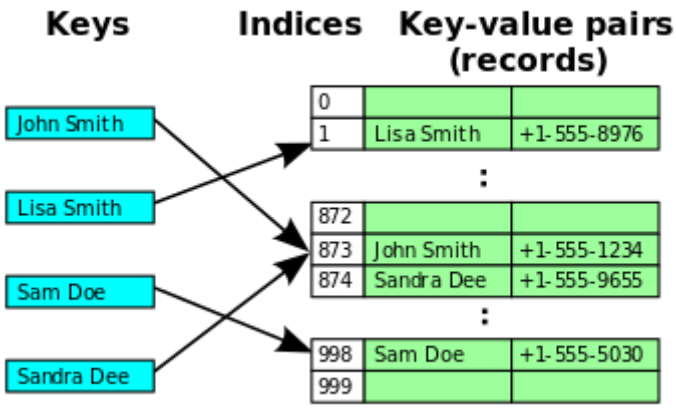
Open addressing

Open addressing, or **closed hashing**, is a method of collision resolution in hash tables. With this method a hash collision is resolved by **probing**, or searching through alternative locations in the array (the *probe sequence*) until either the target record is found, or an unused array slot is found, which indicates that there is no such key in the table.[1] Well-known probe sequences include:



Hash collision resolved by linear probing (interval=1).

Linear probing
: in which the interval between probes is fixed — often set to 1.

Quadratic probing
: in which the interval between probes increases quadratically (hence, the indices are described by a quadratic function).

Double hashing
: in which the interval between probes is fixed for each record but is computed by another hash function.

The main trade offs between these methods are that linear probing has the best cache performance but is most sensitive to clustering, while double hashing has poor cache performance but exhibits virtually no clustering; quadratic probing falls in-between in both areas. Double hashing can also require more computation than other forms of probing.

Some open addressing methods, such as Hopscotch hashing, Robin Hood hashing, last-come-first-served hashing and cuckoo hashing move existing keys around in the array to make room for the new key. This gives better maximum search times than the methods based on probing. [2][3][4][5][6]

A critical influence on performance of an open addressing hash table is the *load factor*; that is, the proportion of the slots in the array that are used. As the load factor increases towards 100%, the number of probes that may be required to find or insert a given key rises dramatically. Once the table becomes full, probing algorithms may even fail to terminate. Even with good hash functions, load factors are normally limited to 80%. A poor hash function can exhibit poor performance even at very low load factors by generating significant clustering, especially with the simplest linear addressing method. Generally typical load factors with most open addressing methods are 50%, whilst separate chaining typically can use up to 100%.

Example pseudocode

The following pseudocode is an implementation of an open addressing hash table with linear probing and single-slot stepping, a common approach that is effective if the hash function is good. Each of the **lookup**, **set** and **remove** functions use a common internal function **find_slot** to locate the array slot that either does

or should contain a given key.

```
record pair { key, value, occupied flag (initially unset) }
var pair slot[0], slot[1], ..., slot[num_slots - 1]
```

```
function find_slot(key)
    i := hash(key) modulo num_slots
    // search until we either find the key, or find an empty slot.
    while (slot[i] is occupied) and (slot[i].key ≠ key)
        i := (i + 1) modulo num_slots
    return i
```

```
function lookup(key)
    i := find_slot(key)
    if slot[i] is occupied   // key is in table
        return slot[i].value
    else                     // key is not in table
        return not found
```

```
function set(key, value)
    i := find_slot(key)
    if slot[i] is occupied   // we found our key
        slot[i].value = value
        return
    if the table is almost full
        rebuild the table larger (note 1)
        i := find_slot(key)
    mark slot[i] as occupied
    slot[i].key = key
    slot[i].value = value
```

note 1

Rebuilding the table requires allocating a larger array and recursively using the **set** operation to insert all the elements of the old array into the new larger array. It is common to increase the array size exponentially, for example by doubling the old array size.

```
function remove(key)
    i := find_slot(key)
    if slot[i] is unoccupied
        return   // key is not in the table
    mark slot[i] as unoccupied
    j := i
    loop (note 2)
        j := (j + 1) modulo num_slots
        if slot[j] is unoccupied
            exit loop
        k := hash(slot[j].key) modulo num_slots
        // determine if k lies cyclically in (i,j]
        // i ≤ j: |    i..k..j    |
        // i > j: |.k..j     i....| or |....j     i..k.|
        if i ≤ j
            if (i < k) and (k ≤ j)
                continue loop
        else
            if (i < k) or (k ≤ j)
                continue loop
        slot[i] := slot[j]
        mark slot[j] as unoccupied
        i := j
```

note 2

For all records in a cluster, there must be no vacant slots between their natural hash position and their current position (else lookups will terminate before finding the record). At this point in the pseudocode, *i* is a vacant slot that might be invalidating this property for subsequent records in the cluster. *j* is such a subsequent record. *k* is the raw hash where the record at *j* would naturally land in the hash table if there were no collisions. This test is

asking if the record at *j* is invalidly positioned with respect to the required properties of a cluster now that *i* is vacant.

Another technique for removal is simply to mark the slot as deleted. However this eventually requires rebuilding the table simply to remove deleted records. The methods above provide *O*(1) updating and removal of existing records, with occasional rebuilding if the high-water mark of the table size grows.

The *O*(1) remove method above is only possible in linearly probed hash tables with single-slot stepping. In the case where many records are to be deleted in one operation, marking the slots for deletion and later rebuilding may be more efficient.

See also

- Lazy deletion – a method of deleting from a hash table using open addressing.

References

1. Tenenbaum, Aaron M.; Langsam, Yedidyah; Augenstein, Moshe J. (1990), *Data Structures Using C*, Prentice Hall, pp. 456–461, pp. 472, ISBN 0-13-199746-7
2. Poblete; Viola; Munro. "The Analysis of a Hashing Scheme by the Diagonal Poisson Transform". p. 95 of Jan van Leeuwen (Ed.) "Algorithms - ESA '94" (https://books.google.com/books?id=2aCoW8m40AwC). 1994.
3. Steve Heller. "Efficient C/C++ Programming: Smaller, Faster, Better" (https://books.google.com/books?id=gaajBQAAQBAJ) 2014. p. 33.
4. Patricio V. Poblete, Alfredo Viola. "Robin Hood Hashing really has constant average search cost and variance in full tables" (https://arxiv.org/abs/1605.04031). 2016.
5. Paul E. Black, "Last-Come First-Served Hashing" (https://xlinux.nist.gov/dads/HTML/LastComeFirstServedHashing.html), in Dictionary of Algorithms and Data Structures [online], Vreda Pieterse and Paul E. Black, eds. 17 September 2015.
6. Paul E. Black, "Robin Hood hashing" (https://www.nist.gov/dads/HTML/robinHoodHashing.html), in Dictionary of Algorithms and Data Structures [online], Vreda Pieterse and Paul E. Black, eds. 17 September 2015.