How does java Hashmap work internally

***What is Hashing?***  
Hashing in its simplest form, is a way to assigning a unique code for any variable/object after applying any formula/algorithm on its properties. A true Hashing function must follow this rule:  
  
Hash function should return the same hash code each and every time, when function is applied on same or equal objects. In other words, two equal objects must produce same hash code consistently.  
  
***Note:*** All objects in java inherit a default implementation of **hashCode()** function defined in Object class. This function produce hash code by typically converting the internal address of the object into an integer, thus producing different hash codes for all different objects.  
  
**HashMap is an array of Entry objects**  
Consider HashMap as just an array of objects.  
  
Have a look what this Object is:  
static class Entry<K,V> implements Map.Entry<K,V> {

final K key;

V value;

Entry<K,V> next;

final int hash;

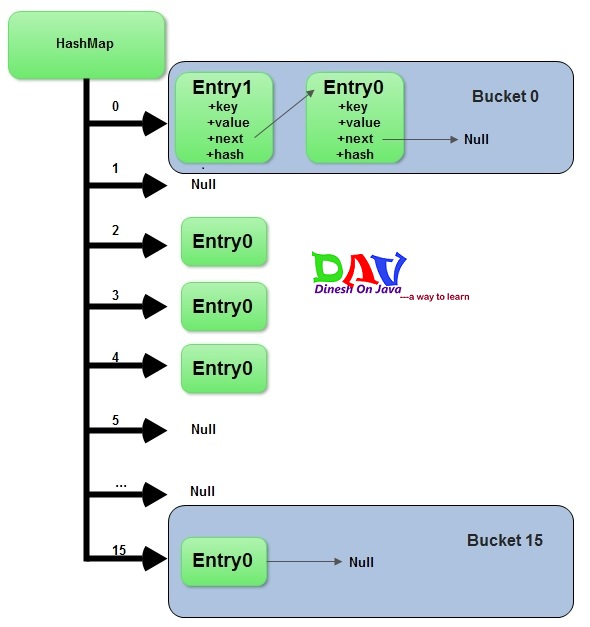
...

}

Each Entry object represents key-value pair. Field next refers to other Entry object if a bucket has more than 1 Entry.  
  
Sometimes it might happen that ***hashCodes***for 2 different objects are the same. In this case 2 objects will be saved in one bucket and will be presented as ***LinkedList***. The entry point is more recently added object. This object refers to other object with next field and so one. Last entry refers to null.  
When you create ***HashMap***with default constructor

HashMap hashMap = **new** HashMap();

Array is gets created with size 16 and default 0.75 load balance.



**Adding a new key-value pair**

1. Calculate ***hashcode***for the key
2. Calculate position hash % (arrayLength-1)) where element should be placed(bucket number)
3. If you try to add a value with a key which has already been saved in ***HashMap***, then value gets overwritten.
4. Otherwise element is added to the bucket. If bucket has already at least one element - a new one is gets added and placed in the first position in the bucket. Its next field refers to the old element.

**Deletion:**

1. Calculate ***hashcode***for the given key
2. Calculate bucket number (hash % (arrayLength-1))
3. Get a reference to the first Entry object in the bucket and by means of equals method iterate over all entries in the given bucket. Eventually we will find correct Entry. If desired element is not found - return null

**What *put()* method actually does:**  
Before going into **put()** method’s implementation, it is very important to learn that instances of ***Entry***class are stored in an array.***HashMap***class defines this variable as:

**transient** Entry[] table;

public V put(K key, V value) {

if (key == null)

return putForNullKey(value);

int hash = hash(key.hashCode());

int i = indexFor(hash, table.length);

for (Entry<k , V> e = table[i]; e != null; e = e.next) {

Object k;

if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {

V oldValue = e.value;

e.value = value;

e.recordAccess(this);

return oldValue;

}

}

modCount++;

addEntry(hash, key, value, i);

return null;

}

**Lets note down the steps one by one:**  
  
**Step1-** First of all, key object is checked for null. If key is null, value is stored in table[0] position. Because hash code for null is always 0.  
  
**Step2**- Then on next step, a hash value is calculated using key’s hash code by calling its **hashCode()** method. This hash value is used to calculate index in array for storing Entry object. JDK designers well assumed that there might be some poorly written***hashCode()*** functions that can return very high or low hash code value. To solve this issue, they introduced another hash() function, and passed the object’s hash code to this hash() function to bring hash value in range of array index size.  
  
**Step3-**Now***indexFor(hash, table.length)*** function is called to calculate exact index position for storing the Entry object.  
  
**Step4**- Here comes the main part. Now, as we know that two unequal objects can have same hash code value, how two different objects will be stored in same array location [called bucket].  
  
Answer is **LinkedList**. If you remember, Entry class had an attribute “next”. This attribute always points to next object in chain. This is exactly the behavior of LinkedList.  
  
So, in case of collision, Entry objects are stored in LinkedList form. When an Entry object needs to be stored in particular index, HashMap checks whether there is already an entry?? If there is no entry already present, Entry object is stored in this location.  
  
If there is already an object sitting on calculated index, its next attribute is checked. If it is null, and current Entry object becomes next node in LinkedList. If next variable is not null, procedure is followed until next is evaluated as null.  
  
What if we add the another value object with same key as entered before. Logically, it should replace the old value. How it is done? Well, after determining the index position of Entry object, while iterating over LinkedList on calculated index, HashMap calls equals method on key object for each Entry object. All these Entry objects in LinkedList will have similar hash code but equals() method will test for true equality. If key.equals(k) will be true then both keys are treated as same key object. This will cause the replacing of value object inside Entry object only.  
  
In this way, HashMap ensure the uniqueness of keys.  
  
**How *get()* methods works internally**  
Now we have got the idea, how key-value pairs are stored in HashMap. Next big question is : what happens when an object is passed in get method of HashMap? How the value object is determined?  
  
Answer we already should know that the way key uniqueness is determined in put() method , same logic is applied in get() method also. The moment HashMap identify exact match for the key object passed as argument, it simply returns the value object stored in current Entry object.  
  
If no match is found, get() method returns null.  
  
Let have a look at code:

public V get(Object key) {

if (key == null)

return getForNullKey();

int hash = hash(key.hashCode());

for (Entry<k , V> e = table[indexFor(hash, table.length)]; e != null; e = e.next) {

Object k;

if (e.hash == hash && ((k = e.key) == key || key.equals(k)))

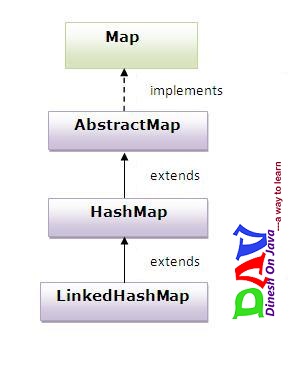
return e.value;

}

return null;

}

**Hierarchy of *LinkedHashMap*class:**



**Methods of HashMap**

static int hash(int h) {

// This function ensures that hashCodes that differ only by

// constant multiples at each bit position have a bounded

// number of collisions (approximately 8 at default load factor).

**h ^= (h >>> 20) ^ (h >>> 12);**

**return h ^ (h >>> 7) ^ (h >>> 4);**

}

/\*\*

\* Returns index for hash code h.

\*/

static int indexFor(int h, int length) {

**return h & (length-1);**

}

As the HashMap is always a power of 2 in size you can use as **hash = (null != key) ? hash(key) : 0;**

**bucketIndex = indexFor(hash, table.length);**

**public** V put(K key, V value) {

**if** (table == *EMPTY\_TABLE*) {

inflateTable(threshold);

}

**if** (key == **null**)

**return** putForNullKey(value);

**int** hash = hash(key);

**int** i = *indexFor*(hash, table.length);

**for** (Entry<K,V> e = table[i]; e != **null**; e = e.next) {

Object k;

**if** (e.hash == hash && ((k = e.key) == key || **key.equals(k)))** {

V oldValue = e.value;

e.value = value;

e.recordAccess(**this**);

**return** oldValue;

}

}

modCount++;

addEntry(hash, key, value, i);

**return** **null**;

}

**private** V putForNullKey(V value) {

**for** (Entry<K,V> e = table[0]; e != **null**; e = e.next) {

**if** (e.key == **null**) {

V oldValue = e.value;

e.value = value;

e.recordAccess(**this**);

**return** oldValue;

}

}

modCount++;

addEntry(0, **null**, value, 0);

**return** **null**;

}

**void** resize(**int** newCapacity) {

Entry[] oldTable = table;

**int** oldCapacity = oldTable.length;

**if** (oldCapacity == *MAXIMUM\_CAPACITY*) {

threshold = Integer.*MAX\_VALUE*;

**return**;

}

Entry[] newTable = **new** Entry[newCapacity];

transfer(newTable, initHashSeedAsNeeded(newCapacity));

table = newTable;

threshold = (**int**)Math.*min*(newCapacity \* loadFactor, *MAXIMUM\_CAPACITY* + 1);

}

**public** V get(Object key) {

**if** (key == **null**)

**return** getForNullKey();

Entry<K,V> entry = getEntry(key);

**return** **null** == entry ? **null** : entry.getValue();

}

**private** V getForNullKey() {

**if** (size == 0) {

**return** **null**;

}

**for** (Entry<K,V> e = table[0]; e != **null**; e = e.next) {

**if** (e.key == **null**)

**return** e.value;

}

**return** **null**;

}

**final** Entry<K,V> getEntry(Object key) {

**if** (size == 0) {

**return** **null**;

}

**int** hash = (key == **null**) ? 0 : hash(key);

**for** (Entry<K,V> e = table[*indexFor*(hash, table.length)];

e != **null**;

e = e.next) {

Object k;

**if** (e.hash == hash &&

((k = e.key) == key || (key != **null** && key.equals(k))))

**return** e;

}

**return** **null**;

}

/\*\*

\* The default initial capacity - MUST be a power of two.

\*/

**static** **final** **int** *DEFAULT\_INITIAL\_CAPACITY* = 16;

/\*\*

\* The maximum capacity, used if a higher value is implicitly specified

\* by either of the constructors with arguments.

\* MUST be a power of two <= 1<<30.

\*/

**static** **final** **int** *MAXIMUM\_CAPACITY* = 1 << 30;

hashcode method is used at the time of put and get, equals method is used at the time of get for collision and while inserting to check for duplicate.

**How TreeMap Works**

Treemap main advantage is that it allows to store the key-value mappings in a sorted order. Treemap internally uses red black tree.

From the javadocs:

A Red-Black tree based NavigableMap implementation. The map is sorted according to the natural ordering of its keys, or by a Comparator provided at map creation time, depending on which constructor is used.

This implementation provides guaranteed log(n) time cost for the containsKey, get, put and remove operations. Algorithms are adaptations of those in Cormen, Leiserson, and Rivest's Introduction to Algorithms.

**Red-black** tree from Wiki:

A red–black tree is a type of self-balancing binary search tree, a data structure used in computer science. The self-balancing is provided by painting each node with one of two colors (these are typically called 'red' and 'black', hence the name of the trees) in such a way that the resulting painted tree satisfies certain properties that don't allow it to become significantly unbalanced. When the tree is modified, the new tree is subsequently rearranged and repainted to restore the coloring properties. The properties are designed in such a way that this rearranging and recoloring can be performed efficiently. The balancing of the tree is not perfect but it is good enough to allow it to guarantee searching in O(log n) time, where n is the total number of elements in the tree. The insertion, and deletion operations, along with the tree rearrangement and recoloring are also performed in O(log n)

Basic source code for treemap is given below.

final int compare(Object k1, Object k2) {

return comparator==null ? ((Comparable<? super K>)k1).compareTo((K)k2)

: comparator.compare((K)k1, (K)k2);

}

public V put(K key, V value) {

Entry<K,V> t = root;

if (t == null) {

compare(key, key); // type (and possibly null) check

root = new Entry<>(key, value, null);

size = 1;

modCount++;

return null;

}

int cmp;

Entry<K,V> parent;

// split comparator and comparable paths

Comparator<? super K> cpr = comparator;

if (cpr != null) {

do {

parent = t;

cmp = cpr.compare(key, t.key);

if (cmp < 0)

t = t.left;

else if (cmp > 0)

t = t.right;

else

return t.setValue(value);

} while (t != null);

}

else {

if (key == null)

throw new NullPointerException();

Comparable<? super K> k = (Comparable<? super K>) key;

do {

parent = t;

cmp = k.compareTo(t.key);

if (cmp < 0)

t = t.left;

else if (cmp > 0)

t = t.right;

else

return t.setValue(value);

} while (t != null);

}

Entry<K,V> e = new Entry<>(key, value, parent);

if (cmp < 0)

parent.left = e;

else

parent.right = e;

fixAfterInsertion(e);

size++;

modCount++;

return null;

}

public V get(Object key) {

Entry<K,V> p = getEntry(key);

return (p==null ? null : p.value);

}

final Entry<K,V> getEntry(Object key) {

// Offload comparator-based version for sake of performance

if (comparator != null)

return getEntryUsingComparator(key);

if (key == null)

throw new NullPointerException();

Comparable<? super K> k = (Comparable<? super K>) key;

Entry<K,V> p = root;

while (p != null) {

int cmp = k.compareTo(p.key);

if (cmp < 0)

p = p.left;

else if (cmp > 0)

p = p.right;

else

return p;

}

return null;

}

final Entry<K,V> getEntryUsingComparator(Object key) {

K k = (K) key;

Comparator<? super K> cpr = comparator;

if (cpr != null) {

Entry<K,V> p = root;

while (p != null) {

int cmp = cpr.compare(k, p.key);

if (cmp < 0)

p = p.left;

else if (cmp > 0)

p = p.right;

else

return p;

}

}

return null;

}

**Quick Sort**

Quicksort is a divide and conquer algorithm. Quicksort first divides a large list into two smaller sub-lists: the low elements and the high elements. Quicksort can then recursively sort the sub-lists.

The steps are:

Pick an element, called a pivot, from the list.

Reorder the list so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.

Recursively apply the above steps to the sub-list of elements with smaller values and separately the sub-list of elements with greater values.

**Variant**

There are four well known variants of quicksort:

**Balanced quicksort**: choose a pivot likely to represent the middle of the values to be sorted, and then follow the regular quicksort algorithm.

**External quicksort**: The same as regular quicksort except the pivot is replaced by a buffer. First, read the M/2 first and last elements into the buffer and sort them. Read the next element from the beginning or end to balance writing. If the next element is less than the least of the buffer, write it to available space at the beginning. If greater than the greatest, write it to the end. Otherwise write the greatest or least of the buffer, and put the next element in the buffer. Keep the maximum lower and minimum upper keys written to avoid resorting middle elements that are in order. When done, write the buffer. Recursively sort the smaller partition, and loop to sort the remaining partition. This is a kind of three-way quicksort in which the middle partition (buffer) represents a sorted subarray of elements that are approximately equal to the pivot.

**Three-way radix quicksort** (developed by Sedgewick and also known as multikey quicksort): is a combination of radix sort and quicksort. Pick an element from the array (the pivot) and consider the first character (key) of the string (multikey). Partition the remaining elements into three sets: those whose corresponding character is less than, equal to, and greater than the pivot's character. Recursively sort the "less than" and "greater than" partitions on the same character. Recursively sort the "equal to" partition by the next character (key). Given we sort using bytes or words of length W bits, the best case is O(KN) and the worst case O(2KN) or at least O(N2) as for standard quicksort, given for unique keys N<2K, and K is a hidden constant in all standard comparison sort algorithms including quicksort. This is a kind of three-way quicksort in which the middle partition represents a (trivially) sorted subarray of elements that are exactly equal to the pivot.

**Quick radix sort** (also developed by Powers as a o(K) parallel PRAM algorithm). This is again a combination of radix sort and quicksort but the quicksort left/right partition decision is made on successive bits of the key, and is thus O(KN) for N K-bit keys. Note that all comparison sort algorithms effectively assume an ideal K of O(logN) as if k is smaller we can sort in O(N) using a hash table or integer sorting, and if K >> logN but elements are unique within O(logN) bits, the remaining bits will not be looked at by either quicksort or quick radix sort, and otherwise all comparison sorting algorithms will also have the same overhead of looking through O(K) relatively useless bits but quick radix sort will avoid the worst case O(N2) behaviours of standard quicksort and quick radix sort, and will be faster even in the best case of those comparison algorithms under these conditions of uniqueprefix(K) >> logN. See Powers [13] for further discussion of the hidden overheads in comparison, radix and parallel sorting.

The most direct competitor of quicksort is heapsort. Heapsort's worst-case running time is always O(n log n). But, heapsort is assumed to be on average somewhat slower than standard in-place quicksort.

Quicksort's running time depends on the result of the partitioning routine - whether it's balanced or unbalanced. This is determined by the pivot element used for partitioning. If the result of the partition is unbalanced, quicksort can run as slowly as insertion sort; if it's balanced, the algorithm runs asymptotically as fast as merge sort. That is why picking the "best" pivot is a crucial design decision.

The Wrong Way: the popular way of choosing the pivot is to use the first element; this is acceptable only if the input is random, but if the input is presorted, or in the reverse order, then the first elements provides a bad, unbalanced, partition. All the elements go either into S[p...q-1] or S[q+1..r]. If the input is presorted and as the first element is chosen consistently throughout the recursive calls, quicksort has taken quadratic time to do nothing at all.

The Safe Way: the safe way to choose a pivot is to simply pick one randomly; it is unlikely that a random pivot would consistently provide us with a bad partition throughout the course of the sort.

Median-of-Three Way: best case partitioning would occur if PARTITION produces two subproblems of almost equal size - one of size [n/2] and the other of size [n/2]-1. In order to achieve this partition, the pivot would have to be the median of the entire input; unfortunately this is hard to calculate and would consume much of the time, slowing down the algorithm considerably. A decent estimate can be obtained by choosing three elements randomly and using the median of these three as the pivot.

**Complexity of Quicksort**

**Worst-case: O(N2)**

This happens when the pivot is the smallest (or the largest) element.   
Then one of the partitions is empty, and we repeat recursively the procedure for N-1 elements.

**Best-case O(N logN)** The best case is when the pivot is the median of the array,   
and then the left and the right part will have same size.

There are logN partitions, and to obtain each partitions we do N comparisons   
(and not more than N/2 swaps). Hence the complexity is O(NlogN)

**Average-case - O(N logN)**

**Advantages:**

One of the fastest algorithms on average. It does not need additional memory (the sorting takes place in the array - this is called in-place processing). Compare with mergesort: mergesort needs additional memory for merging.

**Disadvantages: The worst-case complexity is O(N2)**

Applications:

Commercial applications use Quicksort - generally it runs fast, no additional memory,   
this compensates for the rare occasions when it runs with O(N2)

**Never use in applications which require guaranteed response time:**

**Life-critical (medical monitoring, life support in aircraft and space craft)**

**Mission-critical (monitoring and control in industrial and research plants   
handling dangerous materials, control for aircraft, defense, etc)**

**unless you assume the worst-case response time.**

Comparison with mergesort:

mergesort guarantees O(NlogN) time, however it requires additional memory with size N. quicksort does not require additional memory, however the speed is not quaranteed usually mergesort is not used for main memory sorting, only for external memory sorting. So far, our best sorting algorithm has O(*n*log *n*) performance: can we do any better?

*In general,* the answer is no.

**Note**

**It typically depends on the data structures involved. Quick sort is typically the fastest, but it doesn't guarantee O(n\*log(n)); there are degenerate cases where it becomes O(n^2). Heap sort is the usual alternative; it guarantees O(n\*log(n)), regardless of the initial order, but it has a much higher constant factor. It's usually used when you need a hard upper limit to the time taken. Some more recent algorithms use quick sort, but attempt to recognize when it starts to degenerate, and switch to heap sort then. Merge sort is used when the data structure doesn't support random access, since it works with pure sequential access (forward iterators, rather than random access iterators). It's used in std::list<>::sort, for example. It's also widely used for external sorting, where random access can be very, very expensive compared to sequential access. (When sorting a file which doesn't fit into memory, you might break it into chunks which fit into memory, sort these using quicksort, writing each out to a file, then merge sort the generated files.)**

Time complexity comparison sorts.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Timsort** | [**Merge sort**](http://en.wikipedia.org/wiki/Merge_sort) | [**Quicksort**](http://en.wikipedia.org/wiki/Quicksort) | [**Insertion sort**](http://en.wikipedia.org/wiki/Insertion_sort) | [**Selection sort**](http://en.wikipedia.org/wiki/Selection_sort) | [**Smoothsort**](http://en.wikipedia.org/wiki/Smoothsort) |
| **Best case** | \Theta(n) | \Theta(n \log n) | \Theta(n \log n) | \Theta(n) | \Theta(n^2) | \Theta(n) |
| **Average case** | \Theta(n \log n) | \Theta(n \log n) | \Theta(n \log n) | \Theta(n^2) | \Theta(n^2) | \Theta(n \log n) |
| **Worst case** | \Theta(n \log n) | \Theta(n \log n) | \Theta(n^2) | \Theta(n^2) | \Theta(n^2) | \Theta(n \log n) |

The following table provides a comparison of the space complexities of the various sorting techniques. Note that for merge sort, the *worst case* space complexity is usually O(n).

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Timsort** | [**Merge sort**](http://en.wikipedia.org/wiki/Merge_sort) | [**Quicksort**](http://en.wikipedia.org/wiki/Quicksort) | [**Insertion sort**](http://en.wikipedia.org/wiki/Insertion_sort) | [**Selection sort**](http://en.wikipedia.org/wiki/Selection_sort) | [**Smoothsort**](http://en.wikipedia.org/wiki/Smoothsort) |
| **Space complexity** | O(n) | O(n) | O(\log n) | O(1) | O(1) | O(\log n) |

Quick sort is inplace (doesn't need extra memmory, other than a constant amount.)

Mergesort is quicker when dealing with linked lists. This is because pointers can easily be changed when merging lists. It only requires one pass (O(n)) through the list.

Quicksort's in-place algorithm requires the movement (swapping) of data. While this can be very efficient for in-memory dataset, it can be much more expensive if your dataset doesn't fit in memory. The result would be lots of I/O.

Quick sort is typically faster than merge sort when the data is stored in memory. However, when the data set is huge and is stored on external devices such as a hard drive, merge sort is the clear winner in terms of speed. It minimizes the expensive reads of the external drive and also lends itself well to parallel computing.

In Arrays.sort, Java uses insertion sort if the array length is less than 7 and int type array and the length is greater than 7 it uses tuned quicksort. For Object[] type array it uses MergeSort.

**MergeSort**

Conceptually, a merge sort works as follows

Divide the unsorted list into n sublists, each containing 1 element (a list of 1 element is considered sorted).

Repeatedly merge sublists to produce new sorted sublists until there is only 1 sublist remaining. This will be the sorted list.

**1.** **Divide Step**

If a given array A has zero or one element, simply return; it is already sorted. Otherwise, split A[p .. r] into two subarrays A[p .. q] and A[q + 1 .. r], each containing about half of the elements of A[p .. r]. That is, q is the halfway point of A[p .. r].

**2. Conquer Step**

Conquer by recursively sorting the two subarrays A[p .. q] and A[q + 1 .. r].

**3. Combine Step**

Combine the elements back in A[p .. r] by merging the two sorted subarrays A[p .. q] and A[q + 1 .. r] into a sorted sequence. To accomplish this step, we will define a procedure MERGE (A, p, q, r).

The mergesort algorithm is based on the classical divide-and-conquer paradigm. It operates as follows:

**DIVIDE: Partition the n-element sequence to be sorted into two subsequences of n/2 elements each.**

**CONQUER: Sort the two subsequences recursively using the mergesort.**

**COMBINE: Merge the two sorted sorted subsequences of size n/2 each to produce the sorted sequence consisting of n elements.**

**Complexity**

**Worst case performance O(n log n)**

**Best case performance O(n log n) typical, O(n) natural variant**

**Average case performance O(n log n)**

**Worst case space complexity O(n) auxiliary**

**Auxiliary Space: O(n)**

**Applications of Merge Sort**

1) Merge Sort is useful for sorting linked lists in O(nLogn) time. Other nlogn algorithms like Heap Sort, Quick Sort (average case nLogn) cannot be applied to linked lists.

2) Inversion Count Problem

3) Used in External Sorting

Merge sort is often preferred for sorting a linked list. The slow random-access performance of a linked list makes some other algorithms (such as quicksort) perform poorly, and others (such as heapsort) completely impossible.

Advantages:

Guaranteed to run in ? (nlgn)

Disadvantage

Requires extra space ? N

Advantage of merge sort

Good for sorting slow-access data e.g. tape drive or hard disk.

It is excellent for sorting data that are normally accessed sequentially. e.g. linked lists, tape drive, hard disk and receiving online data one item at a time

Advantage over Quicksort

Better at handling sequential-accessed lists

If two equal valued items are in the list, then their relative locations are preserved (this is called "sort-stable") i.e. if item A = "cat" and item C = "cat" then the sorted list will have AC. Quicksort does not always keep this order - it could be AC or CA

Disadvantages

In many implementations, if the list is N long, then it needs 2 x N memory space to handle the sort.

If recursion is used to code the algorithm then it uses twice as much stack memory as quicksort - on the other hand it is not difficult to code using iteration rather than recursion and so avoid the memory penalty.

Quicksort is considered the fastest method on most types of lists.

**Comment on Performance**

Merge sort always takes 2N log2 N steps. On average quick sort takes N log2 N steps. You would expect merge sort

to be about twice as slow due to copying to b[] and back again to a[]. However, the performance of quick sort

depends on a good choice of pivot. In our approach we choose the first element as the pivot. If an array is already

nearly sorted, this can lead to very poor performance for quick sort. In its worst case quick sort requires N2

/2 steps

which is the same performance as bubble sort and selection sort.

The main drawbacks of merge sort as opposed to quick sort are:

• it requires an extra auxiliary array b[p]

• on average is about twice as slow

However, in situations where the data can vary in how much it is already sorted, merge sort is more stable.

**Insertion Sort**

Insertion sort is a simple sorting algorithm: a comparison sort in which the sorted array (or list) is built one entry at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort. However, insertion sort provides several advantages:

**Algorithm**

Every repetition of insertion sort removes an element from the input data, inserting it into the correct position in the already-sorted list, until no input elements remain. The choice of which element to remove from the input is arbitrary, and can be made using almost any choice algorithm.

Sorting is typically done in-place. The resulting array after k iterations has the property where the first k + 1 entries are sorted. In each iteration the first remaining entry of the input is removed, inserted into the result at the correct position, thus extending the result:

**Complexity**

Worst case performance (n2) comparisons, swaps

Best case performance O(n) comparisons, O(1) swaps

Average case performance (n2) comparisons, swaps

Worst case space complexity (n) total, O(1) auxiliary

**Sorting algorithms**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Algorithm | Data Structure | Time Complexity:Best | Time Complexity:Average | Time Complexity:Worst | Space Complexity:Worst |
| Quick Sort | Array | O(n log(n)) | O(n log(n)) | O(n^2) | O(log(n)) |
| Merge sort | Array | O(n log(n)) | O(n log(n)) | O(n log(n)) | O(n) |
| Bubble sort | Array | O(n) | O(n^2) | O(n^2) | O(1) |
| Insertion sort | Array | O(n) | O(n^2) | O(n^2) | O(1) |
| Selection sort | Array | O(n^2) | O(n^2) | O(n^2) | O(1) |

**Best, worst, and average cases**

**The best case input is an array that is already sorted**. In this case insertion sort has a linear running time (i.e., [Θ](http://en.wikipedia.org/wiki/Big_Theta_notation)(*n*)). During each iteration, the first remaining element of the input is only compared with the right-most element of the sorted subsection of the array.

**The simplest worst case input is an array sorted in reverse order**. The set of all worst case inputs consists of all arrays where each element is the smallest or second-smallest of the elements before it. In these cases every iteration of the inner loop will scan and shift the entire sorted subsection of the array before inserting the next element. This gives insertion sort a quadratic running time (i.e., O(*n*2)).

**The average case is also quadratic, which makes insertion sort impractical for sorting large arrays. However, insertion sort is one of the fastest algorithms for sorting very small arrays, even faster than**[**quicksort**](http://en.wikipedia.org/wiki/Quicksort)**; indeed, good**[**quicksort**](http://en.wikipedia.org/wiki/Quicksort)**implementations use insertion sort for arrays smaller than a certain threshold, also when arising as subproblems; the exact threshold must be determined experimentally and depends on the machine, but is commonly around ten.**

Red–black tree

A **red–black tree** is a type of [self-balancing binary search tree](http://en.wikipedia.org/wiki/Self-balancing_binary_search_tree), a [data structure](http://en.wikipedia.org/wiki/Data_structure) used in [computer science](http://en.wikipedia.org/wiki/Computer_science).

The self-balancing is provided by painting each node with one of two colors (these are typically called 'red' and 'black', hence the name of the trees) in such a way that the resulting painted tree satisfies certain properties that don't allow it to become significantly unbalanced. When the tree is modified, the new tree is subsequently rearranged and repainted to restore the coloring properties. The properties are designed in such a way that this rearranging and recoloring can be performed efficiently.

The balancing of the tree is not perfect but it is good enough to allow it to guarantee searching in [O](http://en.wikipedia.org/wiki/Big-O_notation)(log *n*) time, where *n* is the total number of elements in the tree. The insertion, and deletion operations, along with the tree rearrangement and recoloring are also performed in [O](http://en.wikipedia.org/wiki/Big-O_notation)(log *n*) time.[[1]](http://en.wikipedia.org/wiki/Red%E2%80%93black_tree#cite_note-1)

Tracking the color of each node requires only 1 bit of information per node because there are only two colors. The tree does not contain any other data specific to its being a red–black tree so its memory footprint is almost identical to classic (uncolored) binary search tree. In many cases the additional bit of information can be stored at no additional memory cost.

**Complexity**

**Average Worst case**

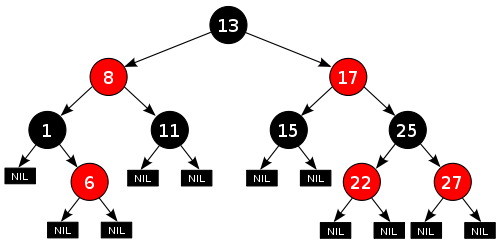
**Space O(n) O(n)**

**Search O(log n) O(log n)**

**Insert O(log n) O(log n)**

**Delete O(log n) O(log n)**

## Properties[[edit](http://en.wikipedia.org/w/index.php?title=Red%E2%80%93black_tree&action=edit&section=3)]

[](http://en.wikipedia.org/wiki/File:Red-black_tree_example.svg)

[http://bits.wikimedia.org/static-1.23wmf10/skins/common/images/magnify-clip.png](http://en.wikipedia.org/wiki/File:Red-black_tree_example.svg)

An example of a red–black tree

In addition to the requirements imposed on a [binary search tree](http://en.wikipedia.org/wiki/Binary_search_tree) the following must be satisfied by a red–black tree:[[5]](http://en.wikipedia.org/wiki/Red%E2%80%93black_tree#cite_note-5)

1. A node is either red or black.
2. The root is black. (This rule is sometimes omitted. Since the root can always be changed from red to black, but not necessarily vice-versa, this rule has little effect on analysis.)
3. All leaves (NIL) are black. (All leaves are same color as the root.)
4. Every red node must have two black child nodes.
5. Every [path](http://en.wikipedia.org/wiki/Path_(graph_theory)) from a given node to any of its descendant leaves contains the same number of black nodes.

These constraints enforce a critical property of red–black trees: that the path from the root to the furthest leaf is no more than twice as long as the path from the root to the nearest leaf. The result is that the tree is roughly height-balanced. Since operations such as inserting, deleting, and finding values require worst-case time proportional to the height of the tree, this theoretical upper bound on the height allows red–black trees to be efficient in the worst case, unlike ordinary [binary search trees](http://en.wikipedia.org/wiki/Binary_search_tree).

To see why this is guaranteed, it suffices to consider the effect of properties 4 and 5 together. For a red–black tree T, let B be the number of black nodes in property 5. Let the shortest possible path from the root of T to any leaf consist of B black nodes. Longer possible paths may be constructed by inserting red nodes. However, property 4 makes it impossible to insert more than one consecutive red node. Therefore the longest possible path consists of 2B nodes, alternating black and red.

The shortest possible path has all black nodes, and the longest possible path alternates between red and black nodes. Since all maximal paths have the same number of black nodes, by property 5, this shows that no path is more than twice as long as any other path.

## Operations[[edit](http://en.wikipedia.org/w/index.php?title=Red%E2%80%93black_tree&action=edit&section=6)]

Read-only operations on a red–black tree require no modification from those used for [binary search trees](http://en.wikipedia.org/wiki/Binary_search_tree), because every red–black tree is a special case of a simple binary search tree. However, the immediate result of an insertion or removal may violate the properties of a red–black tree. Restoring the red–black properties requires a small number ([O](http://en.wikipedia.org/wiki/Big-O_notation)(log *n*) or [amortized O(1)](http://en.wikipedia.org/wiki/Amortized_analysis)) of color changes (which are very quick in practice) and no more than three [tree rotations](http://en.wikipedia.org/wiki/Tree_rotation) (two for insertion). Although insert and delete operations are complicated, their times remain O(log *n*).

### Insertion[[edit](http://en.wikipedia.org/w/index.php?title=Red%E2%80%93black_tree&action=edit&section=7)]

Insertion begins by adding the node as any [binary search tree insertion](http://en.wikipedia.org/wiki/Binary_search_tree#Insertion) does and by coloring it red. Whereas in the binary search tree, we always add a leaf, in the red–black tree leaves contain no information, so instead we add a red interior node, with two black leaves, in place of an existing black leaf.

What happens next depends on the color of other nearby nodes. The term *uncle node* will be used to refer to the sibling of a node's parent, as in human family trees. Note that:

* property 3 (all leaves are black) always holds.
* property 4 (both children of every red node are black) is threatened only by adding a red node, repainting a black node red, or a rotation.
* property 5 (all paths from any given node to its leaf nodes contain the same number of black nodes) is threatened only by adding a black node, repainting a red node black (or vice versa), or a rotation.

*Note*: The label **N** will be used to denote the current node (colored red). At the beginning, this is the new node being inserted, but the entire procedure may also be applied recursively to other nodes (see case 3). **P** will denote **N'**s parent node, **G** will denote **N'**s grandparent, and **U** will denote **N'**s uncle. Note that in between some cases, the roles and labels of the nodes are exchanged, but in each case, every label continues to represent the same node it represented at the beginning of the case. Any color shown in the diagram is either assumed in its case or implied by those assumptions. A numbered triangle represents a subtree of unspecified depth. A black circle atop the triangle designates a black root node, otherwise the root node's color is unspecified.

Each case will be demonstrated with example [C](http://en.wikipedia.org/wiki/C_(programming_language)) code. The uncle and grandparent nodes can be found by these functions:

struct node \*grandparent(struct node \*n)

{

if ((n != NULL) && (n->parent != NULL))

return n->parent->parent;

else

return NULL;

}

struct node \*uncle(struct node \*n)

{

struct node \*g = grandparent(n);

if (g == NULL)

return NULL; *// No grandparent means no uncle*

if (n->parent == g->left)

return g->right;

else

return g->left;

}

**Case 1:** The current node **N** is at the root of the tree. In this case, it is repainted black to satisfy property 2 (the root is black). Since this adds one black node to every path at once, property 5 (all paths from any given node to its leaf nodes contain the same number of black nodes) is not violated.

void insert\_case1(struct node \*n)

{

if (n->parent == NULL)

n->color = BLACK;

else

insert\_case2(n);

}

**Case 2:** The current node's parent **P** is black, so property 4 (both children of every red node are black) is not invalidated. In this case, the tree is still valid. Property 5 (all paths from any given node to its leaf nodes contain the same number of black nodes) is not threatened, because the current node **N** has two black leaf children, but because **N** is red, the paths through each of its children have the same number of black nodes as the path through the leaf it replaced, which was black, and so this property remains satisfied.

void insert\_case2(struct node \*n)

{

if (n->parent->color == BLACK)

return; */\* Tree is still valid \*/*

else

insert\_case3(n);

}

*Note:* In the following cases it can be assumed that **N** has a grandparent node **G**, because its parent **P** is red, and if it were the root, it would be black. Thus, **N** also has an uncle node **U**, although it may be a leaf in cases 4 and 5.

|  |
| --- |
| [Diagram of case 3](http://en.wikipedia.org/wiki/File:Red-black_tree_insert_case_3.png)  **Case 3:** If both the parent **P** and the uncle **U** are red, then both of them can be repainted black and the grandparent **G** becomes red (to maintain property 5 (all paths from any given node to its leaf nodes contain the same number of black nodes)). Now, the current red node **N**has a black parent. Since any path through the parent or uncle must pass through the grandparent, the number of black nodes on these paths has not changed. However, the grandparent **G** may now violate properties 2 (The root is black) or 4 (Both children of every red node are black) (property 4 possibly being violated since **G** may have a red parent). To fix this, the entire procedure is recursively performed on **G**from case 1. Note that this is a tail-recursive call, so it could be rewritten as a loop; since this is the only loop, and any rotations occur after this loop, this proves that a constant number of rotations occur. |

void insert\_case3(struct node \*n)

{

struct node \*u = uncle(n), \*g;

if ((u != NULL) && (u->color == RED)) {

n->parent->color = BLACK;

u->color = BLACK;

g = grandparent(n);

g->color = RED;

insert\_case1(g);

} else {

insert\_case4(n);

}

}

*Note:* In the remaining cases, it is assumed that the parent node **P** is the left child of its parent. If it is the right child, *left* and *right* should be reversed throughout cases 4 and 5. The code samples take care of this.

|  |
| --- |
| [Diagram of case 4](http://en.wikipedia.org/wiki/File:Red-black_tree_insert_case_4.png)  **Case 4:** The parent **P** is red but the uncle **U** is black; also, the current node **N** is the right child of **P**, and **P** in turn is the left child of its parent**G**. In this case, a [left rotation](http://en.wikipedia.org/wiki/Tree_rotation) on **P** that switches the roles of the current node **N** and its parent **P** can be performed; then, the former parent node **P** is dealt with using case 5 (relabeling **N** and **P**) because property 4 (both children of every red node are black) is still violated. The rotation causes some paths (those in the sub-tree labelled "1") to pass through the node **N** where they did not before. It also causes some paths (those in the sub-tree labelled "3") not to pass through the node **P** where they did before. However, both of these nodes are red, so property 5 (all paths from any given node to its leaf nodes contain the same number of black nodes) is not violated by the rotation. After this case has been completed, property 4 (both children of every red node are black) is still violated, but now we can resolve this by continuing to case 5. |

void insert\_case4(struct node \*n)

{

struct node \*g = grandparent(n);

if ((n == n->parent->right) && (n->parent == g->left)) {

rotate\_left(n->parent);

*/\**

*\* rotate\_left can be the below because of already having \*g = grandparent(n)*

*\**

*\* struct node \*saved\_p=g->left, \*saved\_left\_n=n->left;*

*\* g->left=n;*

*\* n->left=saved\_p;*

*\* saved\_p->right=saved\_left\_n;*

*\**

*\* and modify the parent's nodes properly*

*\*/*

n = n->left;

} else if ((n == n->parent->left) && (n->parent == g->right)) {

rotate\_right(n->parent);

*/\**

*\* rotate\_right can be the below to take advantage of already having \*g = grandparent(n)*

*\**

*\* struct node \*saved\_p=g->right, \*saved\_right\_n=n->right;*

*\* g->right=n;*

*\* n->right=saved\_p;*

*\* saved\_p->left=saved\_right\_n;*

*\**

*\*/*

n = n->right;

}

insert\_case5(n);

}

|  |
| --- |
| [Diagram of case 5](http://en.wikipedia.org/wiki/File:Red-black_tree_insert_case_5.png)  **Case 5:** The parent **P** is red but the uncle **U** is black, the current node **N** is the left child of **P**, and **P** is the left child of its parent **G**. In this case, a [right rotation](http://en.wikipedia.org/wiki/Tree_rotation) on **G** is performed; the result is a tree where the former parent **P** is now the parent of both the current node **N** and the former grandparent **G**. **G** is known to be black, since its former child **P** could not have been red otherwise (without violating property 4). Then, the colors of **P** and **G** are switched, and the resulting tree satisfies property 4 (both children of every red node are black). Property 5 (all paths from any given node to its leaf nodes contain the same number of black nodes) also remains satisfied, since all paths that went through any of these three nodes went through **G** before, and now they all go through **P**. In each case, this is the only black node of the three. |

void insert\_case5(struct node \*n)

{

struct node \*g = grandparent(n);

n->parent->color = BLACK;

g->color = RED;

if (n == n->parent->left)

rotate\_right(g);

else

rotate\_left(g);

}

Note that inserting is actually [in-place](http://en.wikipedia.org/wiki/In-place_algorithm), since all the calls above use [tail recursion](http://en.wikipedia.org/wiki/Tail_recursion).

### Removal[[edit](http://en.wikipedia.org/w/index.php?title=Red%E2%80%93black_tree&action=edit&section=8)]

In a regular binary search tree when deleting a node with two non-leaf children, we find either the maximum element in its left subtree (which is the in-order predecessor) or the minimum element in its right subtree (which is the in-order successor) and move its value into the node being deleted (as shown [here](http://en.wikipedia.org/wiki/Binary_search_tree#Deletion)). We then delete the node we copied the value from, which must have fewer than two non-leaf children. (Non-leaf children, rather than all children, are specified here because unlike normal binary search trees, red–black trees can have leaf nodes anywhere, so that all nodes are either internal nodes with two children or leaf nodes with, by definition, zero children. In effect, internal nodes having two leaf children in a red–black tree are like the leaf nodes in a regular binary search tree.) Because merely copying a value does not violate any red–black properties, this reduces to the problem of deleting a node with at most one non-leaf child. Once we have solved that problem, the solution applies equally to the case where the node we originally want to delete has at most one non-leaf child as to the case just considered where it has two non-leaf children.

Therefore, for the remainder of this discussion we address the deletion of a node with at most one non-leaf child. We use the label **M** to denote the node to be deleted; **C** will denote a selected child of **M**, which we will also call "its child". If **M** does have a non-leaf child, call that its child, **C**; otherwise, choose either leaf as its child, **C**.

If **M** is a red node, we simply replace it with its child **C**, which must be black by property 4. (This can only occur when **M** has two leaf children, because if the red node **M** had a black non-leaf child on one side but just a leaf child on the other side, then the count of black nodes on both sides would be different, thus the tree would violate property 5.) All paths through the deleted node will simply pass through one fewer red node, and both the deleted node's parent and child must be black, so property 3 (all leaves are black) and property 4 (both children of every red node are black) still hold.

Another simple case is when **M** is black and **C** is red. Simply removing a black node could break Properties 4 (“Both children of every red node are black”) and 5 (“All paths from any given node to its leaf nodes contain the same number of black nodes”), but if we repaint **C** black, both of these properties are preserved.

The complex case is when both **M** and **C** are black. (This can only occur when deleting a black node which has two leaf children, because if the black node **M** had a black non-leaf child on one side but just a leaf child on the other side, then the count of black nodes on both sides would be different, thus the tree would have been an invalid red–black tree by violation of property 5.) We begin by replacing **M** with its child **C**. We will call (or *label*—that is, *relabel*) this child (in its new position) **N**, and its sibling (its new parent's other child) **S**. (**S** was previously the sibling of **M**.) In the diagrams below, we will also use **P** for **N'**s new parent (**M'**s old parent), **SL** for **S'**s left child, and **SR** for **S'**s right child (**S** cannot be a leaf because if **M** and **C** were black, then **P'**s one subtree which included **M** counted two black-height and thus **P'**s other subtree which includes **S** must also count two black-height, which cannot be the case if **S** is a leaf node).

*Note*: In between some cases, we exchange the roles and labels of the nodes, but in each case, every label continues to represent the same node it represented at the beginning of the case. Any color shown in the diagram is either assumed in its case or implied by those assumptions. White represents an unknown color (either red or black).

We will find the sibling using this function:

struct node \*sibling(struct node \*n)

{

if (n == n->parent->left)

return n->parent->right;

else

return n->parent->left;

}

*Note*: In order that the tree remains well-defined, we need that every null leaf remains a leaf after all transformations (that it will not have any children). If the node we are deleting has a non-leaf (non-null) child **N**, it is easy to see that the property is satisfied. If, on the other hand, **N** would be a null leaf, it can be verified from the diagrams (or code) for all the cases that the property is satisfied as well.

We can perform the steps outlined above with the following code, where the function replace\_node substitutes child into n's place in the tree. For convenience, code in this section will assume that null leaves are represented by actual node objects rather than NULL (the code in the *Insertion* section works with either representation).

void delete\_one\_child(struct node \*n)

{

*/\**

*\* Precondition: n has at most one non-null child.*

*\*/*

struct node \*child = is\_leaf(n->right) ? n->left : n->right;

replace\_node(n, child);

if (n->color == BLACK) {

if (child->color == RED)

child->color = BLACK;

else

delete\_case1(child);

}

free(n);

}

*Note*: If **N** is a null leaf and we do not want to represent null leaves as actual node objects, we can modify the algorithm by first calling delete\_case1() on its parent (the node that we delete, nin the code above) and deleting it afterwards. We can do this because the parent is black, so it behaves in the same way as a null leaf (and is sometimes called a 'phantom' leaf). And we can safely delete it at the end as n will remain a leaf after all operations, as shown above.

If both **N** and its original parent are black, then deleting this original parent causes paths which proceed through **N** to have one fewer black node than paths that do not. As this violates property 5 (all paths from any given node to its leaf nodes contain the same number of black nodes), the tree must be rebalanced. There are several cases to consider:

**Case 1:** **N** is the new root. In this case, we are done. We removed one black node from every path, and the new root is black, so the properties are preserved.

void delete\_case1(struct node \*n)

{

if (n->parent != NULL)

delete\_case2(n);

}

*Note*: In cases 2, 5, and 6, we assume **N** is the left child of its parent **P**. If it is the right child, *left* and *right* should be reversed throughout these three cases. Again, the code examples take both cases into account.

|  |
| --- |
| [Diagram of case 2](http://en.wikipedia.org/wiki/File:Red-black_tree_delete_case_2.png)  **Case 2:** **S** is red. In this case we reverse the colors of **P** and **S**, and then [rotate](http://en.wikipedia.org/wiki/Tree_rotation) left at **P**, turning **S** into **N'**s grandparent. Note that **P** has to be black as it had a red child. Although all paths still have the same number of black nodes, now **N** has a black sibling and a red parent, so we can proceed to step 4, 5, or 6. (Its new sibling is black because it was once the child of the red **S**.) In later cases, we will relabel **N'**s new sibling as **S**. |

void delete\_case2(struct node \*n)

{

struct node \*s = sibling(n);

if (s->color == RED) {

n->parent->color = RED;

s->color = BLACK;

if (n == n->parent->left)

rotate\_left(n->parent);

else

rotate\_right(n->parent);

}

delete\_case3(n);

}

|  |
| --- |
| [Diagram of case 3](http://en.wikipedia.org/wiki/File:Red-black_tree_delete_case_3.png)  **Case 3:** **P**, **S**, and **S'**s children are black. In this case, we simply repaint **S** red. The result is that all paths passing through **S**, which are precisely those paths *not* passing through **N**, have one less black node. Because deleting **N'**s original parent made all paths passing through **N** have one less black node, this evens things up. However, all paths through **P** now have one fewer black node than paths that do not pass through **P**, so property 5 (all paths from any given node to its leaf nodes contain the same number of black nodes) is still violated. To correct this, we perform the rebalancing procedure on **P**, starting at case 1. |

void delete\_case3(struct node \*n)

{

struct node \*s = sibling(n);

if ((n->parent->color == BLACK) &&

(s->color == BLACK) &&

(s->left->color == BLACK) &&

(s->right->color == BLACK)) {

s->color = RED;

delete\_case1(n->parent);

} else

delete\_case4(n);

}

|  |
| --- |
| [Diagram of case 4](http://en.wikipedia.org/wiki/File:Red-black_tree_delete_case_4.png)  **Case 4:** **S** and **S'**s children are black, but **P** is red. In this case, we simply exchange the colors of **S** and **P**. This does not affect the number of black nodes on paths going through **S**, but it does add one to the number of black nodes on paths going through **N**, making up for the deleted black node on those paths. |

void delete\_case4(struct node \*n)

{

struct node \*s = sibling(n);

if ((n->parent->color == RED) &&

(s->color == BLACK) &&

(s->left->color == BLACK) &&

(s->right->color == BLACK)) {

s->color = RED;

n->parent->color = BLACK;

} else

delete\_case5(n);

}

|  |
| --- |
| [Diagram of case 5](http://en.wikipedia.org/wiki/File:Red-black_tree_delete_case_5.png)  **Case 5:** **S** is black, **S'**s left child is red, **S'**s right child is black, and **N** is the left child of its parent. In this case we rotate right at **S**, so that **S'**s left child becomes **S'**s parent and **N'**s new sibling. We then exchange the colors of **S** and its new parent. All paths still have the same number of black nodes, but now **N** has a black sibling whose right child is red, so we fall into case 6. Neither **N** nor its parent are affected by this transformation. (Again, for case 6, we relabel **N'**s new sibling as **S**.) |

void delete\_case5(struct node \*n)

{

struct node \*s = sibling(n);

if (s->color == BLACK) { */\* this if statement is trivial,*

*due to case 2 (even though case 2 changed the sibling to a sibling's child,*

*the sibling's child can't be red, since no red parent can have a red child). \*/*

*/\* the following statements just force the red to be on the left of the left of the parent,*

*or right of the right, so case six will rotate correctly. \*/*

if ((n == n->parent->left) &&

(s->right->color == BLACK) &&

(s->left->color == RED)) { */\* this last test is trivial too due to cases 2-4. \*/*

s->color = RED;

s->left->color = BLACK;

rotate\_right(s);

} else if ((n == n->parent->right) &&

(s->left->color == BLACK) &&

(s->right->color == RED)) {*/\* this last test is trivial too due to cases 2-4. \*/*

s->color = RED;

s->right->color = BLACK;

rotate\_left(s);

}

}

delete\_case6(n);

}

|  |
| --- |
| [Diagram of case 6](http://en.wikipedia.org/wiki/File:Red-black_tree_delete_case_6.png)  **Case 6:** **S** is black, **S'**s right child is red, and **N** is the left child of its parent **P**. In this case we rotate left at **P**, so that **S** becomes the parent of **P** and **S'**s right child. We then exchange the colors of **P** and **S**, and make **S'**s right child black. The subtree still has the same color at its root, so Properties 4 (Both children of every red node are black) and 5 (All paths from any given node to its leaf nodes contain the same number of black nodes) are not violated. However, **N** now has one additional black ancestor: either **P** has become black, or it was black and**S** was added as a black grandparent. Thus, the paths passing through **N** pass through one additional black node.  Meanwhile, if a path does not go through **N**, then there are two possibilities:   * It goes through **N'**s new sibling. Then, it must go through **S** and **P**, both formerly and currently, as they have only exchanged colors and places. Thus the path contains the same number of black nodes. * It goes through **N'**s new uncle, **S'**s right child. Then, it formerly went through **S**, **S'**s parent, and **S'**s right child (which was red), but now only goes through **S**, which has assumed the color of its former parent, and **S'**s right child, which has changed from red to black (assuming **S'**s color: black). The net effect is that this path goes through the same number of black nodes.   Either way, the number of black nodes on these paths does not change. Thus, we have restored Properties 4 (Both children of every red node are black) and 5 (All paths from any given node to its leaf nodes contain the same number of black nodes). The white node in the diagram can be either red or black, but must refer to the same color both before and after the transformation. |

void delete\_case6(struct node \*n)

{

struct node \*s = sibling(n);

s->color = n->parent->color;

n->parent->color = BLACK;

if (n == n->parent->left) {

s->right->color = BLACK;

rotate\_left(n->parent);

} else {

s->left->color = BLACK;

rotate\_right(n->parent);

}

}

Again, the function calls all use [tail recursion](http://en.wikipedia.org/wiki/Tail_recursion), so the algorithm is [in-place](http://en.wikipedia.org/wiki/In-place_algorithm). In the algorithm above, all cases are chained in order, except in delete case 3 where it can recurse to case 1 back to the parent node: this is the only case where an in-place implementation will effectively loop (after only one rotation in case 3).

Additionally, no tail recursion ever occurs on a child node, so the tail recursion loop can only move from a child back to its successive ancestors. No more than O(log *n*) loops back to case 1 will occur (where *n* is the total number of nodes in the tree before deletion). If a rotation occurs in case 2 (which is the only possibility of rotation within the loop of cases 1–3), then the parent of the node **N** becomes red after the rotation and we will exit the loop. Therefore at most one rotation will occur within this loop. Since no more than two additional rotations will occur after exiting the loop, at most three rotations occur in total.

**Red-black trees** are an evolution of binary search trees that aim to keep the tree balanced without affecting the complexity of the primitive operations. This is done by coloring each node in the tree with either red or black and preserving a set of properties that guarantee that the deepest path in the tree is not longer than twice the shortest one.

A red-black tree is a binary search tree with the following properties:

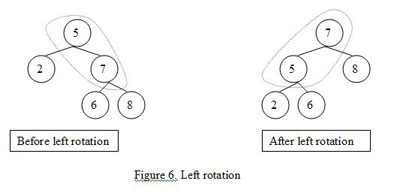
1. Every node is colored with either red or black.
2. All leaf (nil) nodes are colored with black; if a node’s child is missing then we will assume that it has a nil child in that place and this nil child is always colored black.
3. Both children of a red node must be black nodes.
4. Every path from a node n to a descendent leaf has the same number of black nodes (not counting node n). We call this number the **black height** of n, which is denoted by bh(n).

**Rotations**

How does inserting or deleting nodes affect a red-black tree? To ensure that its color scheme and properties don't get thrown off, red-black trees employ a key operation known as **rotation**. Rotation is a binary operation, between a parent node and one of its children, that swaps nodes and modifys their pointers while preserving the inorder traversal of the tree (so that elements are still sorted).

There are two types of rotations: left rotation and right rotation. Left rotation swaps the parent node with its right child, while right rotation swaps the parent node with its left child. Here are the steps involved in for left rotation (for right rotations just change "left" to "right" below):

* Assume node x is the parent and node y is a non-leaf right child.
* Let y be the parent and x be its left child.
* Let y’s left child be x’s right child.



**Operations on red-black tree (insertion, deletion and retrieval)**

Red-black tree operations are a modified version of BST operations, with the modifications aiming to preserve the properties of red-black trees while keeping the operations complexity a function of tree height.

**Red-black tree insertion:**  
Inserting a node in a red-black tree is a two step process:

1. A BST insertion, which takes O(log n) as shown before.
2. Fixing any violations to red-black tree properties that may occur after applying step 1. This step is O(log n) also, as we start by fixing the newly inserted node, continuing up along the path to the root node and fixing nodes along that path. Fixing a node is done in constant time and involves re-coloring some nodes and doing rotations.

Accordingly the total running time of the insertion process is O(log n).

**Red-black tree deletion:**  
The same concept behind red-black tree insertions applies here. Removing a node from a red-black tree makes use of the BST deletion procedure and then restores the red-black tree properties in O(log n). The total running time for the deletion process takes O(log n) time, then, which meets the complexity requirements for the primitive operations.

**Red-black tree retrieval:**  
Retrieving a node from a red-black tree doesn’t require more than the use of the BST procedure, which takes O(log n) time.

A [red-black tree](http://en.wikipedia.org/wiki/Red-black_tree) is a particular implementation of a [self-balancing binary search tree](http://en.wikipedia.org/wiki/Self-balancing_binary_search_tree), and today it seems to be the most popular choice of implementation.

[Binary search trees](http://en.wikipedia.org/wiki/Binary_search_tree) are used to implement finite maps, where you store a set of keys with associated values. You can also implement sets by only using the keys and not storing any values.

Balancing the tree is needed to guarantee good performance, as otherwise the tree could degenerate into a list, for example if you insert keys which are already sorted.

**Operation Time**

**Search O(log N)**

**Insert O(log N)**

**Delete O(log N)**

**Binary search tree**

In computer science, a binary search tree (BST), sometimes also called an ordered or sorted binary tree, is a node-based binary tree data structure which has the following properties:[1]

The left subtree of a node contains only nodes with keys less than the node's key.

The right subtree of a node contains only nodes with keys greater than the node's key.

The left and right subtree each must also be a binary search tree.

There must be no duplicate nodes.

Generally, the information represented by each node is a record rather than a single data element. However, for sequencing purposes, nodes are compared according to their keys rather than any part of their associated records.

The major advantage of binary search trees over other data structures is that the related sorting algorithms and search algorithms such as in-order traversal can be very efficient.

**Average Worst case**

**Space O(n) O(n)**

**Search O(log n) O(n)**

**Insert O(log n) O(n)**

**Delete O(log n) O(n)**

* A **balanced binary tree** is commonly defined as a binary tree in which the depth of the left and right subtrees of every node differ by 1 or less,[[4]](http://en.wikipedia.org/wiki/Binary_tree#cite_note-4)although in general it is a binary tree where no leaf is much farther away from the root than any other leaf. (Different balancing schemes allow different definitions of "much farther".[[5]](http://en.wikipedia.org/wiki/Binary_tree#cite_note-5)) Binary trees that are balanced according to this definition have a predictable depth (how many nodes are traversed from the root to a leaf, counting the root as node 0 and subsequent nodes as 1, 2, ..., *n*). This depth (also called the height) is equal to the integer part of log2(*n*), where *n* is the number of nodes on the balanced tree. For example, for a balanced tree with only 1 node, log2(1) = 0, so the depth of the tree is 0. For a balanced tree with 100 nodes, log2(100) = 6.64, so it has a depth of 6.

**Searching for a value is in a tree of N nodes is:**

**O(log N) if the tree is “balanced”**

**O(N) if the tree is “unbalanced”**

Properties of Binary Trees

**A binary tree is a full binary tree if and only if:**

**Each non leaf node has exactly two child nodes**

**All leaf nodes have identical path length**

**It is called full since all possible node slots are occupied**

**A height-balanced binary tree is a binary tree such that:**

**The left & right subtrees for any given node differ in height by no more than one**

**Note: Each complete binary tree is a height-balanced binary tree**

**Binary search trees provide O(log N) search times provided that the nodes are**

**distributed in a reasonably “balanced” manner. When a BST becomes badly unbalanced, the search behavior can degenerate to that of a sorted linked list, O(N).**

**AVL tree**

**\*: a binary search tree in which the heights of the left and right**

**subtrees of the root differ by at most 1, and in which the left and**

**right subtrees are themselves AVL trees. How effective is this? The height of an AVL tree with N nodes never exceeds**

**1.44 log N and is typically much closer to log N.**

**Example of Binary Search Tree**

\begin{figure}
\centerline{\psfig{figure=figures/Fbstexample.ps}}
\end{figure}