CMSC 420: Spring 2023

Programming Assignment 2: Sliding-Midpoint kd-Trees

Handed out: Mon, Mar 27. Part 1 (worth 20%) due **Fri, Apr 7, 11:59pm** and Part 2 (worth 80%) due **Fri, Apr 21, 11:59pm**.

Overview: In this assignment we will implement a variant of the kd-tree data structure, called a *sliding-midpoint kd-tree* (or `SMkdTree`) to store a set of points in 2-dimensional space. This data structure involves a number of notable features:

- It is an extended binary tree consisting of internal and external nodes. Points are stored only in the external nodes.
- Its subtrees periodically rebalance themselves. Each node maintains a counter which is incremented as insertions are made. When the counter is large enough, we rebuild the subtree completely from scratch.
- Splitting is based on a method called the *sliding-midpoint rule*, which attempts to keep cells as nearly square as possible, subject to the restriction that every cell has at least one point.

Points, Labeled Points, and Rectangles: The objects to be stored in the trees are 2-dimensional points. To save you some effort, as part of the skeleton code, we will provide you with a class for 2-dimensional points, called `Point2D.java`. This class will provide you with some utility functions, such as accessing individual coordinates and computing distances.

The points stored in your tree will be enhancements of the `Point2D` type, called an `LPoint` (for *labeled point*), which also stores a string label. An `LPoint` implements the following Java interface:

```
public interface LabeledPoint2D {
    public double getX();          // get point's x-coordinate
    public double getY();          // get point's y-coordinate
    public double get(int i);      // get point's i-th coordinate (0=x, 1=y)
    public Point2D getPoint2D();   // get the point itself (without the label)
    public String getLabel();      // get the label (without the point)
}
```

For example, given an `LPoint` called `q` with coordinates $(31.2, -7.5)$ and label `"ABC"`, the operation `q.getX()` returns 31.2, `q.getLabel()` returns `"ABC"`, and `q.getPoint2D()` returns the `Point2D` structure for the point $(31.2, -7.5)$. The `SMkdTree` is templated with the labeled point type:

```
public class SMkdTree<LPoint extends LabeledPoint2D> { /* fill this in */ }
```

We will also provide you with a class for storing axis-aligned rectangles, called `Rectangle2D.java`. This also provides a number of useful functions, such as testing whether two rectangles are disjoint or whether one contains the other.

Node Structure: The `SMkdTree` data structure is an *extended binary tree* (see Fig. 1). As mentioned above, this is an extended binary tree. Each *external node* stores a single point (that is, a reference to an `LPoint`). We allow external nodes to be *empty*, in the sense that they store no point. Each *internal node* stores the node's cutting dimension (either 0 or 1), its cutting value (a `double`), and pointers to its left and right children. Following the same convention as standard kd-trees, if a point lies on the cutting line, it is placed in the right subtree.

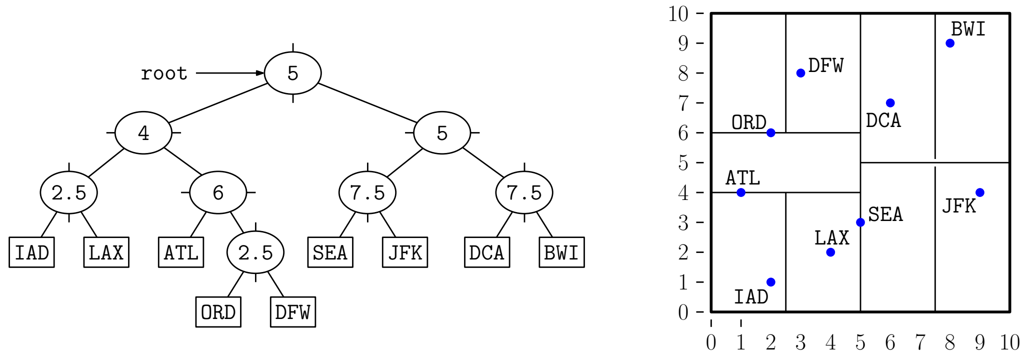


Figure 1: An example of an SMkd-Tree with bounding box $[0, 10] \times [0, 10]$.

Note that there are no `null` child pointers. Every internal node has two non-null children, which may be either internal or external nodes. External nodes have no children by definition. Even the root pointer is non-null. An empty tree is represented having the `root` point to a single external node whose point is `null`.

A natural way to handle the two different node types in Java is to create an abstract inner class, `Node`. This stores all information that is common to both node types (such as declarations of helper functions). From this we derive two subclasses, one for internal nodes, and the other for external nodes, as shown below.

```
public class SMkdTree<LPoint extends LabeledPoint2D> {

    private abstract class Node {                 // abstract node type
        abstract LPoint find(...);                // helpers and utilities
        abstract Node insert(...);
        abstract Node delete(...);
        // ... (other abstract methods)
    }

    private class InternalNode extends Node {     // internal node
        int cutDim;                               // cutting dimension
        double cutVal;                            // the cutting value
        Node left, right;                         // children
        // ... (other data and implementation of methods)
    }

    private class ExternalNode extends Node {     // external node
        LPoint point;                             // the point (null if empty)
        // ... (other data and implementation of methods)
    }
```

The tree itself stores a pointer to the root node and a bounding axis-aligned box (i.e., `Rectangle2D`), which contains all the points. This bounding box serves as the cell for the root node. It maintains additional information, which will be discussed below.

You might observe that we don't need to store the node types. This is handled automatically by Java's inheritance mechanisms. For example, suppose that we want to invoke the `find` helper function on the root. Each node type defines its own helper. We then invoke `root.find(pt)`. If `root` is an internal node, this invokes the internal-node find helper, and otherwise it invokes the external-node find helper.

Sliding-Midpoint Rule: This splitting rule is unusual in that it prioritizes producing cells that are nearly square, rather than producing a tree of balanced height. This is useful in applications, such as finite element analysis, where the shapes of the cells are of greater importance than the height of the tree.

Suppose for now that we want to store a set S of two or more points, all of which lie within a given rectangular cell R. (Later we will see how to apply this to insert individual points.)

The sliding-midpoint rule first determines the splitting dimension as follows. It takes the longer dimension of R as the default (see Fig. 2(a)).[1] If the length and width are the same, then cut vertically.

Next, sort the points along the cutting dimension.[2] If we are in a badly degenerate situation where all the points have the same coordinates along the cutting dimension (which is easily determined by comparing the first and last points in sorted order), then this cutting dimension is no good. Instead, use the other axis as the cutting dimension (see Fig. 2(b)).[3]

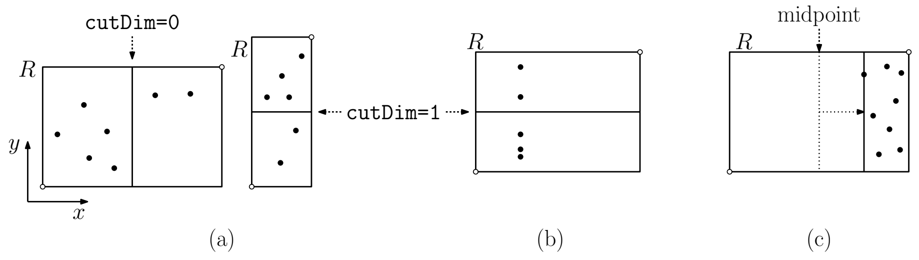


Figure 2: Sliding-midpoint splitting rule.

Now that we have selected the cutting dimension, we need to select the cutting value. The default cutting value is the midpoint of R along the cutting dimension. However, if all the points lie on one side of the midpoint, we move (or "slide") the cutting line until it coincides with the first point (see Fig. 2(c)).

[1]Thus, if the rectangle is wider than tall, the cut is vertical, and otherwise it is horizontal. Remember that the cutting dimension is perpendicular to the direction of the cut, so a vertical cut uses 0 (or x) for the cutting dimension, and a horizontal cut uses 1 (or y).

[2]**Hint:** Don't write your own sorter. The best way to do this is to use Java's built-in sorting function, `Collections.sort`, and create two `Comparator` objects, one that sorts by x and another by y.

[3]**Hint:** If you implemented your comparator to sort lexicographically, then you won't need to resort after changing your cutting dimension.

By our convention that points lying on the splitting line are placed in the right subtree, it is possible to produce external nodes that contain no point (this happens to the left subtree in Fig. 2(c)).

Bulk Creation: A useful utility program that you should implement creates a new subtree for a set of zero or more points and a given rectangular cell. If there are zero points in the set, create a single external node whose associated point is `null`. If there is one point in the set, create a single external node that contains this point. If there are two or more points in the set, apply the above sliding-midpoint rule to determine the cutting dimension and cutting value. Partition the points about the splitting line (remembering that ties are broken in favor of the right subtree). Then recursively build the left subtree and right subtree from these two subsets.[4] Finally, create a new internal node having this cutting dimension, cutting value, and these subtrees as children.

Here is a suggestion of how this function might be defined. It is passed in the points as a Java `ArrayList` and the rectangular cell, and it returns a pointer to the root node of the kd-tree that is constructed.

```
Node bulkCreate(List<LPoint> pts, Rectangle2D cell)
```

Comparing and Sorting Points: Buld creation involves sorting points, either by x or by y. You should use Java's built-in sorting function, `Collections.sort`. To instruct it how to sort, you provide it a comparison function. (In the case of partitioning for insertion, this is either lexicographically by (x, y) or lexicographically by (y, x), depending on the cutting dimension).

In Java, this is done defining a class that implements the `Comparator` interface. Such a class defines a single function, called `compare`, which compares two objects of the desired type, and returns a negative, zero, or positive result depending on which argument is larger. In our case, the objects are labeled points, `LPoint`. Here is a brief example on how you might set this up. (Some examples can be found `here`.)

```
private class ByXThenY implements Comparator<LPoint> {
    public int compare(LPoint pt1, LPoint pt2) {
        /* compare pt1 and pt2 lexicographically by x then y */
    }
}
private class ByYThenX implements Comparator<LPoint> {
    public int compare(LPoint pt1, LPoint pt2) {
        /* compare pt1 and pt2 lexicographically by y then x */
    }
}
```

Beware when trying to compare points for equality. Given two `LPoint` objects `p` and `q`, the operation `p == q` just checks whether they reference the same block of memory. Instead, you can use the `equals` function in the `Point2D` class. To access this, you first need to convert the labeled points into regular points, and then they can be compared: `p.getPoint2D().equals(q.getPoint2D())`.

[4] **Hint:** Given that you already sorted your points, a convenient way to do this is to use Java's `subList` function, which allows you to treat a Java `ArrayList` as two separate sublists. Be careful when using it. The starting index of the sublist is inclusive, but the ending index is non-inclusive.

Inserting and Deleting: To insert a new point in the sliding-midpoint tree, we first search for the point. If we find it in the tree, then we throw an exception. If not, the search ends at an external node. If this node has no point, then we store the point here. If it has a point, then we create a Java `ArrayList` with two points, the existing point and the newly inserted point. We then invoke the `bulkCreate` function described above, and replace the external node with the resulting subtree.

Deleting a point is very simple. We first search for the point to be deleted. If we do not find it, we thrown an exception. Otherwise, it will be found in some external node. We simply make the node empty by setting the point reference in this node to `null`. This will produce a hole in the tree. If we are lucky, a subsequent insertion may fill in this hole. But if not, we may have many holes. We describe a process below for eliminating these holes by rebuilding the tree.

Keeping Balance: As points are inserted to and deleted, the tree will become unbalanced and may have many holes resulting from deletions. (In the context of the sliding-midpoint rule, good balance is reflected in having cells that are nearly square. The height of the tree is not important.) We balance subtrees by invoking the `bulkCreate` function to rebuilding them from scratch. We use two different mechanisms to trigger rebuilding, one for insertion and one for deletion.

For insertion, Intuitively, if the number of insertions is high relative to the subtree's size, then the tree may be out of balance. To remedy this, we rebuild the entire subtree using the `bulkCreate` function described above.

There are two ways that subtree rebuilding is triggered.

Insertion: Each internal node maintains two quantities. Its *size* stores the total number of points within the subtree rooted at this node, and its *insertion counter* stores the number of insertions that have taken place into this subtree since it was created. Each time we insert a point that lies within the subtree rooted at this node, we increment both its insertion counter and its size. When we delete a point, we decrement its size, but its insertion counter is unchanged. Rebuilding is triggered by the relationship between these quantities, as described below.

When the tree is first constructed, the user provides an integer parameter called the *rebuild offset*, which never changes. After we successfully perform an insertion of some point `p` and update the insertion counters and subtree sizes, we retrace the search path top-down from the root to the external node containing `p`, and for each node `u` on this path, we check whether

```
u.insertionCounter > (u.size + rebuildOffset)/2
```

Ignoring the rebuild offset, intuitively, this means that we have inserted sufficiently many points that this subtree may be out of balance. The purpose of the rebuild offset is to provide the user with a bit of fine control, so we are not constantly rebuilding small subtrees.

If this condition is not satisfied by any node, then there is nothing to do. Otherwise, for the first node `u` that satisfies this condition, we rebuild its subtree from scratch. To do this, we perform a traversal of the tree, storing all of its points in a Java `ArrayList`, and then we invoke `bulkCreate` on the resulting list and the existing cell for this node. This subtree replaces the current one. All internal nodes in the rebuilt subtree should have their insertion counters reset to zero, and all sizes should be correctly computed. (It is important to note that the process runs *top-down*, and it stops with the first rebuilding.)

Deletion: Deletion is handled through a more global process. We maintain a single counter, called the *deletion counter*, for the entire tree. Each time a successful deletion is performed, we increment this counter. Following a successful deletion, we check whether the deletion counter is strictly larger than the number of points currently in the tree. If so, we rebuild the entire tree.[5]

Operations: You are to fill in the missing details of the file `SMkdTree.java`, which we will provide to you. Here is the public interface.

`SMkdTree(int rebuildOffset, Rectangle2D rootCell):` This constructs a new (empty) `SMkdTree` with the given rebuild offset and bounding box.

`void clear():` This resets the tree to its initial (empty) condition.

`int size():` Returns the number of points in the tree.

`int deleteCount():` Returns the current value of the deletion counter, described above. (This is for testing and debugging purposes, since the user need not be aware of its value.)

`LPoint find(Point2D q):` Determines whether a point coordinates `pt` occurs within the tree, and if so, it returns a reference to the associated `LPoint`. Otherwise, it returns `null`.

`void insert(LPoint pt) throws Exception:` If the given point lies outside the root's bounding box (given in the constructor), this throws an `Exception` with the message `"Attempt to insert a point outside bounding box"`. If there exists a point with the same coordinates already in the tree, it throws an `Exception` with the message `"Insertion of duplicate point"`. Otherwise, it inserts the given in the tree, through the process described above. (Note that this may involve rebuilding one of the subtrees.)

`void delete(Point2D pt) throws Exception:` If there is no point with the given coordinates, this throws an `Exception` with the message `"Deletion of nonexistent point"`. Otherwise, it deletes the point from the tree by the process described above.

`ArrayList<String> list():` This operation generates a right-to-left preorder traversal of the nodes in the tree. There is one entry in the list for each node of the tree. The output is described below:

Internal nodes: Depending on whether the cutting dimension is x or y, this generates either:

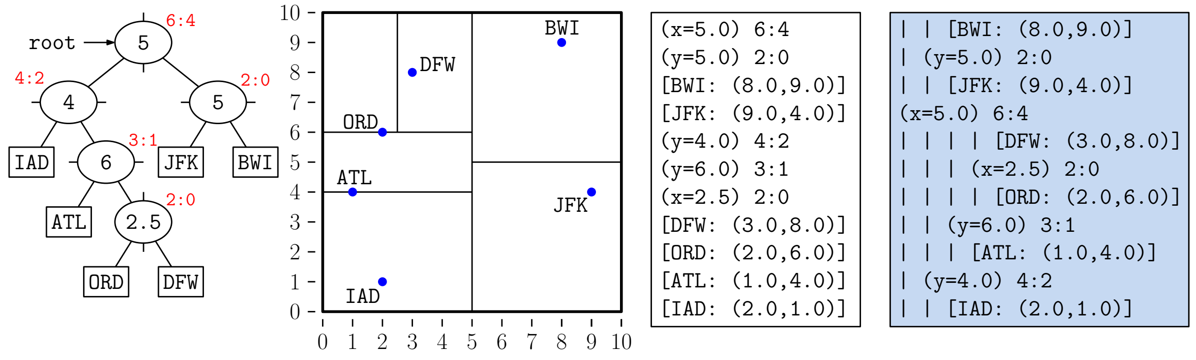
```
"(x=" + cutVal + ")␣" + size + ":" + insertCt     – or –
"(y=" + cutVal + ")␣" + size + ":" + insertCt
```

where `cutVal` is the node's cutting value, `size` is the size of the node's subtree, `insertCt` is the value of the node's insertion counter, and ␣ is a space.

[5]You might wonder why we use different mechanisms for insertion and deletion. We will discuss this when we cover the topic of *scapegoat trees*, which use a similar rebuilding process.

External nodes: If the node contains a non-null point, called `point`, this generates the string `"[" + point.toString() + "]"` (where we are using the `toString` function provided by the `LPoint` object). If the point is null, this generates the string `"[null]"`.

An example of the result of list is shown in Fig. 3(c). Our command handler will convert this into a more readable indented form as shown in Fig. 3(d).[6]



```
(x=5.0) 6:4
(y=5.0) 2:0
[BWI: (8.0,9.0)]
[JFK: (9.0,4.0)]
(y=4.0) 4:2
(y=6.0) 3:1
(x=2.5) 2:0
[DFW: (3.0,8.0)]
[ORD: (2.0,6.0)]
[ATL: (1.0,4.0)]
[IAD: (2.0,1.0)]
```

```
| | [BWI: (8.0,9.0)]
| (y=5.0) 2:0
| | [JFK: (9.0,4.0)]
(x=5.0) 6:4
| | | | [DFW: (3.0,8.0)]
| | | (x=2.5) 2:0
| | | | [ORD: (2.0,6.0)]
| | (y=6.0) 3:1
| | | [ATL: (1.0,4.0)]
| (y=4.0) 4:2
| | [IAD: (2.0,1.0)]
```

(a) (b) (c) (d)

Figure 3: (a) An SMkd-tree, (b) the associated subdivision, (c) the output of the `list` function, and (d) formatted output generated by the command handler. The red values associated with each node are the size and insertion counter.

`LPoint nearestNeighbor(Point2D center):` This function returns a reference to the closest point in terms of squared Euclidean distances to the given query point. If the tree is empty, it returns `null`. If there are multiple nearest neighbors at the same distance (see Fig. 4), it returns the point that is smallest in (x, y) lexicographical order (smallest x with ties broken in favor of smallest y).

`ArrayList<LPoint> nearestNeighborVisit(Point2D center):` This is the same as the above function `nearestNeighbor`, but it is designed for testing/debugging. It performs the same search as `nearestNeighbor`, but it returns a list of all the points visited by the search algorithm. For the sake of consistency, these points should be listed in increasing (x, y) lexicographical order.[7]

Computing distances involves computing square roots, which is both unnecessary and introduces floating-point errors. Instead, you should compute squared Euclidean distances. (This does not change the identity of the closest point). To assist you, the `Point2D` and `Rectangle2D` classes both provide a utility function `distanceSq(Point2D pt)`, which computes the squared distance from the current object to point `pt`.

[6]**Hint:** Differences between your tree and ours will first show up in the unformatted list output. Rather than trying to interpret the error there, instead look for the difference in the formatted output that is closest to the root. That will indicate which subtree is incorrect.

[7]**Hint:** You can implement both functions using just one helper function. For example, you can use the return value from the helper function to store the closest point seen so far in the search, and pass a reference to the array list of visited point as an argument. As you visit successive points, just append them to the array list. Depending on the desired result, you will either return the closest point or the sorted array list.

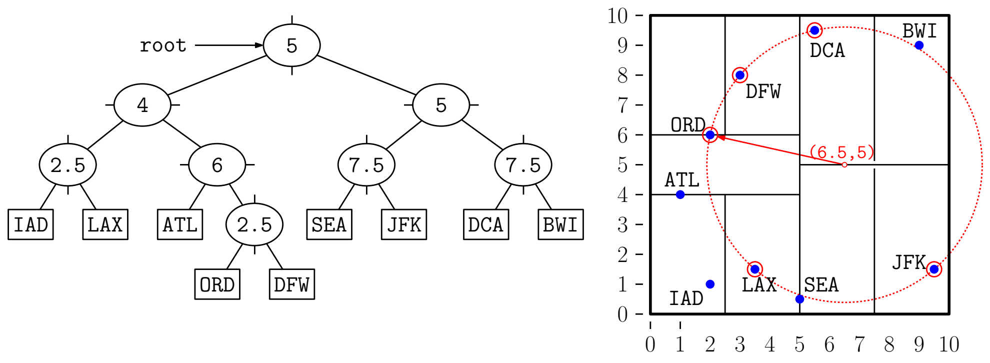


Figure 4: Nearest-neighbor query. Among the equally close candidates (`ORD`, `DFW`, `DCA`, ...), `ORD` is smallest in lexicographical order.

You should adapt the algorithm given in class to the context of extended trees. In particular, when visiting an internal node, you should first visit the subtree that is closer to the query point. Second, you should avoid visiting nodes that you can infer cannot contain the nearest neighbor. To do this, keep track of the closest point seen so far, and if node's cell is farther away from the query point than this, then you should avoid visiting the node. (Or if you visit it, you should discover this and return immediately.)

Parts: For the first part of the assignment (due Wed, Apr 5), implement the constructor and the functions `size`, `find`, `insert`, `clear`, and `list`. We will use test cases `test01-input.txt` and `test02-input.txt` for testing this part. For the second part (due Wed, Apr 19), implement all the functionality.

Skeleton Code: As usual, we will provide skeleton code on the class Projects Page. You will need to fill in the implementation of the `SMkdTree.java`. We also provide some utility classes (e.g., `Point2D` and `Rectangle2D`). You should not modify any of the other files, but you can add new files of your own. For example, if you wanted to add additional functions to any of the classes, such as `Point2D` or `Rectangle2D`, it would be preferable to create an entirely new class (e.g., `MyRect2D`), which you will upload with your submission.

As with the previous assignment, the package "`cmsc420_s23`" is required for all your source files. As usual, we will provide a driver programs (tester and command-handler) for processing input and output. You should not modify the signature of the public functions, but you are free to set up the internal structure however you like.

Efficiency requirements: (10% of the final grade) The function `size` should run in constant time, the dictionary operations `find`, `insert`, and `delete` should run in time proportional to the height of the tree, assuming no rebuilding takes place. The time for rebuilding should be $O(m \log m)$ for each node you rebuild, where m is the number of points in the subtree being rebuilt. This means that you have enough time to sort all the points within a node during the rebuilding process. You may assume that the Java `Collections.sort` function operations in $O(m \log m)$ time on a list of size m. The nearest neighbor functions should be true in running time to the algorithm given in class. For `nearestNeighborVisit`, you can also use an additional $O(m \log m)$ time to lexicographically sort the m points that are reported.

Style requirements: (5% of the final grade) The style requirements are essentially the same as for the previous assignment. Your code should be relatively clean, and comments should be provided to explain important functions. You may refer to our canonical solutions for guidance (but you do not need to be quite as excessive in commenting as I am).

Testing/Grading: As always, we will provide some sample test data and expected results along with the skeleton code.

As before, we will be using Gradescope's autograder for grading your submissions. **Unless you created your own additional files, you should only submit one file, `SMkdTree.java`.** If you created any additional files for utility objects, you will need to upload those as well.

Challenge Problem: Implement an iterator for your tree, which generates all the points stored in the tree according to an inorder traversal. To do this, generate a (private) inner class that implements the Java interface `Iterator<LPoint>` (defined in Java.util). You may name this whatever you like, but for concreteness, let's call it `LPointIterator`. Your `SMkdTree` should support a public function `LPointIterator iterator()`, which returns such an iterator object. Your `LPointIterator` class should implement the following functions:

`LPointIterator()`: A constructor, which initializes the iterator.

`boolean hasNext()`: Returns `true` if there are elements remaining to be iterated.

`LPoint next()`: Returns a reference to the next point in the the tree, according to an inorder traversal. If there are no more point (that is, if `hasNext()` returns `false`), this throws a `NoSuchElementException` (which is defined in Java.util).

Note that the iterator should not return references to `null` points. If points have been deleted from your structure, your iterator should jump over them, and return the next available non-null point. Your iterator should take $O(h)$ space, where h is the height of your tree. If there are no empty external nodes, the operations should run in time proportional to the height of the tree. If there are empty external nodes, each operation can take $O(h \cdot e)$ time, where e is the number of empty external nodes you need to skip over.

Hint: A natural approach is to represent the iterator as a reference to a node in your tree, and the `next()` function advances this pointer to the next external node in inorder, skipping over empty external nodes. Implementing this in $O(1)$ space requires that you have parent links (or some other way of navigating the tree from any node). Alternatively, you can store a stack of ancestor nodes in $O(h)$ space, and use this to guide the search for the next node according to an inorder traversal. Either approach is fine for full credit.

Beware: If you are considering exploiting an auxiliary structure (such as a Java `ArrayList`) that provides its own iterator, you will need to do this within the style/efficiency requirement given above. For example, a `HashMap` supports fast insertion and deletion, but sorting it will take too long. List structures like `ArrayList` and `LinkedList` can be maintained in sorted order, but insertions and deletions are probably too slow.

Insertion/BulkCreate Example: To get a better understanding of how insertion works with bulk creation, suppose we have external node that stores the labeled point MSP at coordinates $(2.5, 8.5)$ (see Fig. 5(a)). It lies within a cell whose lower corner is $(2, 6)$ and whose upper corner is $(5, 10)$. We will represent rectangles by giving their lower-left and upper-right points, so this is $[(2, 6), (5, 10)]$. Suppose we want to insert a new point SEA with coordinates $(4.0, 9.5)$ (see Fig. 5(b)).

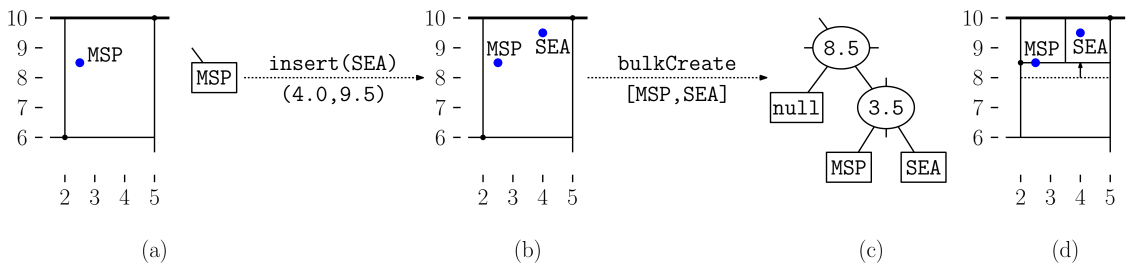


Figure 5: Insertion and bulk create. The insertion of SEA at $(4.0, 9.5)$ causes the external node with MSP to split. Given these two points, bulkCreate generate a a four-node subtree, which replaces the original external node.

We create a 2-element ArrayList storing these two points, and invoke bulkCreate on them with the current cell.

`bulkCreate([MSP,SEA], [(2,6),(5,10)])`: Since this cell is taller than wide, we take the cutting dimension to be 1 and split horizontally. The ideal split would be at $y = 8$, but since both points lie above this, we "slide" the splitting line up to $y = 8.5$ so it passes through MSP. We generate an internal node with `cutDim = 1` and `cutVal = 8.5`, and partition the points about $y = 8.5$. By our convention, the point MSP goes in the right subtree as does SEA. The left side of the partition is empty. We then invoke bulkCreate recursively to generate its children:

- `bulkCreate([], [(2,6),(5,8.5)])`: There are no points, so this generates an empty external node and returns.
- `bulkCreate([MSP,SEA], [(2,8.5),(5,10)])`: Since this cell is wider than tall, we take the cutting dimension to be 0 and split vertically through the midpoint at $x = 3.5$ (no sliding needed). We partition the points, with MSP on the left side and SEA on the right. We create a new internal node with `cutDim = 0` and `cutVal = 3.5`. We then invoke bulkCreate recursively to generate its children:
  - `bulkCreate([MSP], [(2,8.5),(3.5,10)])`: This generates a single external node with MSP.
  - `bulkCreate([SEA], [(3.5,8.5),(5,10)])`: This generates a single external node with SEA.

The final subtree is shown in Fig. 5(c) and the resulting subdivision of the cell is shown in Fig. 5(d). The insertion procedure returns this subtree to replace the original external node with MSP.