NIL
5	NIL
6	NIL
7	NIL
8	NIL
9	NIL
10	NIL
11	50
12	11

Example: First attempt of deletion

Hash-Delete-1(T, 50)

q ←Hash-Search(T, 50) = 11

T[11] == NIL

j	T[j]
0	NIL
1	NIL
2	NIL
3	NIL
4	NIL
5	NIL
6	NIL
7	NIL
8	NIL
9	NIL
10	NIL
11	NIL
12	11

Example: First attempt of deletion

Hash-Search(T, 11)

i ← 0

q ←h(11, 0) = 11

T[q] == NIL

return NIL

But 11 is in the table, at location 12!!!

j	T[j]
0	NIL
1	NIL
2	NIL
3	NIL
4	NIL
5	NIL
6	NIL
7	NIL
8	NIL
9	NIL
10	NIL
11	NIL
12	11

Deletion

- How do we solve this problem?
- Use a special key—DELETED.
- Hash-Insert should be modified accordingly, treating DELETED the same as NIL.

Hash-Delete(T, k): Using DELETED

```
if Hash-Search(T, k) != NIL then
        q := Hash-Search(T, k)
        T[q] := DELETED
return
```

Example: Deletion

Hash-Delete(T, 50)

q ←Hash-Search(T, 50) = 11

T[11] == DELETED

j	T[j]
0	NIL
1	NIL
2	NIL
3	NIL
4	NIL
5	NIL
6	NIL
7	NIL
8	NIL
9	NIL
10	NIL
11	50
12	11

Example: Deletion

Hash-Delete(T, 50)

q ←Hash-Search(T, 50) = 11

T[11] == DELETED

j	T[j]
0	NIL
1	NIL
2	NIL
3	NIL
4	NIL
5	NIL
6	NIL
7	NIL
8	NIL
9	NIL
10	NIL
11	DELETED
12	11

Example: Deletion

Hash-Search(T, 11)

i ← 0

q ←h(11, 0) = 11

T[q] == DELETED != NIL

i ← 1

q ←h(11, 1) = 12

T[q] == 11

return 12

It works!!!

j	T[j]
0	NIL
1	NIL
2	NIL
3	NIL
4	NIL
5	NIL
6	NIL
7	NIL
8	NIL
9	NIL
10	NIL
11	DELETED
12	11

Hash-Insert(T, k): dealing with DELETED

```
i := 0
repeat
        q := h(k, i)
        if T[q] == NIL or T[j] == DELETED then
                T[q] := k
                return q
        else    i := i+1
until i == m
error "hash table overflow"
```

Example: Insertion after deletion

Hash-Insert(T, 24)

i ← 0

q ←h(24, 0) = 11

T[q] == DELETED

T[11] ← 24

return 11

j	T[j]
0	NIL
1	NIL
2	NIL
3	NIL
4	NIL
5	NIL
6	NIL
7	NIL
8	NIL
9	NIL
10	NIL
11	DELETED
12	11

Example: Insertion after deletion

Hash-Insert(T, 24)

i ← 0

q ←h(24, 0) = 11

T[q] == DELETED

T[11] ← 24

return 11

It works!!!

j	T[j]
0	NIL
1	NIL
2	NIL
3	NIL
4	NIL
5	NIL
6	NIL
7	NIL
8	NIL
9	NIL
10	NIL
11	24
12	11

Worst-case time complexities

- Insertion takes O(m) time in the worst-case
- Searching takes O(m) time in the worst-case
- Deletion takes O(m) time in the worst-case

- Please note that the delete algorithm is DIFFERENT from the Linear-Probing-Hash-Delete on p. 303 of the textbook.

Hashing with open addressing

- A hash table of m slots with n elements
- load factor is $\alpha = n/m < 1$.
- The expected number of probs for an <u>unsuccessful search</u> is at most **$1/(1-\alpha)$**
- The expected number of probs for a <u>successful search</u> is at most **$-1/\alpha \ln (1-\alpha)$**

Common methods for open addressing

Linear probing

- h(k, i) = (h’(k) + i) mod m, i=0, 1, 2, m-1

Quadratic probing

- h(k, i) = (h’(k) + c1 i + c2 i2) mod m, i=0, 1, 2, m-1

Double Hashing

- h(k, i) = (h1 (k) + i h2 (k)) mod m, i=0, 1, 2, m-1

Hashing Part 4

Hash functions

- **What makes a good hash function?**
- **Interpreting keys as natural numbers**
  - pt treated as (112, 116)…ASCII values…
  - key(pt)=112*128 + 116 = 14452
- **The division method**
- **The multiplication method**

Hash functions: (1) The division method

- **The division method**
  - $h(k) = k \bmod m$
- **Requirements:**
  - m not a power of 2
  - choose m as a prime number not close to a power of 2

Hash functions: (2) The multiplication method

- The multiplication method
- Let A be a constant in (0, 1). Multiply the key k by A and extract the fractional part of kA. Then we multiply this value by m and take the floor of the result:

$$h(k) = \lfloor m(kA \bmod 1) \rfloor$$

- Use $A=(\sqrt{5}-1)/2=0.6180339887$ and m a power of 2.
- The value of m is not critical here.
- See pages 284-285 in textbook for an example.
- m=16384, k=123456, h(k)=67.

Hash functions: (2) The multiplication method

$$h(k) = \lfloor m(kA \bmod 1) \rfloor$$

- A=(√5-1)/2=0.6180339887, m=16384, k=123456.
- h(k) = ?
- k A = 76300.0041089472
- (k A) mod 1 = 0.0041089472
- m ((k A) mod 1) = 67.3209909248
- h(k) = 67

Summary

- Hashing is simple and useful
- Hash tables
- Collision resolutions
- Hash functions

Ira A. Fulton Schools of
Engineering
Arizona State University