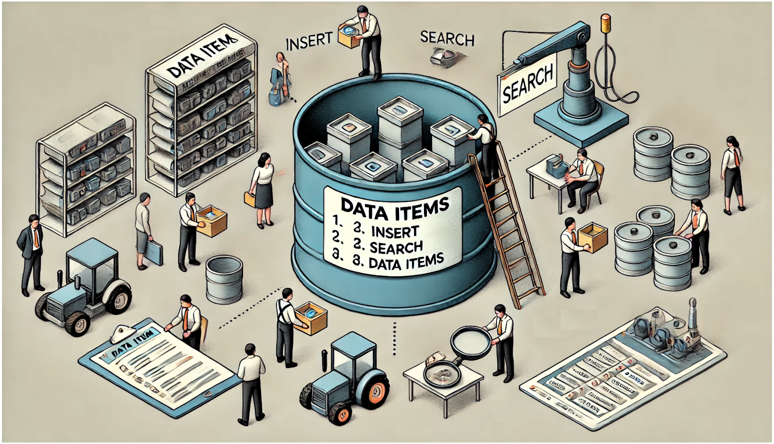
Hash Table

Course: Design and Analysis of Algorithms

Dr. Laltu Sardar

Institute for Advancing Intelligence (IAI),
TCG Centres for Research and Education in Science and Technology (TCG Crest)

November 4, 2024

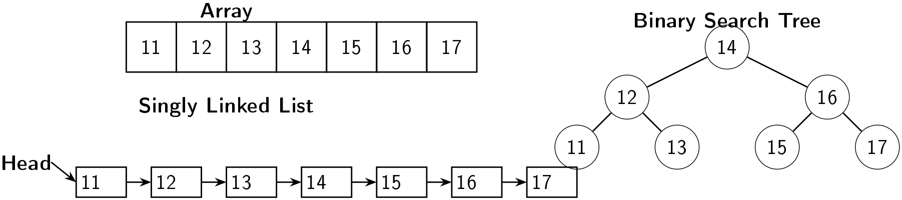


- Computer science starts when we require to compute over data.
- Data can be seen as binary strings (of fixed or variable size).
- Often data is not processed immediately => need to be **stored**.
- Whenever require, first **search** to retrieve, If necessary, we need to **add**/**append** or **delete**/**remove** new/old data items.

tcg crest
Inventing Harmonious Future

Data Structures

- For efficient handling we need to structure data
- Example Data Structures: Arrays, linked lists, BST,
- Main operations are - add, search, and delete



Data Structures: Time complexities

Data Structure	**Add (Insert)**	**Search**	**Delete** (after search)
Array	O(n)	O(n)	O(1)
Linked List	O(1)	O(n)	O(1)
Binary Search Tree	O(log n)	O(log n)	O(log n)

Table: Time Complexity: in General

Question: Can we both **add and search in O(1) time**?
Solution: **Hash Table** data structure

Dictionary and Hash Table

Dictionary

- A dictionary is a list/set of key-value pairs.
- For example: Key can be a **word** and value can be **meaning string**
- We add key-value pair, search with key and delete value having key

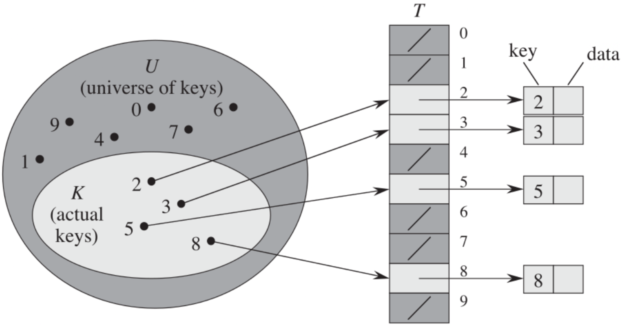
Hash Table

- is a data structure that store values (and keys).
- Mainly is an array (or combination of array and linked list)
- A **hash function** computes an index from the key, and the value is stored at that index.
- Handles operations like **Insert**, **Search**, and **Delete**.

Direct-address Table

Direct hash table

- Suppose, keys are from $\{0, 1, 2, \cdots, m-1\}$ (key universe)
- Then we can take an array T of length m.
- store item x with key $x.key$ at $T[x.key] = x$ $(x = \{key, value\})$



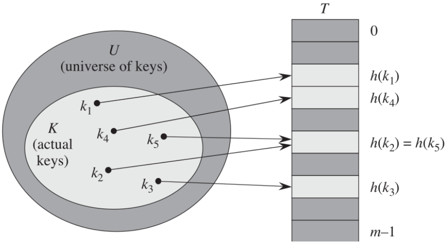
Hash Table

Let $K=$ set of all keys in the table, $\mathcal{U}$ = set of all possible keys

What if key-universe is too big?

Hashing

- hash map $h : \{0,1\}^* \rightarrow \{0,1\}^k$, (Key-universe to table index)
- Maps arbitrary size keys to fixed-size
- For example $k = log|K|$, $h(x.key)$ maps to some index of T



Collision

Since domain is too big, collision is inevitable.

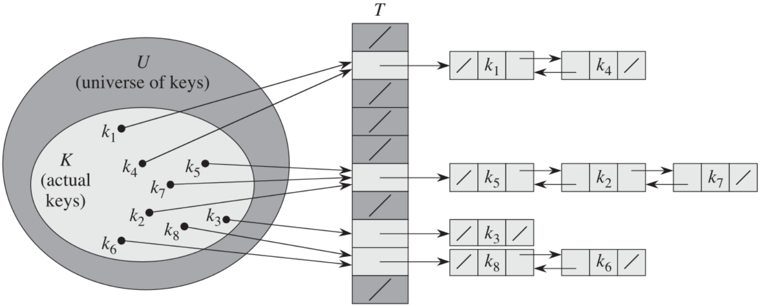
How to handle collision?

1. Chaining
   - Items with keys that maps to same index, are kept in a linked list (chain)
   - Instead of keeping items in the array T, head of the linked lists are kept in T
2. Open Addressing
   - Items are kept the array T.
   - If collision occurs, find for the next possible empty space in T

Chaining

In **chaining**, each index in the hash table points to a linked list of key-value pairs that have the same hash value.

- When a collision occurs, the new entry is added to the end of the list.
- Searching, inserting, and deleting involve traversing the list at the specific index.



Pseudocode for Insertion with Chaining

```
procedure Insert(key, value)
    index = HashFunction(key)
    if table[index] is empty then
        table[index] = new linked_list
    end if
    table[index].append((key, value))
end procedure
```

Chaining

Complexity

If hash function distributes keys well (uniformly)

- Collision is minimal
- Every keys will be mapped to a single index, on average
- average case: Searching, inserting, and deleting can be done in $O(1)$ time
- worst case: All maps to same index, is same as a single linked list

Problems

Cache Performance not Good: Elements are not stored contiguously,

Analysis of Chaining

hash table T $\rightarrow$ with m slots $\rightarrow$ stores n elements.
Load factor $\alpha = n/m$

average-case

depends on how well h distributes K among the m slots.

Theorem

successful/ unsuccessful search $\rightarrow$ *average-case time* $\Theta(1 + \alpha)$, *under the assumption of simple uniform hashing.*

Simple Uniform Hashing

- Each key is equally likely to be mapped to any of the available slots.
- The distribution of keys into the slots is completely random and independent of the keys themselves.

Universal Hashing

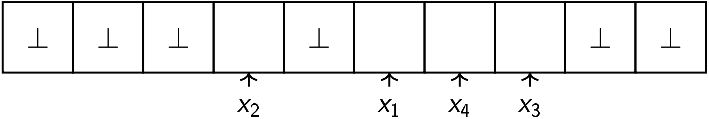
is a method in which a hash function is randomly chosen from a family of hash functions, providing a probabilistic guarantee against worst-case collision.

Example of a Universal Hash Family

- **Hash Function Family**: Consider the hash function $h_{a,b}(x) = ((a \cdot x + b) \mod p) \mod m$
- **Parameters**:
  - p: A large prime number $> |U|$
  - m: Size of the hash table.
  - a, b: Random integers chosen such that $1 \leq a < p$ and $0 \leq b < p$.
- This family is universal, providing a uniform distribution of hashed values across the table.

Open Addressing

- All items are stored in the hash table itself (contiguous memory).
- When a collision occurs, the algorithm searches for the next available slot in the table.



$$h(x_1.\text{key}) = 5 \Rightarrow T[5] = x_1$$
$$h(x_2.\text{key}) = 3 \Rightarrow T[3] == x_2$$
$$h(x_3.\text{key}) = 7 \Rightarrow T[7] = x_3$$
$$h(x_4.\text{key}) = 5 \Rightarrow \text{Collision} \Rightarrow T[6] == x_4$$

Pseudocode for Open Addressing I

```
    function insert(key)
        i = 0
        repeat
            j = (hash(key) + probe(i)) mod m
            if table[j] is empty or marked deleted then
                table[j] = key
                return
            i = i + 1
        until i == m
        error "Hash␣table␣is␣full"
```

```
    function search(key)
        i = 0
        repeat
            j = (hash(key) + probe(i)) mod m
            if table[j] == key then
                return j
            else if table[j] is empty then
                return "Not␣found"
            i = i + 1
```

Pseudocode for Open Addressing II

```
        until i == m
        return "Not␣found"
```

```
    function delete(key)
        location = search(key)
        if location is not "Not␣found" then
            table[location] = marked deleted
```

Complexity: Average Case: takes $O(1)$ time
Worst Case: $O(m)$, where m is the size of the hash table.

Probing

How to determine next slot, if there is collision?

- **Linear Probing**: Search the next consecutive slot.
    - $h(k, i) = (h(k) + i) \mod m$
    - Fast but can lead to primary clustering.
- **Quadratic Probing**: Use a quadratic function.
    - $h(k, i) = (h(k) + c_1 \cdot i + c_2 \cdot i^2) \mod m$
    - Reduces primary clustering but introduces secondary clustering.
- **Double Hashing**: Use a second hash function.
    - $h(k, i) = (h_1(k) + i \cdot h_2(k)) \mod m$
    - Minimizes clustering but is more complex.

tcg crest
Inventing Harmonious Future

Expected Performance

Successful Search:

- The expected number of probes is approximately: $\frac{1}{\alpha} \ln\left(\frac{1}{1-\alpha}\right)$
- Example: If $\alpha = 0.5$ (50% load factor), then $\frac{1}{0.5} \ln\left(\frac{1}{1-0.5}\right) = 2\ln(2) \approx 1.39$
- This is close to $O(1)$

Unsuccessful Search:

- The expected number of probes is approximately: $\frac{1}{1-\alpha}$
- Example: If $\alpha = 0.75$ (75% load factor), then $\frac{1}{1-0.75} = 4$
- This implies that, on average, 4 probes are needed

Insertion:

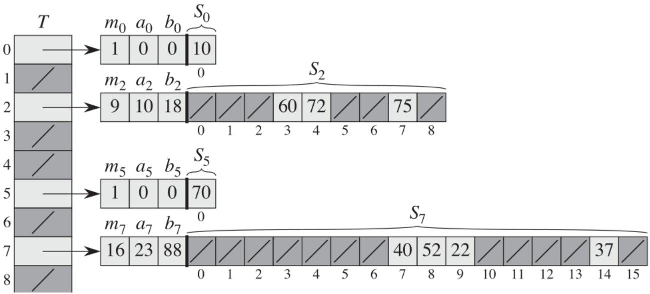
- The expected time is similar unsuccessful search: $O\left(\frac{1}{1-\alpha}\right)$
- Example: If $\alpha = 0.85$ (85% load factor), then $O\left(\frac{1}{1-0.85}\right) = O(6.67)$

Indicating performance degradation as α increases.

Perfect Hashing

Perfect Hashing is a technique where no collisions occur. It is usually implemented with two-level hashing:

- The first hash function distributes keys into buckets.
- The second level uses a perfect hash function to resolve collisions within each bucket.
- Used in **static sets**



tcg crest
Inventing Harmonious Future

Dr. Laltu Sardar
laltu.sardar@tcgcrest.org
https://laltu-sardar.github.io.