# Lookup and Retrieval Data Structures

DANIEL ANDERSON [1]

Storing and retrieving data is one of the most common scenarios in which we employ advanced data structures. We will look at some advanced concepts related to two common lookup data structures that you should already be familar with, hashtables and binary search trees. We will also see some additional applications of string hashing related to pattern matching.

**Summary: Lookup and Retrieval Structures**

In this lecture, we cover:

**Hashing and hashtables:**

- Hashing for integer types
- Polynomial hash functions for strings
- The rolling polynomial hash
- The Rabin-Karp pattern matching algorithm
- Hashtables with cuckoo hashing

**Efficient binary search trees:**

- AVL trees

**Recommended Resources: Hashing and Hashtables**

- `http://users.monash.edu/~lloyd/tildeAlgDS/Table/`
- CLRS, Introduction to Algorithms, Chapter 11 and Section 32.2
- Weiss, Data Structures and Algorithm Analysis, Chapter 5
- Knuth, The Art of Computer Programming, Volume 3, Section 6.4

**Recommended Resources: Binary Search Trees**

- `http://users.monash.edu/~lloyd/tildeAlgDS/Tree/`
- CLRS, Introduction to Algorithms, Chapter 12
- Weiss, Data Structures and Algorithm Analysis, Chapter 4

## Hashing and Hashtables

Recall from your previous studies that a hash function is a function that takes a *key* from some specific domain and maps it onto some finite set of integer values, called the key's *hash value.* Hash values are commonly employed to implement *hashtables,* data structures which store elements associated with a given set of keys in a table at positions determined by their hash values. Hashtables are designed to provide average case $O(1)$ time lookup, insertion and deletion for a trade-off of a large amount of memory.

Since the range of a hash function should be finite, but the domain of keys might be infinite, two distinct keys may map to the same hash value. We call such an occurrence a *collision.* Dealing with collisions is the primary motivation for studying and seeking out good hash functions and good techniques for implementing hashtables.

[1] Faculty of Information Technology, Monash University: `daniel.anderson@monash.edu`. These notes are based on the lecture slides developed by Arun Konagurthu for FIT2004.

## What is a Good Hash Function?

Good hash functions are those which minimise the probability of collisions occurring. This is very easy to say, but very difficult, and sometimes impossible to achieve in practice. Assuming that the keys to be hashed are uniformly distributed implies that an ideal hash function maps keys uniformly onto the range of potential hash values. For example, to hash keys which are floating point values in the range $0 \leq k \leq 1$, a hash function that maps onto the integers 0 to $m - 1$ which satisfies this ideal is

$$h(k) = \lfloor km \rfloor$$

While this might be ideal under the assumption that the keys $k$ are uniformly distributed in $0 \leq k \leq 1$, this is a terrible hash function if it turns out that 99% of the keys $k$ turn out in practice to be very close together, as this hash function maps keys that are very close to the same hash value. In this scenario, an ideal hash function would be one that maps keys that are very close together to hash values that are very far apart.

What this really means is that unfortunately, it is not possible in general to design a hash function that is perfect for all situations. Designing robust and effective hash functions that minimise collisions must therefore almost always account for the nature of the keys that are expected and exploit their expected distribution to attempt to produce nicely distributed hash values.

## How likely are collisions?

Although collisions might seem rare, they are unintuitively common. One way to intuit this is to consider the famous *birthday problem*.

**Problem Statement: The Birthday Problem**

Given a group of $n$ people, find the probability that there is at least one pair of people with the same birthday.

Very unintuitively, the probably reaches 50% at just 23 people. At 367 people, the probability is 100% since there are only 366 possible birthdays. If we consider the analogy with respect to a hash table, we have a table of size 366, and with just 23 keys to insert, it is already more likely than not that a collision will occur. In general, with $n$ keys and a table of size $m$ where $n < m$, the probability that there exists a colliding pair is

$$p(n, m) = 1 - \frac{m!}{(m-n)!m^n}$$

which rapidly approaches 1 as $n \to m$. In practice, the best we can do is to minimise the probability of a collision by designing a good hash function. It is not realistic to try to avoid collisions 100% of the time, so the design of good hashtable data structures is therefore often heavily focused not only on avoiding collisions, but handling them in an efficient way when they do inevitably occur.

For the mathematically inclined, the rigorous analysis of hash collisions is commonly analysed using "balls in buckets" models from asymptotic combinatorics. Interested students should read *Probability and Computing*, Mitzenmacher & Upfal.

## Hashing for Integer Types

The common types of keys that we need to hash when implementing hashtables are integer types. Even when the types that we are dealing with might not be explicitly integers, there is often a simple and convenient way to interpret them as integers. For example, characters from a finite alphabet can be interpreted as integers corresponding to their position in the alphabet. A generalisation of this allows us to consider strings as integers in which we consider the characters as place values in some fixed radix system (covered in the next section.)

If the hashes are intended to be used by a hashtable, then the range of our hash functions should be the size of the table. For other applications, the range of the hash function should be chosen to minimise the chance of collisions. Let's look at a few examples of good hash functions for integer types.

### The divisional method

The first and simplest method for producing a hash function for a set of integral keys onto a table of $m$ slots is to simply take the value of the key mod $m$. That is, for each key $k$, the hash function is

$$h(k) = k \mod m.$$

This hash function produces hash values that are usually well distributed provided that $m$ is selected well. In particular, one should take care to avoid using powers of two, since taking $k \mod 2^p$ simply extracts the $p$ least significant bits of $k$. A good hash function should depend on the values of all of the bits in the key to increase the likelihood that the hashes are well distributed. Prime values of $m$, particularly those which are not close to a power of two are strong choices since some number theory will tell us that consecutive multiples of a key will continuously result in different hash values provided that the key is not a multiple of $m$ itself.

### The multiplicative method

The multiplicative hash works as follows. We select a constant $A$ where $A < 0 < 1$ and take the fractional part of $kA$, which is $kA - \lfloor kA \rfloor$. We then scale $m$ by the resulting value to produce a hash like so

$$h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor.$$

Unlike the divisional method, the value of $m$ does not heavily influence the quality of the hash, so taking $m$ as a power of two is fine and even preferable since the computation of the hash can then be sped up by taking advantage of bitwise operations.

The main decision to make is the value of $A$, whose optimal choice is heavily influenced by the characteristics of the keys being hashed. According to Knuth[2], the value

$$A = \varphi^{-1} = \frac{\sqrt{5}-1}{2} \approx 0.61803...$$

is a strong choice.

## Polynomial Hash Functions for Strings

The polynomial hashing technique is a common and effective method for hashing strings. Given a string `S[1..N]`, if we associate each character of `S` with a numeric value (its position in the alphabet for example), then the polynomial hash is given by

$$S[1]x^{N-1} + S[2]x^{N-2} + ... + S[N-1]x + S[N]$$

To reduce the chance of collisions, the value used for the base $x$ should be greater than the maximum attainable value of $S[i]$. If this is the case, then it is easy to see that the value of the polynomial hash for any string is unique, that is there are no collisions! This sounds great, but of course there are practical considerations that prevent this from actually being the case. The value of the hash function will quickly grow far too large to be usable, so we usually mod the value of the polynomial hash by a chosen value $m$ (if intended for use by a hashtable, this is the size of the table again). The polynomial hash then becomes

$$S[1]x^{N-1} + S[2]x^{N-2} + ... + S[N-1]x + S[N] \mod m.$$

If $m$ is chosen to be a large prime number, then the probability of collisions is usually low. Evaluating the polynomial hash for a given string naively term-by-term will lead to a time complexity of $O(N^2)$ or $O(N\log(N))$ if fast exponentiation is used, but this can be easily improved by using *Horner's Method.*

[2]Donald E. Knuth. Sorting and Searching, volume 3 of *The Art of Computer Programming.* Addison-Wesley, 1973. Second edition, 1998

**Definition: Horner's Method**

Given the polynomial

$$p(x) = a_0 + a_1 x + a_2 x^2 + ... + a_n x^n$$

we can evaluate it in $O(n)$ time by computing the sequence of values

$$\begin{cases} b_n = a_n \\ b_{n-1} = a_{n-1} + b_n x \\ \vdots \\ b_0 = a_0 + b_1 x \end{cases}$$

where $b_0$ is the value of $p(x)$.

Horner's Method works because we can notice that a polynomial

$$p(x) = a_0 + a_1 x + a_2 x^2 + ... + a_n x^n$$

can be rewritten as

$$p(x) = a_0 + x(a_1 + x(a_2 + ... + x(a_{n-1} + a_n x))).$$

Horner's method then simply evaluates this expression from the innermost bracketed part outwards.

## Rolling the Polynomial Hash Function

One of the brilliant properties of the polynomial hash function that makes it so useful is that if we know the hash value of a particular substring of a given string, then we can compute the hash value of an adjacent substring (one differing in just one character) in just $O(1)$ time. Suppose that we first pre-compute the value of $x^{N-1}$ and we know the polynomial hash for some substring `S[i..j]`. Then the polynomial hash for the substring `S[i..j+1]` is given by

$$\texttt{hash(S[i..j+1])} = (x \times \texttt{hash(S[i..j])} + S[j+1]) \mod m,$$

and the polynomial hash for the substring `S[i+1..j]` can be computed as

$$\texttt{hash(S[i+1..j])} = \texttt{hash(S[i..j])} - S[i]x^{N-1} \mod m$$

This allows us to "roll" the polynomial hash across a string and compute the hash values for all substrings of a fixed length in just linear time, rather than the quadratic time that would be required to do each substring separately.

## Application to Pattern Matching – The Rabin-Karp Algorithm

Hash functions are not only useful for implementing hashtables. A powerful application of the rolling polynomial hash is the Rabin-Karp algorithm for pattern matching.

**Problem Statement: Pattern Matching**

Given a text string `T[1..N]` and a pattern string `P[1..M]`, find all occurences of `P` as substrings of `T`. In other words, find all `i` such that `T[i..i+M-1] = P[1..M]`.

The Rabin-Karp algorithm utilises the fact that if a substring `T[i..i+M]` matches the pattern `P[1..M]` then it must have the same hash value and that substrings that do not match are likely (but not guaranteed) to have different hash values.

We can precompute the hash value of the pattern `P[1..M]` in $O(M)$ time and then roll the polynomial hash for substrings of length $M$ in `T[1..N]`, checking whether they are a match.

**Algorithm: Rabin-Karp Pattern Matching**

```
function RABIN_KARP(S[1..N], P[1..M])
    Set p_hash = P[1], t_hash = T[1]
    Set xn = 1
    for i = 2 to M do
        p_hash = p_hash * x + P[i]   mod $m$
        t_hash = t_hash * x + T[i]   mod $m$
        xn = xn * x   mod $m$
    end for
    for i = 1 to N - M do
        if p_hash == t_hash and P[1..M] = T[i..M+i-1] then
            Pattern found at position i
        end if
        if i $\leq$ N-M then
            t_hash = (t_hash - T[i] * xn) * x + T[i+M]   mod $m$
        end if
    end for
end function
```

The total time complexity of the Rabin-Karp algorithm is $O(N + M + kM)$ where $k$ is the number of matching hashes. In the worst case, where the hashes match at all positions in T, then the performance of the Rabin-Karp algorithm is $O((N - M)M)$, which is no better than the naive pattern matching algorithm.

In "realistic" text data, this is unlikely to happen. If the hashes of T are distributed uniformly, the the probability of any particular substring sharing the same hash as P is $1/m$. Therefore the expected number of hash matches is $n/m$, leading to an expected average complexity of

$$O\left(N + M + \frac{N}{m}M\right) = O(N + M)$$

provided that $m \geq \min(N, M)$.

Alternatively, if speed is extremely important and false positives are acceptable, we can remove the test on line 10 that verifies that the match is real and the algorithm attains a guaranteed worst-case complexity of $O(N + M)$. This is an example of a trade-off of correctness for time, rather than space for time or vice versa, which is common in probabilistic algorithms.

## Cuckoo Hashing for Hashtables

Cuckoo hashing is a hashing scheme for hashtables that guarantees worst case $O(1)$ lookup.

**Key Ideas: Cuckoo Hashing**

1. Maintain two hashtables $T_1$ and $T_2$
2. Use two hash functions $f(key)$ and $g(key)$
3. If collisions occur, move items to their alternate position in the other table

**To lookup an element $x$:**

1. Look at $T_1[f(x)]$, if found, return it
2. Look at $T_2[g(x)]$, if found return it
3. Otherwise, element not found

**Algorithm: Cuckoo Hashing Lookup**

```
function LOOKUP(x)
    return $T_1$[f(x)] = x or $T_2$[g(x)] = x
end function
```

**To insert an element $x$:**

1. Attempt to insert the key $x$ into $T_1$ at position $f(x)$
2. If $x$ collides with an existing key in $T_1$, then move $y = T_1[f(x)]$ into $T_2$ at position $g(y)$.
3. If moving $y$ into $T_2$ also caused a collision, take $x = T_2[g(y)]$ and attempt to insert it into $T_1$.
4. Continue replacing and reinserting until every item has found a location or until the process **cycles**
5. If a cycle occurs, resize the tables and start from scratch

**Algorithm: Cuckoo Hashing Insertion**

```
function INSERT(x)
    if not lookup(x) then
        for i = 1 to MAX_LOOPS do
            swap(x, $T_1$[f(x)])
            If x = empty then return
            swap(x, $T_2$[g(x)])
            If x = empty then return
        end for
        resize_table()
        insert(x)
    end if
end function
```

**To erase an element $x$:**

1. Look at $T_1[f(x)]$, if found, erase it
2. Look at $T_2[g(x)]$, if found, erase it
3. Otherwise, element not found

**Algorithm: Cuckoo Hashing Erasure**

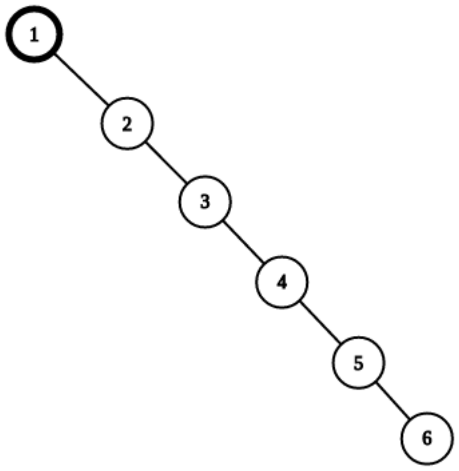
```
function ERASE(x)
    if $T_1$[f(x)] = x then
        $T_1$[f(x)] = empty
    end if
    if $T_2$[g(x)] = x then
        $T_2$[g(x)] = empty
    end if
end function
```

In practice, detecting a cycle (infinite loop) at the insertion step is tricky, so we usually set a threshold and decide that if the insertion process takes more than that many steps then we stop and start over. This threshold is shown as `MAX_LOOPS` in the pseudocode above. A common threshold to use is roughly $\log(C)$ where $C$ is the capacity of the hashtable. Lookup and deletion from a hashtable using Cuckoo hashing clearly has a worst-case $O(1)$ complexity since it suffices to simply check two addresses.

It can also be shown using some probabilistic analysis under some ideal assumptions about the distribution of the hashes that if the table remains under half capacity, that the probability that any given insertion causes a rehash is $O(1/n^2)$, and hence the probability that the rehash succeeds (does not cause a further rehash) is $1 - O(1/n)$. Finally, the expected number of iterations required to perform an insertion can be shown to be $O(1)$ and hence $n$ keys can be inserted in expected $O(n)$ time. Those who are interested in the analysis should read *Cuckoo Hashing, Pagh & Rodler* (`http://www.it-c.dk/people/pagh/papers/cuckoo-jour.pdf`)

## Binary Search Trees and AVL Trees

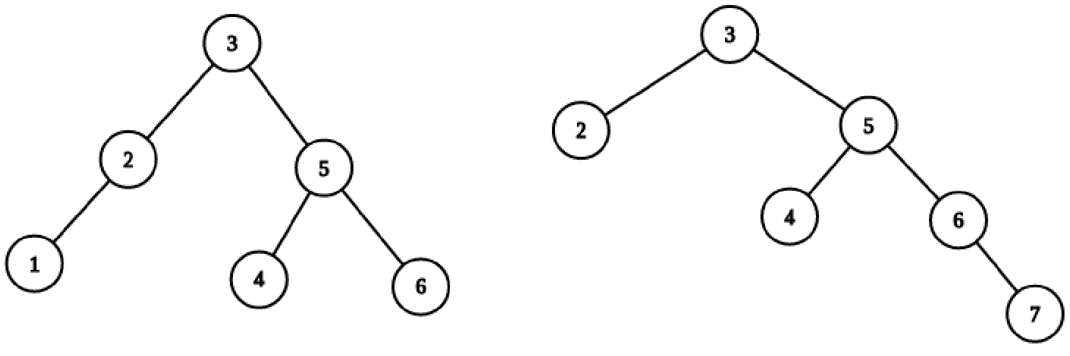
Standing side-by-side with hashtables in the lineup of lookup and retrieval data structures are binary search trees. Recall from your previous studies that a binary search tree is a rooted binary tree such that for each node, the keys in its left subtree compare less than its own and the keys in its right subtree compare greater than its own. Binary search trees are an efficient data structure for storing and retrieving elements of an ordered set in average $O(\log(N))$ time per operation. Pathological cases can occur however when the tree becomes imbalanced, where performance degrades in the worst case to linear time. We will study one variety of augmented binary search trees called an AVL[3] tree which self-adjusts to prevent imbalance and keeps all operations running in guaranteed worst case $O(\log(N))$ time.



An example of a very imbalanced binary search tree resulting from inserting the keys $1, 2, 3, 4, 5, 6$ in sorted order. Performance in such a tree is equivalent to a linked list with worst-case $O(N)$ lookup and insertion.

### When is a tree balanced?

A tree is considered to be balanced if for any node in the tree, the heights of their left and right subtrees differ by at most one. If a tree is balanced, its height is worst-case $O(\log(N))$. Since lookup, insertion and deletion all take time proportional to the height of the tree, the tree being balanced implies that lookup, insertion and deletion can all be performed in worst-case $O(\log(N))$ time.



(a) Balanced

(b) Imbalanced

A balanced binary search tree (a) and an imbalanced binary search tree (b). (b) is imbalanced because the heights of the root node's left and right children are 1 and 3 respectively.

[3]AVL trees are named after Adelson, Velskii, and Landis, the original inventors of the data structure.

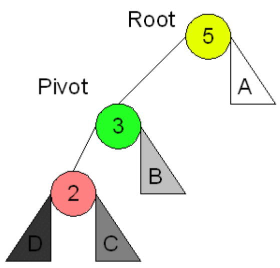
## Rebalancing

AVL trees maintain balance by performing *rotations* whenever an insertion or deletion operation violates the balance property. Rotations move some of the elements of the tree around in order to restore balance. We define the *balance factor* of a node to be the difference between the heights of its left and right subtrees, ie.

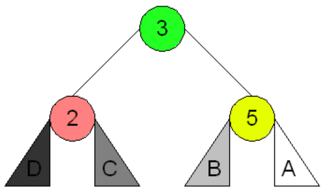
$$\texttt{balance\_factor(u)} = \texttt{height(u.left) - height(u.right)},$$

where `height(null) = 0`. A tree is therefore imbalanced if there is a node `u` such that $|\texttt{balance\_factor}(u)| \geq 2$. There are four separate cases that can occur when a tree is imbalanced.

### Left-left imbalance



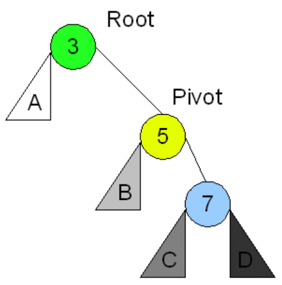
A left-left imbalance occurs when a node's balance factor is 2 (its left subtree is taller than its right subtree) and the left node's left subtree is at least as tall as the left node's right subtree (in other words, the balance factor of the left node is non-negative).
To remedy a left-left imbalance, we perform a rightwards rotation, in which the left node becomes the new root, and the children are shifted accordingly. (Image source: Wikimedia Commons)



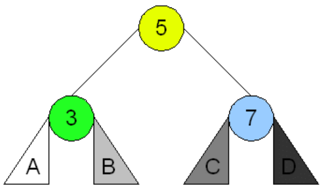
**Algorithm: Right Rotation**

```
1: function ROTATE_RIGHT(root)
2:     Set par = root.parent
3:     Set pivot = root.left
4:     Set temp = pivot.right
5:     pivot.set_right_child(root)
6:     root.set_left_child(temp)
7:     par.swap_child(root, pivot)
8: end function
```

### Right-right imbalance



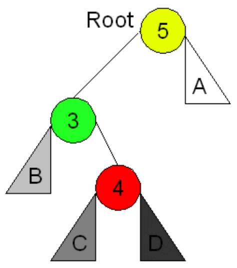
A right-right imbalance occurs when a node's balance factor is $-2$ (its right subtree is taller than its left subtree) and the right node's right subtree is at least as tall as the right node's left subtree (in other words, the balance factor of the right node is non-positive).
To remedy a right-right imbalance, we perform a leftwards rotation, in which the right node becomes the new root, and the children are shifted accordingly. (Image source: Wikimedia Commons)



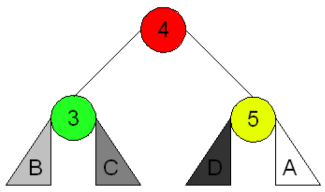
**Algorithm: Left Rotation**

```
1: function ROTATE_LEFT(root)
2:     Set par = root.parent
3:     Set pivot = root.right
4:     Set temp = pivot.left
5:     pivot.set_left_child(root)
6:     root.set_right_child(temp)
7:     par.swap_child(root, pivot)
8: end function
```

### Left-right imbalance



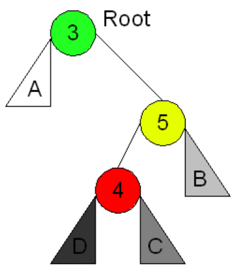
A left-right imbalance occurs when a node's balance factor is 2 (its left subtree is taller than its right subtree) and the left node's right subtree is taller than the left node's left subtree (in other words, the balance factor of the left node is negative).
To remedy a left-right imbalance, we first perform a leftward rotation on the left node, which converts the imbalance into the left-left situation. We then perform a rightward rotation on the root to balance it. (Image source: Wikimedia Commons)



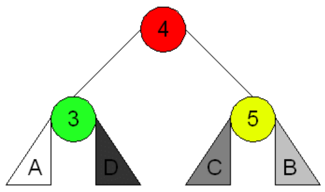
**Algorithm: Double-Right Rotation**

```
1: function DOUBLE_ROTATE_RIGHT(root)
2:     rotate_left(root.left)
3:     rotate_right(root)
4: end function
```

### Right-left imbalance



A right-left imbalance occurs when a node's balance factor is $-2$ (its right subtree is taller than its left subtree) and the right node's left subtree is taller than the right node's right subtree (in other words, the balance factor of the right node is positive).
To remedy a right-left imbalance, we first perform a rightward rotation on the right node, which converts the imbalance into the right-right situation. We then perform a leftward rotation on the root node to balance it. (Image source: Wikimedia Commons)



**Algorithm: Double-Left Rotation**

```
1: function DOUBLE_ROTATE_LEFT(root)
2:     rotate_right(root.right)
3:     rotate_left(root)
4: end function
```

Note that in the above algorithms, it is crucial that the functions for manipulating the children also correctly update the node's parent pointers! Failing to do so is the most common bug that programmers encounter when attempting to implement self-balancing binary search trees. We should also make sure that we handle the `null` cases correctly when the children or the parent nodes are `null`. In implementations of self-balancing binary search trees, it is common to use special dummy nodes to represent `null` nodes rather than using the language's actual null pointer to avoid having to include special cases all throughout the code.

Combining each of these together, the entire rebalancing procedure can be summarised like so.

**Algorithm: Rebalance**

```
function REBALANCE(node)
    if balance_factor(node) = 2 then
        if balance_factor(node.left) $<$ 0 then
            double_rotate_right(node)
        else
            rotate_right(node)
        end if
    else if balance_factor(node) = -2 then
        if balance_factor(node.right) $>$ 0 then
            double_rotate_left(node)
        else
            rotate_left(node)
        end if
    end if
end function
```

We should call the `rebalance` procedure whenever we insert or delete a node on all node's whose height was affected by the modification. Since the tree is assumed to be balanced before we modify it, the heights of at most $O(\log(N))$ nodes are affected by any one modification operation and hence we require at most $O(\log(N))$ rebalances, each of which can be executed in constant time. Therefore the total time complexity of insertion and deletion for an AVL tree is worst-case $O(\log(N))$. Since the tree is kept balanced by the rebalances, lookup in the tree is also worst-case $O(\log(N))$.