Priority Queue

Recall the definition of a queue. It is a collection where we remove the element that has been in the collection for the longest time. Alternatively stated, we remove the element that first entered the collection. A natural way to implement such a queue was using a linear data structure, such as a linked list or a (circular) array.

A *priority queue* is a different kind of queue, in which the next element to be removed is defined by a priority, which is a more general criterion. For example, in a hospital emergency room, patients are treated not in a first-come first-serve basis, but rather by the urgency of the case. To define the next element to be removed, it is necessary to have some way of comparing any two objects and deciding which has greater priority. Once a comparison method is chosen for determining priority, *the next element to be removed is the one with greatest priority.* Heads up: with priority queues, one typically assigns low numerical values to high priorities. Think "my number one priority", "my number 2 priority", etc.

One way to implement a priority queue of elements (often called *keys*) is to maintain a sorted list. This could be done with a linked list or array list. Each time a element is added, it would need to be inserted into the sorted list. If the number of elements were huge, however, then this would be an inefficient representation since in the worst case the adds and removes would be $O(n)$.

Heaps

The usual way to implement a priority is to use a data structure called a *heap.* To define a heap, we first need to define a complete binary tree. We say a binary tree of height h is *complete* if every level l less than h has the maximum number (2^l) of nodes, and in level h all nodes are as far to the left as possible. A *heap* is a complete binary tree, whose nodes are comparable and satisfy the property that *each node is less than its children.* (To be precise, when I say that the nodes are comparable, I mean that the *elements* are comparable, in the sense that they can all be ordered – recall the *Comparable* interface in Java, presented in lecture 16) This is the default definition of a heap, and is sometimes called a *min heap.*

A *max heap* is defined similarly, except that the element stored at each node is greater than the elements stored at the children of that node. Unless otherwise specified, we will assume a min heap in the next few lectures. Note that it follows from the definition that the smallest element in a heap is stored at the root.

As with stacks and queues, the two main operations we perform on heaps are `add` and `remove`.

add

To add an element to a heap, we create a new node and insert it in the next available position of the complete tree. If level h is not full, then we insert it next to the rightmost leaf. If level h is full, then we start a new level at height $h + 1$.

Once we have inserted the new node, we need to be sure that the heap property is satisfied. The problem could be that the parent of the node is greater than the node. This problem is easy to solve. We can just swap the element of the node and its parent. We then need to repeat the

same test on the new parent node, etc, until we reach either the root, or until the parent node is less than the current node. This process of moving a node up the heap, is often called "upheap".

```
add(element){
   cur = new node at next leaf position
   cur.element = element
   while (cur != root) && (cur.element < cur.parent.element){
       swapElement(cur, parent)
       cur = cur.parent
   }
}
```

You might ask whether swapping the element at a node with its parent's element can cause a problem with the node's sibling (if it exists). No, it cannot. Before the swap, the parent is less than the sibling.[1] So if the current node is less than its parent, then the current node must be less than the sibling too. So, swapping the node's element with its parent's element preserves the heap property with respect to the node's current sibling.

For example, suppose we have a heap with two elements `e` and `g`. Then we add an element to the `*` position below and we find that `* < e`. So we swap them. But if `* < e` then `* < g`.

```
  e
/   \
g    *
```

Here is a bigger example. Suppose we add element `c` to the following heap.

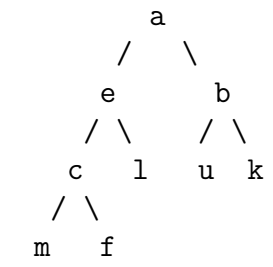
```
          a
        /   \
       e      b
      / \    / \
     f   l  u  k
    /
   m
```

We add a node which is a sibling to `m` and assign `c` as the element of the new node.

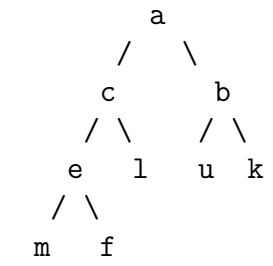
```
          a
        /   \
       e      b
      / \    / \
     f   l  u  k
    / \
   m   c
```

Then we observe that `c` is less than `f`, the element of its parent, so we swap `c,f` to get:

[1] Here I say that one node is less than another, but what I really mean is that the element at one node is less than the element at the other node.



Now we continue up the tree. We compare c with its new parent's element e, see that the elements need to be swapped, and swap them to get:



Again we compare c to its parent. Since c is greater than a, we stop and we're done.

removeMin

Next, let's look at how we remove elements from a heap. Since the heap is used to represent a priority queue, we remove the minimum element, which is the root.

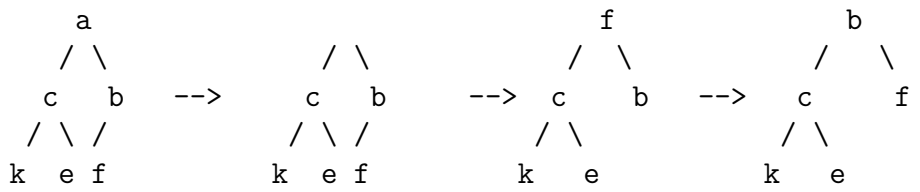
How do we fill the hole that is left by the element we removed ? We first copy the last element in the heap (the rightmost element in level h) into the root, and delete the node containing this last element. We then need to manipulate the elements in the tree to preserve the heap property that each parent is less than its children.

We start at the root, which contains an element that was previously the rightmost leaf in level h. We compare the root to its two children. If the root is greater than at least one of the children, we swap the root with the smaller child. Moving the smaller child to the root does not create a problem with the larger child, since the smaller child is smaller than the larger child.

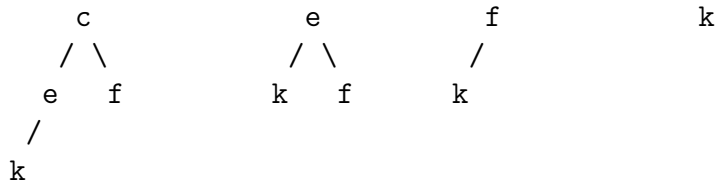
```
removeMin(){                // returns smallest element
   tmp = root.element
   remove last leaf node and put its element into the root
   cur = root
   while ((cur has a left child) and
          ((cur.element > cur.left.element) or
           (cur has a right child and cur.element > cur.right.element)))
       minChild = child with the smaller element
       swapElement(cur, minChild)
       cur = minChild
   }
   return tmp
}
```

The condition in the while loop is rather complicated, and you may have just skipped it. Don't. There are several possible events that can happen and you need to consider each of them. One is that the current node has no children. In that case, there is nothing to do. The second is that the current node has one child, in which case it is the left child. The issue here is that the left child might be smaller. The third is that the current node might have two children. In that case, one of these two children *has to be* smaller than the current node. (See Exercises.)

Here is an example:

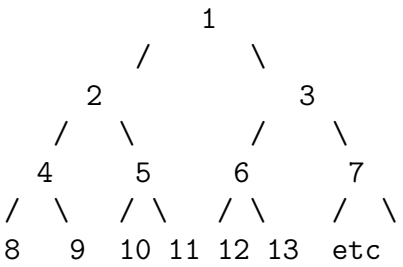


If we apply `removeMin()` again and again until all the elements are gone, we get the following sequence of heaps with elements removed in the following order: `b, c, e, f, k`.



Implementing a heap using an array

A heap is defined to be a complete binary tree. If we number the nodes of a heap by a level order traversal and start with index 1, rather than 0, then we get an indexing scheme as shown below.



These numbers are NOT the elements stored at the node. Rather we are just numbering the nodes so we can index them.

The idea is that an array representation defines a simple relationship between a tree node's index and its children's index. If the node index is i, then its children have indices $2i$ and $2i + 1$. Similarly, if a non-root node has index i then its parent has index $i/2$.

add(element)

Suppose we have a heap with `k` elements which is represented using an array, and now we want to add a `k+1`-th element. Last lecture we sketched an algorithm doing so. Here I'll re-write that algorithm using the simple array indexing scheme. Let `size` be the number of elements in the heap.

These elements are stored in array slots 1 to `size`, i.e. recall that slot 0 is unused so that we can use the simple relationship between a child and parent index.

```
add(element ){
  size = size + 1          // number of elements in heap
  heap[ size ] = element   // assuming array has room for another element
  i = size

//  the following is sometimes called "upHeap"  -- see below

  while (i > 1  and heap[i] < heap[ i/2 ]){
    swapElements( i, i/2 )
    i = i/2
  }
}
```

Example

Suppose we have a heap with eight characters and we add one more, a `c`.

```
1   2   3   4   5   6   7   8   9
-------------------------------------
a   e   b   f   l   u   k   m   c
a   e   b   c   l   u   k   m   f  <----  c swapped with f (slots 9 & 4)
a   c   b   e   l   u   k   m   f  <----  c swapped with e (slots 4 & 2)
```

At the end of last lecture we showed how a heap could be represented using an array, and we rewrote the `add` method using the array representation instead of the binary tree representation. Today we will examine how to build a heap by repeatedly calling the `add` method on a list of elements, and we will analyze the time complexity of building a heap. We will then review the `removeMin` method and rewrite it in terms of the array representation. Finally we will put this all together and show how to sort a list of elements using a heap – called *heapsort*.

Building a heap

We can use the `add` method to build a heap as follows. Suppose we have a list of `size` elements and we want to build a heap.

```
buildHeap(list){
   create an empty heap
   for (k = 0; k < list.size; k++)
        heap.add( list[k] )
   return heap
}
```

We can write this in a slightly different way.

```
buildHeap(list){
  create an array with list.size+1 slots
   for (k = 1; k <= list.size; k++){
       heap[k] = list[k-1]       //  say list indices are 0, .. ,size-1
       upHeap(heap, k)
   }
}
```

where `upHeap(heap, k)` is defined as follows.

```
upHeap(heap, k){
    i = k
    while (i > 1  and heap[i] < heap[ i/2 ]){
       swapElements( i, i/2 )
       i = i/2
    }
}
```

Are we sure that the buildHeap() method indeed builds a heap? Yes, and the argument is basic mathematical induction. Adding the first element gives a heap with one element. If adding the first k elements results in a heap, then adding the $k+1$-th element also results in a heap since the upHeap method ensures this is so.

Time complexity

How long does it take to build a heap in the best and worst case? Before answering this, let's recall some notation. We have seen the "floor" operation a few times. Recall that it rounds down to the nearest integer. If the argument is already an integer then it does nothing. We also can define the ceiling, which rounds up. It is common to use the following notation:

- $\lfloor x \rfloor$ is the largest integer that is less than or equal to x. $\lfloor\ \rfloor$ is called the *floor* operator.
- $\lceil x \rceil$ is the smallest integer that is greater than or equal to x. $\lceil\ \rceil$ is called the *ceiling* operator.

Let i be the index in the array representation of elements/nodes in a heap, then i is found at level *level* in the corresponding binary tree representation. The level of the corresponding node i in the tree is such that

$$2^{level} \leq i < 2^{level+1}$$

or

$$level \leq \log_2 i < level + 1,$$

and so

$$level = \lfloor \log_2 i \rfloor.$$

We can use this to examine the best and worst cases for building a heap.

In the best case, the node i that we add to the heap satisfies the heap property immediately, and no swapping with parents is necessary. In this case, building a heap takes time proportional to the number of nodes n. So, best case is $O(n)$.

What about the worst case? Since $level = \lfloor \log_2 i \rfloor$, when we add element i to the heap, in the *worst case* we need to do $\lfloor \log_2 i \rfloor$ swaps up the tree to bring element i to a position where it is less than its parent, namely we may need to swap it all the way up to the root. If we are adding n nodes in total, the worst case number of swaps is:

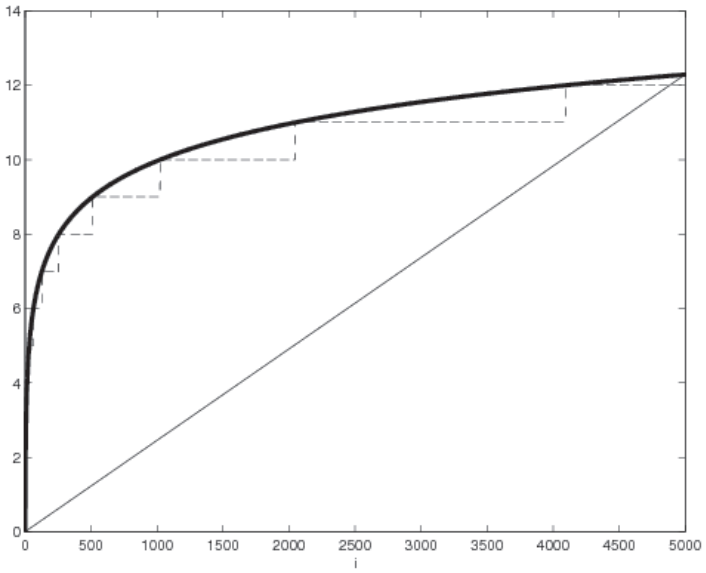
$$t(n) = \sum_{i=1}^{n} \lfloor \log_2 i \rfloor$$

To visualize this sum, consider the plot below which show the functions $\log_2 i$ (thick) and $\lfloor \log_2 i \rfloor$ (dashed) curves up to $i = 5000$. In this figure, $n = 5000$.

The area under the dashed curve is the above summation. It should be visually obvious from the figures that

$$\frac{1}{2} n \log_2 n < t(n) < n \log_2 n$$

where the left side of the inequality is the area under the diagonal line from (0,0) to $(n, \log_2 n)$ and the right side $(n \log_2 n)$ is the area under the rectangle of height $\log_2 n$. From the above inequalities, we conclude that in the worst of building a heap is $O(n \log_2 n)$.



removeMin

Next, recall the `removeMin()` algorithm from last lecture. We can write this algorithm using the array representation of heaps as follows.

```
removeMin(){
    element = heap[1]            //   heap[0] not used.
    heap[1] = heap[size]
    heap[size] = null
```

```
    size = size - 1
    downHeap(1, size)       //   see next page
    return element
}
```

This algorithm saves the root element to be returned later, and then moves the element at position `size` to the root. The situation now is that the two children of the root (node 2 and node 3) and their respective descendents each define a heap. But the tree itself typically won't satisfy the heap property: the new root will be greater than one of its children. In this typical case, the root needs to move down in the heap.

The `downHeap` helper method moves an element from a starting position in the array down to some maximum position in the heap. I will use this helper method in a few ways in this lecture.

```
downHeap(start,maxIndex){     //  move element from starting position
                              //  down to at most position maxIndex
   i = start
   while (2*i <= maxIndex){              // if there is a left child
      child = 2*i
      if (child < maxIndex) {           // if there is a right sibling
         if (heap[child + 1] < heap[child])
                                        //  if rightchild < leftchild ?
         child = child + 1
      }
      if (heap[child] < heap[ i ]){   // swap with child?
         swapElements(i , child)
         i = child
      }
      else  break    // exit while loop
   }
}
```

This is essentially the same algorithm we saw last lecture. What is new here is that (1) I am expressing it in terms of the array indices, and (2) there are parameters that allow the downHeap to start and stop at particular indices.

Heapsort

A heap can be used to sort a set of elements. The idea is simple. Just repeatedly remove the minimum element by calling `removeMin()`. This naturally gives the elements in their proper order.

Here I give an algorithm for sorting "in place". We repeatedly remove the minimum element the heap, reducing the size of the heap by one each time. This frees up a slot in the array, and so we insert the removed element into that freed up slot.

The pseudocode below does exactly what I just described, although it doesn't say "`removeMin()`. Instead, it says to swap the root element i.e. `heap[1]` with the last element in the heap i.e. `heap[size+1-i]`. After `i` times through the loop, the remaining heap has `size - i` elements,

and the last `i` elements in the array hold the smallest `i` elements in the original list. So, we only downheap to index `size - i` .

```
heapsort(list){
   buildheap(list)
   for i = 1 to size-1{
      swapElements( heap[1], heap[size + 1 - i])
      downHeap( 1,  size - i)
   }
   return reverse(heap)
}
```

The end result is an array that is sorted from largest to smallest. Since we want to order from smallest to largest, we must `reverse` the order of the elements.

What is the time complexity of heapsort? If we have n elements, then we have to go through the for loop n times. Each time through, we need to go down at most the height of the tree (and for many elements, we don't have to go that far). Since the height of the tree is about $\log_2 n$, in the worst case, we need to go $O(n \log_2 n)$. The final step, where we reverse the elements in the array only needs to be done once. It is done in $O(n)$ time, by swapping i and $n+1-i$ for $i = 1$ to $\frac{n}{2}$. So heapsort in the worst case is $O(n \log_2 n)$.

Example

The example below shows the state of the array after each pass through the for loop. The vertical line marks the boundary between the remaining heap (on the left) and the sorted elements (on the right).

```
1   2   3   4   5   6   7   8   9
-----------------------------------
a   d   b   e   l   u   k   f   w |
b   d   k   e   l   u   w   f | a   (removed a, put w at root, ...)
d   e   k   f   l   u   w | b   a   (removed b, put f at root, ...)
e   f   k   w   l   u | d   b   a   (removed d, put w at root, ...)
f   l   k   w   u | e   d   b   a   (removed e, put u at root, ...)
k   l   u   w | f   e   d   b   a   (removed f, put u at root, ...)
l   w   u | k   f   e   d   b   a   (removed k, put w at root, ...)
u   w | l   k   f   e   d   b   a   (removed l, put u at root, ...)
w | u   l   k   f   e   d   b   a   (removed u, put w at root, ...)
w   u   l   k   f   e   d   b   a   (removed w, and done)
```

Note that the last pass through the loop doesn't do anything since the heap has only one element left (`w` in this example), which is the largest element. We could have made the loop go from `i = 1 to size - 1`.

What if elements are already sorted (or nearly so)?

You might think the best case for heapsort is that the elements in the list are already in order, since in that case the buildheap step is $O(n)$ rather than $O(n \log_2 n)$, i.e. there are no swaps in the build heap step since each element is compared to its parent and found to be greater than its parent.[2] However, the $n*$downHeap part of the algorithm would be as slow as possible since when the elements are put into the root they greater than all the remaining elements and will need to be downheaped all the way to a leaf.

What if the elements are in the opposite order to begin with? Will heapsort perform better than $O(n \log_2 n)$ in this case? No, it won't. The buildheap step will take time $O(n \log_2 n)$ since each element that is added needs to be swapped all the way to the root.

Discussion...

Why is quicksort in practice preferred over heapsort? First, one needs to note that real quicksort implementations do not choose the last element of the list as the pivot. The reason is that choosing the last element would lead to $O(n^2)$ performance if the list is already sorted or nearly so, and this case does come up in practice sometimes. For example, a better pivot choice is to take the first, middle, and last elements of the list, and choose the median value of these three. If the list is sorted forward or sorted backwards to begin with, then choosing the "median of three" as it is called will split in the middle (yipeee!). And if the list has random order, then the median of three will be more likely to be close to the actual median than if one just chose one of the three. Of course, determining which of these three chosen ones *is* the median requires a bit of extra work for every split, and so it increases the constant factor at each step of the recursion.

This is related to a question that was asked in class: why doesn't heapsort or quicksort just check at the start if the list is already sorted, which would take time only $O(n)$. Yes, you could do that. And it might make sense to do that if this case did arise in practice often enough. But if this case arises only rarely, then this would be extra work that you need to do every time, but that would benefit you rarely.

Maps

The last few lectures, we have looked at ways of organizing a set of elements that are comparable, namely binary search trees and heaps. The goal there was to be able to access elements quickly, rather than having to search through all elements to find the one we want.

Today we will begin looking at ways of organizing things which doesn't require that they are comparable. We will specifically consider how to represent *maps*. You are familiar with maps already. In high school math and in Calculus and linear algebra, you have seen functions that go from (say) $\Re^n$ to $\Re^m$. To be mathematically precise, a map is a set of ordered pairs $\{(x, f(x))\}$ where x belongs to some set called the *domain* of the map, and $f(x)$ belongs to a set called the *co-domain*. The word *range* is used specifically for the set $\{f(x) : (x, f(x)) \text{ is in the map}\}$. That is, some values in the co-domain might not be reached by the map.

[2] In the lecture itself, I rushed through this case and mistakenly said that the buildheap part would be $O(n \log_2 n)$ in this case.

Maps as (key,value) pairs

You are also familiar with the idea of maps in your daily life. You might have an address book which you use to look up home or business addresses, telephone numbers, or email addresses. You index this information with a name. So the mapping is from name to address/phone/email. A related example is "Caller ID" on your phone. Someone calls from a phone number (the index) and the phone tells you the name of the person. Many other traditional examples are social security number or health care number or student number for the index, which maps to a person's employment record, health file, or student record, respectively.

We will use the following definitions and notation for maps. Suppose we have two sets: a set of keys K, and a set of values V. A *map* is a set of ordered pairs

$$M = \{(k, v) : k \in K, v \in V\} \subseteq K \times V.$$

The pairs are called *entries* in the map.

A map cannot just be any set of (key, value) pairs, however. Rather, for any key $k \in K$, there is *at most* one value v such that (k, v) is in the map. We allow two different keys to map to the same value, but we do not allow one key to map to more than one value. Also note that not all elements of K need to be keys in the map.

For example, let K be the set of integers and let V be the set of strings. Then the following is a map:

$$\{(3, \texttt{cat}), (18, \texttt{dog}), (35446, \texttt{meatball}), (5, \texttt{dog}), \}$$

whereas the following is not a map,

$$\{(3, \texttt{cat}), (18, \texttt{dog}), (35446, \texttt{meatball}), (5, \texttt{dog}), (3, \texttt{fish})\}$$

because the key 3 has two values associated with it.

Map ADT

The basic operations that we perform on a map are:

- put(key, value) – this adds a new entry to the map

  What if the map previously contained a mapping for the key? As we see, for a Java `Map` interface, the policy is that the old value is replaced by the new value, and the old value is returned. (If the map didn't contain that key, then `put` returns null.)

- get(key) – this gets the entry (key, value).

  What if the given key did not have an entry in the map? The Java `Map` policy is that `get` would return null.

- remove(key) – this removes the entry (key, value)

  The Java `Map` policy is that it return the value if the key was present, and it returns null if the key wasn't present.

There are other `Map` methods such as `contains(key)` or `contains(value)` as well but the above are the main ones.

Map data structures

How can we represent a map using data structures that we have seen in the course? We might have a key type `K` and a value type `V` and we would like our map data structure to hold a set of object pairs $\{ (k, v) \}$, where k is of type `K` and v is of type `V`. We could use an arraylist or a linked list of entries, for example. This would mean that the operations defined above would be slow in the worst case, however, namely the worst case would be $\Theta(n)$ if the map has n entries.

What if we made a stronger assumption, namely what if the keys of map were comparable? In this case, to organize the entries of the map, we could use a sorted arraylist or a binary search tree. If we used a sorted arraylist, then we could find a key in $O(\log n)$ steps, where n is the number of pairs in the map. Each slot of the sorted arraylist would hold a map entry. We would use binary search to find the entry, based on the key. However, with a sorted array it can be relatively slow to `put` or `remove` a (key,value) pair, namely in the worst case it is $\Theta(n)$.

We could instead use a binary search tree (BST) to store the (k, v) pairs, namely we index by comparing keys. Although the binary search trees we have covered have $\Theta(n)$ worst case behavior, I have told you that there are fancier versions of binary search trees that are balanced that allow adding, removing, finding in $O(\log n)$ time. You'll learn about these in COMP 251.

What about a heap? A heap would not be an appropriate data structure for a map because it only allows you to efficiently get or remove the minimum element. However, for a general `get` operation, a heap would be $\Theta(n)$, since one would have to traverse the entire heap. Note that the heap is represented using an array, so this would just be a loop through the elements of the array.

Another special case to consider is that the keys `K` are positive integers in a small range. In this case, we could just use an array and have the key be an index into the array and the value be the thing stored in the array. Note that this typically will *not* be an arraylist, since there may be gaps in the array between entries. Using an array would give us access to map entries in $O(1)$ time. However, this would only work well if the integer values are within a small range. For example, if the keys were social insurance numbers (in Canada), which have 9 digits, we would need to define an array of size 10^9 which is not feasible since this is too big.

Despite this being infeasbile, let's make sure we understand the main idea here. Suppose we are a company and we want to keep track of employee records. We use social insurance number as a key for accessing a record. Then we could use a (unfeasibly large) array whose entries would be of type `Employee`. That is, you would use someone's social insurance number to index directly into the array, and that indexed array slot would hold a reference to an `Employee` object associated with that social insurance number. Of course this would only retrieve a record if there were an an `Employee` with that social insurance number; otherwise the reference would be null and the `find` call would return null.

For the rest of today, we consider the case that the keys map to a set of integers, without requiring these integers to be a small range. Next lecture we will address the problem of mapping to a small range of integers and treating these integers as an index into an array.

Example: Object.hashCode()

Let's jump right in and consider a map that you may be less comfortable with at this point but which is hugely useful when programming in Java. When a Java programming is running, every object is located somewhere in the memory of the Java Virtual Machine (JVM). This location is

called an *address*. In particular, each Java object has a unique 32 bit number associated with it which by default is the number returned when the object calls `hashCode()` method. (What exactly this 32 bit number means may depend on the implementation of the JVM. We will just assume that the number is the object's (starting) address in the JVM.) Since different objects must be non-overlapping in memory, it follows that they have different addresses. In particular, when the object's `hashCode()` is the object's address, the expression "`obj1 == obj2`" means the same thing as "`obj1.hashCode() == obj2.hashCode()`". The statement can be either true or false, depending on whether the variables `obj1` and `obj2` reference the same object or not.

Example: String.hashCode()

The above discussion about the `hashCode()` method only considers the default implementation. Some classes override this default `hashCode()` method. For example, each String object is also located somewhere in memory but the String class uses a different `hashCode()` method.

To define the `hashCode()` method of the `String` class, let's first consider a simple map from `String` to positive integers. Suppose s is a string which consists of characters $s[0]s[1]...s[s.length-1]$. Consider the function

$$h(s) = \sum_{i=0}^{s.length-1} s[i]$$

which is just the sum of the codes of the individual characters. Notice that two strings that consist of the same letters but in different order would have the same $h()$ value. For example, "eat", "ate", "tea" all would have the same code. Since the codes of `a, e, t` are 97, 101, and 116, the code of each of these strings would be 97+101+116.

In Java, the `String.hashCode()` method is defined[3] :

$$h(s) = \sum_{i=0}^{s.length-1} s[i]\ x^{s.length-i-1}$$

where $x = 31$. So, for example,

$$h(\text{"}eat\text{"}) = 101 * 31^2 + 97 * 31 + 116$$

and

$$h(\text{"}ate\text{"}) = 97 * 31^2 + 116 * 31 + 101$$

Thus, here we have an example of how strings that have the same letters do not have the same hashCode.

What about if two strings have the same hashcode? Can we infer that the two strings are equal? No, we cannot. I will include this as an exercise so that you can see why.

[3]You might wonder why Java uses the value $x = 31$. Why not some other value? There are explanations given on the website stackoverflow, but I am not going to repeat them here. Other values would work fine too.

ASIDE: Horner's rule

Suppose you wish to evaluate a polynomial

$$h(s) = \sum_{k=0}^{N} a_k \; x^k$$

It should be obvious that you don't want to separately calculate $x^2, x^3, x^4, ..., x^N$ since there would be redundancies in doing so. We only should use $O(N)$ multiplications, not $O(N^2)$. Horner's Rule describes how to do so.

The following example gives the idea of Horner's rule for the case of hashcodes for a `String` object with four characters:

$$s[0] * 31^3 + s[1] * 31^2 + s[2] * 31 + s[3] = ((s[0] * 31^1 + s[1]) * 31 + s[2]) * 31 + s[3]$$

For a `String` object of arbitary length, Horner's rule is the following algorithm :

```
h = 0
for  (i = 0;  i < s.length;  i++)
    h = h*31 +  s[i]
```

Recall the scenario from last lecture: suppose we have a map, that is, a set of ordered pairs $\{(k, v)\}$. We want a data structure such that, given a key k, we can quickly access the associated value v. If the keys were integers in a small range, say $0, 1, \ldots, m-1$, then we could just use an array of size m and the keys could indices into the array. The locations k in the array would correspond to the (integer) keys in the map and the slots in the array would hold references to the corresponding values v.

In most situations, the keys are not integers in some small range, but rather they are in a large range, or the keys are not integers at all – they may be strings, or something else. In the more general case, we define a function – called a *hash function* – that maps the keys to the range $0, 1, \ldots, m-1$. Then, we put the two maps together: the hash function maps from keys to a small range of integer values, and from this small range of integer values to the corresponding values v. Let's now say more about how we make hash functions.

Hash function: hash code followed by compression

Given a space of keys K, a *hash function* is a mapping:

$$h : K \to \{0, 1, 2, m-1\}$$

where m is some positive integer. That is, for each key $k \in K$, the hash function specifies some integer $h(k)$ between 0 and $m-1$. The $h(k)$ values in $0, \ldots, m-1$ are called *hash values*.

It is very common to design hash functions by writing them as a composition of two maps. The first map takes keys K to a large set of integers. The second map takes the large set of integers to a small set of integers $\{0, 1, \ldots, m-1\}$. The first map is called a *hash code*, and the integer chosen for a key k is called the *hash code* for that key. The second mapping is called *compression*. Compression maps the hash codes to *hash values*, namely in $\{0, 1, \ldots, m-1\}$.

A typical compression function is the "mod" function. For example, suppose i is the hash code for some key k. Then, the hash value is i mod m. (Often one takes m to be a prime number, though this is not necessary.) The mod function can be defined for negative numbers, such it returns a value in 0 to $m-1$. However, the Java mod function doesn't work like that. e.g. In Java, the expression "-4 mod 3" evaluates to -1 (rather than 2, which is what one would expect after learning about "modulo arithmetic"). To define a compression function using mod in Java, we need to be a bit more careful since we want the result to be in $\{0, 1, \ldots, m-1\}$.

The Java `hashCode()` method returns an `int`, and `int` values can be either positive or negative, specifically, they are in $\{-2^{31}, ... -1, 0, 1, ...2^{31}-1\}$. (You will learn how this works in COMP 273.) The compression function that is used in Java is

$$| \texttt{hashCode}() \mid \text{mod } m,$$

i.e. gives a number in $\{0, 1, \ldots, m-1\}$, as desired.

To summarize, a hash function is typically composed of two functions:

$$\text{hash function } h : \quad \text{compression} \quad \circ \quad \text{hash code}$$

where $\circ$ denotes the composition of two functions, and

$$\text{hash code} \quad : \quad \text{keys K} \rightarrow \text{integers}$$

$$\text{compression} \quad : \quad \text{integers} \rightarrow \{0, .., \text{m} - 1\}$$

and so

$$h : \text{keys } K \rightarrow \{\text{hash values}\},$$

i.e. the set of hash values is $\{0, 1, \ldots, m-1\}$. Note that a hash function is itself a map!

It can happen that two keys k_1 and k_2 have the same hash value. This is called a *collision*. There are two ways a collision can happen. First, two keys might have the same hash code. Second, two keys might have different hash codes, but these two different hash codes map (compress) to the same hash value. In either case we say that a *collision* has occured. We will deal with collisions next.

Hash map (or Hash table)

Let's return to our original problem, in which we have a set of keys of type K and values of type V and we wish to represent a map M which is set of ordered pairs $\{(k, v)\}$, namely some subset of all possible ordered pairs `K` $\times$ `V`.

[ASIDE: Keep in mind there are a few different maps being used here, and in particular the word "value" is being used in two different ways. The values $v \in$ `V` of the map we are ultimately trying to represent are not the same thing as the "hash values" $h(k)$ which are integers in $0, 1, \ldots, m-1$. Values $v \in$ `V` might be `Employee` records, or entries in an telephone book, for example, whereas hash values are indices in $\{0, 1, \ldots, m-1\}$.]

To represent the (k, v) pairs in our map, we use an array. The number of slots m in the array is typically bigger than the number of (k, v) pairs in the map.

As we discussed above, we say that a collision occurs when two keys map to the same hash value. Since the hash values are indices in the array, a collision leads to two keys mapping to the same slot in the array. For example, if $m = 5$ then hash codes 34327 and 83322 produce a collision since 34327 mod 5 = 2, and 83322 mod 5 = 2, and in particular both map to index 2 in the array.

To allow for collisions, we use a linked list of pairs (k, v) at each slot of our hash table array. These linked lists are called *buckets*.[4] Note that we need to store a linked list of pairs (k, v), not just values v. The reason is that when use a key k to try to access a value v, we need to know which of the v's stored in a bucket corresponds to which k. We use the hash function to map the key k to a location (bucket) in the hash table. We then try to find the corresponding value v in the list. We examine each entry (k, v) and check if the search key equals the key in that entry.

Good vs. bad hash functions

Linked lists are used to deal with collisions. We don't want collisions to happen, but they do happen sometimes. A good hash function will distribute the key/value pairs over the buckets such that, if possible, there is at most one pair per bucket. This is only possible if the number of entries in the hash table is no greater than the number of buckets. We define the *load factor* of a hash table to be the number of entries (k, v) in the table divided by the number of slots in the table (m). A load factor that is slightly less than one is recommended for good performance.

Having a load factor less than one does not guarentee good performance, however. In the worst case that all they keys in the collection hash to the same bucket, then we have one linked list only and access is $O(n)$ where n is the number of entries. This is undesirable obviously. To avoid having such long lists, we want to choose a hash function so that there are few, if any, collisions.

The word *hash* means to "chop/mix up".[5] We are free to choose whatever hash function we want and we are free to choose the size m of the array. So, it isn't difficult to choose a hash function that performs well in practice, that is, a hash function that keeps the list lengths short. In this sense, one typically regards hash tables as giving $O(1)$ access. To prove that the performance of hash tables really is this good, one needs to do some math that is beyond COMP 250.

As an example of a good versus bad hash function, consider McGill student ID's which are 9 digits. Many start with the digits 260. If we were to have a hash table of size say $m = 100,000$, then it would be bad to use the first five digits as the hash function since most students IDs would map to one of those two buckets. Instead, using the last five digits of the ID would be better.

Java `HashMap` and `HashSet`

In Java, the `HashMap<K,V>` class implements the hash map that we have been describing. The `hashCode()` method for the key class `K` is composed with the "mod m" (and absolute value) compression function where m is the capacity of an underlying array, and it is an array of linked lists. The linked lists hold `(K,V)` pairs. Have a look at the Java API to see some of the methods and their signatures: `put` , `get`, `remove`, `size`, `containsKey`, `containsValue`, and think of how these might be implemented.

[4] Storing a linked list of (k, v) pairs in each hash bucket is called *chaining*.

[5] It should not be confused with the # symbol which is often the "hash" symbol i.e. hash tag on Twitter, or with other meanings of the word hash that you might have in mind.

In Java, the default maximum load factor for the hash table is than 0.75 and there is a default array capacity as well. The `HashMap` constructor allows you specify the initial capacity of the array, and you can also specify the maximum load factor. If you try to add a new entry – using the `put()` method – to a hash table that would make the load factor go above 0.75 (or the value you specify), then a new hash table is generated, namely there is larger number m of slots and the (key,value) pairs are remapped to the new underlying hash table. This happens "under the hood", similar to what happens with `ArrayList` when the underlying array fills up and so the elements needs to be remapped to a larger underlying array.

Java also has a `HashSet<T>` class. This is similar to a HashMap except that it only holds objects of type `<T>`, not key/value pairs. It uses the `hashCode()` method of the type `T`. Why would this class be useful? Sometimes you want to keep track of a set of elements and you just want to ask questions such as, "does some element e belong to my set, or not?" You can add elements to a set, remove elements from a set, compute intersections of sets or unions of sets." What is nice about the `HashSet` class is that it give you quick access to elements. Unlike a list, which requires $\Theta(n)$ operations to check if an element is present in the worst case, a hash set allows you to check in time $O(1)$ – assuming a good hash function.

Cryptographic hashing

Hash functions comes up in many problems in computer science, not just in hash table data structures. Another common use is in password authentication. For example, when you log in to a website, you typically provide a username or password. On a banking site, you might provide your ATM bank card number or your credit card number, again along with a password. What happens when you do so?

You might think it works like this. The web server has a file that lists all the user names and corresponding passwords (or perhaps a hash map of key-value pairs, namely usernames and passwords, respectively). When you log in, the web server takes your user name, goes to the file, retrieves the password stored there for that username, and compares it with the password that you entered. This is *not* how it works however. The reason is that this method would not be secure. If someone (an employee, or an external hacker) were to break in an steal the file then he would have access to all the passwords and would be able to access everyone's data.

Instead, the way it typically works is that the webserver stores the user name and the *hashed* passwords. Then, when a user logs in, the web server takes the password that the user enters, hashes it (that is, applies the hash function), throws away the password entered by the user, and compares the hashed password to the hashed password that is stored in the file.

Why is then any better? First, notice that if a hacker could access the hashed password, this itself would not be good enough to log in, since the process of logging in requires entering a password, not a hashed password. Second and more interesting, you might wonder if it is possible to compute an *inverse hash map*: given a hashed password, can we compute the original password or perhaps some other password that is hashmapped to the given hashed password. If such a computation were feasible, then this inverse hashmap could be used again to hack in to a user's account.

However, "cryptographic" hashing functions are designed so that such a computation is *not* feasible. These hashing functions have a mixing property that two strings that differ only slightly will map to completely different hash values, and given a hash value, one can say almost nothing about a keyword that could produce that hash function.

A few final observations. First, a cryptographic hashing function does not need to be secret. Indeed there are standard cryptographic hashing functions. One example is MD5 `http://www.miraclesalad.com/webtools/md5.php`. This maps any string to a sequence of 128 bits.[6]

Second, cryptographic hashing is not the same as encryption. With the latter, one tries to encode a string so that it is not possible for someone who is not allowed to know the string to decode the coded string and have the original one again. However, it *should* be possible for someone who *is* allowed to know the original string to be able to decode the encrypted string to get the original message back. So, the main dfference is that with encryption it *is* possible to invert the "hash" function. (It isn't called hashing, when one is doing encryption, but the idea is similar.) You will learn about encryption/decryption if you take MATH 240 Discrete Structures.

[6] If you check out that link, you'll see that the hash values are represented not as 128 bits there, but rather as 32 *hexadecimal* digits.