
Hash Tables

Is balanced BST efficient enough?

- What drives the need for hash tables given the existence of balanced binary search trees?:
 - support relatively fast searches ($O(\log n)$), insertion and deletion
 - support range queries (i.e. return information about a range of records, e.g. “find the ages of all customers whose last name begins with 'S'”)
 - are dynamic (i.e. the number of records to be stored is not fixed)
 - But note the “*relatively fast searches*”. What if we want to make many single searches in a large amount of data? If a BST contains 1,000,000 items then each search requires around $\log_2 1,000,000 = 20$ comparisons.
 - If we had stored the data in an array and could (somehow) know the index (based on the value of the key) then each search would take constant ($O(1)$) time, a twenty-fold improvement.
-

Using arrays

- If the data have conveniently distributed keys that range from 0 to the some value N with no duplicates then we can use an array:
 - An item with key K is stored in the array in the cell with index K .
 - Perfect solution: searching, inserting, deleting in time $O(1)$
 - Drawback: N is usually huge (sometimes even not bounded) – so it requires a lot of memory.
 - Unfortunately this is often the case. Examples: we want to look people up by their phone numbers, or SIDs, or names.
 - Let's look at these examples.
-

Using arrays – Example 1: phone numbers as keys

- For phone numbers we can assume that the values range between 000-000-0000 and 999-999-9999 (in Canada). So let's see how big an array has to be to store all the possible numbers. It's easy to map phone numbers to integers (keys), just get rid of the '-'s. So we have a range from 0 to 9,999,999,999. So we'd need an array of size 10 billion. There are two problems here:
 - The first is that you won't fit the array in main memory. A PC with 2GB of RAM can store only 536,870,912 references (assuming each reference takes only 4 bytes) which is clearly insufficient. Plus we have to store actual data somewhere.
(We could store the array on the hard drive, but it would require 40GB.)
 - The other problem is that such an array would be horribly wasteful. The population of Canada estimated in July 2004 is 32,507,874, so if we assume that that's the approx. number of phone numbers, there is a huge amount of wasted space.
-

Using arrays – Example 2: names as keys

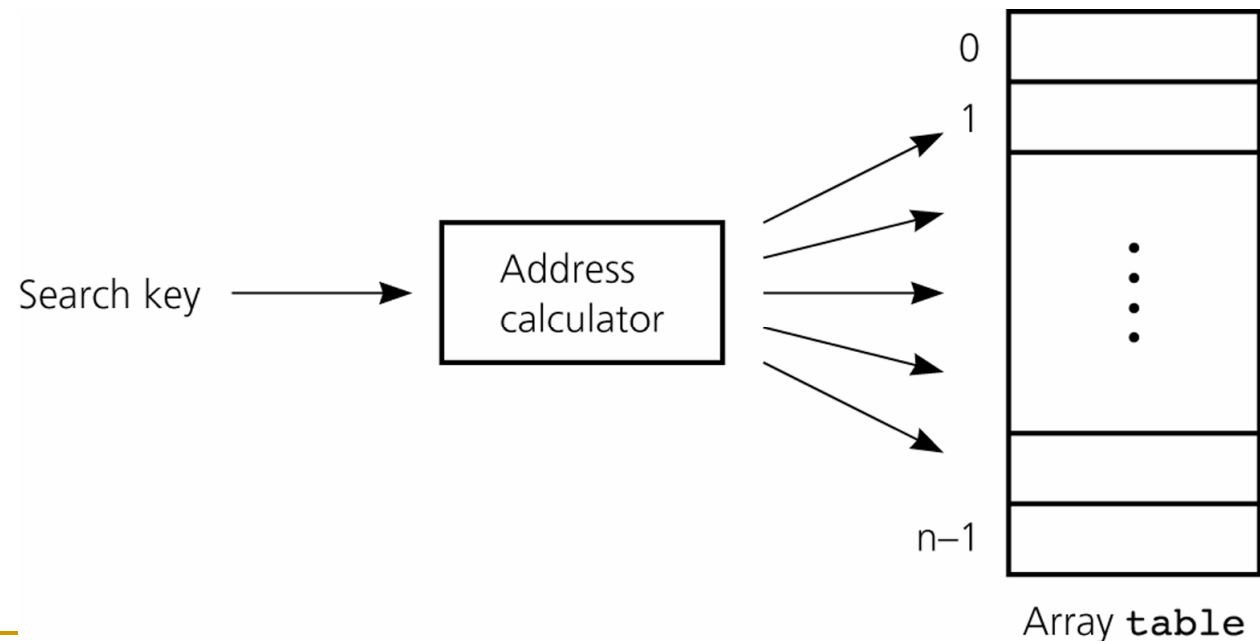
- How do we map strings to integers?
 - One way is to convert each letter to a number, either by mapping them to 0-25 or their ASCII characters or some other method and concatenating those numbers.
 - So how many possible arrangements of letters are there for names with a maximum of (say) 10 characters? The first letter can be one of 26 letters. Then for each of these 26 possible first letters there are 26 possible second letters (for a total of 26×26 or 26^2 arrangements). With ten letters there are 26^{10} possible strings, i.e. 141,167,095,653,376 possible keys!
-

Hash function: mapping key to index

So far this approach (of converting the key to an integer which is then used as an index to an array whose size is equal to the largest possible integer key) is not looking very promising. What would be more useful would be to pick the size of the array we were going to use (based on how many customers, or citizens, or items we think we want to keep track of) and then somehow map the keys to the array indices

(which would range from 0 to our array size-1).

This map is called a **hash function**.



Hash functions.

- Selecting digits
 - Simple to compute but generally do not evenly distribute the items. (we should really utilize the entire key)
 - You need to be careful about which digits you choose. Eg. If you choose the first three digits of SIN you would map all the people in the same region to one location in the array.
 - Folding.
 - Selecting digits and add them.
 - We can also group the digits, add them and then concatenate them to get a key.
 - Eg. 001364825 => 001 + 364 + 825 => 1190
 - Modular arithmetic.
 - Provides a simple and effective method.
 - $h(x) = x \bmod \text{table_size}$.
 - Choosing table_size as a prime number will increase the efficiency.
-

Hash Table

- A hash table consists of
 - an array to store data in and
 - a hash function to map a key an array index.
 - We can assume that the array will contain references to objects of some data structure. This data structure will contain a number of attributes, one of which must be a *key value* used as an index into the hash table. We'll assume that we can *convert the key to an integer* in some way. We can map that to an array index using the modulo (or remainder) function:
 - **Simple hash function:** $h(key) = key \% array_size$ where $h(key)$ is the hash value (array index) and $\%$ is the modulo operator.
 - **Example:** using a customer phone number as the key, assume that there are 500 customer records and that we store them in an array of size 1,000. A record with a phone number of 604-555-1987 would be mapped to array element 987 ($6,045,551,987 \% 1,000 = 987$).
-

How do we map string key to hash code?

■ How do we map strings to integers?

- ❑ Convert each letter to a number, either by mapping them to 0-25 or their ASCII characters or some other method.
 - ❑ Concatenating those values to one huge integer is not very efficient (or if the values are let to overflow, most likely we would just ignore the most of the string).
 - ❑ Summing the number doesn't work well either ('stop', 'tops', 'pots', 'spot')
 - ❑ Use polynomial hash codes:
$$x_0a^{k-1} + x_1a^{k-2} + \dots + x_{k-2}a + x_{k-1},$$

where a is a prime number (33, 37, 39, 41 works best for English words) [remark: and let it overflow]
-

A problem – collisions

- Let's assume that we make the array size (roughly) double the number of values to be stored in it.
 - This a common approach (as it can be shown that this size is minimal possible for which hashing will work efficiently).
 - We now have a way of mapping a numeric key to the range of array indices.
 - However, there is no guarantee that two records (with different keys) won't map to the same array element (consider the phone number 512-555-7987 in the previous example). When this happens it is termed a **collision**.
 - There are *two issues* to consider concerning collisions:
 - *how to minimize the chances of collisions occurring and*
 - *what to do about them when they do occur.*
-

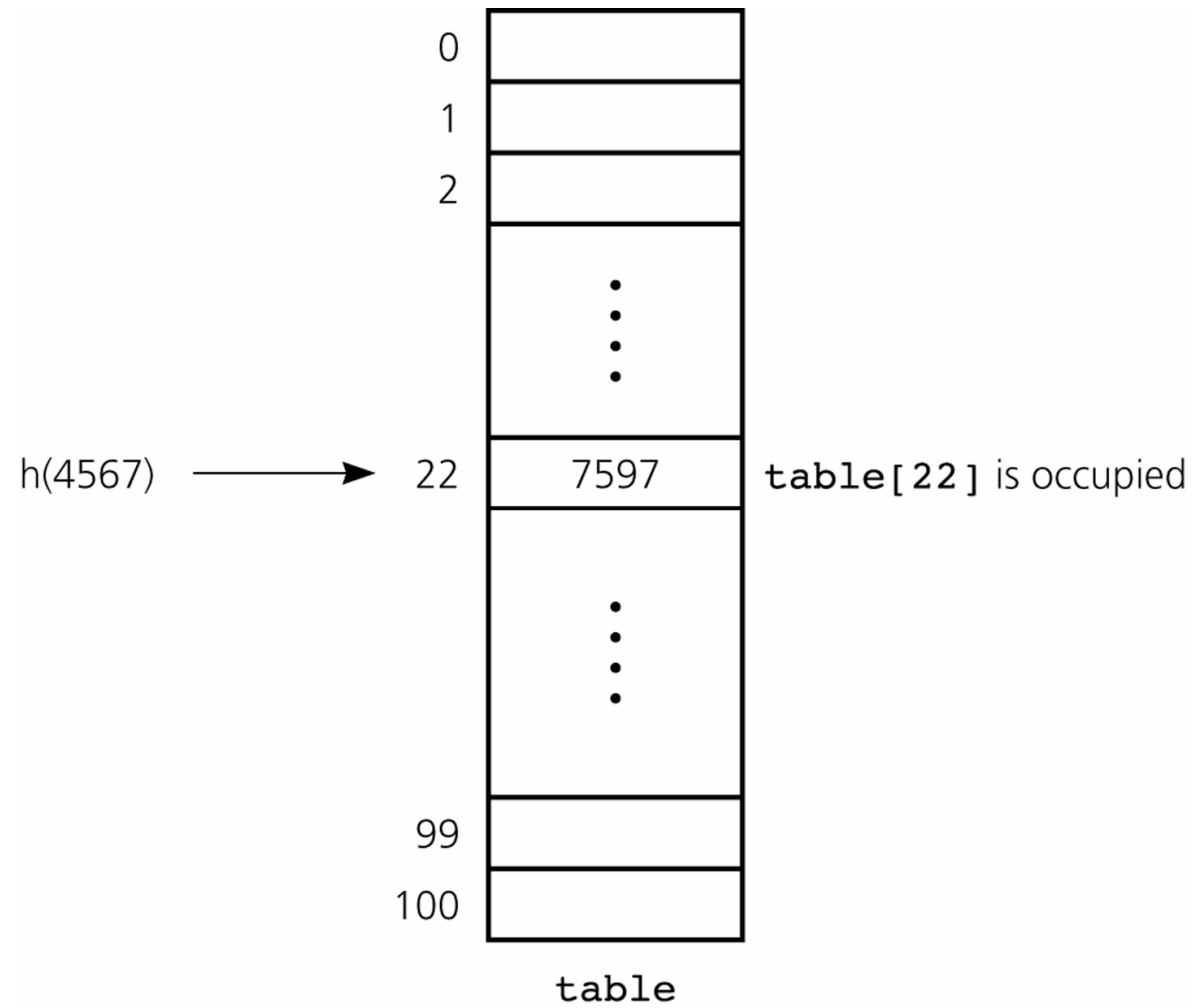


Figure: A collision

Minimizing Collisions by Determining a Good Hash Function

- A good hash function will *reduce the probability of collisions* occurring, while a bad hash function will increase it. Let's look at an example of a bad hash function first to illustrate some of the issues.
 - **Example:** Suppose I want to store a few hundred English words in a hash table. I create an array of 26^2 (676) and map the words based on the first two letters in the word. So, for example the word "earth" might map to index 104 ($e=4, a=0; 4*26 + 0 = 104$) and a word beginning with "zz" would map to index 675 ($z = 25*26 + 25 = 675$).
 - **Problem:** The flaw with this scheme is that the universe of possible English words is not uniformly distributed across the array. There are many more words beginning with "ea" or "th" than there are with "hh" or "zz". So this scheme would probably generate many collisions while some positions in the array would be never used.
 - Remember this is an example of a bad hash function!
-

A Good Hash Function

- First, it should be fast to compute.
 - A good hash function should result in each key being equally likely to hash to any of the array elements. Or other way round: each index in the array should have same probability to be mapped an item (considering the distribution of possible datas).
 - Well, the best function would be a random function, but that **doesn't work**: we would be not able to find an element once we store it in the table, i.e.
the function has to return the same index each time it is a called on the same key.
 - To achieve this it is usual to determine the hash value so that it is independent of any patterns that exist in the data. In the example above the hash value *is* dependent on patterns in the data, hence the problem.
-

A Good Hash Function

- Independent hash function:
 - Express the key as an integer (if it isn't already one), called **hash value** or **hash code**. When doing so remove any non-data (e.g. for a hash table of part numbers where all part numbers begin with 'P', there is don't to include the 'P' as part of the key), otherwise **base the integer on the entire key**.
 - Use a **prime** number as the size of the array (independent from any constants occurring in data).
 - There are other ways of computing hash functions, and much work has been done on this subject, which is beyond the scope of this course.
-

Hashing summary

- Determine the size m of the hash table's underlying array. The size should be:
 - approximately **twice the size** of the expected number of records and
 - a **prime number**, to evenly distribute items over the table.
 - Express the key as the integer such that it depends on the entire key.
 - Map the key to the hash table index by calculating the remainder of the key, k , divided by the size of the hash table m : $h(k) = k \bmod m$.
-

Dealing with collisions

- Even though we can reduce collisions by using a good hash function they will still occur.
 - There are two main approaches of dealing with collisions:
 - The first is to find somewhere else to insert an item that has collided (**open addressing**);
 - the second is to make the hash table an array of linked lists (**separate chaining**).
-

Open Addressing

- The idea behind open addressing is that when a *collision occurs* the new value is inserted in a *different index* in the array.
 - This has to be done in a way that allows the value to be found again.
 - We'll look at three separate versions of open addressing. In each of these versions, the “**step**” **value** is a distance from the original index calculated by the hash function.
 - The original index plus the step gives the new index to insert a record at if a collision occurs.
-

Open addressing: Linear Probing

- the simplest method
 - In linear probing the step increases **by one** each time an insertion fails to find space to insert a record:
 - So, when a record is inserted in the hash table, if the array element that it is mapped to is occupied we look at the next element. If that element is occupied we look at the next one, and so on.
 - **Disadvantage of this method:** sequences of occupied elements build up making the step values larger (and insertion less efficient); this problem is referred to as **primary clustering** (***"The rich gets richer"***).
 - Clustering tends to get worse as the hash table fills up (has many elements – more than $\frac{1}{2}$ full). This means that more comparisons (or probes) are required to look up items, or to insert and delete items, reducing the efficiency of the hash table.
-

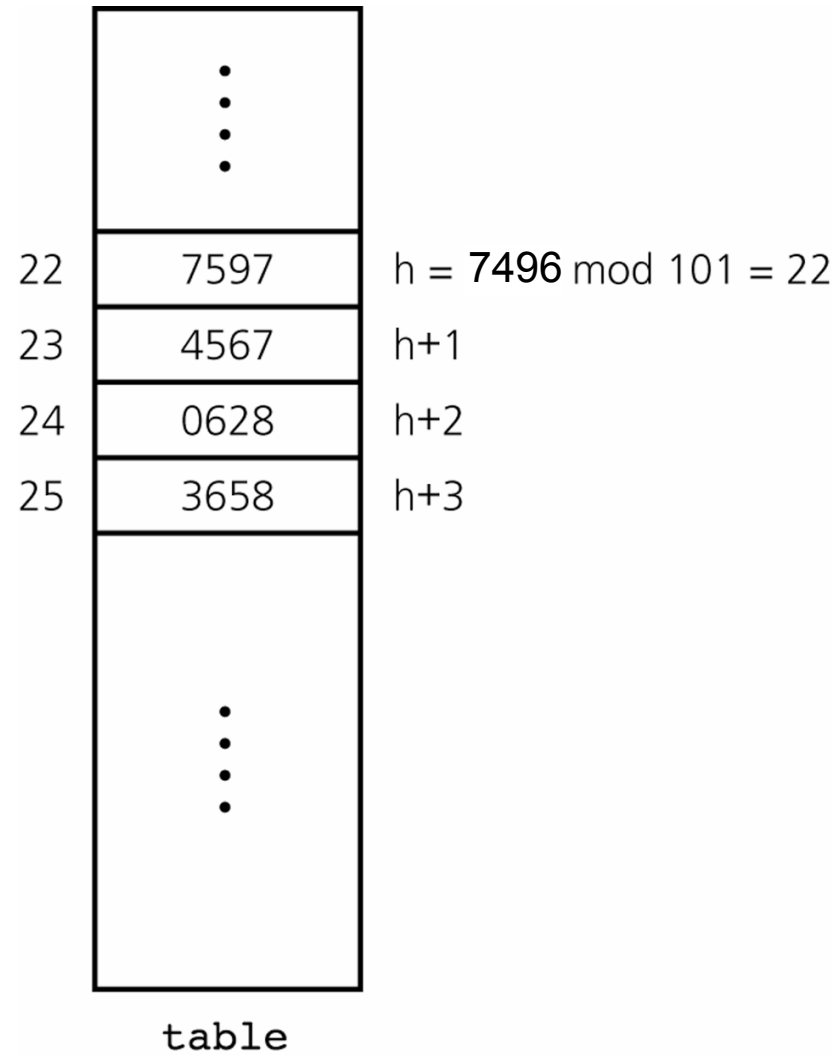


Figure: Linear probing with $h(x) = x \bmod 101$

Implementation

- **Insertion:** described on previous slides
 - **Searching:** it's not enough to look in the hash array at index where the key (hash code) was mapped, but we have to continue “probing” until we find either the element with the searched key or an empty spot (“not found”)
 - **Deleting:** We cannot just make a spot empty, as we could interrupt a probe sequence. Instead we mark it AVAILABLE, to indicate that the spot can be used for insertion, but searching should continue when AVAILABLE spot is encountered.
-

Implementation

■ Interface:

```
public interface HashTableInterface<T extends KeyedItem> {  
    public void insert(T item) throws HashTableFullException;  
    // PRE: item.getKey() != 0  
    public T find(long key);  
    // PRE: item.getKey() != 0  
    // return null if the item with key 'key' was not found  
    public T delete(long key);  
    // PRE: item.getKey() != 0  
    // return null if the item with key 'key' was not found  
}
```

Implementation

■ Data members and helping methods:

```
public class HashTable<T extends KeyedItem>
implements HashTableInterface<T>
{
    private KeyedItem[] table;
    // special values: null = EMPTY, T with key=0 = AVAILABLE
    private static KeyedItem AVAILABLE = new KeyedItem(0);

    private int h(long key) // hash function
    // return index
    { return (int)(key % table.length); } // typecast to int

    private int step(int k) // step function
    { return k; } // linear probing

    public HashTable(int size)
    { table = new KeyedItem[size]; }
```

Implementation

■ Insertion:

```
public void insert(T item) throws HashTableFullException
{
    int index = h(item.getKey());
    int probe = index;
    int k = 1; // probe number
    do {
        if (table[probe]==null || table[probe]==AVAILABLE) {
            // this slot is available
            table[probe] = item;
            return;
        }
        probe = (index + step(k)) % table.length; // check next slot
        k++;
    } while (probe!=index);
    throw new HashTableFullException("Hash table is full.");
}
```

Implementation

■ Helping method for locating index:

```
private int findIndex(long key)
// return -1 if the item with key 'key' was not found
{
    int index = h(key);
    int probe = index;
    int k = 1; // probe number
    do {
        if (table[probe]==null) {
            // probe sequence has ended
            break;
        }
        if (table[probe].getKey()==key)
            return probe;
        probe = (index + step(k)) % table.length; // check next slot
        k++;
    } while (probe!=index);

    return -1; // not found
}
```

Implementation

■ Find and Deleting the item:

```
public T find(long key)
{
    int index = findIndex(key);
    if (index >= 0)
        return (T) table[index];
    else
        return null; // not found
}

public T delete(long key)
{
    int index = findIndex(key);
    if (index >= 0) {
        T item = (T) table[index];
        table[index] = AVAILABLE; // mark available
        return item;
    } else
        return null; // not found
}
```

Open addressing: Quadratic Probing

- Designed to prevent *primary clustering*.
 - It does this by increasing the step by increasingly large amounts as more probes are required to insert a record. This prevents clusters from building up.
 - In quadratic probing the step is equal to the *square* of the probe number.
 - With linear probing the step values for a sequence of probes would be {1, 2, 3, 4, etc}. For quadratic probing the step values would be {1, 2², 3², 4², etc}, i.e. {1, 4, 9, 16, etc}.
 - **Disadvantage of this method:**
 - After a number of probes the sequence of steps repeats itself (remember that the step will be $\text{probe number}^2 \bmod \text{the size of the hash table}$). This repetition occurs when the probe number is roughly half the size of the hash table.
 - **Secondary clustering.**
-

Open addressing: Quadratic Probing

■ Disadvantage of this method:

- ❑ After a number of probes the **sequence** of steps **repeats** itself. => It fails to insert a new item even if there is still a space in the array.
- ❑ **Secondary clustering:** the sequence of probe steps is the same for any insertion. Secondary clustering refers to the increase in the probe length (that is the number of probes required to find a record) for records where *collisions* have occurred (the keys are mapped to the **same** value). Note that this is not as large a problem as *primary clustering*.

- However, it is important to realize that in practice these two issues are not significant, given a large hash table and a good hash function it is extremely unlikely that these issues will affect the performance of the hash table, unless it becomes nearly full.
-

	⋮	
22	7597	$h = 7597 \bmod 101 = 22$
23	4567	$h+1^2$
24		
25		
26	0628	$h+2^2$
	⋮	
31	3658	$h+3^2$
	⋮	

table

Figure: Quadratic probing with $h(x) = x \bmod 101$

Implementation

- It's enough to modify the step helping method:

```
private int step(int k) // step function
{
    return k*k // quadratic probing
}
```

Open addressing: Double Hashing

- Double hashing aims to avoid both *primary* and *secondary clustering* and is guaranteed to find a free element in a hash table as long as the table is not full. It achieves these goals by calculating the step value using a **second hash function h'** .

$$step(k) = k.h'(key)$$

- This new hash function h' should:
 - be different from the original hash function (remember that it was the original hash function that resulted in the collision in the first place) and,
 - not result in zero (as original index + 0 = original index)
-

Open addressing: Double Hashing

- The second hash function is usually chosen as follows:

$$h'(key) = q - (key \% q),$$

where q is a prime number $q < N$ (N is the size of the array).

- **Remark:** It is important that the size of the hash table is a **prime** number if double hashing is to be used. This guarantees that successive probes will (eventually) try every index in the hash table before an index is repeated (which would indicate that the hash table is full).
 - For other hashing's (and for q) we want to use prime numbers to eliminate existing patterns in the data.

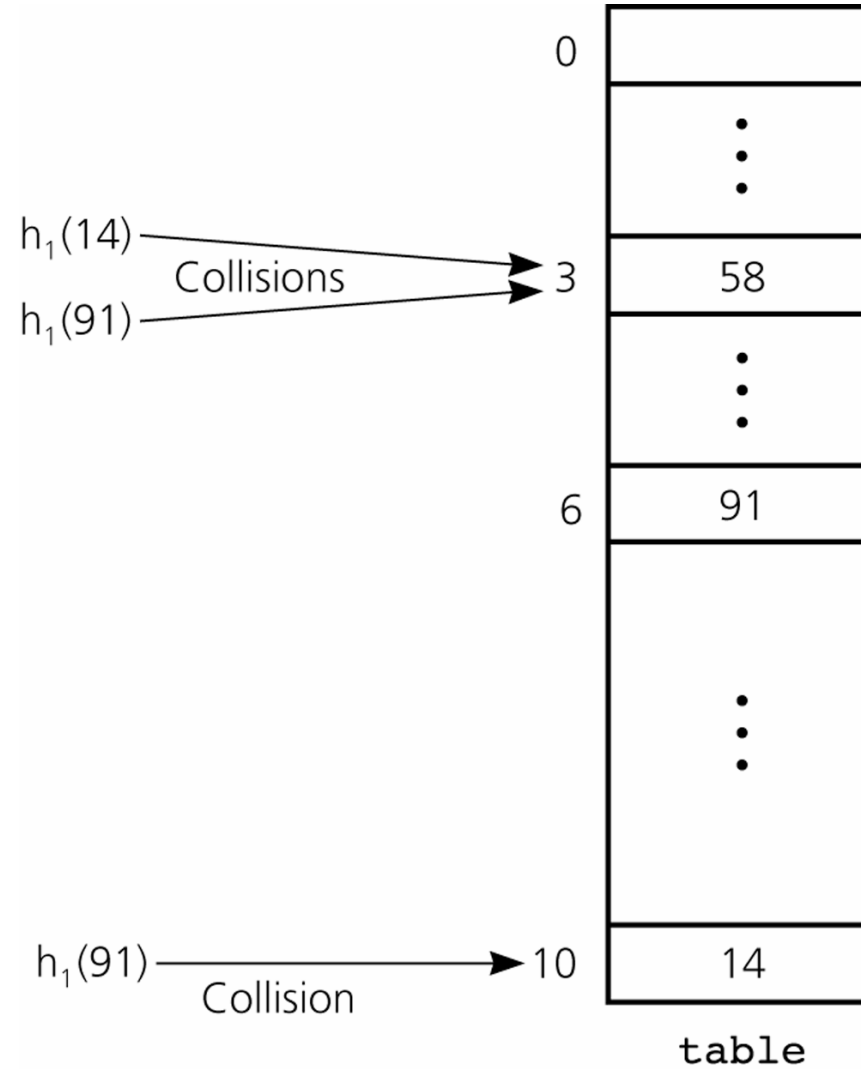


Figure: Double hashing during the insertion of 58, 14, and 91

Double Hashing – Implementation

```
public class DoubleHashTable<T extends KeyedItem>
implements HashTableInterface<T>
{
    private KeyedItem[] table;
    // special values: null = EMPTY, T with key=0 = AVAILABLE
    private static KeyedItem AVAILABLE = new KeyedItem(0);
    private int q; // should be a prime number

    public DoubleHashTable(int size, int q)
    // size: should be a prime number;
    //      recommended roughly twice bigger
    //      than the expected number of elements
    // q: recommended to be a prime number, should be smaller
    //    than size
    {
        table = new KeyedItem[size];
        this.q=q;
    }
}
```

Double Hashing – Implementation

```
private int h(long key) // hash function
// return index
{
    return (int) (key % table.length); // typecast to int
}

private int hh(long key) // second hash function
// return step multiplicative constant
{
    return (int) (q - key%q);
}

private int step(int k, long key) // step function
{
    return k*hh(key);
}
```

Double Hashing – Implementation

- Call `step(k, item.getKey())` instead of `step(k)`

```
public void insert(T item) throws HashTableFullException
{
    int index = h(item.getKey());
    int probe = index;
    int k = 1;
    do {
        if (table[probe]==null || table[probe]==AVAILABLE) {
            // this slot is available
            table[probe] = item;
            return;
        }
        probe = (index + step(k, item.getKey())) % table.length; // check
next slot
        k++;
    } while (probe!=index);
    throw new HashTableFullException("Hash table is full.");
}
```

Open addressing performance

- The performance of a hash table depends on the **load factor** of the table.
 - The **load factor** α is the *ratio of the number of data items to the size of the array*.
 - Linear Probing:
 - $\frac{1}{2}(1 + 1/(1 - \alpha))$ for successful search
 - $\frac{1}{2}(1 + 1/(1 - \alpha)^2)$ for unsuccessful search
 - Quadratic and double hashing
 - $-\log(1 - \alpha) / \alpha$ for successful search
 - $1/(1 - \alpha)$ for unsuccessful search
 - Of the three types of open addressing *double hashing* gives the **best** performance.
 - Overall, open addressing works very well up to load factors of around **0.5** (when 2 probes on average are required to find a record). For load factors greater than 0.6 performance declines dramatically.
-

Rehashing

- If the load factor goes over the safe limit, we should increase the size of the hash table (as for dynamic arrays). This process is called **rehashing**.
 - *Comments:*
 - we cannot just double the size of the table, as the size should be a prime number;
 - it will change the main hash function
 - it's not enough to just copy items
 - Rehashing will take time $O(N)$
-

Dealing with Collisions (2nd approach): Separate Chaining

- In separate chaining the hash table consists of an array of **lists**.
 - When a collision occurs the new record is added to the list.
 - Deletion is straightforward as the record can simply be removed from the list.
 - Finally, separate chaining is less sensitive to load factor and it is normal to aim for a load factor of around 1 (but it will work also for load factors over 1).
-

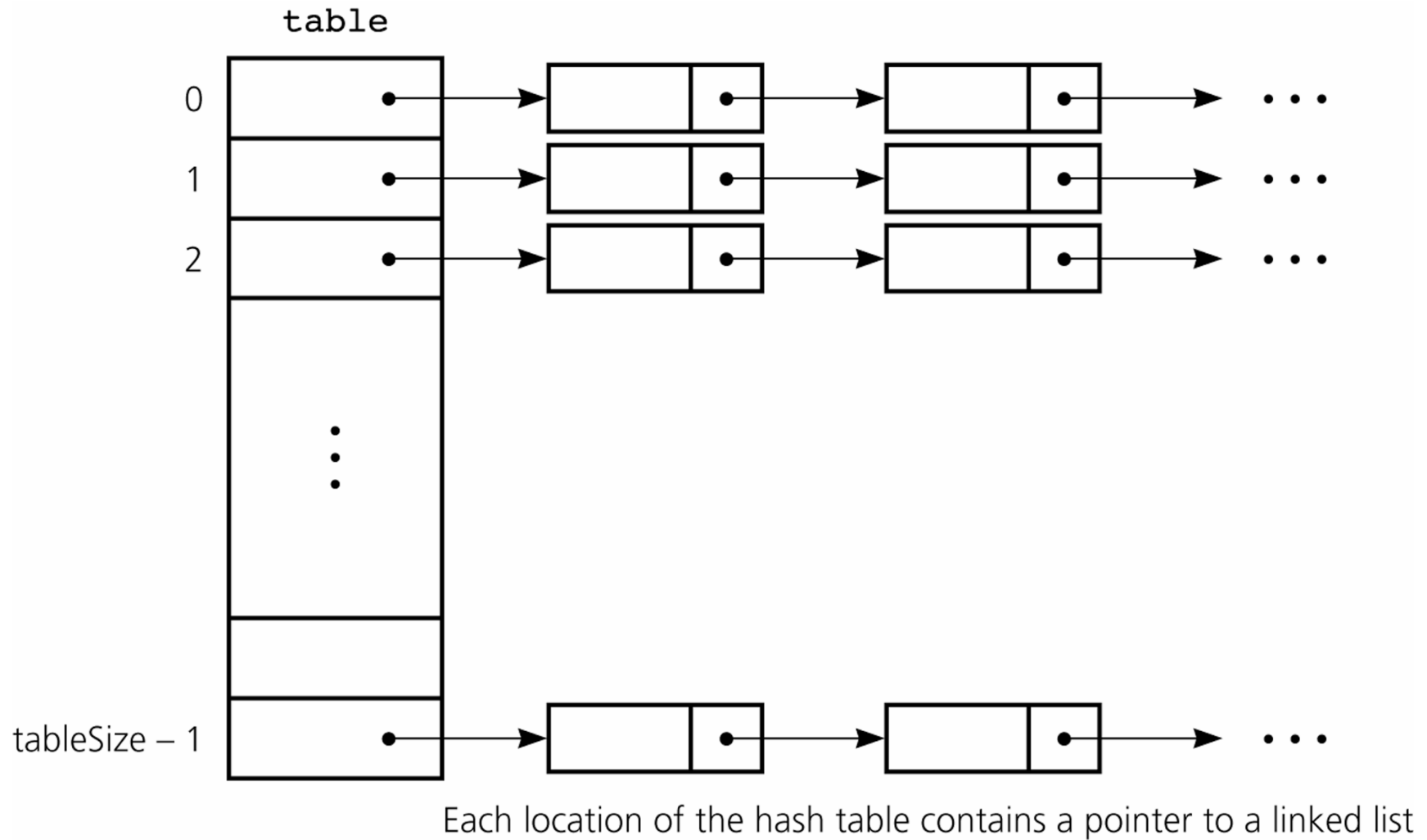


Figure: Separate chaining (using linked lists).

If array-based implementation of list is used: they are called **buckets**.

Implementation – data members

```
public class SHashTable<T extends KeyedItem>
implements HashTableInterface<T>
{
    private List<T>[] table;

    private int h(long key) // hash function
    // return index
    {
        return (int)(key % table.length); // typecast to int
    }

    public SHashTable(int size)
    // recommended size: prime number roughly twice bigger
    // than the expected number of elements
    {
        table = new List[size];
        // initialize the lists
        for (int i=0; i<size; i++)
            table[i] = new List<T>();
    }
}
```

Implementation – insertion

```
public void insert(T item)
{
    int index = h(item.getKey());
    List<T> L = table[index];
    // insert item to L
    L.add(1,item); // if linked list is used,
                  // insertion will be efficient
}
```

Implementation – search

```
private int findIndex(List<T> L, long key)
// search for item with key 'key' in L
// return -1 if the item with key 'key' was not found in L
{
    // search of item with key = 'key'
    for (int i=1; i<=L.size(); i++)
        if (L.get(i).getKey() == key)
            return i;

    return -1; // not found
}

public T find(long key)
{
    int index = h(key);
    List<T> L = table[index];
    int list_index = findIndex(L, key);
    if (index>=0)
        return L.get(list_index);
    else
        return null; // not found
}
```

Implementation – deletion

```
public T delete(long key)
{
    int index = h(key);
    List<T> L = table[index];
    int list_index = findIndex(L, key);
    if (index >= 0) {
        T item = L.get(list_index);
        L.remove(list_index);
        return item;
    } else
        return null; // not found
}
```

Hashing – comparison of different methods

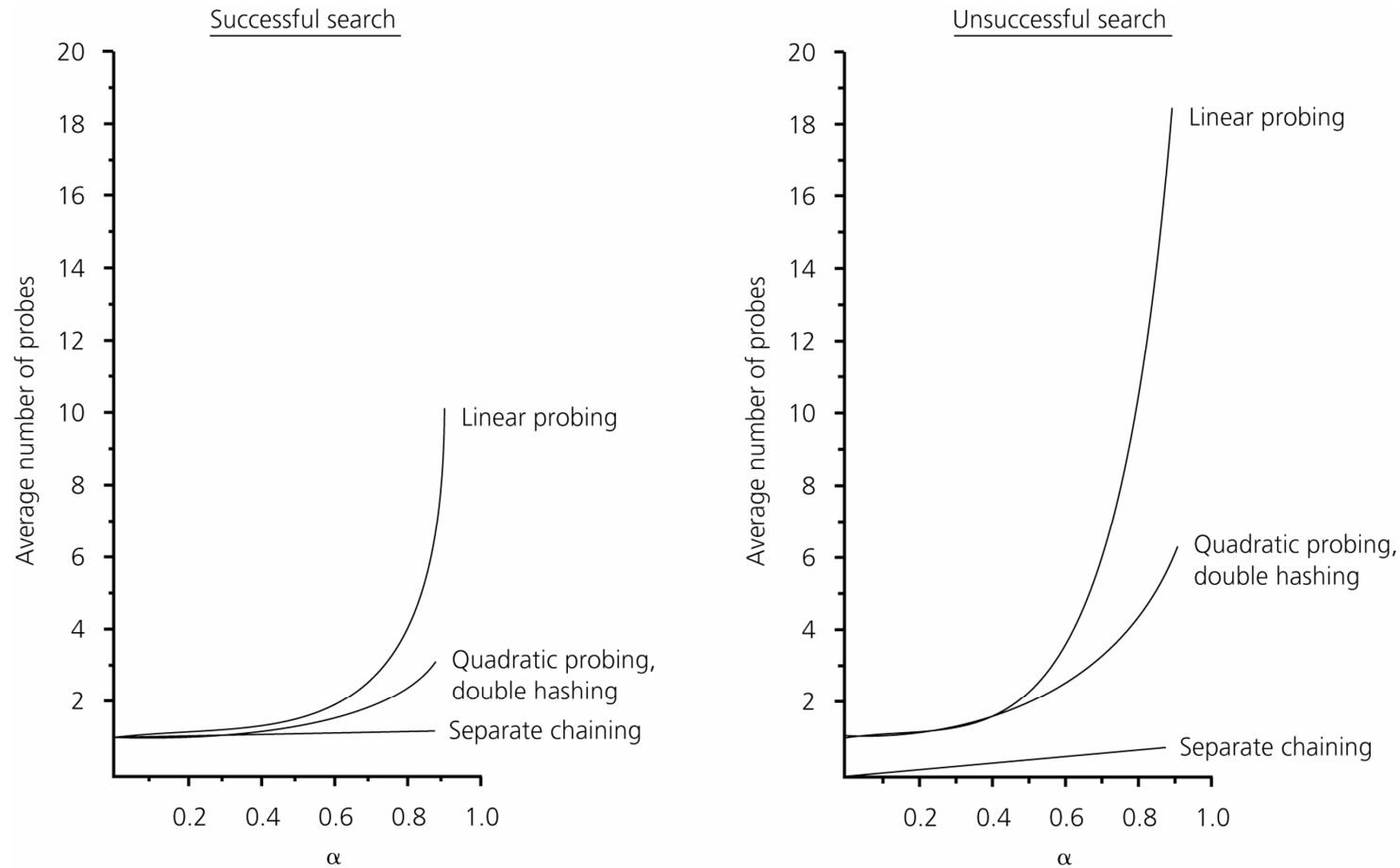


Figure: The relative efficiency of four collision-resolution methods

Comparing hash tables and balanced BSTs

- With good hash function and load kept low, hash tables perform insertions, deletions and search in $O(1)$ time on average, while balanced BSTs in $O(\log n)$ time.
 - However, there are some tasks (order related) for which, hash tables are not suitable:
 - traversing elements in sorted order: $O(N+n \cdot \log n)$ vs. $O(n)$
 - finding minimum or maximum element: $O(N)$ vs. $O(1)$
 - range query: finding elements with keys in an interval $[a,b]$: $O(N)$ vs. $O(\log n + s)$, s is the size of output
 - Depending on what kind of operations you will need to perform on the data and whether you need guaranteed performance on each query, you should choose which implementation to use.
-