C.S. 252 Prof. Roberto Tamassia
Computational Geometry Sem. II, 1993–1994

Priority Search Trees - Part II

Lecture 10
Date: May 22, 1994
Scribe: Michael Shapiro

1 Introduction

In the previous lecture, we looked at *priority search trees* and their application to interval intersection queries. Specifically, we learned how to construct and query a *priority search tree*. In this lecture, we will see that another nice feature of this data structure is that it can easily be made *dynamic*.

We will begin with a dynamic priority search tree constructed from 10 points, labeled a through j, each of which corresponds to an interval, as shown in the previous lecture. The example points and tree are shown in Fig. 1.

2 Observations

Each point corresponds to one of the leaves of the tree, and the order of leaves from left to right corresponds to the order of the x-coordinates of the points. The interior nodes of the tree store the point with the greatest y-coordinate represented by a leaf below this interior node, which is not already stored by an ancestor of this interior node. Overall, we only use n interior and leaf nodes.

If a point is stored at an internal node, then the leaf node which corresponds to this point is considered a *placeholder* for this point. A total of $n - 1$ nodes are left as placeholders, in addition to the n interior and leaf nodes. This is a key difference between the dynamic priority search tree and the standard priority search tree we saw in the previous lecture, where only one node of the tree corresponded to each point. Note also that some interior nodes may not store points.

We will represent the dynamic priority search tree with a *red-black tree*, in order to facilitate insertions and deletions, which we will examine shortly. As a result, we know that our tree will be balanced overall. However, we should note that the subtree of a given interior node which stores a point may be unbalanced, although its height is still at most logarithmic.

3 Three-Sided Range Query

Before we examine the details of how our dynamic priority search tree is maintained, let us briefly verify that the dynamic priority search tree offers the same advantages in terms of

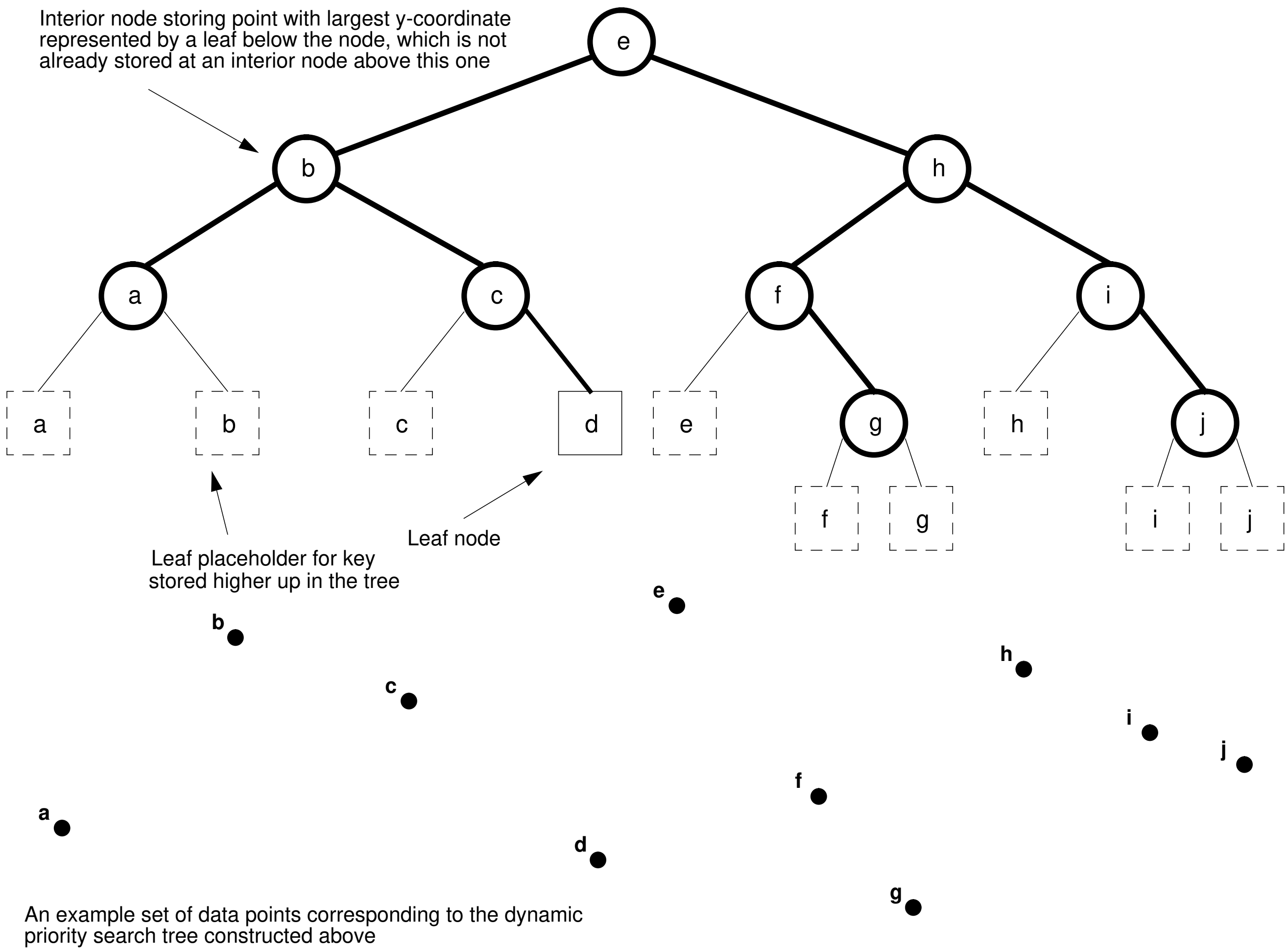


Figure 1: 12 query points and corresponding dynamic priority search tree.

time complexity for search operations as the priority search trees we saw in the last lecture. An example of a three-sided range query is shown in Fig. 2.

We begin at the root of the tree, and first check the y-coordinate of this node against the y-coordinate of the baseline of the query interval. As long as the y-coordinate of the current node is greater than the y-coordinate of the horizontal segment defining the base of our three-sided range, we will continue down the tree. At the root, we see that point e is to the left of the query range, so we traverse to its right subtree.

When we reach the interior node which stores point h, we report point h; this point lies within the query region. Since we are at a successful node, we must traverse to *both* subtrees. When we examine the left subtree of h, we find point f, which also lies inside the query region. There are no points stored in the left subtree of f, so we only need to traverse to the right subtree. Our search terminates at point g because this point lies below the baseline of the query interval. When we search the right subtree of h, we discover that point i is to the right of the query interval. We need to traverse to the left subtree of i, but no points are stored in this subtree, and so our search ends here.

We can see from our example query that the search operation only visits the used nodes

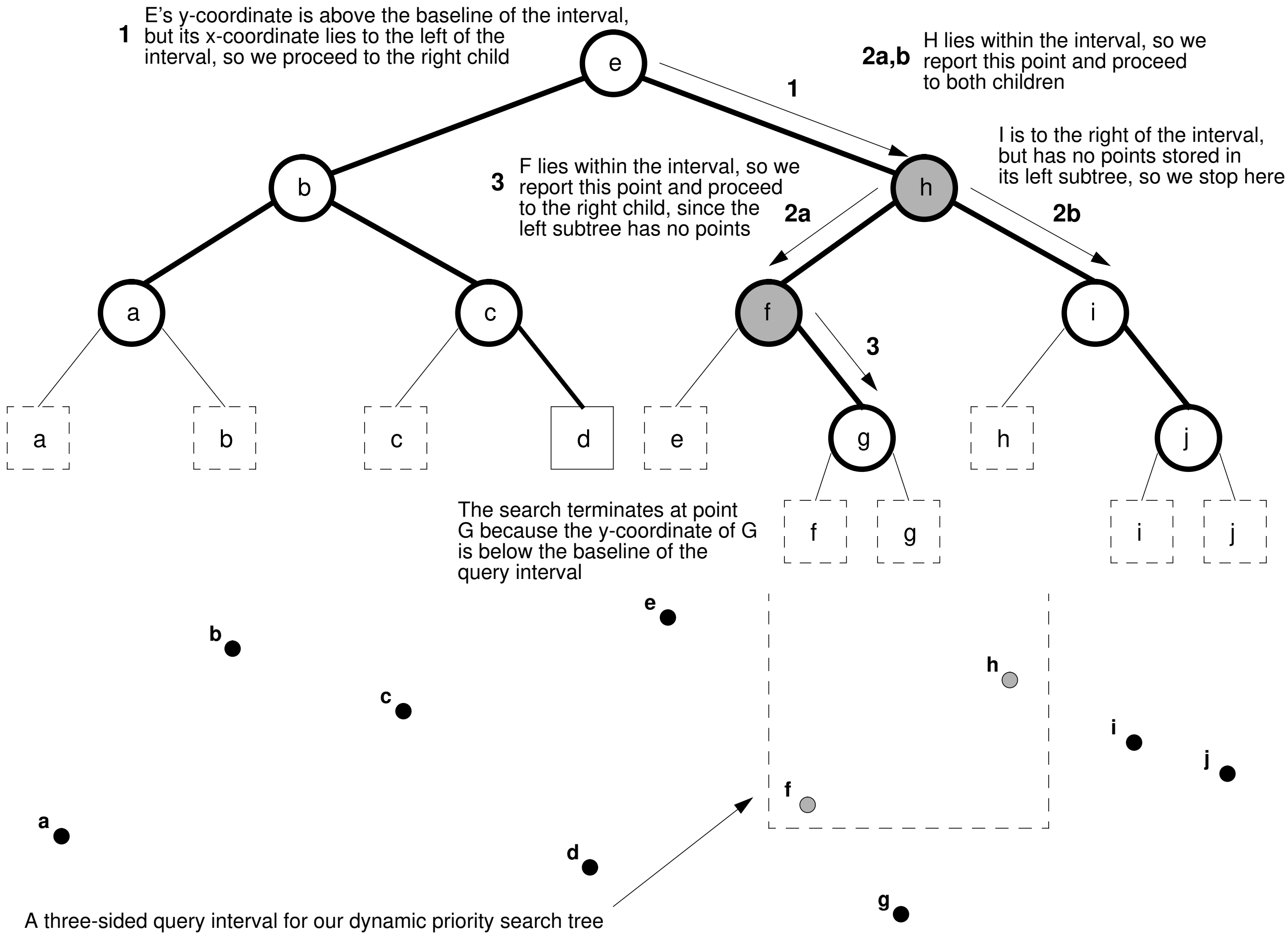


Figure 2: A three-sided range query in the dynamic priority search tree.

of the tree, and thus only traverses the bold lines in the figure. The height of the tree is guaranteed to be $O(log\ n)$, so we know that the search operation takes $O(log\ n + k)$ time, where k is the number of points successfully reported. In general, our search path can be characterized by the pattern shown in Fig. 3. We traverse down a single path which is of length proportional to the height of the tree until we reach an internal node which is inside the query interval. Since the tree is known to be balanced, this path is no longer than $log\ n$. Then we proceed down two separate paths of length $O(log\ n)$ until we reach nodes which are below the horizontal segment defining the bottom of the query interval, or until we reach a leaf node. The nodes in the interior subtrees of these paths will be reported as being within the query region, except possibly for the leaves within these subtrees, which may lie below the base of the query interval.

4 Construction

We can construct the dynamic priority search tree from an initial set of points using a bottom-up construction method similar to the bottom-up construction of a heap. First, we

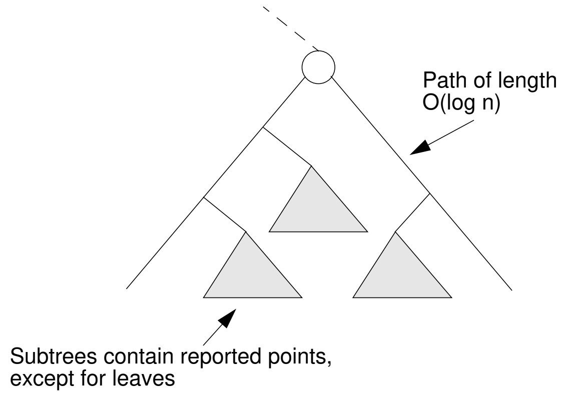


Figure 3: General search path pattern in the priority search tree.

will need to employ any of the well-known efficient sorting algorithms to sort the points by x-coordinate. Now we can associate each point with a placeholder in the priority search tree. Next we select pairs of placeholders and compare them to one another in terms of their y-coordinates, as seen in Fig. 4. The "winners" of this first round of the "tournament" are the points in each pair with the greater y-coordinate, and these points move up to be represented by a new internal node at one level higher in the tree.

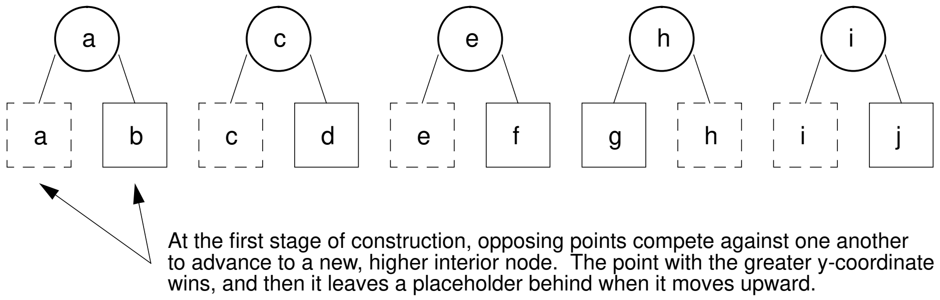


Figure 4: The first phase of the bottom-up construction "tournament".

At the next level of the tree, we perform the same comparisons as before to determine which nodes will advance to the third level. At most $n/2$ nodes are compared at this level. In addition, when a point moves to a new level higher in the tree, this may create a new opening at the current level of the tree. If this occurs, a point which had previously "lost" in the tournament may now be eligible to move up. We must also remember that some interior nodes in the tree may not represent any points, and will remain empty when the construction is complete.

Similar to the bottom-up construction method for a heap, this process will take time proportional to the number of points, or $O(n)$. However, in order to perform the bottom-up construction, we had to first sort the data points, which takes time $O(n\ log\ n)$ using any of the known efficient sorting algorithms. So the time complexity of construction of the dynamic priority search tree is $O(n\ log\ n)$ overall.

5 Insertion

In order to dynamically insert a point into the priority search tree, we must perform the following steps:

1. Add a new leaf (placeholder) for the new point
2. Perform a pushdown operation with the new point starting at the root
3. Perform restructuring in order to maintain the balance of the tree

In Fig. 5, we see that a new point c' has been added to our original set of data points. Our first step will be to add a new placeholder for this point, as shown in Fig. 6. As we discussed earlier, every point in the priority search tree will have a placeholder.

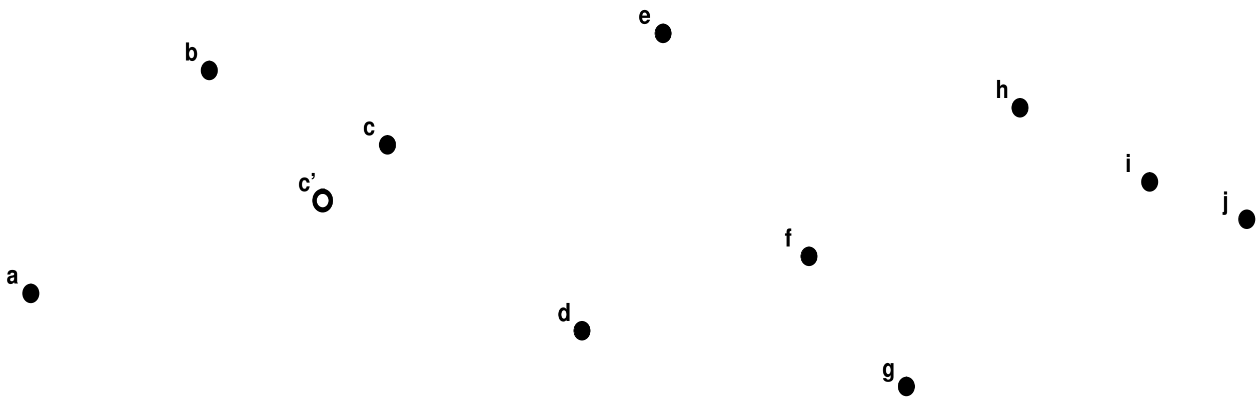


Figure 5: A new point c' has been added to the data set.

For the first step of the insertion algorithm, we can determine the proper location for the placeholder because we know that the priority search tree, viewed in a one-dimensional sense, maintains the property of a binary search tree on the x-coordinates of the points. We traverse along a single path from the root of the tree to a leaf node, comparing the x-coordinate of the new point c' to the x-coordinate of the point represented by the given internal node. If the x-coordinate of c' is less than that of the given point, we proceed to the left subtree; otherwise we proceed to the right subtree.

When the first step of our insertion algorithm reaches a leaf of the tree, we add a new internal node in place of this leaf, and make the existing leaf one of the children of this new internal node. Then we add a new leaf to the tree as the other child, and store the new point c' in this leaf. In Fig. 7, we see that a new empty internal node has been added as the left child of c, and this internal node has a leaf for c' as its left child, and a placeholder for c as its right child.

For the second step of our insertion algorithm, we need to perform a pushdown operation on the new point starting again at the root, to see if the new point is eligible to be represented by an internal node higher up in the tree. The results of this pushdown operation are shown in Fig. 8. At each step of the pushdown operation, we compare the y-coordinate of the point to be inserted with the y-coordinate of the point represented by the given internal node.

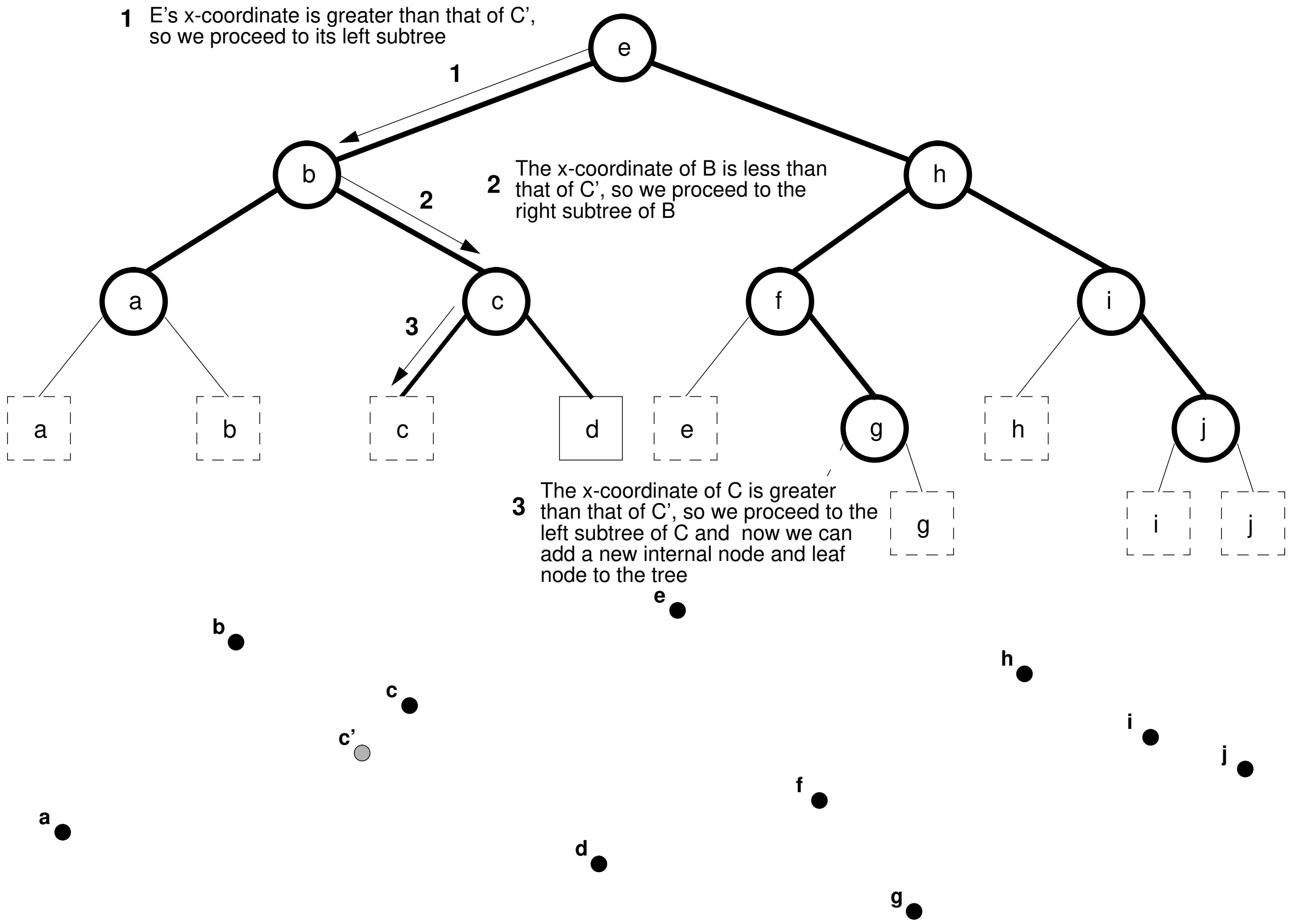


Figure 6: Searching the tree to add a new leaf node for c'.

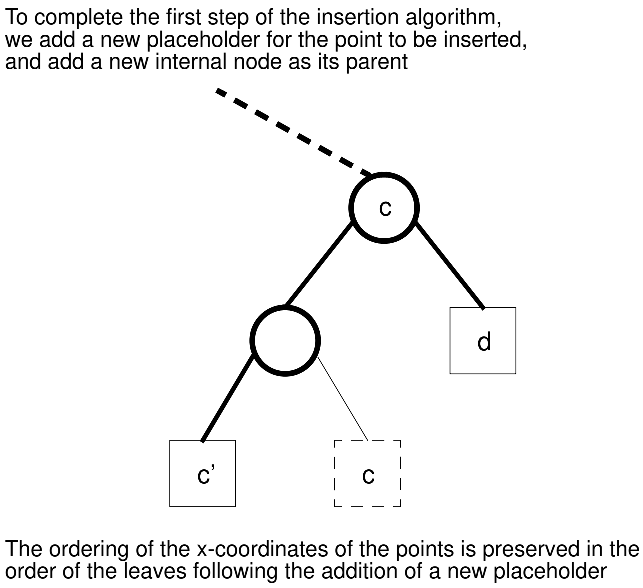


Figure 7: Point c' occupies the new leaf node.

If the y-coordinate of the point to be inserted is less than that of the point stored in the internal node, then we push the point to be inserted into the left or right subtree of the internal node. We decide which subtree to proceed to by comparing the x-coordinate of the point to be inserted with the x-coordinate of the point in the internal node. However, if the y-coordinate of the point to be inserted is greater than the y-coordinate of the point in the internal node, then we store the point to be inserted in this internal node, and take the point which was formerly represented by this internal node and continue the pushdown operation with this point instead.

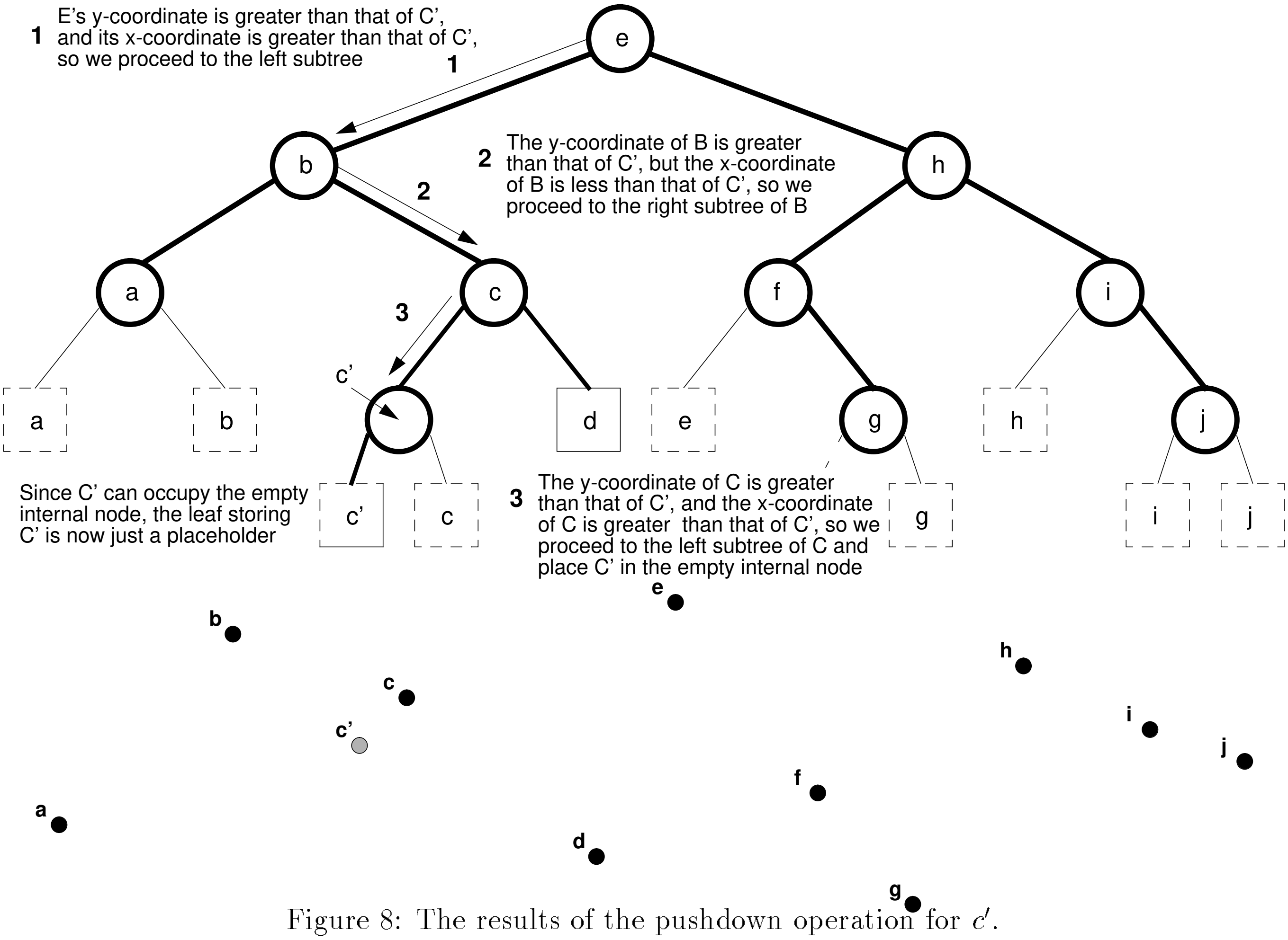


Figure 8: The results of the pushdown operation for c'.

Before our insert operation can be considered to be complete, we must perform any restructuring needed to rebalance the priority search tree. In the example insertion for c', only a promotion is necessary to maintain the balance of the tree. Later we will see an example of an insertion which requires a rotation to balance the tree. Since our priority search tree is represented as a red-black tree, we know that only a constant number of rotations will be required in order to maintain the balance of the tree.

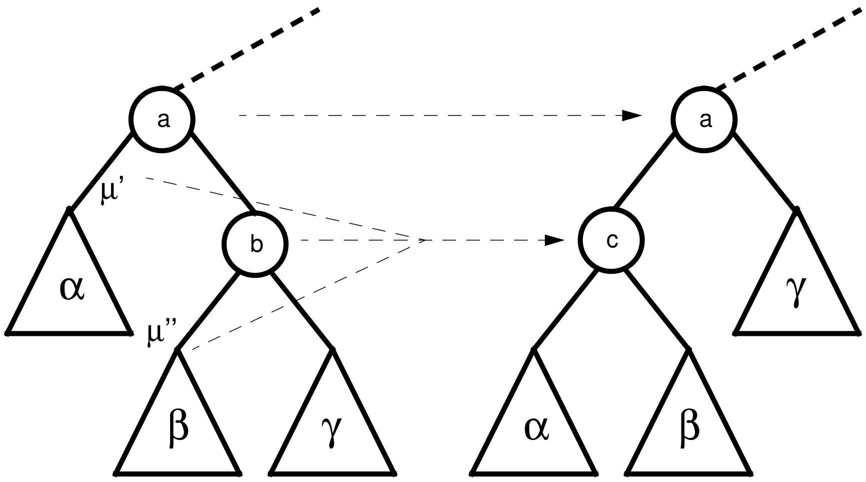
So far we have seen that the first step of the insertion algorithm is a simple binary search traversal on the x-coordinate of the new point, which traverses a single path of length

$O(log\ n)$, and then adds a new leaf node and new internal node. In the second step of insertion, we perform a pushdown operation with the new point, which also traverses a single path of length $O(log\ n)$ in order to store the new point in the tree. Finally, we have claimed that at most a constant number of rotations will be required to maintain the balance of the tree in the final step of the insertion algorithm. We will show in the next section that such rotations can be done in $O(log\ n)$ time, and so we can conclude that the insert operation takes time $O(log\ n)$ overall.

We can see that by using a binary search and then a pushdown operation on the priority search tree during insertion, we have preserved the ordering of both the x-coordinates and the y-coordinates of all points stored in the tree, and thus we can conclude that the insertion algorithm is correct in maintaining the properties of the priority search tree. We must also be concerned with maintaining the balance of our tree as well. In the next section when we discuss rotations, we will also show that rotations will preserve the necessary priority search tree properties.

6 Rotations

In order to keep our tree balanced, we will need to perform right rotations, left rotations, right-left rotations, and left-right rotations. As an example, we will consider a left rotation as shown in Fig. 9. In our example, we need to perform a left rotation about node a . We will restrict our discussion of rotations to this example of a left rotation, because a right rotation is symmetrical to a left rotation, and the double rotations each consist of a single left rotation and a single right rotation.



In the left rotation, node a remains at the same location, and subtree γ becomes the new right subtree of a. Then we select the higher point from μ' (the highest node in α) and μ'' (the highest node in β), and perform a pushdown operation with this point at node c.

Figure 9: An example left rotation.

After our example rotation has been performed, a remains in the same position relative to the rest of the tree. The subtree labeled γ becomes the new right subtree of node a after the rotation, and the node labeled c becomes the new left child of a. We then select the point with the greatest y-coordinate among node μ', the highest node in subtree α, and node μ'', the highest node in subtree β, and perform a *pushdown* operation with this point beginning at node c.

Observe that the properties of the priority search-tree we have already established are preserved after the rotation. Internal nodes in subtree γ are to the right of and below node a before the rotation, and after the rotation. Internal nodes in subtree α are to the left of and below node a before the rotation, and after the rotation. Finally, since node c is the parent of subtrees α and β after the rotation, and we perform a pushdown operation beginning at node c to determine which point should be represented by this internal node.

Now we are ready to make some observations about the time complexity of rotations in the priority search tree. Since we have already chosen to implement our priority search tree as a red-black tree, we know that only a constant number of rotations will be required to balance the tree, and we also know that the restructuring of the subtrees can be done in constant time. If a pushdown operation is needed, it can be done in time proportional to the height of the tree. Again, our implementation using a red-black tree guarantees and overall height of $O(log\ n)$.

Since we already showed that the first two steps of the insertion algorithm only take time $O(log\ n)$, we can now conclude that the overall time complexity for insertion and accompanying rotations is O($log\ n$). If we were to select another balanced tree algorithm for our implementation, such as *AVL-trees*, we would have an overall time complexity of $O(log^2 n)$ since $O(log\ n)$ rotations could occur during an insertion.

7 A Complete Example

Let us now consider a complete example of an insertion operation which requires a rotation. We will start with our original priority search tree and insert a new point g', whose location relative to the other points is shown at the bottom Fig. 10. In order to clarify why a rotation is necessary, edges which are colored red in the underlying red-black tree representation are shaded in this figure.

First we need to insert a new placeholder for the new point g'. We perform a binary search through the priority search tree based on the x-coordinates in order to determine where to add the new internal node and new leaf for point g'. The path of this search is shown in Fig. 10. Now we must add a new internal node and a placeholder for point g', and then perform a pushdown operation with point g' beginning at the root of the tree. This results in the tree structure shown on the left-hand side of Fig. 11. Here we see that point g' is represented by the new internal node, and its placeholder is the right child of the new internal node. But the incoming edge of a new internal node in a red-black tree is colored red, and the incoming edge of the internal node representing g was previously red, so now a left rotation must be performed.

According to the structure of the tree, we will need to perform a right-rotation to re-balance the tree. The right-hand side of Fig. 11 shows the tree structure after the rotation. We can see that the internal node representing point i remains the same, and that each of its children is now an internal node. Initially, the left subtree of the new node i is the left subtree of the old node h'. The right child of the new node i has the placeholder for h' as

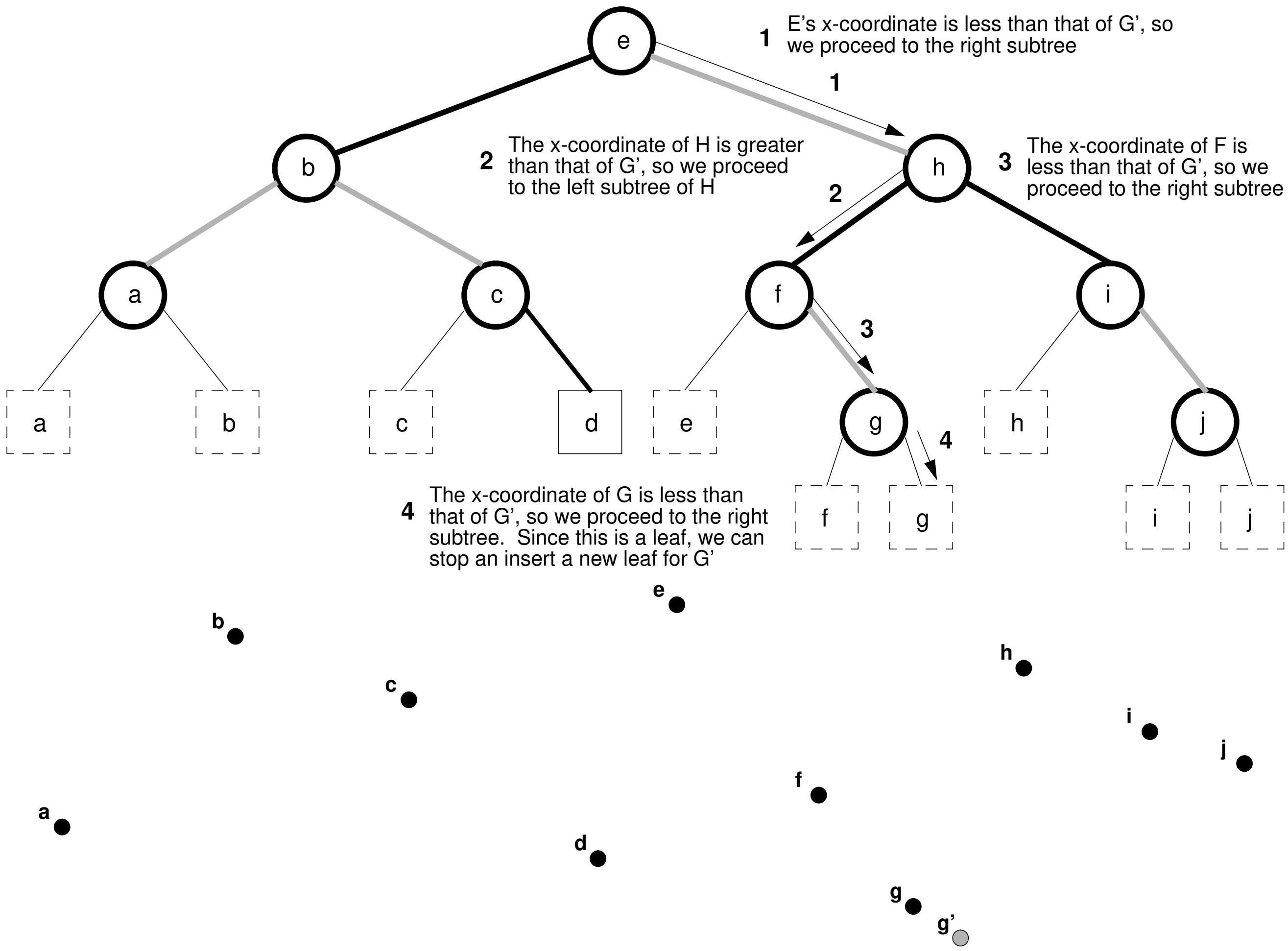


Figure 10: Searching the tree to insert a new leaf node for g'.

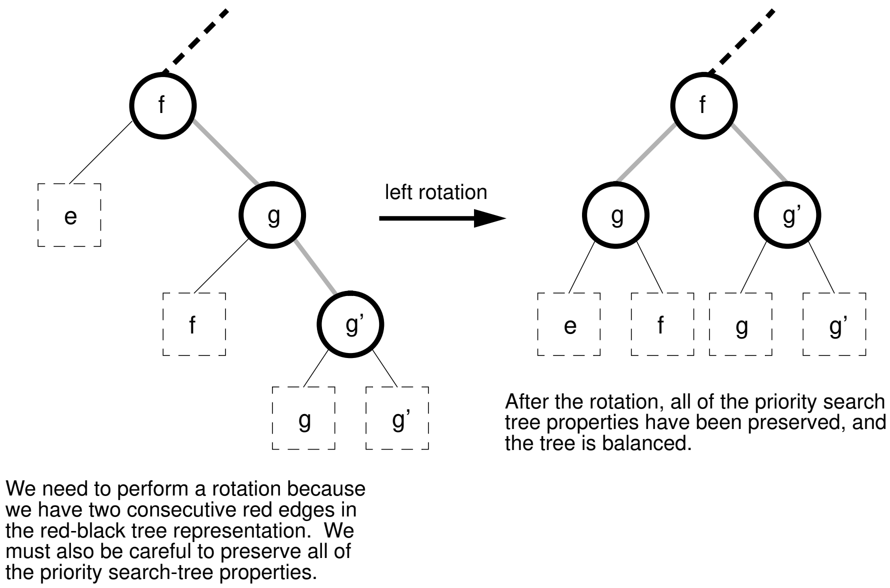


Figure 11: Performing a left rotation in order to rebalance the tree.

its left child, and the placeholder for i as its right child.

According to the system for rotations shown in Fig. 9, we need to make the subtree rooted at g' the new right subtree of f, and the internal node representing g becomes the new left child of f. The placeholders for e and f become the left and right children of g. Now we must see if a pushdown operation is needed. The candidates for the pushdown operation are the highest point represented in the subtree which contains only the placeholder for e, and the highest point represented in subtree which contains only the placeholder for f. Since no points are represented in either of these subtrees, no pushdown operation is needed.

We can verify that all of the priority search tree properties have been preserved after this restructuring by observing that the ordering of nodes by their y-coordinates has been preserved. The left-to-right ordering of the placeholders, which represents the order of the x-coordinates of the points, is the same before and after the rotation. These two observations confirm that we still have a well-formed priority search tree.

8 Deletion

The algorithm for deletion from the dynamic priority search tree is straightforward, and takes advantage of many of the techniques we have already discussed. We must perform the following steps:

1. Locate the interior node representing the point we wish to delete
2. Remove the point from this interior node, and replay a portion of the "tournament" among points below this interior node in order to replace it
3. Delete the leaf which is the placeholder for the point
4. Perform any necessary rotations and restructuring operations

An example of deletion is shown in Fig. 12 and Fig. 13.

The first step in performing a delete operation will be to locate the interior node which represents the point to be deleted. We know that this operation can be performed in time $O(log\ n)$ because of the binary search tree properties maintained by the priority search tree. Once we have located the interior node which represents the point to be deleted, we can remove this point from the tree.

Now that we have removed the point from the tree, we must adjust the points in the tree below the interior node which represented this point. We can consider this interior node to be empty, and other points below it will "replay" their part of the "tournament" to determine who is allowed to advance up into this empty interior node. The first two opponents we must consider are the left and right children of the interior node. In our example, point c has a greater y-coordinate than point a, so it advances up to the interior node which previously stored point b.

Since c has advanced upward, we now proceed down to the right subtree of the interior node and continue pulling points upward. Point c has moved upward, so the interior node which formerly represented c is now empty. This node has two leaf children, representing

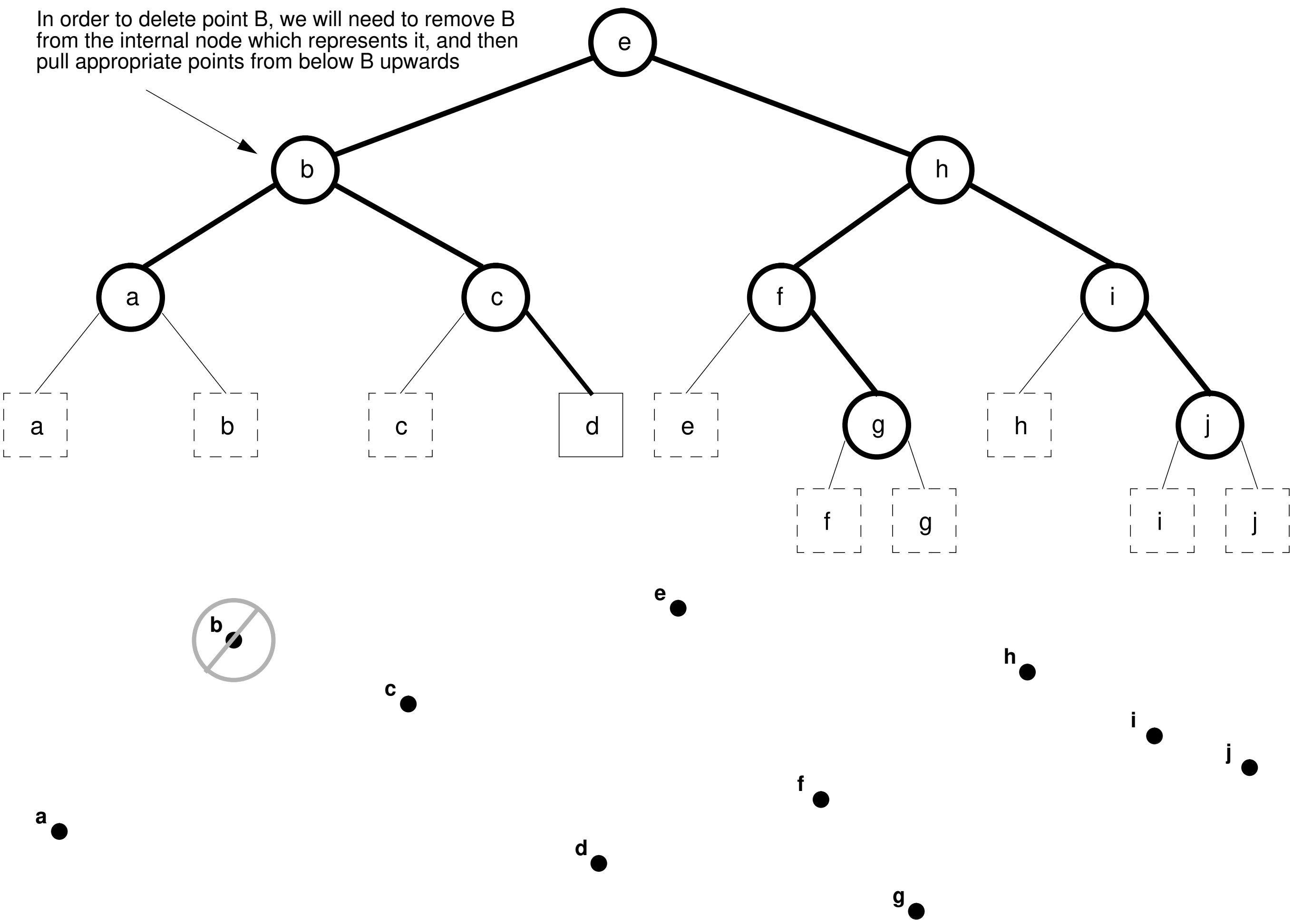


Figure 12: Selecting a point to be deleted.

points c and d respectively. However, point c is already represented higher up in the tree, so point d is the only candidate to fill this empty interior node. Since we have reached a leaf node, we are done with the pulling operation. The pulling occurs along a path which is proportional to the height of the tree, and therefore takes time $O(log\ n)$.

Similar to insertion, the deletion operation may also require a constant number of rotations in order to rebalance the tree. However, the rotations used for deletion are identical to those used for insertion, and we showed earlier that such rotations take no more than time $O(log\ n)$. Therefore we can conclude that deletion can be performed in our dynamic priority search tree in $O(log\ n)$ time.

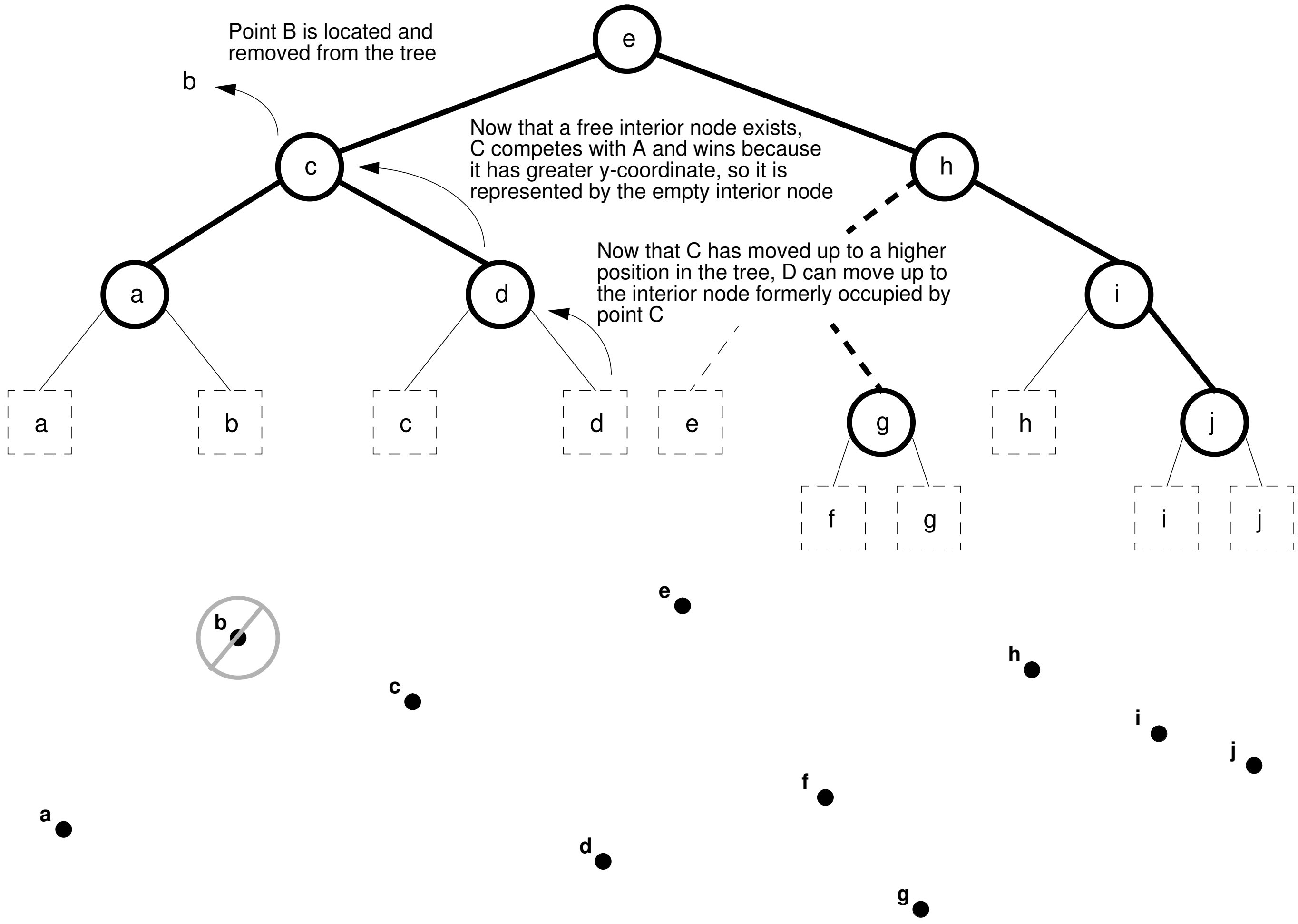


Figure 13: Restructuring the tree after a point is deleted.

9 Summary and Applications

We conclude our discussion of dynamic priority search trees with a brief summary of the operations we have discussed and their time complexities:

- The dynamic priority search tree stores n points in $O(n)$ space.
- We can perform insertion in time $O(log\ n)$.
- We can perform deletion in time $O(log\ n)$.
- Only a constant number of rotations are required for insertion and deletion, taking no more than time $O(log\ n)$ to rebalance the tree.
- We can perform queries such as a three-sided range query in time $O(log\ n + k)$, or a search for all intersecting rectangles in time $O(n\ log\ n + k)$.
- We can construct a dynamic priority search tree using a bottom-up construction method in time $O(n\ log\ n)$.

Priority search trees are useful in a wide variety of applications. McCreight [1] observes that priority search trees can be used in combination with plane-sweep techniques, to implement best-/first-fit storage allocation, and to determine intersection or containment in a set of linear intervals. Another important observation is that dynamic priority search trees can be implemented by the programmer relatively easily. It is particularly worth noting that McCreight [1] presents full source code along with his discussion of the priority search tree algorithm.

References

[1] Edward M. McCreight, "Priority Search Trees," *SIAM J. Comput.*, Vol. 14, No. 2, pp. 257–276, May 1985.

[2] Alok N. Choudhary, "Dynamic Priority Search Trees," November 1987.