CISC 220
Final Review

Prof. Christopher Rasmussen

http://nameless.cis.udel.edu/class_wiki/index.php/CISC220_F2014

University of Delaware
Fall, 2014

Midterm Details

- Monday, December 8, 3:30-5:30 pm for **both** section 010 and 011
  - Smith 130
- Covers all lectures from Tuesday, Oct. 21 through Tuesday, Dec. 2
- Closed book, no notes, no calculators, cell phones, etc.
- Worth 20% of your grade (same as midterm)
- Question types (see posted 2010 final & midterm)
  - Data structure ADTs, application examples
  - Definitions, compare/contrast approaches, etc.
  - Write C++ code for some small functions
    - No questions about file I/O or inheritance
  - Carry out operations and show steps (graph search, probing in hash tables, etc.)

Topics Covered

- Priority queues, heaps (Drozdek, 4.3, 4.6, 6.9)
- Disjoint sets / union-find (Drozdek, 8.4.1; UW slides (first 5 pages))
- Compression (Drozdek, 11-11.2 (skip 11.2.1))
- Hashing (Drozdek, 10-10.2.2, 10.3; cryptographic hashes page)
- Graphs (Drozdek, 8-8.1, 8.5 (Kruskal's only), 8.2, 8.3 (stop after Dijkstra's))
- Sorting (Drozdek, 9-9.1.2, 9.3.2-9.3.4)

Priority Queues

- In some circumstances, the normal FIFO operation of a queue may need to be overridden
- This may occur due to priorities that are associated the elements of the queue that affect the order of processing
- In cases such as these, a ***priority queue*** is used, where the elements are removed based on priority and position
- The difficulty in implementing such a structure is trying to accommodate the priorities while still maintaining efficient enqueuing and dequeuing
- Elements typically arrive randomly, so their order typically reflects no specific priority

Priority Queues (continued)

- The situation is further complicated because there are numerous priority scenarios that could be applied
- There are several ways to represent priority queues
- With linked lists, one arrangement maintains the items in entry order, and another inserts them based on priority
- Another variation, attributed to Blackstone (Blackstone *et. al.* 1981) uses a short ordered list and larger unordered list
- Items are placed in the shorter list based on a calculated threshold priority
- On some occasions, the shorter list could be emptied, requiring the threshold to be dynamically recalculated

Heaps

- A ***heap*** is a special type of binary tree with the following properties:
  - The value of each node is greater than or equal to the values stored in its children
  - The tree is perfectly balanced, and the leaves in the last level are leftmost in the tree
- This actually defines a ***max heap***; if "greater than" is replaced by "less than" in the first property, we have a ***min heap***
- Thus the root of a max heap is the largest element, and the root of a min heap the smallest
- If each nonleaf of a tree exhibits the first property, the tree exhibits the ***heap property***

Heaps (continued)

- Figure 6.51 exhibits some examples; those in Figure 6.51a are heaps, while those in Figure 6.51b violate the first property and those in Figure 6.51c violate the second

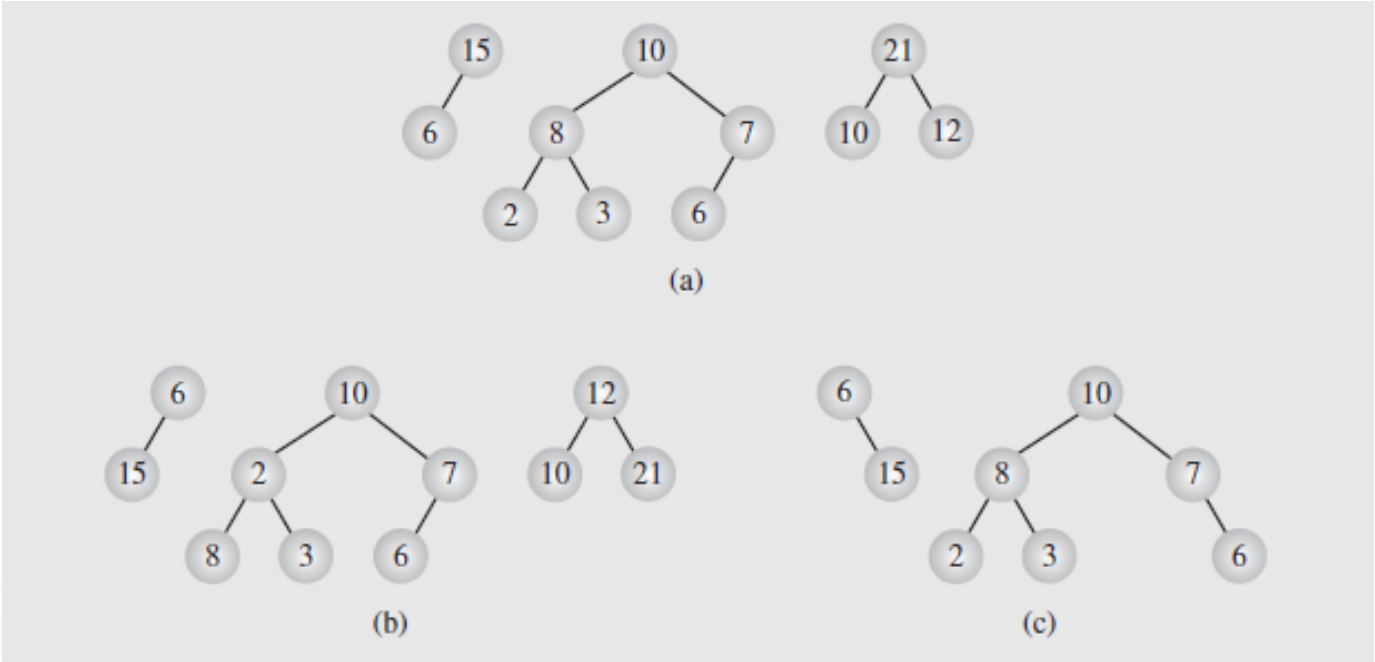


Fig. 6.51 Examples of (a) heaps and (b–c) nonheaps

Heaps (continued)

- Heaps can be implemented as arrays
- As an example, consider the array `data=[2 8 6 1 10 15 3 12 11]` represented as a nonheap tree in Figure 6.52

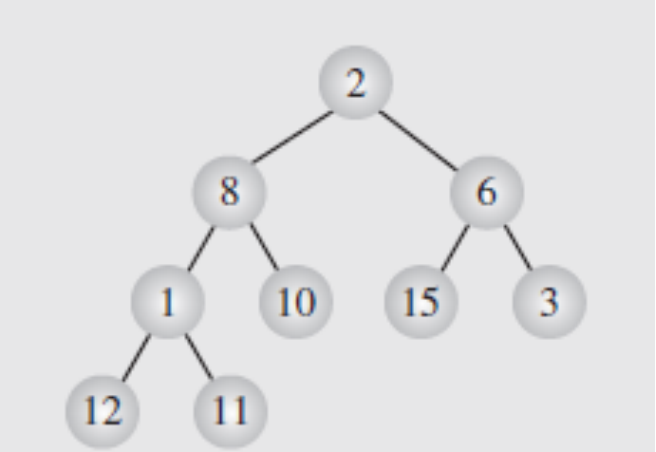


Fig. 6.52 The array [2 8 6 1 10 15 3 12 11] seen as a tree

- The arrangement of the elements reflects the tree from top-to-bottom and left-to-right

Heaps (continued)

- We can define a heap as an array `heap` of length n where

  $$\texttt{heap[i]} \geq \texttt{heap[2i + 1]}, \text{ for } 0 \leq i < (n - 1)/2$$

  and

  $$\texttt{heap[i]} \geq \texttt{heap[2i + 2]}, \text{ for } 0 \leq i < (n - 2)/2$$

- Elements in a heap are not ordered; we only know the root is the largest and the descendants are less than or equal to it
- The relationship between siblings or between elements in adjacent subtrees is undetermined
- All we are aware of is that there is a linear relationship along the lines of descent, but lateral lines are ignored

Heaps (continued)

- This is why, although all the trees in Figure 6.53 are heaps, Figure 6.53b is ordered the best

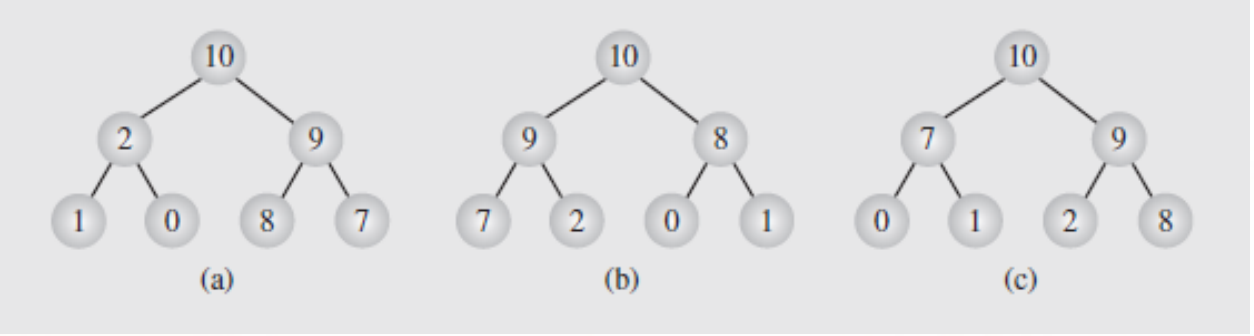


Fig. 6.53 Different heaps constructed with the same elements

Heaps (continued)

- Heaps as Priority Queues
  - Heaps are ideal for implementing priority queues
  - We saw linked lists used to do this in section 4.3, but for large amounts of data, they can become inefficient
  - Because heaps are perfectly balanced trees, the inherent efficiency of searching such structures makes them more useful
  - We will need a couple of routines to enqueue and dequeue elements on the priority queue, though
  - To enqueue, the node is added at the end of the heap as the last leaf
  - If the heap needs to be restructured to preserve the heap property, it can be done by moving the node from last leaf towards the root

Heaps (continued)

- Heaps as Priority Queues (continued)
  - The enqueuing algorithm is as follows:

```
heapEnqueue(el)
  put el at the end of the heap;
    while el is not in the root and el > parent(el)
      swap el with its parent;
```

  - This is illustrated in Figure 6.54a, where the node 15 is added to the heap
  - Because this destroys the heap property, 15 is moved up the tree until it is either the root or finds a parent greater than or equal to 15
  - This is reflected in Figure 6.54b-d

Heaps (continued)

- Heaps as Priority Queues (continued)

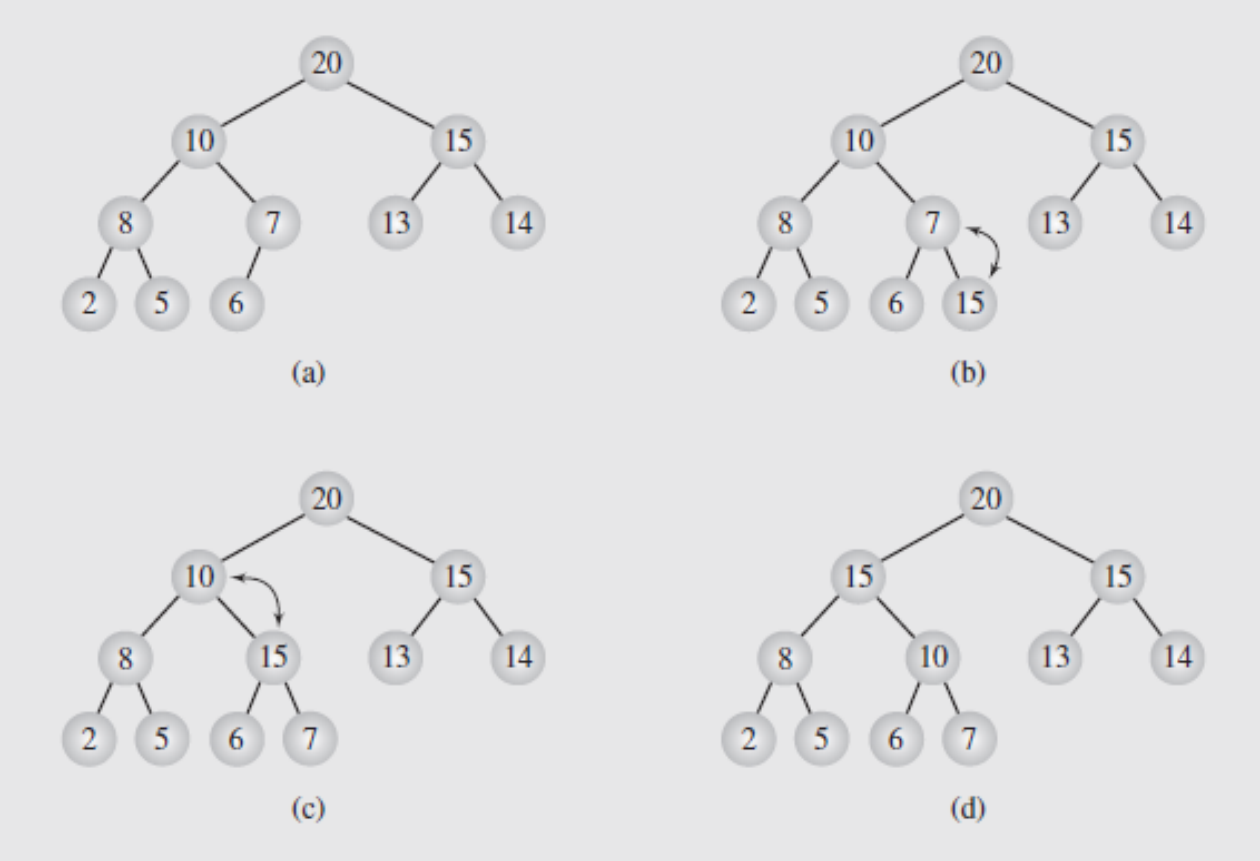


Fig. 6.54 Enqueuing an element to a heap

Heaps (continued)

- Heaps as Priority Queues (continued)
  - Dequeuing an element from a heap simply removes the root (since it is the largest value) and replacing it by the last leaf
  - Since this will most likely violate the heap property, the node is moved down the tree to the appropriate location
  - The algorithm for this looks like:

```
heapDequeue()
    extract the element from the root;
    put the element from the last leaf in its place;
    remove the last leaf;
// both subtrees of the root are heaps
    p = the root;
    while p is not a leaf and p < any of its children
        swap p with the larger child;
```

Heaps (continued)

- Heaps as Priority Queues (continued)
  - This is shown in Figure 6.55; 20 is dequeued and 6 put in its place
  - This is then swapped with 15 (the larger child) and again with 14

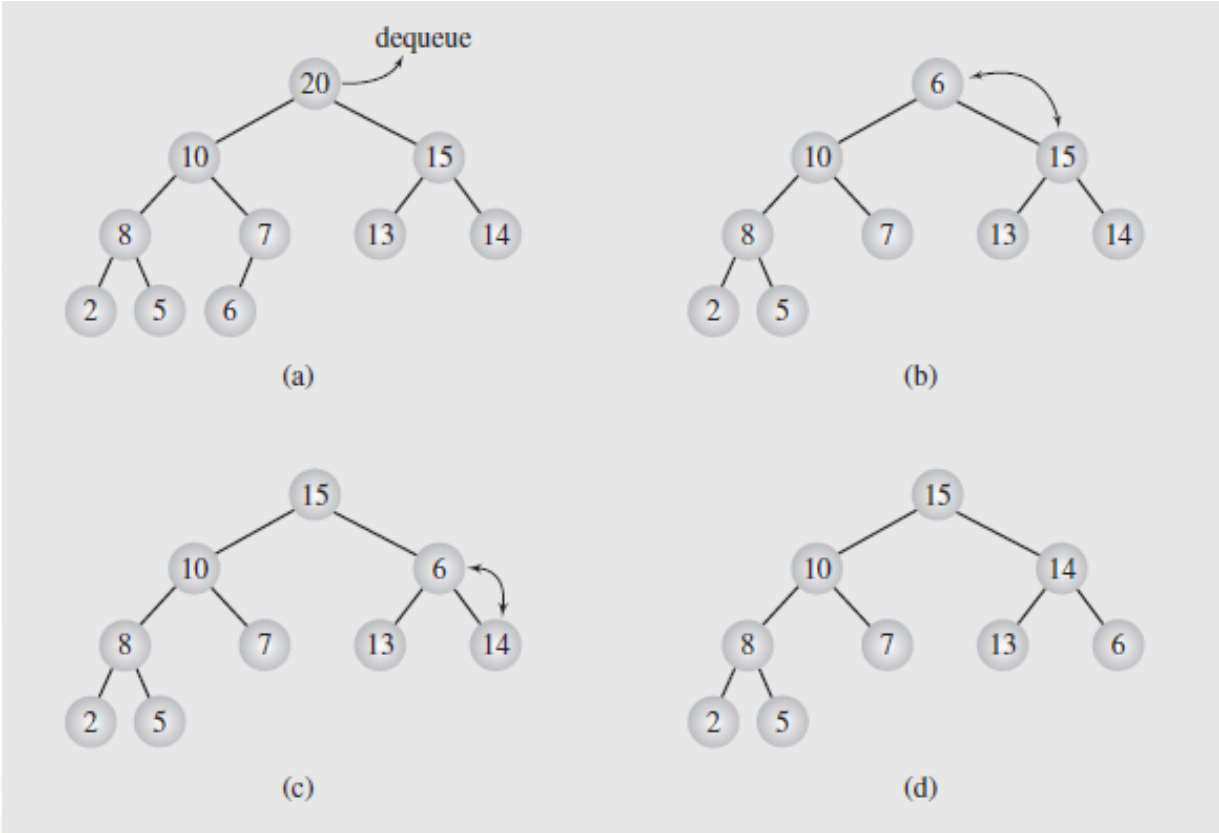


Fig. 6.55 Dequeuing an element from a heap

Heaps (continued)

- Heaps as Priority Queues (continued)
  - The last three lines of this dequeuing algorithm can be used as a stand-alone routine to restore the heap property if it is violated by the root by moving it down the tree; a coded form is shown below:

```
template<class T>
void moveDown (T data[], int first, int last) {
    int largest = 2*first + 1;
    while (largest <= last) {
        if (largest < last && // first has two children (at 2*first+1 and
            data[largest] < data[largest+1]) // 2*first+2) and the second
             largest++;                      // is larger than the first;

        if (data[first] < data[largest]) {     // if necessary,
             swap(data[first],data[largest]);  // swap child and parent,
             first = largest;                  // and move down;
             largest = 2*first+1;
        }
        else largest = last+1;  // to exit the loop: the heap property
    }                           // isn't violated by data[first];
}
```

Fig. 6.56 Implementation of an algorithm to move the root element down a tree

Heaps (continued)

- Organizing Arrays as Heaps
  - As we've seen, heaps can be implemented as arrays, but not all arrays are heaps
  - In some circumstances, though, we need to organize the contents of an array as a heap, such as in the heap sort
  - One of the simpler ways to accomplish this is attributed to John Williams; we start with an empty heap and sequentially add elements
  - This is a top-down technique that extends the heap by enqueuing new elements in the heap
  - This process is described on page 273 and illustrated in Figure 6.57

CSE 332 Data Abstractions: Disjoint Set Union-Find and Minimum Spanning Trees

Kate Deibel

Summer 2012

Making Connections

You have a set of nodes (numbered 1-9) on a network. You are given a sequence of pairwise connections between them:

3-5 4-2 1-6 5-7 4-8 3-7

Q: Are nodes 2 and 4 connected? Indirectly?

Q: How about nodes 3 and 8?

Q: Are any of the paired connections redundant due to indirect connections?

Q: How many sub-networks do you have?

Making Connections

Answering these questions is much easier if we create disjoint sets of nodes that are connected:

Start:	{1} {2} {3} {4} {5} {6} {7} {8} {9}
3-5	{1} {2} {3, 5} {4} {6} {7} {8} {9}
4-2	{1} {2, 4} {3, 5} {6} {7} {8} {9}
1-6	{1, 6} {2, 4} {3, 5} {7} {8} {9}
5-7	{1, 6} {2, 4} {3, 5, 7} {8} {9}
4-8	{1, 6} {2, 4, 8} {3, 5, 7} {9}
3-7	*no change*

Making Connections

Let's ask the questions again.

3-5 4-2 1-6 5-7 4-8 3-7

⇓

{1, 6} {2, 4, 8} {3, 5, 7} {9}

Q: Are nodes 2 and 4 connected? Indirectly?

Q: How about nodes 3 and 8?

Q: Are any of the paired connections redundant due to indirect connections?

Q: How many sub-networks do you have?

Disjoint Set Union-Find ADT

Separate elements into disjoint sets

- If set x ≠ y then x ∩ y = ∅ (i.e. no shared elements)

Each set has a name (usually an element in the set)

union(x,y): take the union of the sets x and y (x ∪ y)

- Given sets: {3,5,7} , {4,2,8}, {9}, {1,6}
- union(5,1) → {3,5,7,1,6}, {4,2,8}, {9},

find(x): return the name of the set containing x.

- Given sets: {3,5,7,1,6}, {4,2,8}, {9},
- find(1) returns 5
- find(4) returns 8

Disjoint Set Union-Find Performance

Believe it or not:

- We can do Union in constant time.
- We can get Find to be ***amortized*** constant time with worst case O(log n) for an individual Find operation

Let's see how…

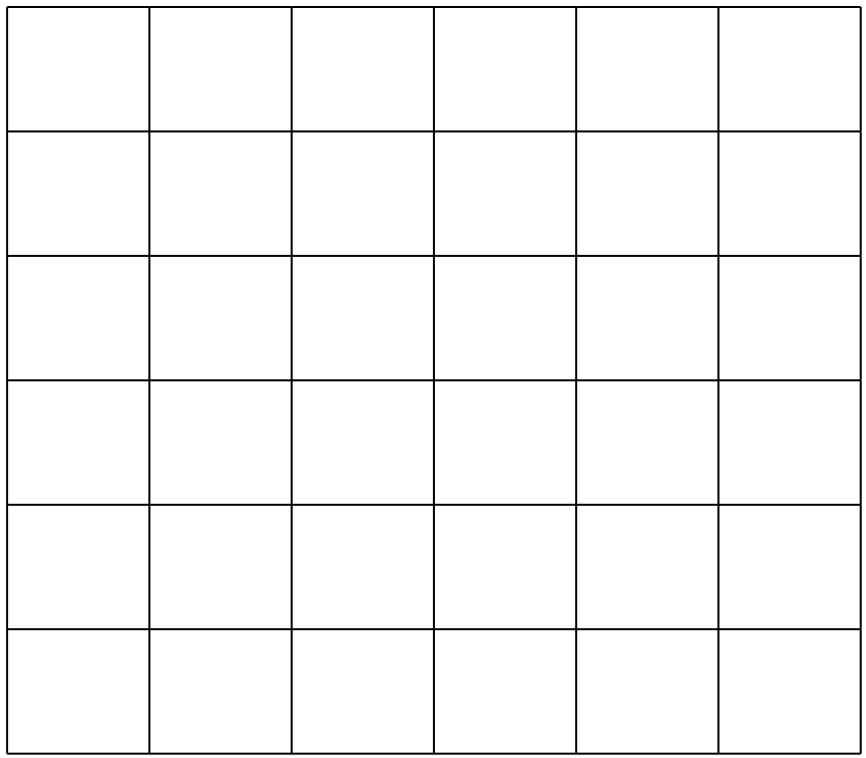
What Makes a Good Maze?

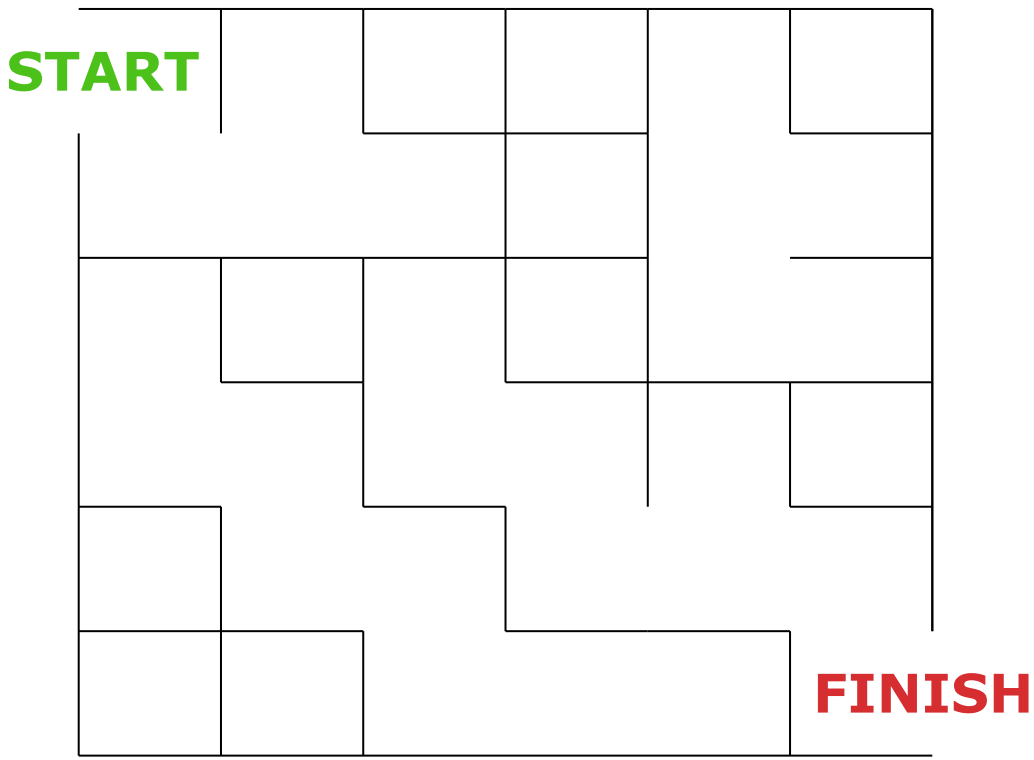
- We can get from any room to any other room (connected)
- There is just one simple path between any two rooms (no loops)
- The maze is not a simple pattern (random)

Making a Maze

A high-level algorithm for a random maze is easy:

- Start with a grid
- Pick Start and Finish
- Randomly erase edges





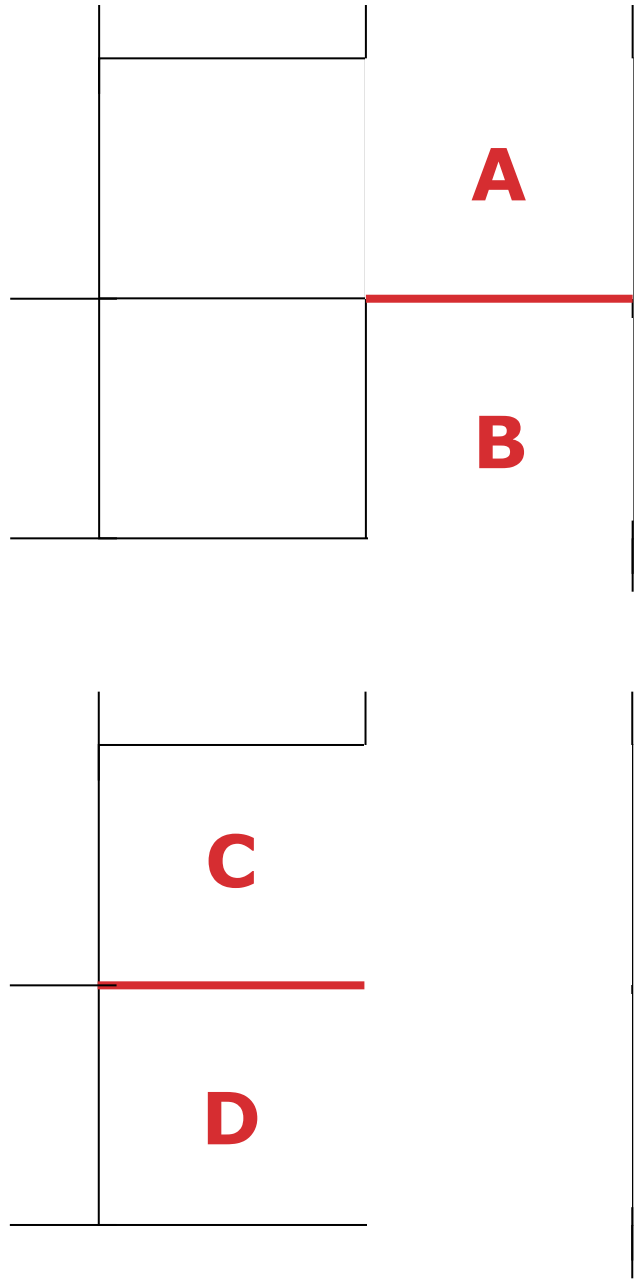
The Middle of the Algorithm

So far, we've knocked down several walls while others still remain.

Consider the walls between A and B and C and D

- Which walls can we knock down and maintain both our connectedness and our no cycles properties?

How do we do this efficiently?



Maze Algorithm: Number the Cells

Number each cell and treat as disjoint sets:

- S ={ {1}, {2}, {3}, {4},… {36} }

Create a set of all edges between cells:

- W ={ (1,2), (1,7), (2,8), (2,3), … } 60 walls total.

START

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

FINISH

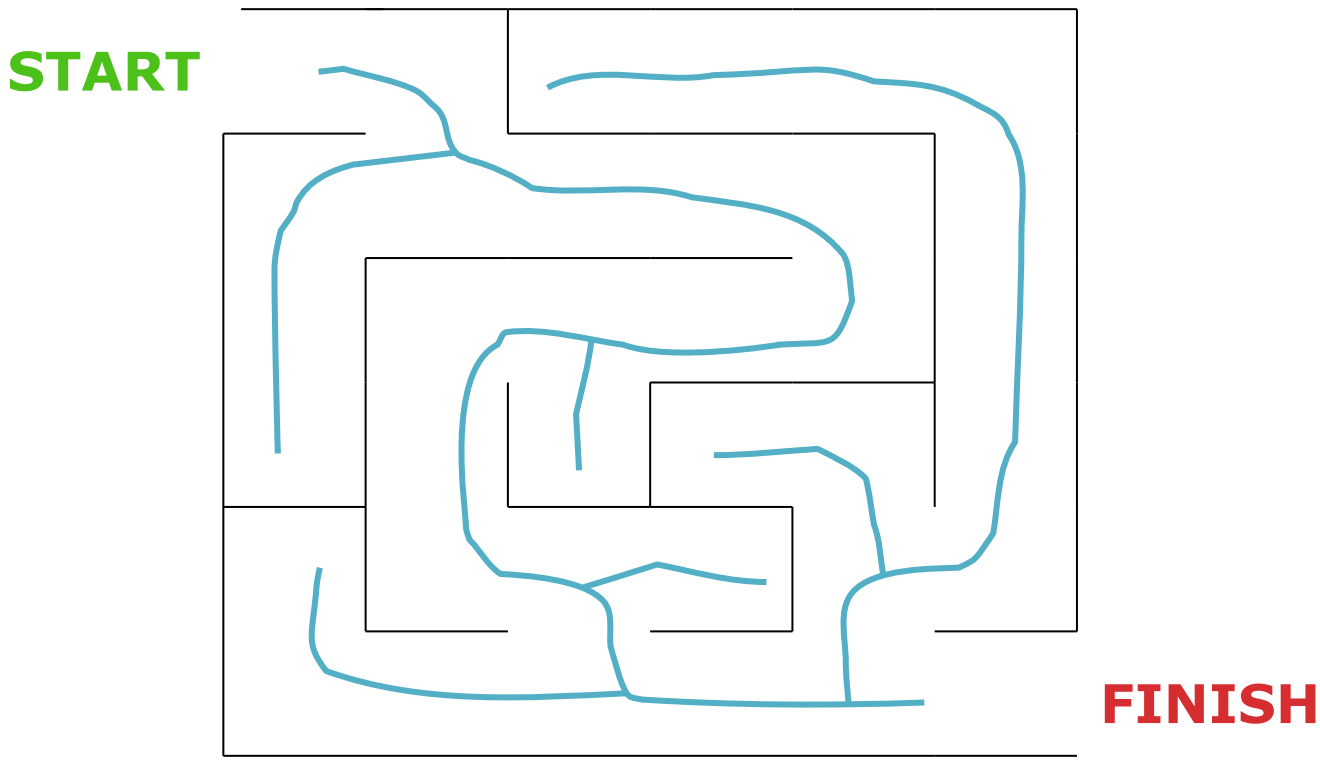
Maze Algorithm: Building with DSUF

Algorithm sketch:

- Choose a wall at random.
- Erase wall if the neighbors are in disjoint sets (this avoids creating cycles)
- Take union of those cell's sets
- Repeat until there is only one set
  - Every cell is thus reachable from every other cell

The Secret To Why This Works

Notice that a connected, acyclic maze is actually a Hidden Tree



This suggests how we should implement the Disjoint Set Union-Find ADT

Up Trees for Disjoint Set Union-Find

Up trees

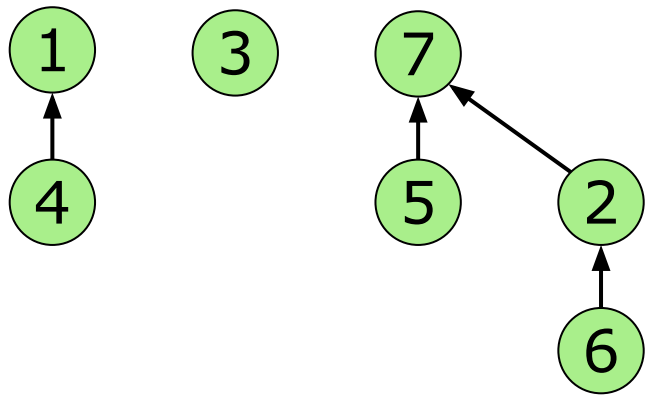
- Notes point to parent, not children
- Thus only one pointer per node

In a DSUF

- Each disjoint set is its own up tree
- The root of the tree is the name for the disjoint set

Initial State

After Unions



Find Operation

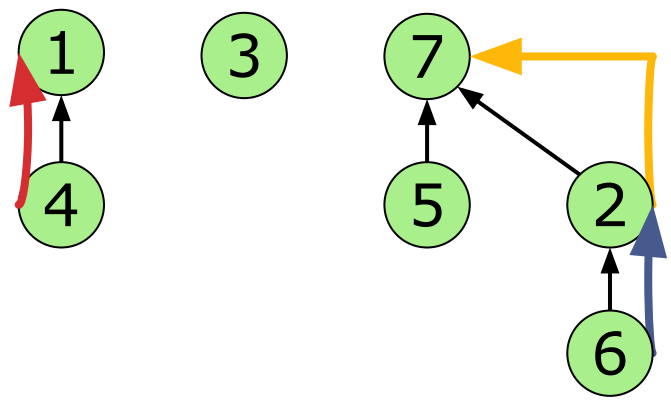
find(x): follow x to the root and return the root (the name of the disjoint set)

find(1) = 1

find(3) = 3

find(4) = 1

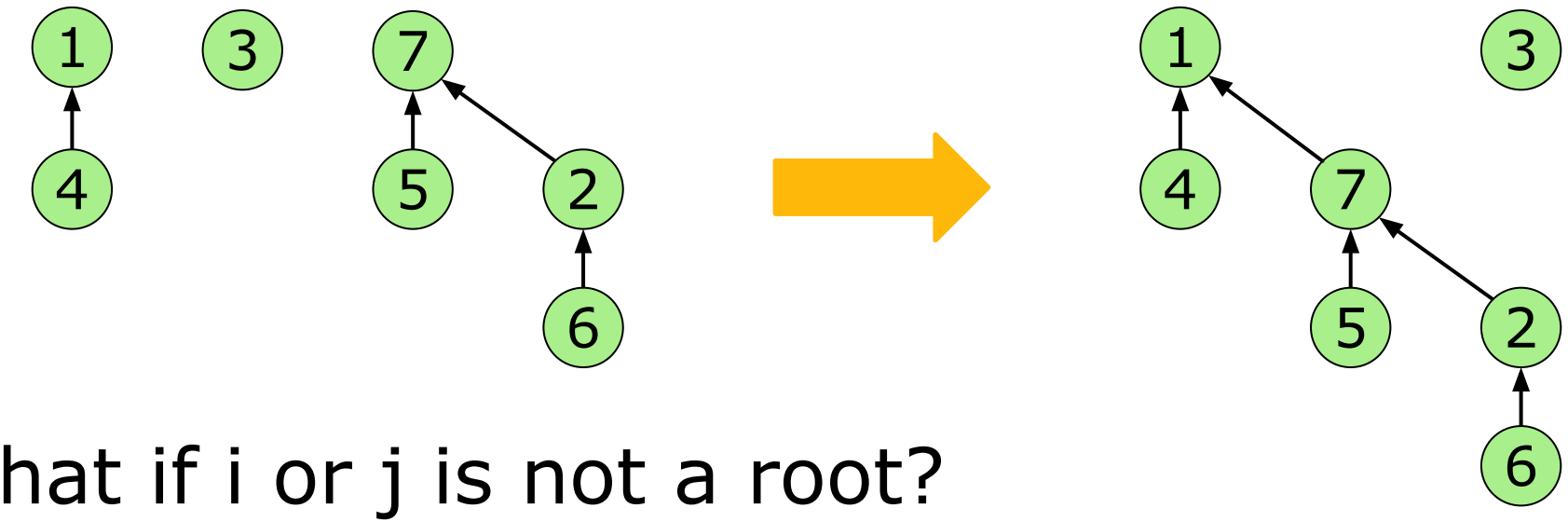
find(6) = 7



Find Operation

union(i,j): assuming i and j are roots, point root i to root j

union(1,7)



What if i or j is not a root?

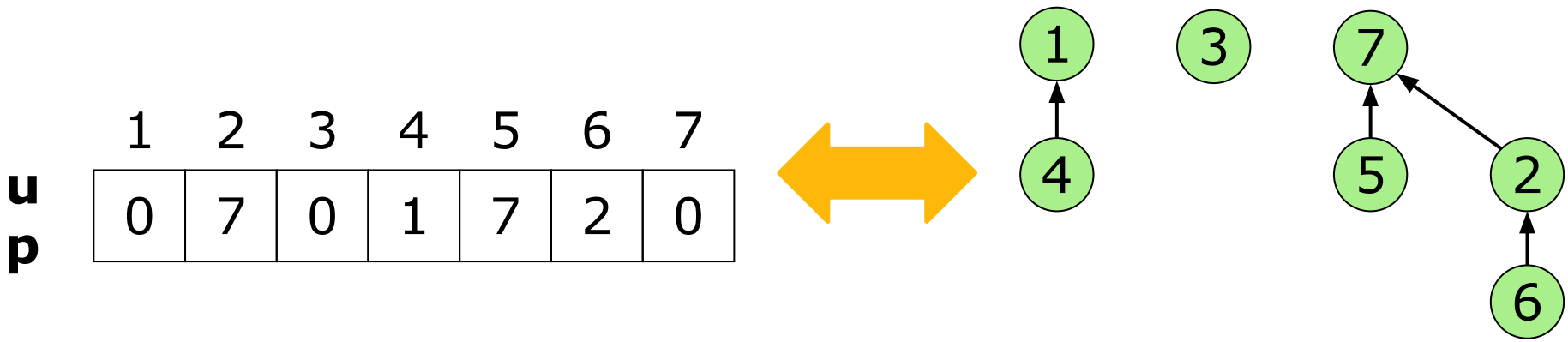
- Run a find on i and j first and use the returned values for the joining

Why do we join roots and not just the nodes?

Simple Implementation

Once again, it is better to implement a tree using an array than with node objects

- Leave up[0] empty (or # of disjoint sets)
- up[x] = i means node x's parent is node i
- up[x] = 0 means x is a root



Performance

Using array-based up trees, what is the cost for

- union(i,j)?
- find(x)?

union(i,j) is O(1) if i and j are roots

- Otherwise depends on cost of find

find(x) is O(n) in worst-case

- What does the worst-case look like?

Performance – Doing Better

The problem is that up trees get too tall

In order to make DSUF perform as we promised, we need to improve both our union and find algorithms:

- Weighted Union
- Path Compression

Only with BOTH of these will we get find to average-case O(log n) and amortized O(1)

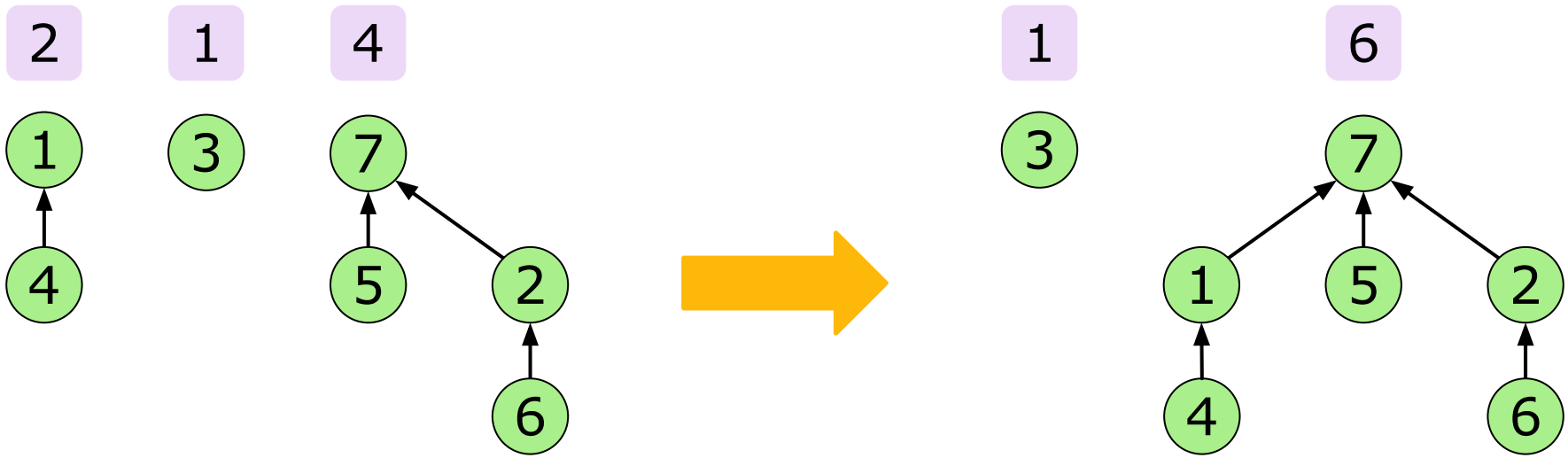
Weighted Union

Instead of arbitrarily joining two roots, always point the smaller tree to the root of the larger tree

- Each up tree has a weight (number of nodes)
- The idea is to limit the height of each up tree
- Trees with more nodes tend to be deeper

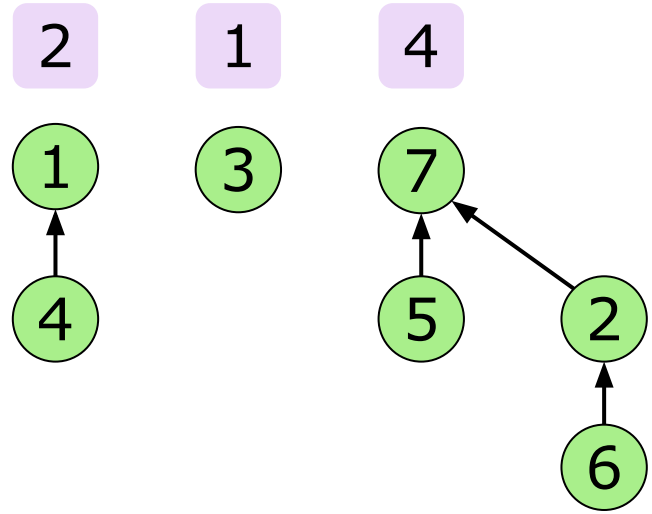
Union by rank or height are similar ideas but more complicated to implement

union(1,7)



Weighted Union Implementation

We can just use an additional array to store weights of the roots…

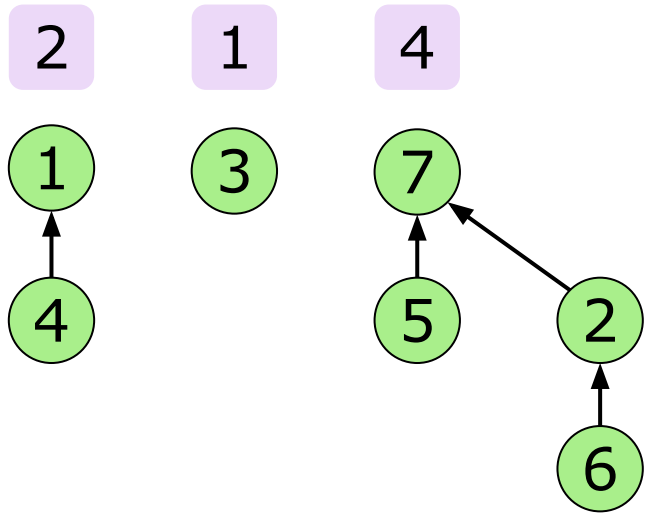


	1	2	3	4	5	6	7
u p	0	7	0	1	7	2	0
weight	2		1				4

Weighted Union Implementation

… or we use negative numbers to represent roots and their weights

But generally, saving O(n) space is not critical



	1	2	3	4	5	6	7
up	-2	7	-1	1	7	2	-4

Weighted Union Performance

Weighted union gives us guaranteed worst-case O(log n) for find

- The union rule prevents linear up trees
- Convince yourself that it will produce at worst a fairly balanced binary tree

However, we promised ourselves O(1) *amortized* time for find

- Weighted union does not give us enough
- Average-case is still O(log n)

Motivating Path Compression

Recall splay trees

- To speed up later finds, we moved searched for nodes to the root
- Also improved performance for finding other nodes
- Can we do something similar here?

Yes, but we cannot move the node to the root

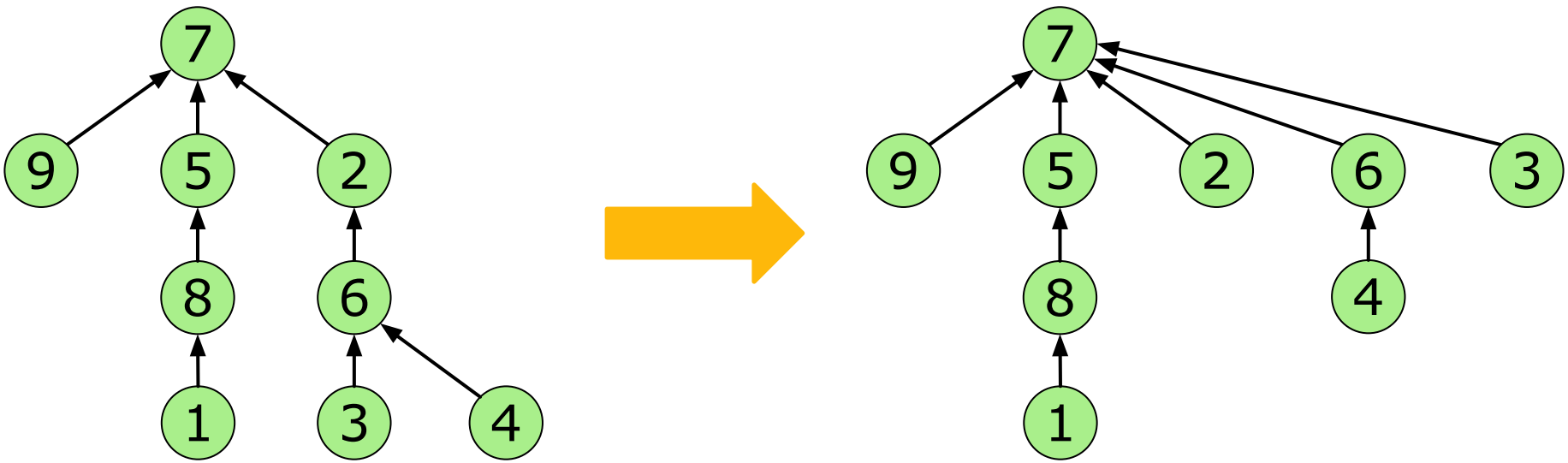
- Roots are the names of the disjoint set
- Plus, we want to move associated nodes up at the same time
- Why not move all nodes touched in a find to point directly to the root?

Path Compression

On a find operation point all the nodes on the search path directly to the root

- Keep a stack/queue as you traverse up
- Then empty to the stack/queue to repoint each stored node to the root

find(3)



Conditions for Data Compression

- The choice of data representation dictates how fast the data can be transmitted
- Careful choice of representation can improve the throughput of a given transmission channel without changing the channel
- A number of methods of ***data compression*** exist that reduce the size of the way the data is represented
- This is done without impacting the information itself

Huffman Coding

- David Huffman developed the construction for an optimal code in 1952, utilizing a binary tree for binary code
- The algorithm is shown on page 592; the tree that results from this has a probability of 1 in its root
- This algorithm is not deterministic in the sense that a unique tree is produced
- This is due to the fact that for trees with equal probability in their roots, the algorithm does not set their positions with respect to each other at the beginning or during execution
- Consequently, different trees can be obtained depending on where trees with equal probability are placed with respect to each other

Huffman Coding (continued)

- Regardless of the shape of the tree, however, the length of the codeword remains the same
- To assess the efficiency of the Huffman algorithm's compression, we will use ***weighted path length***
- The $L(m_i)$ terms represents the number of 0s and 1s in the codeword assign to m_i by the algorithm
- Figure 11.1 illustrates an example for the five letters A, B, C, D, and E with probabilities 0.39, 0.21, 0.19, 0.12, and 0.09, respectively
- The trees in Figure 11.1a-b differ in the way the two nodes with probability 0.21 are combined with a tree of 0.19

Huffman Coding (continued)

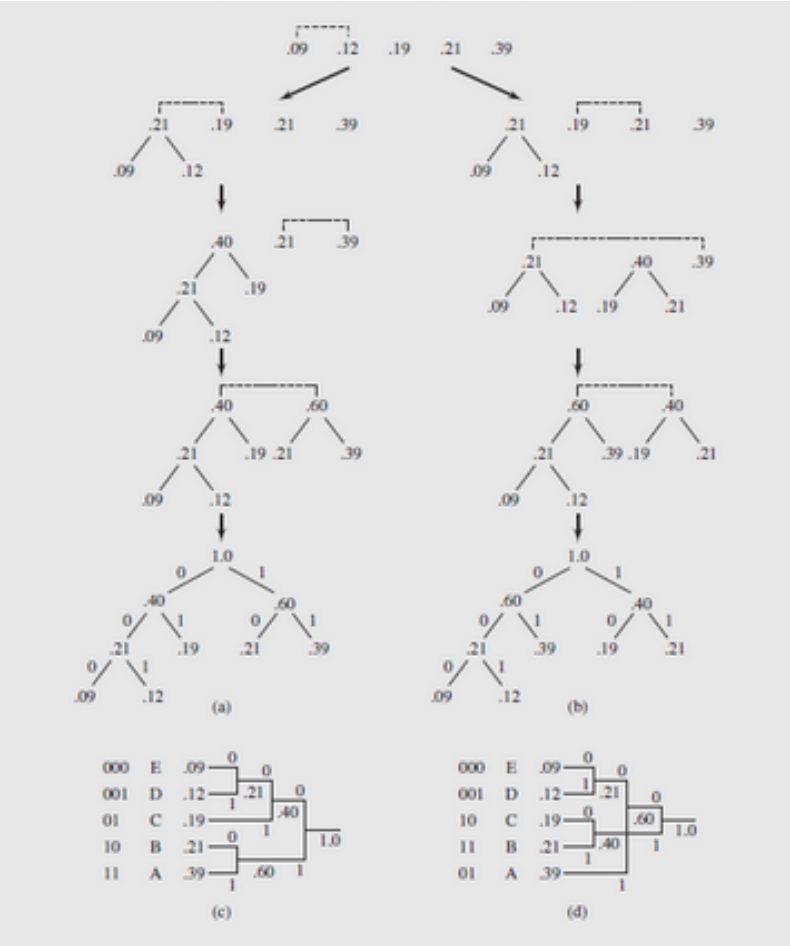


Fig. 11.1 Two Huffman trees created for five letters A, B, C, D, and E with probabilities .39, .21, .19, .12, and .09

Huffman Coding (continued)

- No matter which way is chosen, the codeword lengths for the five letters are the same – 2, 2, 2, 3, and 3 – respectively
- The codewords assigned to them are slightly different, however, as can be seen in Figures 11.1c-d
- Those present abbreviated and more commonly used versions of the way the trees in Figures 11.1a-b were created
- For these latter two trees, their average length is

$$L_{\text{Huf}} = .39 \cdot 2 + .21 \cdot 2 + .19 \cdot 2 + .12 \cdot 3 + .09 \cdot 3 = 2.21$$

- The average length computed by the entropy equation is

$$L_{\text{ave}} = .39 \cdot 1.238 + .21 \cdot 2.252 + .19 \cdot 2.396 + .12 \cdot 3.059 + .09 \cdot 3.474 = 2.09$$

- So these two are very close (within 5 percent)

Huffman Coding (continued)

- Codewords of the same length have been assigned to the corresponding letters in Figures 11.1a-b
- As we've seen, the average length for both trees is the same
- However, each way of building the Huffman tree should result in the same average length, regardless of the shape of the tree, if they start from the same data
- Huffman trees for letters P, Q, R, S, and T with probabilities 0.1, 0.1, 0.1, 0.2, and 0.5 respectively are shown in Figure 11.2
- Different codewords with different lengths may be assigned to these letters depending on how the lowest probabilities are chosen
- The average length remains the same, however, and is 2.0

Huffman Coding (continued)

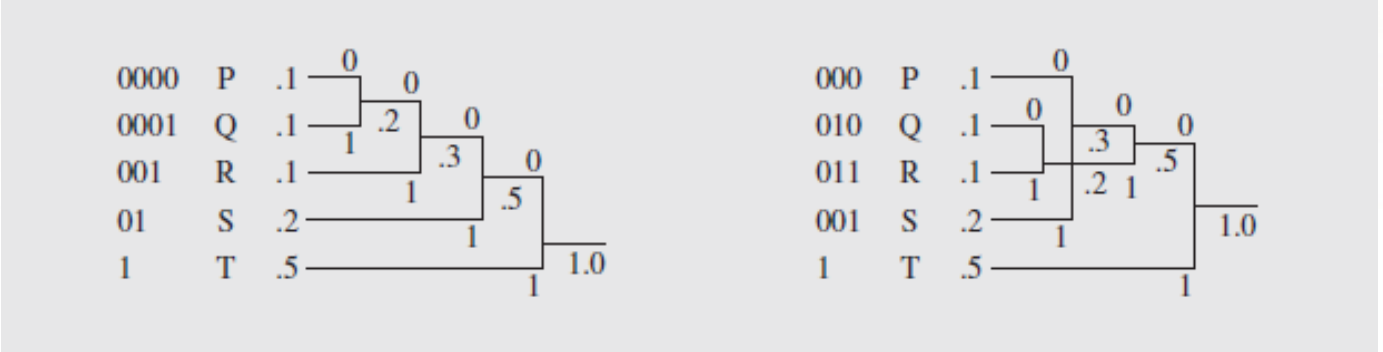


Fig. 11.2 Two Huffman trees generated for letters P, Q, R, S, and T with probabilities .1, .1, .1, .2, and .5

- There are a number of ways to implement the Huffman algorithm, but one of the more natural ways is to use a priority queue
- This is because it requires removing the two smallest probabilities and inserting the largest one
-

DATA STRUCTURES and ALGORITHMS in C++

FOURTH EDITION

Chapter 10: Hashing

Introduction

- In earlier chapters, the main process used by the searching techniques was comparing keys
- In sequential search for instance, the table storing the elements is searched in order, using key comparisons to determine a match
- In binary searching, we successively divide the table into halves to determine which cell to check, and again use key comparison to determine a match
- In binary search trees, the direction to take in the tree is determined by comparing keys in the nodes
- A different way to search can be based on calculating the position of the key in the table, based on the key's value

Introduction (continued)

- Since the value of the key is the only indication of position, if the key is known, we can access the table directly
- This reduces the search time from $O(n)$ or $O(\lg n)$ to 1 or at least $O(1)$
- No matter how many elements there are, the run time is the same
- Unfortunately, this is just an ideal; in real applications we can only approximate this
- The task is to develop a function, *h*, that can transform a key, *K*, into an index for a table used to store items of the same type as *K*
- The function *h* is called a ***hash function***

Introduction (continued)

- If *h* is able to transform different key values into different hash values, it is called a ***perfect hash function***
- For the hash function to be perfect, the table must have as many positions as there are items to be hashed
- However, it is not always possible to know how many elements will be hashed in advance, so some estimating is needed
- Consider a symbol table for a compiler, to store all the variable names
- Given the nature of the variable names typically used, a table with 1000 positions may be more than adequate
- However, even if we wanted to handle all possible variable names, we still need to design an appropriate *h*

Introduction (continued)

- For example, we could define *h* to be the sum of the ASCII values of the letters in the variable name
- If we restrict variables to 31 letters, we will need 3782 positions, since a variable with of 31 characters all "z" would sum to 31 · 122 (the ASCII code for "z") = 3782
- Even then, the function will not produce unique values, for *h* ("abc") = 97 + 98 + 99 = 294, and *h*("acb") = 97 + 99 + 98 = 294
- This is called a ***collision***, and is a measure of the usefulness of a hash function
- Avoiding collisions can be achieved by making *h* more complex, but complexity and speed must be balanced

Hash Functions

- The total possible number of hash functions for n items assigned to m positions in a table ($n \leq m$) is m^n
- The number of perfect hash functions is equal to the number of different placements of these items, and is $\frac{m!}{(m-n)!}$
- With 50 elements and a 100-position array, we would have a total of 100^{50} hash functions and about 10^{94} perfect hash functions (about 1 in a million)
- Most of the perfect hashes are impractical and cannot be expressed in a simple formula
- However, even among those that can, a number of possibilities exist

Hash Functions (continued)

- Division
  - Hash functions must guarantee that the value they produce is a valid index to the table
  - A fairly easy way to ensure this is to use modular division, and divide the keys by the size of the table, so $h(K) = K \bmod TSize$ where $TSize = sizeof(table)$
  - This works best if the table size is a prime number, but if not, we can use $h(K) = (K \bmod p) \bmod TSize$ for a prime $p \geq TSize$
  - However, nonprimes work well for the divisor provided they do not have any prime factors less than 20
  - The division method is frequently used when little is known about the keys

Hash Functions (continued)

- Folding
  - In folding, the keys are divided into parts which are then combined (or "folded") together and often transformed into the address
  - Two types of folding are used, ***shift folding*** and ***boundary folding***
  - In shift folding, the parts are placed underneath each other and then processed (for example, by adding)
  - Using a Social Security number, say 123-45-6789, we can divide it into three parts - 123, 456, and 789 – and add them to get 1368
  - This can then be divided modulo *TSize* to get the address
  - With boundary folding, the key is visualized as being written on a piece of paper and folded on the boundaries between the parts

Hash Functions (continued)

- Folding (continued)
  - The result is that alternating parts of the key are reversed, so the Social Security number part would be 123, 654, 789, totaling 1566
  - As can be seen, in both versions, the key is divided into even length parts of some fixed size, plus any leftover digits
  - Then these are added together and the result is divided modulo the table size
  - Consequently this is very fast and efficient, especially if bit strings are used instead of numbers
  - With character strings, one approach is to exclusively-or the individual character together and use the result
  - In this way, *h*(“abcd”) = “a” ∨ “b” ∨ “c” ∨ “d”

Hash Functions (continued)

- Mid-Square Function
  - In the mid-square approach, the numeric value of the key is squared and the middle part is extracted to serve as the address
  - If the key is non-numeric, some type of preprocessing needs to be done to create a numeric value, such as folding
  - Since the entire key participates in generating the address, there is a better chance of generating different addresses for different keys
  - So if the key is 3121, 3121^2 = 9,740,641, and if the table has 1000 locations, $h(3121) = 406$, which is the middle part of 3121^2
  - In application, powers of two are more efficient for the table size and the middle of the bit string of the square of the key is used
  - Assuming a table size of 1024, 3121^2 is represented by the bit string 1001010 *0101000010* 1100001, and the key, 322, is in italics

Collision Resolution

- The hashing we've looked at so far does have problems with multiple keys hashing to the same location in the table
- For example, consider a function that places names in a table based on hashing the ASCII code of the first letter of the name
- Using this function, all names beginning with the same letter would hash to the same position
- If we attempt to improve the function by hashing the first two letters, we achieve better results, but still have problems
- In fact, even if we used all the letters in the name, there is still a possibility of collisions
- Also, while using all the letters of the name gives a better distribution, if the table only has 26 positions there is no improvement in using the other versions

Collision Resolution (continued)

- So in addition to using more efficient functions, we also need to consider the size of the table being hashed into
- Even then, we cannot guarantee to eliminate collisions; we have to consider approaches that assure a solution
- A number of methods have been developed; we will consider a few in the following slides

Collision Resolution (continued)

- Open Addressing
  - In open addressing, collisions are resolved by finding an available table position other than the one to which the key hashed
  - If the position $h(K)$ is already occupied, positions are tried in the probing sequence

    $$norm(h(K) + p(1)), norm(h(K) + p(2)), \ldots, norm(h(K) + p(i)), \ldots$$

    until an open location is found, the same positions are tried again, or the table is full
  - The function p is called a ***probing function***, i is the ***probe***, and norm is a ***normalization function***, often division modulo the table size
  - The simplest realization of this is ***linear probing***, where the search proceeds sequentially from the point of the collision
  - If the end of the table is reached before finding an empty cell, it continued from the beginning of the table
  - If it reaches the cell before the one causing the collision, it then stops

Collision Resolution (continued)

- Open Addressing (continued)
  - The drawback to linear probing is that clusters of displaced keys tend to form
  - This is illustrated in Figure 10.1, where keys K_i are hashed to locations i
  - In Figure 10.1a, three keys have been hashed to their locations
  - In Figure 10.1b, the key B_5 arrives, but since A_5 is stored there, it is moved to the next location
  - Then A_9 is stored OK, but when B_2 arrives, it has to be placed in location 4, and a large cluster is forming (Figure 10.1b)
  - When B_9 arrives, it has to be placed from the beginning of the table, and finally, when C_2 shows up, it is placed five locations away from its home address

Collision Resolution (continued)

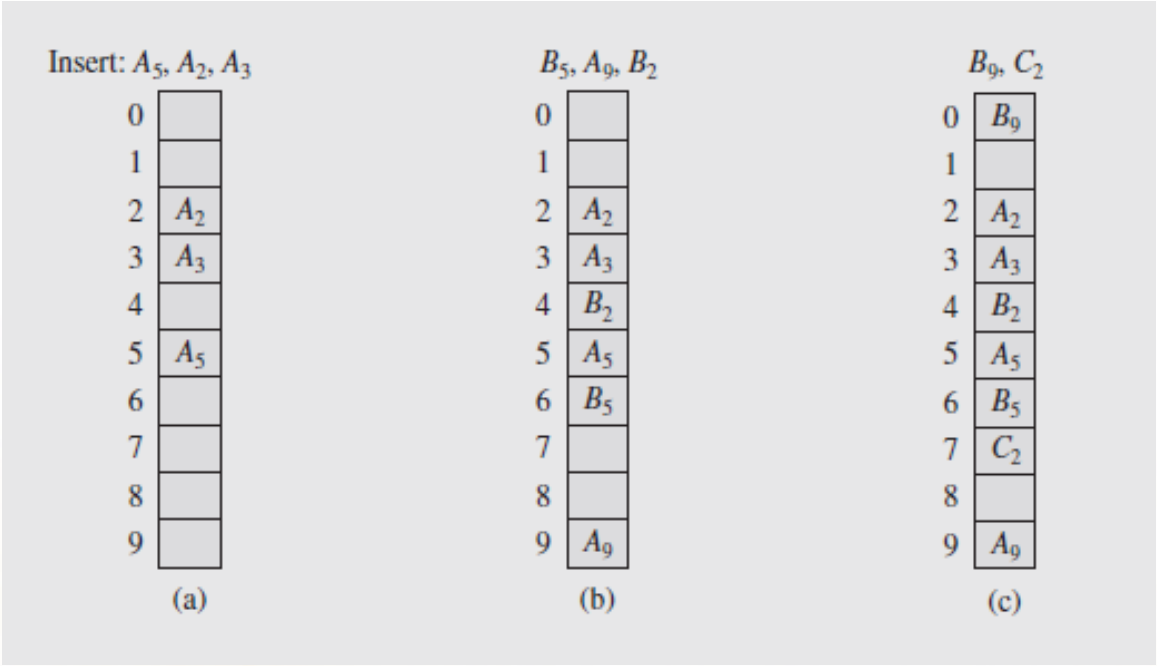


Fig. 10.1 Resolving collisions with the linear probing method. Subscripts indicate the home positions of the keys being hashed

Collision Resolution (continued)

- Open Addressing (continued)
  - As can be seen from the figure, empty cells immediately following clusters tend to be filled more quickly than other locations
  - So if a cluster is created, it tends to grow, and as it grows, it increases the likelihood of growing even larger
  - This behavior significantly reduces the efficiency of the hash table for processing data
  - So to avoid cluster creation and buildup, a better choice of the probing function, p, needs to be found
  - One possibility is to use a quadratic function producing the formula

$$p(i) = h(K) + (-1)^{i-1}((i + 1)/2)^2 \text{ for } i = 1, 2, \ldots, TSize - 1$$

  - Expressed as a sequence of probes, this is

$$h(K) + i^2, h(K) - i^2 \text{ for } i = 1, 2, \ldots, (TSize - 1)/2$$

Collision Resolution (continued)

- Open Addressing (continued)
  - Starting with the first hash, this produces the sequence

$$h(K),\ h(K)+1,\ h(K)-1,\ h(K)+4,\ h(K)-4,\ \ldots,\ h(K)+(TSize-1)^2/4,$$
$$h(K)-(TSize-1)^2/4$$

  - Each of these values is divided modulo *TSize*
  - Because the value of $h(K)$ tries only the even or odd positions in the table, the size of the table should not be an even number
  - The ideal value for the table size is a prime of the form $4j + 3$, which j is an integer
  - This will guarantee that all the table locations will be checked in the probing process
  - Applying this to the example of Figure 10.1 yields the configuration in Figure 10.2; B_2 still takes two probes, but C_2 only takes four

Collision Resolution (continued)

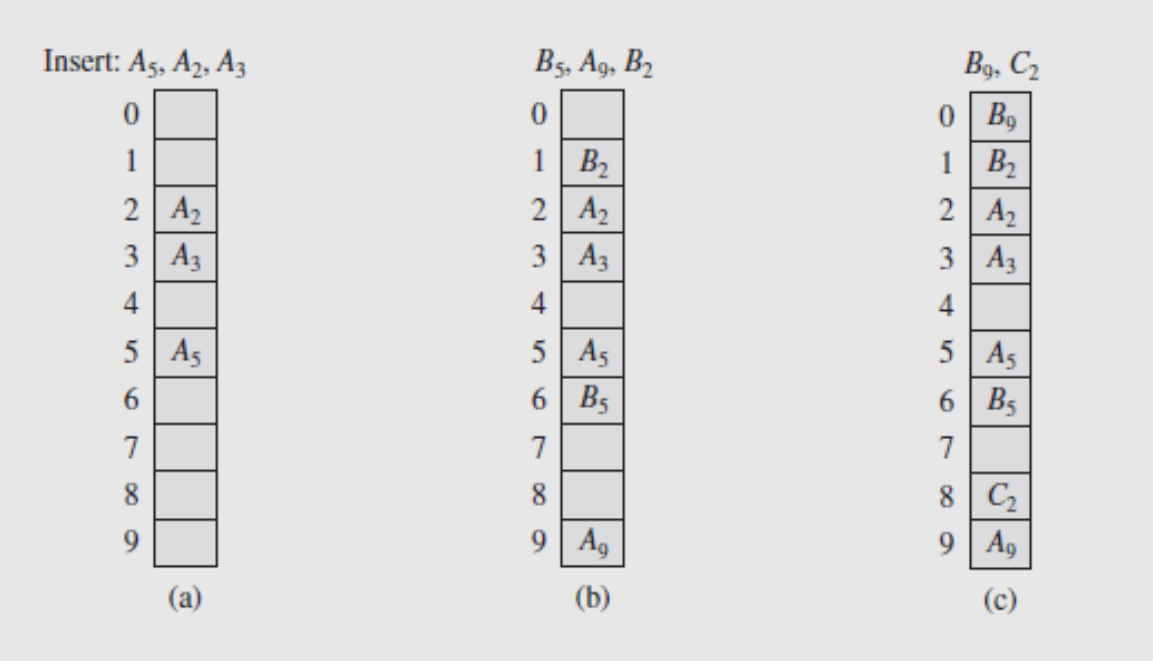


Fig. 10.2 Using quadratic probing for collision resolution

Collision Resolution (continued)

- Open Addressing (continued)
  - Notice that though we obtain better results with this approach, we don't avoid clustering entirely
  - This is because the same probe sequence is used for any collision, creating ***secondary clusters***
  - These are less of a problem than primary clusters, however
  - Another variation is to have *p* be a random number generator (RNG)
  - This eliminates the need to have special conditions on the table size, and does prevent secondary cluster formation
  - However, it does have an issue with repeating the same probing sequence for the same keys

Collision Resolution (continued)

- Open Addressing (continued)
  - The best approach to secondary clustering is through the technique of ***double hashing***
  - This utilizes two hashing functions, one for the primary hash and the other to resolve collisions
  - In this way the probing sequence becomes

$$h(K),\ h(K) + h_p(K),\ \ldots,\ h(K) + i \cdot h_p(K),\ \ldots$$

  - Here, h is the primary hashing function and h_p is the secondary hash
  - The table size should be a prime number so every location is included in the sequence, since the values above are divided modulo *TSize*
  - Empirical evidence shows that this approach works well to eliminate secondary clustering, since the probe sequence is based on h_p

Collision Resolution (continued)

- Open Addressing (continued)
  - This is because the probing sequence for key K_1 hashed to location j is

$$j,\ j + h_p(K_1),\ j + 2 \cdot h_p(K_1),\ \ldots$$

  - And if another key hashes to $j + h_p(K_1)$, the next location to be checked is $j + h_p(K_1) + h_p(K_2)$, not $j + 2 \cdot h_p(K_1)$
  - This avoids secondary clustering, as long as h_p is well chosen
  - So even if two keys hash to the same position initially, the probing sequences can be different for each key
  - The use of two different hash functions can be time-consuming, so it is possible to define the second hash in terms of the first
  - For example, the function could be $h_p(K) = i \cdot h(K) + 1$; for key K_1 the probe sequence is $j,\ 2j + 1,\ 5j + 2,\ \ldots$; if K_2 hashes to $2j + 1$, the sequence is $2j + 1,\ 4j + 3,\ 10j + 11,\ \ldots$

Collision Resolution (continued)

- Chaining
  - In ***chaining***, the keys are not stored in the table, but in the `info` portion of a linked list of nodes associated with each table position
  - This technique is called ***separate chaining***, and the table is called a ***scatter table***
  - This was the table never overflows, as the lists are extended when new keys arrive, as can be seen in Figure 10.5
  - This is very fast for short lists, but as they increase in size, performance can degrade sharply
  - Gains in performance can be made if the lists are ordered so unsuccessful searches don't traverse the entire list, or by using self-organizing linked lists
  - This approach requires additional space for the pointers, so if there are a large number of keys involved, space requirements can be high

Collision Resolution (continued)

- Chaining (continued)

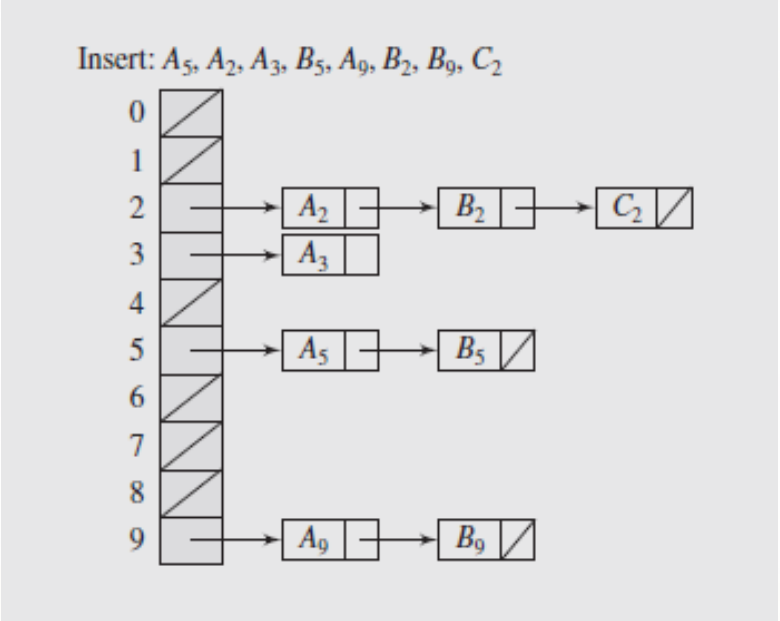


Fig. 10.5 In chaining, colliding keys are put on the same linked list.

Deletion

- How can data be removed from a hash table?
- If chaining is used, the deletion of an element entails deleting the node from the linked list holding the element
- For the other techniques we've considered, deletion usually involves more careful handling of collision issues, unless a perfect hash function is used
- This is illustrated in Figure 10.10a, which stores keys using linear probing
- In Figure 10.10b, when A_4 is deleted, attempts to find B_4 check location 4, which is empty, indicating B_4 is not in the table
- A similar situation occurs in Figure 10.10c, when A_2 is deleted, causing searches for B_1 to stop at position 2

Deletion (continued)

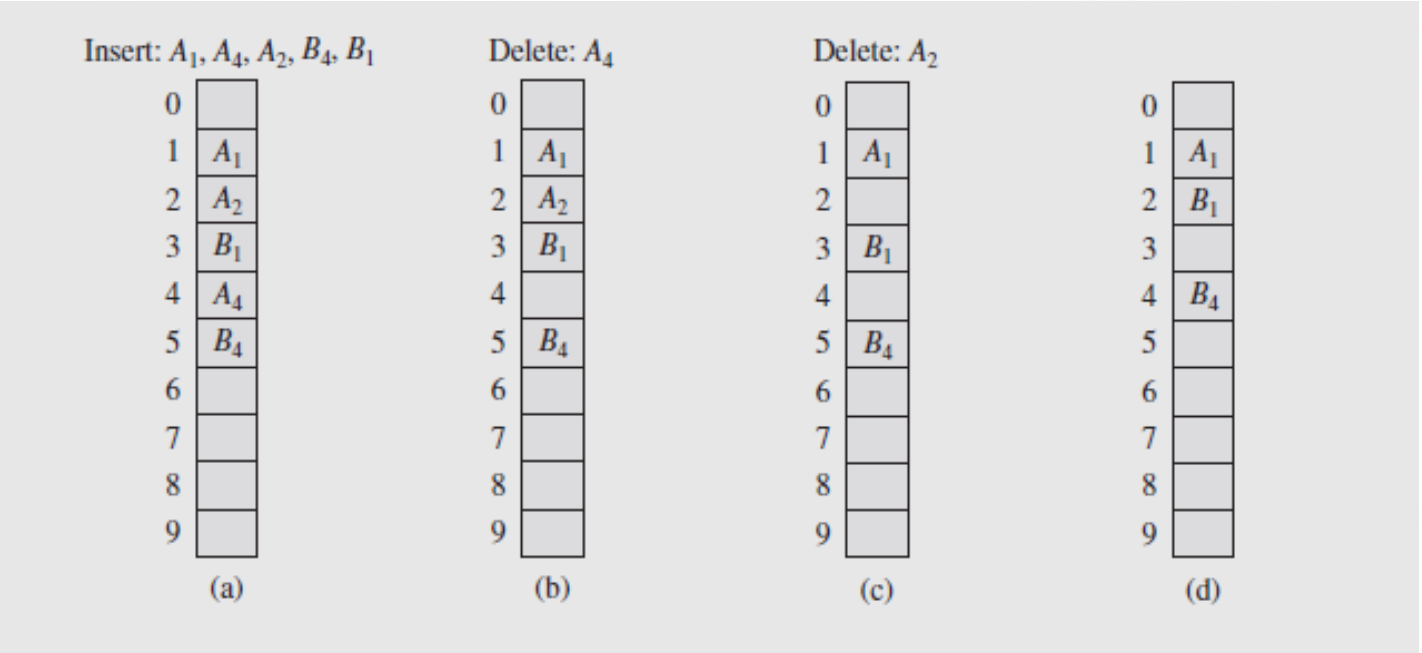


Fig. 10.10 Linear search in the situation where both insertion and deletion of keys are permitted

- A solution to this is to leave the deleted keys in the table with some type of indicator that the keys are not valid
- This way, searches for elements won't terminate prematurely
- When new keys are inserted, they can overwrite the marked keys

Rehashing

- When hash tables become full, no more items can be added
- As they fill up and reach certain levels of occupancy (saturation), their efficiency falls due to increased searching needed to place items
- A solution to these problems is rehashing, allocating a new, larger table, possibly modifying the hash function (and at least *TSize*), and hashing all the items from the old table to the new
- The old table is then discarded and all further hashes are done to the new table with the new function
- The size of the new table can be determined in a number of ways: doubled, a prime closest to doubled, etc.

Also make sure to review your notes & the reading on cryptographic hashes

An Illustrated Guide to Cryptographic Hashes

With the recent news of weaknesses in some common security algorithms (MD4, MD5, SHA-0), many are wondering exactly what these things are: They form the underpinning of much of our electronic infrastructure, and in this Guide we'll try to give an overview of what they are and how to understand them in the context of the recent developments.

Table of Contents

But note: though we're fairly strong on security issues, we are **not** crypto experts. We've done our best to assemble (digest?) the best available information into this Guide, but we welcome being pointed to the errors of our ways.

What is a cryptographic hash?

A "hash" (also called a "digest", and informally a "checksum") is a kind of "signature" for a stream of data that represents the contents. The closest real-life analog we can think is "a tamper-evident seal on a software package": if you open the box (change the file), it's detected.

Let's first see some examples of hashes at work.

Many Unix and Linux systems provide the **md5sum** program, which reads a stream of data and produces a fixed, 128-bit number that summarizes that stream using the popular "MD5" method. Here, the "streams of data" are "files" (two of which we see directly, plus one that's too large to display).

```
$ cat smallfile
This is a very small file with a few characters

$ cat bigfile
This is a larger file that contains more characters.
This demonstrates that no matter how big the input
stream is, the generated hash is the same size (but
of course, not the same value). If two files have
a different hash, they surely contain different data.

$ ls -l empty-file smallfile bigfile linux-kernel
-rw-rw-r--    1 steve    steve           0 2004-08-20 08:58 empty-file
-rw-rw-r--    1 steve    steve          48 2004-08-20 08:48 smallfile
-rw-rw-r--    1 steve    steve         260 2004-08-20 08:48 bigfile
-rw-r--r--    1 root     root      1122363 2003-02-27 07:12 linux-kernel

$ md5sum empty-file smallfile bigfile linux-kernel
d41d8cd98f00b204e9800998ecf8427e  empty-file
75cdbfeb70a06d42210938da88c42991  smallfile
6e0b7a1676ec0279139b3f39bd65e41a  bigfile
c74c812e4d2839fa9acf0aa0c915e022  linux-kernel
```

This shows that *all* input streams yield hashes of the same length, and to experiment, try changing just one character of a small test file. You'll find that even very small changes to the input yields sweeping changes in the value of the hash, and this is known as the avalanche effect.

The avalanche effect can be best seen by hashing two files with nearly identical content. We've changed the first character of a file from **T** to **t**, and when looking at the binary values for these ASCII characters, we see that they differ by just one bit:

```
T -> 0x54 -> 0 1 0 1  0 1 0 0
t -> 0x74 -> 0 1 1 1  0 1 0 0
```

This single bit of change in the input produces a very large change in the output:

DATA STRUCTURES and ALGORITHMS in C++

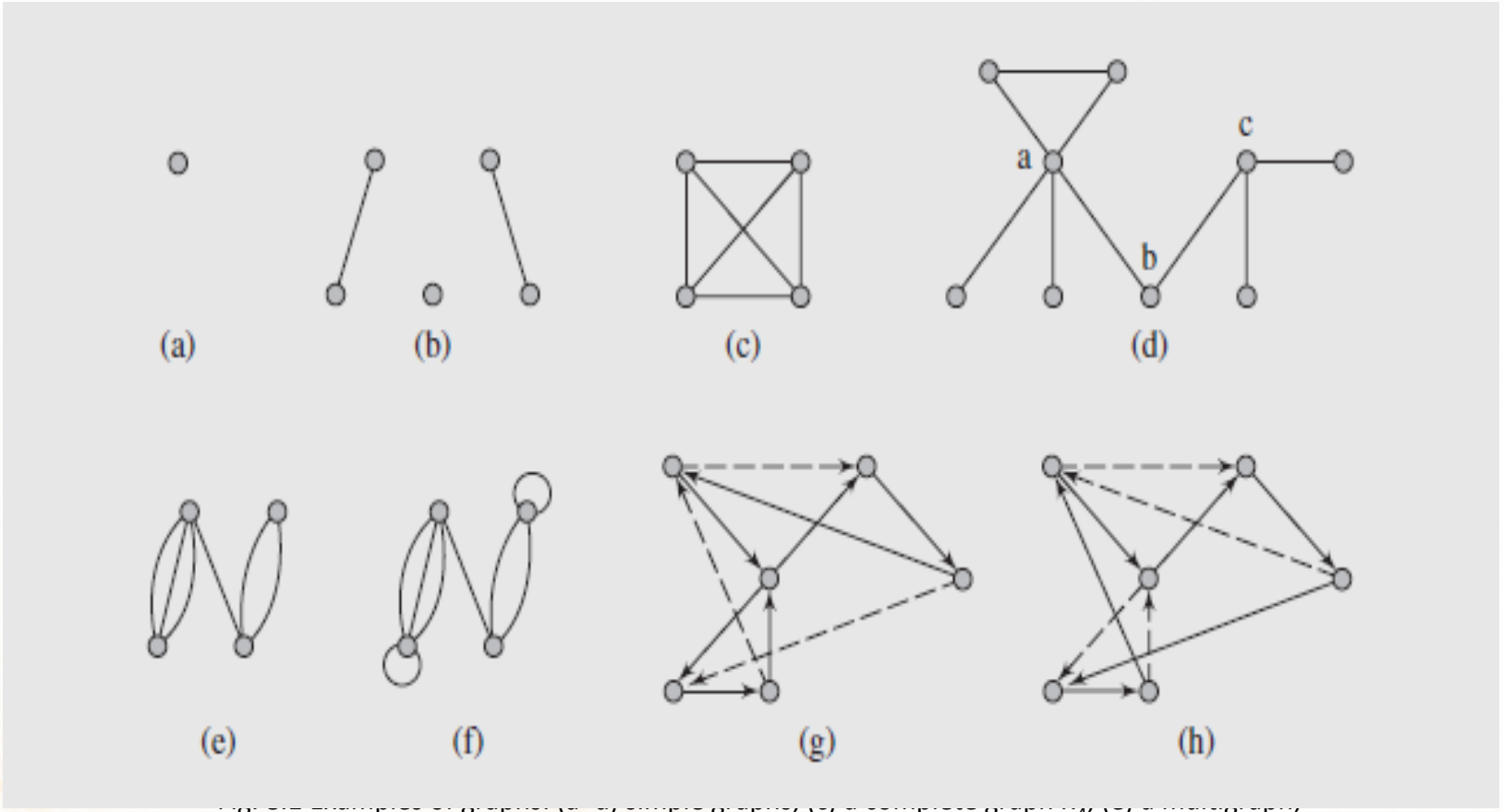
FOURTH EDITION

Chapter 8: Graphs

Introductory Remarks

- Although trees are quite flexible, they have an inherent limitation in that they can only express hierarchical structures
- Fortunately, we can generalize a tree to form a ***graph***, in which this limitation is removed
- Informally, a graph is a collection of nodes and the connections between them
- Figure 8.1 illustrates some examples of graphs; notice there is typically no limitation on the number of vertices or edges
- Consequently, graphs are extremely versatile and applicable to a wide variety of situations
- Graph theory has developed into a sophisticated field of study since its origins in the early 1700s

Introductory Remarks (continued)



(f) a pseudograph; (g) a circuit in a digraph; (h) a cycle in the digraph

Introductory Remarks (continued)

- And, while many results are theoretical, the applications of graphs are numerous and worth consideration
- First, though, we need to consider some definitions
- A ***simple graph*** $G = (V, E)$ consists of a (finite) set denoted by V, and a collection E, of unordered pairs $\{u, v\}$ of distinct elements from V
- Each element of V is called a ***vertex*** or a ***point*** or a ***node***, and each element of E is called an ***edge*** or a ***line*** or a ***link***
- The number of vertices, the ***cardinality*** of V, is called the ***order of graph*** and devoted by $|V|$
- The cardinality of E, called the ***size of graph***, is denoted by $|E|$

Introductory Remarks (continued)

- A graph $G = (V, E)$ is ***directed*** if the edge set is composed of ordered vertex (node) pairs
- Now these definitions restrict the number of edges that can occur between any two vertices to one

Introductory Remarks (continued)

- A ***path*** between vertices v_1 and v_n is a sequence of edges denoted $v_1, v_2, \ldots, v_{n-1}, v_n$
- If $v_1 = v_n$, and the edges don't repeat, it is a ***circuit*** (Figure 8.1g); if the vertices in a circuit are different, it is a ***cycle*** (Figure 8.1h)
- A ***weighted graph*** assigns a value to each edge, based on contextual usage
- A ***complete*** graph of n vertices, denoted K_n, has exactly one edge between each pair of vertices (Figure 8.1c)

Introductory Remarks (continued)

- A ***subgraph*** of a graph G, designated G', is the graph (V', E') where $V' \subseteq V$ and $E' \subseteq E$

- Two vertices are ***adjacent*** if the edge defined by them is in E
- That edge is called ***incident with*** the vertices
- The number of edges incident with a vertex v, is the ***degree*** of the vertex; if the degree is 0, v is called ***isolated***
- Notice that the definition of a graph allows the set E to be empty, so a graph may be composed of isolated vertices

Graph Representation

- Graphs can be represented in a number of ways
- One of the simplest is an ***adjacency list***, where each vertex adjacent to a give vertex is listed
- This can be designed as a table (known as a ***star representation***) or a linked list, shown in Figure 8.2b-c on page 393
- Another representation is as a matrix, which can be designed in two ways
- An ***adjacency matrix*** is a $|V|$ x $|V|$ binary matrix where:

$$a_{ij} = \begin{cases} 1 & \text{if there exists an edge } (v_i v_j) \\ 0 & \text{otherwise} \end{cases}$$

Graph Representation (continued)

- An example of an adjacency matrix is shown in Figure 8.2d
- The order of the vertices in the matrix is arbitrary, so there are n! possible matrices for a graph of *n* vertices

Graph Traversals

- Like tree traversals, graph traversals visit each node once
- However, we cannot apply tree traversal algorithms to graphs because of cycles and isolated vertices
- One algorithm for graph traversal, called the depth-first search, was developed by John Hopcroft and Robert Tarjan in 1974
- In this algorithm, each vertex is visited and then all the unvisited vertices adjacent to that vertex are visited
- If the vertex has no adjacent vertices, or if they have all been visited, we backtrack to that vertex's predecessor
- This continues until we return to the vertex where the traversal started

Graph Traversals (continued)

- If any vertices remain unvisited at this point, the traversal restarts at one of the unvisited vertices
- Although not necessary, the algorithm assigns unique numbers to the vertices, so they are renumbered
- Pseudocode for this algorithm is shown on page 395
- Figure 8.3 shows an example of this traversal; the numbers indicate the order in which the nodes are visited; the solid lines indicate the edges traversed during the search

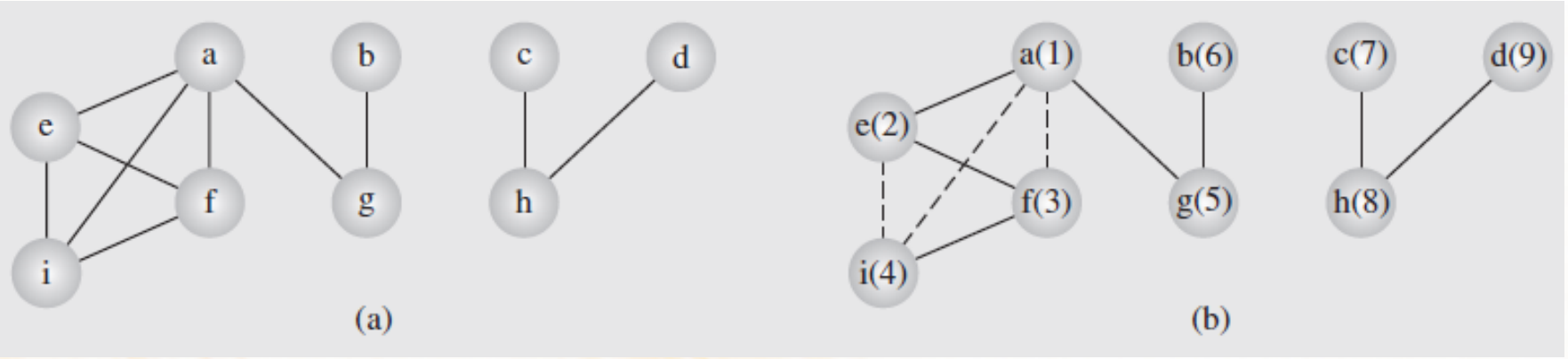


Fig. 8.3 An example of application of the depthFirstSearch() algorithm to a graph

Graph Traversals (continued)

- The algorithm guarantees that we will create a tree (or a forest, which is a set of trees) including the graph's vertices
- Such a tree is called a ***spanning tree***
- The guarantee is based on the algorithm not processing any edge that leads to an already visited node
- Consequently, some edges are not included in the tree (marked with dashed lines)
- The edges included in the tree are called ***forward edges***; those omitted are called ***back edges***
- In Figure 8.4, we can see this algorithm applied to a ***digraph***, which is a graph where the edges have a direction

Graph Traversals (continued)

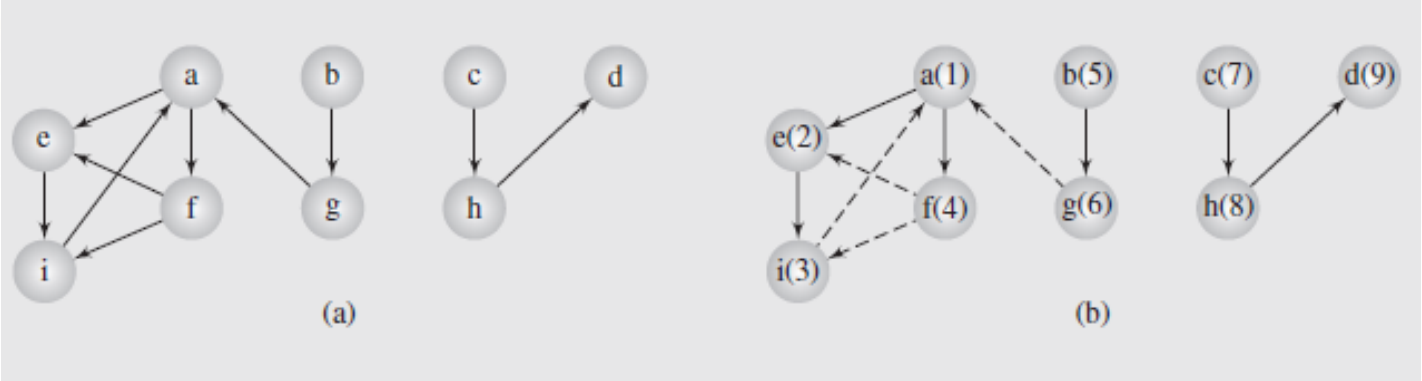


Fig. 8.4 The depthFirstSearch() algorithm applied to a digraph

- Notice in this case we end up with a forest of three trees, because the traversal must follow the direction of the edges
- There are a number of algorithms based on depth-first searching
- However, some are more efficient if the underlying mechanism is breadth-first instead

Graph Traversals (continued)

- Recall from our consideration of tree traversals that depth-first traversals used a stack, while breadth-first used queues
- This can be extended to graphs, as the pseudocode on page 397 illustrates
- Figure 8.4 shows this applied to a graph; Figure 8.5 shows the application to a digraph
- In both, the basic operation is to mark all the vertices accessible from a given vertex, placing them in a queue as they are visited
- The first vertex in the queue is then removed, and the process repeated
- No visited nodes are revisited; if a node has no accessible nodes, the next node in the queue is removed and processed

Graph Traversals (continued)

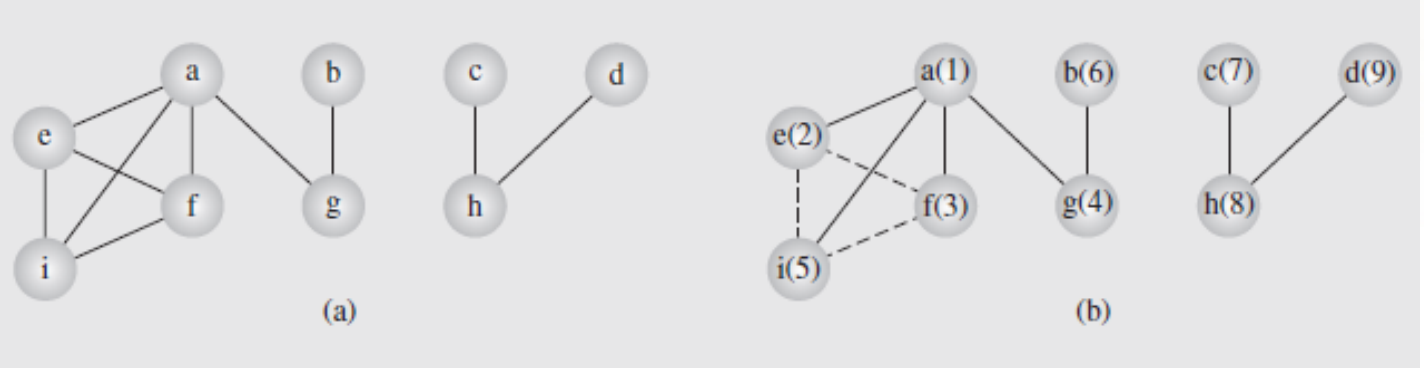


Fig. 8.5 An example of application of the breadthFirstSearch() algorithm to a graph

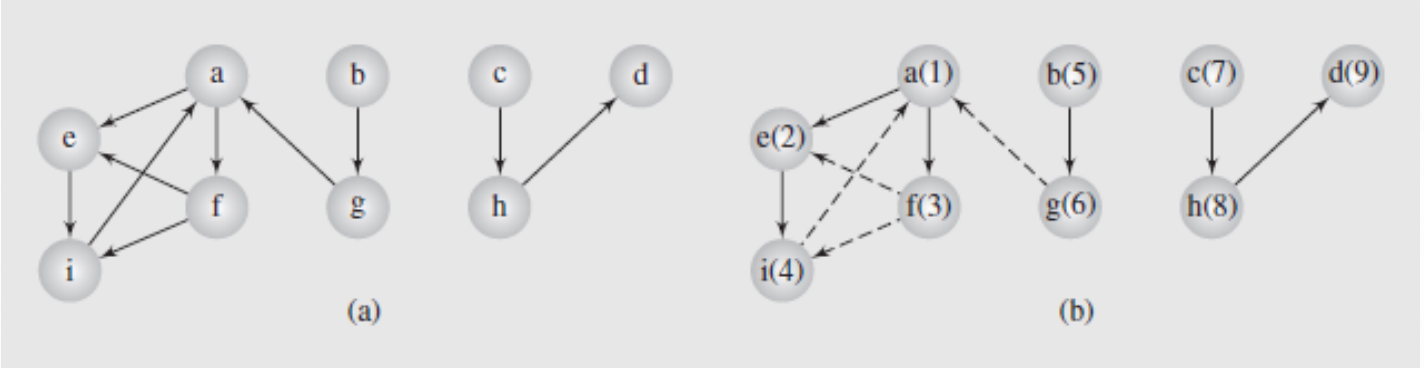


Fig. 8.6 The breadthFirstSearch() algorithm applied to a digraph

Shortest Paths

- A classical problem in graph theory is finding the shortest path between two nodes, with numerous approaches suggested
- The edges of the graph are associated with values denoting such things as distance, time, costs, amounts, etc.
- If we're determining the distance between two vertices, say *v* and *u*, information about the distance between the intermediate vertices in the path, *w*, needs to be kept track of
- This can be recorded as a label associated with the vertices
- The label may simply be the distance between vertices, or the distance along with the current node's predecessor in the path
- Methods for finding shortest paths depend on these labels

Shortest Paths (continued)

- Based on how many times the labels are updated, solutions to the shortest path problem fall into two groups
- In ***label-setting*** methods, one vertex is assigned a value that remains unchanged
- This occurs each time we go through the vertices that remain to be processed
- The main drawback to this is that we cannot process graphs that have negative weights on any edges
- In ***label-correcting*** methods, any label can be changed
- This means it can be applied to graphs with negative weights as long as they don't have negative cycles (a cycle where the sum of the edges is a negative value)

Shortest Paths (continued)

- However this method guarantees that after processing is complete, for all vertices the current distances indicate the shortest path
- Most of these forms (both label-setting and label-correcting) can be looked at as part of the same general process, however
- That is the task of finding the shortest paths from one vertex to all the other vertices, the pseudocode being on page 399
- In this algorithm, a label is defined as:

```
label(v) = (currDist(v),predecessor(v))
```

- Two open issues in the code are the design of the set called `toBeChecked` and the order new values are assigned to *v*
- It is the design of the set that impacts both the choice of *v* and the efficiency of the algorithm

Shortest Paths (continued)

- The distinction between label-setting and label-correcting algorithms is the way the value for vertex *v* is chosen
- This is the vertex in the set `toBeChecked` with the smallest current distance
- In considering label-setting algorithms, one of the first was developed by Edsgar Dijkstra in 1956
- In this algorithm, the shortest from among a number of paths from a vertex, *v*, are tried
- This means that a particular path may be extended by adding one more edge to it each time *v* is checked
- However, if the path is longer than any other path from that point, it is dropped, and the other path is expanded

Shortest Paths (continued)

- Since the vertices may have more than one outgoing edge, each new edge adds possible paths for exploration
- Thus each vertex is visited, the new paths are started, and the vertex is then not used anymore
- Once all the vertices are visited, the algorithm is done
- Dijkstra's algorithm is shown on page 400; it is derived from the general algorithm by changing the line

```
v=a vertex in toBeChecked;
```

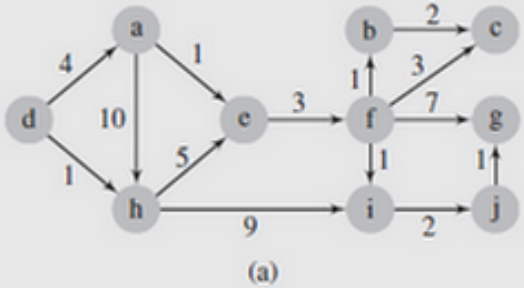
to

```
v=a vertex in toBeChecked with minimal currDist(v);
```

- It also extends the condition in the `if` to make permanent the current distance of vertices eliminated from the set

Shortest Paths (continued)

- Notice that the set's structure is not indicated; recall it is the structure that determines efficiency
- Figure 8.7 illustrates this for the graph in part (a)



(a)

iteration:	init	1	2	3	4	5	6	7	8	9	10
active vertex:		d	h	a	e	f	b	i	c	j	g
a	∞	4	4								
b	∞	∞	∞	∞	∞	9					
c	∞	∞	∞	∞	∞	11	11	11			
d	0										
e	∞	∞	6	5							
f	∞	∞	∞	∞	8						
g	∞	∞	∞	∞	∞	15	15	15	15	12	
h	∞	1									
i	∞	∞	10	10	10	9	9				
j	∞	∞	∞	∞	∞	∞	∞	11	11		

(b)

Fig. 8.7 An execution of DijkstraAlgorithm()

Spanning Trees

- Consider an airline that has routes between seven cities represented as the graph in Figure 8.14a

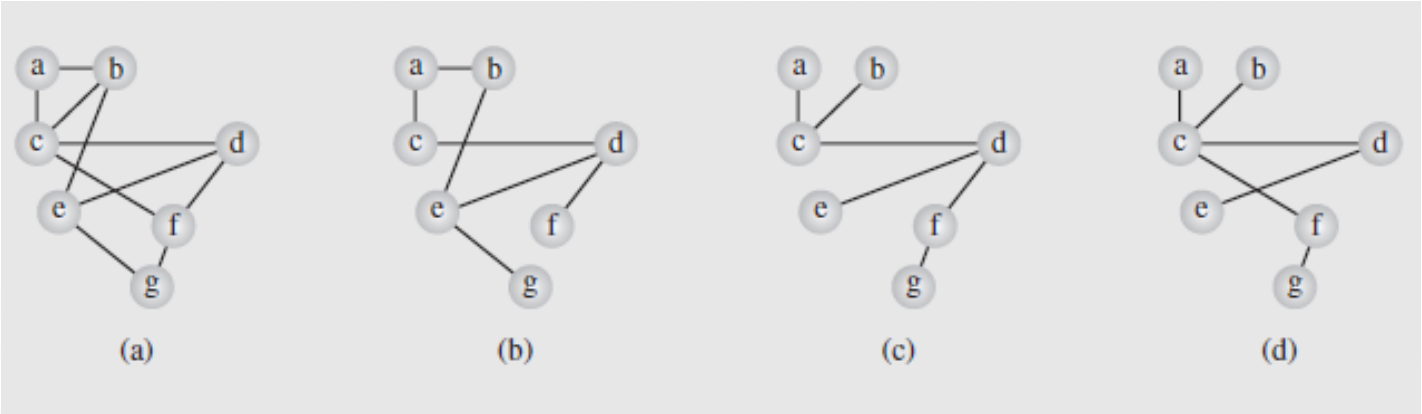


Fig. 8.14 A graph representing (a) the airline connections between seven cities and (b–d) three possible sets of connections

- If economic hardships force the airline to cut routes, which ones should be kept to preserve a route to each city, if only indirectly?
- One possibility is shown in Figure 8.14b

Spanning Trees (continued)

- However, we want to make sure we have the minimum connections necessary to preserve the routes
- To accomplish this, a spanning tree should be used, specifically one created using `depthFirstSearch()`
- There is a possibility of multiple spanning trees (Figure 8.14c-d), but each of these has the minimum number of edges
- We don't know which of these might be optimal, since we haven't taken distances into account
- The airline, wanting to minimize costs, will want to use the shortest distances for the connections
- So what we want to find is the ***minimum spanning tree***, where the sum of the edge weights is minimal

Spanning Trees (continued)

- The problem we looked at earlier involving finding a spanning tree in a simple graph is a case of this where edge weights = 1
- So each spanning tree is a minimum tree in a simple graph
- There are a number of solutions to the minimum spanning tree problem, and we will consider two
- One popular algorithm is Kruskal's algorithm, developed by Joseph Kruskal in 1956
- It orders the edges by weight, and then checks to see if they can be added to the tree under construction
- It will be added if its inclusion doesn't create a cycle

Spanning Trees (continued)

- The algorithm is as follows:

```
KruskalAlgorithm(weighted connected undirected graph)
    tree = null;
    edges = sequence of all edges of graph sorted by weight;
    for (i = 1; i # |E| and |tree| < |V| – 1; i++)
      if e_i from edges does not form a cycle with edges in tree
        add ei to tree;
```

- A step-by-step example of the application of this algorithm is shown in Figure 8-15ba-bf on page 413

DATA STRUCTURES and ALGORITHMS in C++

FOURTH EDITION

Chapter 9: Sorting

Introduction

- Sorting data to improve the efficiency with which it is handled is an accepted part of daily life—it's convenient!
- Using sorted data is addressed computationally by considering the process and deciding which criteria to use in arranging the data
- The choice may vary considerably depending on the application and the user's needs
- Frequently, a natural ordering will suggest itself that may be useful
- Once a criterion is chosen, the second step is determining how to apply it to the data

Introduction (continued)

- Efficiency criteria and methods for comparing algorithms in a quantitative fashion have to be devised
- This evaluation should be machine-independent, because hardware may facilitate or impede the software process
- Common measures are the number of comparisons that occur and the number of data movements that take place
- This isn't surprising, because in sorting, we compare and possibly move data; the size of the data set then plays a role

Introduction (continued)

- Since these values may be difficult to determine exactly, approximations are often used
- These can then be represented using big-O notation to indicate orders of magnitude
- We must also consider the behavior of algorithms
  - Some may differ depending on the original state of the data set (sorted, unsorted, partially sorted)
  - Others may behave the same way regardless of the data
  - Typically, we obtain a best case, worst case, and average case

Introduction (continued)

- We may also find that the number of comparisons and number of data movements don't apparently coincide
- An algorithm maybe very efficient in one case, and perform poorly on the other
- So practical considerations have to be taken into account in choosing the algorithm to use
- The bottom line is that any theoretical results have to be tempered by practical application

Elementary Sorting Algorithms

- Insertion Sort
  - ***Insertion sort*** is a simple algorithm that builds the final sorted list one item at a time
  - Each repetition of the sort takes an element from input and inserts it into the correct position in the already-sorted list, until no input elements remain
  - The choice of which element to remove from the input is arbitrary, and can be made using almost any choice algorithm
  - Sorting is typically done in-place
  - The resulting array after k iterations has the property where the first $k + 1$ entries are sorted
  - In each iteration the first remaining entry of the input is removed and inserted into the result at the correct position

Elementary Sorting Algorithms

- Insertion Sort (continued)
  - The pseudocode for the insertion sort follows:

```
insertionsort(data[],n)
  for i = 1 to n-1
    move all elements data[j] greater than data[i] by one position;
    place data[i] in its proper position;
```

  - Notice that on each pass only a portion of the array is considered; it is only in the last pass that the whole array is processed
  - Figure 9.1 shows the manipulations that occur when the algorithm runs against the list `[5 2 3 8 1]`
  - Since an element with one array is already sorted, the algorithm starts with the second element (in position 1), which is placed in `tmp`
  - We compare this with the elements in position `data[j]`, $0 \leq j \leq i$, and those larger than `tmp` are moved up one position

Elementary Sorting Algorithms

- Insertion Sort (continued)

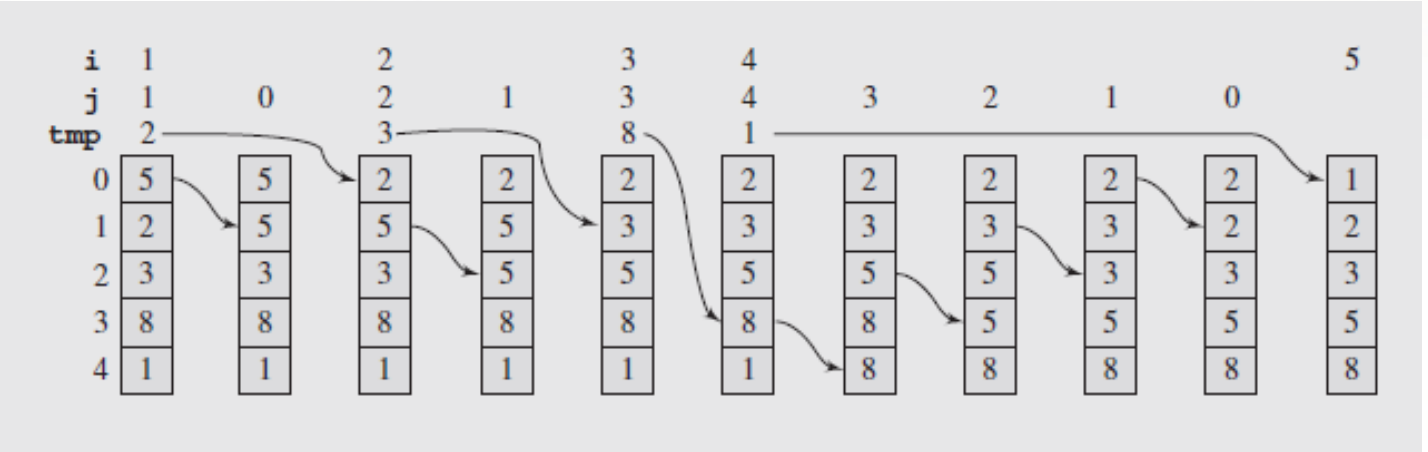


Fig. 9.1 The array [5 2 3 8 1] sorted by insertion sort

– The sort can be implemented with the following code:

```
template<class T>
void insertionSort(T data[], int n) {
  for (int i = 1, j; i < n; i++) {
    T tmp = data[i];
    for (j = i; j > 0 && tmp < data[j - 1]; j--)
      data[j] = data[j - 1];
    data[j] = tmp;  } }
```

Elementary Sorting Algorithms

- Insertion Sort (continued)
  - One important characteristic of the insertion sort is that it only sorts when necessary
  - For instance if the array is already sorted, only the temporary variable is initialized, and that value is moved back to its original location
  - The algorithms also recognizes when the array is partially sorted, and stops accordingly
  - However, it can only recognize that, and elements in their proper locations can be overlooked, so items can be moved and subsequently moved back
  - Another obvious disadvantage is the movement of data items to insert an item, which can occur in any position
  - This data movement, combined with the move-and-replace mentioned above, significantly impacts efficiency

Elementary Sorting Algorithms (continued)

- Selection Sort
  - ***Selection sort*** is an in-place comparison sort that tries to localize the exchange of array elements by finding an unsorted item and putting it in its final location
  - It works by locating the minimum element in the list and swapping it with the item in the first location
  - Then it advances one position and repeats the process with the next smallest element, etc. until it reaches the end of the list
  - Effectively, the list is divided into two parts
  - There is the sublist of items already sorted, which is built up from left to right and is found at the beginning
  - Then there is the sublist of items remaining to be sorted, occupying the remainder of the array

Elementary Sorting Algorithms (continued)

- Selection Sort (continued)
  - The pseudocode for the algorithm reflects its simplicity:

```
selectionsort(data[],n)
  for i = 0 to n-2
    select the smallest element among data[i], . . . , data[n-1];
    swap it with data[i];
```

  - The last value for *i* is *n* – 2 since if all items have been looked at and placed except for the last, then the n^{th} element has to be the largest
  - Figure 9.2 shows an example of this for the same list as Figure 9.1

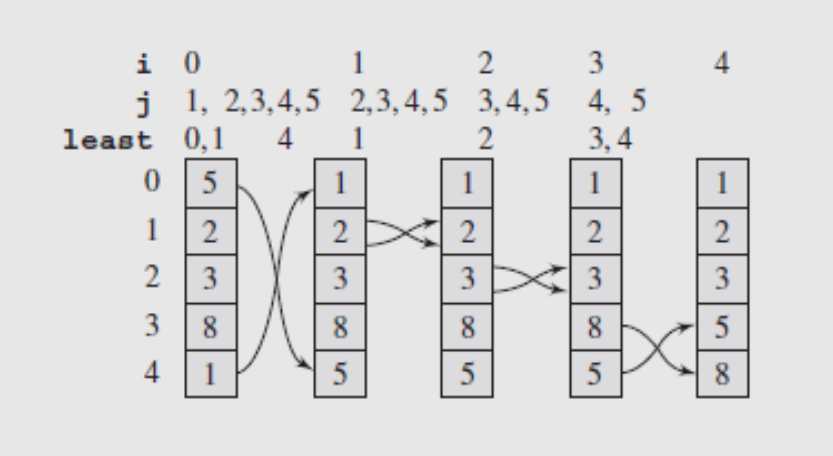


Fig. 9.2 The array [5 2 3 8 1] sorted by selection sort

Elementary Sorting Algorithms (continued)

- Selection Sort (continued)
  - The following code implements this algorithm:

```
template<class T>
void selectionsort(T data[], int n) {
  for (int i = 0,j,least; i < n-1; i++) {
    for (j = i+1, least = i; j < n; j++)
      if (data[j] < data[least])
    least = j;
    swap(data[least],data[i]);
  }
}
```

  - The `swap()` function is used to exchange the elements in the list; not that the variable `least` refers to position of the smallest value, not its position

Efficient Sorting Algorithms (continued)

- Heap Sort
  - Even though selection sort is fairly inefficient ($O(n^2)$), it makes relatively few moves of the data
  - So if the comparison portion of the sort can be improved, its performance can likewise improve
  - This was the motivation behind the development of ***heap sort***, created by John W. J. Williams in 1964
  - Heap sort is a comparison-based, in-place algorithm, but is not a stable sort
  - Although somewhat slower in practice than quicksort, it has the advantage of a more favorable worst-case runtime
  - Recall that selection sort finds the smallest element in the list and places it first, then the next smallest, etc.

Efficient Sorting Algorithms (continued)

- Heap Sort (continued)
  - For ascending order, heap sort places the largest element last in the array, then puts the next largest in front of that, etc.
  - To accomplish this, heap sort uses a two phase process
    - The first phase is to build a heap out of the data
    - The second phase begins by removing the largest item from the heap and inserting that item into the sorted array
    - For the first element, this would be position 0 of the array
    - Then we reconstruct the heap and remove the next largest item, and insert it into the array
    - After all the objects are removed from the heap, we have a sorted array
    - The order of the sorted elements can be selected by choosing a min-heap or max-heap in step one

Efficient Sorting Algorithms (continued)

- Heap Sort (continued)
  - The pseudocode for this process is as follows:

```
heapsort(data[],n)
  transform data into a heap;
    for i = down to 2
    swap the root with the element in position i;
    restore the heap property for the tree
       data[0], . . . , data[i-1];
```

  - The construction of the heap uses the method developed by Floyd and described in Chapter 6
  - This is illustrated for the array `[2 8 6 1 10 15 3 12 11]` in Figure 9.9 on page 510
  - Once the heap is built, the second phase begins, which consists of taking the largest value, 15, and moving it to the end of the array

Efficient Sorting Algorithms (continued)

- Heap Sort (continued)
  - The value in the last location is swapped with the largest, causing a violation the heap property
  - So the heap is restored using the movedown()function (see section 6.9), omitting the last element of the array, which is now in place
  - This process continues until all the elements have been placed in their proper locations; it is illustrated in Figure 9.10 on page 511
  - The code for `heapsort` is shown below:

```
template<class T>
void heapsort(T data[], int n) {
  for (int i = n/2 - 1; i >= 0; --i) // create a heap;
    moveDown (data,i,n-1);
  for (int i = n-1; i >= 1; --i) {
    swap(data[0],data[i]); // move the largest item to data[i];
    moveDown(data,0,i-1);  // restore the heap property;
  }
}
```

Efficient Sorting Algorithms (continued)

- Quicksort
  - The process behind Shell sort was to divide the original array into subarrays, sort those, and then divide the partially sorted array into new subarrays to be sorted, until the entire array was in order
  - This was also the motivation behind ***quicksort***, developed by Sir Charles A. R. Hoare in 1960
  - Initially, the array is divided into two subarrays, one containing items less than or equal to a chosen item called the ***pivot*** or ***bound***, and the other containing elements larger than or equal to the pivot
  - This process is repeated on these two subarrays, creating four subarrays, and it is continued until we have subarrays of one element
  - Because the grouping of items separates them into smaller and larger, these one-element arrays do not need to be sorted at all, they are already arranged in order

Efficient Sorting Algorithms (continued)

- Quicksort (continued)
  - By nature of the partitioning process, quicksort is recursive; the pseudocode for this algorithm is as follows:

```
quicksort(array[])
  if length(array) > 1
    choose bound; // partition array into subarray1 and subarray2
    while there are elements left in array
      include element either in subarray1 = {el: el ≤ bound}
        or in subarray2 = {el: el ≥ bound};
    quicksort(subarray1);
    quicksort(subarray2);
```

  - Two operations need to be performed to partition the array: choosing a pivot, and moving the elements to the proper subarrays
  - Choosing the pivot is non-trivial; the goal is to have the two subarrays to be nearly equal in length

Efficient Sorting Algorithms (continued)

- Quicksort (continued)
  - Several strategies have been developed, but the one incorporated into the code in Figure 9.11 on page 514 simply chooses the item in the middle of the array
  - The pseudocode is vague regarding the second task, separating the elements into the subarrays
  - In particular it does not decide where to put items equal to the bound; we have only indicated it could be put with either list
  - The reasoning behind this is to attempt to keep the lists about the same length
  - Again, the details are in the implementation, and in this case in Figure 9.11
  - Another consideration in this code is the preprocessing that is carried out to locate the largest element and move it to the end of the array

Efficient Sorting Algorithms (continued)

- Quicksort (continued)
  - This is done to simplify the algorithm somewhat, and keep the value `lower` from running off the end of the array
  - The algorithm also uses the principal property of bound, that it is the boundary item between the two arrays, to place it in its final position once it is selected
  - Figure 9.12 on page 515 illustrates the process of partitioning the array for the array `[8 5 4 7 6 1 6 3 8 12 10]`
  - The first partitioning locates the largest element and exchanges it with the value in that position, so the last element no longer needs to be processed
  - This results in `first` = 1, `last` = 9, and the data in the first position is exchanged with the bound value in position 4, so the array becomes `[6 5 4 7 8 1 6 3 8 10 12]`
  - The remainder of the process is continued in Figure 9.12

Efficient Sorting Algorithms (continued)

- Quicksort (continued)
  - One the partitioning is complete, the process continues with the left and right subarrays, then for the subarrays of these subarrays
  - This continues until the subarrays have less than two elements
  - The entire sorting operation is shown in Figure 9.13 on page 517, which also shows the changes in the current arrays

Efficient Sorting Algorithms (continued)

- Mergesort
  - Quicksort's one major drawback is that it has a worst case $O(n^2)$ behavior due to the difficulty of the partitioning process
  - There are numerous techniques of choosing a bound that attempt to address this problem, however, there is no assurance that any approach will result in arrays that are equal in size
  - A different approach entirely simplifies the partitioning as much as can be and focuses on merging the sorted arrays
  - This is the idea behind ***mergesort***, one of the first computerized sorting algorithms, developed by John von Neumann in 1945
  - The key operation in mergesort is the merging of the sorted halves of the array into a single array
  - Of course, these halves must be sorted, which occurs by merging the sorted halves of these halves

Efficient Sorting Algorithms (continued)

- Mergesort (continued)
  - The process of splitting array into halves stops when each array has fewer than two items in it
  - Because of the similarity to the quicksort partitioning process, this can also be implemented recursively, as follows:

```
mergesort(data[])
  if data have at least two elements
    mergesort(left half of data);
    mergesort(right half of data);
    merge(both halves into a sorted list);
```

  - Merging the lists into a single list is also straightforward; the pseudocode for this is shown on the next slide

Efficient Sorting Algorithms (continued)

- Mergesort (continued)

```
merge(array1[], array2[], array3[])
  i1, i2, i3 are properly initialized;
  while both array2 and array3 contain elements
    if array2[i2] < array3[i3]
      array1[i1++] = array2[i2++];
    else array1[i1++] = array3[i3++];
  load into array1 the remaining elements of either array2
    or array3;
```

  - So if `array2 = [1 4 6 8 10]` and `array3 = [2 3 5 22]`, then the resulting `array1 = [1 2 3 4 5 6 8 10 22]`
  - Now, the pseudocode implies that the arrays are physically separate entities
  - However, for the code to work correctly, this is not the case

Efficient Sorting Algorithms (continued)

- Mergesort (continued)
  - The array is actually the concatenated form of the other two arrays, so before the merge, it looks like `[1 4 6 8 10 2 3 5 22]`
  - This creates problems for merging algorithm; for instance after the `while` loop iterates twice, `array2` is `[1 2 6 8 10]` and `array1` is `[1 2 6 8 10 2 3 5 22]`
  - Consequently, a temporary array is needed during the merging process
  - Once the merge is complete, the temporary array can be transferred back into `array1`
  - Since `array2` and `array3` are subarrays of `array1`, we don't need to pass them as parameters to the merge routine
  - Instead, we can pass indexes to the beginning and end of `array1`

Efficient Sorting Algorithms (continued)

- Mergesort (continued)
  - The revised pseudocode now looks like:

```
merge (array1[], first, last)
  mid = (first + last) / 2;
  i1 = 0;
  i2 = first;
  i3 = mid + 1;
  while both left and right subarrays of array1 contain
    elements
    if array1[i2] < array1[i3]
      temp[i1++] = array1[i2++];
    else temp[i1++] = array1[i3++];
  load into temp the remaining elements of array1;
  load to array1 the content of temp;
```

Efficient Sorting Algorithms (continued)

- Mergesort (continued)
  - The complete `array1` is copied to `temp`, then copied back to `array1`
  - So the number of moves when `merge()` executes is always 2 · (last – first + 1)
  - The number of comparisons depends on the ordering of `array1`
  - The pseudocode for the sort process is now:

```
mergesort (data[], first, last)
  if first < last
    mid = (first + last) / 2
    mergesort(data, first, mid);
    mergesort(data, mid+1, last);
    merge(data, first, last);
```

  - An example of this running is shown in Figure 9.14

Efficient Sorting Algorithms (continued)

- Mergesort (continued)

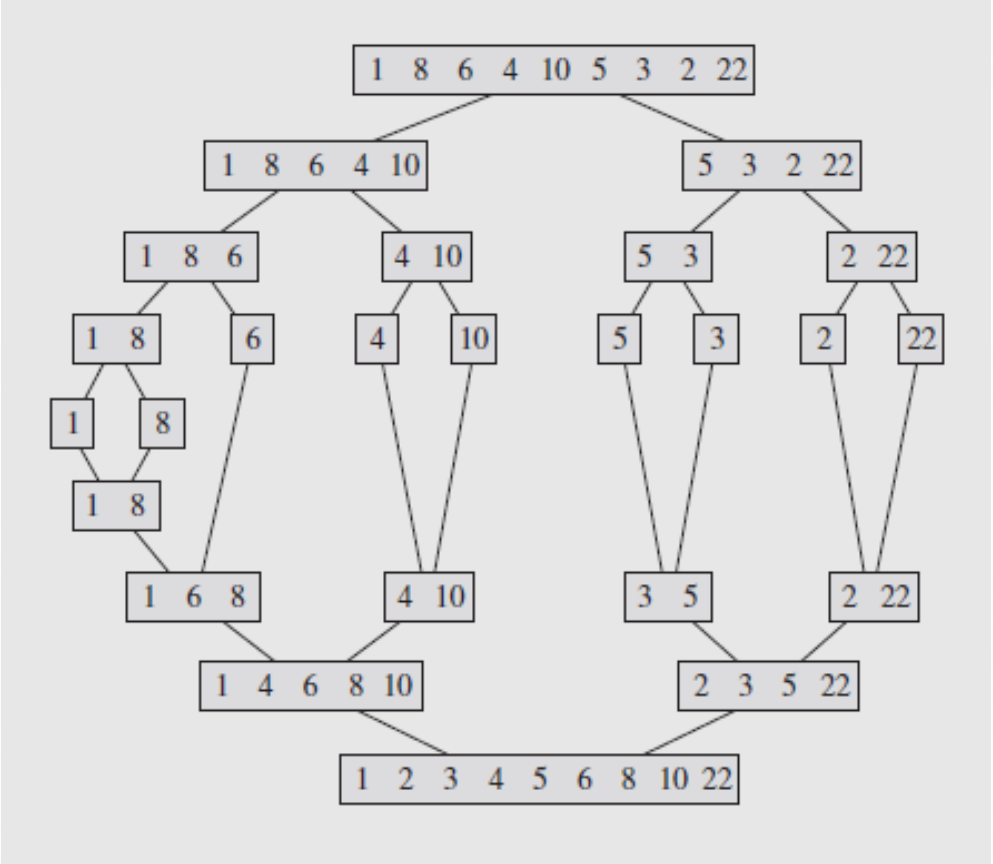


Fig. 9.14 The array [1 8 6 4 10 5 3 2 22] sorted by mergesort