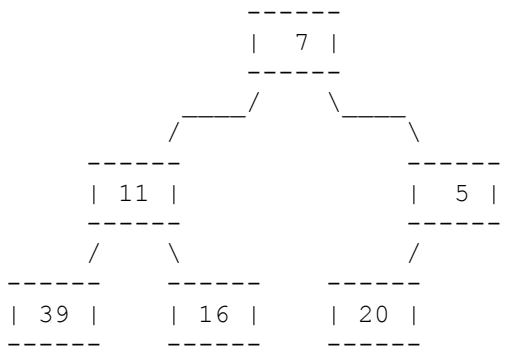
CSCI E-119 Section Notes
Section 10: November 17, 2010 Solutions

1. Heaps

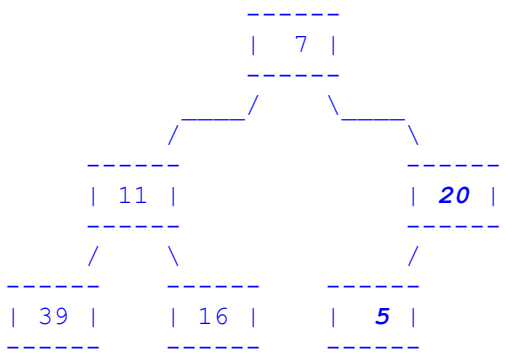
(a) We will now convert the following 6-element array into a valid heap. We start by taking our array and representing it as a heap in the usual manner, with the first element as the root.

0	1	2	3	4	5
7	11	5	39	16	20

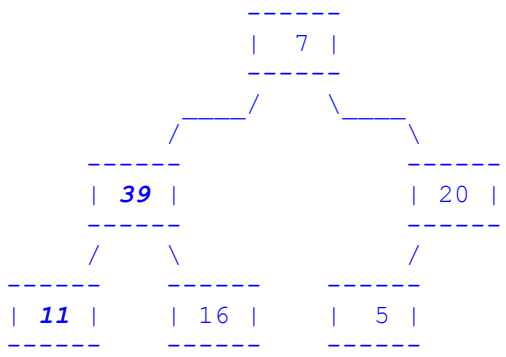
1. This is the corresponding complete tree.



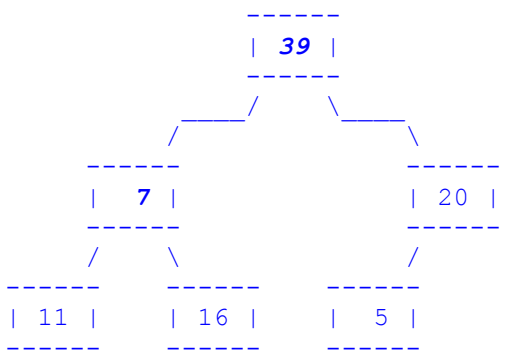
2. We find the parent of the last leaf node, which in our case is the 5 node, and sift it down.



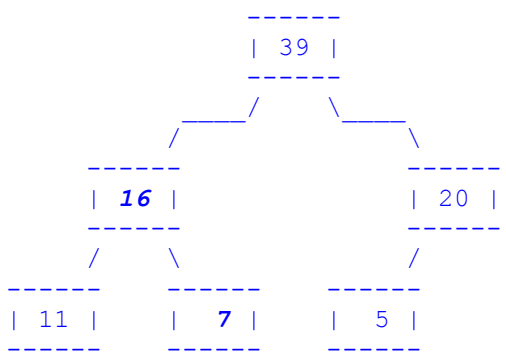
3. We now sift down the other nodes on the same level, which in this case is just the 11 node. Remember to sift correctly by choosing the largest child of a given node, which in this case was the 39 node.



4. Finally, we sift the nodes on the previous level, which in our case is just the root node. In the first step of the sift, the 39 node moves up.



5. Finally, the 16 node moves up. We now have a valid heap.



(b) How long does it take to create a heap from an array?

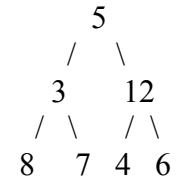
It takes $O(n \log n)$ time to create a heap from an array.

CSCI E-119 Section Notes
Section 10: November 17, 2010 Solutions

2. Heapsort

Here we will run through an example of heapsort. Suppose we have the following numbers in an array: 5 3 12 8 7 4 6. We want to sort the array in ascending order.

If we interpret the array as a complete tree, it looks like this:



(a) What is the first step of heapsort?

We need to turn the array into a heap.

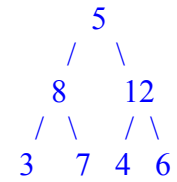
(b) Should we create a min-heap or a max-heap?

We create a max-heap since we will be removing the largest element each time and placing it in its correct location.

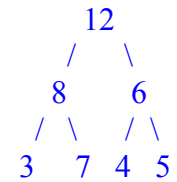
(c) Let's step through turning our array into an actual heap:

Step 1: Sift 12 down. This does not change anything.

Step 2: Sift 3 down:



Step 3: Sift 5 down:



What does the array look like at this point?
12 8 6 3 7 4 5

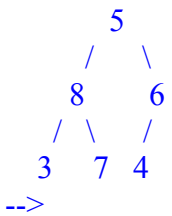
(d) Now that we have made the array into a valid heap, what are the next steps in heapsort?

We need to successively remove the largest element, reheapify, and place the largest element in the correct location in the array. After doing this, we keep removing the largest element and putting it in place until we reach the base case where we have a single element heap.

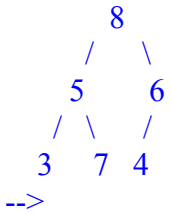
CSCI E-119 Section Notes
Section 10: November 17, 2010 Solutions

(e)Let's step through:

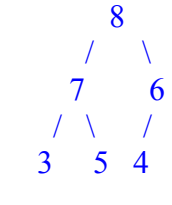
Step 1: Remove 12, and place it in its correct location. Place 5 at the root and sift down:



-->

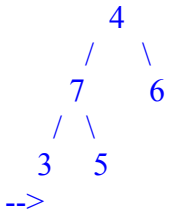


-->

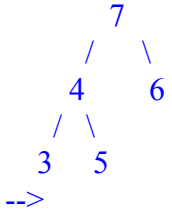


Array: 8 7 6 3 5 4 12

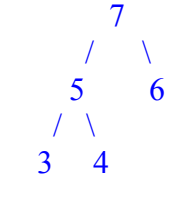
Step 2: Remove 8, Place 4 at the root and sift:



-->

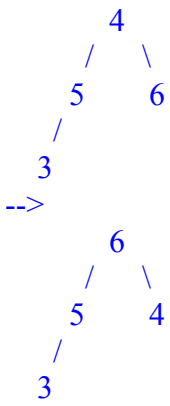


-->



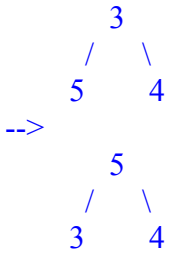
Array: 7 5 6 3 4 8 12

Step 3: Remove 7, Place 4 at the root and sift:



Array: 6 5 4 3 7 8 12

Step 4: Remove 6, Place 3 at the root and sift:



Array: 5 3 4 6 7 8 12

Step 5: Remove 5, Place 4 at the root and sift:

Array: 4 3 5 6 7 8 12

Step 6: Remove 4, Place 3 at the root. Done.

Array: 3 4 5 6 7 8 12

CSCI E-119 Section Notes
Section 10: November 17, 2010 Solutions

3. A* and Greedy Search

Suppose we have the following original configuration of the eight puzzle. We would like to compare the first few iterations of A* search and greedy search:

```
  2 5
4 1 6
8 7 3
```

(a) What is the h(x) for this configuration (The Manhattan Distance)?

h(x) = 1 + 1 + 3 + 1 + 1 + 3 + 0 + 2 = 12

(b) What are the successors of this configuration and their h(x) values? What are the priorities assigned in A* and greedy to each of these configurations?

Successor 1:

```
2 _ 5
4 1 6
8 7 3
```

h(x) = 13, g(x) = 1 [Greedy = -13, A* = -14]

Successor 2:

```
4 2 5
_ 1 6
8 7 3
```

h(x) = 13, g(x) = 1 [Greedy = -13, A* = -14]

In this case, both A* and greedy will assign the successors the same priority.
Our priority queues both look like: 1 2
Therefore, we expand successor 1

(c) What are the successors of successor 1?

Successor 3:

```
2 5 _
4 1 6
8 7 3
```

h(x) = 14, g(x) = 2 [Greedy = -14, A* = -16]

Successor 4:

```
2 1 5
4 _ 6
8 7 3
```

h(x) = 12, g(x) = 2 [Greedy = -12, A* = -14]

Priority Queue for A*: 2 4 3
Priority Queue for Greedy: 4 2 3

(d) After choosing successor 1, what is the next state greedy search looks at? What about A* search?

Looking at the priority queues for the two algorithms, in this step, greedy will next look at successor 4, while A* will next look at successor 2

Similar to the example we have seen in class, we see that A* ranks successors 2 3 4 and equally while Greedy chooses Successor 4 over the others. This is because Greedy does not take into consideration the fact that it requires 2 moves to arrive at successor 4 whereas it only requires 1 move to arrive at successor 2. As a result, at this point, the two algorithms diverge and begin examining and expanding different states.

CSCI E-119 Section Notes
Section 10: November 17, 2010 Solutions

4. Hashing

Suppose we have a 7-element hash table, and we wish to insert following words:
apple, cat, anvil, boy, bag, dog, cup, down

We use hash functions:
h1(key) = index related to first letter of the word ('a' = 0, 'b' = 1, …)
h2(key) = length of the word (ex. h2("apple") = 5)

Let's go through inserting elements using linear probing and count the total length of the probes:

We first insert apple. h1("apple") = 0, and the hash table starts out empty, so we put "apple" in slot 0. Total probe length = 1.
0 | "apple"
1 | _
2 | _
3 | _
4 | _
5 | _
6 | _

Next we take "cat". h1("cat") = 2. slot 2 is empty, so we just place "cat" in index 2. This requires. Total probe length = 1 (from inserting "apple") + 1 (from inserting "cat") = 2:
0 | "apple"
1 | _
2 | "cat"
3 | _
4 | _
5 | _
6 | _

Next we have "anvil". h1("anvil") = 0, which is already occupied. We use linear probing and try to put it in 0 + 1 = 1. This works. Total probe length = 2 + 2 = 4:
0 | "apple"
1 | "anvil"
2 | "cat"
3 | _
4 | _
5 | _
6 | _

Next we have "boy". h1("boy") = 1 which is occupied. We then try 1+1=2 which is also occupied. Then we try 2+1=3 which is not occupied. Total probe length = 4 + 3 = 7:
0 | "apple"
1 | "anvil"
2 | "cat"
3 | "boy"
4 | _
5 | _

6 | _
Next we have "bag". h1("bag") = 1 which is occupied. The next open position is 4. Total probe length = 7 + 4 = 11:
0 | "apple"
1 | "anvil"
2 | "cat"
3 | "boy"
4 | "bag"
5 | _
6 | _
7 | _

Next we have "dog". h1("dog") = 3 which is occupied. The next open position is 5. Total probe length = 11 + 3 = 14:
0 | "apple"
1 | "anvil"
2 | "cat"
3 | "boy"
4 | "bag"
5 | "dog"
6 | _

Next we have "cup". h1("cup") = 2 which is occupied. The next open position is 6. Total probe length = 14 +5 = 19:
0 | "apple"
1 | "anvil"
2 | "cat"
3 | "boy"
4 | "bag"
5 | "dog"
6 | "cup"

Finally, we try to insert "down", but all the slots in the table are filled so overflow occurs. Total probe length = 19 + 7 = 26.

Now let's try quadratic probing:

We first insert apple. The hash table starts out empty and h1("apple") = 0, so we put "apple" in slot 0. Total probe length = 1:

0 | "apple"
1 | _
2 | _
3 | _
4 | _
5 | _
6 | _

Next we insert "cat". h1("cat") = 2 which is not occupied. Total probe length = 1 + 1 = 2:

0 | "apple"
1 | _
2 | "cat"
3 | _
4 | _
5 | _
6 | _

Next we insert "anvil". h1("anvil") = 0 which is occupied. 0 + 1 is not occupied, so we put "anvil" in index 1. Total probe length = 2 + 2 = 4:

0 | "apple"
1 | "anvil"
2 | "cat"
3 | _
4 | _
5 | _
6 | _

Next we insert "boy". h1("boy") = 1 which is occupied. 1 + 1 = 2 is also occupied. 1 + 2^2 = 5 is not occupied, so we put "boy" there. Total probe length = 4 + 3 = 7:

0 | "apple"
1 | "anvil"
2 | "cat"
3 | _
4 | _
5 | "boy"
6 | _

Next we insert "bag". h1("bag") = 1 which is occupied. 1+1 = 2 is occupied, as is 1 + 2^2 = 5. We try 1 + 3^2 = 10 = 3 is not occupied. Total probe length = 7 + 4 = 11.

0 | "apple"
1 | "anvil"
2 | "cat"
3 | "bag"
4 | _
5 | "boy"
6 | _

Next we insert "dog". h1("dog") = 3 which is occupied. 3 + 1 = 4 is not occupied. Total probe length = 11 + 2 = 13:

0 | "apple"
1 | "anvil"
2 | "cat"
3 | "bag"
4 | "dog"
5 | "boy"
6 | _

We next insert "cup". h1("cup") = 2 which is occupied. 2 + 1 = 3 is occupied. 2 + 2^2 = 6 is not occupied so we are successful. Total probe length = 13 + 3 = 16:

0 | "apple"
1 | "anvil"
2 | "cat"
3 | "bag"
4 | "dog"
5 | "boy"
6 | "cup"

Finally, we cannot insert "down" since the table is full. Total probe length = 16 +7 = 23.

Now let's try double hashing:

First we insert "apple". h1("apple") = 0 is not occupied. Total probe length = 1:

0 | "apple"
1 | _
2 | _
3 | _
4 | _
5 | _
6 | _

Next we insert "cat". h1("cat") = 2 is not occupied. Total probe length = 1 + 1 = 2:

0 | "apple"
1 | _
2 | "cat"
3 | _
4 | _
5 | _
6 | _

Next we insert "anvil". h1("anvil") = 0 which is occupied. h2("anvil") = 5. 0 + 5 is not occupied. Total probe length = 2 + 2 = 4:

0 | "apple"
1 | _
2 | "cat"
3 | _
4 | _
5 | "anvil"
6 | _

Next we insert "boy". h1("boy") = 1 which is not occupied. Total probe length = 4 + 1 = 5:

0 | "apple"
1 | "boy"
2 | "cat"
3 | _
4 | _
5 | "anvil"
6 | _

Next we insert "bag". h1("bag") = 1 which is occupied. h2("bag") = 3. 1 + 3 = 4 is unoccupied. Total probe length = 5 + 2 = 7:

0 | "apple"
1 | "boy"
2 | "cat"
3 | _
4 | "bag"
5 | "anvil"
6 | _

Next we insert "dog". h1("dog") = 3 which is not occupied. Total probe length = 7 + 1 = 8:
0 | "apple"
1 | "boy"
2 | "cat"
3 | "dog"
4 | "bag"
5 | "anvil"
6 | _

Next we insert "cup". h1("cup") = 2 which is occupied. h2("cup") = 3. 2 + 3 = 5 is occupied. 2 + 2*3 = 8 = 1 is occupied. 2 + 3*3 = 11 = 4 is occupied. 2 + 4*3 = 14 = 0 is occupied. 2 + 5*3 = 17 = 3 is occupied. 2 + 6*3 = 20 = 6 is not occupied. Total probe length = 8 + 7 = 15:
0 | "apple"
1 | "boy"
2 | "cat"
3 | "dog"
4 | "bag"
5 | "anvil"
6 | "cup"

Again, we cannot insert "down" because the table is full. Total probe length = 15 + 7 = 22.

What do we notice about these counts?
Linear probing requires the most while quadratic probing and double hashing require fewer. This is as expected. The downside of quadratic probing is that sometimes we may not find an open spot even if one exists. In this case, we did not encounter such an instance so quadratic probing performs well and similar to double hashing.

5. The probe() method in our HashTable class
The return value of the probe() method is an integer. In some cases, it represents the index of the key that we're searching for. In other cases, it represents the index of the first empty or removed cell encountered during the search for the specified key.

0	aardvark
1	
2	cat
3	bear
4	
5	dog
6	

The hashtable above has been partially filled using linear probing and the hash function h1 from problem 1. A gray cell indicates that an item has been removed.

One of the items in the table has been inserted incorrectly. Which one, and how do you know?

"dog" is misplaced. Its hash code is 3, because it begins with 'd'. Position 3 may have been filled when it was inserted, which explains why it wasn't put there. However, because position 4 is empty, it should have been inserted there, and it wasn't. Note that position 4 could not have been previously occupied, because it isn't gray.

For each of the keys below, determine:

i. the probe length
ii. the return value of the probe() method

Assume that none of these keys are actually inserted in the table.

a. bear

h1("bear") = 1. Position 1 is a removed cell, so the probe() method takes note of that and continues probing. Position 2 is filled with a different key, so it moves on to position 3, which contains the key we are searching for. Thus, the method returns 3. Probe length = 3 (position 1, 2, and 3).

b. cow

h1("cow") = 2. Position 2 is filled with a different key, so the probe() method moves on to position 3, which is also filled with a different key. Position 4 is empty, so the probe() method breaks out of the while loop and returns 4. Probe length = 3.

c. buffalo

h1("buffalo") = 1. Position 1 is a removed cell, so the probe() method takes note of that and continues probing. Position 2 is filled with a different key, so it moves on to position 3, which is also filled with a different key. Position 4 is empty, so the probe() method breaks out of the while loop. Because it encountered a removed cell (position 1), it returns *its* position, so that a newly inserted value could be put there. Return value = 1. Probe length = 4.

d. giraffe

h1("giraffe") = 6. Position 6 is a removed cell, so the probe() method takes note of that and moves on to position (6 + 1) % 7 = 0, which is filled with a different key, so it moves on to position 1. Position 1 is also a removed cell, but it is not the first one encountered, so the probe() method does not record its position, but moves on to position 2. Position 2 is filled with a different key, so it moves on to position 3, which is also filled with a different key. Position 4 is empty, so the probe() method breaks out of the while loop. It returns the position of the first encountered removed cell. Return value = 6. Probe length = 6.

What is the largest probe length that we could have for this table, regardless of its contents?

7 – the length of the table. After 7 positions, the probe sequence repeats, so the probe() method will give up after trying 7 positions.