Tables

Searching, at best, can be done in $O(\log(n))$ time.

Array indexing is $O(1)$ — can we do information retrieval that quickly?

Generalize arrays as tables — may be $n$-dimensional.

Since memory is 1-dimensional, we need to convert the index (sequence of integers) to an address:

- **Row-major ordering** elements in the same row are adjacent
- **Column-major ordering** elements in the same column are adjacent

C++ (and most languages) uses Row-major ordering, i.e.,

```cpp
int A[10][5]; // 10 rows, 5 columns
for (int r = 0; r < 10; r++) {
    for (int c = 0; c < 5; c++) {
        cout << A[r][c]; // output in order in memory
    }
}
```

The location (address) of $A[r][c]$ is the same as the address of $A[0][0]$ plus $5r + c$.

$5r + c$ is an **index function** — it maps an index to a location

For irregular tables (i.e., rows are of varying lengths) store the offset to the start of each row in a separate access array.

Several access arrays can be used to give different sort orders for the same data (e.g., by name, by phone number, by address).

---

**Table Specification (a.k.a. Map)**

**Description** Map from the index set, $I$, to the base type, $T$.

**State** A function $F : I \mapsto T$ (Equivalently a set $F \subseteq (I \times T)$)

**Operations**

- **table()** — Constructor.
  
  **Post:** $F = \emptyset$ $F$ is the empty set.

- **table()** — Destructor.

- **T retrieve(I i)** — Table access.
  
  **Post:** $\text{Result} \in t \ s.t. (i, t) \in F$ $\text{Result}$ is the value indexed by $i$.

- **insert(I i, T t)** — Insert $(i, t)$ into $F$
  
  **Post:** $(i, t) \in F'$ $\neg (\exists r \in T, r \neq t \land (i, r) \in F')$ $i$ indexes $t$ in the new table.

- **remove(I i)** — Remove $(i, t)$ from $F$
  
  **Post:** $\neg (\exists t \in T, (i, t) \in F')$ The value indexed by $i$ is not in the table.

- Retrieval should be $O(1)$ time.

- There is no requirement of order on $I$—traversal of a table doesn’t always make sense.

- The index set $I$ need not be integers or other numeric type (but we need to figure out some way to map it to natural numbers).
Hash Tables

sparse table: I is large but the domain is relatively small. (i.e., we don’t expect to use all of I)

In a hash table many different indices map to the same location in the array (called a bucket).

A Hash Function maps from index to bucket.

Characteristics of a good hash function:
• Easy and quick to compute.
• Give an even distribution of actual data throughout table.
• Must be deterministic and stateless—the same argument must always give the same result.

Example hash functions:
- **Truncation** ignore part of the key, use the rest (e.g., 9530365 maps to 365).
- **Folding** partition key into parts, combine the parts (e.g., 9530365 maps to (953 + 36 + 5) = 994).
- **Modular Arithmetic** convert to an integer (using one of the above) and take % # of buckets.
  - Distribution is dependent on divisor (# of buckets).
  - Choose prime number. Why?

A collision occurs when the bucket is already in use.

Collision Resolution: Open Addressing

When a collision occurs (either insert or retrieve) we must choose/search a new location.

**Linear Probing** Try the adjacent bucket until we find a space.

Clustering is a problem—buckets tend to fill up in clusters, which increases probability of collision.

**Rehashing** Use a second (third, fourth . . . ) hashing function.

**Quadratic Probing** If h fails, try h + 1, then h + 4, h + 9, . . . , h + i^2

If the table size is prime then this will check up to half of the buckets.

Let n be the number of entries in the table and t be the number of buckets.

**Load factor** (\( \lambda = n/t \)) — the ratio of full buckets to the total # of buckets. \((0 \leq \lambda \leq 1)\)

- Insertion/retrieval becomes slower (more collisions) as \( \lambda \) approaches 1.
- Quadratic probing may overflow if \( \lambda \geq 0.5 \).
- Worst case insertion/retrieval time complexity = \( O(n) \).
- When an item is deleted the bucket must be marked specially.
  - Empty cells are used to stop probing.
  - Need to distinguish between “never been full” and “was full, now empty”
- Algorithms are complicated by deletion.
Collision Resolution: Separate Chaining

Each bucket contains a list of elements.

- Space efficient if records are large.
- Overflow is not a problem (i.e., $\lambda$ is limited only by available memory).
- Deletion is easy.

But . . .

- Overhead for lists (may be significant if records are small).
- Worst case time complexity is still $O(n)$.

Analysis

How many “probes” (comparisons) does it take to retrieve an element?

Chaining

Assume list it has $k$ entries.

Assume uniform distribution: $E(k) = n/t = \lambda$

Unsuccessful search will search the whole list $E(\text{probes}) = \lambda$

Successful search will, on average, search half of it ($\frac{1}{2}(k + 1)$), but $E(k) = 1 + (n - 1)/t \approx 1 + \lambda$ so $E(\text{probes}) = 1 + \frac{\lambda}{2}$

Open Addressing

Linear probing:

$$E(\text{probes}) = \begin{cases} 
\frac{1}{2} \left( 1 + \frac{1}{1-\lambda} \right) & \text{if successful} \\
\frac{1}{2} \left( 1 + \frac{1}{(1-\lambda)^2} \right) & \text{if unsuccessful} 
\end{cases}$$