**MergeSort**

void mergesort(int[] array) {

int[] helper = new int[array.length];

mergesort(array, helper, 0, array.length - 1);

}

void mergesort(int[] array, int[] helper, int low, int high) {

if (low < high) {

int middle = (low + high) / 2;

mergesort(array, helper, low, middle); // Sort left half

mergesort(array, helper, middle+1, high); // Sort right half

merge(array, helper, low, middle, high); // Merge them

}

}

void merge(int[] array, int[] helper, int low, int middle, int high) {

/\* Copy both halves into a helper array \*/

for (int i = low; i <= high; i++) {

helper[i] = array[i];

}

int helperLeft = low;

int helperRight = middle + 1;

int current = low;

/\* Iterate through helper array. Compare the left and right

\* half, copying back the smaller element from the two halves

\* into the original array. \*/

while (helperLeft <= middle && helperRight <= high) {

if (helper[helperLeft] <= helper[helperRight]) {

array[current] = helper[helperLeft];

helperLeft++;

} else { // If right element is smaller than left element

array[current] = helperfhelperRight}

current++;

}

/\* Copy the rest of the left side of the array into the

\* target array \*/

int remaining = middle - helperLeft;

for (int i = 0; i <= remaining; i++) {

array [current + i] = helper[helperLeft + i]j

}

}

**Bloom Filter**

Instead of placing each element into the Data Structure, the Bloom Filter only stores an array of bytes. For each element added to the Bloom Filter, k bits are set in its array. These bits are typically determined by a hashing function.

To check if an element is within the set, you simply check if the bits that would normally be one for this item are actually one. If they all are one (instead of zero), then the item is within the set. If any of the bits are not one, then the item is not within the set.

With every Data Structure there is definitely a draw back to the Bloom Filter. By using the method above, the Bloom Filter can say an element is within the set when it actually isn’t. False positives are possible in the set, and they depend on several factors, such as:

* The size of the byte array
* The number of bits (k) set per element
* The number of items in the set

By tweaking the above values, you can easily get the false positive probability to respectable levels while still saving a large amount of space.

After I discovered the Bloom Filter, I went looking for an implementation in Java. Sadly, a standard implementation doesn’t exist! So, I wrote a quick and simple version of the Bloom Filter for Java. You can find the [source code](https://github.com/ProMobile/BloomFilter)on GitHub.

**Bit Manipulation**

int updateBits(int n, int m, int i, int j) {

/\* Create a mask to clear bits i through j in n

/\* EXAMPLE: i = 2, j = 4. Result should be 11100011.

\* For simplicity, we'll use just 8 bits for the example.

\*/

int allOnes = ~0; // will equal sequence of all Is

// is before position j, then 0s. left = 11100000

int left = allOnes « (j + 1);

// I's after position i. right = 00000011

int right = ((1 « i) - 1);

// All Is, except for 0s between i and j. mask = 11100011

int mask = left | right;

/\* Clear bits j through i then put m in there \*/

int n\_cleared = n & mask; // Clear bits j through i.

int m\_shifted = m « i; // Move m into correct position.

return n\_cleared | m\_shifted; // OR them, and we're done!

}

**OOP – In-Memory File System**

public abstract class Entry {

protected Directory parent;

protected long created;

protected long lastupdated;

protected long lastAccessed;

protected String name;

public Entry(String n, Directory p) {

name = n;

parent = p;

created = System.currentTimeMillis();

lastupdated = System. currentTimeMillisQ;

lastAccessed = System. currentTimeMillisQ;

}

public boolean delete() {

if (parent == null) return false;

return parent.deleteEntry(this);

}

public abstract int size();

public String getFullPath() {

if (parent == null) return name;

else return parent.getFullPathQ + "/" + name;

}

/\* Getters and setters. \*/

public long getCreationTime() { return created; }

public long getLastUpdatedTime() { return lastupdated; }

public long getLastAccessedTime() { return lastAccessed; }

public void changeName(String n) { name = n; }

public String getName() { return name; }

}

public class File extends Entry {

private String content;

private int size;

public File(String n, Directory p, int sz) {

super(n, p);

size = sz;

}

public int size() { return size; }

public String getContents() { return content; }

public void setContents(String c) { content = c; }

}

public class Directory extends Entry {

protected ArrayList<Entry> contents;

public Directory(String n, Directory p) {

super(n, p);

contents = new ArrayList<Entry>();

}

public int size() {

int size = 0;

for (Entry e : contents) {

size += e.sizeQ;

}

return size;

}

public int numberOfFiles() {

int count = 0;

for (Entry e : contents) {

if (e instanceof Directory) {

count++; // Directory counts as a file

Directory d = (Directory) e;

count += d.numberOfFiles();

} else if (e instanceof File) {

count++;

}

}

return count;

}

public boolean deleteEntry(Entry entry) {

return contents.remove(entry);

}

public void addEntry(Entry entry) {

contents.add(entry);

}

protected ArrayList<Entry> getContents() { return contents; }

}