CSE 440: Advanced Algorithms

Lecture 7: Hash Tables

CLRS Ch. 11 (11.1-11.4)

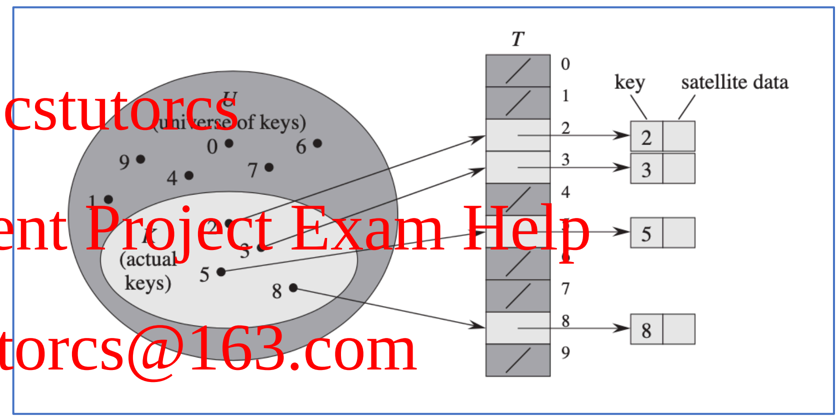
Direct Address Tables

- Works well when universe of keys is small

DIRECT-ADDRESS-SEARCH(T, k)
1 **return** $T[k]$

DIRECT-ADDRESS-INSERT(T, x)
1 $T[x.key] = x$

DIRECT-ADDRESS-DELETE(T, x)
1 $T[x.key] =$ NIL



- Compare with
  - Arrays?
  - Linked Lists?

Hash Tables

n |K|

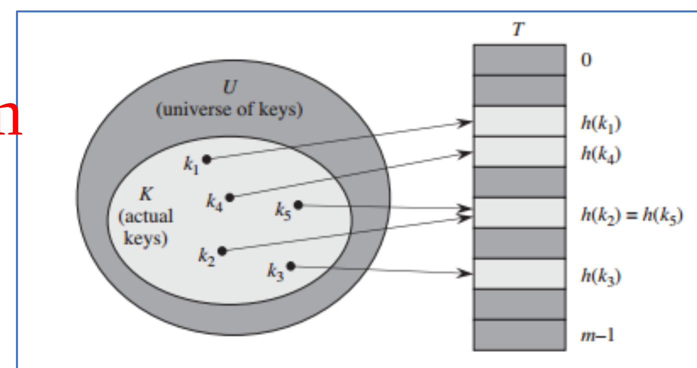
m |Table|

- Observation:
  - if |U| is too la[illegible] using "Direct Access table" is prohibitive expensive (memory)
  - If |K| << |U| then too much space is wasted

- Dual goal:

  - space requirement: $\Theta(|K|)$ space used
  - Expected time for dictionary operations O(1)

- Idea
  - h: U → {0,…,m-1} //m = $\Theta(|K|)$



Hash Tables

- Differences in c[illegible] with direct address tables
  - Hash functions
    - Key k is stored in slot h(k). -- we name h(k) the hash value of k
    - Has to be deterministic!!
    - Possible collision: efficient hash function should reduce its probability
  - Collision resolution
    - Closed addressing (chaining)
    - Open addressing (probing)

Hash Function

- Goal:
  - Simple uniform h… …n key is equally likely to hash to any of the m slots, independ… …ere any other key has hashed to.

- Challenges
  - Requires understanding the distribution from which keys are drawn
  - Keys might not be drawn independently

- Division method
  - Interpret keys as natural numbers
  - H(k) = k mod m
  - Simple solution, but mostly effective if m is prime no too close to an exact power of 2

- More complex solutions (used for security more than balancing)
  - Universal hashing
  - SHA-256

Closed Addressing (Chaining)

CHAINED-HASH-INSERT(T, x)

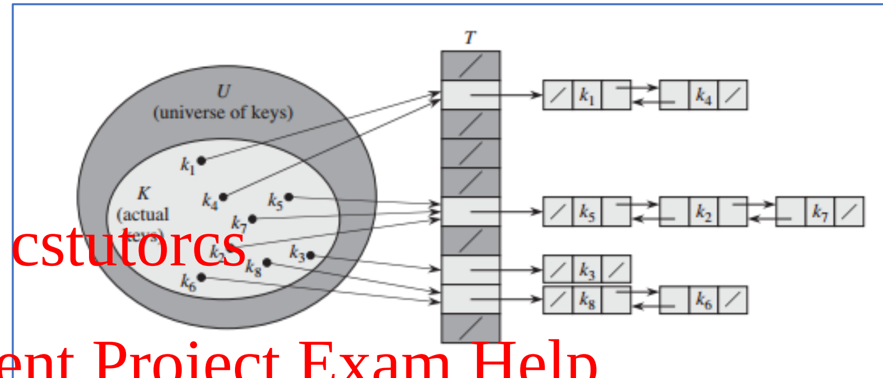
insert x at the head of list $T[h(x.key)]$

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 $T[h(x.key)]$



- Insert: O(1) if no check for duplicates
- Delete: O(1) if doubly linked list and input is element not key
- Main issue is in search

Analysis of Chaining

- Let n = |K| and of array
- Load factor: $\alpha(T) = \alpha = n/m$ is load factor
  - Note: because of chaining, $\alpha(T)$ may be greater than 1
    - In fact, $\alpha(T)$ is a good representation for the number of elements stored in a chain

- Analysis of search
  - Worst case? O(n) linked list
  - Best case? O(1)
  - Average case? $O(\alpha)$

Analysis of Chaining

- Assume uniform hashing: any given element is equally likely to hash into any of the m slots
  - n_j is the length of the list T[j]
  - $E[n_j] = \alpha$

  - If key does not exist

    - Average case time $= \Theta(1+\alpha)$

  - What if key exists?

    - We will show in the next slides that it is also $\Theta(1+\alpha)$

  - If n = O(m)

    - $\Theta(1+\alpha) = O(1)$

Analysis of Chaining

- Number of eleme[illegible]ined
  - Number of elem[illegible]r before x in x's list
  - Which is: numbe[illegible]nts inserted after x

- Definitions
  - X_i = "i-th element added to the table"
  - $k_i = x_i$.key
  - $X_{ij} = I\{h(k_i) = h(k_j)\}$

- Thus
  - $E[X_{ij}] = Pr(h(k_i) = h(k_j)) = 1/m$

  - Expected number of elements examined

$$\mathrm{E}\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n} X_{ij}\right)\right]$$

Prob that k_i is the key we search for

first check

Expected # of elements inserted after k_i in k_i's slot

Analysis of Chaining

$$
\begin{aligned}
\mathrm{E}\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n} X_{ij}\right)\right]
&= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n} \mathrm{E}\left[X_{ij}\right]\right) && \text{(by linearity of expectation)}\\
&= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n} \frac{1}{m}\right)\\
&= 1+\frac{1}{nm}\sum_{i=1}^{n}(n-i)\\
&= 1+\frac{1}{nm}\left(\sum_{i=1}^{n} n-\sum_{i=1}^{n} i\right)\\
&= 1+\frac{1}{nm}\left(n^2-\frac{n(n+1)}{2}\right) && \text{(by equation (A.1))}\\
&= 1+\frac{n-1}{2m}\\
&= \boxed{1+\frac{\alpha}{2}-\frac{\alpha}{2n}}\,.
\end{aligned}
$$

Which is Θ(1+α)

Open Addressing

- Probing within th[illegible]tself
  - No pointers at al[illegible]
  - α is never greater than 1

- Hash function includes probe number
  - h: U $\times$ {0,1,…,m-1} $\rightarrow$ {0,1,…,m-1}
  - Probe sequence:
    - <h(k,0), h(k,1),…, h(k,m-1)>
    - Has to be a permutation of <0, 1, m-1>
    - Better if uniform hashing: equally likely to be any of the m! permutations

```
HASH-INSERT(T, k)
i = 0
repeat
    j = h(k, i)
    if T[j] == NIL
        T[j] = k
        return j
    else i = i + 1
until i == m
error "hash table overflow"
```

```
HASH-SEARCH(T, k)
i = 0
repeat
    j = h(k, i)
    if T[j] == k
        return j
    i = i + 1
until T[j] == NIL or i == m
return NIL
```

What About Deletion?

- Cannot simply re y with Nil
  - Probing will not

- **Delete(T,k)**

  i = hash-search(T,k)

  T[i] = “Delete”

- We modify Hash-insert accordingly:

T[j] == nil or T[j] == “Delete”

HASH-INSERT(T, k)

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

Probe Sequencing

- Linear probing
  - h(k,i) = (h'(k) + i
  - Main drawback: clustering
    - long runs of occupied cells, which increases average search time

m Possible Probing Sequences

- Quadratic probing
  - h(k,i) = (h'(k) + (c1*i)+ (c2 *c2*i)) mod m
  - Main drawback: secondary clustering
    - Initial probe determines the entire sequence.

- Double hashing
  - h(k,i) = (h1(k) + i h2(k)
  - Better performance: more possible probe sequences, less clustering
  - Still not as ideal as uniform hashing

m^2

Analysis of Open Addressing

- Assume uniform hashing
  - Not true for other hashing techniques, but simplifies analysis

- Unsuccessful search: E[number of probes] ≤ $1/(1-\alpha)$
  - Informal justification (formal proof in textbook - skip)
    - $1/(1-\alpha) = 1 + \alpha + \alpha^2 + \alpha^3 + \ldots$

First probe always happen

Second probe happens with probability α

Third probe happens with probability α^2

- Successful search is even better
  - $E[X] \le (1/\alpha)\ln(1/(1-\alpha))$ (proof skipped)

Back to Dynamic Tables

Discussion

Discussion

- What about hash tables with closed addressing (e.g., chaining)?
  - Same idea, with slight change of definition of α
    - Expand/shrink table based on threshold of bucket sizes
    - Examples:
      - Maintain per-bucket/per-table metadata with each insert/delete
      - Sample the sizes of randomly selected buckets

  - Possible improvement
    - Resize each bucket independently
    - Example:
      Kevin Williams, Joe Foster, Athicha Srivirote, Ahmed Hassan, Joseph Tassarotti, Lewis Tseng, Roberto Palmieri: **"On Building Modular and Elastic Data Structures with Bulk Operations",** ICDCN'21

- What about hash tables with open addressing?
  - No difference for expansion
  - Contraction depends on the way we handle deletes

Discussion

2 16

Freeze

- What about conc namic tables
  - Although expans amortized, it blocks any concurrent operation until i .
  - Solutions?

    - Amortize the actual resizing not its cost
      - Allocate new table eagerly, move data to new table lazily

      - Key idea: resizing is a structural change not a semantic change.
      - Depends on implementation details

Semantic

Structural

- Example: Yujie Liu, Kunlong Zhang, and Michael Spear **"Dynamic-Sized Nonblocking Hash Tables"**, PODC'14

  - Freeze objects in old hash table
  - A new operation (insert/delete/lookup) on a frozen object x first copies x to the new hash table

  - Another resize first completes the previous resize (hopefully by then most elements will be already moved).

Thanks!