Name	x	y
Chicago	35	42
Mobile	52	10
Toronto	62	77
Buffalo	82	65
Denver	5	45
Omaha	27	35
Atlanta	85	15
Miami	90	5

Figure 2.1: Sample list of cities with their x and y coordinate values.

2.3 Point Quadtrees

The point quadtree, invented by Finkel and Bentley [229], is a marriage of the fixed-grid method and the binary search tree that results in a tree-like directory with non-uniform sized cells containing one element apiece. The k-d tree, invented by Bentley [52] (and discussed in Section 2.4), is an improvement over the point quadtree because it reduces the branching factor at each node and the storage requirements. Other data structures, which are primarily of theoretical rather than practical interest, have been described by Lueker [439], Lee and Wong [420], Willard [799], Bentley [55], and Bentley and Maurer [61].

The point quadtree is implemented as a multidimensional generalization of a binary search tree. In two dimensions each data point is represented as a node in a quadtree in the form of a record of type *node* containing seven fields. The first four fields contain pointers to the node's four sons corresponding to the directions (i.e., quadrants) `NW`, `NE`, `SW`, and `SE`. If P is a pointer to a node and I is a quadrant, then these fields are referenced as `SON`(P,I). We can determine the specific quadrant in which a node, say P, lies relative to its father by use of the function `SONTYPE`(P), which has a value of I if `SON(FATHER`(P)`,`I)`$= P$. `XCOORD` and `YCOORD` contain the values of the $x$ and $y$ coordinates, respectively, of the data point. The `NAME` field contains descriptive information about the node (e.g., city name, etc.). For example, see Figure 2.4 which is the point quadtree corresponding to the data of Figure 2.1.

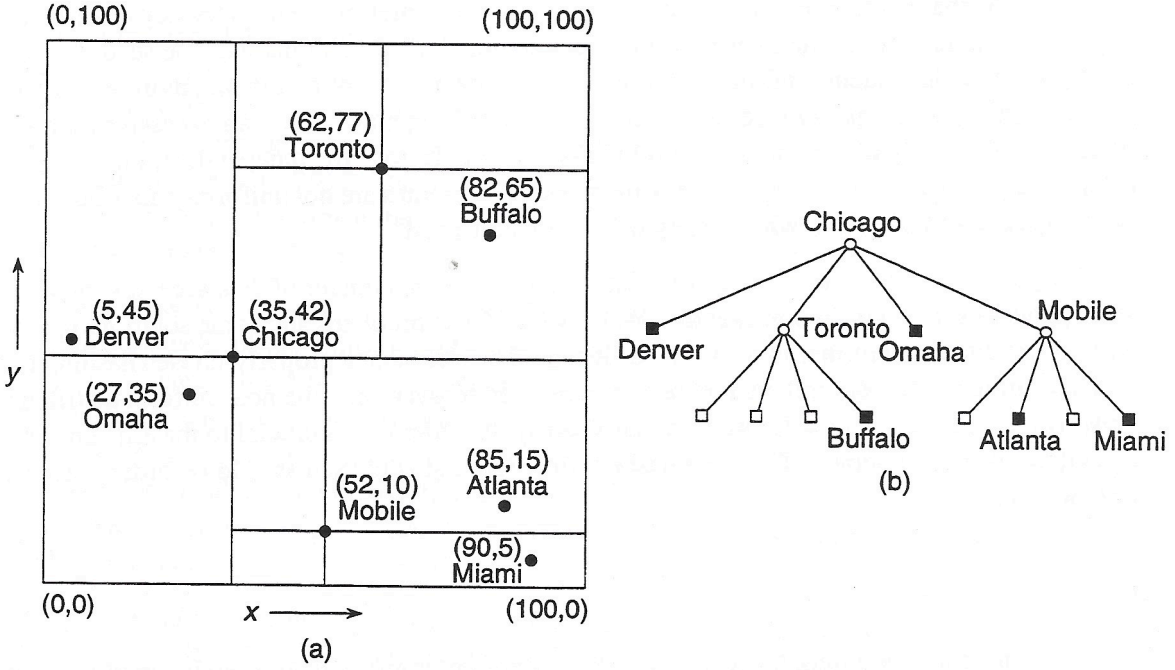


Figure 2.4: A point quadtree and the records it represents corresponding to Figure 2.1: (a) the resulting partition of space, and (b) the tree representation.

We assume that each data point is unique. Should an application permit collisions (i.e., several data points with the same coordinate values), then our data structure could contain an additional field in which a pointer to an overflow collision list would be stored. It could be argued that devoting an extra field to a rare event such as a collision wastes storage. However, without it, we would require a considerably more complicated node insertion procedure.

2.3.1 Insertion

Records are inserted into point quadtrees in a manner similar to that done for binary search trees. In essence, we search for the desired record based on its x and y coordinate values. At each node a comparison is made by use of procedure `PT_COMPARE` and the appropriate subtree is chosen for the next test. Upon reaching the bottom of the tree (i.e., when a `NIL` pointer is encountered), we find the location where the record is to be inserted. For example, the tree in Figure 2.4 was built for the sequence `Chicago`, `Mobile`, `Toronto`, `Buffalo`, `Denver`, `Omaha`, `Atlanta`, and `Miami`. Figure 2.5 shows how the tree was constructed in an incremental fashion for `Chicago`, `Mobile`, `Toronto`, and `Buffalo` by giving the appropriate block decompositions.

To cope with data points that lie directly on one of the quadrant lines emanating from a data point, say P, we adopt the same conventions that were used for the grid method — that is, the lower and left boundaries of each block are closed while the upper and right boundaries of each block are open. For example, in Figure 2.4, insertion of `Memphis` with coordinate values `(35,20)` would lead to its placement somewhere in quadrant `SE` of the tree rooted at `Chicago` (i.e., at `(35,42)`).

```
quadrant procedure PT_COMPARE(P,R);
/* Return the quadrant of the point quadtree rooted at node R in which node P belongs. */
begin
  value pointer node P,R;
  return(if XCOORD(P)<XCOORD(R) then
           if YCOORD(P)<YCOORD(R) then 'SW'
           else 'NW'
         else if YCOORD(P)<YCOORD(R) then 'SE'
         else 'NE');
end;

procedure PT_INSERT(P,R);
/* Attempt to insert node P in the point quadtree rooted at node R. */
begin
  value pointer node P;
  reference pointer node R;
  pointer node F,T;
  quadrant Q;
  if NULL(R) then R←P /* The tree at R is initially empty */
  else
    begin
      T←R;
      while NOT(NULL(T)) AND NOT(EQUAL_COORD(P,T)) do
        begin
```

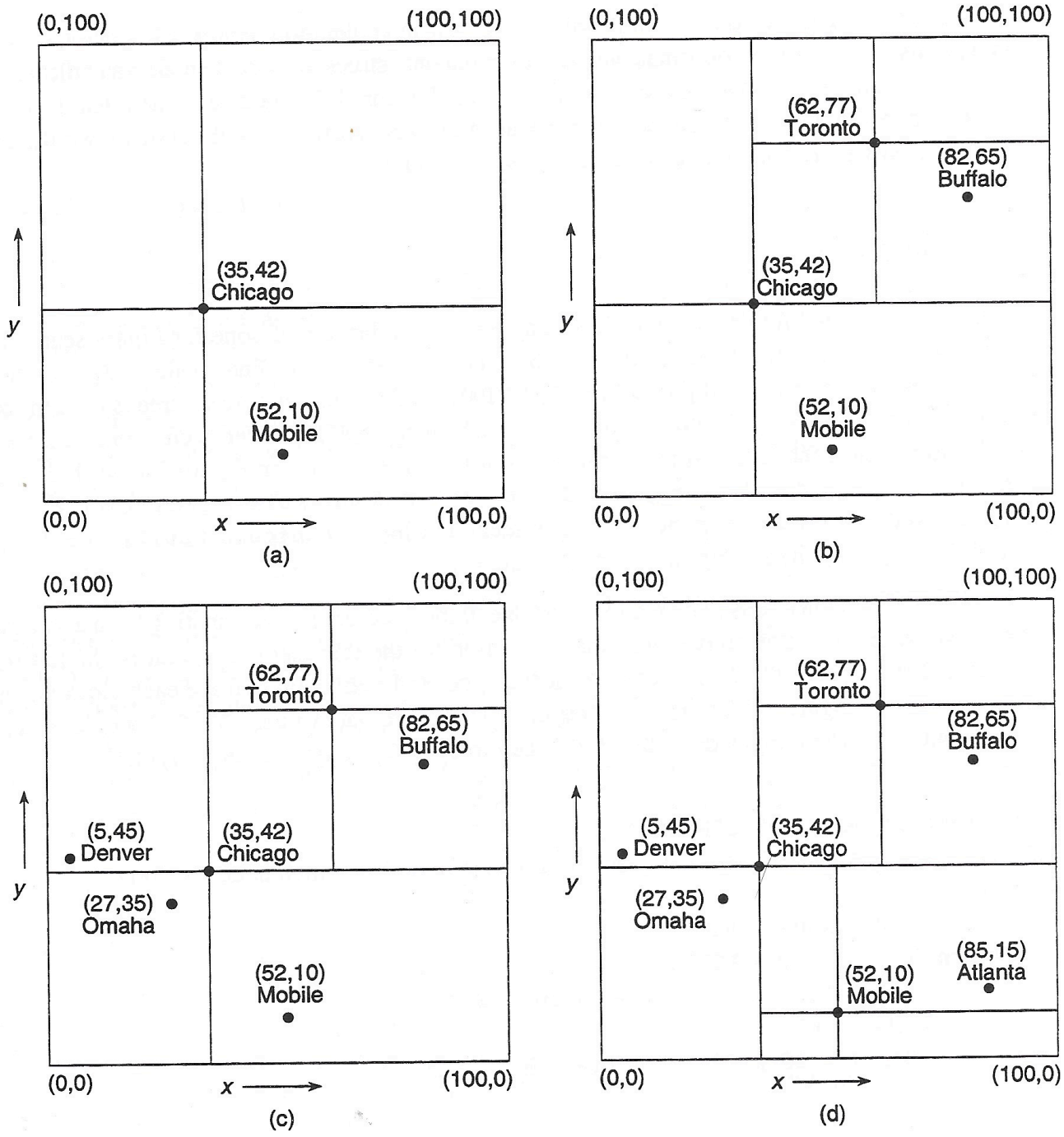


Figure 2.5: Sequence of partial block decompositions showing how a point quadtree is built when adding (a) Chicago, (b) Mobile, (c) Toronto, and (d) Buffalo corresponding to Figure 2.1.

```
            F←T; /* Remember the father */
            Q←PT_COMPARE(P,T);
            T←SON(T,Q);
          end;
        if NULL(T) then SON(F,Q)←P; /* P is not already in the tree */
      end;
end;
```

The amount of work expended in building a point quadtree is equal to the total path length (TPL)

[383] of the tree as it reflects the cost of searching for all of the elements. Finkel and Bentley [229] have shown empirically that the TPL of a point quadtree under random insertion is roughly proportional to $N \cdot \log_4 N$, which yields an average cost of inserting, as well as searching for (i.e., a point query), a node of $O(\log_4 N)$. The extreme case is much worse (see Exercise 8) and is a function of the shape of the resulting point quadtree. This is dependent on the order in which nodes are inserted into it. The worst case arises when each successive node is the son of the currently deepest node in the tree. Consequently, there has been some interest in reducing the TPL. Two techniques for achieving this reduction are described below.

Finkel and Bentley [229] propose one approach that assumes that all the nodes are known *a priori*. They define an *optimized point quadtree* so that given a node A, no subtree of A accounts for more than one half of the nodes in the tree rooted at A. Building an optimized point quadtree from a file requires that the records in the file be sorted primarily by one key and secondarily by the other key. The root of the tree is set to the median value of the sorted file and the remaining records are regrouped into four subcollections that will form the four subtrees of A. The process is recursively applied to the four subtrees.

The reason that this technique works is that all records preceding A in the sorted list will lie in the `NW` and `SW` quadrants (assuming that the x coordinate value serves as the primary key) and all records following A will lie in the `NE` and `SE` quadrants. Thus, the requirement that no subtree can contain more than one half of the total number of nodes is fulfilled. Of course, this construction method still does not guarantee that the resulting tree will be complete[3] (see Exercise 2).

The optimized point quadtree requires that all of the data points are known *a priori*. Overmars and van Leeuwen [550] discuss an alternative approach which is a dynamic formulation of the above method — that is, the optimized point quadtree is built as the data points are inserted into it. The algorithm is similar to the one used to construct Finkel and Bentley's optimized point quadtree, except that every time the tree fails to meet a predefined balance criterion, the tree is partially rebalanced.

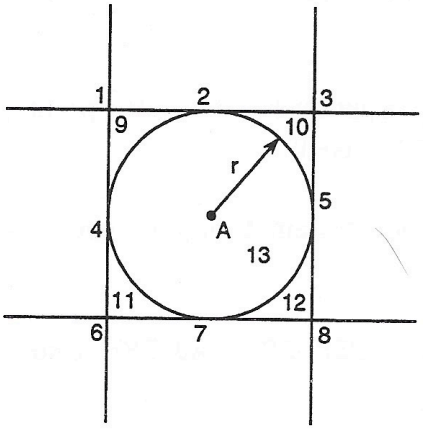
[3] A t-ary tree containing N nodes is *complete* if we can map it onto a one-dimensional array so that the first element consists of the root, the next t elements are the roots of its t subtrees ordered from left to right, the next t^2 elements are the roots of all of the subtrees of the previous t elements again ordered from left to right, etc. This process stops once N is exhausted. Thus, each level of the tree, with the exception of the deepest level, contains a maximum number of nodes. The deepest level is partially full but has no empty positions when using this array mapping. For more details, see Knuth [383, pp. 400-401].

2.3.3 Search

The point quadtree is suited for applications that involve proximity search. A typical query is one that requests the determination of all nodes within a specified distance of a given data point — e.g., all cities within 50 miles of Washington, D.C. The efficiency of the point quadtree data structure lies in its role as a pruning device on the amount of search that is required. Thus, many records will not need to be examined.

For example, suppose that in the hypothetical database of Figure 2.1 we wish to find all cities within eight units of a data point with coordinate values `(83,10)`. In such a case, there is no need to search the `NW`, `NE`, and `SW` quadrants of the root (i.e., `Chicago` located at `(35,42)`). Thus, we can restrict our search to the `SE` quadrant of the tree rooted at `Chicago`. Similarly, there is no need to search the `NW` and `SW` quadrants of the tree rooted at `Mobile` (i.e., located at (52,10)).

As a further illustration of the amount of pruning of the search space that is achievable by use of the point quadtree, we make use of Figure 2.17. In particular, given the problem of finding all nodes within radius `r` of point `A`, use of the figure indicates which quadrants need not be examined when the root of the search space, say R, is in one of the numbered regions. For example, if R is in region 9, then all but its `NW` quadrants must be searched. If R is in region 7, then the search can be restricted to the `NW` and `NE` quadrants of R.



Problem: Find all nodes within radius r of point A
Solution: If the root is in region I (I=1...13), then continue to search in the quadrant specified by I

1. SE	6. NE	11. All but SW
2. SE, SW	7. NE, NW	12. All but SE
3. SW	8. NW	13. All
4. SE, NE	9. All but NW	
5. SW, NW	10. All but NE	

Figure 2.17: Relationship between a circular search space and the regions in which a root of a point quadtree may reside.

Similar techniques can be used to search for data points in any connected figure. For example, Finkel and Bentley [229] give algorithms for searching within a rectangular window of arbitrary size. To handle more complex search regions such as halfspaces and convex polygons, Willard [799] defines a *polygon tree* where the x-y plane is subdivided by J lines that need not be orthogonal, although there are other restrictions on these lines (see Exercise 6). When $J = 2$, the result is a point quadtree with non-orthogonal axes.

Thus, we see that quadtrees can be used to handle all three types of queries specified by Knuth [384].

Range and Boolean queries are described immediately above while simple queries (e.g., what city is located at a given pair of coordinate values) are a byproduct of the point quadtree insertion process described earlier.

The cost of search in a point quadtree has been studied by Bentley, Stanat, and Williams [67] (see Exercise 2) and Lee and Wong [420]. In particular, Lee and Wong show that in the worst case, range searching in a complete two-dimensional point quadtree takes $O(2 \cdot N^{1/2})$ time. This result can be extended to k dimensions to take $O(k \cdot N^{1-1/k})$ time, and is derived in a manner analogous to that for a k-d tree as discussed in Section 2.4. Note that complete point quadtrees are not always achievable as seen in Exercise 2 in Section 2.3.1. Partial range queries can be handled in the same way as range searching. Lee and Wong [420] show that when ranges for s out of k keys are specified, then the algorithm has a worst case running time of $O(s \cdot N^{1-1/k})$.

where x is in $[0,1]$. Compare the amount of overwork for an inverted list (i.e., inverted file) and a perfect point quadtree.

4. Prove that the worst case running time for a partial range query such that ranges for s out of k keys are specified is $O(s \cdot N^{1-1/k})$.

5. Perform an average case analysis for a region query in a point quadtree.

6. What are the restrictions on the choice of subdivision lines in Willard's polygon tree [799]?

2.4 K-d Trees

Point quadtrees have several deficiencies. First, at each node a test (i.e., a `PT_COMPARE` operation) requires testing all k keys for a k-dimensional quadtree. Second, each leaf node is rather costly in terms of the amount of space that is required due to a multitude of `NIL` links) [5]. Third, the node size gets rather large for a k-dimensional tree, since $k + 2^k + 1$ words are required for each node.

[5] Of course, these `NIL` pointers can be avoided in an efficient implementation (e.g., see Section 2.1.2 of [642]).