CS 61BL Summer 2021 — Final Exam Review

Monday August 9, 2021

1 Hashing Asymptotics

Here is a video walkthrough of the solutions.

Suppose we set the `hashCode` and `equals` methods of the `ArrayList` class as follows.

```
/* Returns true iff the lists have the same elements in the same ordering */
@Override
public boolean equals(Object o) {
    if (o == null || o.getClass() != this.getClass() || o.size() != this.size()) {
        return false;
    }
    ArrayList<T> other = (ArrayList<T>) o;
    for (int i = 0; i < this.size(); i++) {
        if (other.get(i) != this.get(i)) {
            return false;
        }
    }
    return true;
}

/* Returns the sum of the hashCodes in the list. Assume the sum is a cached instance variable. */
@Override
public int hashCode() {
    return sum;
}
```

(a) Give the best and worst case runtime of `hashContents` in $\Theta(.)$ notation as a function of `N`, where `N` is initial size of the `list`. Assume the length of `set`'s underlying array is `N` and the `set` does **not** resize. Assume the `hashCode` of an `Integer` is itself. Admittedly, the `ArrayList` class does not have the method `removeLast`, but assume it does for this problem, and is implemented the same as in Project 1. Finally, assume `f` accepts two **`int`**s, returns an unknown **`int`**, and runs in constant time.

```
static void hashContents(HashSet<ArrayList<Integer>> set, ArrayList<Integer> list) {
    if (list.size() <= 1) {
        return;
    }
    int last = list.removeLast();
    list.set(0, f(list.get(0), last));
    set.add(list);
    hashContents(set, list);
}
```

Best Case: $\Theta($ $)$, Worst Case: $\Theta($ $)$

Solution:
Best Case: $\Theta(N)$, Worst Case: $\Theta(N^2)$

(b) Continuing from the previous part, how can we define `f` to **ensure** the worst case runtime? How can we define `f` to **ensure** the best case runtime? There may be multiple possible answers.

1. Worst case:

```
int f(int first, int last) {
    return ____________________________;
}
```

Solution:

```
int f(int first, int last) {
    return first + last;
}
```

2. Best case:

```
int f(int first, int last) {
    return ____________________________;
}
```

Solution:

```
int f(int first, int last) {
    return first + last + 1;
}
```

Alternate solution:

```
int f(int first, int last) {
    return first + last - 1;
}
```

2 Sorted Runtimes

Here is a video walkthrough of the solutions.

We want to sort an array of N **unique** numbers in ascending order. Determine the best case and worst case runtimes of the following sorts:

(a) Once the runs in merge sort are of $size <= N/100$, we perform insertion sort on them.

Best Case: $\Theta($ $)$, Worst Case: $\Theta($ $)$

Solution:
Best Case: $\Theta(N)$, Worst Case: $\Theta(N^2)$

Once we have 100 runs of size N/100, insertion sort will take best case $\Theta(N)$ and worst case $\Theta(N^2)$ time. The constant number of linear time merging operations don't add to the runtime.

(b) We can only swap adjacent elements in selection sort.

Best Case: $\Theta($ $)$, Worst Case: $\Theta($ $)$

Solution:
Best Case: $\Theta(N^2)$, Worst Case: $\Theta(N^2)$

The best case and worst case don't change since swapping at most doubles the work each iteration, which produces the same asymptotic runtime as normal selection sort.

(c) We use a linear time median finding algorithm to select the pivot in quicksort.

Best Case: $\Theta($ $)$, Worst Case: $\Theta($ $)$

Solution:
Best Case: $\Theta(N \log(N))$, Worst Case: $\Theta(N \log(N))$

Doing an extra N work each iteration of quicksort doesn't asymptotically change the best case runtime, but it improves the worst case runtime.

(d) We implement heapsort with a min-heap instead of a max-heap. You may modify heapsort but must maintain constant space complexity.

Best Case: $\Theta($ $)$, Worst Case: $\Theta($ $)$

Solution:
Best Case: $\Theta(N \log(N))$, Worst Case: $\Theta(N \log(N))$

While a max-heap is better, we can make do with a min-heap by placing the smallest element at the right end of the list until the list is sorted in **descending order**. Once the list is in descending order, it can be sorted in ascending order with a simple linear time pass.

(e) We run an optimal sorting algorithm of our choosing knowing:

- There are at most N inversions

  Best Case: $\Theta($ $)$, Worst Case: $\Theta($ $)$

Solution:
Best Case: $\Theta(N)$, Worst Case: $\Theta(N)$

Recall that insertion sort takes $\Theta(N + K)$ time, where K is the number of inversions. If K is at most N, then, insertion sort has the best and worst case runtime of $\Theta(N)$. Here is an explanation for why no sorting algorithm can surpass this. Notice for our algorithm to terminate we *either* need to address every inversion or look at every element. Since there are at most N inversions, knowing that we have addressed every inversion would take us at least $\Theta(N)$ time. Looking at every element in the list would also take us $\Theta(N)$ time. In either case, we see the runtime of any sorting algorithm cannot be faster than $\Theta(N)$.

- There is exactly 1 inversion

  Best Case: $\Theta($ $)$, Worst Case: $\Theta($ $)$

  Solution:
  Best Case: $\Theta(1)$, Worst Case: $\Theta(N)$

  The inversion may be the first two elements, in which case constant time is needed. Or, it may involve elements at the end, in which case N time is needed. It can be proven quite simply that no sorting algorithm can achieve a better runtime than above for the best and worst case.

- There are exactly $(N^2 - N)/2$ inversions

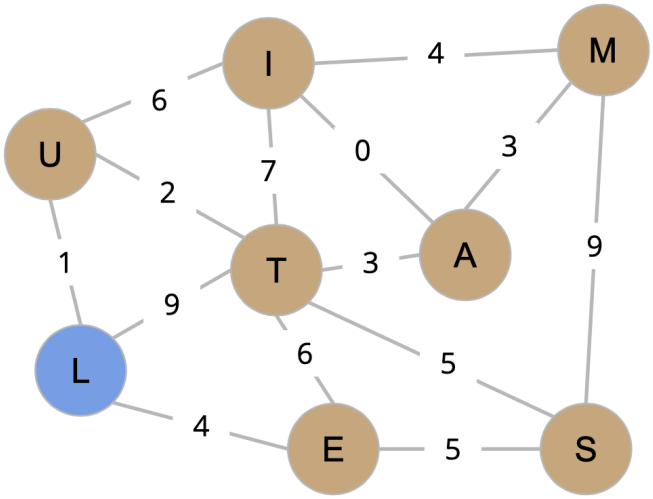
  Best Case: $\Theta($ $)$, Worst Case: $\Theta($ $)$

  Solution:
  Best Case: $\Theta(N)$, Worst Case: $\Theta(N)$

  If a list has $N(N-1)/2$ inversions, it means it is sorted in descending order! So, it can be sorted in ascending order with a simple linear time pass. We know that reversing any array is a linear time operation, so the optimal runtime of any sorting algorithm is $\Theta(N)$.

3 Dijkstra's and A*

Given the graph below, answer the following questions:



(a) What edges are in the shortest paths tree (SPT) starting from **L**?

Solution:
Edges: LU, LE, UT, AT, ST, AM, AI
Here is a video walkthrough of the solutions.

(b) Decreasing **which edge** by 2 changes the SPT from **L**? Assume the SPT tree was created by running Dijkstra's from **L**. There may be more than one correct answer, determine **all**!

Solution:
Edges: UI, IM, ES, EL, AI
Here is a video walkthrough of the solutions.

(c) We will define the heuristic of a vertex v as the shortest distance from v to I. For instance, the heuristic of T is 3.

Given that I is the end vertex, what start vertex would visit the most vertices on one run of A*? Recall that A* terminates after removing the goal. If multiple answers produce the maximum, select all.

Solution:
Vertex: L
Here is a video walkthrough of the solutions.

4 Prim's

Here is a video walkthrough of the solutions.

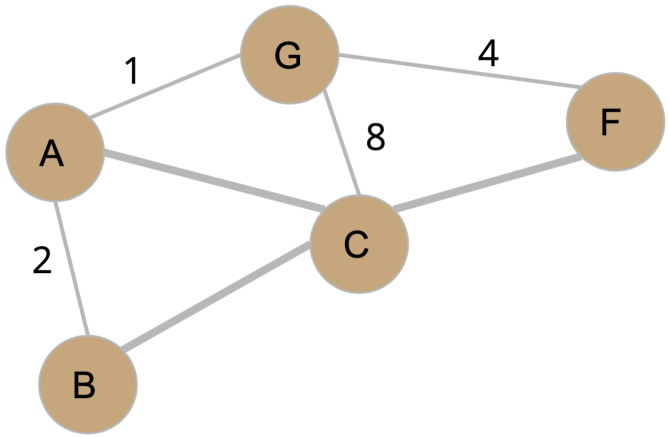
(a) In an arbitrary graph, Prim's can change the priority of a vertex `v` in the priority queue a **maximum** of _____ times and a **minimum** of _____ times. Assume **v** is not the start vertex and the graph is connected and undirected. Give **tight** bounds specific to **v**. Assume we set all priorities to infinity initially.

Solution:
In an arbitrary graph, Prim's can change the priority of a vertex `v` in the priority queue a **maximum** of `degree(v)` times and a **minimum** of 1 times.

Explanation:
Recall that the `degree(v)` of `v` is the neighbors `v` has. It's possible that we *every* neighbor finds a **better** way of getting to `v`, as the second part shows. As such, `degree(v)` is the **maximum** number of change priority operations we can call for `v` since we can only call `changePriority(v, ...)` when are at a neighbor of v.

Next, since all vertices start at priority infinity and the graph is connected, the final priority of each vertex has to change. As such, 1 is the **minimum** number of times. For an example, every time we run Prim's, the first vertex we visit after the start vertex has its priority changed exactly once. Convince yourself why this is the case.

(b) Suppose we run Prim's from A on the graph below.



Fill in the missing edges in the graph to the right so that

1. The priority of C is changed the **maximum** number of times, i.e. the first blank from above.

Solution:
There are many possible solutions to this part. One such solution is setting `AC` to 9, `BC` to 7, and `CF` to 6. To generalize, a valid solution simply

needs to satisfy the following inequalities:

(a) $\texttt{BC} > \texttt{4}$

(b) $\texttt{AC} > \texttt{GC} > \texttt{BC} > \texttt{FC}$

Explanation:
The intuition for why these inequalities must hold is twofold. First, we need C to be the *last* vertex visited. To ensure this is the case, edges AC and BC must be greater than 4, which is accounted for in the check $\texttt{BC} > \texttt{4}$. The reason they must be greater than 4 is to ensure that we visit F before C.

Second, we need to ensure that every edge we consider finds a better way of getting to C. To ensure this is the case, notice that we visit vertices in this order: $\texttt{A} \rightarrow \texttt{G} \rightarrow \texttt{B} \rightarrow \texttt{F}$, assuming that C is the last vertex visited. Accordingly, we must impose the following inequality between the adjacent edges of C: $\texttt{AC} > \texttt{GC} > \texttt{BC} > \texttt{FC}$.

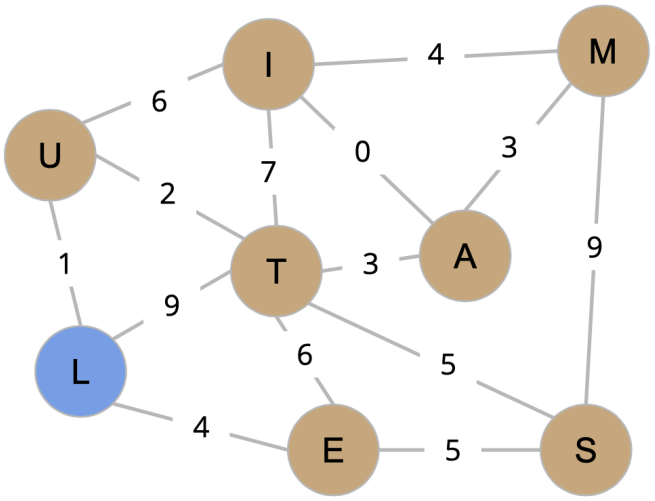
2. The priority of every vertex is changed the **minimum** number of times, i.e. the second blank from above.

Solution:
There are many possible solutions to this part. One such solution is setting AC to 1, BC to 100, and CF to 101. To generalize, a valid solution simply needs AC to be *lighter* than BC, GC, and FC or AC to be *lighter* than AG and AB.

Explanation:
The intuition for the first solution (make AC *lighter* than BC, GC, and FC) is that we want to change the priority of C exactly once. The first edge we consider adjacent to C is AC, so this must be the smallest adjacent edge.

The second, equally valid solution (make AC *lighter* than AG and AB) works because C becomes the first vertex we visit, and we change its priority only once.

5 Kruskal's



(a) We want to run Kruskal's, but we have no cycle detection, so we terminate upon inserting $V - 1$ edges. Will this produce a valid MST on the graph above? If not, determine which edge(s) need to be changed, and to what. If there are many possibilities, choose the one that involves the minimum added/removed weight.

Assume ties are broken alphabetically, and edges are written in alphabetical order, and compared as such. For instance, if edges (A, Z) and (E, H) are equal, (A, Z) would be chosen before (E, H).

Solution:
This will not produce a valid MST. The problem is that we consider `IM` before adding the last vertex `S` to the MST. So, for Kruskal's to work, you *either* need to

1. change `IM` to 5
2. change `ES` to 4

so that the edge `ES` is considered before `IM`.

Here is a video walkthrough of the solutions.

(b) After completing the previous part, Sohum wondered if it's possible to run Kruskal's with limited cycle detection. More specifically, he pondered: what if we can only detect a maximum of **k** cycles during one run of Kruskal's?

Looking at the specific instance of a 6 vertex graph, what is the **minimum** value of **k** for which we can ensure that Kruskal's will always work?

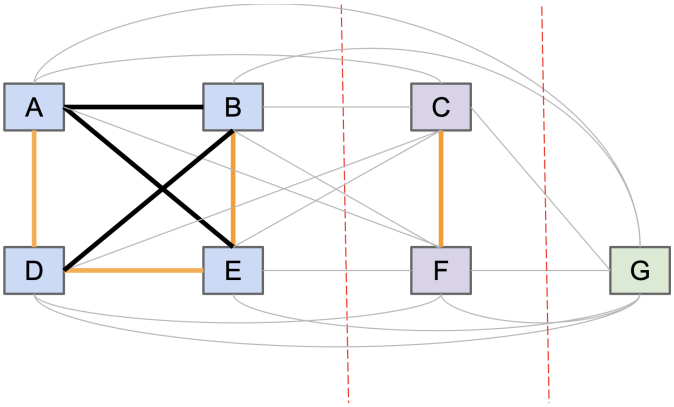
Solution: 6

Explanation:
To find the minimum value of **k** for which we can ensure Kruskal's will always work,

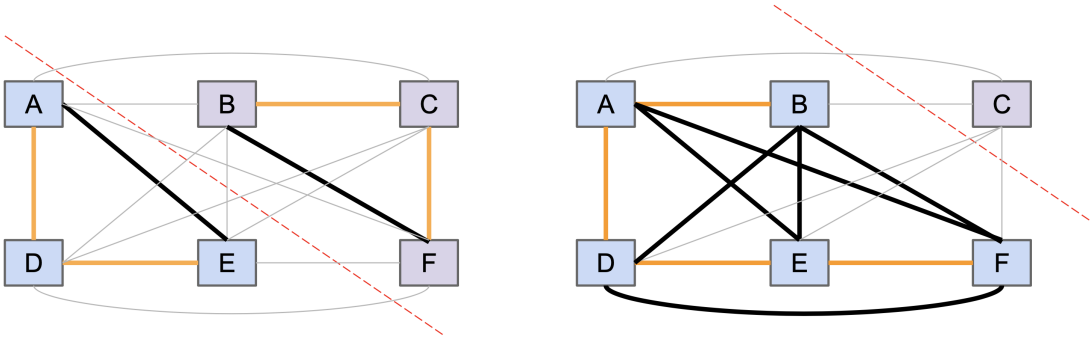
we want to find the **maximum** number of edges we could possibly consider in one run of Kruksal's before terminating. Well, when do we terminate in Kruskal's? We terminate after adding **V - 1** edges, i.e. when the MST contains all the vertices. So, if we haven't finished Kruskal's, it means that we haven't added **V - 1** edges and we have two or more components that are *not* connected to each other. In order for there to be two or more components that are not connected to each other, note that there must be some cut that we haven't considered *any* crossing edges of. For instance, looking at the partial state of an example of Kruskal's below, notice that we haven't considered any crossing edges of two cuts.

Note that:

1. orange edges: in the MST
2. black edges: edges considered but cause cycle
3. light grey edges: edges yet to be considered)



Next, since we want to find the **maximum** number of edges we could possibly consider in Kruksal's before terminating, let's focus the case when Kruskal's is *about* to finish and we have considered the maximum number of edges prior to this point. If Kruskal's is about to finish, it means there is only one more edge to be added, i.e. one more cut to find a crossing edge through. Since we want to consider the maximum number of edges, we want the number of edges in this "final" cut to be as few as possible, since the more edges there are in the final cut, the fewer edges we could've considered prior to this point. And, we want there to be as many edges as possible "outside" of this cut.



Looking at the two possible "almost finished" states of Kruskal's above, i.e. states

with two connected components remaining and one cut to find a crossing edge through, notice the number of edges in the cut on the right is far fewer than the number of edges on the cut on the left. To generalize this finding, the fewest number of crossing edges we can have in *any* cut between sets `A` and `B` occurs when the set `A` or `B` contains only one vertex. As such, the **maximum** number of edges we can consider in one run of Kruskal's occurs in the specific instance of the right graph above where the crossing edges of one "small" cut in the graph haven't been considered while every edge in the remaining graph has.

Notice that out of the edges we've considered in the right graph above, the `4` orange edges don't need cycle detection since they are part of the MST. However, for the remaining `6 black edges`, they need cycle detection, and we can say the **minimum** value of **k** is **6**.

If that was a lot, here is a video walkthrough of the solutions.

6 Graph Algorithm Design

Here is a video walkthrough of the solutions. Note that the order of the subparts have changed. In the video, we first go over part c, then part a, then part b, and finally part d.

Given a **undirected**, **weighted** graph G with **positive**, **integer** edge weights, we want to find a path from **u** to **v** that minimizes the total cost. For each "catch" below, find the path of optimal cost no slower than $O(ElogV)$.

(a) Excluding the start and end vertex, we partition the vertices into 5 subsets, and we must visit vertices in order of their subset. That is, if we are in subset **k**, the next vertex we visit must be in subset $\mathbf{k+1}$.

Solution:
Modify Dijkstra's algorithm so that when we visit a vertex in a subset **k**, we *only* consider neighbors in the subset $\mathbf{k+1}$.

Alternate Solution:
Modify the graph by removing all edges that do **not** connect vertices of adjacent subsets. Next, for each remaining edge, we know it must connect vertices in adjacent subsets, let's call these subsets **k** and $\mathbf{k+1}$. Replace each undirected edge with a directed edge from **k** to $\mathbf{k+1}$. Run Dijkstra's from **u** to **v**.

(b) We must visit two designated vertices s and k on our path.

Explanation:
Notice that the shortest path from **u** to **v** will either go $\mathbf{u} \to \mathbf{s} \to \mathbf{k} \to \mathbf{v}$ or $\mathbf{u} \to \mathbf{k} \to \mathbf{s} \to \mathbf{v}$, where $\mathbf{s} \to \mathbf{t}$ corresponds to taking a path from **s** to **t**. Since we want the final path of minimum cost, each of these paths should be shortest paths. As such, we want the following shortest paths:

- $\mathbf{u} \to \mathbf{s}$
- $\mathbf{s} \to \mathbf{k}$
- $\mathbf{k} \to \mathbf{v}$
- $\mathbf{u} \to \mathbf{k}$
- $\mathbf{k} \to \mathbf{s}$
- $\mathbf{s} \to \mathbf{v}$

However, since the graph is undirected, we know that the shortest path from **s** to **k** is the same as the shortest path from **k** to **s**, and we can reduce the required shortest paths to the below:

- $\mathbf{s} \to \mathbf{u}$
- $\mathbf{s} \to \mathbf{k}$
- $\mathbf{s} \to \mathbf{v}$
- $\mathbf{k} \to \mathbf{v}$
- $\mathbf{k} \to \mathbf{v}$

Well, we did a bit more than "reduce" in the step above, since we also changed the order of some of the shortest paths to highlight that all of the shortest paths we need either start from **s** or **k**.

Solution:

Run Dijkstra's from **s** and from **k** to find the needed shortest paths in the previous list. Plug in the calculated shortest paths into the expressions below.

1. $\mathbf{u} \to \mathbf{s} \to \mathbf{k} \to \mathbf{v}$
2. $\mathbf{u} \to \mathbf{k} \to \mathbf{s} \to \mathbf{v}$

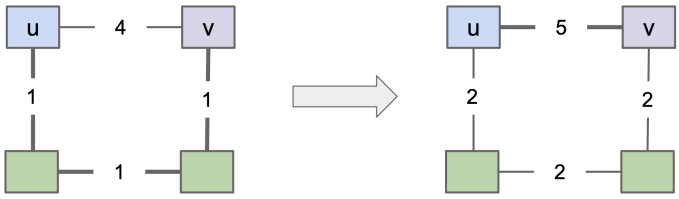
Return the path with the minimum total weight.

(c) If two paths from u to v are of the same cost, we will choose the path with fewer edges.

Solution:

Add $1/E$ to the weight of each edge where E is the number of vertices in the graph. Run Dijkstra's from **u** to **v**.

So, why does this work, and where did we get the idea to add $1/E$ to each edge? Let's begin with the second question. Intuitively, adding a little to each edge **discourages** taking paths with many edges. So why not add 1 to each edge? Looking at the graph below, if we add 1 to every edge, we have now **changed** the shortest path from **u** to **v**! The *only* purpose of the added weight should bes to break ties between two paths of equal length.



Okay, so we want the amount added to be really small, so why not 0.001, or even 0.00001? The problem with any constant is that the same problem showed above may occur for a graph that is really, really big.

Okay, so what if we add an offset that takes in consideration the number of edges in the graph, like $1/E$ (or anything smaller than this proportional to E, e.g. $1/E^2$).

This would work! If we ever have two paths of equal cost, the smallest one path can be is 1 edge and the largest the other path can be is $E-1$ edges. On the larger path, notice that the sum of all the offsets comes out to $(E-1)/E$, which is less than 1! Thus, since we are using **integer** edge weights, this added offset can only serve to break ties, and is not susceptible to the problem described above.

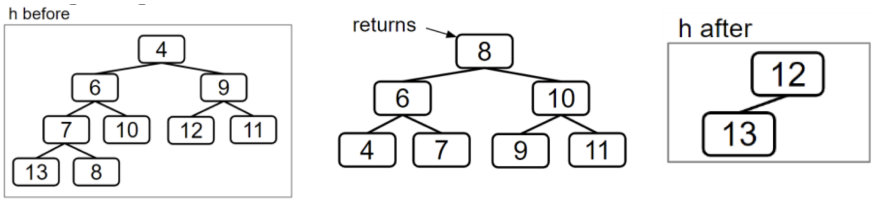
(d) Instead of starting from u and ending at v, we can start from any vertex in a subset of vertices and end at any vertex in a subset of vertices. Each subset is of size k.

Solution:
Create two dummy nodes d_1 and d_2. Connect d_1 to every vertex in the start subset with an edge of zero weight. Connect d_2 to every vertex in the ending subset with an edge of zero weight. Run Dijkstra's from d_1 to d_2. Ignore the edges connected to d_1 and d_2 in the shortest path.

7 Yggdrasil (Heaps)

This problem was taken from Spring 2019 Midterm 2.

This is a very challenging problem. Write a function that takes an integer `k` and a min-heap `h` (in tree representation) and removes the **k smallest values** and returns them organized into **valid perfectly balanced BST**. For example, if we call `heapToBBST(7, h)` on the MinHeap in the left figure, it returns the Tree in the middle figure, and as a side-effect, `h` becomes the MinHeap in the right figure.



This should be done in-place, i.e., reusing the TreeNodes from the min-heap. Your function should complete in $O(NlogN)$ time, where N is the number of items in the min-heap, and use no more than $O(logN)$ additional memory while it is running. For full credit, it must work for arbitrary values for k, but you can earn almost full credit if your solution works for $k = 2^H$–1 (i.e. powers of 2 minus 1).

```
public class TreeNode {
    public int item;
    public TreeNode left;
    public TreeNode right;
}
public class MinHeap {
    /* Even though a MinHeap is made up of TreeNodes, the instance
    * variables are private. You cant directly access them. */

    /* removes the minimum node and returns it */
    public TreeNode removeMin() { /* ... */ }
}
public static TreeNode heapToBBST(int k, MinHeap h) {
    if (k == 0) {
        return null;
    }
    _________________________________
    _________________________________
    _________________________________
    _________________________________
    _________________________________
    _________________________________
    _________________________________
    _________________________________
    _________________________________
}
```

Solution:

Here is a video walkthrough of the solutions.

```
public static TreeNode heapToBBST(int k, MinHeap h) {
    if (k == 0) {
        return null;
    }
    TreeNode left = heapToBBst(k / 2, h);
    TreeNode middle = h.removeMin();
    TreeNode right = heapToBBst(k  (k / 2)  1, h);
    middle.left = left;
    middle.right = right;
    return middle;
}
```