CS F364: Design & Analysis of Algorithms

Lecture-kt09: Hashing (contd.) + Binary Search Tree

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

Feb 09, 2017 (Campus @ BITS-Pilani Jan-May 2017)

Recap: Hashing

- **Hashing** supports dictionary operations Insert, Search, and Delete in $O(1)$ expected time

Recap: Hashing

- **Hashing** supports dictionary operations Insert, Search, and Delete in $O(1)$ expected time
- **Direct-address Table** is used when the universe U of keys is reasonably small. (All operations take $O(1)$ time)

Recap: Hashing

- **Hashing** supports dictionary operations Insert, Search, and Delete in $O(1)$ expected time
- **Direct-address Table** is used when the universe U of keys is reasonably small. (All operations take $O(1)$ time)

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

Recap: Hashing

- **Hashing** supports dictionary operations Insert, Search, and Delete in $O(1)$ expected time
- **Direct-address Table** is used when the universe U of keys is reasonably small. (All operations take $O(1)$ time)

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

Recap: Hashing

- **Hashing** supports dictionary operations Insert, Search, and Delete in $O(1)$ expected time
- **Direct-address Table** is used when the universe U of keys is reasonably small. (All operations take $O(1)$ time)

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

Recap: Hashing

- **Hashing** supports dictionary operations Insert, Search, and Delete in $O(1)$ expected time
- **Direct-address Table** is used when the universe U of keys is reasonably small. (All operations take $O(1)$ time)

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

- **Hash table** is used when the set of keys is much smaller then the size of universe. Uses $h : U \rightarrow \{0, ..., m - 1\}$. **Collision** is resolved using **chaining**

Recap: Hashing

- **Hashing** supports dictionary operations Insert, Search, and Delete in $O(1)$ expected time
- **Direct-address Table** is used when the universe U of keys is reasonably small. (All operations take $O(1)$ time)

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

- **Hash table** is used when the set of keys is much smaller then the size of universe. Uses $h : U \rightarrow \{0, ..., m-1\}$. **Collision** is resolved using **chaining**

Chained-Hash-Insert(T, x)	insert *x* at head of list T[h(x.key)]

Recap: Hashing

- **Hashing** supports dictionary operations Insert, Search, and Delete in $O(1)$ expected time
- **Direct-address Table** is used when the universe U of keys is reasonably small. (All operations take $O(1)$ time)

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

- **Hash table** is used when the set of keys is much smaller then the size of universe. Uses $h : U \to \{0, ..., m-1\}$. **Collision** is resolved using **chaining**

Chained-Hash-Insert(T, x)	insert *x* at head of list T[h(x.key)]
Chained-Hash-Search(T, k)	search *x* in list T[h(k)]

Recap: Hashing

- **Hashing** supports dictionary operations Insert, Search, and Delete in $O(1)$ expected time
- **Direct-address Table** is used when the universe U of keys is reasonably small. (All operations take $O(1)$ time)

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

- **Hash table** is used when the set of keys is much smaller then the size of universe. Uses $h : U \rightarrow \{0, ..., m-1\}$. **Collision** is resolved using **chaining**

Chained-Hash-Insert(T, x)	insert *x* at head of list T[h(x.key)]
Chained-Hash-Search(T, k)	search *x* in list T[h(k)]
Chained-Hash-Delete(T, x)	delete *x* from list T[h(x.key)]

Recap: Hashing

- **Hashing** supports dictionary operations Insert, Search, and Delete in $O(1)$ expected time
- **Direct-address Table** is used when the universe U of keys is reasonably small. (All operations take $O(1)$ time)

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

- **Hash table** is used when the set of keys is much smaller then the size of universe. Uses $h : U \rightarrow \{0, ..., m-1\}$. **Collision** is resolved using **chaining**

Chained-Hash-Insert(T, x)	insert *x* at head of list T[h(x.key)]
Chained-Hash-Search(T, k)	search *x* in list T[h(k)]
Chained-Hash-Delete(T, x)	delete *x* from list T[h(x.key)]

Assuming **load factor** $\alpha = n/m$ to be a constant under **simple uniform hashing** all the operations becomes $O(1)$ time

Universal Hashing

- 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

- 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 a universal class of hash function

- Let $a \in \{0, 1, 2, ..., m-1\}^{r+1}$ where m is prime.

Design of a universal class of hash function

- Let $a \in \{0, 1, 2, ..., m-1\}^{r+1}$ where m is prime.
- Decompose x as $< x_0, x_1, x_2, ..., x_r >$ where $x_i < m \;\; \forall i$

Design of a universal class of hash function

- Let $a \in \{0, 1, 2, ..., m-1\}^{r+1}$ where m is prime.
- Decompose x as $< x_0, x_1, x_2, ..., x_r >$ where $x_i < m \ \ \forall i$
- Define $h_a \in H$ as $\quad h_a(x) = \sum_{i=0}^{r} a_i x_i \pmod{m}$

Design of a universal class of hash function

- Let $a \in \{0, 1, 2, ..., m - 1\}^{r+1}$ where m is prime.
- Decompose x as $< x_0, x_1, x_2, ..., x_r >$ where $x_i < m \;\; \forall i$
- Define $h_a \in H$ as $\qquad h_a(x) = \sum_{i=0}^{r} a_i x_i \pmod{m}$
- Size of H is m^{r+1}

Design of a universal class of hash function

- Let $a \in \{0, 1, 2, ..., m-1\}^{r+1}$ where m is prime.
- Decompose x as $< x_0, x_1, x_2, ..., x_r >$ where $x_i < m \;\; \forall i$
- Define $h_a \in H$ as $\qquad h_a(x) = \sum_{i=0}^{r} a_i x_i \pmod{m}$
- Size of H is m^{r+1}

Proof: Let $x =< x_0, x_1, x_2, ..., x_r >$, $y =< y_0, y_1, y_2, ..., y_r >$

Design of a universal class of hash function

- Let $a \in \{0, 1, 2, ..., m-1\}^{r+1}$ where m is prime.
- Decompose x as $< x_0, x_1, x_2, ..., x_r >$ where $x_i < m \ \ \forall i$
- Define $h_a \in H$ as $\qquad h_a(x) = \sum_{i=0}^{r} a_i x_i \pmod m$
- Size of H is m^{r+1}

Proof: Let $x =< x_0, x_1, x_2, ..., x_r >$, $y =< y_0, y_1, y_2, ..., y_r >$

$h_a(x) = h_a(y) \Rightarrow \boxed{a_0 = (x_0 - y_0)^{-1} \sum_{i=1}^{r} a_i(y_i - x_i)}$

Design of a universal class of hash function

- Let $a \in \{0, 1, 2, ..., m-1\}^{r+1}$ where m is prime.
- Decompose x as $< x_0, x_1, x_2, ..., x_r >$ where $x_i < m \ \ \forall i$
- Define $h_a \in H$ as $\qquad h_a(x) = \sum_{i=0}^{r} a_i x_i \pmod{m}$
- Size of H is m^{r+1}

Proof: Let $x =< x_0, x_1, x_2, ..., x_r >$, $y =< y_0, y_1, y_2, ..., y_r >$

$$h_a(x) = h_a(y) \Rightarrow \boxed{a_0 = (x_0 - y_0)^{-1} \sum_{i=1}^{r} a_i(y_i - x_i)}$$

- This equation can be satisfied in m^r ways. Because one can choose $a_i \in \{0, 1, 2, ..., m-1\}^r$ in m^r ways and for given x and y it gives unique a_0

Design of a universal class of hash function

- Let $a \in \{0, 1, 2, ..., m-1\}^{r+1}$ where m is prime.
- Decompose x as $< x_0, x_1, x_2, ..., x_r >$ where $x_i < m \;\; \forall i$
- Define $h_a \in H$ as $\qquad h_a(x) = \sum_{i=0}^{r} a_i x_i \pmod{m}$
- Size of H is m^{r+1}

Proof: Let $x =< x_0, x_1, x_2, ..., x_r >$, $y =< y_0, y_1, y_2, ..., y_r >$
$h_a(x) = h_a(y) \Rightarrow \boxed{a_0 = (x_0 - y_0)^{-1} \sum_{i=1}^{r} a_i(y_i - x_i)}$

- This equation can be satisfied in m^r ways. Because one can choose $a_i \in \{0, 1, 2, ..., m-1\}^r$ in m^r ways and for given x and y it gives unique a_0

Therefore, x and y collides on m^r values of a, as they collide exactly once for each possible values of $< a_1, a_2, ..., a_r >$.

Design of a universal class of hash function

- Let $a \in \{0, 1, 2, ..., m-1\}^{r+1}$ where m is prime.
- Decompose x as $< x_0, x_1, x_2, ..., x_r >$ where $x_i < m \ \ \forall i$
- Define $h_a \in H$ as $\qquad h_a(x) = \sum_{i=0}^{r} a_i x_i \pmod{m}$
- Size of H is m^{r+1}

Proof: Let $x =< x_0, x_1, x_2, ..., x_r >$, $y =< y_0, y_1, y_2, ..., y_r >$

$h_a(x) = h_a(y) \Rightarrow \boxed{a_0 = (x_0 - y_0)^{-1} \sum_{i=1}^{r} a_i(y_i - x_i)}$

- This equation can be satisfied in m^r ways. Because one can choose $a_i \in \{0, 1, 2, ..., m-1\}^r$ in m^r ways and for given x and y it gives unique a_0

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 $m^r/m^{r+1} = 1/m$.

Design of a universal class of hash function

- Let $a \in \{0, 1, 2, ..., m-1\}^{r+1}$ where m is prime.
- Decompose x as $< x_0, x_1, x_2, ..., x_r >$ where $x_i < m \;\; \forall i$
- Define $h_a \in H$ as $\quad h_a(x) = \sum_{i=0}^{r} a_i x_i \pmod m$
- Size of H is m^{r+1}

Proof: Let $x =< x_0, x_1, x_2, ..., x_r >$, $y =< y_0, y_1, y_2, ..., y_r >$
$h_a(x) = h_a(y) \Rightarrow \boxed{a_0 = (x_0 - y_0)^{-1} \sum_{i=1}^{r} a_i(y_i - x_i)}$

- This equation can be satisfied in m^r ways. Because one can choose $a_i \in \{0, 1, 2, ..., m-1\}^r$ in m^r ways and for given x and y it gives unique a_0

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.

Open Addressing

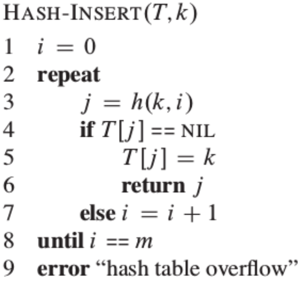
- In open addressing elements are stored in the hash table itself.
- Table entry contains either an element or Nil.

Open Addressing

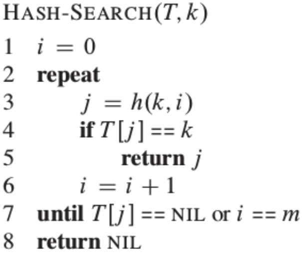
- In open addressing elements are stored in the hash table itself.
- Table entry contains either an element or Nil.
- To perform insertion, we successively examine, or probe the hash table until we find an empty slot in which to put the key.

Open Addressing

- In open addressing elements are stored in the hash table itself.
- Table entry contains either an element or Nil.
- To perform insertion, we successively examine, or probe the hash table until we find an empty slot in which to put the key.



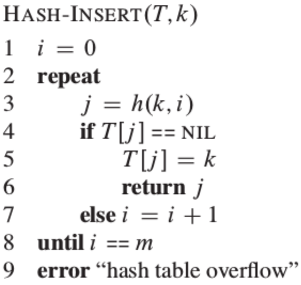
```
HASH-INSERT$(T, k)$
$i = 0$
repeat
    $j = h(k, i)$
    if $T[j]$ == NIL
        $T[j] = k$
        return $j$
    else $i = i + 1$
until $i$ == $m$
error "hash table overflow"
```



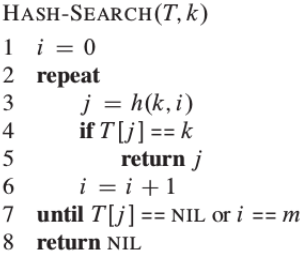
```
HASH-SEARCH$(T, k)$
$i = 0$
repeat
    $j = h(k, i)$
    if $T[j]$ == $k$
        return $j$
    $i = i + 1$
until $T[j]$ == NIL or $i$ == $m$
return NIL
```

Open Addressing

- In open addressing elements are stored in the hash table itself.
- Table entry contains either an element or Nil.
- To perform insertion, we successively examine, or probe the hash table until we find an empty slot in which to put the key.



```
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
```

Deletion from an open-address hash table is difficult.

Probe sequence

Probe sequence is $< h(k,0), h(k,1), ..., h(k,m-1) >$ that can be either

- ***Linear Probing:***
  $h(k,i) = (h'(k) + i) \bmod m$
  That suffers from primary clustering

Probe sequence

Probe sequence is $< h(k,0), h(k,1), ..., h(k,m-1) >$ that can be either

- ***Linear Probing:***
  $h(k,i) = (h'(k) + i) \bmod m$
  That suffers from primary clustering
- ***Quadratic Probing:***
  $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$
  That suffers from secondary clustering

Probe sequence

Probe sequence is $< h(k,0), h(k,1), ..., h(k,m-1) >$ that can be either

- ***Linear Probing:***
  $h(k,i) = (h'(k) + i) \bmod m$
  That suffers from primary clustering
- ***Quadratic Probing:***
  $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$
  That suffers from secondary clustering
- ***Double Hashing:***
  $h(k,i) = (h_1(k) + i \times h_2(k)) \bmod m$

In open addressing an unsuccessful search has probe $1/(1-\alpha)$ and in successful one is $\frac{1}{\alpha} \ln \frac{1}{1-\alpha} + \frac{1}{\alpha}$. Where $\alpha = n/m < 1$.

Binary Search Tree

- Search tree data structure supports many dynamic-set operations, including SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, and DELETE

Binary Search Tree

- Search tree data structure supports many dynamic-set operations, including SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, and DELETE
- Can be used as a dictionary and as a priority queue

Binary Search Tree

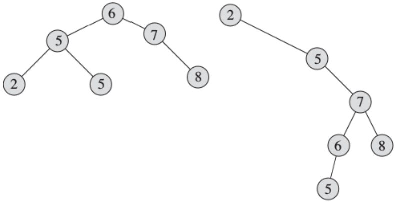
- Search tree data structure supports many dynamic-set operations, including SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, and DELETE
- Can be used as a dictionary and as a priority queue
- Basic operations on a binary search tree take time proportional to the height of the tree

Binary Search Tree

- Search tree data structure supports many dynamic-set operations, including SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, and DELETE
- Can be used as a dictionary and as a priority queue
- Basic operations on a binary search tree take time proportional to the height of the tree
- **Binary-search-tree property:** Let x be a node in a binary search tree. If y is a node in the
    - **left** subtree of x, then $y.key \leq x.key$
    - **right** subtree of x, then $y.key \geq x.key$.

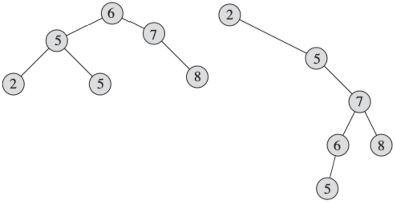
Binary Search Tree

- Search tree data structure supports many dynamic-set operations, including SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, and DELETE
- Can be used as a dictionary and as a priority queue
- Basic operations on a binary search tree take time proportional to the height of the tree
- **Binary-search-tree property:** Let x be a node in a binary search tree. If y is a node in the
    - **left** subtree of x, then $y.key \leq x.key$
    - **right** subtree of x, then $y.key \geq x.key$.



Binary Search Tree

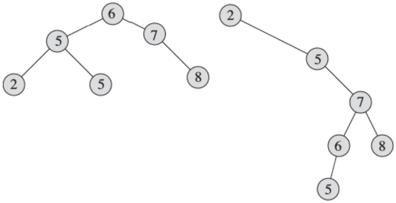
- Search tree data structure supports many dynamic-set operations, including SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, and DELETE
- Can be used as a dictionary and as a priority queue
- Basic operations on a binary search tree take time proportional to the height of the tree
- **Binary-search-tree property:** Let *x* be a node in a binary search tree. If *y* is a node in the
    - **left** subtree of *x*, then $y.key \leq x.key$
    - **right** subtree of *x*, then $y.key \geq x.key$.



- We may safely assume all items to be different

Binary Search Tree

- Search tree data structure supports many dynamic-set operations, including SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, and DELETE
- Can be used as a dictionary and as a priority queue
- Basic operations on a binary search tree take time proportional to the height of the tree
- **Binary-search-tree property:** Let x be a node in a binary search tree. If y is a node in the
    - **left** subtree of x, then $y.key \leq x.key$
    - **right** subtree of x, then $y.key \geq x.key$.



- We may safely assume all items to be different
- In order walk is monotonous

Thank You!

Thank you very much for your attention!

Queries ?