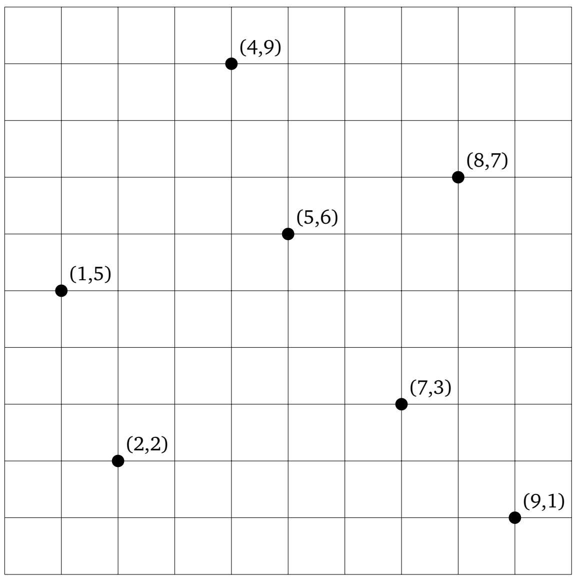
Section 05: Solutions

1. k-d Trees

(a) Given the points shown in the grid to the right, draw a perfectly balanced k-d tree (where $k = 2$, i.e. a 2-d tree) in the box below. For this tree, first split on the x dimension. The resulting tree should be complete with height 2. Then, draw the corresponding splitting planes on the grid to the right.

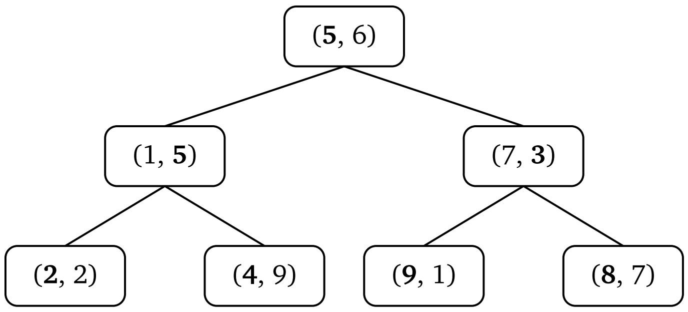


Solution:

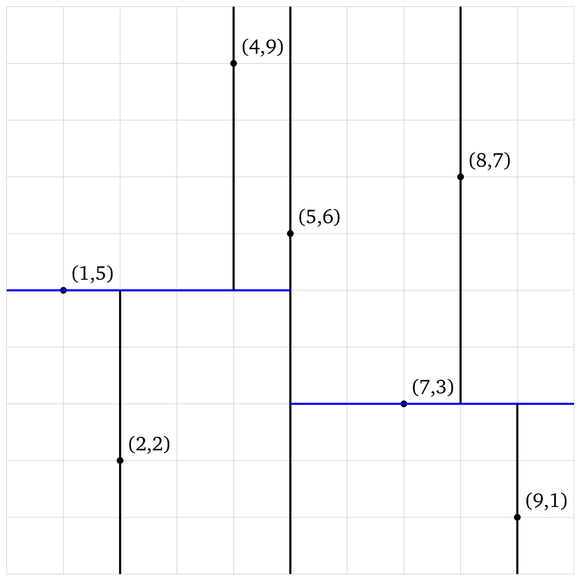
There are several different ways to create a k-d tree and we will explore the two most common approaches. The first approach will be to create a perfectly balanced tree given all of the points at the start. Given the full set of points what we want to have is that each node should partition the remaining points into two equal halves which will be passed down to the node's left and right children. In doing this we need to adhere to the invariants of the k-d tree. For example if we are splitting on x the points passed into the left child should have x-values less than the median point's x-value and the points passed into the right child should have the x-values greater than the median point's x-value. As we have already sorted the list we can take the sublist of the points before the median and pass them along to the left subtree and then take the points after the median and pass them along to the right subtree.

In order to implement this at each node we should sort the points based off of their x or y coordinates depending on if we are splitting on x or y. After we have done this we will select the median point to be the current point and then we will create left and right subtrees based off of the sublists of points split by the median.

For our case of points we can see that if we first split on x we will sort the points on x yielding (1,5), (2,2), (4,9), (5,6), (7,3), (8,7), (9,1). The median of this list is (5,6) so this will become our trees root. We will make recursive calls passing in the list of points (1,5), (2,2), (4,9) and (7,3), (8,7), (9,1) to the left and right subtrees. This process will repeat until we are left with the following tree which will be perfectly balanced.



We can also visualize the k-d tree points in space by drawing on the splitting planes as is shown below.



The secondary approach to constructing a k-d tree is to simply add the points one at a time. The trade off here is that the code for this method will likely be simpler, but now we are not guaranteed to have a perfectly balanced tree. Thinking back to BSTs there were cases where if you inserted elements in a certain order the resulting BST would end up being spindly and the same thing can happen for k-d trees. If we want to improve our runtime compared to the naive solution, we want to have at least a roughly balanced tree. One thing that we can do to increase the likelihood of creating a roughly balanced tree is to randomize the order of insertion. This again will not create a perfectly balanced tree as before, but in expectation the tree should be roughly balanced. In doing this we expect to come close to a perfectly balanced tree, but with much simpler code.

(b) Insert the point (6, 2) into the k-d tree you drew below. Then, add that point to the grid and draw the corresponding splitting plane.

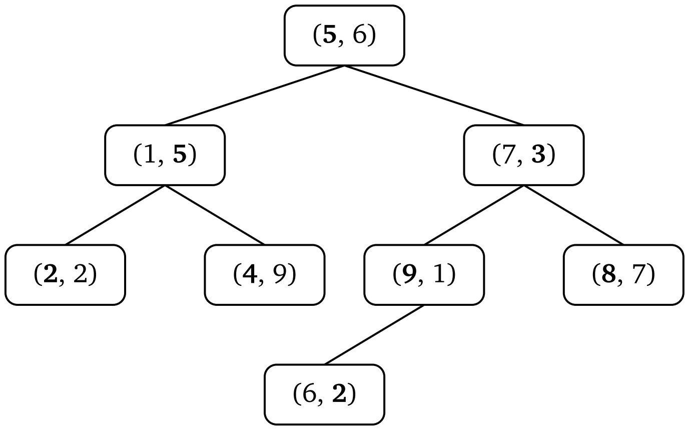
Solution:

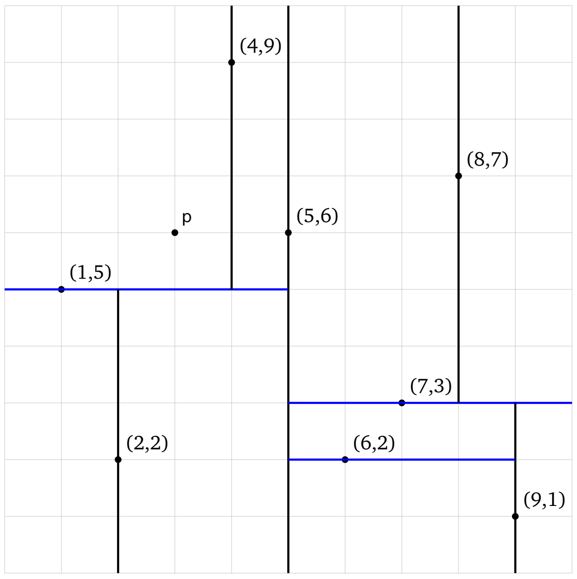
Now we will walk through the insertion of the single point (6,2) into our exising k-d tree from (a). We begin at the root which corresponds to the point (5,6). We are splitting on x at this top node and the x value of the point to be inserted, 6, is greater than that of the root's x-value, 5, so we proceed to the right.

Next the current point corresponds to the point (7,3). We are splitting on y at this node and the y value of the point to be inserted, 2, is less than that of the current node's y-value, 3, so we proceed to the left.

Now the current point corresponds to the point (9,1). We are splitting on x at this node and the x value of the point to be inserted, 6, is less than that of the current node's x-value, 9, so we proceed to the left. This corresponds to a null child, so we will insert the new point at this location in our tree.

Below we can see updated diagrams corresponding to the two views of our k-d tree.





(c) Find the nearest point to (3, 6) in your k-d tree. Mark each branch that is not visited (pruned in execution of `nearest`) with an `X` through the branch.
Solution:

K-d Tree Nearest Algorithm:
Given a query point `p` we will do the following starting at the root node. We will not recurse on all nodes in the tree, as we will likely be able to prune large portions of the tree. One way to structure this is to have the call to nearest call a helper function that takes in 3 arguments: `n` the node in the tree that we are currently visiting, `p` the query point, and `globalBest` the closest node we have seen so far. To start we will make the call `nearestHelper(root, p, root)`.

(i) We check if the current node, `n` we are at in the tree is closer to `p` than the `globalBest` that was passed in. If it is indeed closer update `globalBest` to be `n`.

(ii) Next we must recurse in either direction of the splitting plane in order to determine if there is a closer node to the query point in that half of the tree (or subtree). As we know where the query point lies with respect to the splitting plane we will first recurse in the direction of the query point (intuitively we expect the closest point to be in the direction of the query point although this is not always the case). This side of the tree we can think of as the ”good side” For example if `n` represents the point (3,4), we are splitting on x, and the query point `p` is (2,5) then we will first recurse on the left child of `n` as $2 < 3$. This recursive call will either return the current value of `globalBest` if no closer node was discovered or it will return the new closest point which `globalBest` should be updated to.

(iii) Now we need to consider if the other side of the tree (the ”bad side”) needs to be checked. We only want to check this side of the tree if it is possible that a closer point than the `globalBest` could exist on that side of the tree. If it is impossible for such a point to lie on that side of the tree then we will prune that branch. For this we will do the following:

  (i) We can imagine a circle which is centered at the query point, `p`, with the same radius as the distance from the `p` to the `globalBest`. This circle represents the region in cartesian space where it is possible that a closer point than `globalBest` could lie if it does exist. To see if we need to check the other side of the tree we can see if this circle intersects the splitting plane such that some portion of the circle lies on either side of the splitting plane. (Note in 2-dimensions we can use circles and our splitting planes will be lines, but in higher dimensions the circles will be replaced with hyperspheres and the splitting planes will be hyperplanes).

  To determine if the circle intersects the splitting plane, we can reduce this to a slightly simpler calculation. We notice that if a point did exist in the other side of the splitting plane, the closest that it could exist to the query point would be perpendicular to the splitting plane in line with the query point. For example suppose the query point is (5,5) and the current node contains the point (4, 3) where we split on x. Space is split into points with $x > 4$ and $x < 4$. For this we would first search the tree corresponding to points with $x > 4$. If the current best found when searching this half of the tree is (5,7). The closest point to the query point that is on the other side of the splitting plane would be (4,5). Now we can see that the current best is distance 2 away from the query point, but there are possible points on the other side of the splitting plane that could be distance 1 away, so we *must* check the other side of the plane.

  (ii) If we did end up recursing on the other side of the tree (or subtree), similarly this will either return the current value of `globalBest` if no closer node was discovered or it will return the new closest point which `globalBest` should be updated to.

(iv) Return `globalBest`.

Solution:

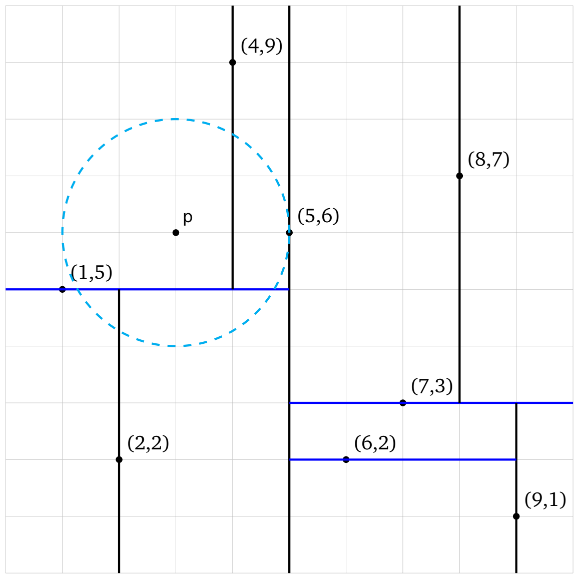
Walkthrough of Specific Call to Nearest
We begin at the root of the tree, so `n` corresponds to the point (5,6), additionally `globalBest` will correspond to (5,6). The query point `p` will be (3,6). In this case `n` is the same as `globalBest` so we do not need to update `globalBest`. Next `n` splits on x, so we compare the x-values of `n` and `p`. We see that the good side of the tree will be the left side (corresponding to x-values less than 5) as $3 < 5$, so we will first recurse on the left side.

Now `n` corresponds to the point (1,5), and `p` and `globalBest` remain unchanged. We see that `n` is distance 2.236 away from `p` which is worse than `globalBest` which is distance 2 away, so we do not need to update `globalBest`. Next `n` splits on y, so we compare the y-values of `n` and `p`. We see that the good side of the tree will be the right side (corresponding to y-values greater than 5) as $6 > 5$, so we will first recurse on the right side.

Now `n` corresponds to the point (4,9), and `p` and `globalBest` remain unchanged. We see that `n` is distance 3.162 away from `p` which is worse than `globalBest` which is distance 2 away, so we do not need to update `globalBest`. Next `n` splits on x, so we compare the x-values of `n` and `p`. We see that the good side of the tree will be the left side (corresponding to x-values less than 4) as $3 < 4$, so we will first recurse on the left side.

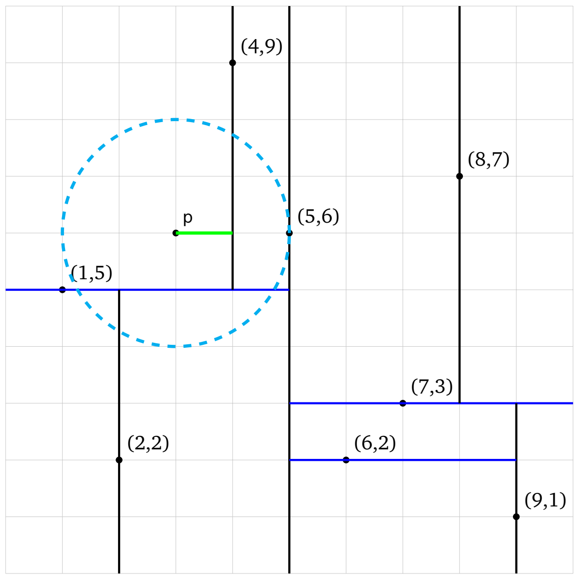
Now `n` corresponds to null as there is no left child of the node corresponding to (4, 9). In this case we simply return `globalBest` which is (5,6).

After returning from this recursive call we are now once again at the node where `n` corresponds to the point (4,9). Now we need to make a decision of whether or not we should visit the “bad side” of the tree. We can begin by visualizing the circle mentioned above.



If a point exists that is closer to `p` than `globalBest` then it must be less than distance 2 away from `p`. The circle corresponds to the region which would contain these points if they exist in our tree. What we can see is that this circle overlaps the splitting plane (the line corresponding to the x-value 4). This means that it is possible that a point does exist on the other side of the splitting plane that is closer to `p` than `globalBest`, so we must recurse on the "bad side."

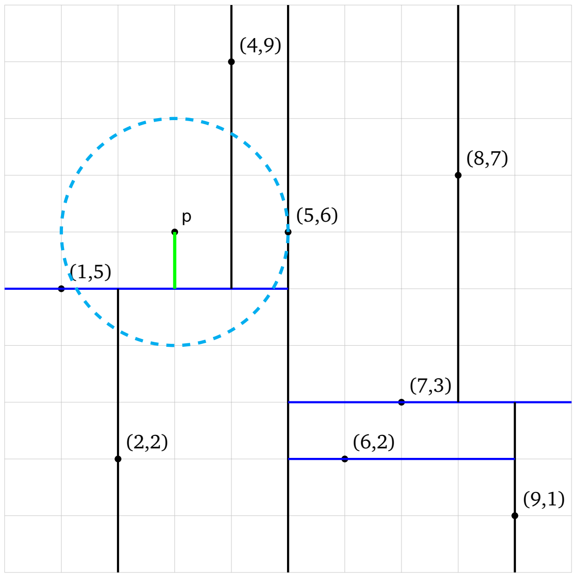
In code however the notion of creating a circle can be simplified to the following. We can consider the closest point to `p` on the other side of the splitting plane. This will correspond to the point that lies at the intersection of the splitting plane corresponding to `n` and the line perpendicular from the splitting plane that passes through the query point `p`. In this case we can see that this closest point that could lie on the other side of the splititng plane defined by the point (4,9) split on x would be the point (4,6). Now we can compare the distance from `globalBest` to `p` to the distance from this hypothetical new best point to `p`. If the distance from `globalBest` to `p` is greater then we must check the "bad side," otherwise we can prune.



Again we are considering the case where `n` is (4,9), `p` is (3,6), and the `globalBest` is (5,6). The green line above corresponds to the shortest distance to the splitting plane which would be of distance 1. This is less than 2, the distance from `p` to `globalBest`. So we must visit the "bad side".

When we recurse on the bad side `n` corresponds to null as there is no right child of the node corresponding to (4, 9). In this case we simply return `globalBest` which is (5,6). Now that we have visited all of the children of (4,9) we can return the `globalBest` which will still be (5,6).

Now we have returned back to where `n` corresponds to the point (1,5). We have visited the “good side” where we did not find anything better than the `globalBest`, so `globalBest` is not updated. Next we need to consider if we need to explore the “bad side” of `n`. Once again we can visualize this using the same rule as above.

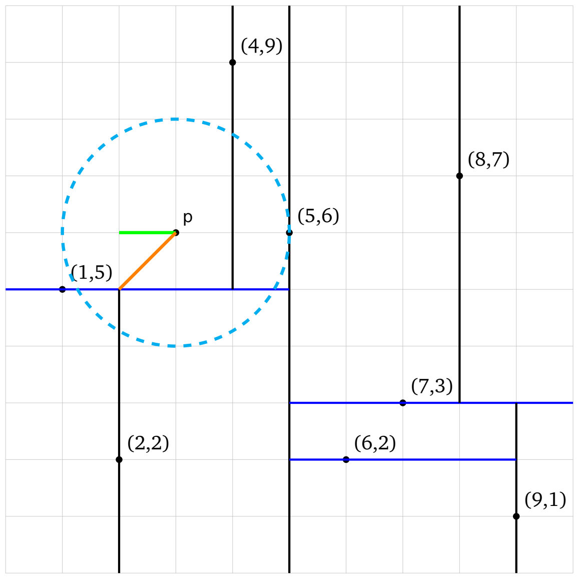


We see that as the circle does overlap the splitting plan we will need to visit the “bad side” of the tree. Again we can also look at the closest hypothetical point to `p` on the other side of the splitting plane. This again corresponds to the green line above, and we can see that this distance is again 1, compared to the distance of 2 between `globalBest` and `p`, so we again we must explore the other side of the tree.

Now `n` corresponds to the point (2,2) We see that `n` is distance 4.123 away from `p` which is worse than `globalBest` which is distance 2 away, so we do not need to update `globalBest`. Next `n` splits on x, so we compare the x-values of `n` and `p`. We see that the “good side” of the tree will be the right side (corresponding to x-values greater than 2) as $3 > 2$, so we will first recurse on the left side.

After recursing, `n` corresponds to null as there is no right child of the node corresponding to (2, 2). In this case we simply return `globalBest` which is still (5,6).

Next we have to consider if we need to check the “bad side” of `n`. In this case the check is a bit iteresting. What we can see is the points corresponding to the left child of `n` (which again is (2,2)) will be points with x-values less than 2 and y-values less than 5. When we consider doing the same perpendicular closest point we can see that this will correspond to the point (2,6) which is actually not contained within this region. Essentially the rule we introduced above gives a weaker condition of pruning, as the closest actual point to `p` that exists in this region of points would be the upper right hand corner of this region, namely (2,5). Since the closest actual point would be (2,5), ideally we should be comparing that distance from `p` to this point (2,5). We can see this visualized below.



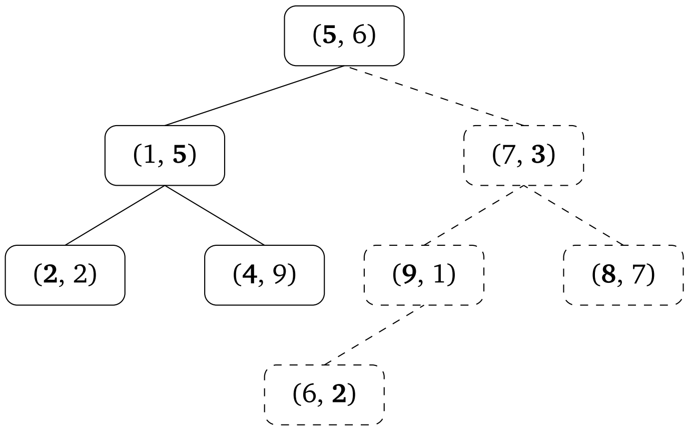
In the above the green line corresponds to the distance rule we have been using, whereas the orange line corresponds to the distance that would be used in the optimal pruning case. From this what we can see is that the perpendicular distance will always be less than this diagonal distance, so by doing the simpler pruning check our algorithm is still correct although it might visit slightly more nodes in our tree than necessary. For implementing this code, it will be much simpler to just use the simpler perpendicular rule. As such we will choose to ignore this slightly more optimal pruning rule, but it is important to know that it does exist.

Regardless we see that we do need to check the "bad side" so we recurse on the left child of the node corresponding to (2,2). In this case now `n` will again be null so we return the current best, (5,6).

At this point we can now return to the call where `n` is (1,5). We have visited both the "good side" and the "bad side" so we can return the `globalBest` but this has not been updated so we will return the node corresponding to (5,6).

Finally we return to the call where `n` is the root, (5,6). Once more we need to check if we need to explore the other side of the tree. In this case we can see that the circle is tangent to the splitting plane, or equivalently the best possible point perpendicular from `p` is distance 2 away from `p`. As we already have the `globalBest` storing a point that is distance 2 away from `p` then we do not need to explore the other side of the tree, as the best we can do is equivalent to what we already have. Here we prune the entire right half of the tree and return `globalBest` which corresponds to the point (5,6).

Below is the tree that shows which nodes were explored and which we were able to prune where the dashed lines and nodes corresponds to what we pruned.



2. Hash Functions

For the following implementations of the `Integer` class's `hashCode()` function, answer whether it is valid or invalid. If it is invalid, explain why. If it is valid, point out a flaw or disadvantage.

Note: The `Integer` class extends the `Number` class, a direct subclass of `Object`. The `Number` class' `hashCode()` method directly calls the `Object` class' `hashCode()` method.

(a) `return -1;`

Solution:

Valid. As required, this hash function returns the same `hashCode` for Integers that are `equals()` to each other. However, this is a terrible hash code because collisions are extremely frequent (collisions occur 100% of the time).

(b) `return intValue() * intValue();`

Solution:

Valid. Similar to (a), this hash function returns the same `hashCode` for integers that are `equals()`. However, integers that share the same absolute values will collide (for example, $x = 5$ and $x = -5$ will have the same hash code). A better hash function would be to just return the `intValue()` itself.

(c) `return super.hashCode();`

Solution:

Invalid. This is not a valid hash function because integers that are `equals()` to each other will not have the same hash code. Instead, this hash function returns some integer corresponding to the integer object's location in memory.

3. Hashing

(a) Suppose we have a hash table that uses separate chaining and has an internal capacity of 12. Assume that each bucket is a linked list where new elements are added to the front of the list.

Insert the following elements in the EXACT order given using the hash function $h(x) = 4x$ (don't worry about resizing the internal array):

0, 4, 7, 1, 2, 3, 6, 11, 16

Solution:

To make the problem easier for ourselves, we first start by computing the hash values and initial indices:

key	hash	index (pre probing)
0	0	0
4	16	4
7	28	4
1	4	4
2	8	8
3	12	0
6	24	0
11	44	8
16	64	4

The state of the internal array will be

$6 \to 3 \to 0$	/	/	/	$16 \to 1 \to 7 \to 4$	/	/	/	$11 \to 2$	/	/	/

(b) Suppose we have a hash table with an initial capacity of 12. We resize the hash table by doubling the capacity. Suppose we insert integer keys into this table using the hash function $h(x) = 4x$.

Why is this choice of hash function and initial capacity suboptimal? How can we fix it?

Solution:

Notice that the hash function will initially always cause the keys to be hashed to at most one of three spots: 12 is evenly divided by 4.

This means that the likelihood of a key colliding with another one dramatically increases, decreasing performance.

This situation does not improve as we resize, since the hash function will continue to map to only a fourth of the available indices.

We can fix this by either picking a new hash function that's relatively prime to 12 (e.g. $h(x) = 5x$), by picking a different initial table capacity, or by resizing the table using a strategy other then doubling (such as picking the next prime that's roughly double the initial size).

4. Hashing: Ice Cream

Assume we have a hash table that maps `Characters` to `Integers`, and starts with an internal capacity of 4 and resizes by doubling the capacity if the insert would cause the load factor to exceed 0.75. Also assume that the hash table uses separate chaining with linked lists where new elements are added to the front of the list.

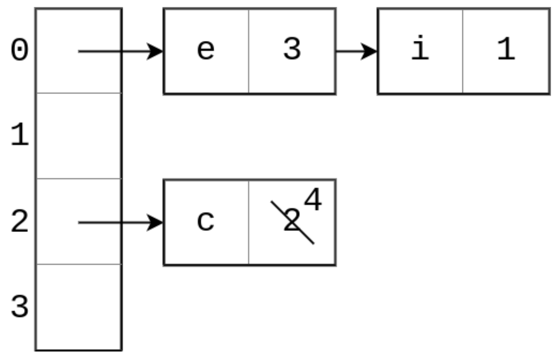
(a) Draw the hash table after inserting the following items in the given order:

`(‘i’, 1), (‘c’, 2), (‘e’, 3), (‘c’, 4), (‘r’, 5), (‘e’, 6), (‘a’, 7), (‘m’, 8)`

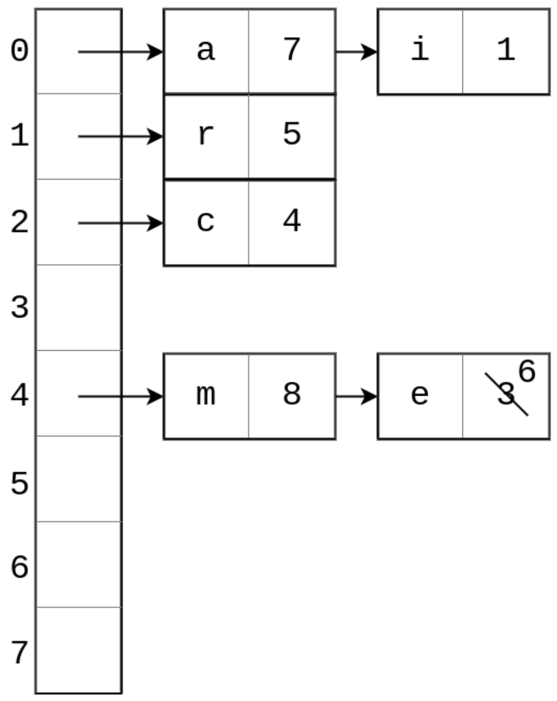
Assume the hash code for `‘a’` is 0, `‘c’` is 2, `‘e’` is 4, `‘i’` is 8, `‘m’` is 12, and `‘r’` is 17.

Solution:

The hash table after adding `(‘i’, 1), (‘c’, 2), (‘e’, 3), (‘c’, 4)`



Once `(‘r’, 5)` is inserted, the hash table will resize. The hash table after adding all elements will look as follows:



(b) How many hash collisions occur? Do not count the ones that occur during resizing.

Solution:

3 (updates to ‘`c`’ and ‘`e`’ do not count as hash collisions, they just update the value that those keys map to)

(c) How many hash collisions occur during resizing?

Solution:

0 (during resizing, none of the elements get placed at the same index)

(d) What is the current load factor after all insertions?

Solution:

$6/8 = 0.75$