HASH TABLES (PART 2)

EECS 2011

1

Collision Handling

- Separate chaining
- Probing (open addressing): the colliding item is placed in a different cell of the table
  - Linear probing
  - Quadratic probing
  - Double hashing

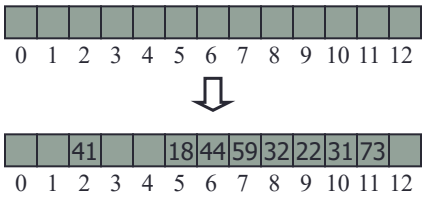
2

3

Linear Probing

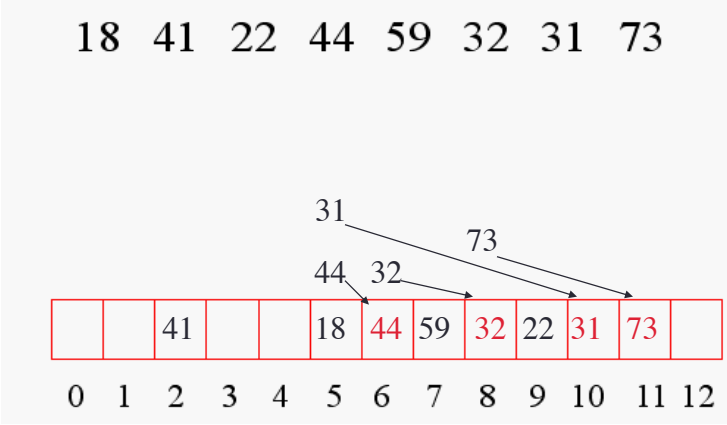
- Linear probing handles collisions by placing the colliding item in the next (circularly) available table cell
- Each table cell inspected is referred to as a "probe"
- Colliding items lump together; future collisions will cause a longer sequence of probes

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order
  - Remove 44, 32, 73, 31



4

Linear Probing Example



5

Search with Linear Probing

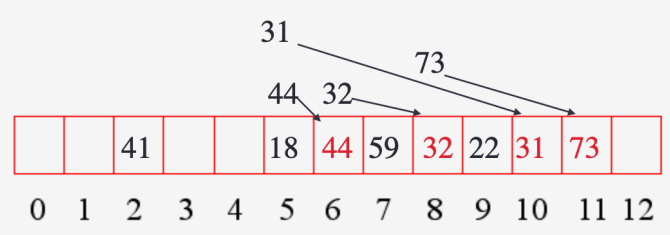
- Consider a hash table A that uses linear probing
- $get(k)$
  - We start at cell $h(k)$
  - We probe consecutive locations until one of the following occurs
    - An item with key k is found, or
    - An empty cell (null) is found, or
    - N cells have been unsuccessfully probed

```
Algorithm get(k)
    i ← h(k)
    p ← 0
    repeat
        c ← A[i]
        if c == ∅
            return NULL
        else if c.getKey () = k
            return c.getValue()
        else
            i ← (i + 1) mod N
            p ← p + 1
    until p = N
    return NULL
```

5

6

Search Example



- To search for key 31 (31 % 13 = 5), start at index 5 and keep going until seeing 31 at index 10.
- To search for key 5 (5 % 13 = 5), start at index 5 and keep going until seeing an empty bucket, at index 12 → meaning key is not found.

6

7

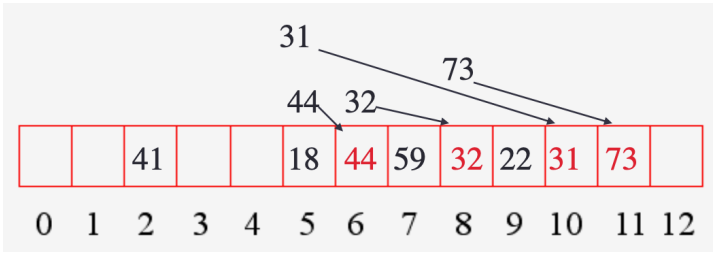
Removal with Probing

- Let's consider *remove(k)*
  - Call *get*(*k*) to get the element.
  - Should we set the now empty cell to NULL?
    - No. It would mess up the search procedure.
    - See example on the next slide.

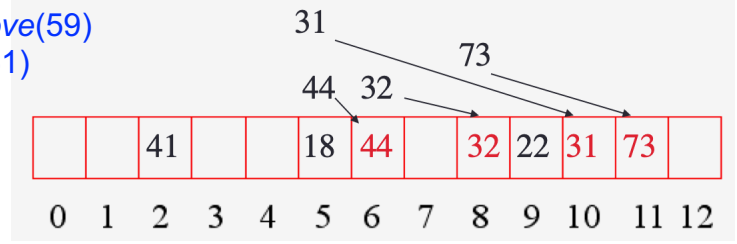
7

8

Example with *remove*(*k*)



remove(59)
get(31)



8

Removal with Probing (2)

- A cell has three states:
  - null: brand new, never used. *get*(*x*) stops when a null cell is reached.
  - in use (not null): currently used.
  - DEFUNCT: previously used, now available but unused. *get*(*x*) continues the search when encountering a DEFUNCT cell.
- Method **`put(k)`** should remember a defunct location encountered during the search for k.
  - If no existing entry is found beyond the defunct location, then put **`k`** into the defunct location.

9

Remove and Insert with Probing

- To handle insertions and deletions, we introduce a special object, called ***DEFUNCT***, which replaces deleted elements
- remove(k)
  - We search for an entry with key k
  - If such an entry (k, o) is found, we replace it with the special item ***DEFUNCT*** and we return element o
  - Else, we return ***null***
- put(k, o)
  - We throw an exception if the table is full
  - We start at cell $h(k)$
  - We probe consecutive cells until one of the following occurs
    - A cell i is found that is either empty or stores ***DEFUNCT***, or
    - N cells have been unsuccessfully probed
  - We store (k, o) in cell i

10

Collision Handling

- Separate chaining
- Probing (open addressing)
  - Linear probing
  - Quadratic probing
  - Double hashing

11

Quadratic Probing

- Linear probing:
  Insert item (k, e)
  i = h(k)
  A[i] is occupied
  Try A[(i+1) mod N]: used
  Try A[(i+2) mod N]
  and so on until
  an empty cell is found

- Quadratic probing
  A[i] is occupied
  Try A[(i+1) mod N]: used
  Try A[(i+2^2) mod N]: used
  Try A[(i+3^2) mod N]
  and so on

- May not be able to find an empty cell if *N* is not prime, or the hash table is at least half full.
- Secondary clustering

12

13

Double Hashing

- Double hashing uses a secondary hash function $d(k)$ and handles collisions by placing an item in the first available cell of the series
  $(i + j \times d(k)) \bmod N$
  for $j = 0, 1, \ldots, N - 1$
- The secondary hash function $d(k)$ cannot have zero values
- The table size N must be a prime to allow probing of all the cells

Insert item (k, e)
i = h(k)
A[i] is occupied
Try A[(i+d(k))mod N]: used
Try A[(i+2d(k))mod N]: used
Try A[(i+3d(k))mod N]
and so on until
an empty cell is found

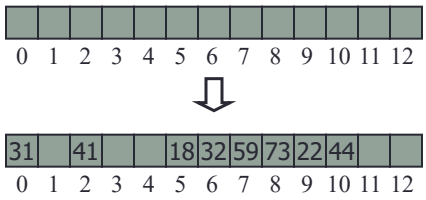
13

14

Example of Double Hashing

- Consider a hash table storing integer keys that handles collision with double hashing
  - $N = 13$
  - $h(k) = k \bmod 13$
  - $d(k) = 7 - k \bmod 7$
- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

k	$h(k)$	$d(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		



14

15

Double Hashing (2)

- $d(k)$ should be chosen to minimize clustering
- Common choice of compression map for the secondary hash function:

  $d(k) = q - k \bmod q$

  where
  - $q < N$
  - q is a prime
- The possible values for $d(k)$ are

  $1, 2, \ldots, q$

- Note: linear probing has $d(k) = 1$.

15

16

Open Addressing: Load Factor

- Define the load factor λ = n/N
  - n = number of elements in the hash table
  - N = hash table size (prime number)
- To obtain best performance, ensure that $\lambda < 0.5$ for linear probing, and only a bit higher for other open addressing schemes.
- As we add more elements to the hash table, λ goes up $\Rightarrow$ rehashing (allocate a bigger table, define a new compression function, and map the elements to the new array).

16

17

Comparing Collision Handling Schemes

- Separate chaining:
  - simple implementation
  - faster than open addressing in general
  - using more memory
- Open addressing:
  - using less memory
  - slower than chaining in general
  - more complex removals
- Linear probing: items are clustered into contiguous runs called *primary clustering*.
- Quadratic probing: items are clustered into contiguous runs called *secondary clustering*.
- Double hashing: distributes keys more uniformly than linear and quadratic probing.

17

18

Performance of Hashing

- The expected running time of all the dictionary ADT operations in a hash table is $O(1)$.
- The load factor $\lambda = n/N$ affects the performance of a hash table.
- In the worst case, searches, insertions and removals on a hash table take $O(n)$ time.
- The worst case occurs when all the keys inserted into the map collide.
- In practice, hashing is very fast provided that $\lambda < 0.9$ for separate chaining and $\lambda < 0.5$ for open addressing.

18

19

Summary

- Purpose of hash tables: to obtain $O(1)$ *expected* query time.
- If the keys are not integers (e.g., strings), convert them to integer keys.
- Map integer keys to the hash table entries using a *compression map* function.
- If collision occurs, use one of the collision handling schemes, taking into account available memory space.
  - Separate chaining
  - Open addressing
- If the load factor $\lambda = n/N$ approaches the specified threshold, *rehash*.

19

20

Next lecture …

- Graphs: for self-study, not on final exam.

20