C.S. 252 Prof. Roberto Tamassia
Computational Geometry Sem. II, 1992–1993

Priority Search Trees - Part I

Lecture 9
Date: March 8, 1993
Scribe: Dina Q Goldin Karon

1 Introduction

In the last lecture, we looked at *interval trees*. For *interval point enclosure problems*, they use linear space and optimal time. Today, we shall study *priority search trees*, useful for problems that involve intervals intersecting other intervals.

Our examples will be based on a set of 13 intervals a through m along a line l, shown in Fig. 1. The intervals have been drawn underneath the line so as to distinguish overlapping sections.

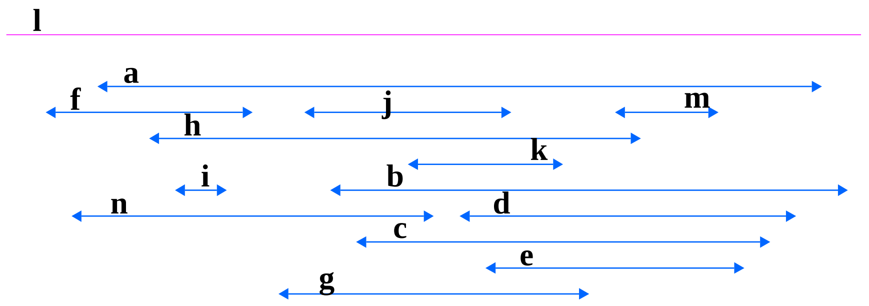


Figure 1: Thirteen intervals for interval intersection problems.

We start off by marking off on l the endpoints of each interval. Then, we extend the two endpoints of each interval diagonally till they meet (see Fig. 2).

We now have a one-to-one and onto mapping between points above the line l and intervals on l. We shall use small letters for names of intervals, and corresponding capital letters for names of points.

2 Range Queries

We have looked at *range queries* in earlier lectures. They answer questions of the following type: *Given a set of points S and a range R, find all points from S which fall within R.* A *range* is specified by providing one or more of the following: (1) an upper bound on X, (2) a lower bound on X, (3) an upper bound on Y, (4) a lower bound on Y.

Now, let us consider the following three questions on a fixed set of intervals:

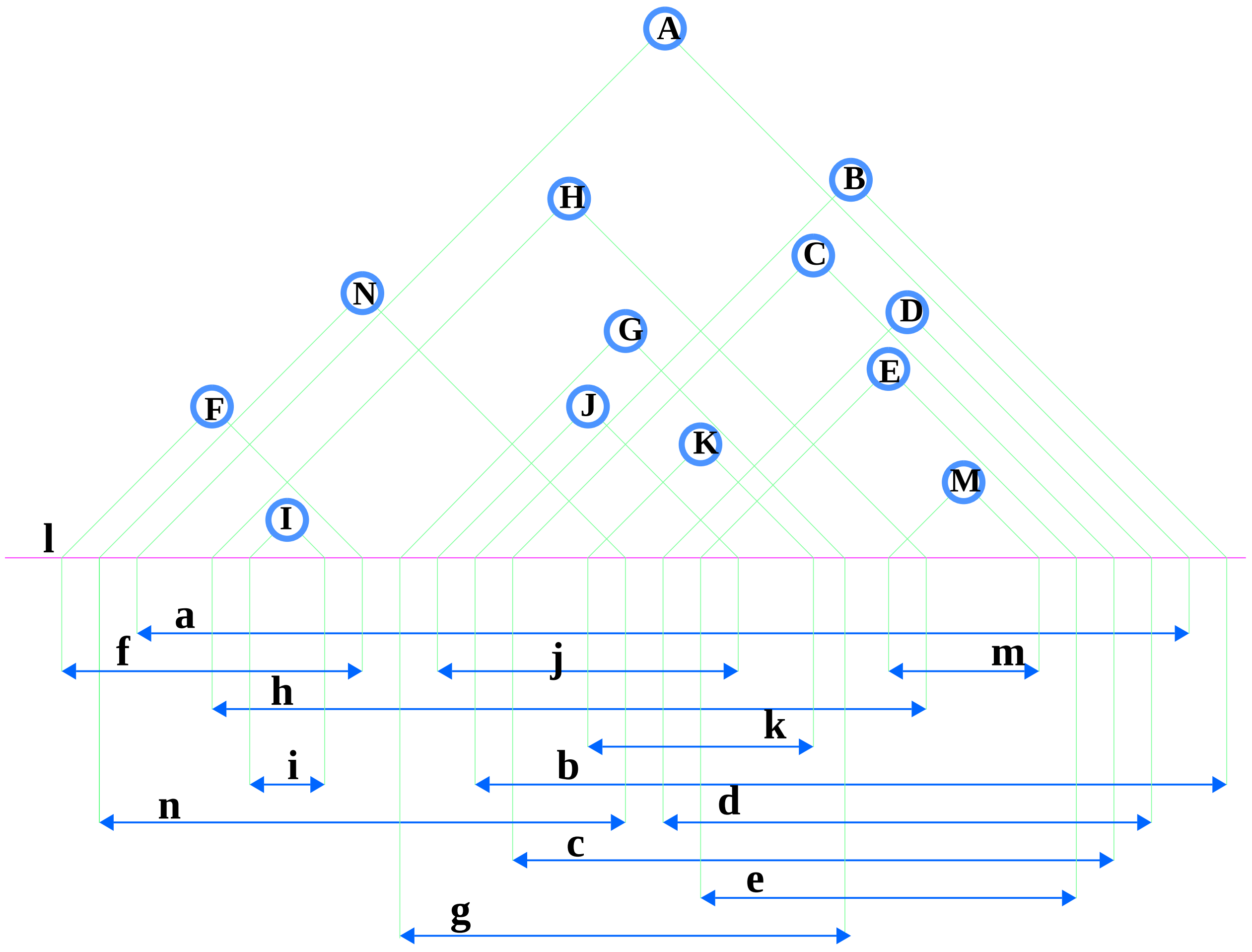


Figure 2: Mapping intervals on l to points above l.

1. Given an interval X, which intervals are contained in X?
2. Given an interval X, which intervals intersect it?
3. Given an interval X, which intervals contain X?

We will show that all these questions can be represented as range queries on the set of points created as above. For this purpose, we first want to rotate Fig. 2 by 45 degrees, obtaining Fig. 3.

The next three diagrams (Figs. 4, 5, 6) show how the above three queries on intervals are represented as queries on the corresponding points. In each case, the query is a quadrangle delineated by two rays, and the answer to the query is the set of intervals which correspond to the points that fall within the quadrangle (they are thickened in the diagrams).

In a regular range query, the range is a rectangle (i.e., all four of the bounds are specified). We have already seen that *balanced search trees* are the best data structure for these queries. However, as we can see from the diagrams above, in the range queries used for interval containment problems, the range is always *unbounded* on two sides. This led to a search for a faster algorithm that would somehow benefit from this fact.

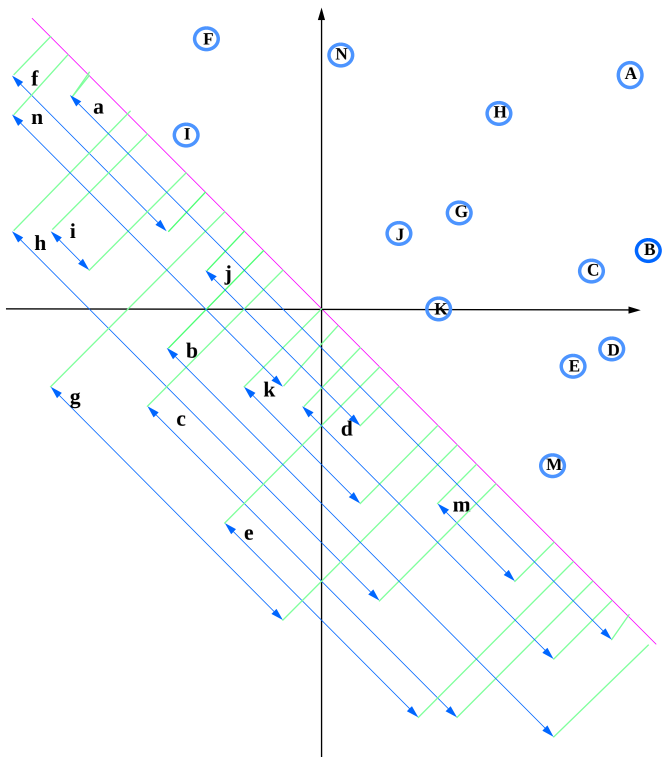


Figure 3: Rotated mapping of intervals on l to points above l.

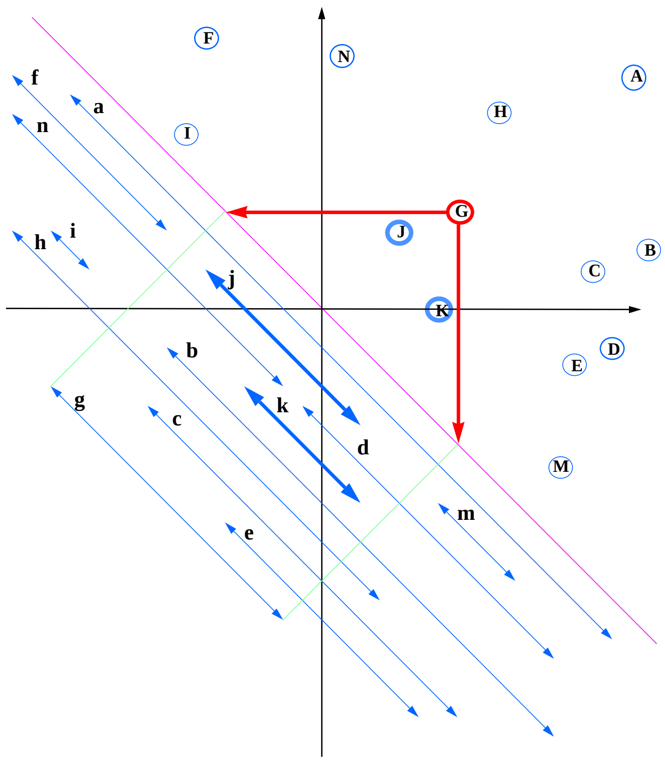


Figure 4: Which intervals are contained in g?

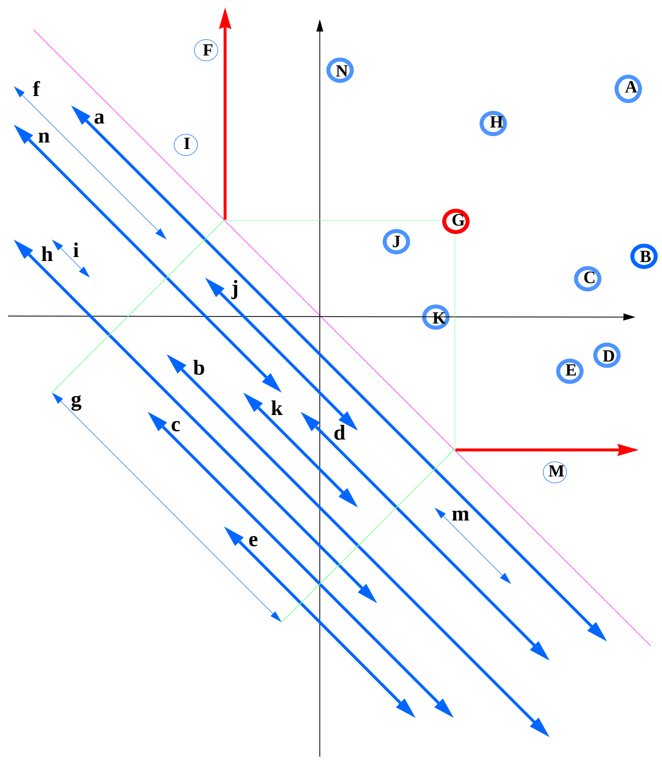


Figure 5: Which intervals intersect g?

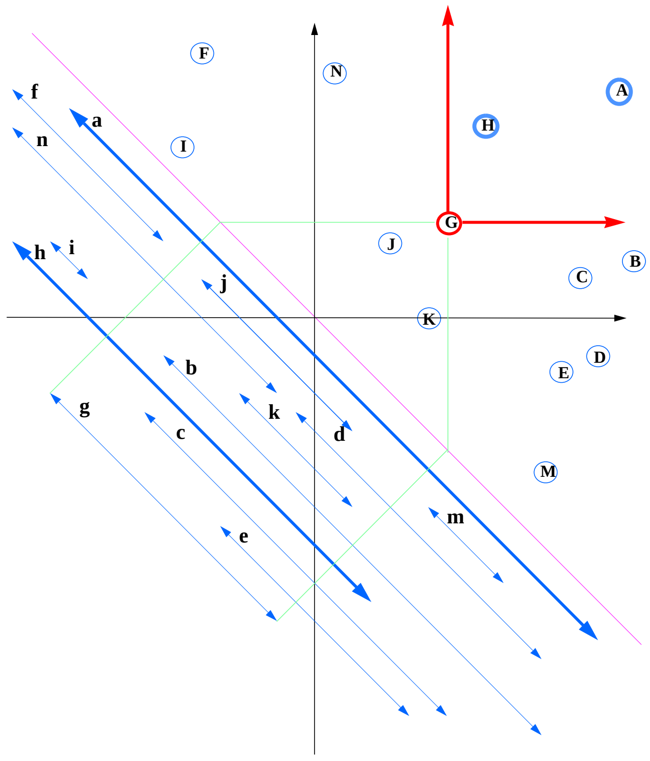


Figure 6: Which intervals contain g?

3 Priority Search Trees: the Background

We can find other examples of range searching on a non-rectangular range. For example, finding all points between two given vertical lines (see Fig. 7 (a)). The best approach for this problem would be to use a *balanced binary search tree*.

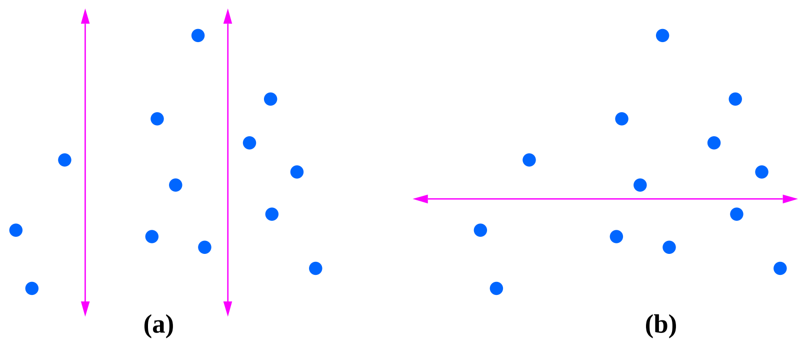


Figure 7: Simpler cases of range search problems

Another example would be to find all points above a given horizontal line (see Fig. 7 (b)). Though we could also use a balanced search tree here, a better approach in this case is a *heap*.

A *priority search tree* is a hybrid of a heap and a balanced search tree. They were discovered recently [1], to be used for range queries where at least one of the sides of the range is unbounded.

The rest of the paper assumes that we are dealing with a priority search tree where *the upper bound on* Y *is missing* (i.e., the range for the queries will only specify a lower bound on Y). It should be easy to modify the algorithms to apply to cases where a different bound is missing.

4 Creating a Priority Search Tree

For a given set of points S, we create a priority search tree as follows:

- If S is empty, we return a NULL pointer and do not continue.
- The point P in S with the the greatest Y-coordinate becomes the root R.
- If $S - P$ is empty, R is also a leaf; we return R and do not continue.
- Let $X(P)$ be a value such that half of points in $S - P$ have X-coordinate lower than $X(P)$, and half have higher.
- Recursively create a priority search tree on the lower half of $S - P$, let its root be the left child of R.
- Recursively create a priority search tree on the upper half of $S - P$, let its root be the right child of R.

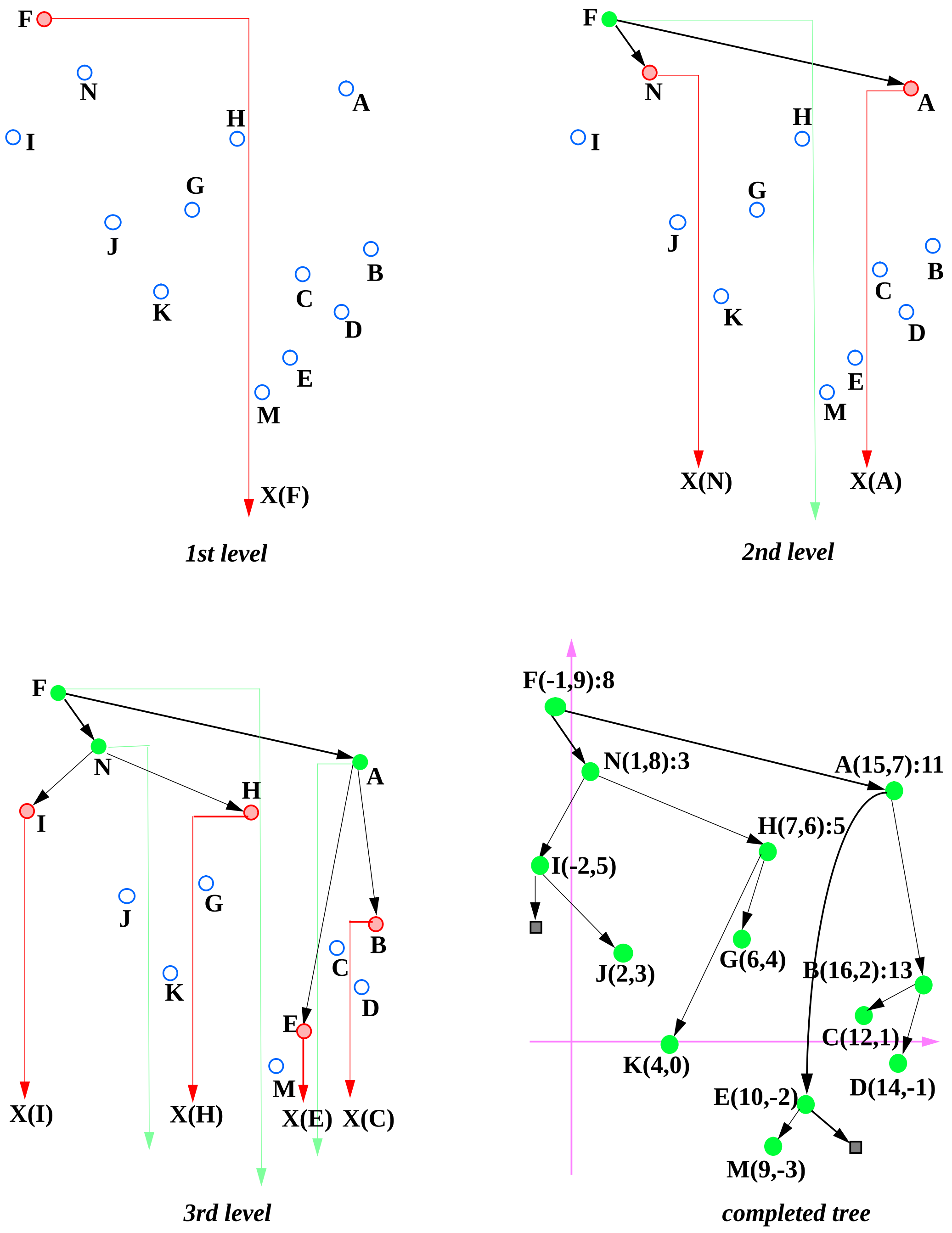


Figure 8: Creating a Priority Search Tree

Fig. 8 illustrates the construction of a priority search tree on our set of 13 points. The points already in the tree are solid; the points being chosen next are shaded; null pointers are shown as pointers to little boxes. In the final picture, each tree node P is labeled by its coordinates, followed by X(P) *if it is different from the X-coordinate*. These labels will be used in the next chapter.

5 Querying a Priority Search Tree

We shall not be concerned with updating the priority search tree dynamically; this topic will be addressed in the next lecture.

A query on a priority search tree is as follows: *given x', x", and y', what are all the points in the tree whose X-coordinate is between x' and x", and whose Y-coordinate is greater than y'*? The following is the algorithm for performing this query:

- If the tree is NULL, we return without finding any points.
- Let R be the root of the tree, x be its X-coordinate, y be its Y-coordinate, and $X(R)$ be the value of the axis separating the X-ranges of R's child subtrees.
- Compare y to y'. If $y < y'$, we return without finding any points (all other nodes in the tree will have an even smaller Y-coordinate).
- If $x' \le x \le x''$, report the root point.
- If $x' < X(R)$, the X-range of the left subtree must overlap with the X-range of the query. Recursively search the left subtree of R.
- If $X(R) < x''$, the X-range of the right subtree must overlap with the X-range of the query. Recursively search the right subtree of R.

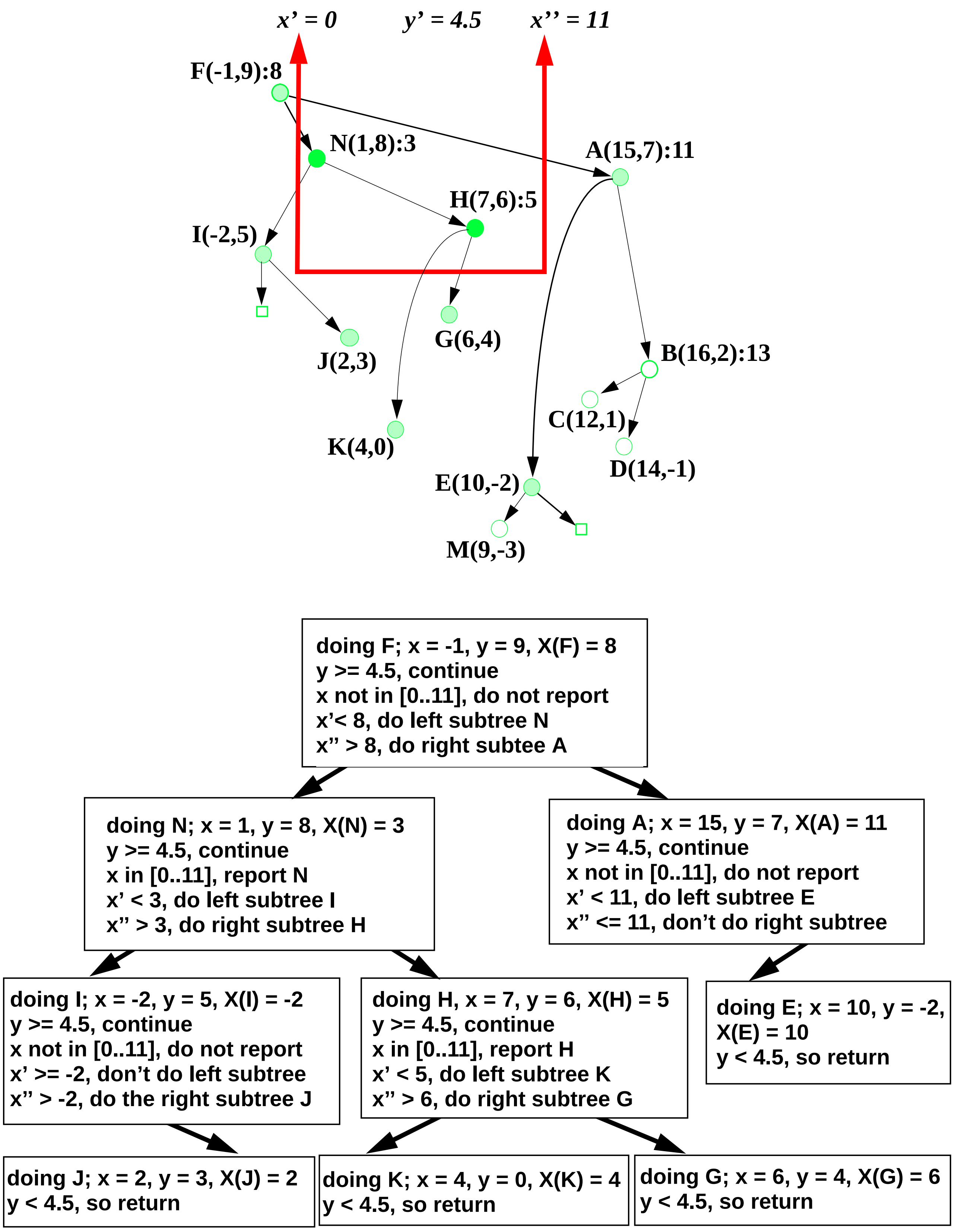
An example of a priority search tree query, with $x' = 0, x'' = 11, y' = 4.5$, is shown in Fig. 9. Nodes that are visited but not reported are shaded; nodes that are visited and reported are solid.

6 Time and Space Analysis

It is clear that a priority search tree on n points takes up *space* $O(n)$, since there is one node for each point. The *height* of the tree is $O(\log n)$, since the nodes are partitioned in half at each level. It remains to show that for a query with an answer of size k, *query time* will be $O(k + \log n)$.

The query time is proportional to the number of nodes visited. Let us categorize all the visited nodes as follows (refer to Fig. 9):

1. A node is visited and reported.
   - Nodes N, H are in this category.
   - The number of reported nodes is k by definition.
2. A node is visited and not reported, but its X-coordinate falls within the $[x', x'']$ range.
   - Nodes J, K, G, E are in this category.
   - This node must have a bad Y-coordinate.



doing F; x = -1, y = 9, X(F) = 8
y >= 4.5, continue
x not in [0..11], do not report
x'< 8, do left subtree N
x" > 8, do right subtee A

doing N; x = 1, y = 8, X(N) = 3
y >= 4.5, continue
x in [0..11], report N
x' < 3, do left subtree I
x" > 3, do right subtree H

doing A; x = 15, y = 7, X(A) = 11
y >= 4.5, continue
x not in [0..11], do not report
x' < 11, do left subtree E
x" <= 11, don't do right subtree

doing I; x = -2, y = 5, X(I) = -2
y >= 4.5, continue
x not in [0..11], do not report
x' >= -2, don't do left subtree
x" > -2, do the right subtree J

doing H, x = 7, y = 6, X(H) = 5
y >= 4.5, continue
x in [0..11], report H
x' < 5, do left subtree K
x" > 6, do right subtree G

doing E; x = 10, y = -2,
X(E) = 10
y < 4.5, so return

doing J; x = 2, y = 3, X(J) = 2
y < 4.5, so return

doing K; x = 4, y = 0, X(K) = 4
y < 4.5, so return

doing G; x = 6, y = 4, X(G) = 6
y < 4.5, so return

Figure 9: Searching a Priority Search Tree

- In this case, its parent had to be visited, and have a good Y-coordinate, falling into one of the other categories.
- Therefore, the total number of these nodes is at most 2 * (all other visited nodes).

3. A node is visited and not reported because its X-coordinate is bad.
   - Nodes F, I, A are in this category.
   - If, for each level of the tree, we list the nodes in order from left to right, this list will be sorted by ascending X-coordinate.
   - Next, we note the following: if two adjacent nodes in the list are both to the left (right) of the $[x', x'']$ range, the leftmost (rightmost) of them cannot possibly be visited by the search algorithm.
   - Therefore, there will be at most 2 nodes per level which fall outside the $[x', x'']$ range (one on the right and one on the left) which can be visited.
   - Since there are $O(\log n)$ levels in the tree, there are at most $O(\log n)$ nodes in this category.

Summing up the total number of visited nodes, we obtain a query time of $O(k + \log n)$, as is desired.

References

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