UNIT V
GRAPHS
Syllabus

Graph Terminology - Graph Traversal - Topological sorting - Minimum spanning tree - Prims - Kruskals - Network flow problem - Bi-connected components - Hashing - Hash functions - Collision avoidance - Separate chaining - Open addressing - Linear probing - Quadratic Probing - Rehashing - Extensible Hashing
Collision

- There is possibility that two keys result in same value. The situation where a newly inserted key maps to an already occupied slot in hash table is called collision

How to handle Collisions?

1) Separate Chaining
2) Open Addressing
Separate Chaining:

- The idea is to make each cell of hash table point to a linked list of records that have same hash function value.
- Let us consider a simple hash function as “key mod 7” and sequence of keys as 50, 700, 76, 85, 92, 73, 101.
Consider a simple hash function as “key mod 7” and sequence of keys as 50, 700, 76, 85, 92, 73, 101.

<table>
<thead>
<tr>
<th>S.n.</th>
<th>Key</th>
<th>Hash</th>
<th>Array Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>50</td>
<td>50%7 = 1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>700</td>
<td>700%7 = 0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>76</td>
<td>76%7 = 6</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>85</td>
<td>85%7 = 1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>92</td>
<td>92%7 = 1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>73</td>
<td>73%7 = 3</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>101</td>
<td>101%7 = 3</td>
<td>3</td>
</tr>
</tbody>
</table>
Consider a simple hash function as “key mod 7” and sequence of keys as 50, 700, 76, 85, 92, 73, 101
Separate chaining pros and cons

Advantages:
1) Simple to implement.
2) Hash table never fills up, we can always add more elements to chain
3) Less sensitive to the hash function or load factors.
4) It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

Disadvantages:
1) Cache performance of chaining is not good as keys are stored in LL.
2) Wastage of Space (Some Parts of hash table are never used)
3) If the chain becomes long, search time can be O(n) in worst case
4) Uses extra space for links.
Open Addressing

- In Open Addressing, all elements are stored in the hash table itself.
- So at any point, size of table must be greater than or equal to total number of keys (Note that we can increase table size by copying old data if needed).
- Insert(k): Keep probing until an empty slot is found. Once an empty slot is found, insert k.
- Search(k): Keep probing until slot’s key doesn’t become equal to k or an empty slot is reached.
- Delete(k): Delete operation is interesting. If we simply delete a key, then search may fail. So slots of deleted keys are marked specially as “deleted”.
- Insert can insert an item in a deleted slot, but search doesn’t stop at a deleted slot.
Open Addressing is done following ways:

a) Linear Probing
b) Quadratic Probing
c) Double Hashing
Linear Probing

- We look for $i^{2\text{th}}$ slot in $i^{\text{th}}$ iteration.
- Let $hash(x)$ be the slot index computed using hash function.
  - If slot $hash(x) \mod S$ is full, then we try $(hash(x) + 1) \mod S$
  - If $(hash(x) + 1) \mod S$ is also full, then we try $(hash(x) + 2) \mod S$
  - If $(hash(x) + 2) \mod S$ is also full, then we try $(hash(x) + 3) \mod S$

Let us consider a simple hash function as “key mod 7” and sequence of keys as 50, 700, 76, 85, 92, 73, 101.

Clustering:
- The main problem with linear probing is clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search an element.
Consider a simple hash function as “key mod 7” and sequence of keys as 50, 700, 76, 85, 92, 73, 101

<table>
<thead>
<tr>
<th>S.n.</th>
<th>Key</th>
<th>Hash</th>
<th>Array Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>50</td>
<td>$50 % 7 = 1$</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>700</td>
<td>$700 % 7 = 0$</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>76</td>
<td>$76 % 7 = 6$</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>85</td>
<td>$85 % 7 = 1$</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>92</td>
<td>$92 % 7 = 1$</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>73</td>
<td>$73 % 7 = 3$</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>101</td>
<td>$101 % 7 = 3$</td>
<td>3</td>
</tr>
</tbody>
</table>
Consider a simple hash function as “key mod 7” and sequence of keys as 50, 700, 76, 85, 92, 73, 101.
Quadratic Probing

- We look for $i^{2\text{th}}$ slot in $i^{\text{th}}$ iteration.
- Let $\text{hash}(x)$ be the slot index computed using hash function.
  - If slot $\text{hash}(x) \% S$ is full, then we try $(\text{hash}(x) + 1\times 1) \% S$
  - If $(\text{hash}(x) + 1\times 1) \% S$ is also full, then we try $(\text{hash}(x) + 2\times 2) \% S$
  - If $(\text{hash}(x) + 2\times 2) \% S$ is also full, then we try $(\text{hash}(x) + 3\times 3) \% S$
Double Hashing

- We use another hash function $\text{hash2}(x)$ and look for $i\times \text{hash2}(x)$ slot in $i$’th rotation.
- Let $\text{hash}(x)$ be the slot index computed using hash function.
  - If slot $\text{hash}(x) \mod S$ is full, then we try $(\text{hash}(x) + 1\times \text{hash2}(x)) \mod S$
  - If $(\text{hash}(x)+1\times \text{hash2}(x))\mod S$ is full, then $(\text{hash}(x) + 2\times \text{hash2}(x)) \mod S$
  - If $(\text{hash}(x) + 2\times \text{hash2}(x)) \mod S$ is full, then $(\text{hash}(x) + 3\times \text{hash2}(x)) \mod S$
## Comparison

<table>
<thead>
<tr>
<th></th>
<th>Linear Probing</th>
<th>Quadratic Probing</th>
<th>Double Hashing</th>
</tr>
</thead>
<tbody>
<tr>
<td>ith iteration</td>
<td>(hash(x) + i)</td>
<td>(hash(x) + i*i)</td>
<td>(hash(x)+i*(hash2(x))</td>
</tr>
<tr>
<td>Cache Performance</td>
<td>Best</td>
<td>Medium</td>
<td>Worst</td>
</tr>
<tr>
<td>Clustering</td>
<td>Worst</td>
<td>Medium</td>
<td>Best</td>
</tr>
<tr>
<td>Computation time</td>
<td>Less</td>
<td>Moderate</td>
<td>More</td>
</tr>
</tbody>
</table>
Solve Worksheet 8 & 9
Thank You