CS F364: Design & Analysis of Algorithms

Lecture-kt08: Hashing

Dr. Kamlesh Tiwari,
Assistant Professor,
Department of Computer Science and Information Systems,
BITS Pilani, Rajasthan-333031 INDIA

Feb 07, 2017 (Campus @ BITS-Pilani Jan-May 2017)

Recap: Order Statics

- Order statistic can be solved in $O(n\log(n))$ time, since we can sort

Recap: Order Statics

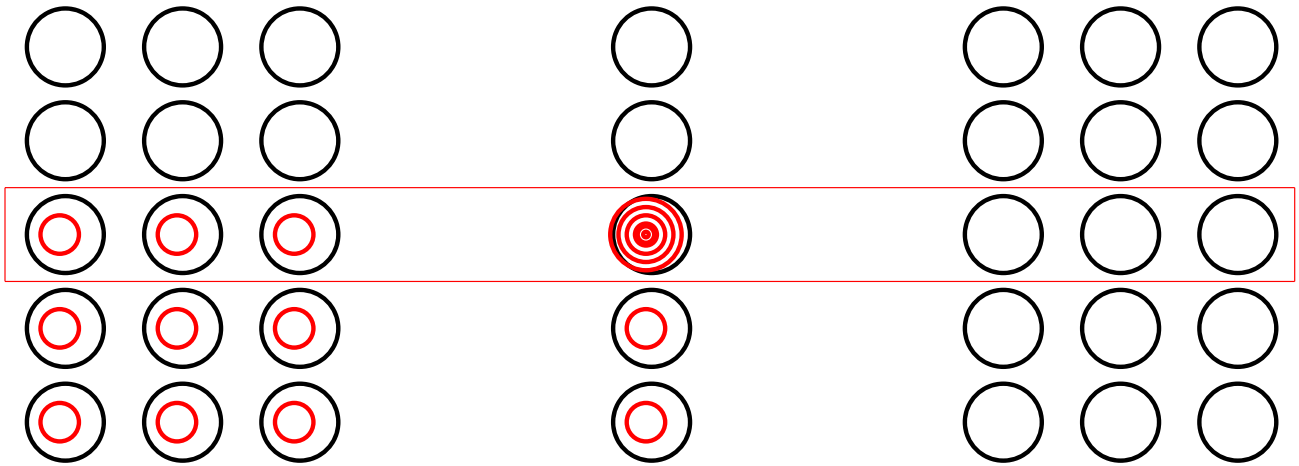
- Order statistic can be solved in $O(n\log(n))$ time, since we can sort
- As 1^{st}, 2^{nd} and n^{th} can be done in $O(n)$ time want same for others

Recap: Order Statics

- Order statistic can be solved in $O(n\log(n))$ time, since we can sort
- As 1^{st}, 2^{nd} and n^{th} can be done in $O(n)$ time want same for others
- **Randomized-Select** takes $O(n)$ expected time. But, in worst case it can take $O(n^2)$ time ☹

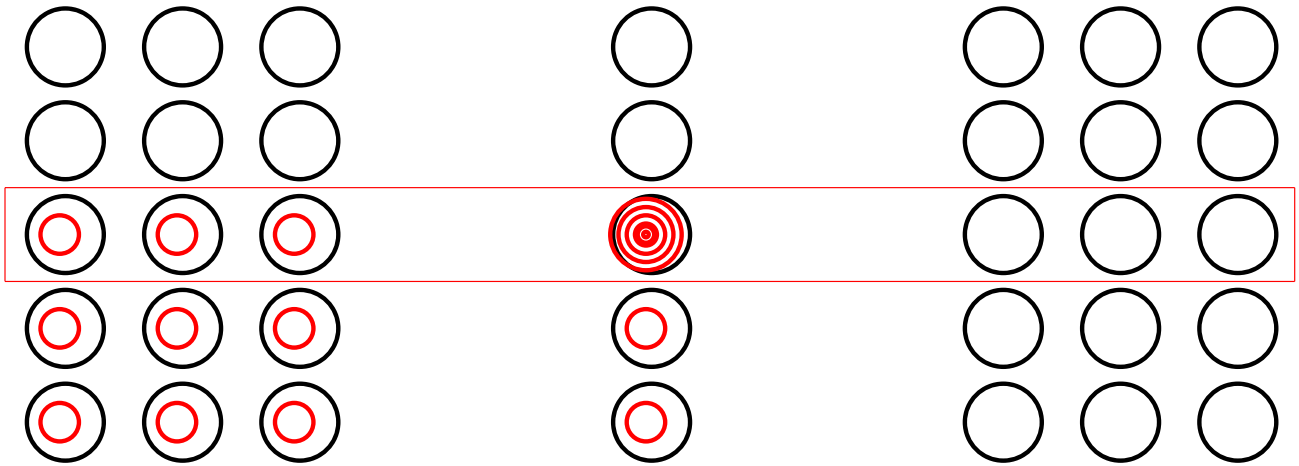
Recap: Order Statics

- Order statistic can be solved in $O(n\log(n))$ time, since we can sort
- As 1^{st}, 2^{nd} and n^{th} can be done in $O(n)$ time want same for others
- **Randomized-Select** takes $O(n)$ expected time. But, in worst case it can take $O(n^2)$ time ☹
- **Median of Medians** can be found in $T(n) = O(n)$ time (as $T(n) = T(n/5) + O(n)$) which is guaranteed to be larger than $3n/10$ items



Recap: Order Statics

- Order statistic can be solved in $O(n\log(n))$ time, since we can sort
- As 1^{st}, 2^{nd} and n^{th} can be done in $O(n)$ time want same for others
- **Randomized-Select** takes $O(n)$ expected time. But, in worst case it can take $O(n^2)$ time ☹
- **Median of Medians** can be found in $T(n) = O(n)$ time (as $T(n) = T(n/5) + O(n)$) which is guaranteed to be larger than $3n/10$ items



- Running time for order statics is $T(n) = T(7n/10) + O(n)$ that solves to $O(n)$ in worst case. ☺

Hashing

- Hash table is a dynamic set that supports dictionary operations Insert, Search, and Delete.

Hashing

- Hash table is a dynamic set that supports dictionary operations Insert, Search, and Delete.
- Expected time to search for an element in a hash table is $O(1)$.

Hashing

- Hash table is a dynamic set that supports dictionary operations Insert, Search, and Delete.
- Expected time to search for an element in a hash table is $O(1)$.

Direct-address Table

1. Used when the universe U of keys is reasonably small.

Hashing

- Hash table is a dynamic set that supports dictionary operations Insert, Search, and Delete.
- Expected time to search for an element in a hash table is $O(1)$.

Direct-address Table

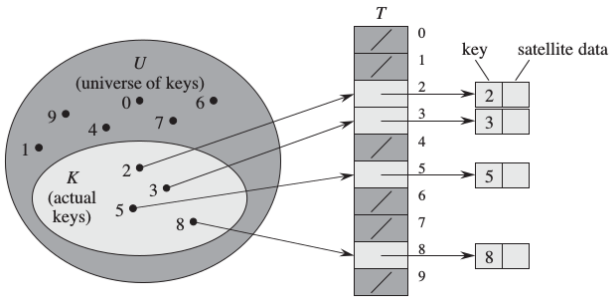
1. Used when the universe U of keys is reasonably small.
2. An array called direct-address-table represents the dynamic set.

Hashing

- Hash table is a dynamic set that supports dictionary operations Insert, Search, and Delete.
- Expected time to search for an element in a hash table is $O(1)$.

Direct-address Table

1. Used when the universe U of keys is reasonably small.
2. An array called direct-address-table represents the dynamic set.



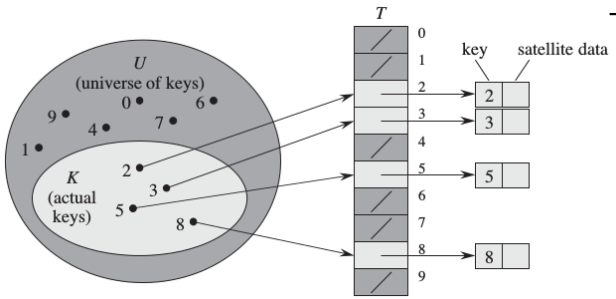
Hashing

- Hash table is a dynamic set that supports dictionary operations Insert, Search, and Delete.
- Expected time to search for an element in a hash table is $O(1)$.

Direct-address Table

1. Used when the universe U of keys is reasonably small.
2. An array called direct-address-table represents the dynamic set.

```
Direct-Address-Search(T, k)
    Return T[k]
```

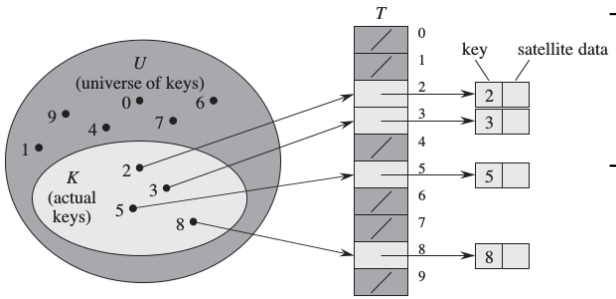


Hashing

- Hash table is a dynamic set that supports dictionary operations Insert, Search, and Delete.
- Expected time to search for an element in a hash table is $O(1)$.

Direct-address Table

1. Used when the universe U of keys is reasonably small.
2. An array called direct-address-table represents the dynamic set.




```
Direct-Address-Search(T, k)
    Return T[k]
```

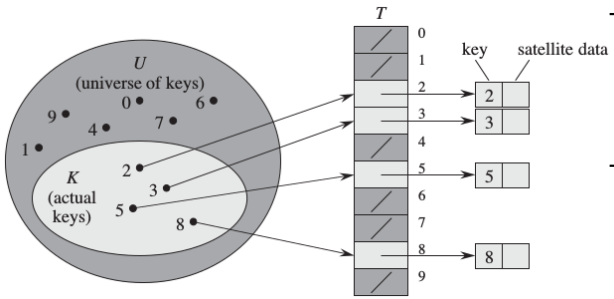
```
Direct-Address-Insert(T, x)
    T[key[x]] = x
```

Hashing

- Hash table is a dynamic set that supports dictionary operations Insert, Search, and Delete.
- Expected time to search for an element in a hash table is $O(1)$.

Direct-address Table

1. Used when the universe U of keys is reasonably small.
2. An array called direct-address-table represents the dynamic set.



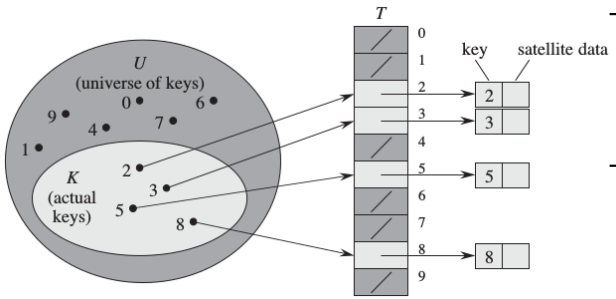
Direct-Address-Search(T, k)
Return T[k]
Direct-Address-Insert(T, x)
T[key[x]] = x
Direct-Address-Delete(T, x)
T[key[x]] = Nil

Hashing

- Hash table is a dynamic set that supports dictionary operations Insert, Search, and Delete.
- Expected time to search for an element in a hash table is $O(1)$.

Direct-address Table

1. Used when the universe U of keys is reasonably small.
2. An array called direct-address-table represents the dynamic set.



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

All operations take $O(n)$ time

Hash Table

- Hash table is used when the set of keys is much smaller then the size of universe

Hash Table

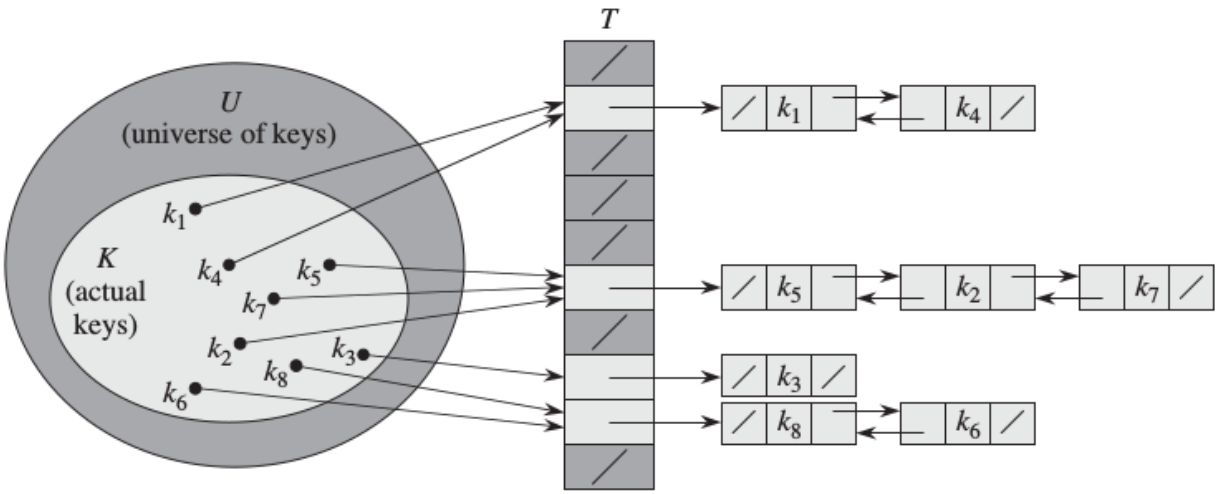
- Hash table is used when the set of keys is much smaller then the size of universe
- Element with key k is stored in slot $h(k)$ for $h : U \rightarrow \{0, ..., m-1\}$

Hash Table

- Hash table is used when the set of keys is much smaller then the size of universe
- Element with key k is stored in slot $h(k)$ for $h : U \rightarrow \{0, ..., m-1\}$
- **Collision:** two keys may hash to same slot. **Chaining** to resolve

Hash Table

- Hash table is used when the set of keys is much smaller then the size of universe
- Element with key k is stored in slot $h(k)$ for $h : U \rightarrow \{0, ..., m-1\}$
- **Collision:** two keys may hash to same slot. **Chaining** to resolve



Dictionary operations in Hash Table

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 $T[h(x.key)]$

Hash Table

- Insert item x at the head of list $T[h(key[x])]$ in $O(1)$ time

Hash Table

- Insert item x at the head of list $T[h(key[x])]$ in $O(1)$ time
- Worst case searching or deletion time is propositional to the length of the longest list. Hash tables are not used for their worst case behavior, it no good than a link list in that case.

Hash Table

- Insert item x at the head of list $T[h(key[x])]$ in $O(1)$ time
- Worst case searching or deletion time is propositional to the length of the longest list. Hash tables are not used for their worst case behavior, it no good than a link list in that case.
- With m slots storing n elements **load factor** $\alpha = n/m$

Hash Table

- Insert item x at the head of list $T[h(key[x])]$ in $O(1)$ time
- Worst case searching or deletion time is propositional to the length of the longest list. Hash tables are not used for their worst case behavior, it no good than a link list in that case.
- With m slots storing n elements **load factor** $\alpha = n/m$
- When any given element is equally likely to hash into any of the m slots, independently of whether any other element has hashed to, we call this the assumption of **simple uniform hashing**.

Hash Table

- Insert item x at the head of list $T[h(key[x])]$ in $O(1)$ time
- Worst case searching or deletion time is propositional to the length of the longest list. Hash tables are not used for their worst case behavior, it no good than a link list in that case.
- With m slots storing n elements **load factor** $\alpha = n/m$
- When any given element is equally likely to hash into any of the m slots, independently of whether any other element has hashed to, we call this the assumption of **simple uniform hashing**.
- Under this assumption search takes $\Theta(1 + \alpha)$ time on the average.

Hash Table

- Insert item x at the head of list $T[h(key[x])]$ in $O(1)$ time
- Worst case searching or deletion time is propositional to the length of the longest list. Hash tables are not used for their worst case behavior, it no good than a link list in that case.
- With m slots storing n elements **load factor** $\alpha = n/m$
- When any given element is equally likely to hash into any of the m slots, independently of whether any other element has hashed to, we call this the assumption of **simple uniform hashing**.
- Under this assumption search takes $\Theta(1 + \alpha)$ time on the average.
- Assuming α to be constant, operations essentially becomes $O(1)$.

Hash Table

- Insert item x at the head of list $T[h(key[x])]$ in $O(1)$ time
- Worst case searching or deletion time is propositional to the length of the longest list. Hash tables are not used for their worst case behavior, it no good than a link list in that case.
- With m slots storing n elements **load factor** $\alpha = n/m$
- When any given element is equally likely to hash into any of the m slots, independently of whether any other element has hashed to, we call this the assumption of **simple uniform hashing**.
- Under this assumption search takes $\Theta(1 + \alpha)$ time on the average.
- Assuming α to be constant, operations essentially becomes $O(1)$.
- Successful and unsuccessful search takes $\Theta(\alpha + 1)$ time on an average

Universal Hashing

- Example Hash Function
  - $h(k) = k \bmod m$
  - $h(k) = \lfloor m(k.a - \lfloor k.a \rfloor) \rfloor$

Universal Hashing

- Example Hash Function
  - $h(k) = k \bmod m$
  - $h(k) = \lfloor m(k.a - \lfloor k.a \rfloor) \rfloor$
- A malicious adversary may choose keys in such a way they all hash in same slot

Universal Hashing

- Example Hash Function
  - $h(k) = k \bmod m$
  - $h(k) = \lfloor m(k.a - \lfloor k.a \rfloor) \rfloor$
- A malicious adversary may choose keys in such a way they all hash in same slot
- Therefore, choose the hash function randomly independently of the keys to be stored
- The approach is called **universal hashing**

Universal Hashing

- Example Hash Function
  - $h(k) = k \bmod m$
  - $h(k) = \lfloor m(k.a - \lfloor k.a \rfloor) \rfloor$
- A malicious adversary may choose keys in such a way they all hash in same slot
- Therefore, choose the hash function randomly independently of the keys to be stored
- The approach is called **universal hashing**
- Let H be finite collection of hash function that map a given universe U of keys into the range $\{0, 1, 2, ..., m\}$. such a collection is said to be **universal** if for each pair of distinct keys $x, y \in U$, the number of hash functions $h \in H$ for which $h(x) = h(y)$ is precisely $|H|/m$.

Design of universal class of hash function

- Let m be a prime number

Design of universal class of hash function

- Let m be a prime number
- Decompose x into $r + 1$ equal parts $< x_0, x_1, x_2, ..., x_r >$ such that maximum value of a part does not exceed m.

Design of universal class of hash function

- Let m be a prime number
- Decompose x into $r+1$ equal parts $< x_0, x_1, x_2, ..., x_r >$ such that maximum value of a part does not exceed m.
- Let $a =< a_0, a_1, a_2, ..., a_r >$ denote a sequence of $r+1$ elements chosen randomly from the set $\{0, 1, 2, ..., m\}$.

Design of universal class of hash function

- Let m be a prime number
- Decompose x into $r+1$ equal parts $< x_0, x_1, x_2, ..., x_r >$ such that maximum value of a part does not exceed m.
- Let $a =< a_0, a_1, a_2, ..., a_r >$ denote a sequence of $r+1$ elements chosen randomly from the set $\{0, 1, 2, ..., m\}$.
- We define m^{r+1} size H, having hash function $h_a \in H$ as

$$h_a(x) = \sum_{i=0}^{r} a_i x_i \bmod m$$

Proof: Assume $x_0 \neq y_0$ for distinct x, y. For any $< a_1, a_2, ..., a_r >$ there is exactly one a_0 satisfying $h(x) = h(y)$; this a_0 is the solution to $a_0(x_0 - y_0) \equiv -\sum_{i=1}^{r} a_i(x_i - y_i) (\text{mod } m)$. Since m is prime, the nonzero quantity $x_0 - y_0$ has multiplicative inverse modulo m, and thus there is a unique solution for a_0 modulo m. Therefore, x and y collides on m^r values of a, as they collide exactly once for each possible values of $< a_1, a_2, ..., a_r >$. Since $|a| = m^{r+1}$, the key x and y collide with probability exactly $m^r/m^{r+1} = 1/m$. Therefore H is universal.

Thank You!

Thank you very much for your attention!

Queries ?