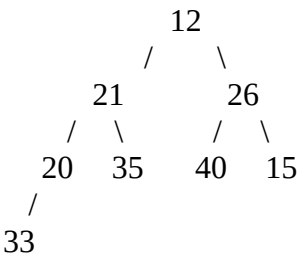
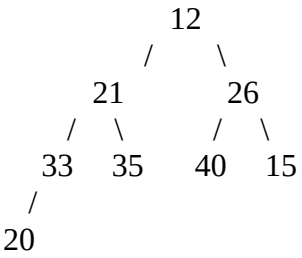
CSCI E-22
PS-5 Solutions

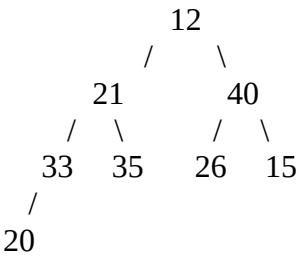
Problem 1
(a) Initial tree:



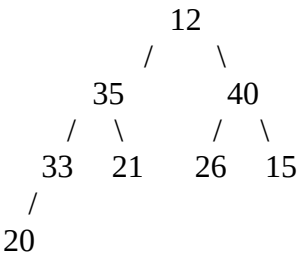
First sift down 20:



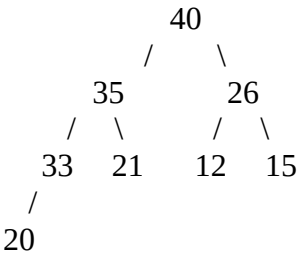
Then sift down 26:



Then sift down 21:

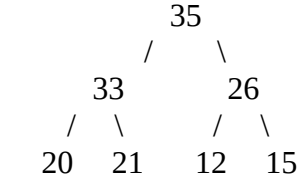


Finally sift down 12, which requires two swaps:



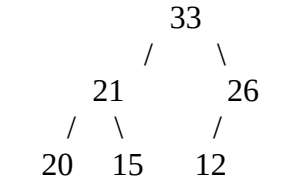
(b) {40, 35, 26, 33, 21, 12, 15, 20}

(c) 1st iteration: remove 40 from the heap, replacing it with 20, which is sifted down to give



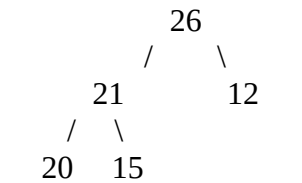
40 is put in place in the last position of the array to give:
{35, 33, 26, 20, 21, 12, 15, **40**}

2nd iteration: remove 35 from the heap, replacing it with 15, which is sifted down to give



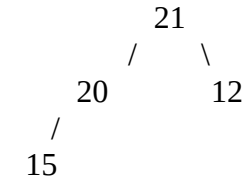
35 is put in place in the second-to-last position of the array to give:
{33, 21, 26, 20, 15, 12, **35**, **40**}

3rd iteration: remove 33 from the heap, replacing it with 12, which is sifted down to give



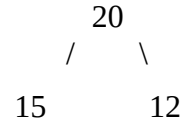
33 is put in place in the third-to-last position of the array to give:
{26, 21, 12, 20, 15, **33**, **35**, **40**}

4th iteration: remove 26 from the heap, replacing it with 15, which is sifted down to give



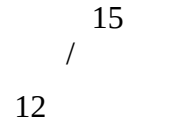
26 is put in place in the fourth-to-last position of the array to give:
{21, 20, 12, 15, **26**, **33**, **35**, **40**}

5th iteration: remove 21 from the heap, replacing it with 15, which is sifted down to give



21 is put in place in the fifth-to-last position of the array to give:
{20, 15, 12, **21**, **26**, **33**, **35**, **40**}

6th iteration: remove 20 from the heap, replacing it with 12, which is sifted down to give



20 is put in place in the sixth-to-last position of the array to give:
{15, 12, **20**, **21**, **26**, **33**, **35**, **40**}

last iteration: remove 15 from the heap, replacing it with 12, which is sifted down to give

15 is put in place in the sixth-to-last position of the array to give:
{12, **15**, **20**, **21**, **26**, **33**, **35**, **40**}

And now the array is fully sorted.

Problem 2

(a) The algorithm overflows at “duck,” at which point the entire hash table is full.

0 ant
1 flea
2 bat
3 cat
4 goat
5 dog
6 bird
7 bison

(b) The table overflows at “ant”. The table looks like:
0
1
2
3 cat
4 goat
5 bird
6 bison
7 dog

(c) The algorithm overflows at “duck,” at which point the entire hash table is full.
0 ant
1 bat
2 flea
3 cat
4 goat
5 bison
6 bird
7 dog

Problem 3
Let h(s) = the value of the heuristic function for state s.

a) h(s) = manh_dist(1 tile) + manh_dist(2 tile) + …
   h(a) = 2 + 3 + 1 + 1 + 3 + 2 + 2 + 2 = 16 priority(a) = **–16**
   h(c) = 2 + 3 + 1 + 1 + 3 + 2 + **3** + 2 = 17 priority(c) = **–17**
   h(f) = 2 + 3 + 1 + 1 + 3 + 2 + 3 + **3** = 18 priority(f) = **–18**
   h(m) = 2 + 3 + 1 + **0** + 3 + 2 + 3 + 3 = 17 priority(m) = **–17**

b) A* uses the same h values, but it also incorporates g(s), which is the depth of s in the tree:
   priority(a) = –1 * (16 + 0) = **–16**
   priority(c) = –1 * (17 + 1) = **–18**
   priority(f) = –1 * (18 + 2) = **–20**
   priority(m) = –1 * (17 + 3) = **–20**

Problem 4
(a) One option is to maintain a sorted descending linked list where the head element is always the largest element. Then, the remove() method merely returns a reference to this head node, and sets the new head of the priority queue to be the next element. We could also use an array-based list sorted in ascending order. Then, the remove() method would return the last item in the list and decrement the number of items.
(b) The insert operation is O(n), because we must maintain the sortedness of the list. In a linked list, we must iterate over the list to find the proper sorted position of the element. In an array-based list, we can find the proper position in O(log n) time using binary search, but we need to shift elements over to make room, which requires O(n) time.

Problem 5

(a) One possible solution:

```
public static boolean findPath(Vertex source, Vertex dest) {
    if (source == null || dest == null)
        throw new IllegalArgumentException();

    if (source == dest)
        return true;

    source.done = true;
    Edge e = source.edges;
    while (e != null) {
        Vertex v = e.end;
        if (!v.done) {
            v.parent = source;
            if (findPath(v, dest))
                return true;
        }
        e = e.next;
    }

    return false;
}
```

(b) This method essentially performs a depth-first traversal of the graph, stopping if it finds the destination. The best-case efficiency would be O(1), if the destination is the same as the source or if the destination is at the front of the source's adjacency list. The worst-case efficiency occurs when the destination is not found, or when it is found at the end of the traversal. This means that the worst-case efficiency is the same as the worst-case for a full traversal: O(V + E) for a sparse graph, and $O(V^2)$ for a dense graph.

Problem 6

(a) L.A., Denver, Seattle, St.Louis, Atlanta, Washington, New York., Boston

(b) When a city is added to the queue, its parent field is set to the just visited neighbor. Thus, New York's parent is St. Louis, and St. Louis's parent is L.A., and the path from L.A. to New York is:

L.A. → St. Louis → New York

(c) L.A., Denver, St. Louis, Atlanta, Washington, New York, Boston, Seattle

(d) A city's parent is given by the second parameter in its recursive call. Thus, New York's parent is Washington, Washington's parent is Atlanta, Atlanta's parent is St. Louis, St. Louis's parent is Denver, and Denver's parent is L.A. Reversing these gives the path from L.A. to New York:

L.A. → Denver → St. Louis → Atlanta → Washington → New York

Problem 7
(L.A., Denver)
(Denver, St. Louis)
(St. Louis, Atlanta)
(Atlanta, Washington)
(Washington, New York)
(New York, Boston)
(L.A., Seattle)

Problem 8
(a)

Atlanta	inf	inf	inf	inf	***2090***			
Boston	inf	inf	inf	inf	*2590*	2590	2590	***2580***
Denver	inf	***800***						
L.A.	**0**							
New York	inf	inf	inf	inf	*2390*	2390	**2390**	
Seattle	inf	*950*	**950**					
St. Louis	inf	*1600*	*1590*	**1590**				
Washington	inf	inf	inf	inf	*2290*	**2290**		

final results: L.A. (0), Denver (800), Seattle (950), St. Louis (1590), Atlanta (2090), Washington (2290), New York (2390), Boston (2580).

(b) The algorithm's initial path to Boston is: L.A. → Denver → St. Louis → Boston
It then replaces it with the final, shorter path: L.A. → Denver → St. Louis → New York → Boston

Problem 9
If we treat the table as an adjacency matrix for a graph, we are essentially trying to find the shortest paths from server #5 to every other server in that graph. Thus, we apply Dijkstra's algorithm starting from server #5, as shown below. Here again, italics show when an estimate changes, and bold type shows when it is finalized.

1	inf	*325*	325	*275*	**275**			
2	inf	*150*	**150**					
3	inf	***120***						
4	inf	*275*	*260*	**260**				
5	***0***							
6	inf	inf	inf	inf	inf	*775*	**775**	
7	inf	inf	inf	inf	inf	inf	*945*	***890***
8	inf	*625*	625	625	*545*	**545**		

Running the algorithm also allows us to determine the following shortest paths from server #5 to each other server:

1: 5 → 2 → 1
2: 5 → 2
3: 5 → 3

4: 5 → 3 → 4
6: 5 → 2 → 1 → 6
7: 5 → 2 → 1 → 6 → 7
8: 5 → 3 → 4 → 8

When routing packets to a given server, server #5 should send them to the first server encountered on the path from 5 to that server:

1: 2
2: 2
3: 3
4: 3
6: 2
7: 2
8: 3

Problem 10

(a) Graph 10-1 is a DAG.

Topological sort: Below is one possible version, in which we consider vertices in alphabetical order on each iteration.

C has no successors, so we mark it as visited and push it onto the stack.
S = {C}

D now has no unvisited successors, so we mark it as visited and push it onto the stack.
S = {D, C}

A now has no unvisited successors, so we mark it as visited and push it onto the stack.
S = {A, D, C}

B now has no unvisited successors, so we mark it as visited and push it onto the stack.
S = {B, A, D, C}

E now has no unvisited successors, so we mark it as visited and push it onto the stack.
S = {E, B, A, D, C}

F now has no unvisited successors, so we mark it as visited and push it onto the stack.
S = {F, E, B, A, D, C}

topological ordering: F, E, B, A, D, C

Other possible answers: F, B, E, A, D, C
F, E, A, B, D, C

(b) Graph 10-2 is *not* a DAG.

Cycles:A → F → B → E → A
A → F → C → B → E → A

Problem 11

Here is the order in which the edges are added by Kruskal's:

1. (Boston, New York)
2. (New York, Washington)
3. (Atlanta, St. Louis)
4. (Atlanta, Washington)
5. (Denver, St. Louis)
6. (Denver, L.A.)
7. (L.A., Seattle)

Problem 12

In adapting Prim's algorithm, we select the largest edge spanning the two subsets, rather than the smallest edge.

Pseudocode for the Prim-based algorithm:

```
let A = {some initial vertex}
let B = {all other vertices}

while (B is not empty) {
    (u, v) = largest edge between a vertex in A and a vertex in B (u is in A, v is in B)
    add (u,v) to maximum spanning tree
    A = A + {v}
    B = B – {v}
}
```