Data Structures

Xiaoqiang Wang
Florida State University

Changed from Dr. Erlebacher's Lecture

Reference

- The Algorithm Design Manual
  by Steven S. Skiena

- Handbook of Data Structures and Applications
  Dinesh P. Mehta and Sartaj Sahni
  CRC Press (1392 pages)

Outline

- Data and Data Structure
- Array
- Linked List
- Ex: Stacks, Queues, Sets
- Hash Tables
- Trees: Quadtree, Octree
- Other data structures

Data and Data Structure

- In computer science, a data structure is a particular way of *storing* and *organizing* data in a computer so that it can be used efficiently.
  - Data
  - Data acess, operations, manipulation
  - Hidden for general users

Data

- List of numbers
- Collection of objects
- Airplane scheduling
- Database transactions
- Database data
- 2D and 3D data
- Collection of points, rectangles, polyhedra
- Nested adaptive meshes
- Wavelet decompositions
- Sparse matrices

Container: Data is stored in containers

- Stack
- Queue
- Table
- Dictionary
- Priority Queues
- Sets
- Trees
- Graphs
- ….

Data access, operations, manipulation

- Data insertion, deletion
- Data search
- Maximum, minimum of data
- Neighbor search
- Distance between data elements
- Approximate distances

Data Structure examples

- Arrays
- Linked lists
- Doubly linked lists
- Hash tables
- Trees
- Many many others …

Encapsulation

- Using the language of your choice (C++, Java, Fortran 90)
  - Create objects that contain the data.
  - The objects have methods through which the user interacts
    - Pop(), push(), insert(), search(), add(), delete(), find(), etc.
  - The implementation of these methods is done through data structures, the special way to organize data.

Container Class Examples

- The following three examples are interfaces to containers (abstract classes in C++ or interfaces in Java).
- These interfaces determine how the user interacts with these objects.
- Data-structures are required to implement these interfaces

Set class

```
template <class T>
class Set {
    …
    insert(T obj);
    int count();
    T first();
    T next();
};
```

Stack class

```
template <class T>
class Stack {
    Stack(int n);
    void push(T obj);
    T pop();
};
```

Table class

```
template <class T>
class Table {
      Table(int n);
      T getElement(int i);
      void putElement(T elem, int i);
      ….
};
```

Data Structure Array class

```
template <class T>
class Array {
      Array(int n);
      T Operator[](int i);
      void putElement(T elem, int i);
      ….
};
```

(maps nicely to a table)

Data Structure Linked List class

```
template <class T>
class LinkedList {
      T* first_element;
      T* next();
      T* first();
      LinkedList(int n);
      void putElement(T* elem, int i);
      void deleteElement(T* elem);
      ….
};
```

(does not map nicely to a table)

Primary Considerations

- Search speed
- Insertion speed
- Deletion speed
- Memory usage
- Parallelization issues
- Cache
- disk storage
- virtual memory

Secondary Considerations

- Is there upper limit to the number of elements
- Are the elements in the container actual elements (or containers) or pointers to the actual data/containers?
- Memory allocation schemes
- Are all the data in RAM, or is the number of elements large enough to require storage on disk?
- Does the data have a natural ordering? Is access in sorted order a requirement?

Data structure implementations

- Depends on:
  - Required methods
  - Number of elements
  - In or out of core storage
    - Available memory
  - Parallel or serial algorithms
  - Local or distributed clusters
    - Slower speed might require less memory

Dimensionality

- One-dimensional data:
  - Compression of signals
  - Sparse 1D arrays
- Two-dimensional data
  - Compression of images
  - Voronoi meshes (see lab)
  - Sparse matrices
- Three-dimensional data
  - Adaptive 3D grids
  - Scene to be ray-traced

Examples

- Adaptive Integration (Course work)
- Voronoi Meshes (Lab work) (www.voronoi.com)
- Handling of adaptive grids (ACS-2)
- Triangular meshes
- Adaptive Multiresolution meshes
- Searching for the polygon containing a point on a tiled grid
- Dividing a random set of points (with attached weights) into n sets (I.e., load balancing)

Outline

- Data and Data Structure
- Array
- Linked List
- Ex: Stacks, Queues, Sets
- Hash Tables
- Trees: Quadtree, Octree
- Other data structures

1D Array

- In its most abstract sense, an array is a collection of (index, value) pairs:
  - (3, ‘house’)
  - (2, ‘grid’)
- Notation:
  - a(3) = ‘house’ // F90
  - a(2) = ‘grid’
- Altenative notations:
  - a[2] = ‘house’   // C++
  - a[2] <--- ‘house’ // computer science

1D Array

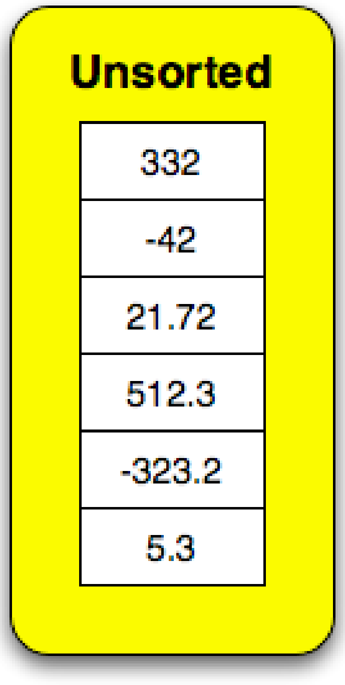
- Mathematically :
  - An array is a function that maps an integer index i to some content.
  - The content is usually homogeneous across all array elements
  - The contents of the array are usually (not always) contiguous in memory
- Arrays can be static or dynamic

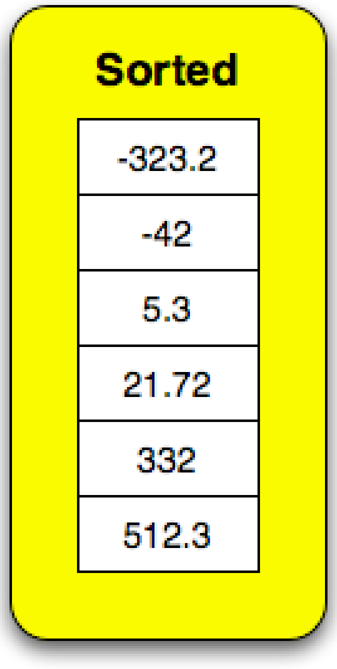
Array

- Static array: the array size is set at compile time and cannot be changed
- Dynamic array: a pointer points to where the array data is located in memory, usually allocated via *new*
  - Memory can be allocated and released during the simulation
  - The array size can be increased/decreased during the simulation
- Data in the array can be sorted or unsorted

Array

- Assume array elements are real numbers





Cost

- What is the cost of
  - Sorting an array
  - Finding the maximum element
  - Adding an element at
    - The beginning of the array
    - At the end of the array
  - …
- Answer: it depends on the underlying data structure

Asymptotic behavior of *f(n)*

- Usually concerned with behavior of *f(n)* as *n* becomes arbitrarily large (goes to infinity)
- If *p(n)* and *q(n)* are non-negative functions, *p(n)* is asymptotically bigger than *q(n)* if and only if

$$\lim_{n\to\infty} \frac{q(n)}{p(n)} = 0$$

- Similar definition for asymptotically smaller
- Asymptotically equal occurs when it is neither small or larger

Example

$$\lim_{n\to\infty} \frac{10n+7}{3n^2+2n+6} = 0$$

Therefore, $3n^2 + 2n + 6$ is asymptotically larger than $10n + 7$

Usually, identify the dominant term in an expression to evaluate the above ratio. In the above, this would be $3n^2$ for the denominator and $10n$ for the numerator. Therefore,

$$\lim_{n\to\infty} \frac{10n+7}{3n^2+2n+6} = \lim_{n\to\infty} \frac{10n}{3n^2} = \lim_{n\to\infty} \frac{10}{3n} = 0$$

O(n) notation

$$f(n) = O(g(n))$$

is equivalent to stating that $f(n)$ is asymptotically smaller or equal to $g(n)$

Large *n*

$$10n + 7 = O(3n^2 + 2n + 6)$$
$$= O(2n^3 + 3)$$
$$2n^2 + 3 \neq O(n)$$

Small *n* -> *0*

$$3n^2 + 2n + 6 = O(10n + 7)$$
$$= O(8)$$
$$2n + 3 \neq O(n^2)$$

Asymptotic Ordering

$$\frac{1}{n!} < \frac{1}{n\log n} < n^{-1} < 1 < \log(n) < n < n\log n < n^2 < n^3 < n!$$

< means asymptotically smaller

$\log n$ is asymptotically smaller than n

$n\log n$ is asymptotically smaller than n^3

$$\log n = O(n)$$
$$n\log n = O\left(n^3\right)$$

Asymptotic complexity

Given a function $f(n)$ as a sum of terms, find the dominant term, which becomes the asymptotic estimate. Finally, set the numerical coefficient to 1.

$$3n^2 \log n + n + 3 = O\left(3n^2 \log n\right) = O\left(n \log n\right)$$

$$\frac{3n^2 - 2n\log n}{n^3} == \frac{3n}{n^2} - \frac{2\log n}{n^2} = O\left(3n^{-1}\right) = O\left(n^{-1}\right)$$

When estimating asymptotic complexity, one usually finds the lowest upper asymptotic bound

$$3n^{3/2} + 2n\log n + 3n + 2 = \boxed{O\left(n^{3/2}\right)} = O\left(n^2\right)$$

The asymptotic complexity is $O\left(n^{3/2}\right)$

Cost of insertion

Given an array of size n and capacity N (number of elements allocated), add a number R after element $i < n$

$$a_{j+1} \leftarrow a_j,\ j = n, n-1, \cdots, i+1$$
$$a_{i+1} \leftarrow R$$

Cost: $n - (i+1) + 1 = n - i$

I counted each shift as one operation

Cost of random insertions

Now assume one performs this operation for each i ranging between 0 and $n-1$. The average cost is thus

$$\frac{(n-0)+(n-1)+\cdots+1}{n}=\frac{n^2-(0+1+\cdots+n-1)}{n}$$
$$=\frac{n^2-\dfrac{(n-1)n}{2}}{n}=\frac{n^2+n}{2n}=O(n)$$

Therefore, if one adds new elements to an array of length n at random locations, the average cost is $O(n)$

Sorted vs unsorted

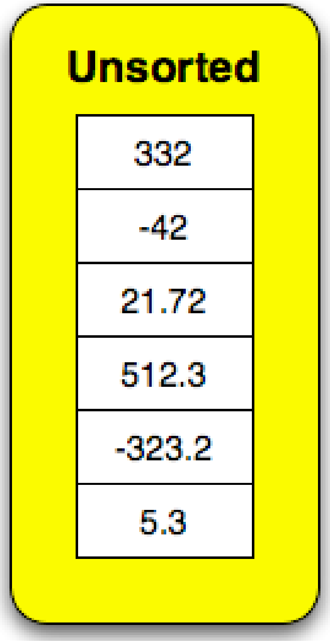
- Unsorted containers
  - Sets, dictionaries, tables
- Sorted containers
  - Tables, trees

- Storing elements according to some intrinsic ordering often speeds up search operations
- Not all data structures can take advantage of sorting

Unsorted Array

- Search for the number 512.3
- Must sequentially check each element of the array
- Average cost for random element: O(n) where n is the number of array elements

```
find(double val) {
   for i=0; i < length(array); i++) {
      if (array[i] == 512.3) break;
   }
   return i;
}
….
 int indx = find(512.3)
```



Sorted Array: binary search

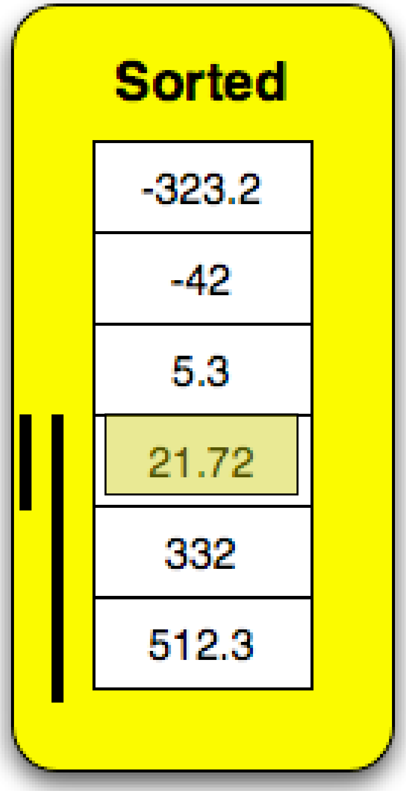
```
int find(DynamicArray& arr, double val)
{
    int sz = arr.getSize();
    int n1=0, n2= sz-1, indx;
    bool found=false;

    while (!found) {
        indx = n1 + (n2-n1)/2;
        printf("n1,n2,indx= %d, %d, %d\n", n1, n2, indx);
        if (arr[indx] == val) {
            found = true;
            break;
        }
        if (val <= arr[indx]) {
            n2=indx;
        else
            n1 = indx+1;
    }
    if (found) return indx;

    return(-1);
}
```

Divide search interval in half, find the interval that contains val, repeat recursively

Search time for random element is O(log *n*)



Array: insertion/deletion

- Insertion cost
  - Front: O(n)
  - Rear: O(1)
  - Arbitrary: O(n) if adding many elements at random locations
- Deletion cost
  - Front: O(n)
  - Rear: O(1)
  - Deletion of arbitrary elements: O(n)

A 1D Matrix

- Full matrix (of numbers)
  - Sometimes the numbers are unsorted, e.g. to store a temperature profile. To compute derivatives, adjacent elements are required
  - Sometimes they are sorted: if order is not important, yet retrieval of arbitrary elements is important.

1D Matrix

$$a = \begin{pmatrix} 3 & 2 & -1 & -2.2 & -1.2 & 0.7 & 1.7 & 2.6 & 3.9 & 5.8 \end{pmatrix}$$

Implementation

$a_0 = 3$

$a_1 = 2$

$a_6 = 1.7$

$a_9 = 58$

```
float a[10];
float* b = new float[10];
```

C++

```
real, dimension(10) :: a
real, dimension(:), allocatable :: b
allocate b(10)
```

Fortran

```
float[] a = new float [10];
```

Java

Operations

- Define a 1D matrix *a* of size 10
- Insert the 5 elements (in order): -2, 3.5, -1.2, 7.2, 6.1

$$a[0] = -2$$
$$a[1] = 3.5$$
$$a[2] = -1.2$$
$$a[3] = 7.2$$
$$a[4] = 6.1$$

1. Find the maximum element
2. Find the minimum element
3. Find the 2nd element
4. Insert an element after the 3rd element
5. Delete the 3rd element

Operations

Length of array: *n*

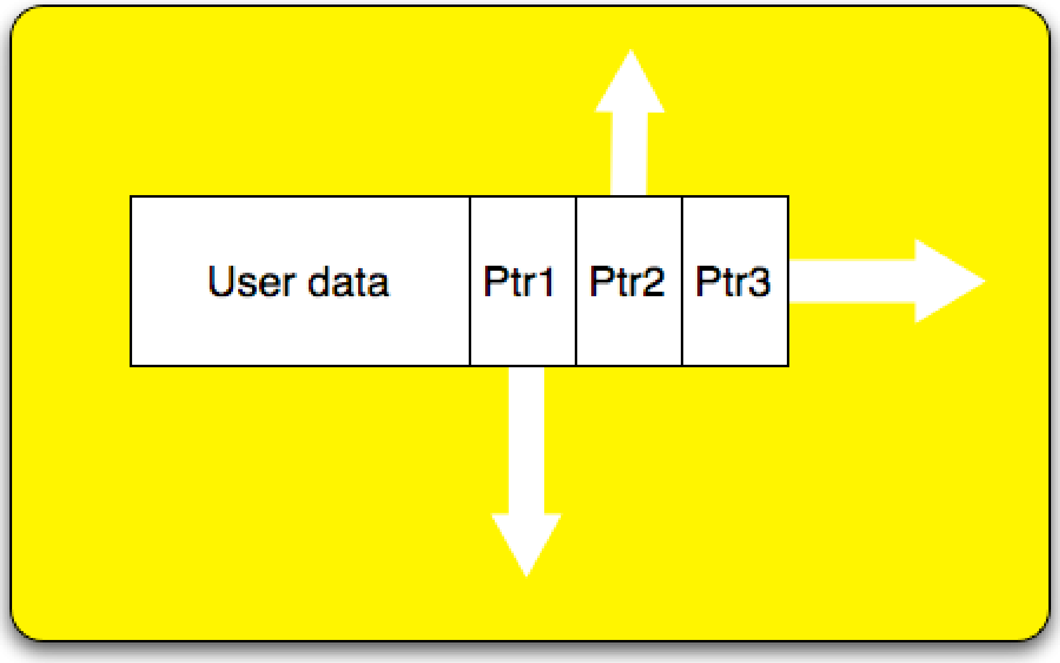
- Find the maximum element
  - Cost: O(n)
- Find the minimum element
  - Cost: O(n)
- Find the 2nd element
  - Cost: O(1)
- Insert an element after the 3rd element
  - Elements a[3] to a[n-1] must be shifted upwards
  - Average Cost: O(n)
- Delete the 3rd element
  - Elements a[2] to a[n-1] must be shifted downwards
  - Average Cost: O(n)
- Element search
  - Average cost: *O(n)*

Outline

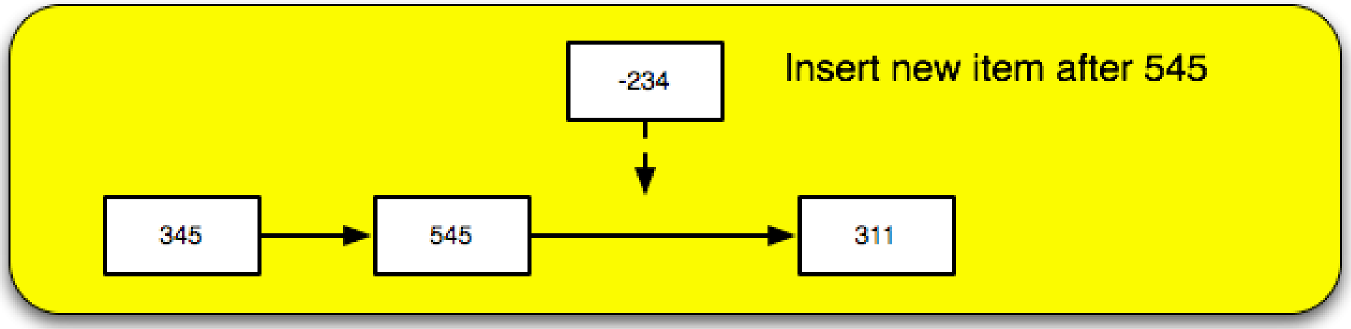
- Data and Data Structure
- Array
- Linked List
- Ex: Stacks, Queues, Sets
- Hash Tables
- Trees: Quadtree, Octree
- Other data structures

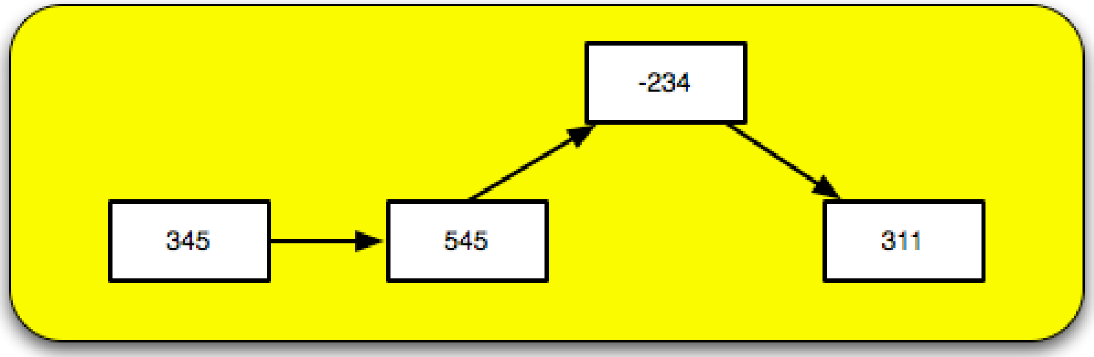
Typical Structure Element

A structure has any number of pointers pointing to other structures, similar or not

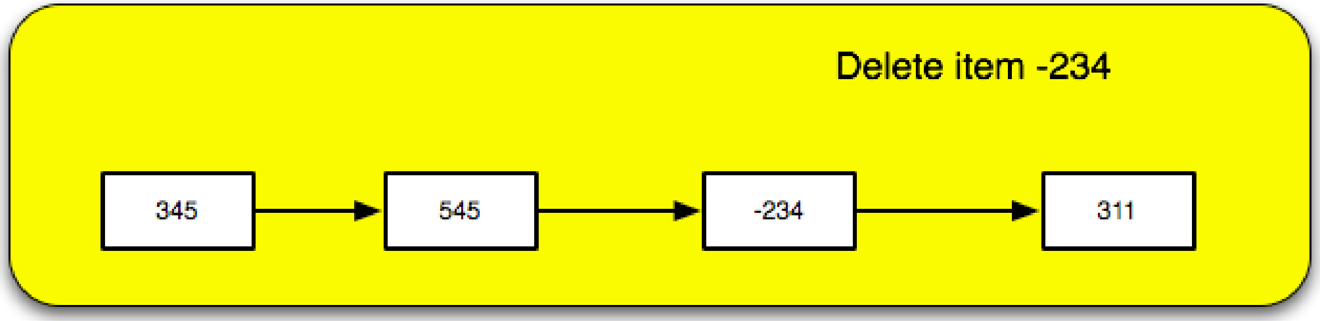


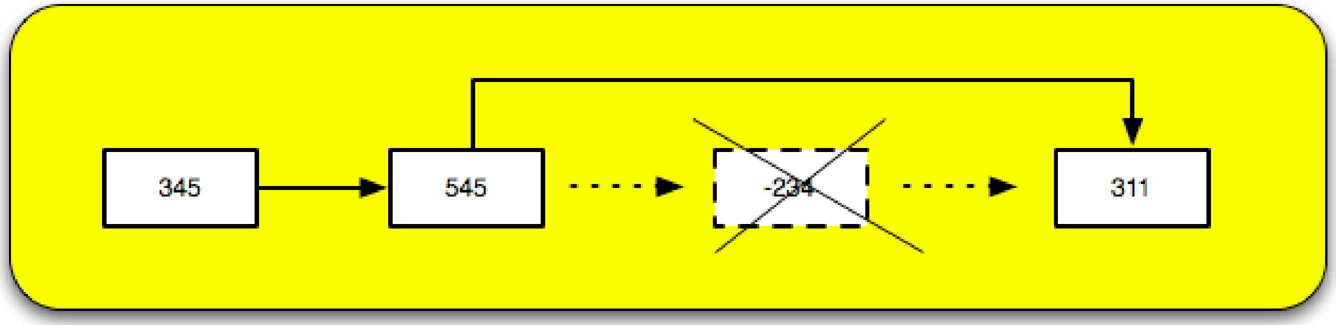
Linked List





Linked List

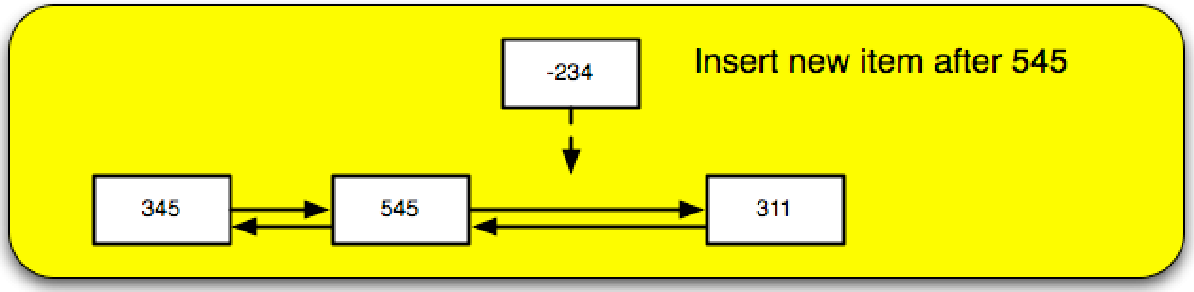


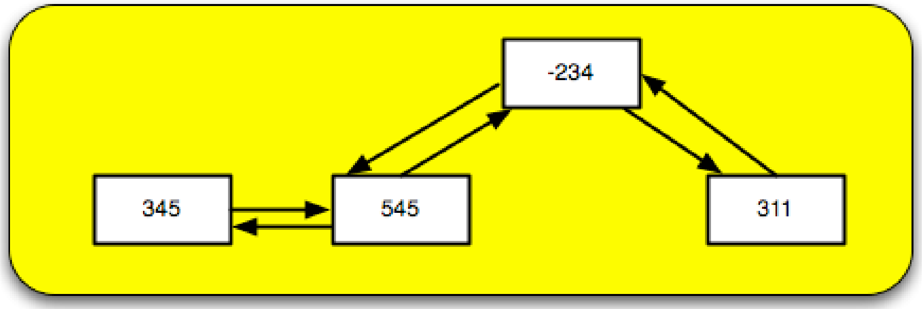


Linked List Efficiency

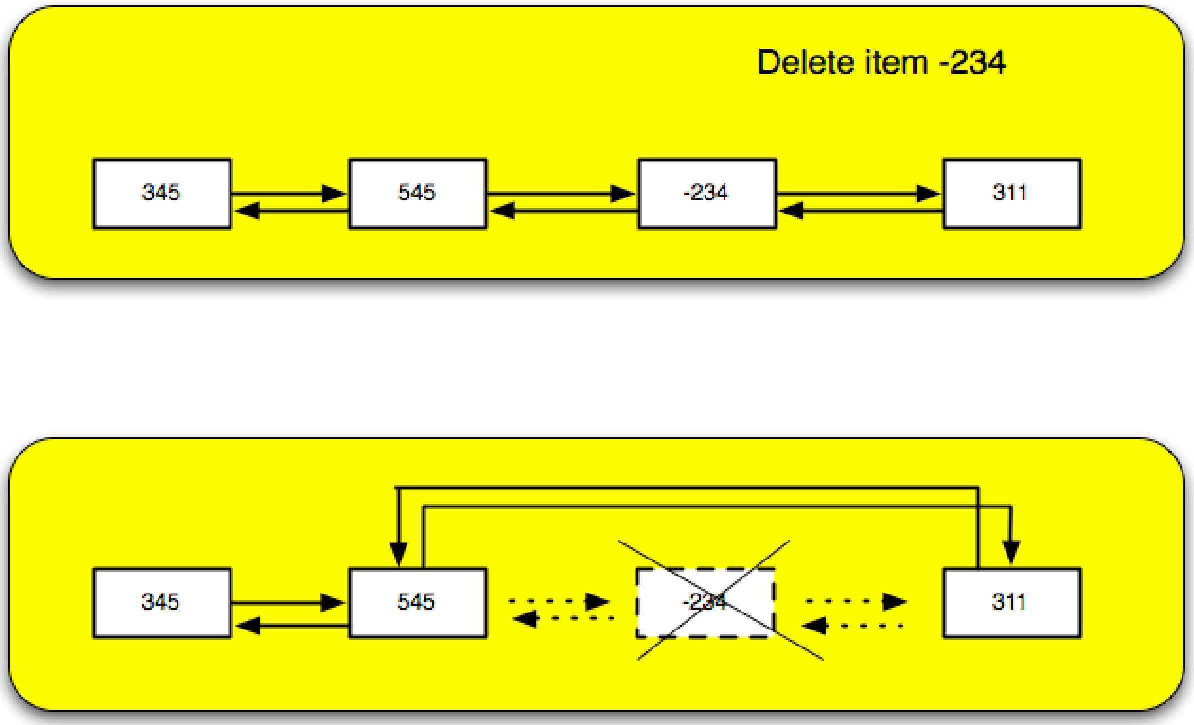
- Number of elements: n
- Insertion: O(1)
- Deletion: O(n)
- Searching: O(n)

Doubly Linked List





Doubly Linked List



Doubly Linked List

- Data structure has pointers to both next and previous element
- This allows insertion and deletion at a cost of *O(1)* (the cost is independent of the length of the list)
- Can run through the list forward and backward, starting from any element

Considerations

- Often, higher efficiency is obtained at the cost of
  - Higher memory usage
  - Higher preprocessing cost
- If data is predetermined, expensive preprocessing is acceptable if only done once, and if the result is fast access, searching, deletion
- Requirements of data structure depends on the data to be encoded

Google

- Use google with:
  - Data structures filetype:pdf
  - Huge selection. Demonstrate different ways of using data structures

Outline

- Data and Data Structure
- Array
- Linked List
- Ex: Stacks, Queues, Sets
- Hash Tables
- Trees: Quadtree, Octree
- Other data structures

Stacks

Last In First Out

- A user inserts a series of objects into a container
- The object removed should be the last one inserted
  - In graphics: push and pop matrix
  - Stack of plates: take off last one added
- Operations:
  - push() (add element to the stack)
  - Pop()   (delete element from the stack)

Stack implementations

- Using arrays
  - Possible. Disadvantages: must know maximum array length, unless array is dynamic
  - Adding and deleting are *O(1)* operations if elements are inserted/deleted from the end of the array
  - Memory efficient
- Using a linked list
  - Insertions and deletions are O(1) operations
  - Disadvantage: slightly higher memory usage since one must store a pointer (4 or 8 bytes) for each element.

Queue

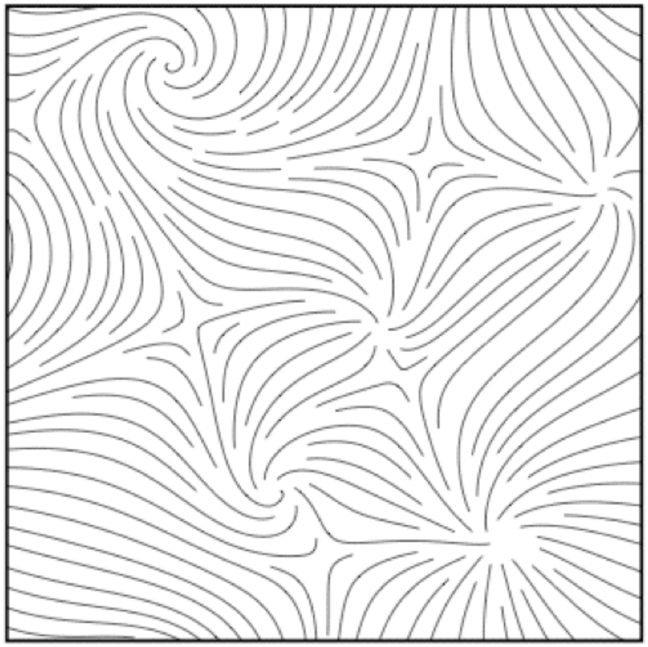
First In First Out

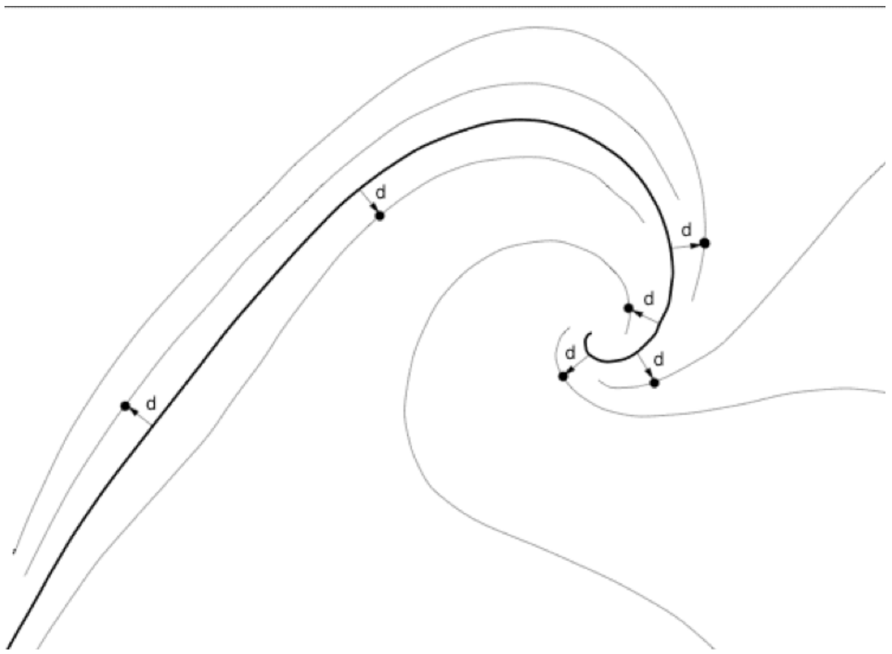
- Insert a series of objects into a container
- Remove the objects from the container in the same order they are added
  - Non-prioritized batch system for job submissions on supercomputer
  - Customer support phone calls
  - Airplane departures

Queue implementations

- Using arrays
  - Possible. Disadvantages: must know maximum array length, unless array is dynamic
  - Add to the front of the array, delete from the rear.
  - Most efficient implementation with circular array
- Using a linked list
  - Insertions and deletions are O(1) operations
  - Disadvantage: slightly higher memory usage since one must store a pointer (4 or 8 bytes) for each element
  - Use a circular linked list

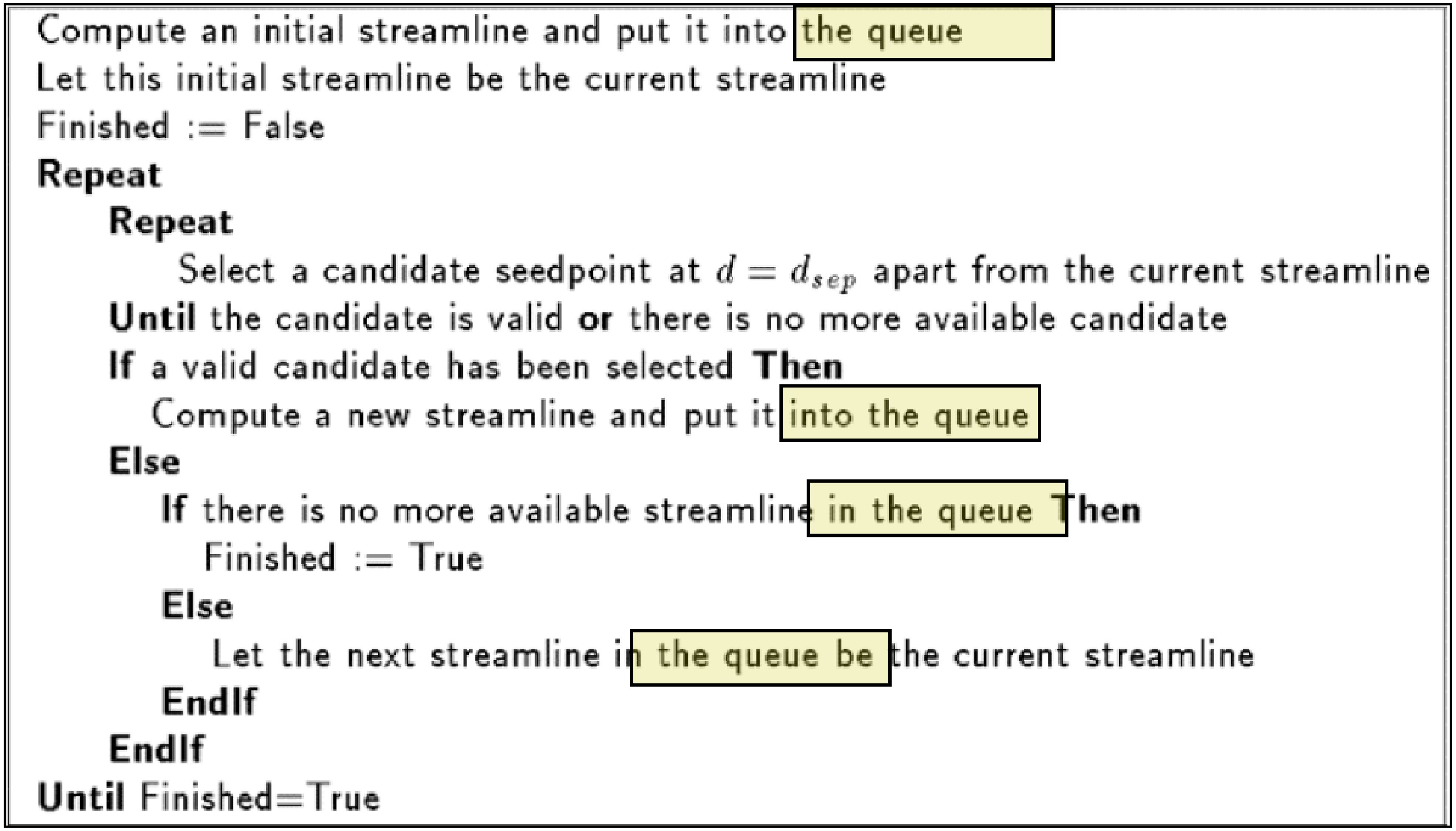
Equidistant streamlines
(Jobard & Lefer)





Equidistant streamlines

(Jobard & Lefer)



```
Compute an initial streamline and put it into the queue
Let this initial streamline be the current streamline
Finished := False
Repeat
    Repeat
        Select a candidate seedpoint at $d = d_{sep}$ apart from the current streamline
    Until the candidate is valid or there is no more available candidate
    If a valid candidate has been selected Then
        Compute a new streamline and put it into the queue
    Else
        If there is no more available streamline in the queue Then
            Finished := True
        Else
            Let the next streamline in the queue be the current streamline
        EndIf
    EndIf
Until Finished=True
```

Stacks and Queues

- Implemented with Arrays
- Implemented with Linked Lists

- Both implementations are simple. Linked lists are more general:
  - They use more memory
  - They are faster
  - All operations are O(1)
  - Searching operations are not required

Sets

- A set is a collection of objects
  - Uniform or non-uniform objects
  - In Java, everything is a subclass of object, so, polymorphically, one can create collections of anything
- When accessing a sequence of objects, I cannot guarantee order (as I can with an array)
- Objects can be added to the set in random order
- Operations: inclusion, intersection, union

Sets: implementation

- Linked lists
- Array
- Hash tables

Hash Tables

- Data stored in a container is usually a collection of information (i.e., an instance of a class or a structure)
- Each instance is associated with a key
- A deterministic algorithm maps each key to a hash (I.e., an integer index)
- Each value of the hash is associated with an area in memory where the data is stored.
- Usually, each hash value is associated with a unique, or small number of objects (class instances)

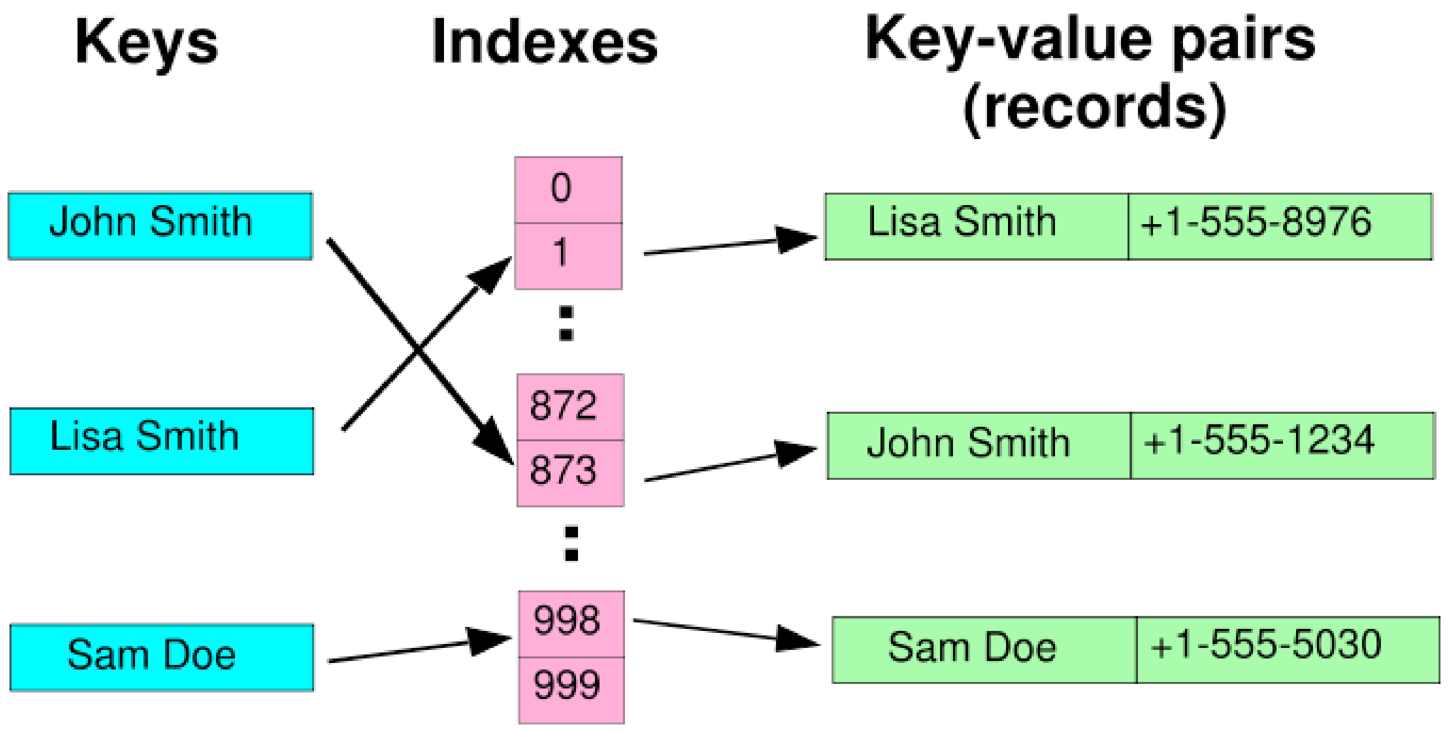
Hash Tables: interface

```
template <class KEY, class VAL>
class Hash {
    Hash(int maxKeys);
    void store(KEY key, VAL val);
    VAL get(KEY key);
    void delete(KEY key);
}
```

Hash Table: example

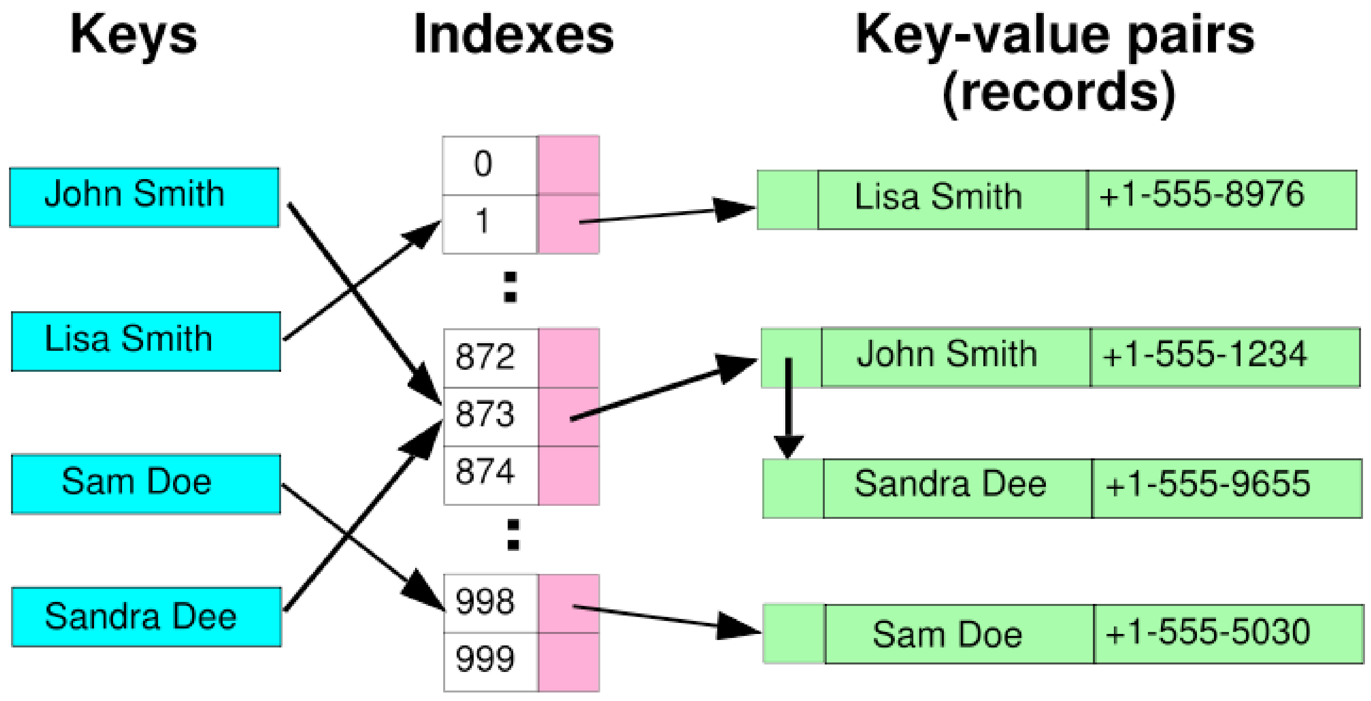
```
Hash hash(128); // max number el: 128
Student* markus = new Student(….);
hash.put(‘mark’, markus);
Student* billy = new Student(…);
hash.put(‘bill’, billy);
….
Student* who = hash.get(‘mark’);
….
```

Hash table



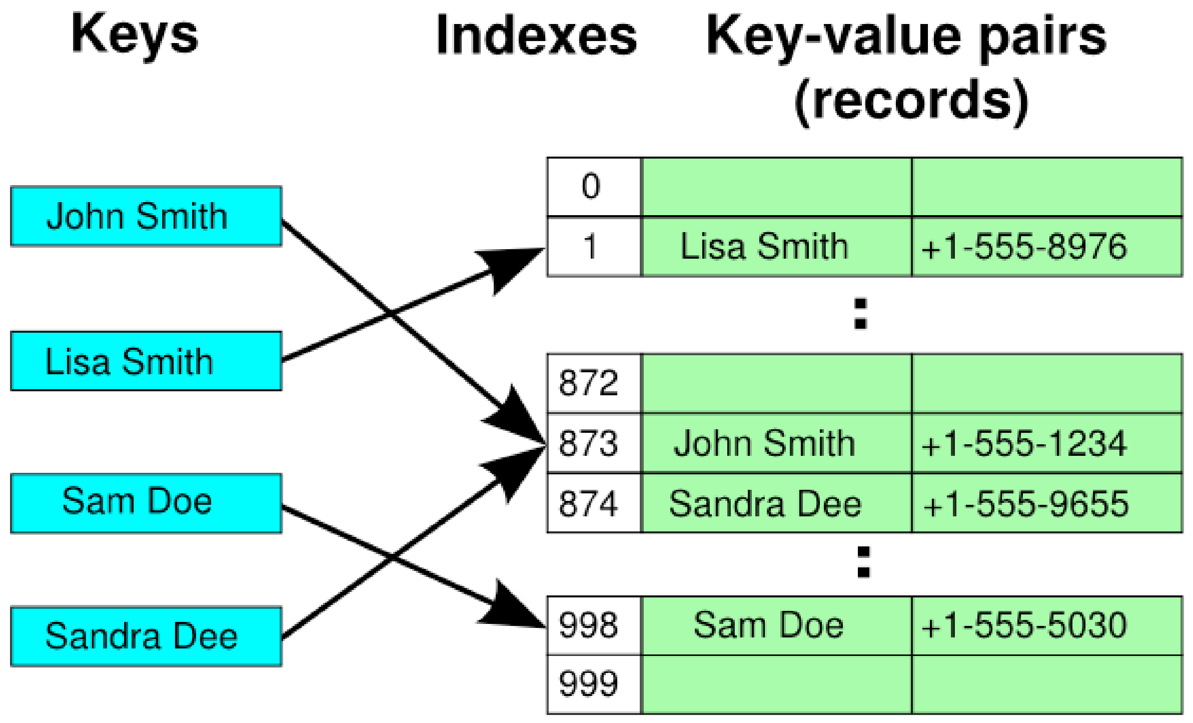
Keys
Indexes
Key-value pairs (records)
John Smith
Lisa Smith
Sam Doe
0
1
872
873
998
999
Lisa Smith
+1-555-8976
John Smith
+1-555-1234
Sam Doe
+1-555-5030

Hash Table: simple chaining



Data is stored ouside the hash array (linked list)

Hash Table: open chaining

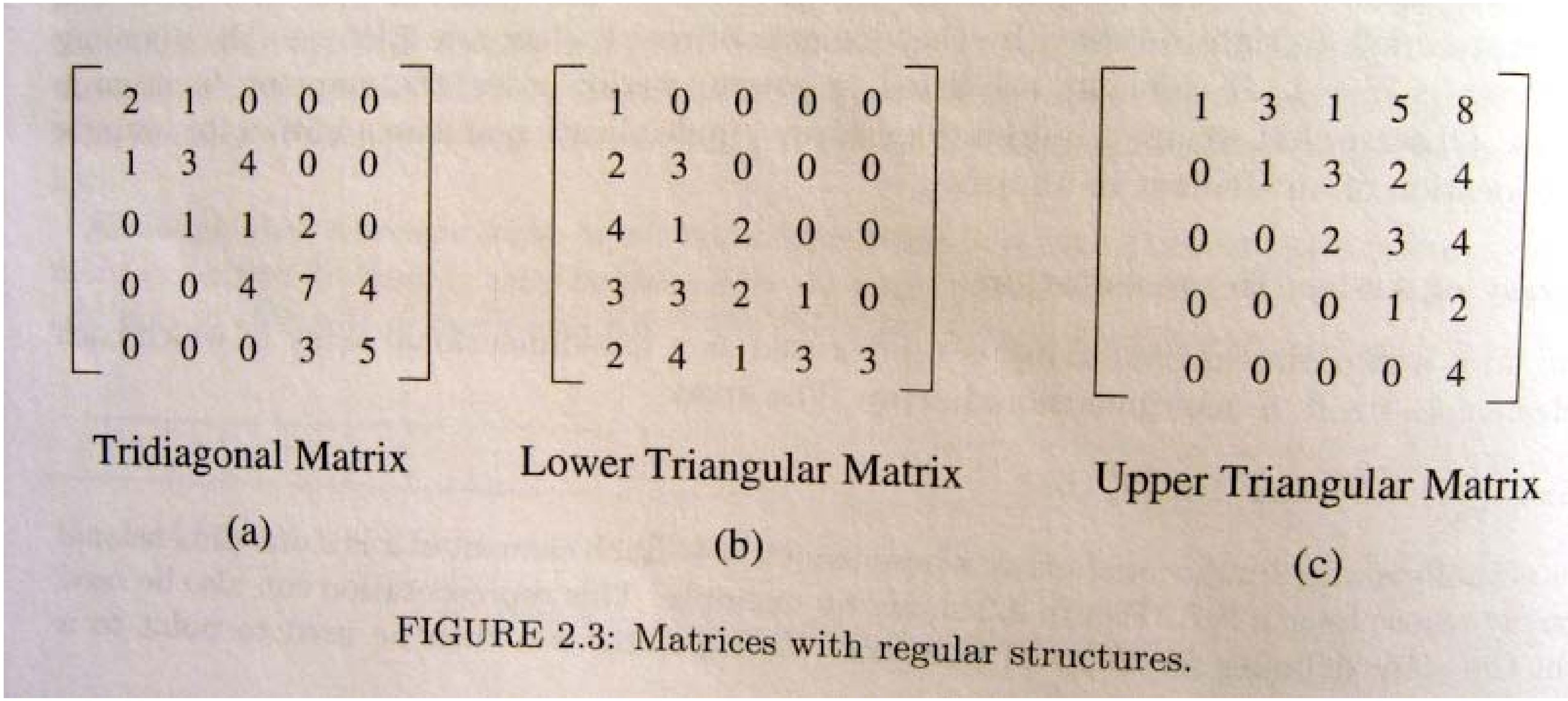


Data is stored within the array

Hash Table cost

- Insertion: O(1)
- Deletion: O(1)
- Access a particular element: O(1)
- Sorting keys: depends on sorting algorithm, but can be efficient
- Sorting on values: quite expensive since order is not known (similar to linked lists)
- Hashing must be reasonably efficient
- Poor hashes can lead to too many elements in a single bin

Matrix Containers



$$\begin{bmatrix} 2 & 1 & 0 & 0 & 0 \\ 1 & 3 & 4 & 0 & 0 \\ 0 & 1 & 1 & 2 & 0 \\ 0 & 0 & 4 & 7 & 4 \\ 0 & 0 & 0 & 3 & 5 \end{bmatrix}$$

Tridiagonal Matrix

(a)

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 2 & 3 & 0 & 0 & 0 \\ 4 & 1 & 2 & 0 & 0 \\ 3 & 3 & 2 & 1 & 0 \\ 2 & 4 & 1 & 3 & 3 \end{bmatrix}$$

Lower Triangular Matrix

(b)

$$\begin{bmatrix} 1 & 3 & 1 & 5 & 8 \\ 0 & 1 & 3 & 2 & 4 \\ 0 & 0 & 2 & 3 & 4 \\ 0 & 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 0 & 4 \end{bmatrix}$$

Upper Triangular Matrix

(c)

FIGURE 2.3: Matrices with regular structures.

How C/C++ and Java handle multi-dimensional arrays

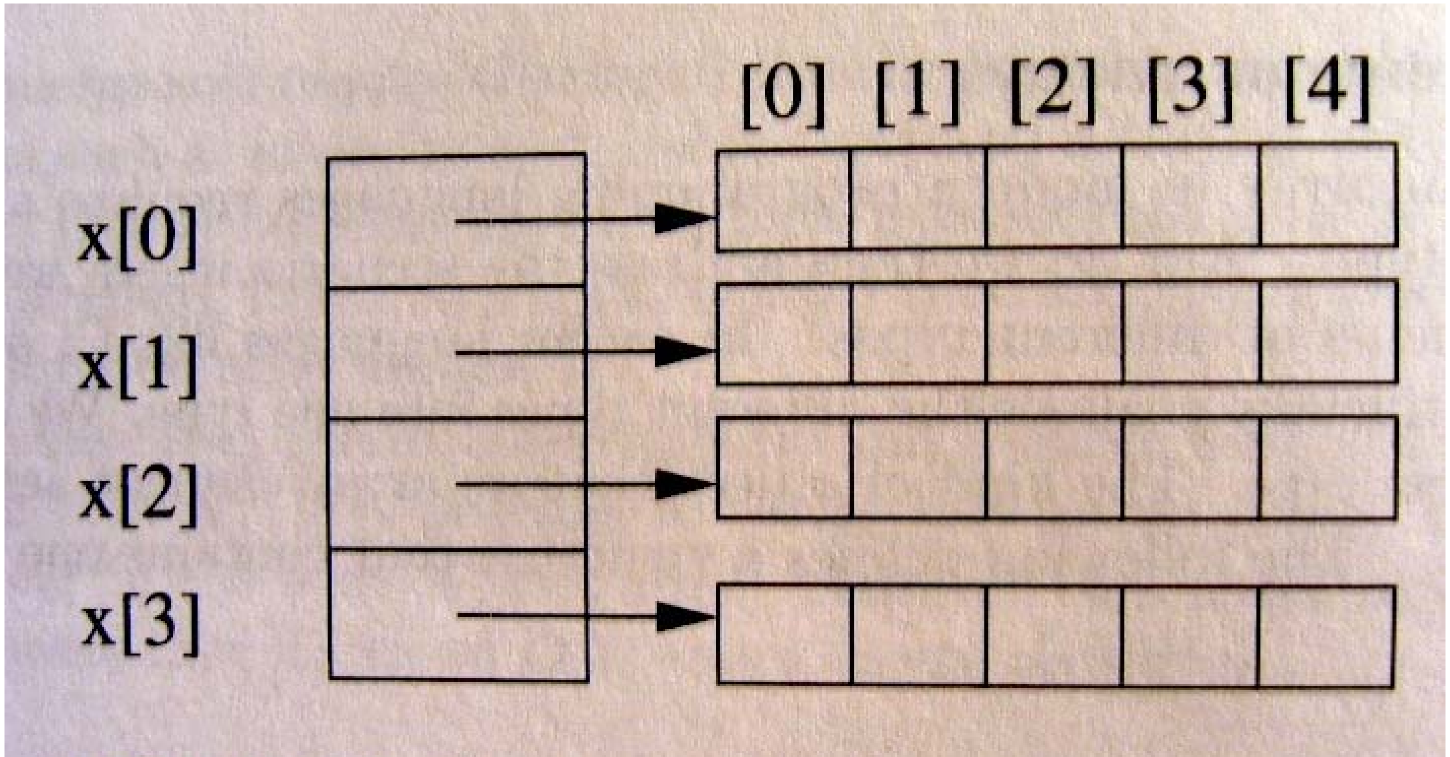
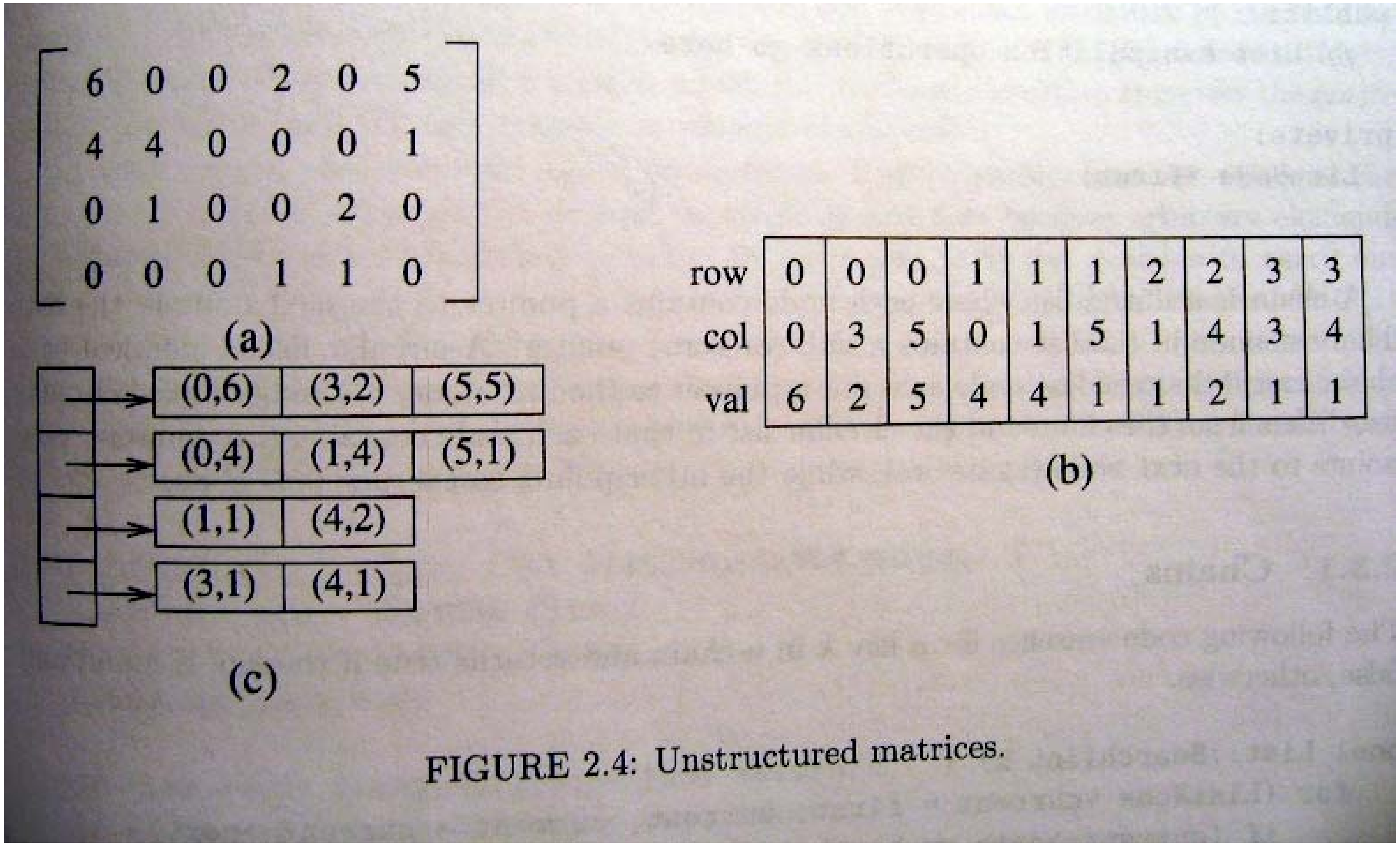


FIGURE 2.2: The Array of Arrays Representation.

Sparse Matrices



$$\begin{bmatrix} 6 & 0 & 0 & 2 & 0 & 5 \\ 4 & 4 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

(a)

row	0	0	0	1	1	1	2	2	3	3
col	0	3	5	0	1	5	1	4	3	4
val	6	2	5	4	4	1	1	2	1	1

(b)

(c)

FIGURE 2.4: Unstructured matrices.

A Queue as a circular array

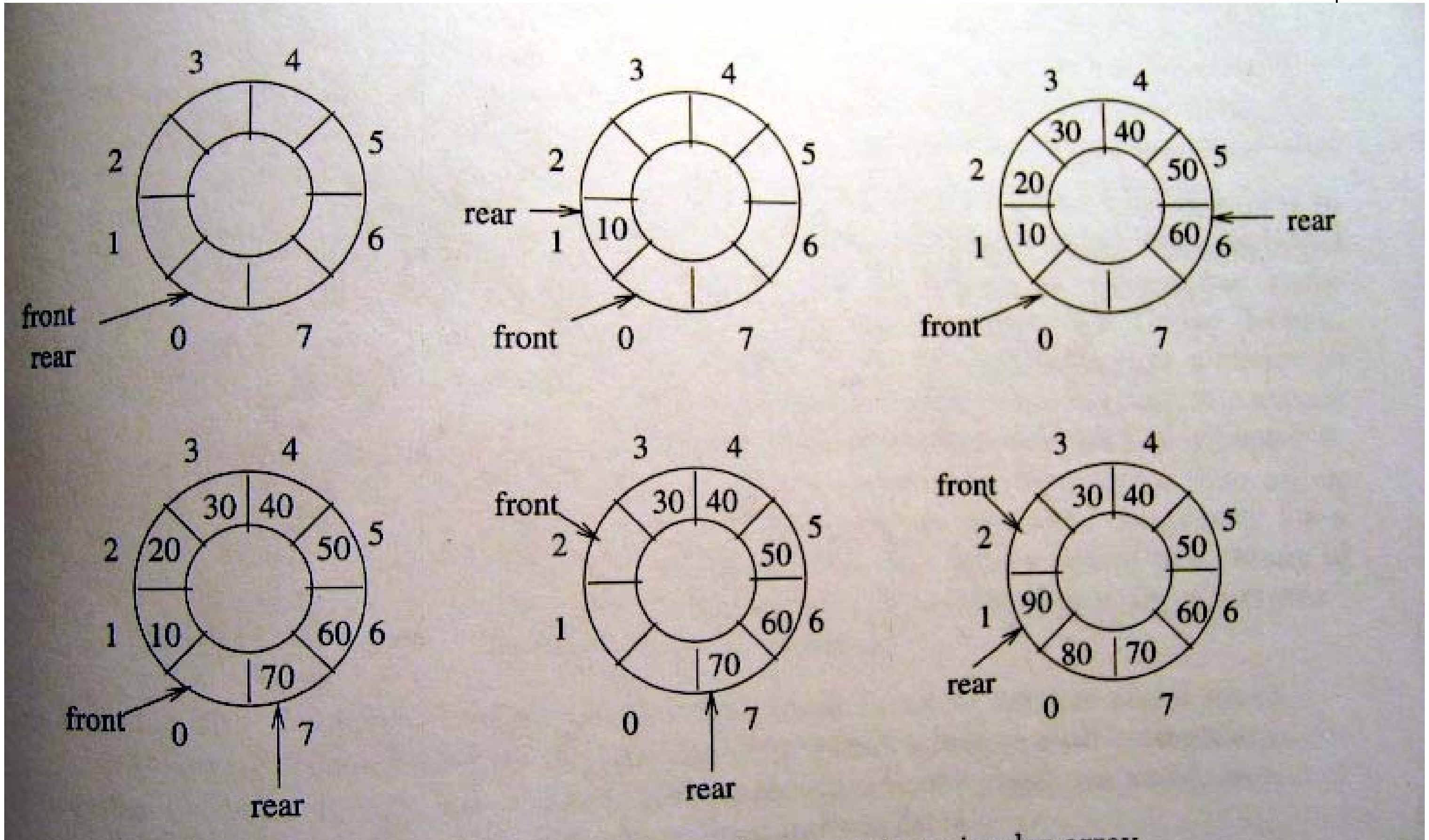


FIGURE 2.13: Implementation of a queue in a circular array.

Generalized List

Each element can point to both another element and another list

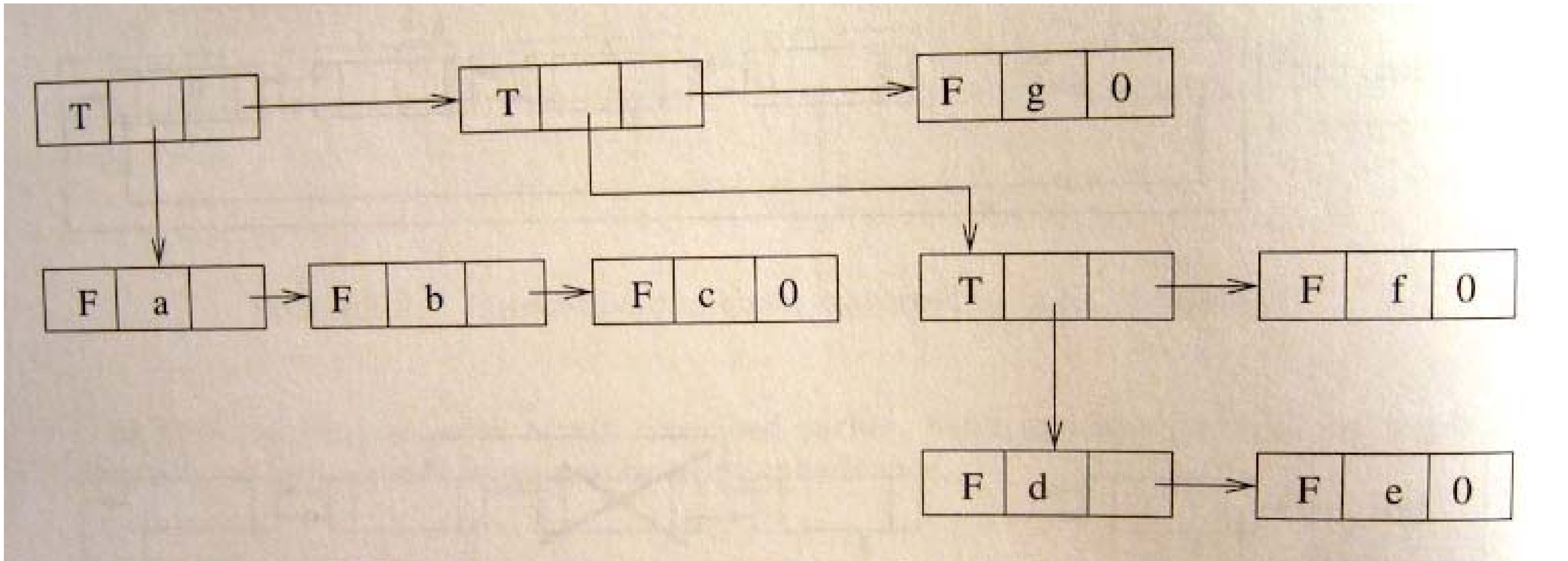


FIGURE 2.10: Generalized List for ((a,b,c),((d,e),f),g).

Circular List

Last element points back to the first

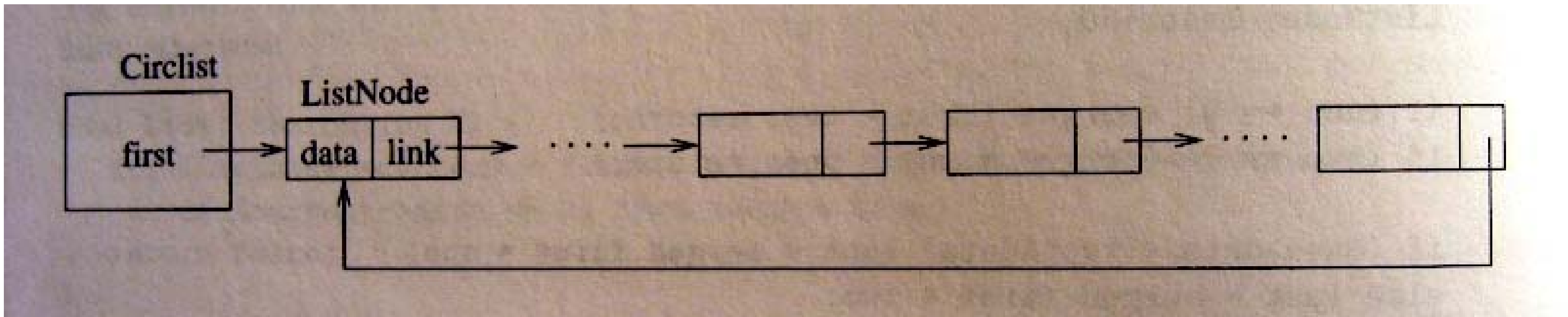


FIGURE 2.8: A circular list.

A Tree

All nodes have at most one parent

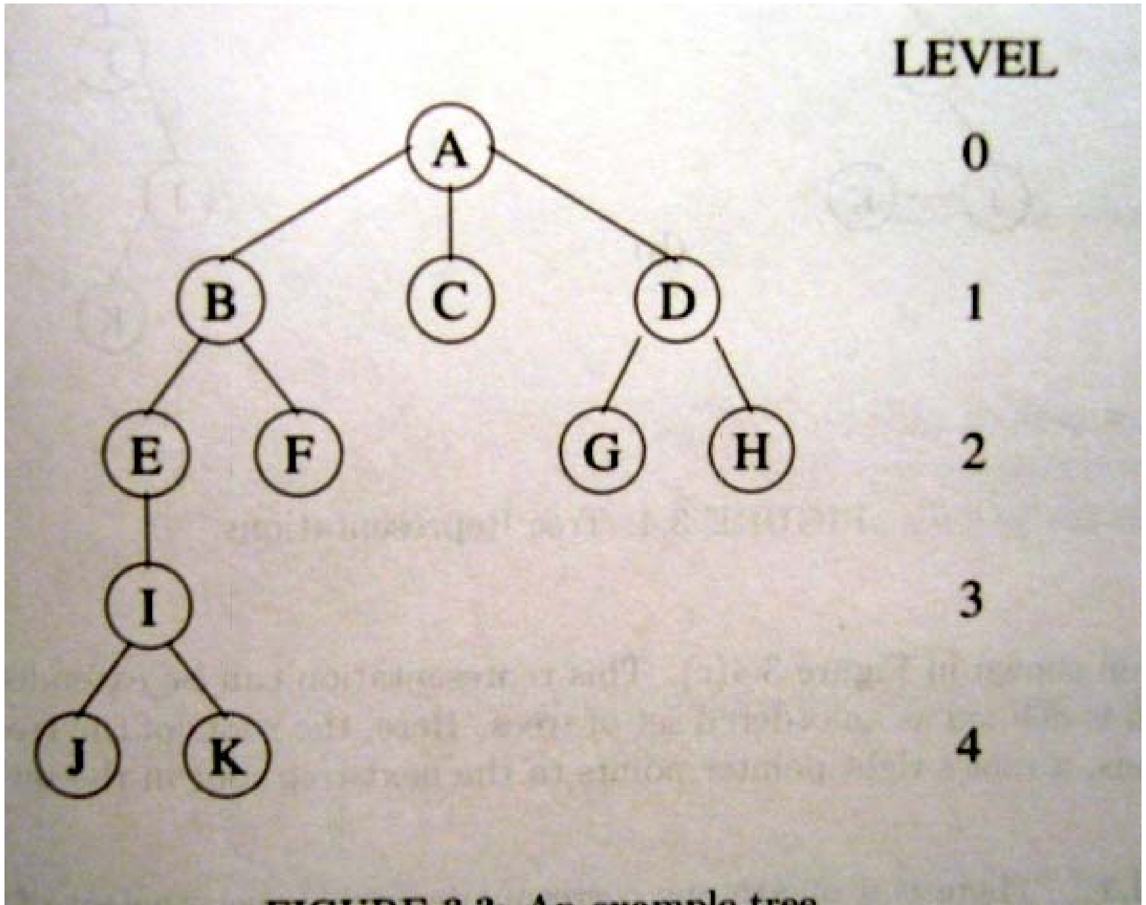
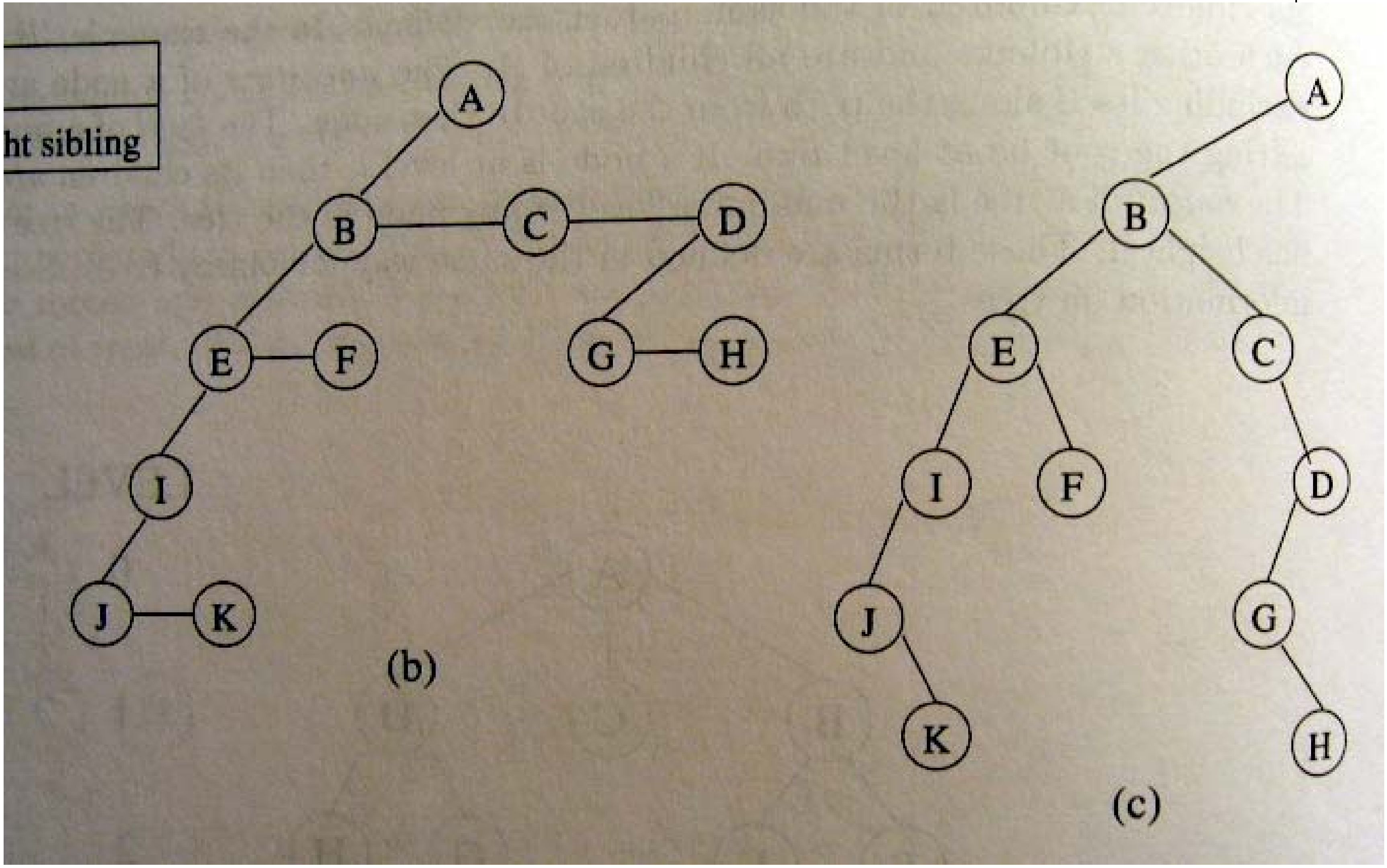


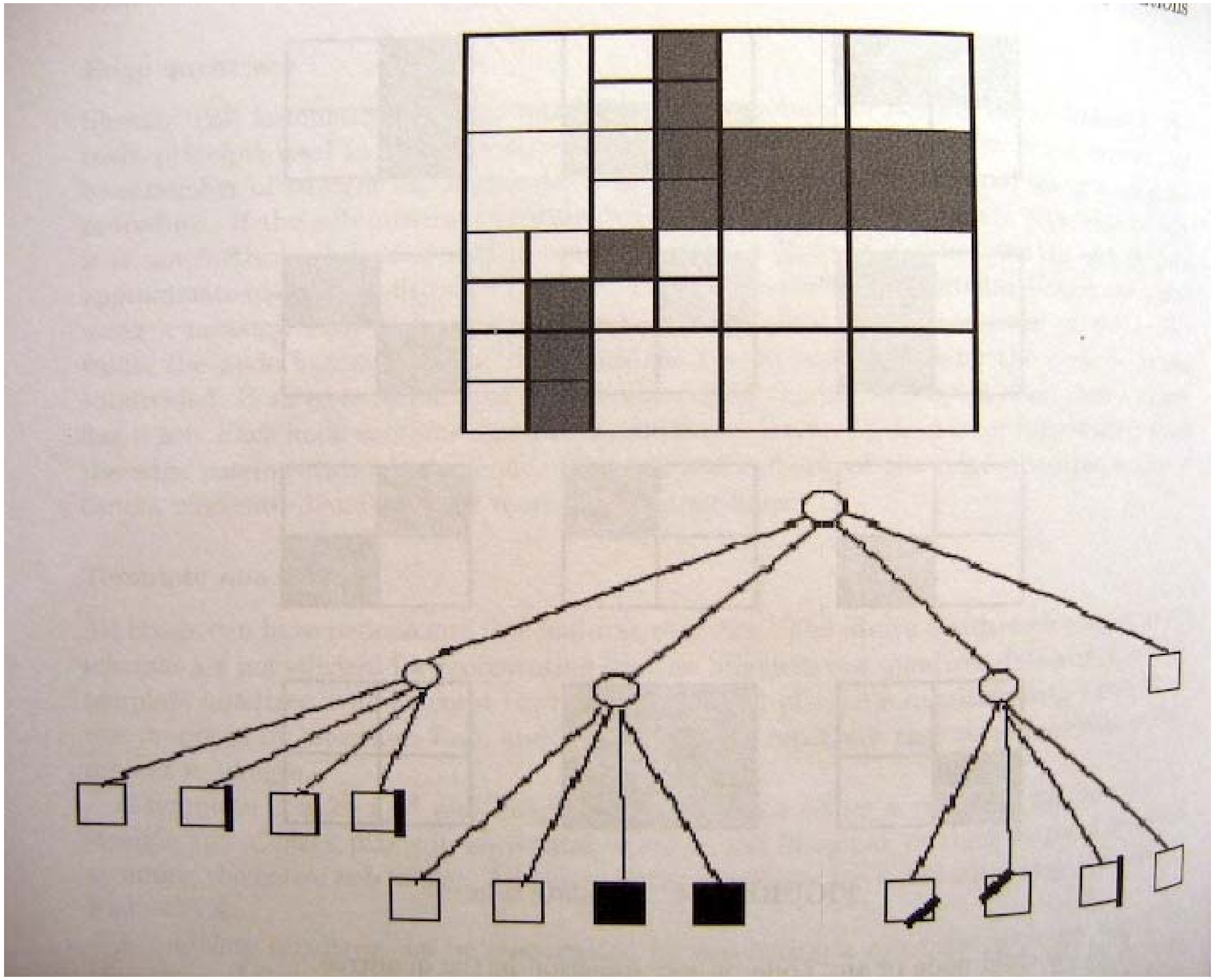
FIGURE 3.3: An example tree.

Different tree representations

Data is the same in both cases



Quadtree



Octree

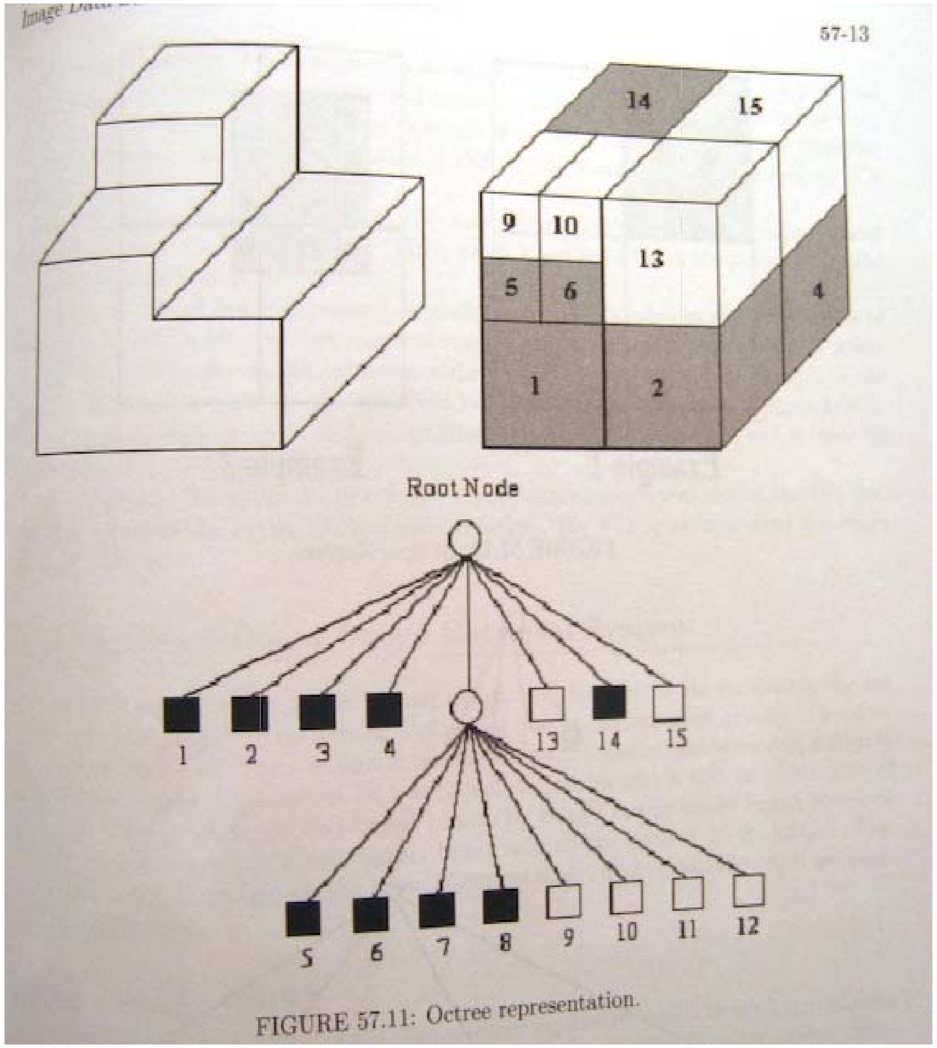
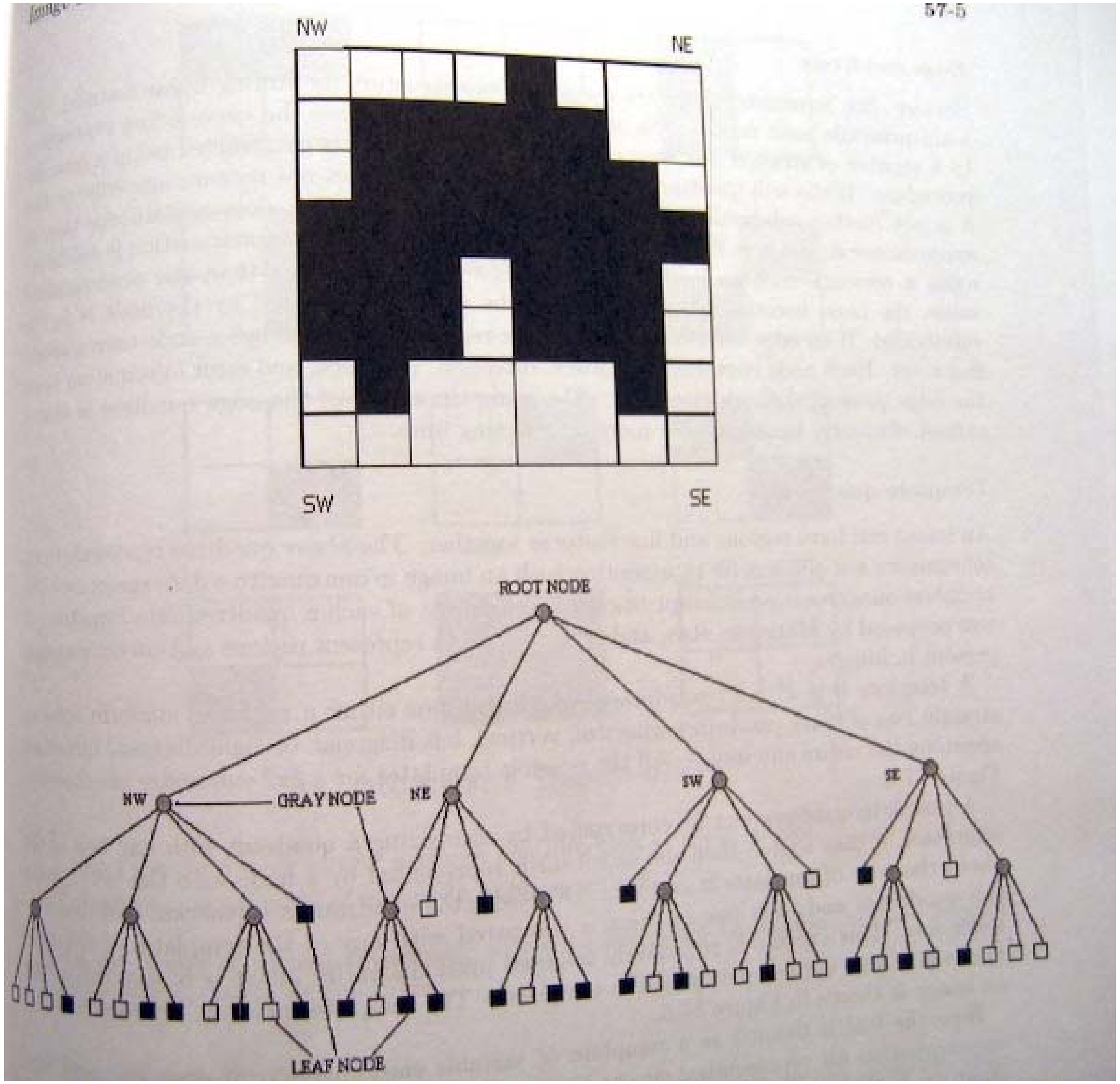
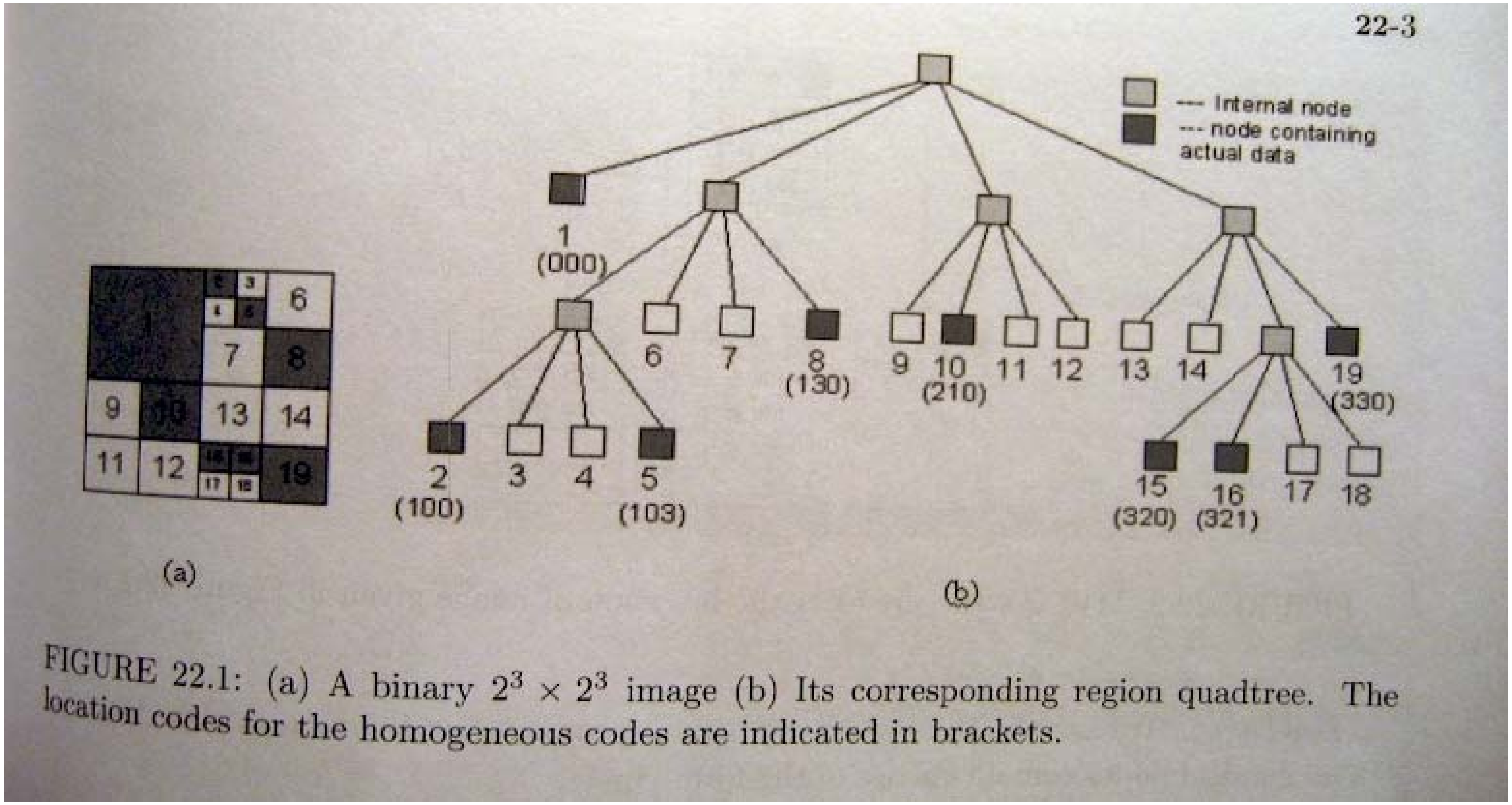


FIGURE 57.11: Octree representation.

Quadtree



Quadtree



22-3

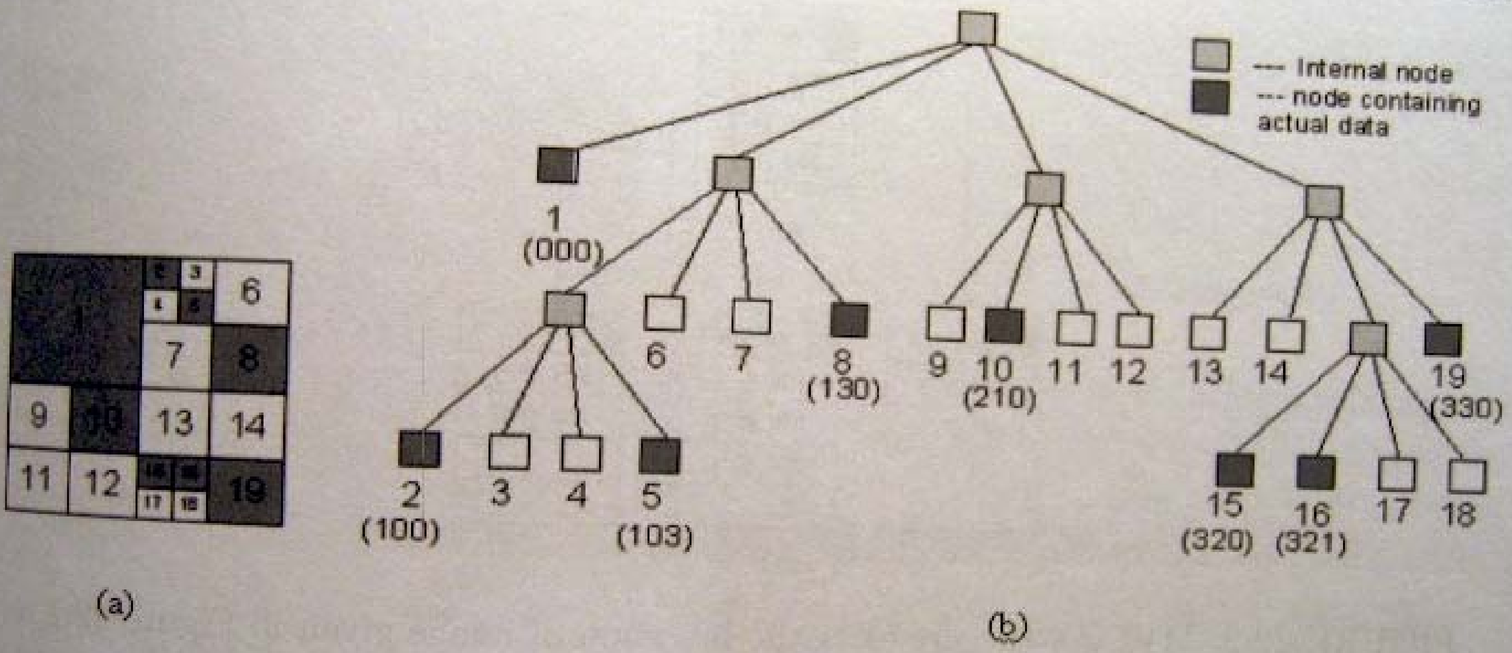
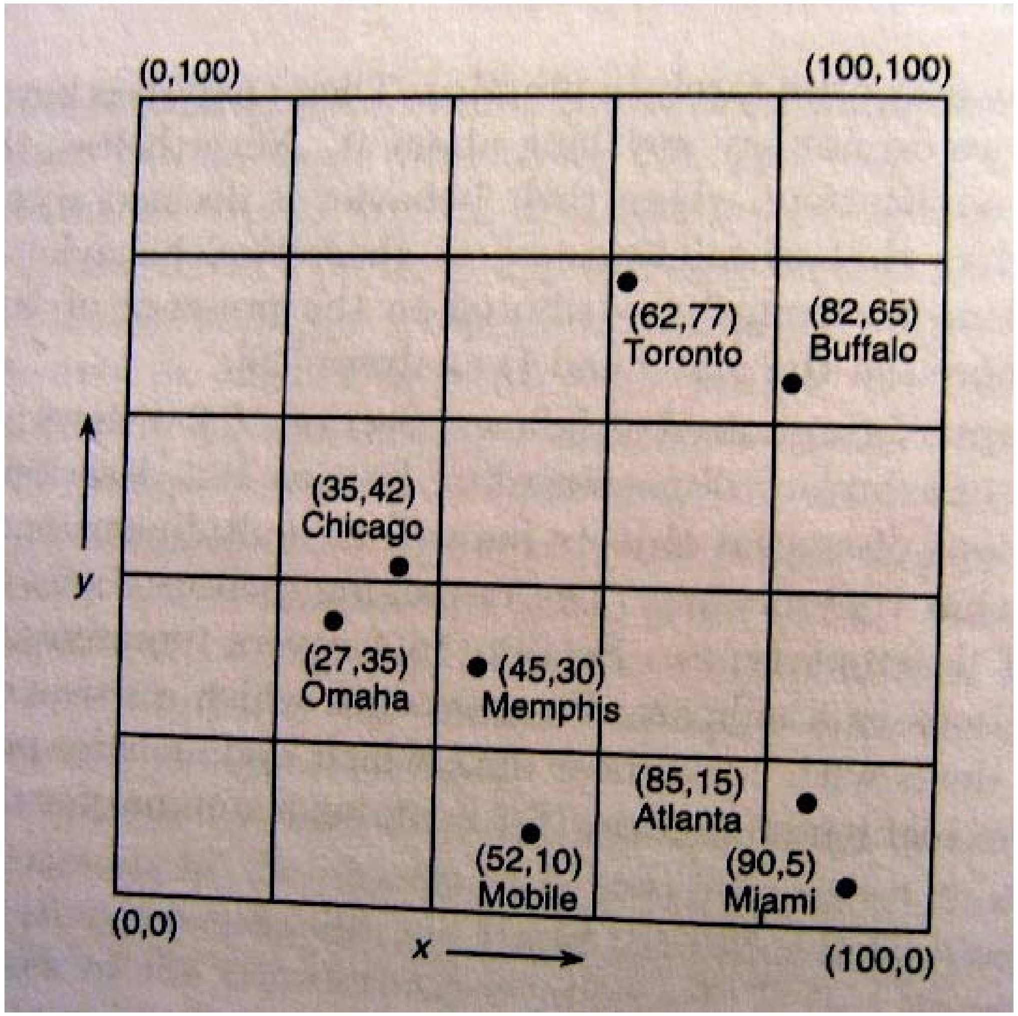


FIGURE 22.1: (a) A binary $2^3 \times 2^3$ image (b) Its corresponding region quadtree. The location codes for the homogeneous codes are indicated in brackets.

Point data: uniform structure



Point data: quadtree

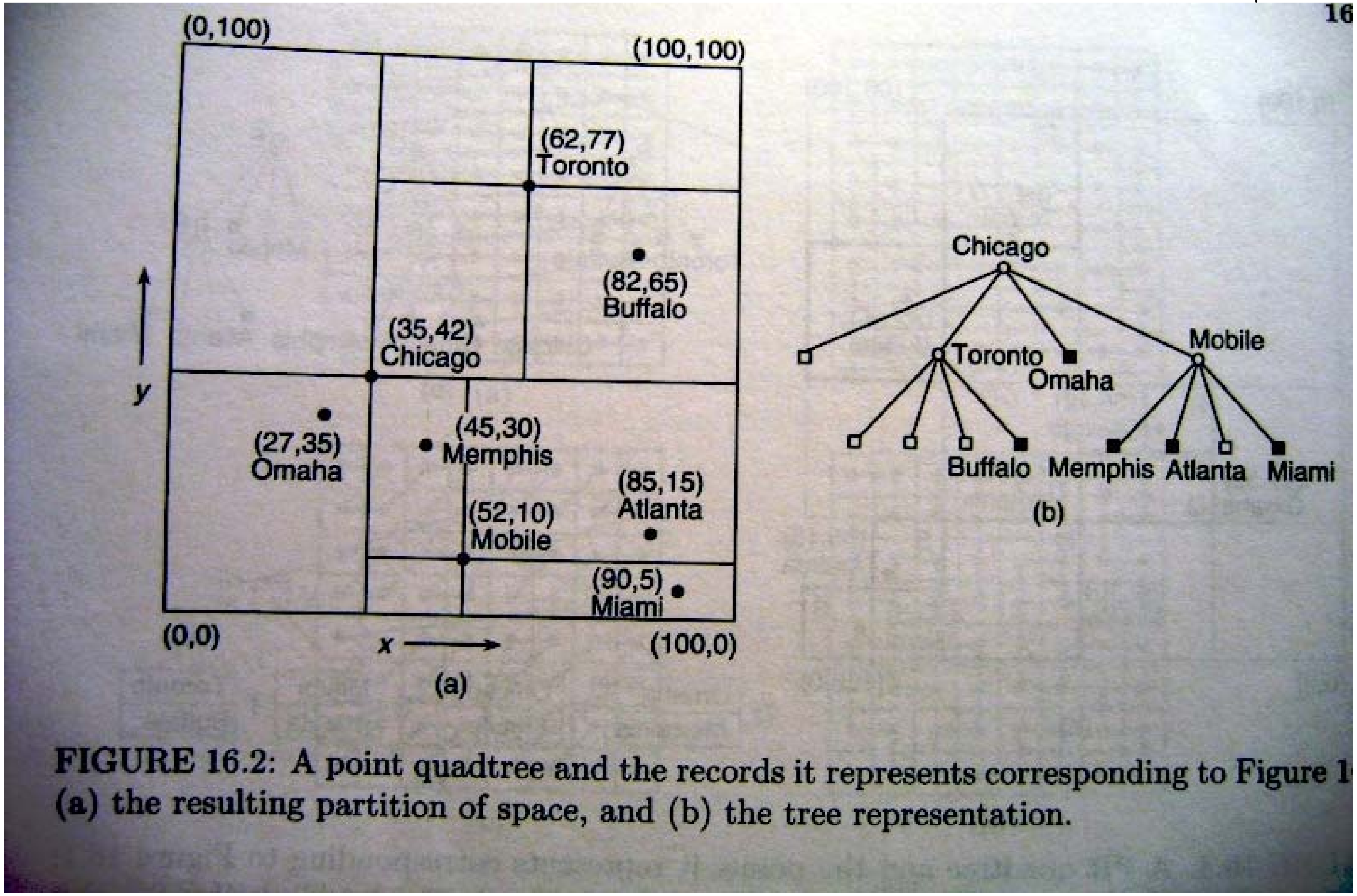


FIGURE 16.2: A point quadtree and the records it represents corresponding to Figure 1
(a) the resulting partition of space, and (b) the tree representation.

Point data: PR quadtree

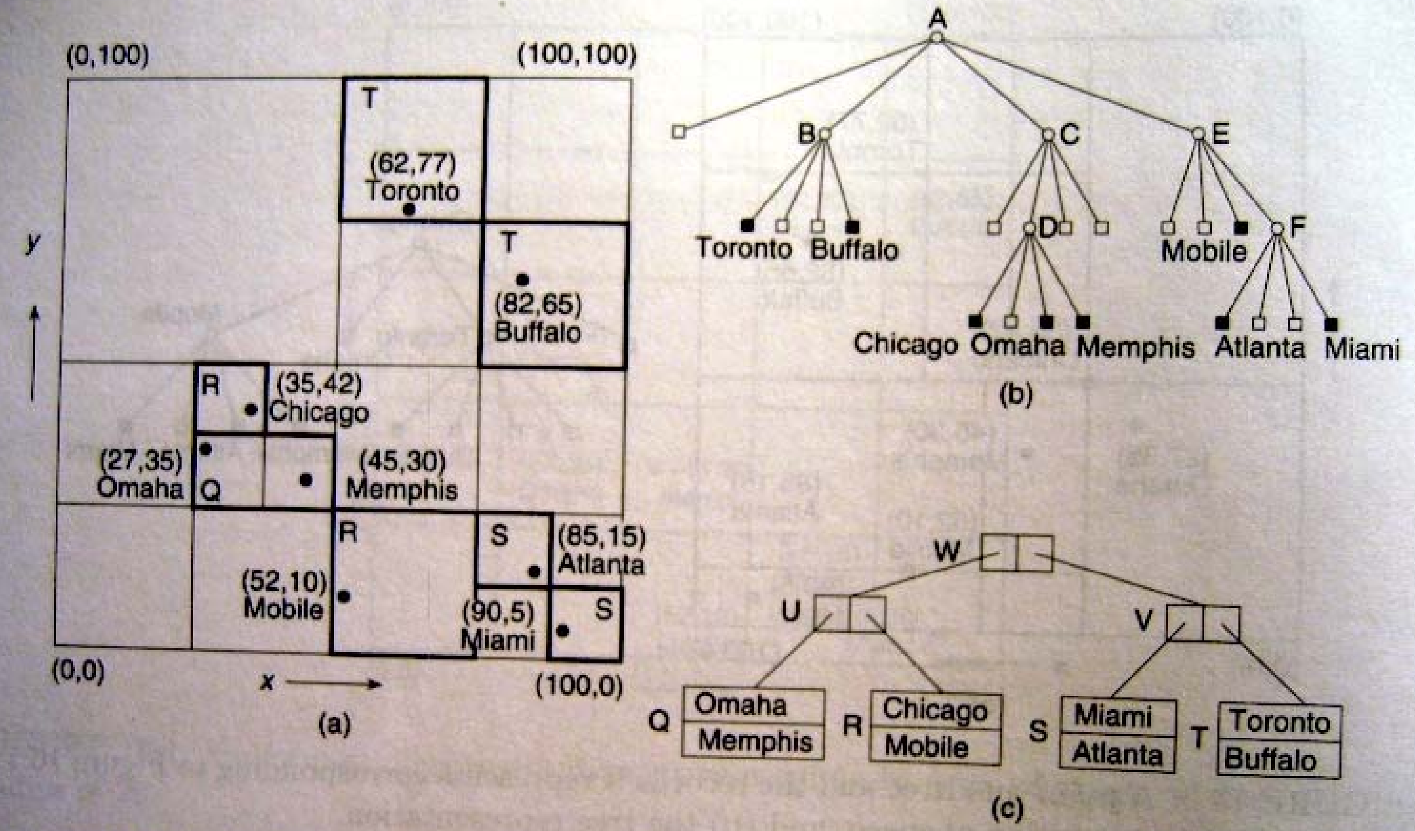


FIGURE 16.3: A PR quadtree and the points it represents corresponding to Figure 16.1: (a) the resulting partition of space, (b) the tree representation, and (c) one possible B+-tree for the nonempty leaf grid cells where each node has a minimum of 2 and a maximum of 3 entries. The nonempty grid cells in (a) have been labeled with the name of the B+-tree leaf node in which they are a

Line data: quadtre

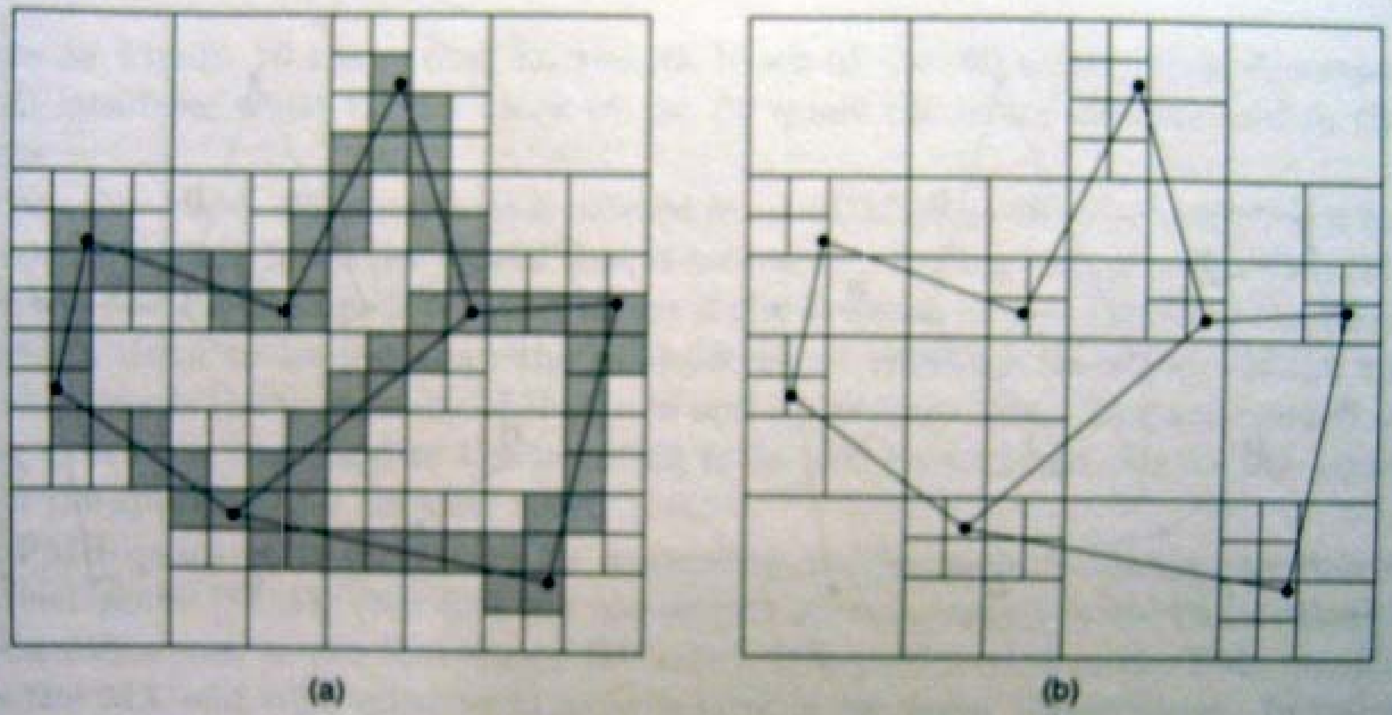


FIGURE 16.12: (a) MX quadtree and (b) edge quadtree for the collection of line segments of Figure 16.5.

Rectangles: quadtree

Multidimensional Spatial Data Structures 16-19

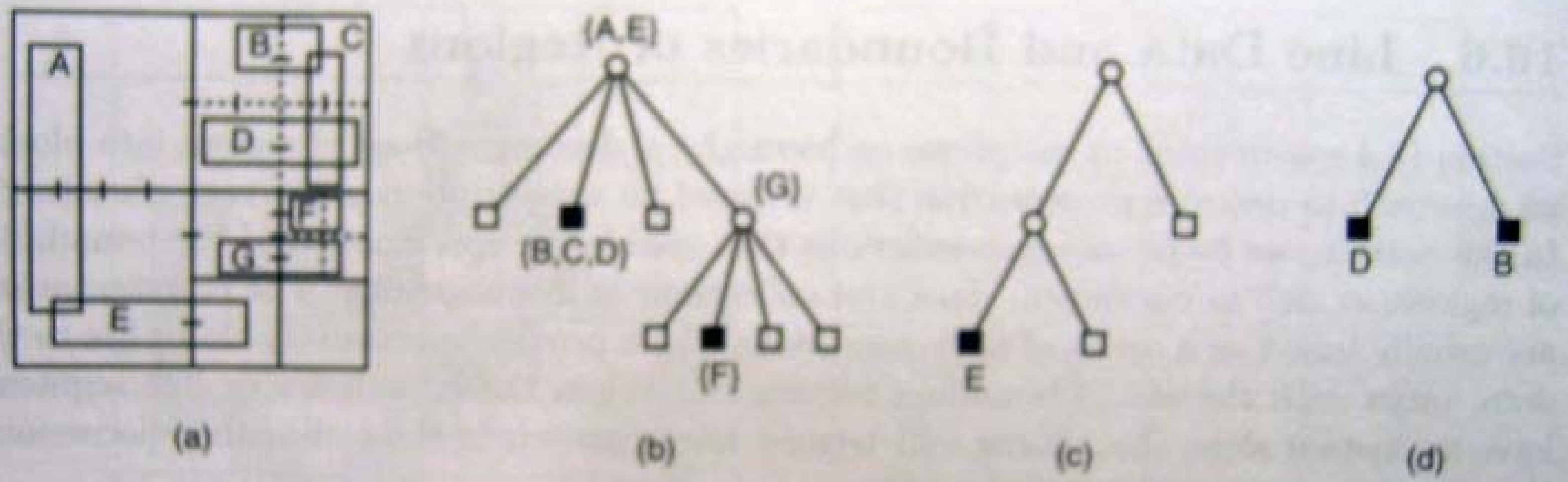
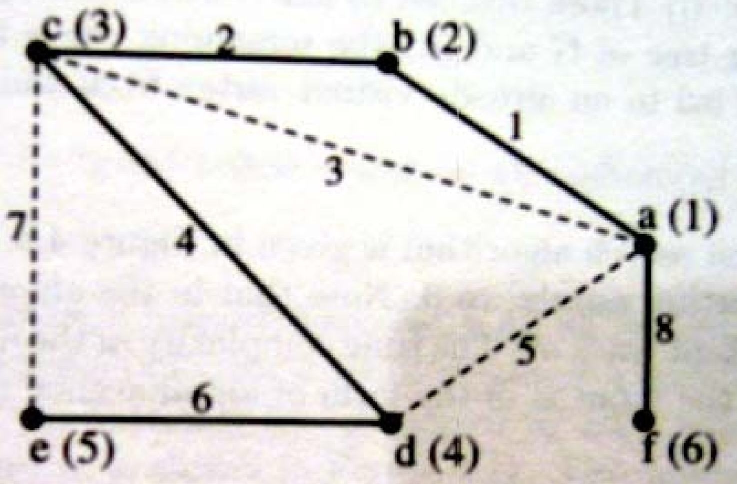


FIGURE 16.11: (a) Collection of rectangles and the block decomposition induced by the MX-CIF quadtree; (b) the tree representation of (a); the binary trees for the y axes passing through the root of the tree in (b), and (d) the NE son of the root of the tree in (b).

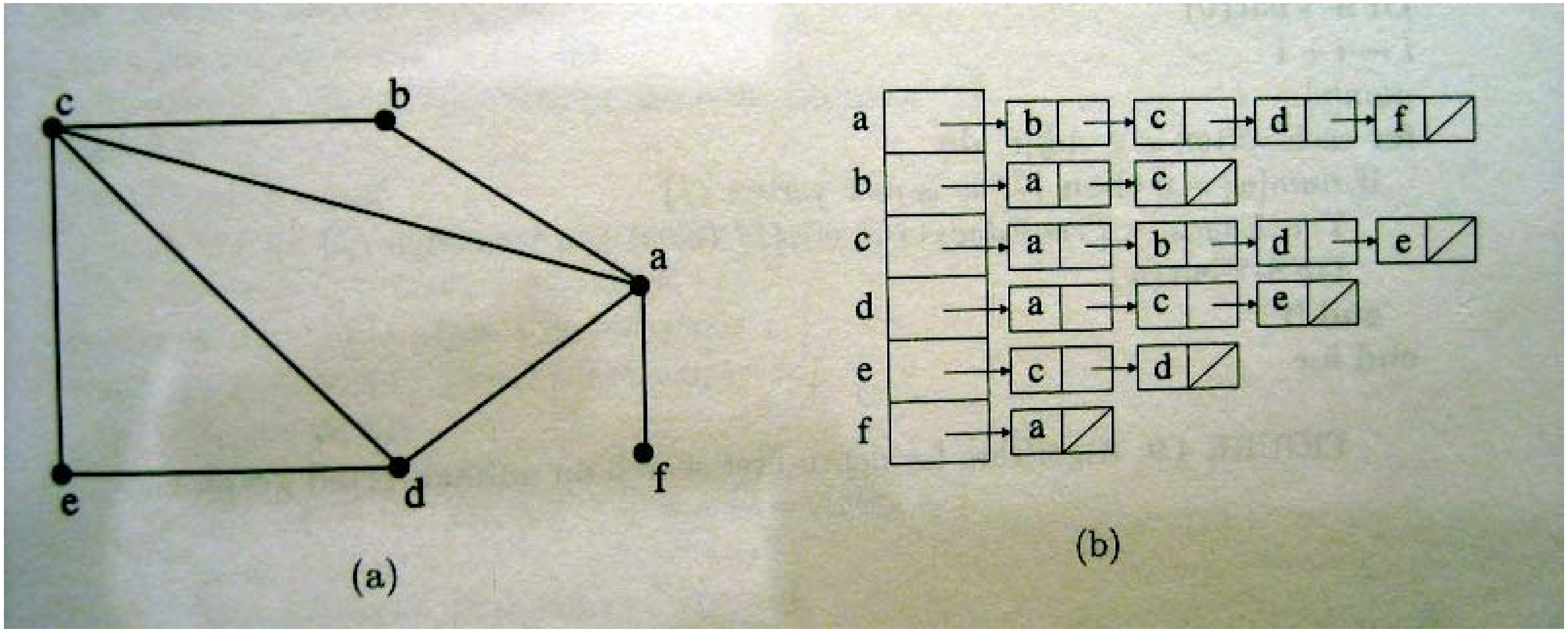
Graph: adjacency list



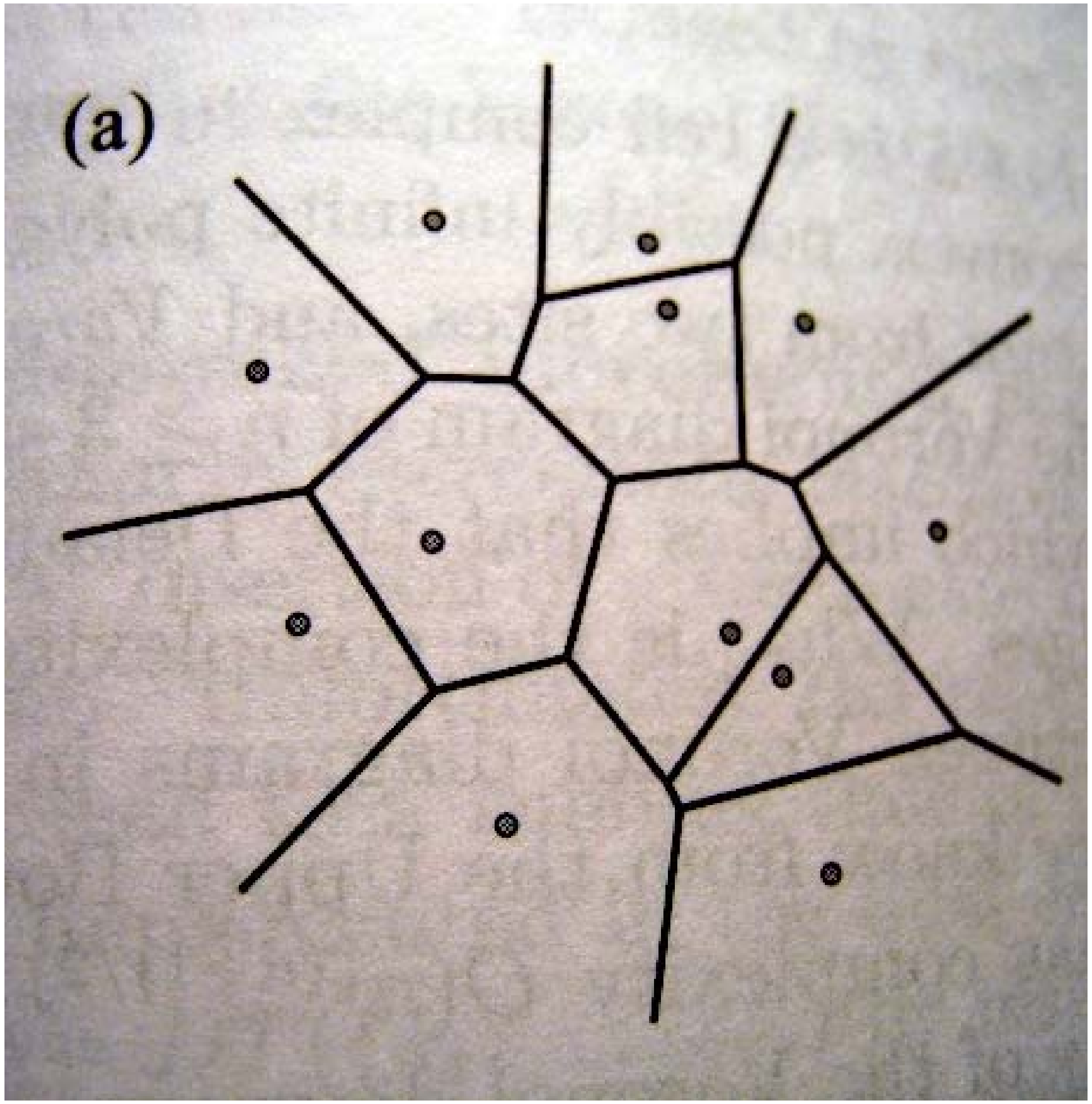
(c)

FIGURE 4.8: A graph (a); its adjacency lists (b); and its depth-first traversal (c
numbers are the order in which vertices were visited and edges traversed. Edges
traversal led to new vertices are shown with thick lines, and edges that led to vertic
were already visited are shown with dashed lines.

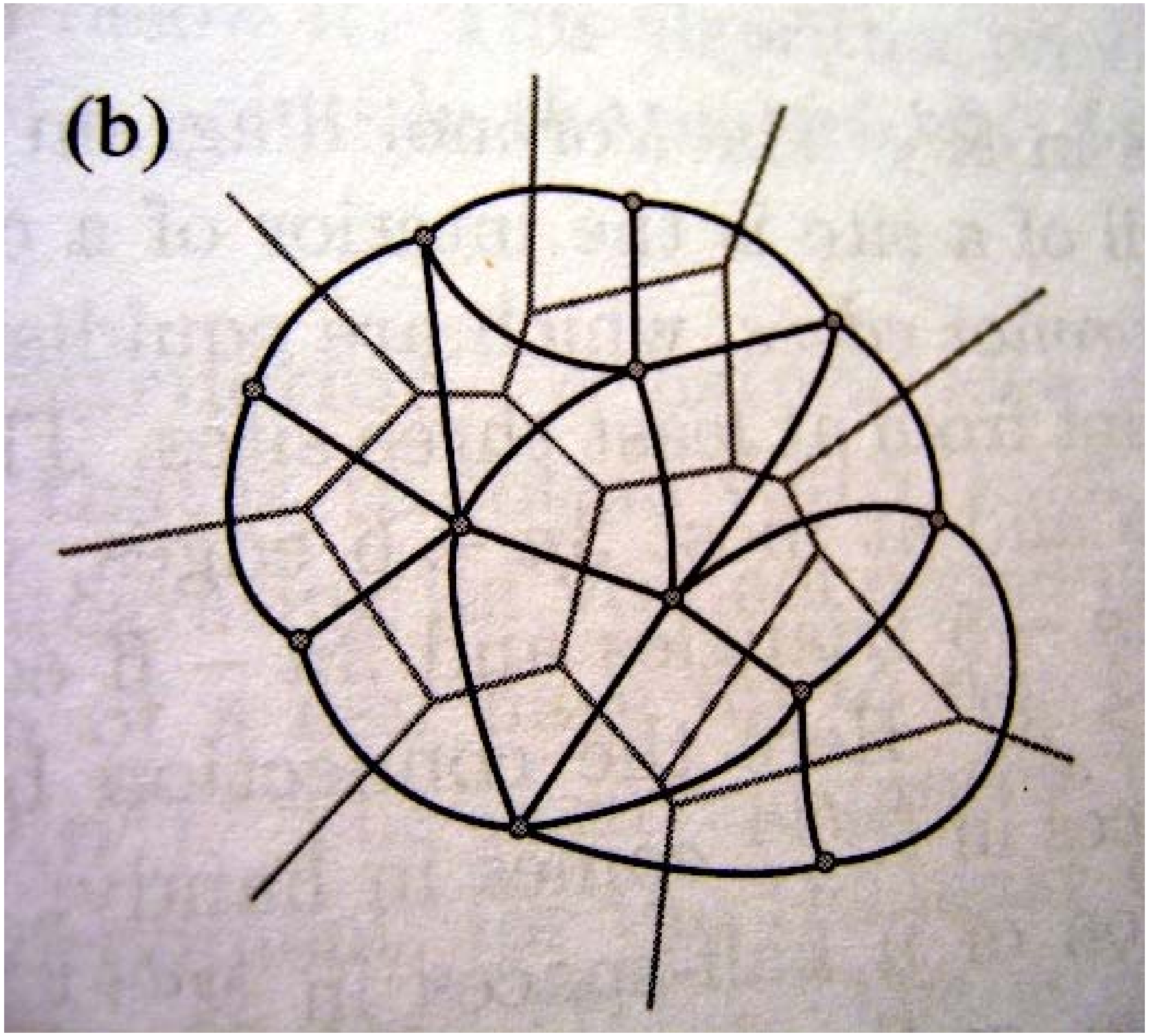
Graph: adjacency list



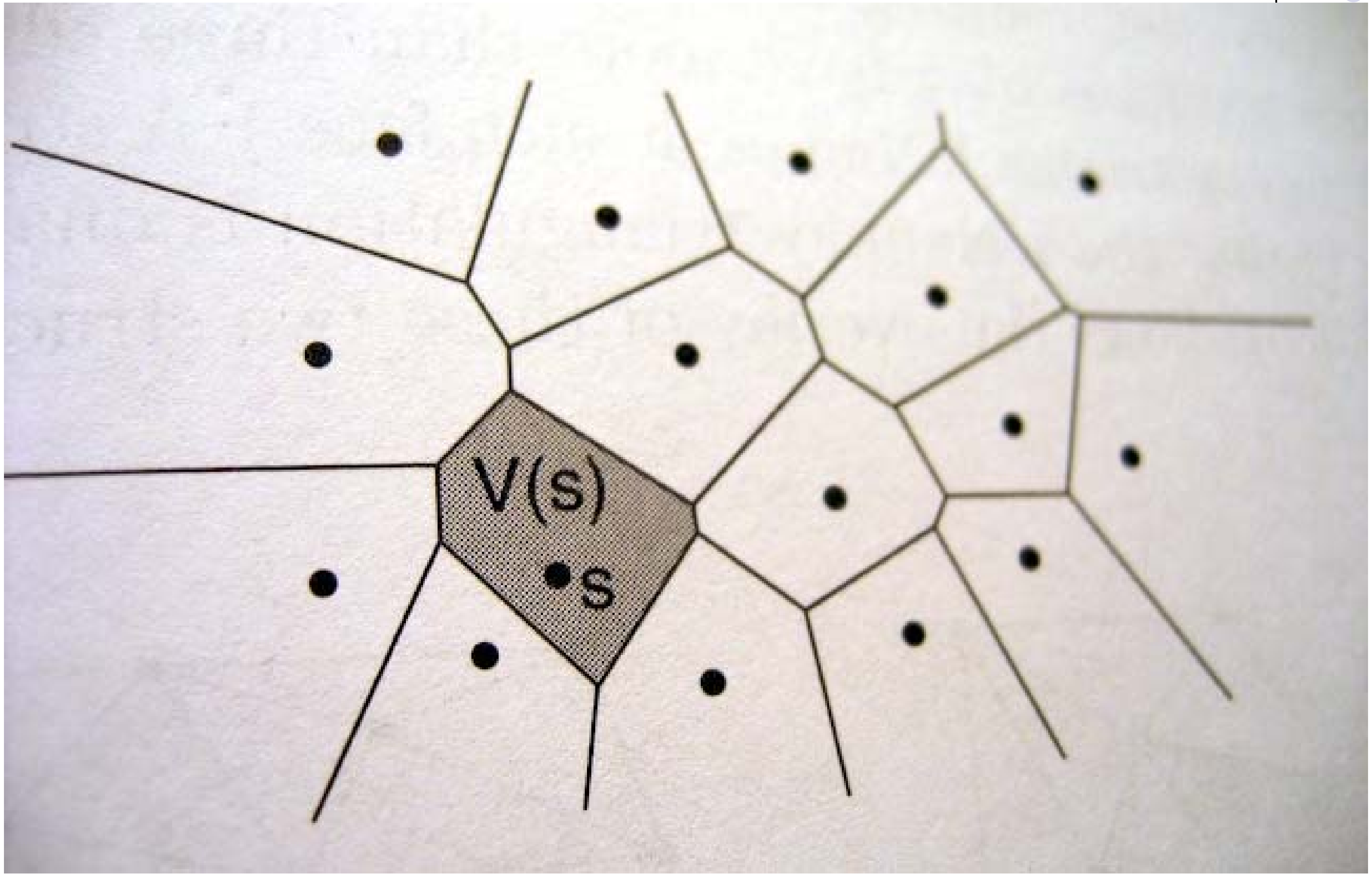
Voronoi Mesh



Delaunay Triangulation: Dual of Voronoi Mesh



Cell V(s)
points closer to s than to any other point



Winged-Edge Structure

Encoding of arbitrary polygons

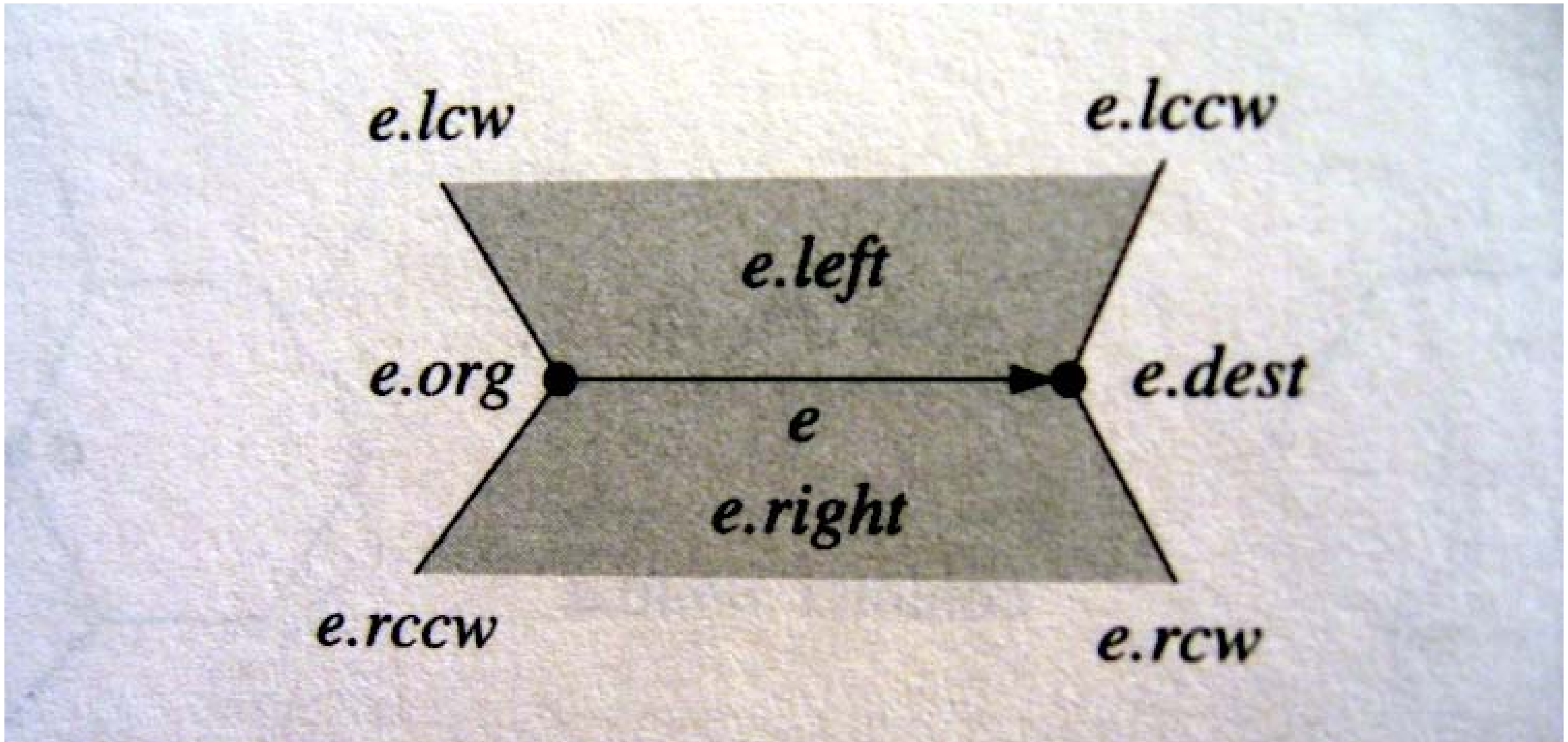


FIGURE 17.8: Winged-edge data structure.

Halfedge structure
Encoding of arbitrary polygons

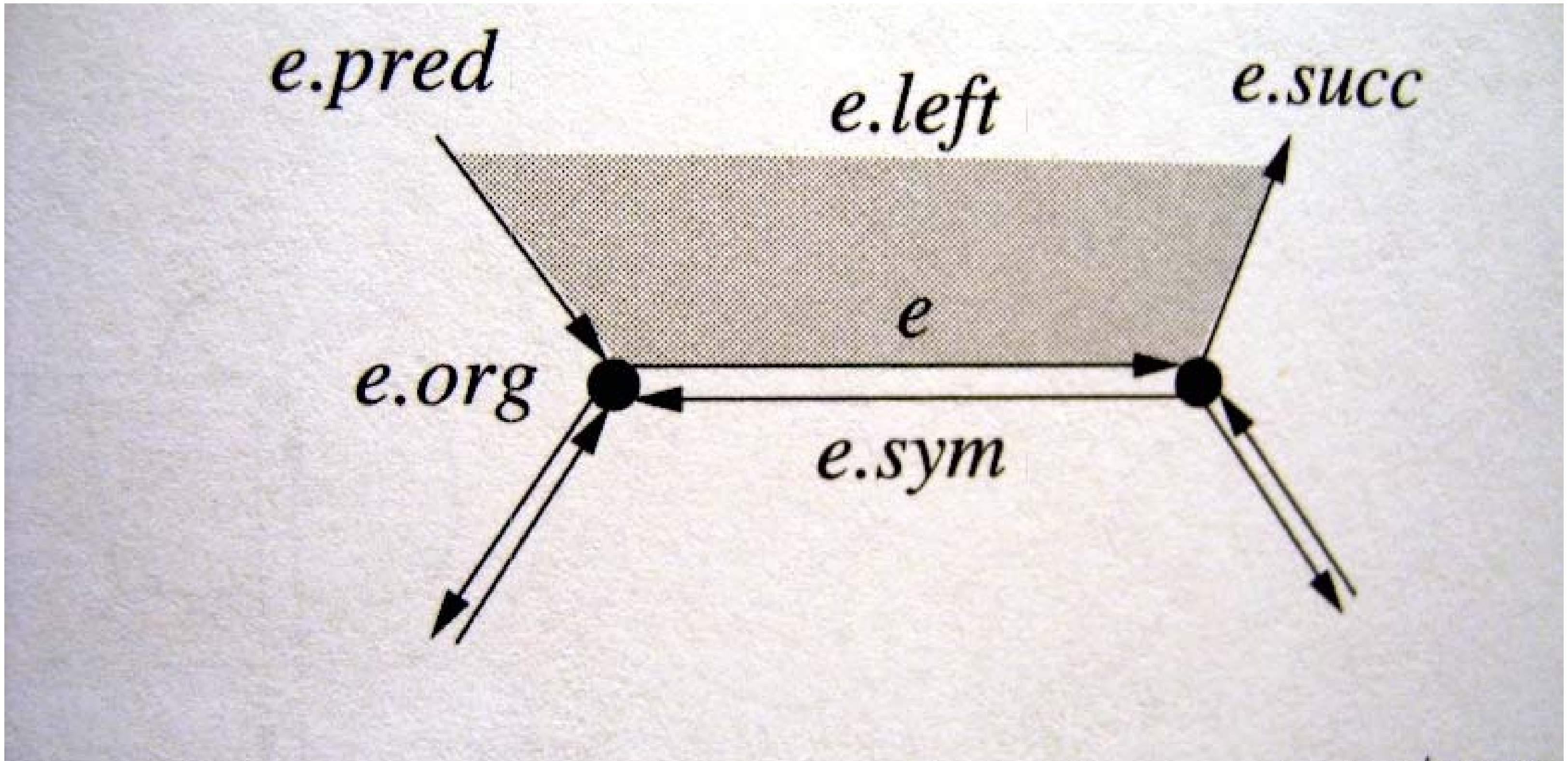


FIGURE 17.9: Halfedge data structure.

Derivatives on triangular mesh

Integral Theorem: exact formula

$$\iint (\nabla f)\, dS = \oint f\ \mathrm{d}\mathbf{l} \times \mathbf{z}$$

Approximate formula for derivative at vertex *j*

$$(\nabla f)_j = \frac{\sum_{i(j)} 0.5\left(f_i + f_{i+1}\right)\left(\mathbf{r}_{i+1} - \mathbf{r}_i\right) \times \mathbf{z}}{S}$$