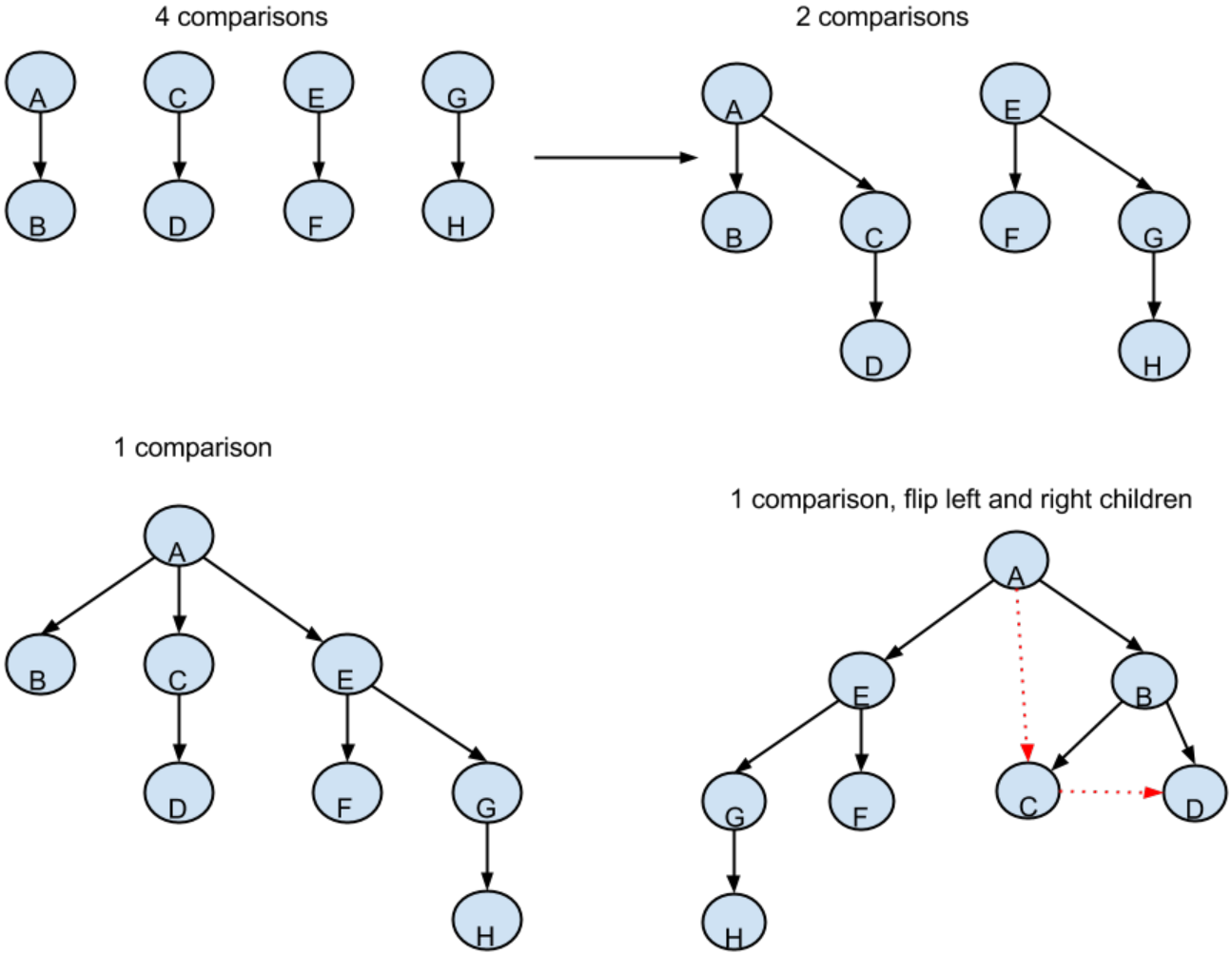
Homework 4 and 5

6.7 b. Show that a heap of eight elements can be constructed in eight comparisons between heap elements.

Tournament of pairwise comparisons


4 comparisons
2 comparisons
1 comparison
1 comparison, flip left and right children
A
B
C
D
E
F
G
H

6.8 Show the following regarding the maximum item in the heap:

a. It must be at one of the leaves.

b. There are exactly N/2 leaves.

c. Every leaf must be examined to find it.

a - Proof by contradiction

Assume that it is not a leaf. Therefore, it must have at least one child. If it has a child, and it is the maximum element, then it must be greater than it’s child. This violates the heap order property, which is a contradiction since the structure is a heap.

6.8 b - Rely on the fact that the tree is complete.

1. A perfect tree of depth k has exactly $2^{k+1} - 1$ nodes.
2. Assume that the heap reaches depth k. Thus
    1. up to level $k - 1$ the tree is perfect (and has $2^k - 1$ nodes there)
    2. on the last level, there are exactly $n - 2^k + 1$ nodes, which are all leaves.
3. Each leaf on the kth level has a parent. Moreover, each two consecutive leaves have the same father (maybe except for the last node, whose father has only one child)
4. Thus, out of the 2^{k-1} nodes at level $k - 1$, $\left\lceil \frac{n-2^k+1}{2} \right\rceil$ are parents, and the rest $2^{k-1} - \left\lceil \frac{n-2^k+1}{2} \right\rceil$ are leaves.
5. The total amount of leaves is

$$n - 2^k + 1 + 2^{k-1} - \left\lceil \frac{n - 2^k + 1}{2} \right\rceil$$

6.8 Show the following regarding the maximum item in the heap:

c. Every leaf must be examined to find it.

Assume you have a heap that is a perfect tree of N nodes. Call it's maximum element *m* Now add N+1 nodes which are all greater than *m.* These values will all end up in the leaves of the heap in the order in which they are inserted. Any of them may be the greatest node in the entire heap. Therefore, the maximum element could be in any leaf.

Given input {4371, 1323, 6173, 4199, 4344, 9679, 1989} and a hash function h(x) = x mod 10, show the resulting:

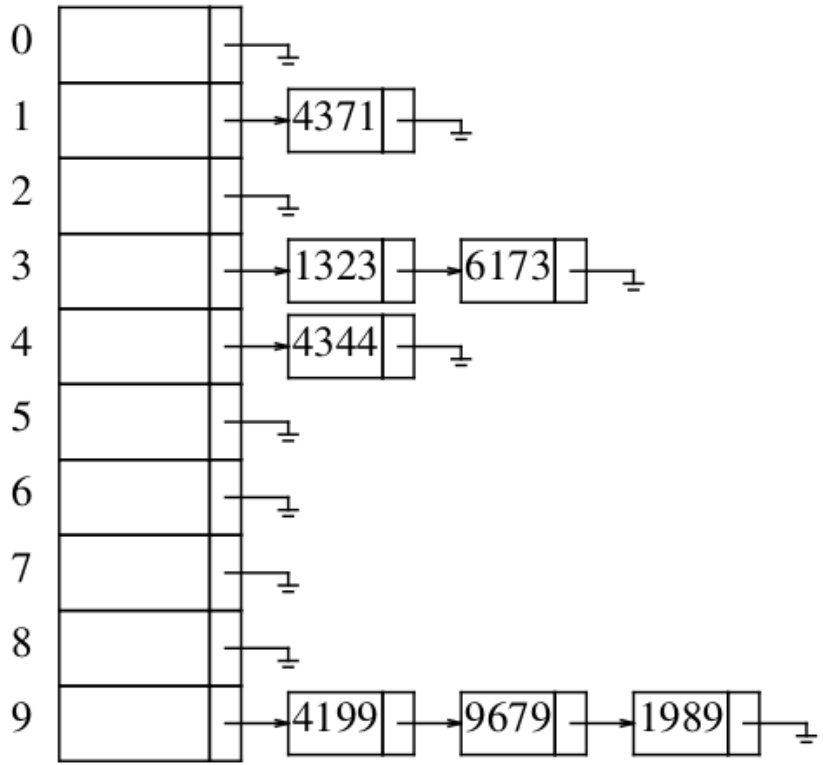
a. Separate chaining hash table.

b. Hash table using linear probing.

c. Hash table using quadratic probing.

d. Hash table with second hash function h2 (x) = 7 − (x mod 7).

(a) On the assumption that we add collisions to the end of the list (which is the easier way if a hash table is being built by hand), the separate chaining hash table that results is shown here.



Given input {4371, 1323, 6173, 4199, 4344, 9679, 1989} and a hash function h(x) = x mod 10, show the resulting:

b. Hash table using linear probing.

0	9679
1	4371
2	1989
3	1323
4	6173
5	4344
6	
7	
8	
9	4199

Given input {4371, 1323, 6173, 4199, 4344, 9679, 1989} and a hash function h(x) = x mod 10, show the resulting:

c. Hash table using quadratic probing.

0	9679
1	4371
2	
3	1323
4	6173
5	4344
6	
7	
8	1989
9	4199

Given input {4371, 1323, 6173, 4199, 4344, 9679, 1989} and a hash function h(x) = x mod 10, show the resulting:

d. Hash table with second hash function h2 (x) = 7 − (x mod 7).

1989 cannot be inserted into the table because $hash_2(1989) = 6$, and the alternative locations 5, 1, 7, and 3 are already taken. The table at this point is as follows:

0	
1	4371
2	
3	1323
4	6173
5	9679
6	
7	4344
8	
9	4199

5.2. Rehash using a table size of 19 and a hash function of h(x) = x mod 19.
When rehashing, we choose a table size that is roughly twice as large and prime, which is 19.

(a) Scanning down the separate chaining hash table, the new locations are 4371 in list 1, 1323 in list 12, 6173 in list 17, 4344 in list 12, 4199 in list 0, 9679 in list 8, and 1989 in list 13.
(b) The new locations are 9679 in bucket 8, 4371 in bucket 1, 1989 in bucket 13, 1323 in bucket 12, 6173 in bucket 17, 4344 in bucket 14 because both 12 and 13 are already occupied, and 4199 in bucket 0.
(c) The new locations are 9679 in bucket 8, 4371 in bucket 1, 1989 in bucket 13, 1323 in bucket 12, 6173 in bucket 17, 4344 in bucket 16 because both 12 and 13 are already occupied, and 4199 in bucket 0.
(d) The new locations are 9679 in bucket 8, 4371 in bucket 1, 1989 in bucket 13, 1323 in bucket 12, 6173 in bucket 17, 4344 in bucket 15 because 12 is already occupied, and 4199 in bucket 0

5.6 The isEmpty routine for quadratic probing has not been written. Can you implement it by returning the expression currentSize==0?

No.

```
public void remove( AnyType x )
{
 int currentPos = findPos( x );
 if( isActive( currentPos ) )
   array[ currentPos ].isActive = false;
}
```

Remove doesn’t change currentSize! If it did, probing might fail.

7.12 What is the running time of heapsort for presorted input?

Heapsort uses at least (roughly) N log N comparisons on any input, so there are no particu-

larly good inputs. Even if the ordering is optimal for constructing the heap, pulling elements off will cause O(N log N) comparisons.

This bound is tight; see the paper by Schaeffer and Sedgewick.

7.35

a) In how many ways can two sorted arrays of N elements be merged?

We have 2 arrays of size A and B of size N which will be sorted into an array C of 2N objects. If we could figure out which N positions the items in A would go into, then there would be exactly 1 way the objects in the B would be added - we could treat B as a queue, then read C from left to right and and deque from B into C whenever an empty cell is encountered. Therefore, we are only free to choose the indices that A will fill, which means there are 2N choose N possible solutions**.**

b) Give a nontrivial lower bound on the number of comparisons required to merge

two sorted lists of N elements, by taking the logarithm of your answer in part (a).

The information-theoretic lower bound is log 2N . Applying Stirling's formula, we

N can estimate the bound as 2N − 1⁄2log N. http://en.wikipedia.org/wiki/Stirling%27s_approximation

7.37 Consider the following algorithm for sorting six numbers:

Sort the first three numbers using Algorithm A.

Sort the second three numbers using Algorithm B.

Merge the two sorted groups using Algorithm C.

Show that this algorithm is suboptimal, regardless of the choices for Algorithms A, B, and C

Algorithms A and B require at least three comparisons based on the information-theoretic lower bound (ceil($\log_2(n!)$) = 3). Algorithm C requires at least five comparisons, based on Exercise 7.35 (b).

So the algorithm requires 11 comparisons; but six elements can be sorted in 10 comparisons

9.2 If a stack is used instead of a queue for the topological sort algorithm in Section 9.2, does a different ordering result? Why might one data structure give a “better” answer?

The following ordering is arrived at by using a queue and assumes that vertices appear on an adjacency list alphabetically. The topological order that results is then

s, G, D, H, A, B, E, I, F, C, t

Assuming the same adjacency list, the topological order produced when a stack is used is

s, G, H, D, A, E, I, F, B, C, t

Because a topological sort using queues processes vertices in the same manner as a breadth-first search, it tends to produce a more natural ordering.

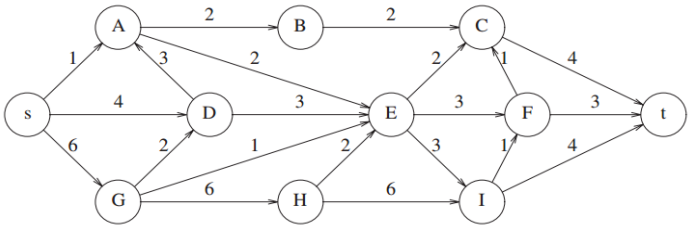


Figure 9.81 Graph used in Exercises 9.1 and 9.11

9.19 If all the edges in a graph have weights between 1 and |E|, how fast can the minimum spanning tree be computed?

Remember that the worst-case running time of Kurskals algorithm is O(|E| log |E|), which is dominated by the heap operations.

Using a heap to find the lowest edge value is not as efficient as the O(N) bucket sort.

The obvious solution using elementary methods is to bucket sort the edge weights in linear time. Then the running time of Kruskal's algorithm is dominated by the union/find operations

9.20 Give an algorithm to find a maximum spanning tree. Is this harder than finding a minimum spanning tree?

Since the minimum spanning tree algorithm works for negative edge costs, an obvious solution is to replace all the edge costs by their negatives and use the minimum spanning tree algorithm. This is obviously exactly the equivalent problem.