Bounding sphere methods / SS-Tree

¿Por qué utilizar bounding boxes en forma de hyperspheres?

- A medida que la dimensionalidad del espacio aumenta, el número de **hyperrectangular data regions** intersectados por un **hypersphere query** aumenta también.

SS-Tree: R*-Tree which makes use of minimum bounding hyperspheres

Sphere-Tree: R-Tree …

SS-Tree

> The SS-tree insertion algorithm combines aspects of both the R-tree [791] and the R*-tree [152]. In particular, initially, an object o is inserted into the node whose centroid is the closest to the centroid of o. This step is similar to the first step of the R-tree insertion algorithm and is motivated by the desire to minimize the coverage of the existing nodes (but see Exercise 2).

> Overflow in an existing SS-tree node is resolved in a similar manner to the way it is handled in an R*-tree in that forced reinsertion is used. In particular, if a node A is full, then a fraction of the elements in A are reinserted (e.g., 30% [1991]). The elements to reinsert are the ones farthest from the centroid. These elements are removed from A, the centroid and radius of A are adjusted, and the elements are inserted again into the SS-tree starting from the root. If A overflows again during the reinsertion, A is split rather than attempting reinsertion again. Note that this is different from the reinsertion convention of the R*-tree, where reinsertion is invoked on only one node at any given level (i.e., any node at the same level of A that overflows during the reinsertion process is split).

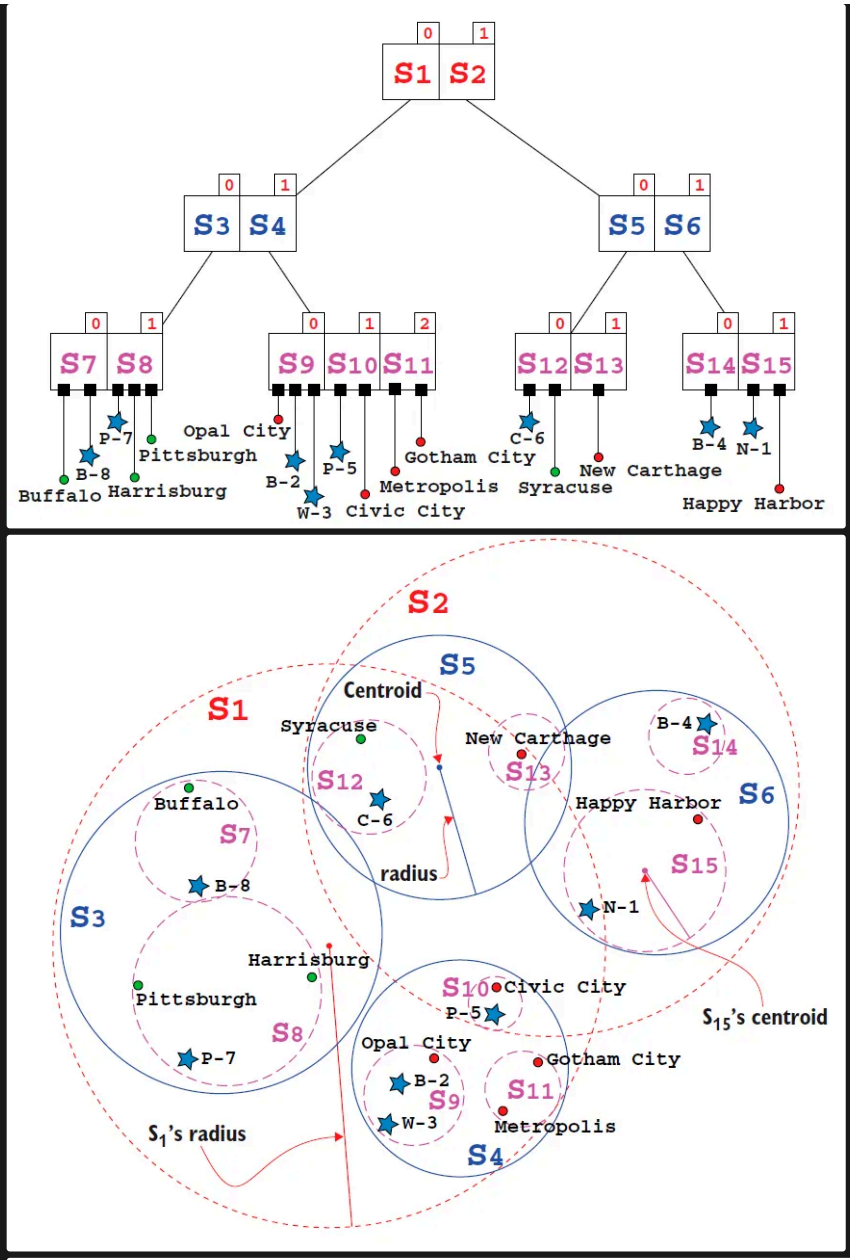


Figure 10.9 Representation of a possible SS-tree covering the same dataset of figures 10.4 and 10.5, with parameters `m==1` and `M==3`. As you can see, the tree structure is similar to R-trees'. For the sake of avoiding clutter, only a few spheres' centroids and radii are shown. For the tree, we use the compact representation (as shown in figures 10.5 and 10.8).

Search

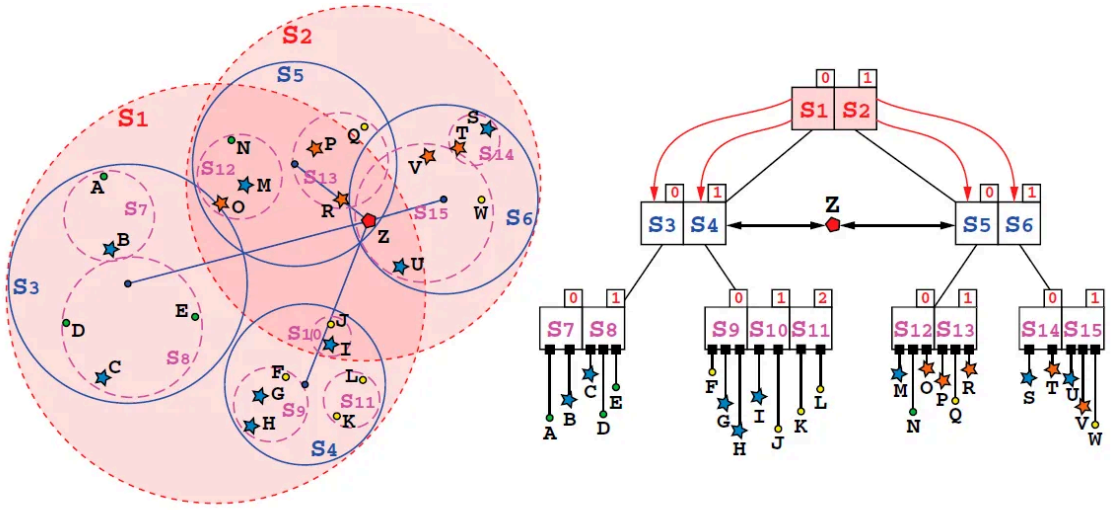


Figure 10.10 Search on a SS-tree: the first few steps of searching for point Z. The SS-tree shown is derived from the one in figure 10.9, with a few minor changes; the name of the entries have been removed here and letters from A to W are used to reduce clutter. (Top) The first step of the search is comparing Z to the spheres in the tree's root: for each of them, computes the distance between Z and its centroid, and checks if it's smaller than the sphere's radius. (Bottom) Since both S_1 and S_2 intersect Z, we need to traverse both branches and check spheres S_3 to S_6 for intersection with Z.

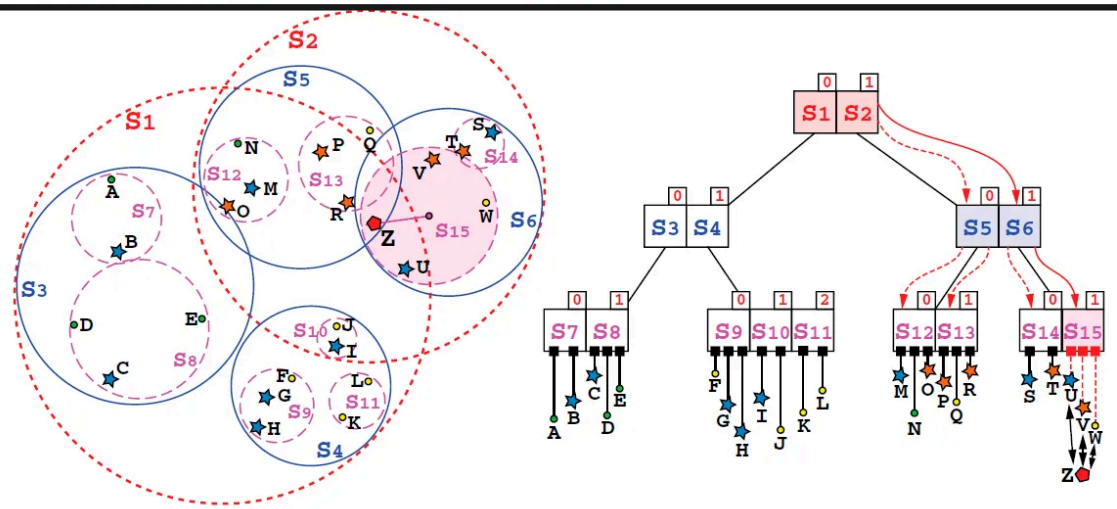


Figure 10.11 Search on an SS-tree: Continuing from figure 10.10, we traverse the tree up to leaves. At each step, the spheres highlighted are the ones whose children are being currently traversed (in other words, at each step the union of the highlighted spheres is the smallest area where the searched point could lie).

```
function search(node, target) if node.leaf then for point in node.points do if
point == target then return node else for childNode in node.children do if
childNode.intersectsPoint(target) then result = search(childNode, target) if
result != null then return result return null
```

Listing 10.3 The `search` method

Method `search` returns the tree leaf that contains a target point if the point is stored in the tree; it returns `null` otherwise. We explicitly pass the root of the (sub)tree we want to search so we can reuse this function for sub-trees.

Checks if `node` is a leaf or an internal node

If `node` is a leaf, goes through all the points held, and checks whether any match `target`

```
function search(node, target)
  if node.leaf then
    for point in node.points do
      if point == target then
        return node
  else
    for childNode in node.children do
      if childNode.intersectsPoint(target) then
```

If a match is found, returns current leaf

Checks if `childNode` could contain `target`; that is, if `target` is within `childNode`'s bounding envelope. See listing 10.4 for an implementation.

Otherwise, if we are traversing an internal node, goes through all its children and checks which ones could contain `target`. In other words, for each children `childNode`, we check the distance between its centroid and the target point, and if this is smaller than the bounding envelope's radius of `childNode`, we recursively traverse `childNode`.

If no child of current node could contain the `target`, or if we are at a leaf and no point matches `target`, then we end up at this line and just return `null` as the result of an unsuccessful search.

If that's the case, performs a recursive search on `childNode`'s branch, and if the result is an actual node (and not `null`), we have found what we were looking for and we can return.

```
      result ← search(childNode, target)
      if result != null then
        return result
  return null
```

Insert

- Buscar la hoja adecuada para insertar el punto:
  - **En cada nivel, seleccionar el subárbol cuyo centroide este mas cerca al punto insertado.**
  - Note que esta búsqueda es distinta a la anterior descrita (simpler version of the search method).

Listing 10.5 The `searchParentLeaf` method

This search method returns the closest tree leaf to a target point.

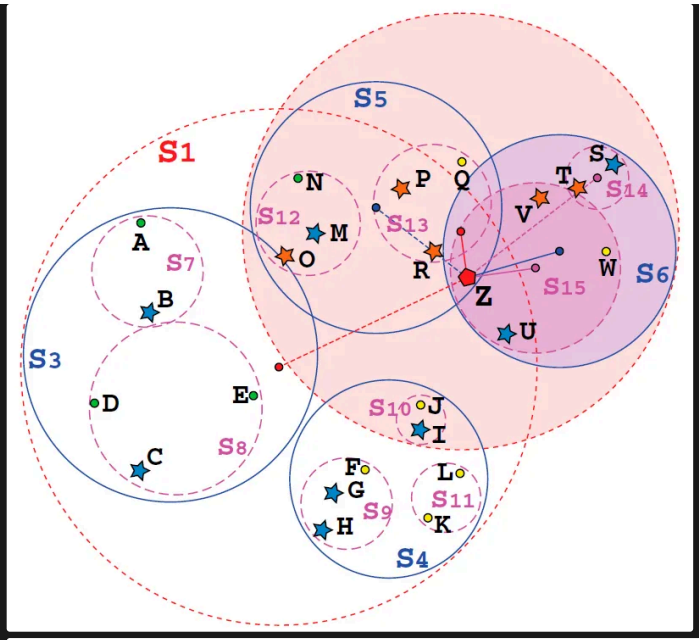
Checks if `node` is a leaf. If it is, we can return it.

Otherwise, we are traversing an internal node and need to find which branch to go next. We run the heuristic `findClosestChild` to decide (see listing 10.8 for an implementation).

```
function searchParentLeaf(node, target)
  if node.leaf then
    return node
  else
    child ← node.findClosestChild(target)
  return searchParentLeaf(child, target)
```

Recursively traverses the chosen branch and returns the result

- Ejemplo:
  - $\text{insert}(\text{Z})$
  - Los nodos visitados son: S2 → S6 → S15



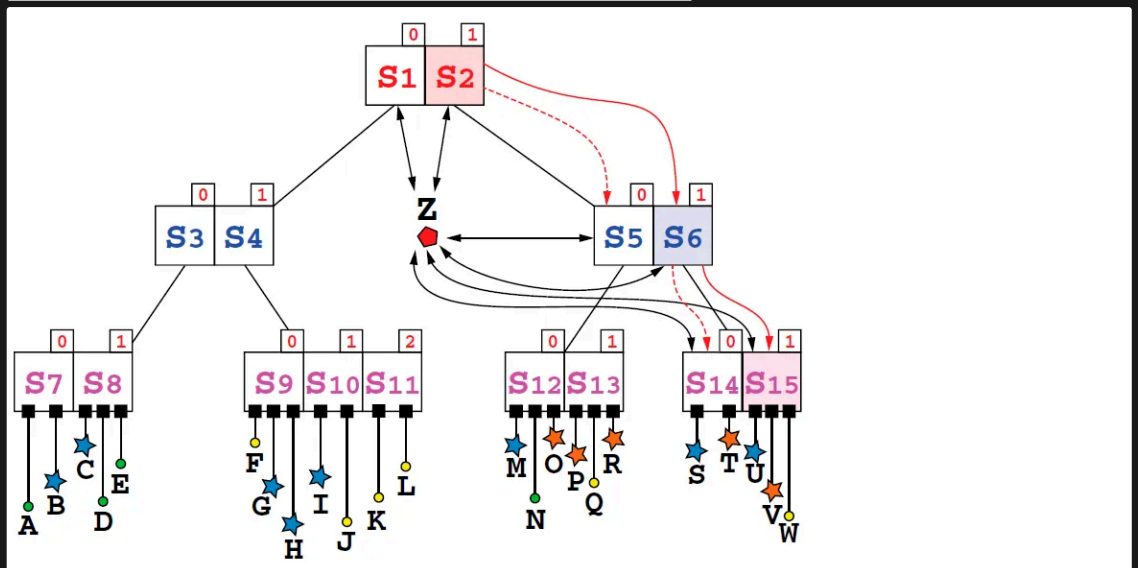


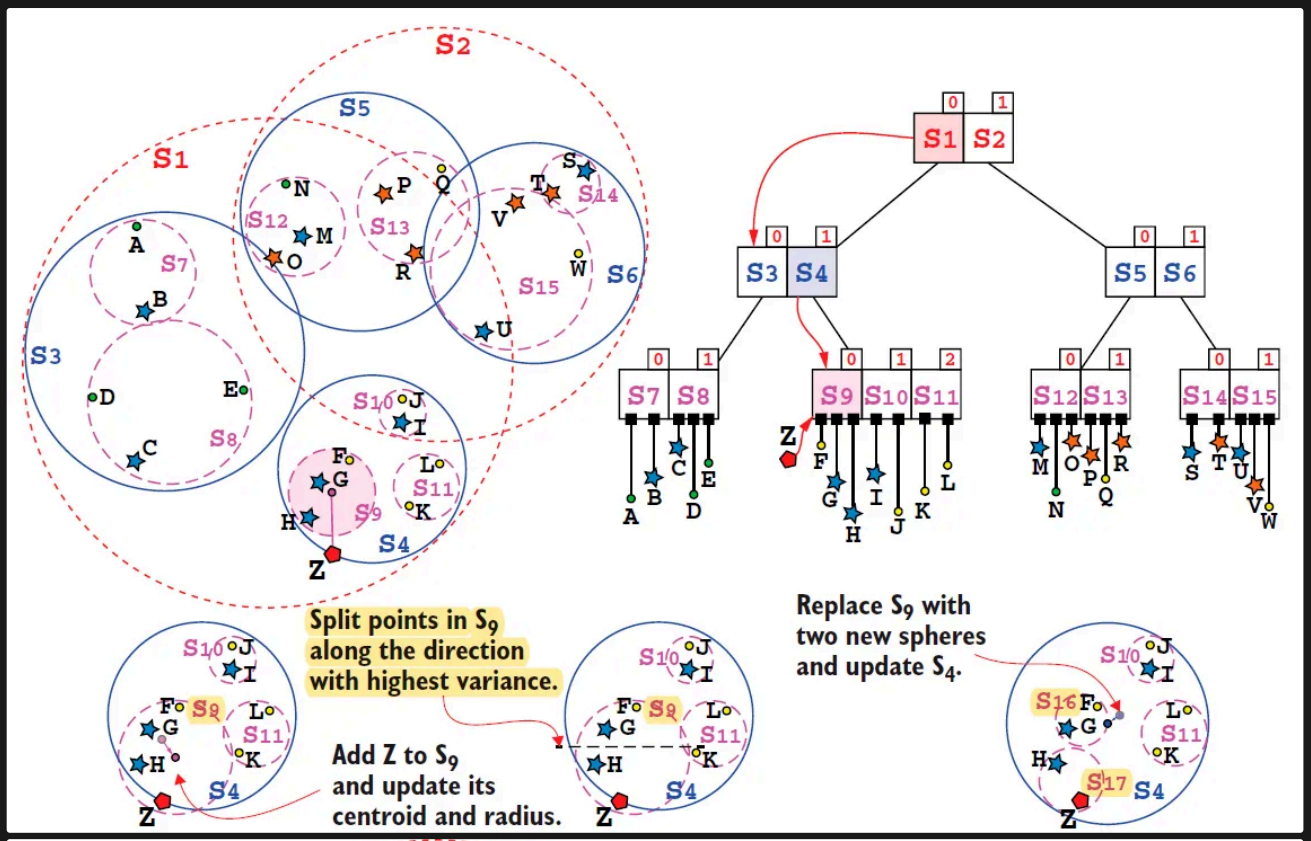
Figure 10.13 An example of the tree traversing for method `searchParentLeaf`. In contrast with figures 10.10 and 10.11, here the steps are condensed into a single diagram for the sake of space. The fact that only one path is traversed allows this compact representation. Notice how at each step the distance between `z` and the centroids in the current node are computed (in this figure, we used for distances the same level-based color code as for spheres, and the segments drawn for distances have one end in the center of the sphere they are computed from, so it's easy to spot the distance to the root node, and to spheres at level 1, and so on), and only the branch with the shortest distance (drawn as a thicker, solid line) is chosen. The spheres' branches traversed are highlighted on both representations.

Inserting a point in a full leaf

Ejemplo:

- El punto Z debe ser insertado en S_9.
- S_9 esta lleno. Dividimos S_9 en S_{16} y S_{17} (en la otra sección hay detalles sobre como se hace el split).
- Actualizamos el centroide y radio de S_4.

- Backtrack (S_4 esta lleno) ...



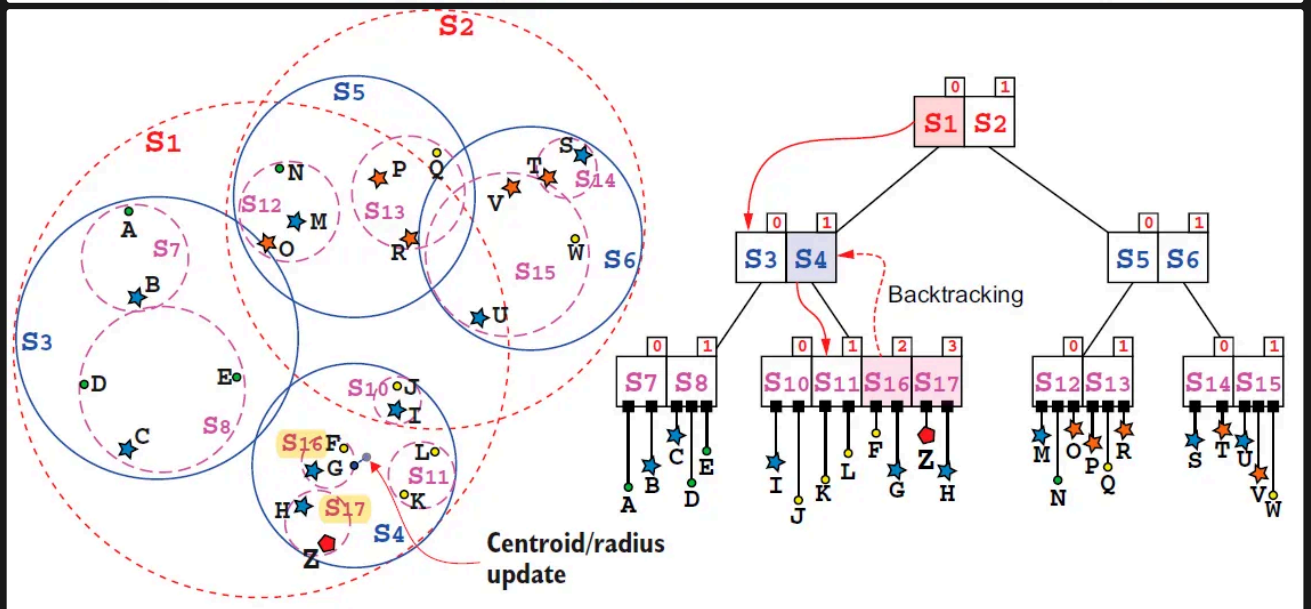
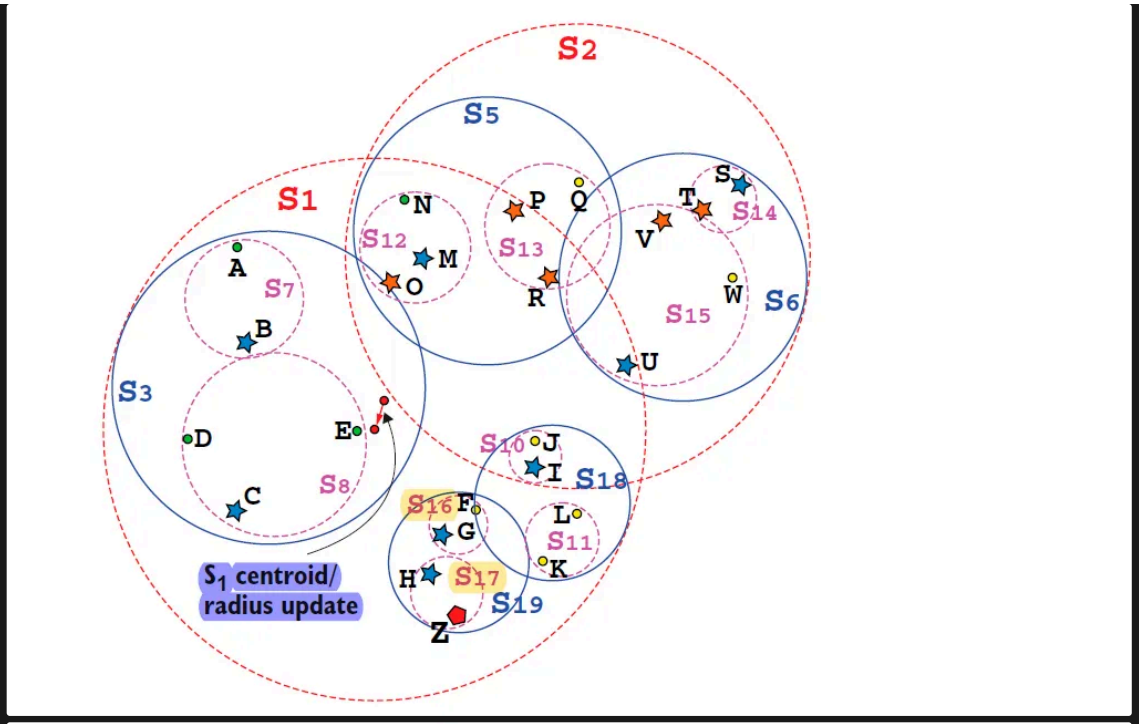


Figure 10.14 **Inserting a point in a full leaf.** (Top) The search step to find the right leaf. (Center) A closeup of the area involved. S_9 needs to be updated, recomputing its centroid and radius. **Then we can find the direction along which points have the highest variance (y, in the example)** and split the points so that the variance of the two new point sets is minimal. Finally, we remove S_9 from its parent S_4 and add two new leaves containing the two point sets resulting from the split. (Bottom) The final result is that we now need to update S_4's centroid and radius and backtrack.



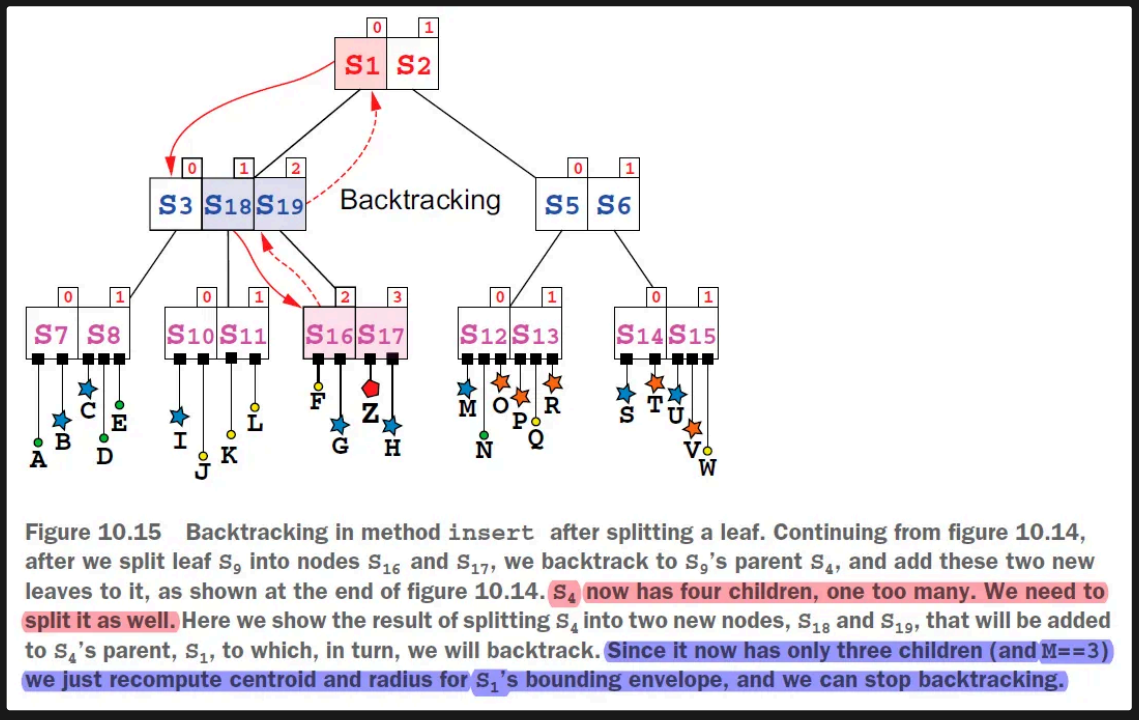


Figure 10.15 Backtracking in method `insert` after splitting a leaf. Continuing from figure 10.14, after we split leaf S_9 into nodes S_{16} and S_{17}, we backtrack to S_9's parent S_4, and add these two new leaves to it, as shown at the end of figure 10.14. S_4 now has four children, one too many. We need to split it as well. Here we show the result of splitting S_4 into two new nodes, S_{18} and S_{19}, that will be added to S_4's parent, S_1, to which, in turn, we will backtrack. Since it now has only three children (and `M==3`) we just recompute centroid and radius for S_1's bounding envelope, and we can stop backtracking.

Split

Detalles sobre el split

¡Importante! en caso de overflow

- Dividir la hoja en la dirección de **máxima varianza**.
- En el ejemplo, la dirección óptima es y

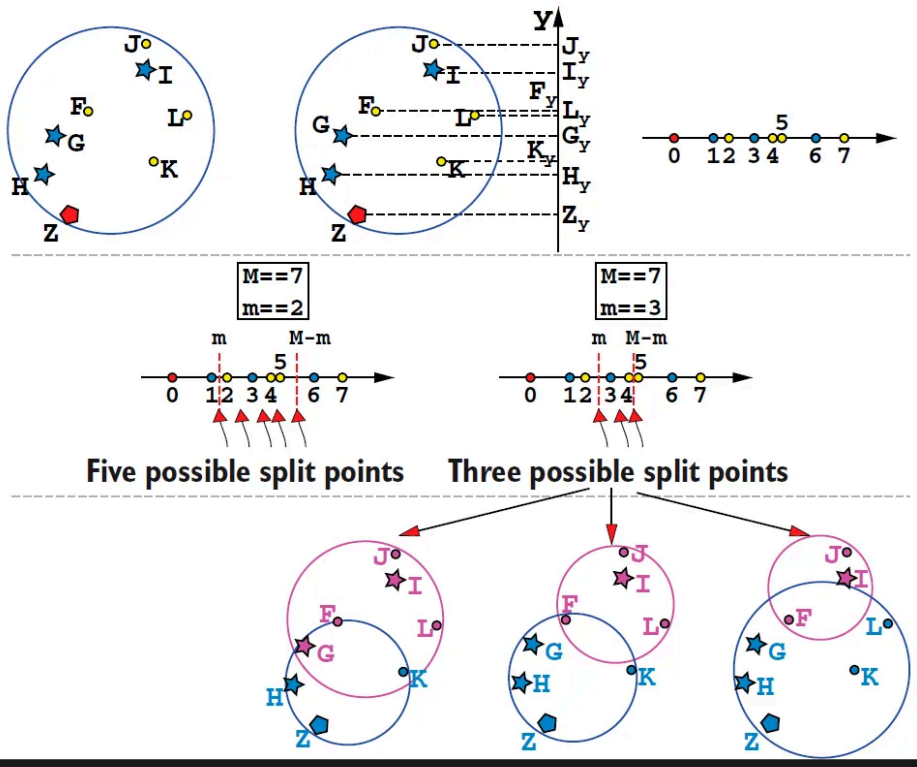


Figure 10.18 Splitting a set of along the direction of maximum (Top) The bounding envelope and to split; the direction of maximun is along the y axis (center), so v all points on this axis. On the ri rotate the axis for convenience a the point labels with indices. (M Given that there are 8 points, we M must be equal to 7. Then m c value ≤3. Since the algorithm c single split index, partitioning the its two sides, and each partition have at least m points, dependin actual value of m, we can have a number of choices for the split i (Bottom) We show the three po resulting splits for the case whe the split index can be 3, 4, or 5 choose the option for which the s variances for the two sets is mi

¡Error!

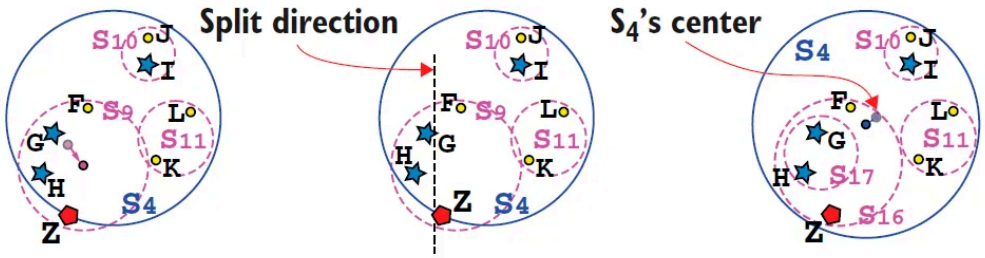


Figure 10.16 Splitting a leaf along a non-optimal direction. In this case, the x axis is the direction with minimal variance. Comparing the final result to figure 10.14, although S4's shape doesn't change significantly, S_{16} has more than doubled in size and completely overlaps S_{17}; this means that any search targeted within S_{17} will also have to traverse S_{16}.

Delete

SS-Tree

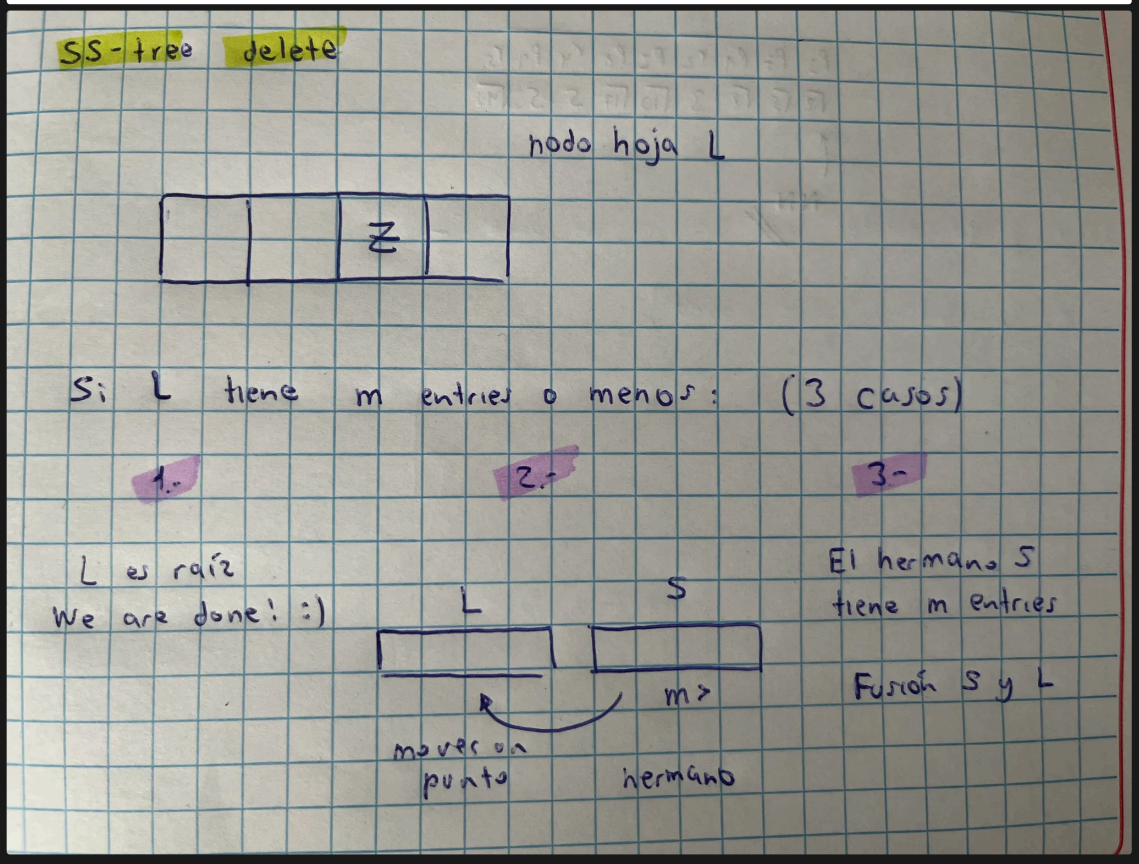
Borrado

Buscamos el nodo a eliminar, Z, en el árbol. Dentro del nodo hoja L, ejecutamos:

Si la hoja contiene más de m puntos, simplemente eliminamos Z de L y actualizamos su sobre delimitador.

De lo contrario:

1. Si L es la raíz, estamos bien y no tenemos que hacer nada.
2. Si L tiene al menos un hermano S con más de m puntos, podemos mover un punto de S a L.
3. Si ningún hermano de L puede prestarle un punto, entonces tendremos que fusionarnos L con uno de sus hermanos.



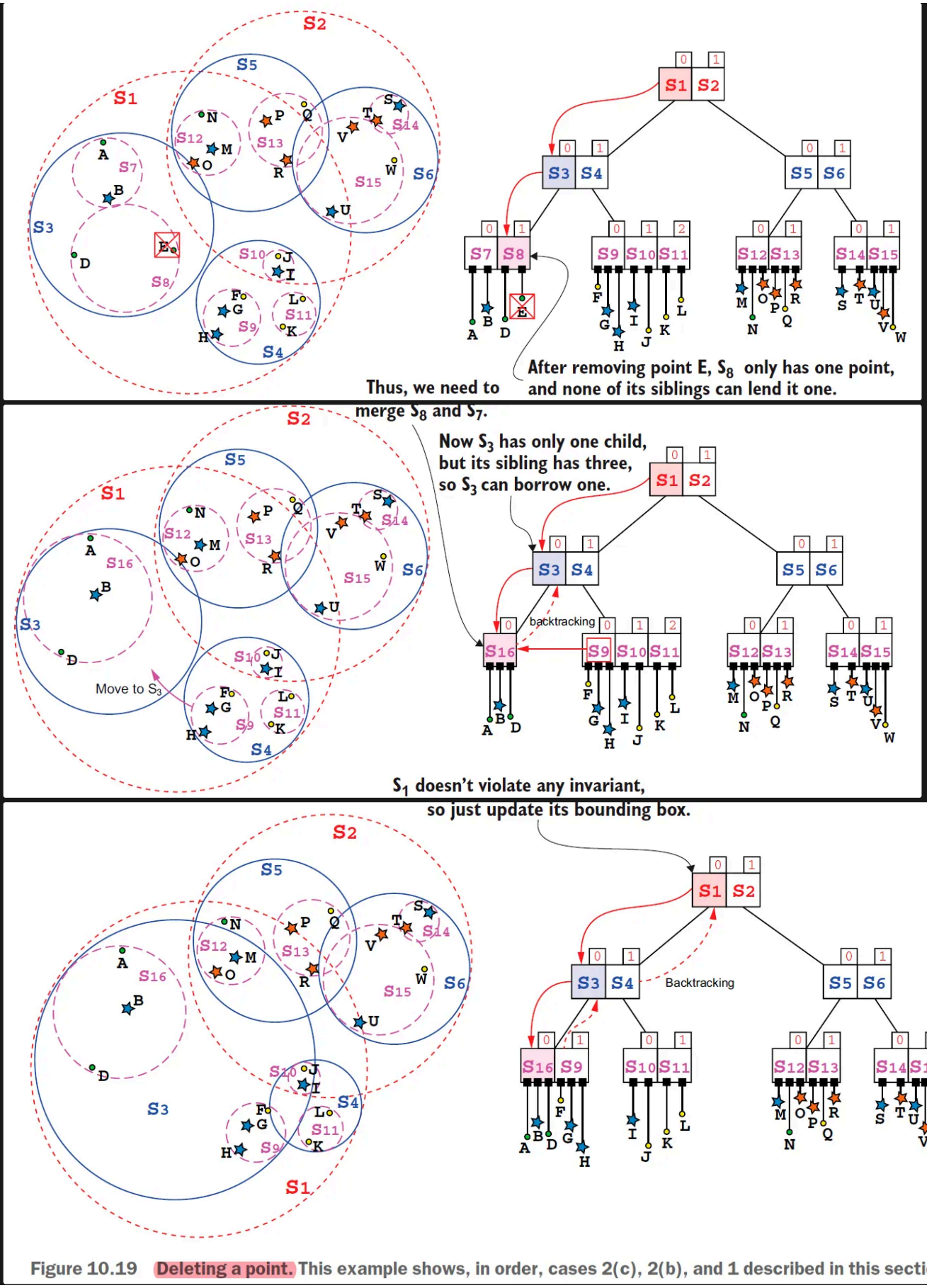


Figure 10.19 **Deleting a point.** This example shows, in order, cases 2(c), 2(b), and 1 described in this section.