RESONANCE

Introduction

1 Foundations
  1.1 Background
  1.2 Complexity
  1.3 Measurements
  1.4 Asymptotic Notation
  1.5 Recurrence Relations
  1.6 Assignment #1 (due We, Sept 29)
  1.7 Feasibility and Intractability
  1.8 Data Abstraction and Structure

Linear Structures

2 Lists
  2.1 Specification
  2.2 Array Representation
  2.3 One-way Linked Representation
  2.4 Circular One-way Linked Representation
  2.5 Doubly Linked Representation
3 Stacks
  3.1 Specification
  3.2 Array Representation
  3.3 One-way Linked Representation
4 Queues
  4.1 Specification
  4.2 Circular Array Representation
  4.3 Circular One-way Linked Representation
  4.4 Doubly Linked Representation

Trees

5 General Trees
5.1 Properties
5.2 Traversal
5.3 Representations
5.4 Demo Applets
5.5 Assignment #2 (due We, Oct 6)
6 Selection Trees
6.1 Modifying a Key
6.2 Initialization
6.3 Application: External Sort

Priority Queue Trees

7 Heaps
7.1 Adding an Element
7.2 Deleting an Element
7.3 Initialization
Brute Force
Efficient
7.4 Applications
7.5 Implementation
7.6 Demo Applets
8 Height-Biased Leftist Trees
8.1 Definitions
8.2 Merging Height-Biased Leftist Trees
Time complexity
8.3 Application: Priority Queues
Merge
Add
Delete
Initialization
8.4 Weight-Biased Leftist Trees
8.5 Assignment #3 (due We, Oct 13)

Search Trees

9 Binary Search Trees
  9.1 Characteristics
  9.2 Search
  9.3 Insertion
  9.4 Deletion
  9.5 Demo Applets
10 AVL Trees
  10.1 Definitions
  10.2 Height
  10.3 Rotations
  10.4 Insertions and Deletions
  10.5 Demo Applets
  10.6 Assignment #4 (due We, Oct 20)
11 Red-black Trees
  11.1 Properties
  11.2 Insertions
  11.3 Deletions
  11.4 Demo Applet
  11.5 Assignment #5 (due Fr, Oct 29)
12 Multi-way Trees
  12.1 Definition
  12.2 Insertions
  12.3 Deletions
13 B-Trees
  13.1 Definition
  13.2 Insertions
  13.3 Deletions
  13.4 Assignment #6 (due We, Nov 3)

Graphs

14 Graph Traversal
  14.1 Depth-First Traversal
  14.2 Breadth-First Traversal
  14.3 Representations of Graphs
    Adjacency Matrices
    Adjacency Lists

14.4 Demo Applets
15 Least Cost Spanning Trees
15.1 Prim s Algorithm
15.2 Kruskal s Algorithm
15.3 Union-Find Algorithm
15.4 Demo Applets and Resources
15.5 Assignment #7 (due We, Nov 10)

General Algorithms

16 Brute Force Algorithms
16.1 Sequential Search
16.2 Hamilton Circuits
17 Greedy Algorithms
17.1 Prim s Minimal Spanning Tree Algorithm
17.2 Kruskal s Minimal Spanning Tree Algorithm
17.3 Dijkstra s Single-Source Shortest Paths Algorithm
17.4 Coin Change
17.5 Egyptian Fractions
17.6 Map Coloring
17.7 Voting Districts
17.8 Vertex Cover
17.9 0/1 Knapsack
17.10 Demo Applets
18 Divide-and-Conquer Algorithms
18.1 Merge Sort
18.2 Quick Sort
18.3 Tiling with L-Grouped Tiles
18.4 Closest Pair
18.5 Strassen s Matrix Multiplication
Brute Force
Divide and Conquer
18.6 Assignment #8 (due We, Nov 17)

State Search Algorithms

19 Backtracking Algorithms
  19.1 Solution Spaces
  19.2 Traveling Salesperson
  19.3 The Queens Problem
  19.4 Convex Hull (Graham s Scan)
  19.5 Generating Permutations
  19.6 Demo Applets
20 Branch-and-Bound Algorithms
  20.1 Cost-Based Tree Traversal
  20.2 Mazes
  20.3 Traveling Salesperson Problem
  20.4 Job Scheduling
  20.5 Integer Linear Programming
21 Dynamic Programming Algorithms
  21.1 Fibonnaci Numbers
  21.2 0/1 Knapsack Problem
  21.3 All Pairs Shortest Paths (Floyd-Warshall)
  21.4 Demo Applets
  21.5 Assignment #9 (due We, Dec 1)

[front] [up]

Chapter 18 Divide-and-Conquer Algorithms

A divide-and-conquer algorithm

- Derives the output directly, for small instances
- Divides large instances to smaller ones, and (recursively) applies the algorithm on the smaller instances.
- Combines the solutions for the subinstances, to produce a solution for the original instance.

18.1 Merge Sort

Sets of cardinality greater than one are decomposed into two equal subsets, the algorithm is recursively invoked on the subsets, and the returned ordered subsets are merged to provide a sorted variant of the original set.

The time complexity of the algorithm satisfies the recurrence equation

$$T(n) = \begin{cases} 2T(n/2) + O(n) & \text{if } n > 1 \\ 0 & \text{if } n = 1 \end{cases}$$

whose solution is T(n) = O(n log n).

18.2 Quick Sort

Sets of cardinality greater than one are partitioned into two subsets, and then the algorithm is recursively invoked on the subsets. The partitioning uses a key from the set as a pivot. Values smaller than the pivot are sent to one subset, and values greater than the pivot are sent to the other subset.

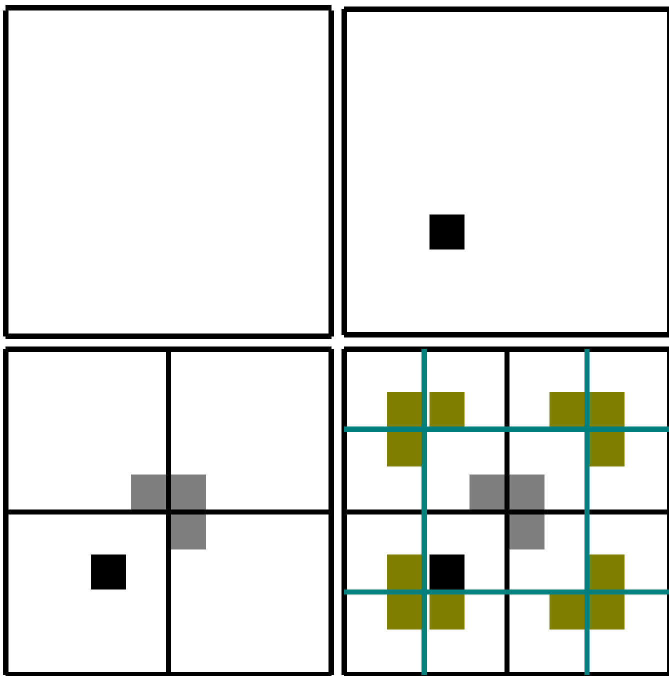
For randomly chosen pivots, the expected running time of the algorithm satisfies the recurrence equation

$$T(n) = \begin{cases} 2T(n/2) + O(n) & \text{if } n > 1 \\ 0 & \text{if } n = 1 \end{cases}$$

whose solution is T(n) = O(n log n). The worst case time complexity of the algorithm is $O(n^2)$.

18.3 Tiling with L-Grouped Tiles

The problem is to tile a board of size $2^k \times 2^k$ with one single tile and $2^{2k} - 1$ L-shaped groups of 3 tiles. A divide-and-conquer approach can recursively divide the board into four, and place a L-grouped set of 3 tiles in the center at the parts that have no extra tile.

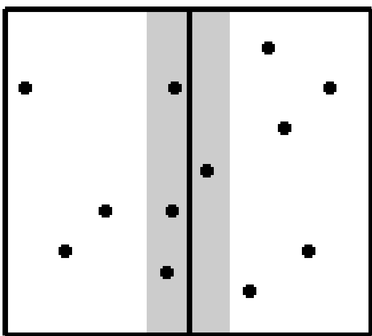


18.4 Closest Pair

Given a set of n points (x_i, y_i) the problem asks what is the distance between the two closest points. A brute force approach in which the distances for all the pairs are measured takes $O(n^2)$ time.

A divide-and-conquer algorithm can sort the points along the x-axis, partition the region into two parts R_{left} and R_{right} having equal number of points, recursively apply the algorithm on the sub-regions, and then derive the minimal distance in the original region.

The closest pair resides in the left region, the right region, or across the borderline. The last case needs to deal only with points at distance $\delta = \min(\delta_{left}, \delta_{right})$ from the dividing line, where δ_{right} and δ_{right} are the minimal distances for the left and right regions, respectively.



The points in the region around the boundary line are sorted along the y coordinate, and processed in that order. The processing consists of comparing each of these points with points that are ahead at most δ in their y coordinate. Since a window of size $\delta \times 2\delta$ can contain at most 6 points, at most five distances need to be evaluated for each of these points.

The sorting of the points along the x and y coordinates can be done before applying the recursive divide-and-conquer algorithm; they require $O(n \log n)$ time.

The processing of the points along the boundary line takes O(n) time. Hence, the recurrence equation for the time complexity of the algorithm:

$$T(n) = \begin{cases} 2T(n/2) + O(n) & \text{if } n > 1 \\ 0 & \text{if } n = 1 \end{cases}$$

The solution of the equation is T(n) = O(n log n).

18.5 Strassen s Matrix Multiplication

Given: Two N by N matrices A and B.

Problem: Compute C = A × B

Brute Force

```
for i:= 1 to N do
  for j:=1 to N do
     C[i,j] := 0;
     for k := 1 to N do
        C[i,j] := C[i,j] + A[i,k] * B[k,j]
```

O(N^3) multiplications

Divide and Conquer

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$P_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$P_2 = (A_{21} + A_{22})B_{11}$$

$$P_3 = A_{11}(B_{12} - B_{22})$$

$$P_4 = A_{22}(B_{21} - B_{11})$$

$$P_5 = (A_{11} + A_{12})B_{22}$$

$$P_6 = (A_{21} - A_{11})(B_{11} + B_{12})$$

$$P_7 = (A_{12} - A_{22})(B_{21} + B_{22})$$

$$C_{11} = P_1 + P_4 - P_5 + P_7$$

$$C_{12} = P_3 + P_5$$

$$C_{21} = P_2 + P_4$$

$$C_{22} = P_1 + P_3 - P_2 + P_6$$

From

$$T(n) = \begin{cases} 7T(n/2) + cn & \text{if } n > 1 \\ c & \text{if } n = 1 \end{cases}$$

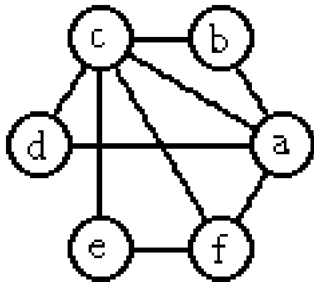
$T(n) = O(n^{\log 7}) = O(n^{2.81})$.

Best known upper bound is $n^{2.376}$

18.6 Assignment #8 (due We, Nov 17)

1. Show that our greedy algorithms for the Egyptian fractions and vertex cover problems need not provide optimal answers.

   An optimal vertex cover is one which uses a minimal number of vertices, and an optimal Egyptian number is one which uses a minimal number of terms.
2. Assume the greedy algorithm for the vertex cover problem uses a heap-based priority queue for selecting a node at each stage. Consider the following graph.

   i. Provide an initial priority queue for the algorithm on the above graph.
   ii. Show all the the transformations the priority queue goes through, in response to the selection of the first node by the greedy algorithm.
   iii. Find the time and space complexities of the algorithm on arbitrary graphs G = {V, E}.
3. Find out what is Huffman s compression algorithm, and explain what makes it a greedy algorithm.
4. Provide a divide-and-conquer algorithm for determining the largest and second largest values in a given unordered set of numbers. Provide a recurrence equation expressing the time complexity of the algorithm, and derive it s solution.

[prev] [prev-tail] [front] [up]

NAME

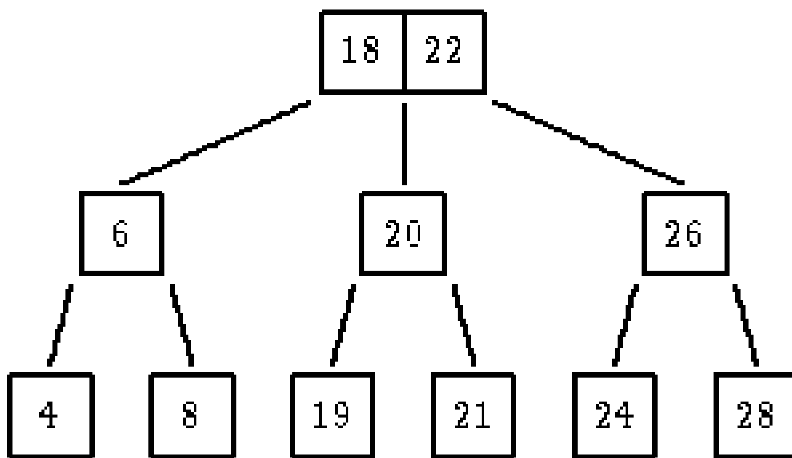
1-4 CHARACTERS CODE(if you want your final grade posted)

CIS 680: Final Exam

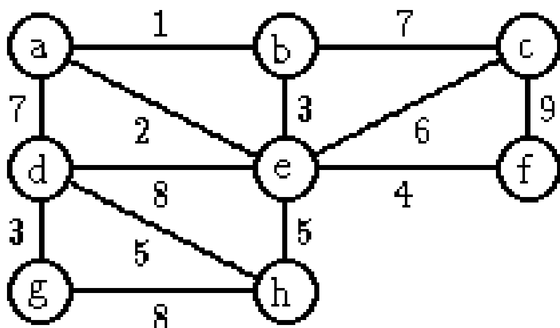
1:50 minutes
Open Notes, Open Books
The exam consists of five problems
Answers not appearing in designated spaces WILL NOT be graded

- **Problem #1** (10 points)

  Add the keys 6.1, 6.2, 6.3, 6.4, and 6.5 to the following B-tree of degree 3, and show all the intermediate trees and the final tree.



- **Problem #2** (10 points)
  a. Describe an algorithm to determine whether a directed graph has exactly one topological order.
  b. What is the time complexity of your algorithm?
- **Problem #3** (10 points) Show the linked lists of the union-find algorithm, at all the instances of applying Kruskal s algorithm on the following graph.



- **Problem #4** (10 points) The 3-vertex cover problem for graphs asks to assign a vertex cover of cardinality three to the given graphs. Assume an undirected graph whose edges are (1,2), (1,4), (1,5), (2,3), (2,4), (3,4), (4,5).
  a. Provide a spanning tree of a solution space, for the 3-vertex cover problem on the given

graph.

b. What vertex cover the backtracking algorithm provides for the above tree?

c. What vertex cover the FIFO branch-and-bound algorithm provides on the above tree?

- **Problem #5** (10 points) Consider the following recursive function.

```
function value(i) {
   if (i >= n) return 0;
   return 1 + max( value(2*i+1), value(2*i+2),
                   value(4*i+3) + value(4*i+4) + value(4*i+5),
                   value(4*i+5) + value(4*i+6)                  );
}
```

Translate the code into an iterative bottom-up dynamic programming algorithm.

NAME

CIS 680: Mid Term Exam

Mo, October 25, 1999, 50 minutes
Open Notes, Open Books
The exam consists of seven problems
Answers not appearing in designated spaces WILL NOT be graded

- **Problem #1** (10 points) What are the time and space complexities of the following code segments.

  (a)

  ```
  for i := 1 to n do
     for j := 1 to n^2 do
       for k := 1 to n^3do
         a[i,j,k] := 0
  ```

  (b)

  ```
  function comp(n)
  {
    if n < 2 then return 1
    return comp( n / 2 )
  }
  ```

- **Problem #2** (10 points) Assume n is an integer value divisible by 6 and larger than 1000.

  (a)

  Solve the following recurrence equation exactly.

  $T(n) = \begin{cases} 2T(n-3) & \text{if } n \geq 3 \\ 1 & \text{if } n < 3 \end{cases}$ (b)

  Does the solution to the following recurrence equation equal to, smaller than, or greater than the solution to the equation of (a).

  $T(n) = \begin{cases} 2T(n-2) & \text{if } n \geq 3 \\ 1 & \text{if } n < 3 \end{cases}$ ____________

- **Problem #3** (10 points) Which of the following statements is correct?

  a. Each AVL tree is also a balanced tree
  b. Each heap tree is also a height-biased leftist tree
  c. The ratio between the longest and shortest paths from a root to a leaf in a red-black tree is at most 2.
  d. Each height-biased leftist tree is also a binary search tree.
  e. An AVL tree with more than 3 keys can t be a binary search tree.

- **Problem #4** (10 points)

a. Let the height of a tree be the number of nodes in the longest path from the root to the leaves. Provide a recursive algorithm that when given a binary tree determines the height of the tree.

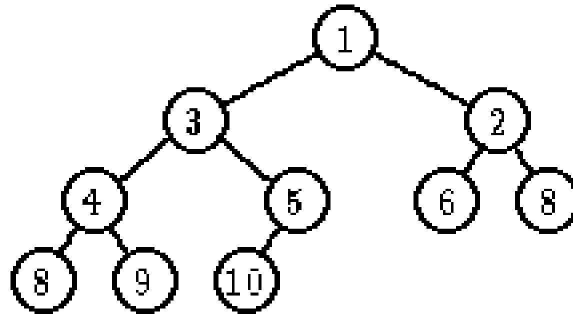
b. What is the time and space complexities of your algorithm?

c. What is the time and space complexities of your algorithm, if only AVL trees are allowed?

d. What is the time and space complexities of your algorithm, if only height-biased leftist trees are involved?

- **Problem #5** (10 points)

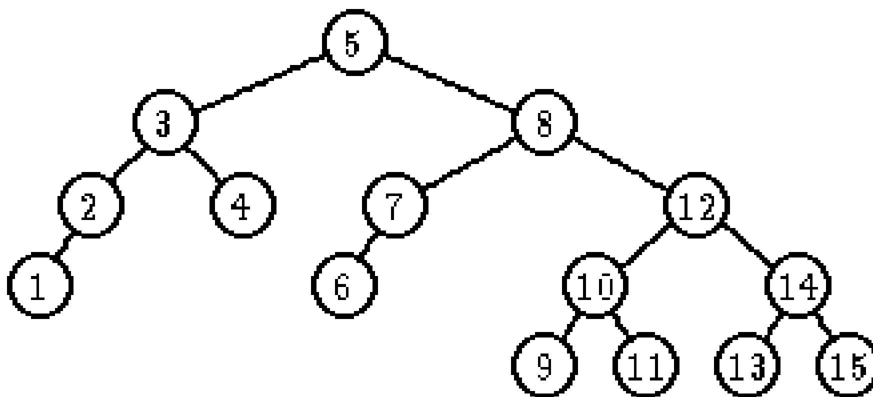
Consider the priority queue represented by the given height-biased leftist tree. Show the modified tree under each of the following operations. (Note: The two operations are independent. Each of them starts from the above tree.)



Insertion of the key 7.

Deletion of a key.

- **Problem #6** (10 points) Consider the following AVL tree.



Show the modified tree under each of the following operations. (Note: The two operations are independent. Each of them starts from the above tree.)

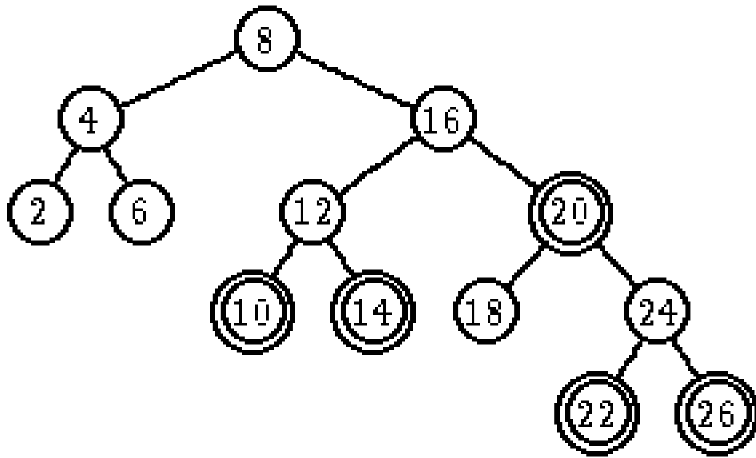
Deletion of the key 4

Insertion of the key 16.

- **Problem #7** (10 points)

Consider the following red-black tree (the red nodes are double circled). What kind of discrepancy

(Lb0, RRb, etc.) each of the following operations create. (Note: The different operations are independent. Each of them starts from the above tree.)



a. Addition of the key 11.
b. Addition of the key 13.
c. Deletion of the key 20.
d. Deletion of the key 12.
e. Deletion of the subtree rooted at 4.

grades

Introduction

1 Foundations
- 1.1 Background
- 1.2 Complexity
- 1.3 Measurements
- 1.4 Asymptotic Notation
- 1.5 Recurrence Relations
- 1.6 Assignment #1 (due We, Sept 29)
- 1.7 Feasibility and Intractability
- 1.8 Data Abstraction and Structure

Linear Structures

2 Lists
- 2.1 Specification
- 2.2 Array Representation
- 2.3 One-way Linked Representation
- 2.4 Circular One-way Linked Representation
- 2.5 Doubly Linked Representation

3 Stacks
- 3.1 Specification
- 3.2 Array Representation
- 3.3 One-way Linked Representation

4 Queues
- 4.1 Specification
- 4.2 Circular Array Representation
- 4.3 Circular One-way Linked Representation
- 4.4 Doubly Linked Representation

Trees

5 General Trees
- 5.1 Properties
- 5.2 Traversal
- 5.3 Representations
- 5.4 Demo Applets
- 5.5 Assignment #2 (due We, Oct 6)

6 Selection Trees
- 6.1 Modifying a Key
- 6.2 Initialization
- 6.3 Application: External Sort

Priority Queue Trees

7 Heaps

7.1 Adding an Element
7.2 Deleting an Element
7.3 Initialization
7.4 Applications
7.5 Implementation
7.6 Demo Applets
8 Height-Biased Leftist Trees
8.1 Definitions
8.2 Merging Height-Biased Leftist Trees
8.3 Application: Priority Queues
8.4 Weight-Biased Leftist Trees
8.5 Assignment #3 (due We, Oct 13)

Search Trees

9 Binary Search Trees
9.1 Characteristics
9.2 Search
9.3 Insertion
9.4 Deletion
9.5 Demo Applets
10 AVL Trees
10.1 Definitions
10.2 Height
10.3 Rotations
10.4 Insertions and Deletions
10.5 Demo Applets
10.6 Assignment #4 (due We, Oct 20)
11 Red-black Trees
11.1 Properties
11.2 Insertions
11.3 Deletions
11.4 Demo Applet
11.5 Assignment #5 (due Fr, Oct 29)
12 Multi-way Trees
12.1 Definition
12.2 Insertions
12.3 Deletions
13 B-Trees
13.1 Definition
13.2 Insertions
13.3 Deletions

13.4 Assignment #6 (due We, Nov 3)

Graphs

14 Graph Traversal
14.1 Depth-First Traversal
14.2 Breadth-First Traversal
14.3 Representations of Graphs
14.4 Demo Applets
15 Least Cost Spanning Trees
15.1 Prim s Algorithm
15.2 Kruskal s Algorithm
15.3 Union-Find Algorithm
15.4 Demo Applets and Resources
15.5 Assignment #7 (due We, Nov 10)

General Algorithms

16 Brute Force Algorithms
16.1 Sequential Search
16.2 Hamilton Circuits
17 Greedy Algorithms
17.1 Prim s Minimal Spanning Tree Algorithm
17.2 Kruskal s Minimal Spanning Tree Algorithm
17.3 Dijkstra s Single-Source Shortest Paths Algorithm
17.4 Coin Change
17.5 Egyptian Fractions
17.6 Map Coloring
17.7 Voting Districts
17.8 Vertex Cover
17.9 0/1 Knapsack
17.10 Demo Applets
18 Divide-and-Conquer Algorithms
18.1 Merge Sort
18.2 Quick Sort
18.3 Tiling with L-Grouped Tiles
18.4 Closest Pair
18.5 Strassen s Matrix Multiplication
18.6 Assignment #8 (due We, Nov 17)

State Search Algorithms

19 Backtracking Algorithms
19.1 Solution Spaces
19.2 Traveling Salesperson

19.3 The Queens Problem
19.4 Convex Hull (Graham s Scan)
19.5 Generating Permutations
19.6 Demo Applets
20 Branch-and-Bound Algorithms
20.1 Cost-Based Tree Traversal
20.2 Mazes
20.3 Traveling Salesperson Problem
20.4 Job Scheduling
20.5 Integer Linear Programming
21 Dynamic Programming Algorithms
21.1 Fibonnaci Numbers
21.2 0/1 Knapsack Problem
21.3 All Pairs Shortest Paths (Floyd-Warshall)
21.4 Demo Applets
21.5 Assignment #9 (due We, Dec 1)

1. 70
2. 66
3. 66
4. 64
5. 63
6. 61
7. 61
8. 60
9. 58
10. 57
11. 57
12. 57
13. 56
14. 55
15. 54
16. 54
17. 52
18. 52
19. 51
20. 50
21. 50
22. 50
23. 49
24. 49
25. 49
26. 48
27. 48
28. 48
29. 48
30. 47
31. 47
32. 44
33. 43
34. 43
35. 42
36. 42
37. 42

38. 41
39. 41
40. 41
41. 38
42. 38
43. 38
44. 37
45. 36
46. 35
47. 34
48. 34
49. 33
50. 31
51. 30
52. 30
53. 28
54. 23

NAME

1-4 CHARACTERS CODE(if you want your final grade posted)

CIS 680: Final Exam

1:50 minutes
Open Notes, Open Books
The exam consists of five problems
Answers not appearing in designated spaces WILL NOT be graded

- **Problem #1** (10 points) In Kruskal s algorithm, the union-find algorithm merges the longer linked lists at the end of the shorter linked lists. The algorithm requires O(|V | log |V |) time. What contribution the union-find algorithm provides to the time complexity of Kruskel s algorithm, if the union operation merges the lists in arbitrary manner, without taking into account their lengths or other ranks.
- **Problem #2** (10 points) Consider the closest pair algorithm. How many distances the algorithm computes, in each of the following cases.
    a. The points are at locations (0,0), (1,0), (2,0), (3,0), (4,0), (5,0), (6,0), (7,0).
    b. The points are at locations (0,0), (1,0), (2,0), (3,0), (0,1), (1,1), (2,1), (3,1).
- **Problem #3** (10 points) Consider the Graham s Scan algorithm for solving the Convex Hull problem. For each of the following cases, draw a set of 8 points satisfying the specified condition.
    a. The algorithm constructs the convex hull without ever backtracking.
    b. The algorithm constructs a convex hull containing a minimal number of boundary points
    c. The algorithm constructs a convex hull containing 5 boundary nodes, and doesn t visit the omitted nodes consecutively.
- **Problem #4** (10 points) For each of the following problems, provide a tree generator of a solution space.
    a. The **assignment problem** assumes n jobs to be assigned to n workers, and a function c(i, j) which specifies the cost of assigning job i to worker j. The problem is interested in finding an assignment having a minimal cost.
    b. The **partition problem** assumes a set of items, and a function v(i) assigning to each item a value. The problem asks whether the set can be partitioned into two subsets having the same total value.
- **Problem #5** (10 points) The number of ways of choosing k items from a set of n items is represented by the following recurrence relation.

  $C(n, k) = C(n - 1, k - 1) + C(n - 1, k)$

    a. Provide a dynamic programming algorithm for calculating C(n, k).
    b. What calculations the dynamic programming algorithm saves in computing C(5, 2)

course grades

1. A 0979
2. A 1501
3. A amst
4. A bile
5. A guno
6. A notr
7. A tito
8. A w1b9
9. A- rs77
10. A- blud
11. A- bnl
12. A- xxx
13. B 1563test
14. B 6740
15. B 0424
16. B 3035
17. B 6876
18. B 7547
19. B 7604
20. B illb
21. B jojo
22. B xxx
23. B xxxx
24. B+ 3125
25. B+ br88
26. B+ slim
27. B+ xxxx
28. B- irot
29. B- sk88
30. B- trj
31. B- xxxx
32. C wing
33. C car
34. C slrp
35. C xxxx
36. C 9508
37. C iawa

38. C+ cn
39. C+ dive
40. C+ tuba
41. C- 8067
42. C- bass
43. C- rgb
44. C- soh
45. C- xxxx
46. C- 1323

- Terms and Definitions
- A Cookbook (Niemann)
- The Combinatorial Object Server
- Animations (Max-Planck-Institut fur Informatik)
- Intro to alg and data struc (Brown U)
- Cawsey s course notes
- Holte s course notes
- Gasch s Algorithm Archive
- Wessels s course notes

We choose to count just the comparison operation for keys.

We can take the instance characteristics to be the size of our data base.

```
Best time for successful search:        1
Best time for unsuccessful search:      n
Worst time for (un)successful search:   n
```

Average successful search, assuming all keys have equal probability to appear

$${1\over n}\sum _{i=1}^n i = (n+1)/2$$

Average unsuccessful search: n

Hence, best and worth time are n, and the average approaches this value.

We can go down to $n/2$ by assuming sorted lists

more

Time: Unordered: $O(n) = n$, $\Omega(n) = n$, $\Theta(n) = n$

Ordered: $O(n) = n$, $\Omega(n) = n$, $\Theta(n) = n$

Space:

$O(1)$

Time: $O(n) = \log n$, $\Omega(n) = \log n$, $\Theta(n) = \log n$

Space:

$O(1)$

NAME

CIS 680: Mid Term Exam

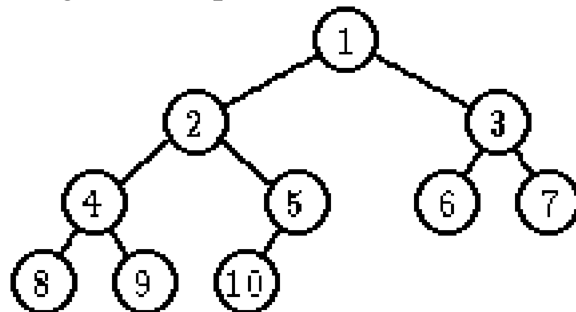
Mo, May 3, 1999, 50 minutes
Open Notes, Open Books
The exam consists of seven problems
Answers not appearing in designated spaces WILL NOT be graded

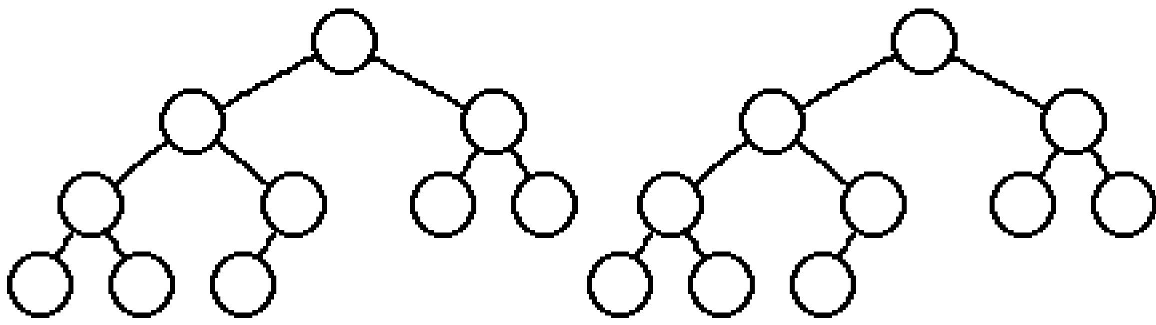
- **Problem #1** (10 points)
  a. What are the time and space complexities of the following code segment.
```
read( n )
for i = 1 to n do
  for j = 1 to i do   write( i, j )
```
  b. Solve the following reccurence equation exactly.

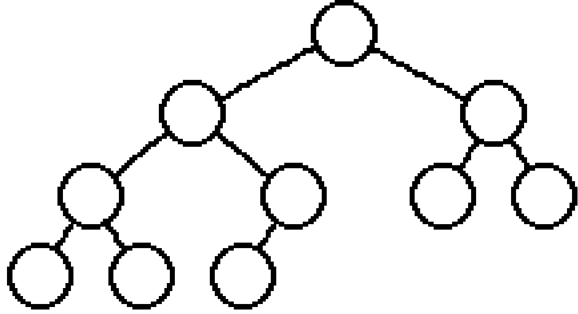
$$T(n) = \begin{cases} 3T(n-2) & \text{if } n \geq 2 \\ 1 & \text{if } n < 2 \end{cases}$$

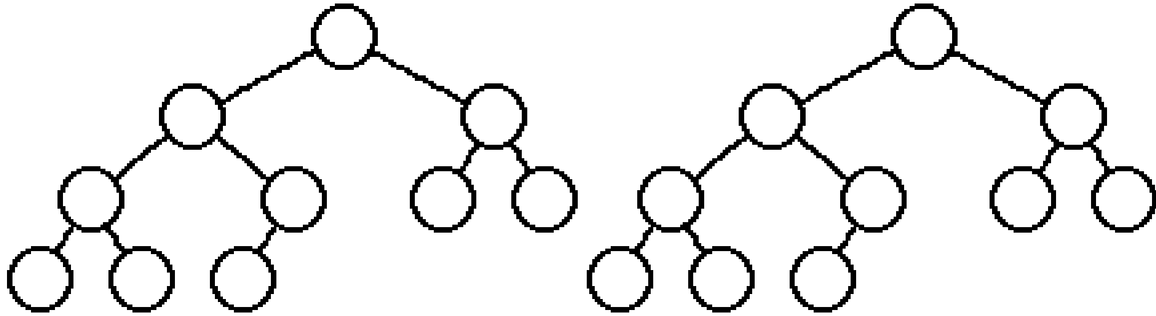
- **Problem #2** (10 points)
  a. Provide a recursive algorithm that given a binary tree determines the number of leaves in the tree.
  b. What is the time and space complexities of your algorithm?
- **Problem #3** (10 points) Let the height of a tree be the number of nodes in the longest path from the root to the leaves. For each of the following types of trees, determine the minimum number of nodes a tree of height 4 can have.
  a. Heaps
  b. Height-biased leftist trees
  c. Binary search trees
  d. AVL trees
  e. Red-black trees
- **Problem #4** (10 points)

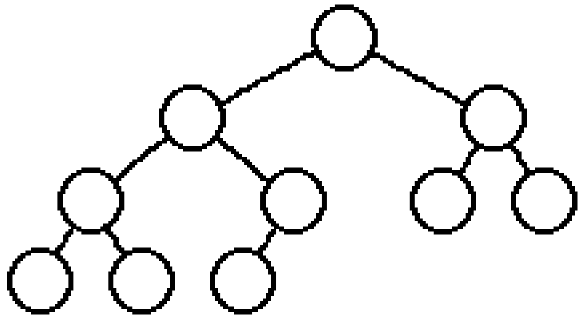
  Consider the linear time algorithm for initializing max heaps. Show all the intermediate, and final, trees the algorithm creates on the given tree.

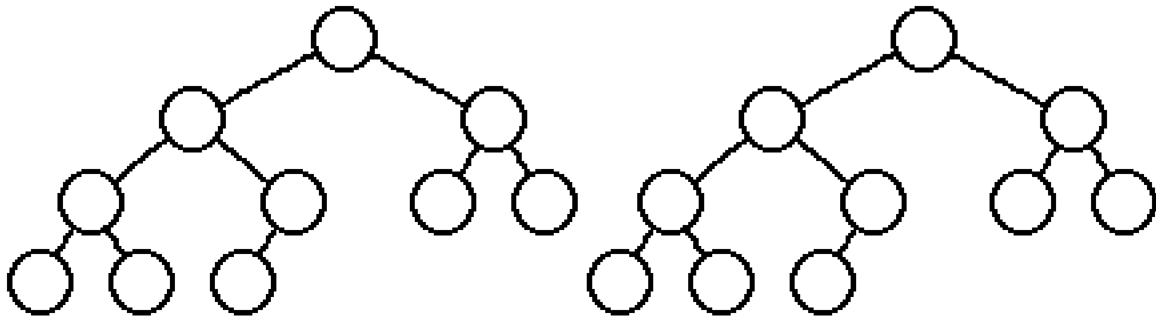


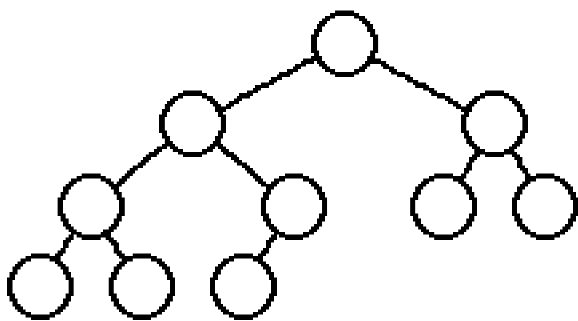






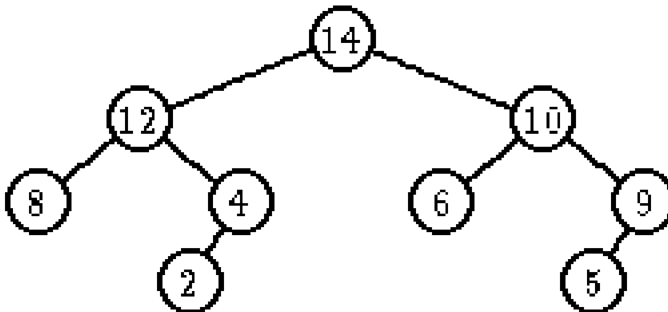






- **Problem #5** (10 points)

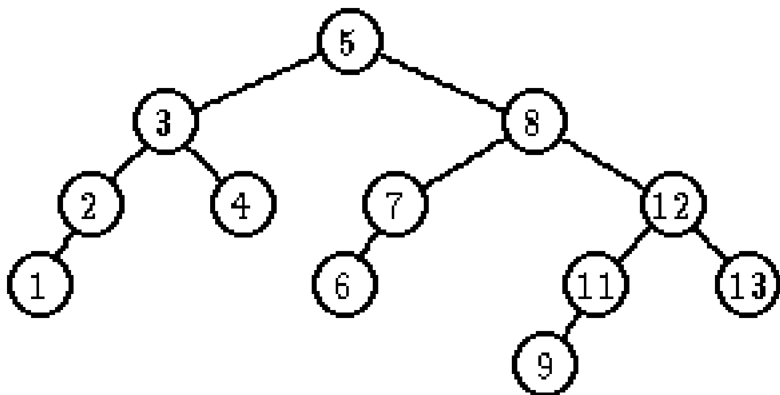
  Consider the priority queue represented by the given height-biased leftist tree. Show the modified tree under each of the following operations. (Note: The two operations are independent. Each of them starts from the above tree.)

  Insertion of the key 7.

  Deletion of the largest key.

- **Problem #6** (10 points) Consider the following AVL tree.

  Show the modified tree under each of the following operations. (Note: The two operations are independent. Each of them starts from the above tree.)

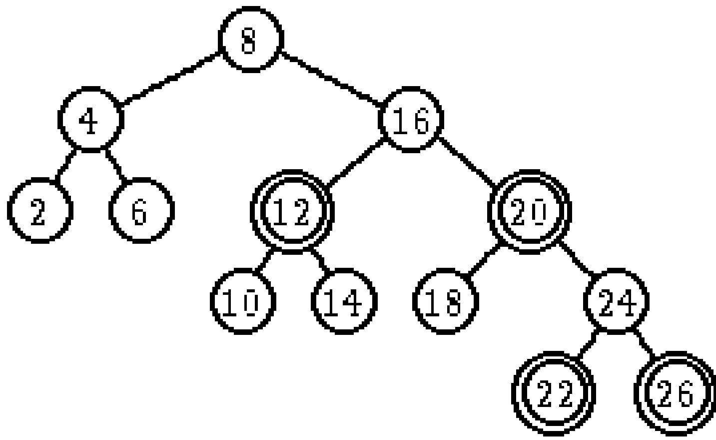
  Deletion of the key 5

  Insertion of the key 10.

- **Problem #7** (10 points)

  Consider the following red-black tree (the red nodes are double circled). What kind of discrepency

(Lb0, RRb, etc.) each of the following operations create.



a. Addission of the key 21.
b. Addission of the key 25.
c. Deletion of the key 18.
d. Deletion of the key 2.
e. Deletion of the subtree rooted at 12.

[tail] [up]

Chapter 1 Foundations

1.1 Background

Programs: solutions to **problems** expressed as **sequences of instructions** that manipulate **data**.

Issues: correctness, maintainability, and efficiency

Tools:

Abstractions
: Reduce complexity of problems.

Algorithms
: Taming complexity in processes

Data structures
: Making data easier to approach

Example: Multiplication of integers in unary and binary representations

1.2 Complexity

```
#include <iostream.h>
#include <time.h>
main(){       int start, finish, i;
  start = clock();
  ........body.........
  finish = clock();
  cout << (finish - start) / CLK_TCK ;
}
```

The `start = clock();` command might be problematic on a shared system. The shell command `time job` should be more appropriate.

A body of `i=1000000; while(i-);` gives a program of constant time complexity.

A body of `read i; while(i-);` gives a program of linear time complexity.

Resources:

Time
: T(P) = constant + T(instance characteristics)

Space
: S(P) = constant + S(instance characteristics)

1.3 Measurements

Example Sequential search in a set of distinct numbers.

```
get key
for i=1 to n do
  if key == a[i] then exit
```

[more]

- Measurements: experimental and analytical.
  - Best
  - Worst
  - Average/expected (difficult to determine)

$$\sum_I Pr(I) \cdot count_for(I)$$

- Analytical measurements: require an identification of key operations that contribute the most to the complexity.
- Asymptotic behavior: We care about growth behavior of complexity functions, not about constants.

1.4 Asymptotic Notation

Big Oh (O)-Upper Bound

$f(n) = O(g(n))$ iff $f(n) \leq cg(n)$

for some constants c and n_0, and all $n \geq n_0$

Omega (Ω)-Lower Bound

$f(n) = \Omega(g(n))$ iff $f(n) \geq cg(n)$

for some constants c and n_0, and all $n \geq n_0$

Theta (Θ)-Two-way Bound

$f(n) = \Theta(g(n))$ iff $c_1 g(n) \leq f(n) \leq c_2 g(n)$

for some constants c_1, c_2, and n_0, and all $n \geq n_0$

Little Oh (o)-Only Upper Bound

$f(n) = o(g(n))$ iff

$f(n) = O(g(n))$ and $f(n) \neq \Omega(g(n))$

Example: Take a program that reads an input, character by character, and copies that information into the output. The program will take O(n) time to process all the characters, and O(1) space for its variables. Specifically,

```
var x
while( not end-of-input ){
  read x
  write  x
}
```

Example: Take a program that reads an input, character by character, stores it into a stack, and upon reaching the end of the input pops the content of the stack into the output. The program will take O(n) time to process the characters, and O(n) space for the stack. Specifically,

```
var x, i, A[]
i = 0
while( not end-of-input ){
  read x
  A[i] := x
  i := i + 1
}
while( i > 0 ){
  i := i - 1
  write A[i]
}
```

Example: Sequential search in a set of distinct numbers.

```
get key
for i=1 to n do
  if key == a[i] then exit
```

[more]

Example: Binary search in an ordered set of distinct numbers.

[more]

1.5 Recurrence Relations

Sequential search: $T(n) = \begin{cases} T(n-1) + a & \text{if } n > 1 \\ b & \text{if } n = 1 \end{cases}$

T(n) = a(n - 1) + b = an + (b - a) = O(n)

Binary search: $T(n) = \begin{cases} T(\lceil \frac{n}{2} \rceil) + a & \text{if } n > 1 \\ b & \text{if } n = 1 \end{cases}$

T(n) = a log n + b = O(log n)

1.6 Assignment #1 (due We, Sept 29)

a. Find the exact solution for the following recurrence equation.

$$T(n) = \begin{cases} 3T(n/2) + n & \text{if } n > 0 \\ 0 & \text{otherwise} \end{cases}$$

b. Analyze the running time of the following code segment.

```
sum = 0
for i=1 to N do
  for j=1 to i do
    sum = sum + i
  end
end
```

c. Write programs of time complexity $\Theta(1)$, $\Theta(n)$, and $\Theta(f(n))$ for a function f(n) of your choice. Provide time measurements for execution times of the programs, and derive the coefficients for the time complexity functions.

1.7 Feasibility and Intractability

Common functions: O(1)-constant; O(log n)-logarithmic; O(n)-linear; $O(n^k)$-polynomial; $O(2^n)$-exponential

Theoretical: Feasible--polynomial time; Infeasible--exponential time.

Practical feasibility: only low degree polynomials

Example: the traveling salesman problem is intractable.

1.8 Data Abstraction and Structure

Data Objects

A set of instances or values.

Examples: digits, integers, letters, strings.

Data objects can be primitive or composite elements.

Data structure

A data object together with the relationships that exist among the instances that compose the object.

Abstract data type

Data objects and a set of operations for acting on the objects. Abstract data types are independent of any representation we might have in mind.

[front] [up]

[next] [prev] [prev-tail] [tail] [up]

Chapter 10
AVL Trees

10.1 Definitions

Named after Adelson, Velskii, and Landis.

Trees of height O(log n) are said to be **balanced**. AVL trees consist of a special case in which the subtrees of each node differ by at most 1 in their height.

Balanced trees can be used to search, insert, and delete arbitrary keys in O(log n) time. In contrast, height-biased leftist trees rely on non-balanced trees to speed-up insertions and deletions in priority queues.

10.2 Height

Claim: AVL trees are balanced.

Proof. Let N_h denote the number of nodes in an AVL tree of depth h

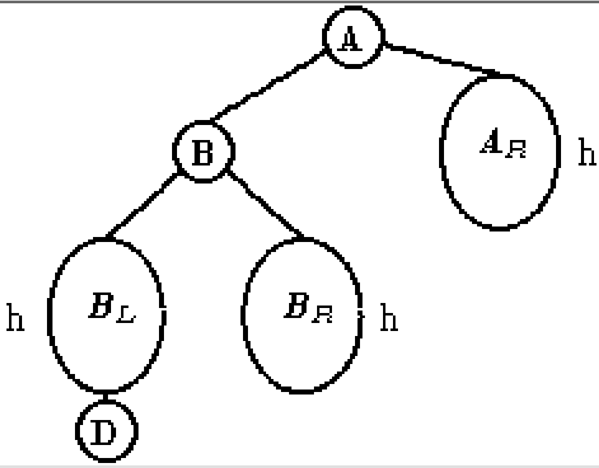
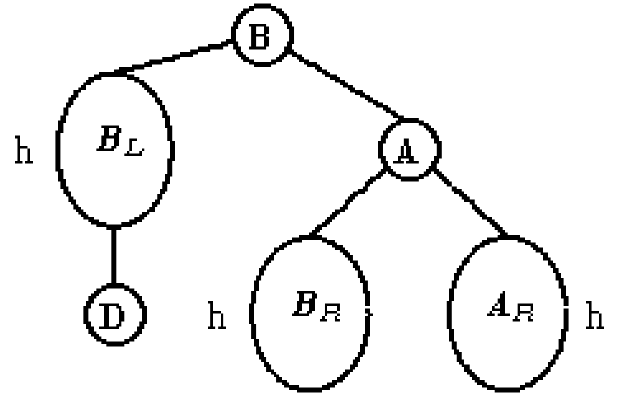
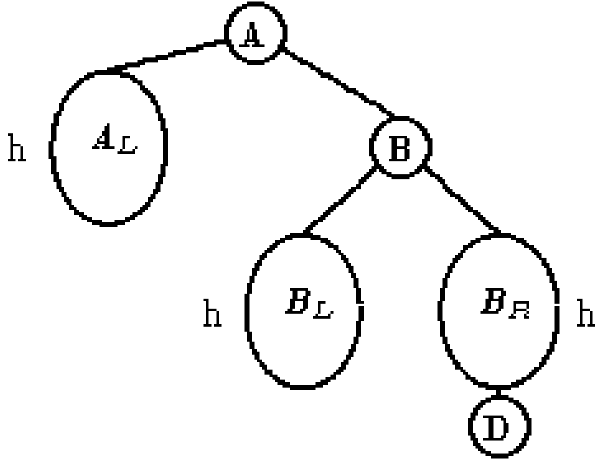
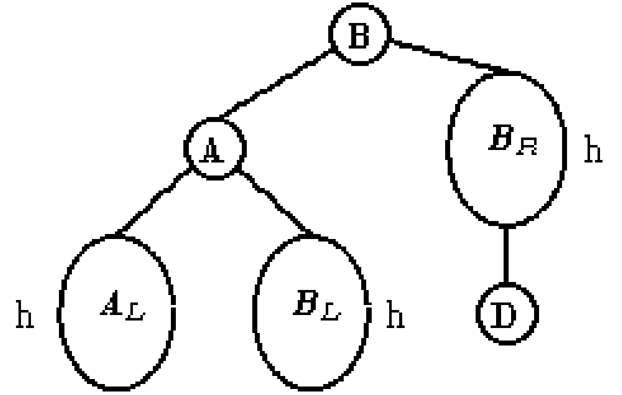
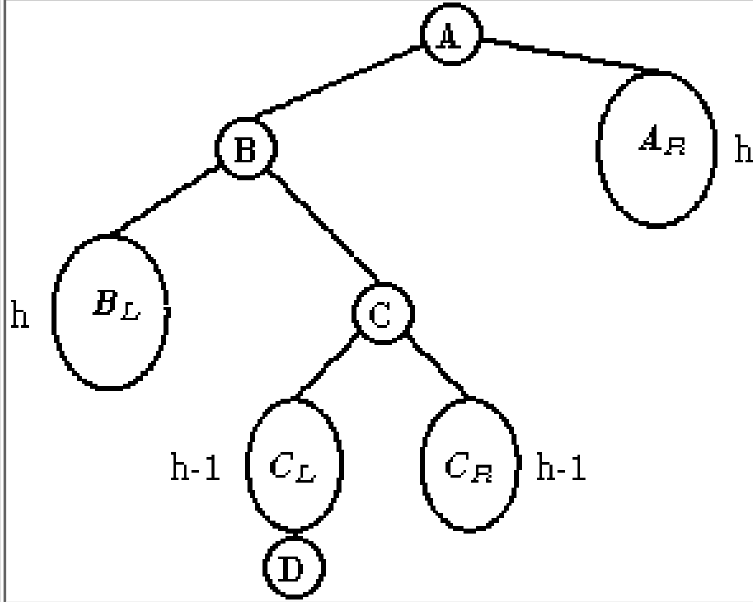
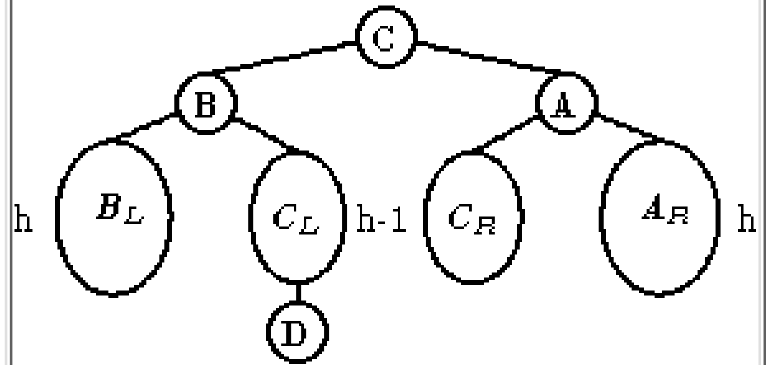
$$\begin{aligned} N_h &\geq N_{h-1} + N_{h-2} + 1 \\ &\geq 2N_{h-2} + 1 \\ &\geq 1 + 2(1 + 2N_{h-4}) \\ &= 1 + 2 + 2^2 N_{h-4} \\ &\geq 1 + 2 + 2^2 + 2^3 N_{h-6} \\ &\dots \\ &\geq 1 + 2 + 2^2 + 2^3 + \dots + 2^{h/2} \\ &= 2^{h/2} - 1 \end{aligned}$$

Hence,

$$\begin{aligned} 2^{h/2} - 1 &\leq n \\ h/2 &\leq \log_2(n + 1) \\ h &\leq 2 \log_2(n + 1) \end{aligned}$$

A more careful analysis, based on Fibonacci numbers theory, implies the tighter bound of $1.44 \log_2(n + 2)$.

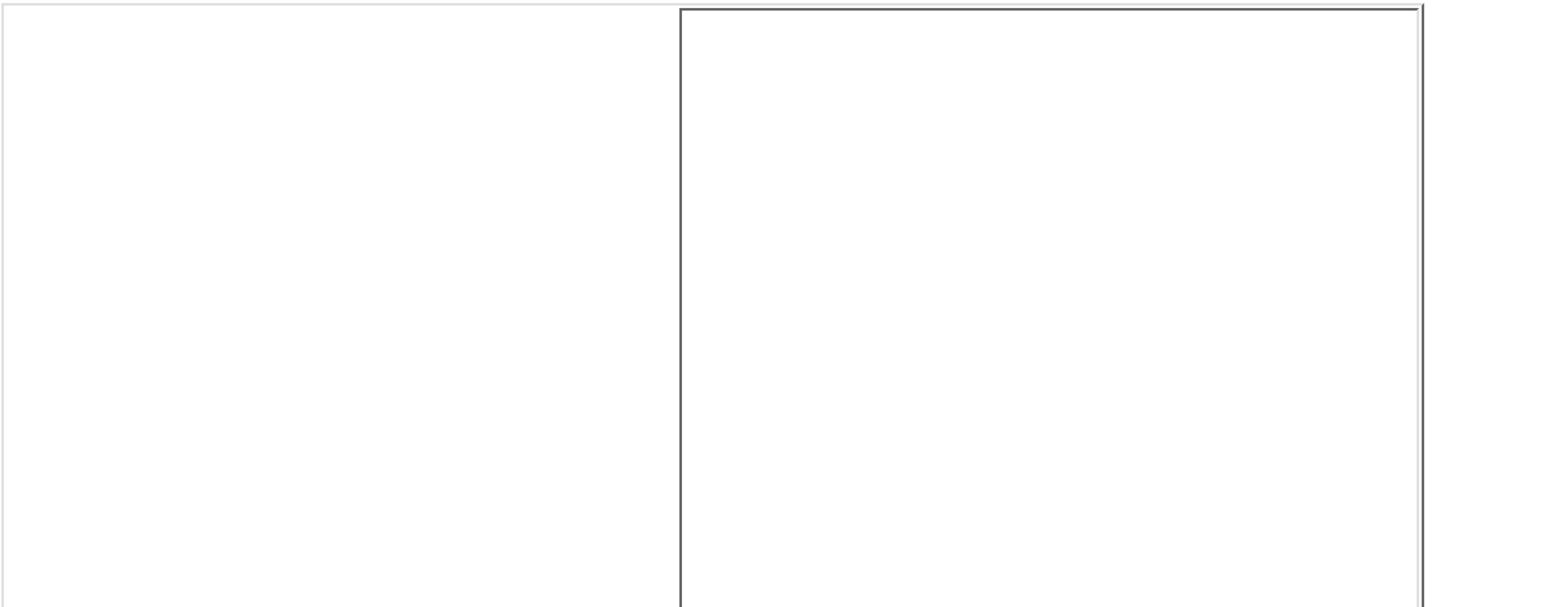
10.3 Rotations

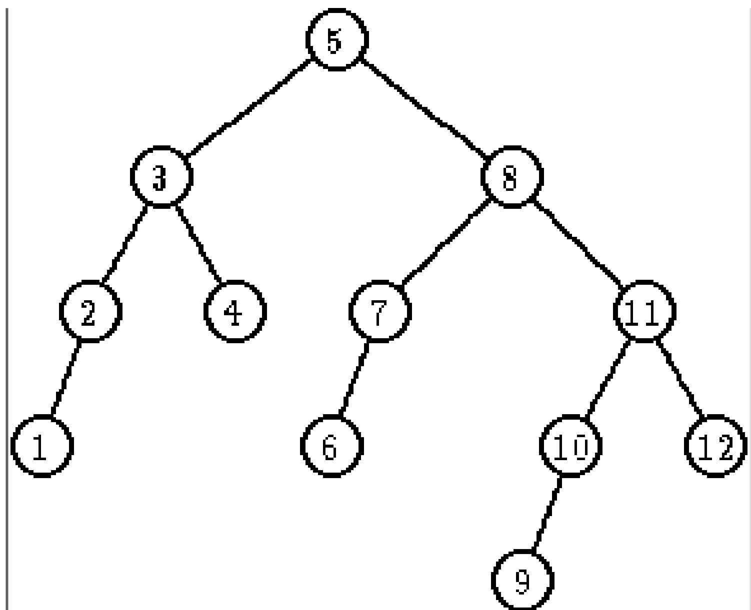
LL		
RR		
LR		

RL		
LL & LR ⇒ LL		

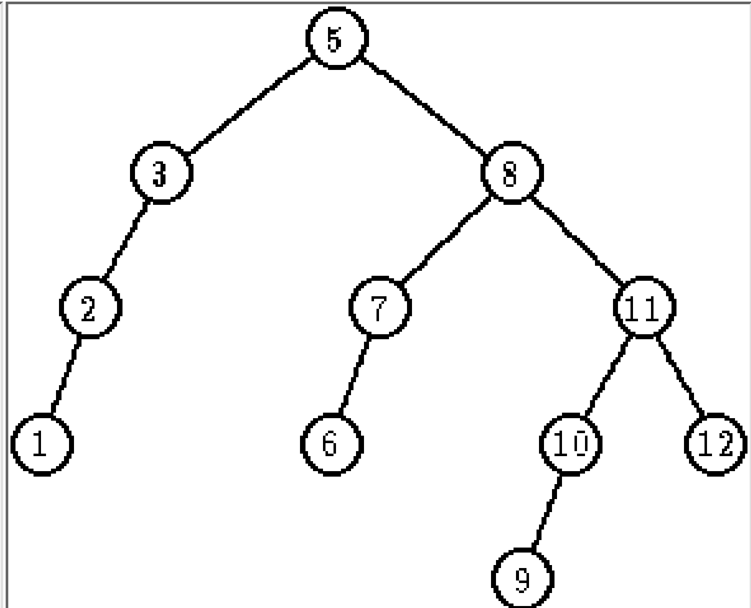
10.4 Insertions and Deletions

Insertions and deletions are performed as in binary search trees, and followed by rotations to correct imbalances in the outcome trees. In the case of insertions, one rotation is sufficient. In the case of deletions, O(log n) rotations at most are needed from the first point of discrepancy going up toward the root.

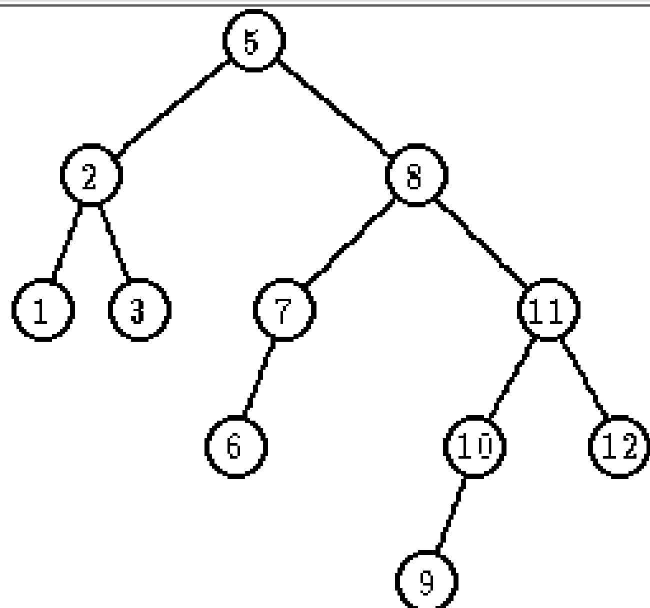


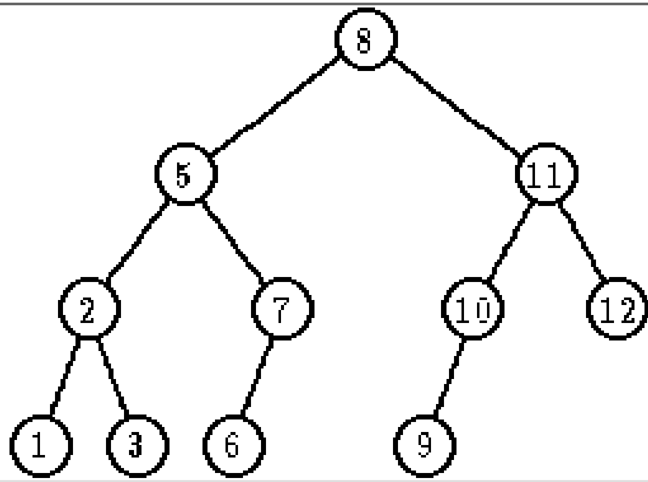


Delete 4



Imbalance at 3 implies a LL rotation with 2



Imbalance at 5 implies a RR rotation with 8 .	

10.5 Demo Applets

- demo of AVL trees
- demo of AVL trees

10.6 Assignment #4 (due We, Oct 20)

1. Provide an algorithm that, when given a binary search tree, removes in constant space all the nodes from the tree, in ascending order of keys. How much time your algorithm requires? Show the time and space analysis of your algorithm.
2. Provide an algorithm that, when given a binary search tree, removes in linear time all the nodes from the tree, in ascending order of keys. How much space your algorithm requires? Show the time and space analysis of your algorithm.
3. Construct a binary search tree by introducing the following keys in the given order: 1, 2, 7, 6, 3, 4, 5. Then repeatedly use AVL rotations to transform the tree into an AVL tree, while showing all the intermediate trees being created in the process. In each stage, the AVL transformation should be conducted at a discrepancy that is farthest from the root.

[next] [prev] [prev-tail] [front] [up]

[next] [prev] [prev-tail] [tail] [up]

Chapter 11
Red-black Trees

11.1 Properties

A binary search tree in which

- The root is colored black
- All the paths from the root to the leaves agree on the number of black nodes
- No path from the root to a leaf may contain two consecutive nodes colored red

Empty subtrees of a node are treated as subtrees with roots of black color.

The relation $n \geq 2^{h/2} - 1$ implies the bound $h \leq 2 \log_2(n + 1)$.

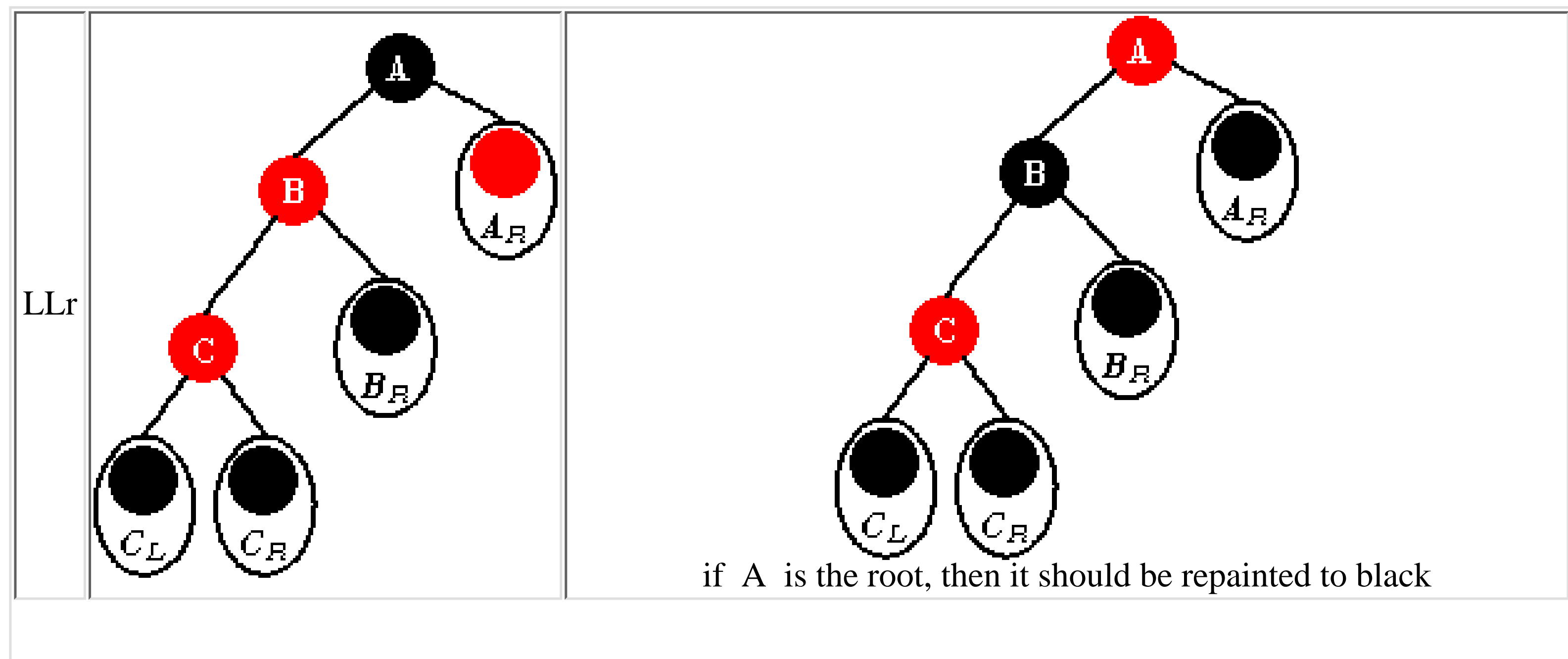
11.2 Insertions

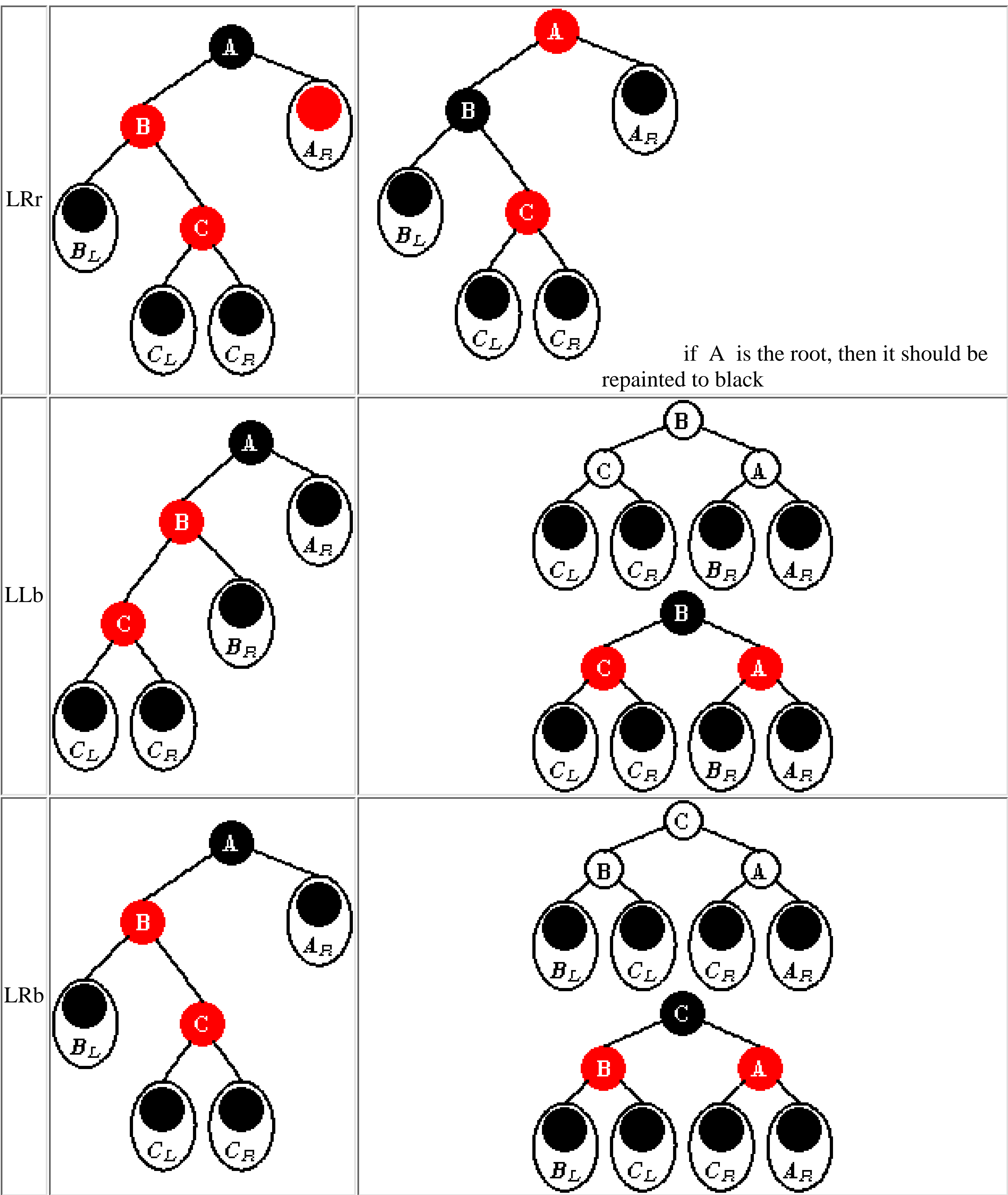
- Insert the new node the way it is done in binary search trees
- Color the node red
- If a discrepancy arises for the red-black tree, fix the tree according to the type of discrepancy.

A discrepancy can result from a parent and a child both having a red color. The type of discrepancy is determined by the location of the node with respect to its grand parent, and the color of the sibling of the parent.

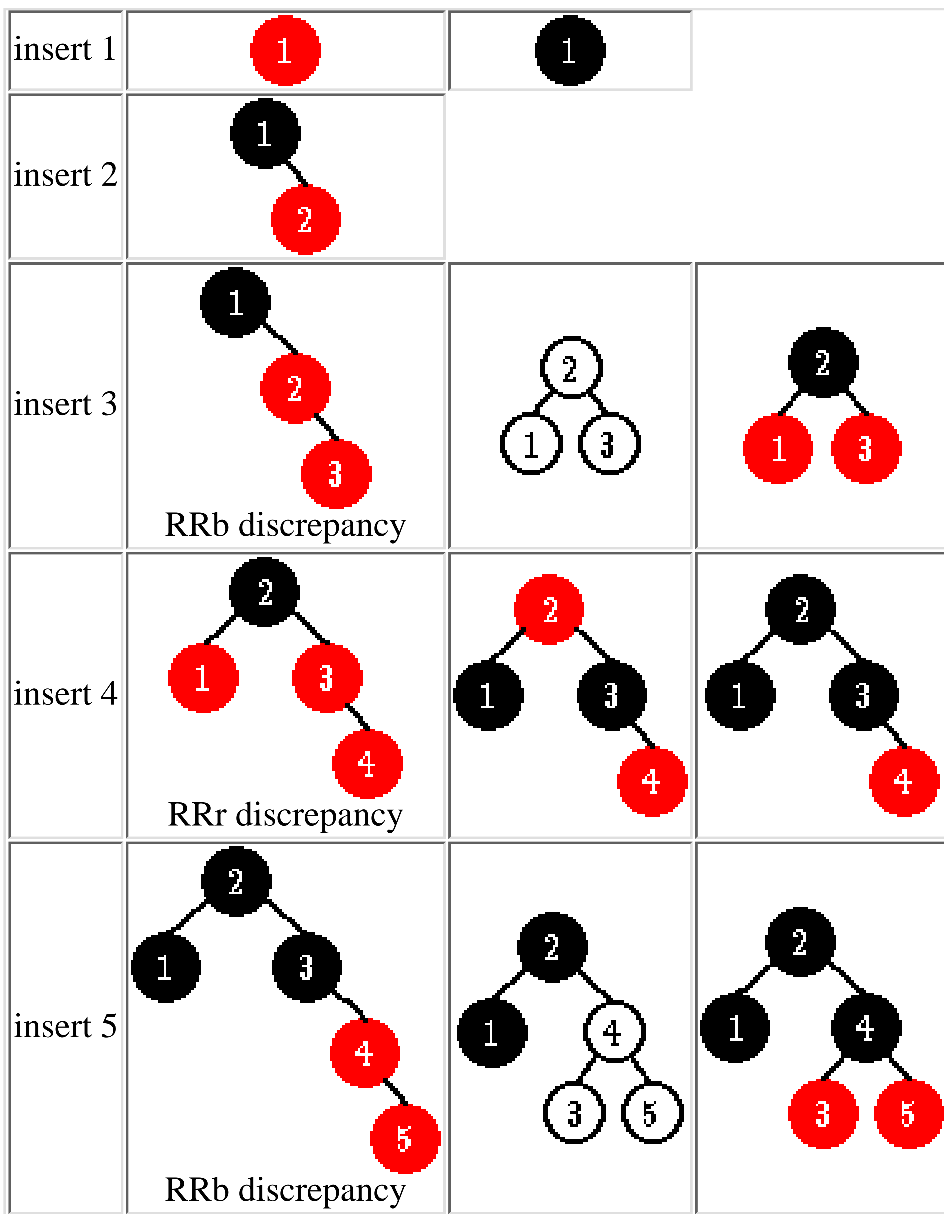
Discrepancies in which the sibling is red, are fixed by changes in color. Discrepancies in which the siblings are black, are fixed through AVL-like rotations.

Changes in color may propagate the problem up toward the root. On the other hand, at most one rotation is sufficient for fixing a discrepancy.





Discrepancies of type RRr, RLr, RRb, and RLb are handled in a similar manner.

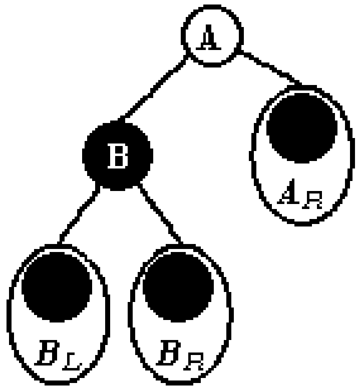
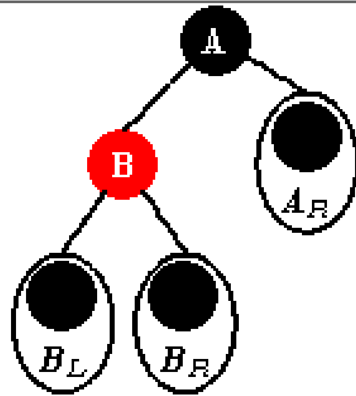


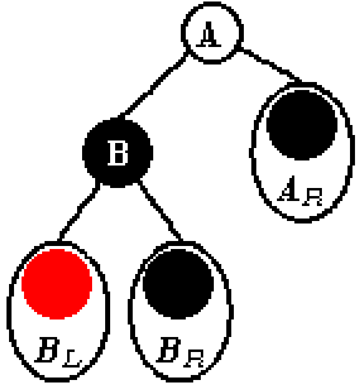
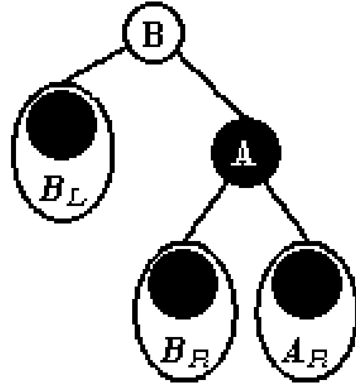
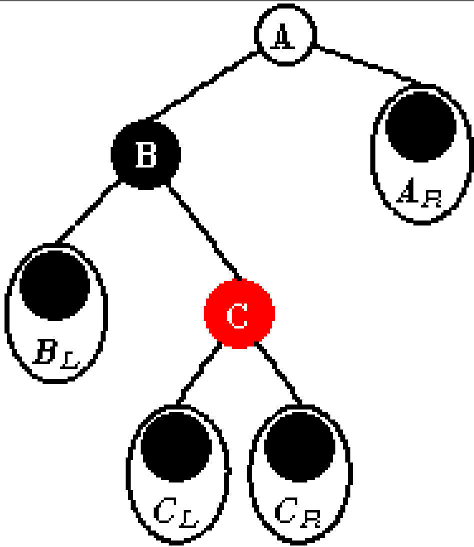
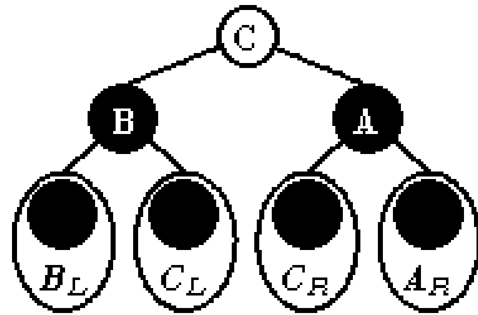
11.3 Deletions

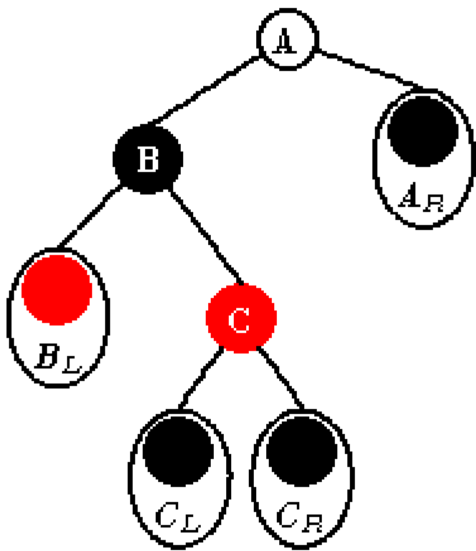
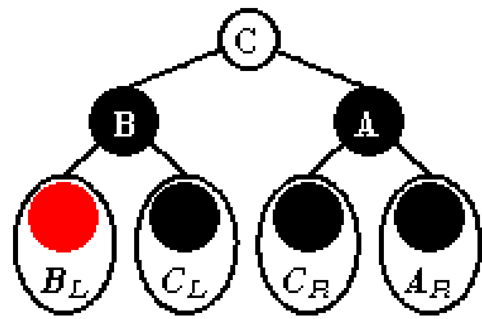
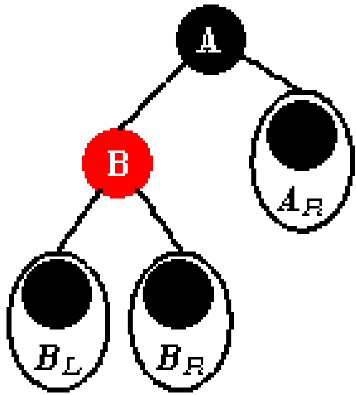
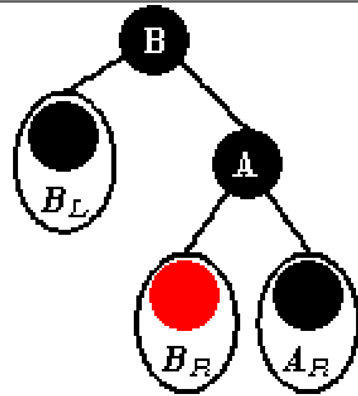
- Delete a key, and a node, the way it is done in binary search trees.
- A node to be deleted will have at most one child. If the deleted node is red, the tree is still a red-black tree. If the deleted node has a red child, repaint the child to black.
- If a discrepancy arises for the red-black tree, fix the tree according to the type of discrepancy. A discrepancy can result only from a loss of a black node.

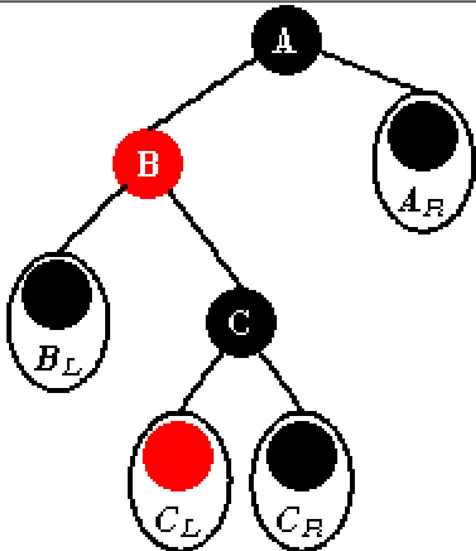
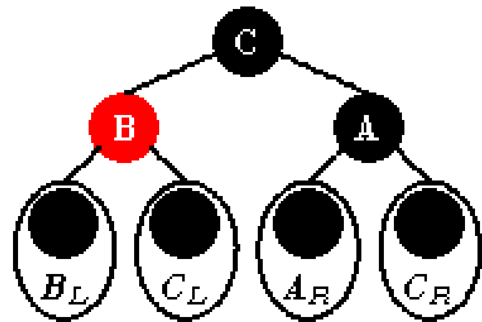
Let A denote the lowest node with unbalanced subtrees. The type of discrepancy is determined by the location of the deleted node (**R**ight or **L**eft), the color of the sibling (**b**lack or **r**ed), the number of red children in the case of the black siblings, and and the number of grand-children in the case of red siblings.

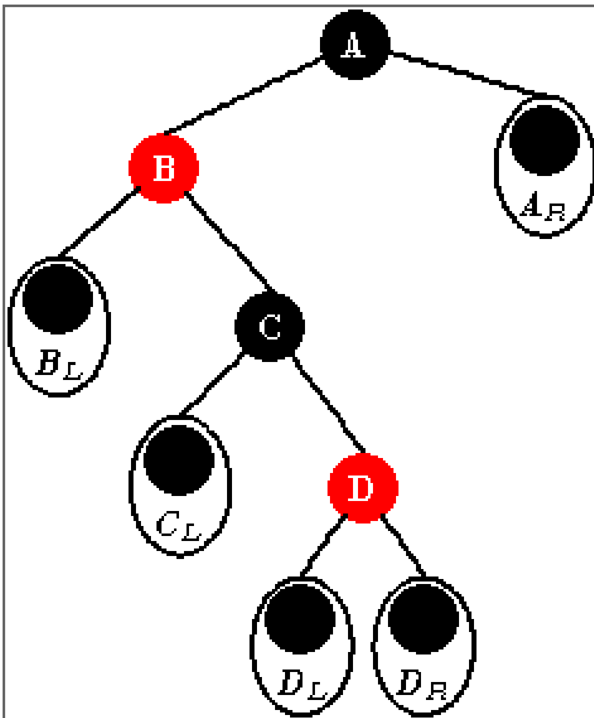
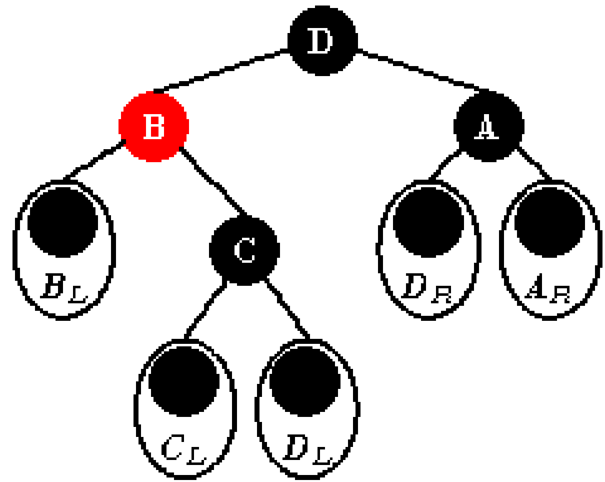
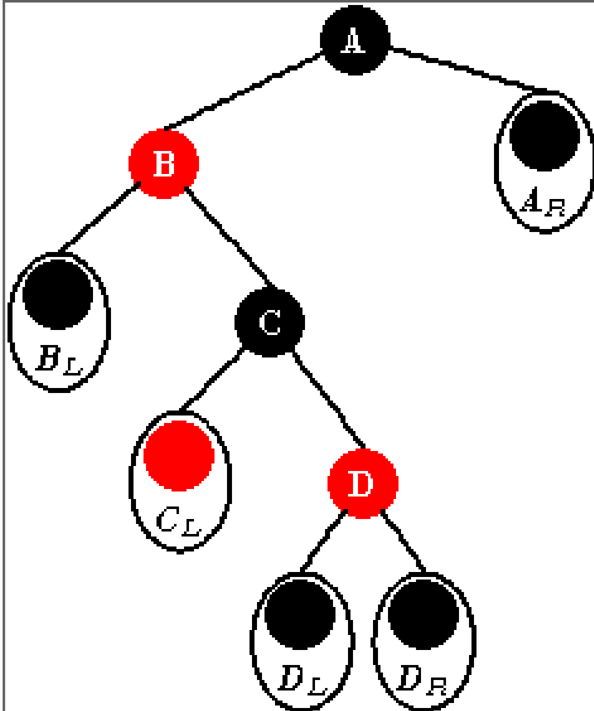
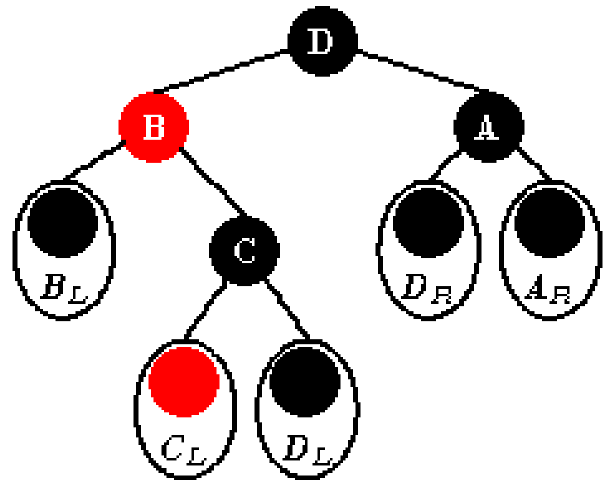
In the case of discrepancies which result from the addition of nodes, the correction mechanism may propagate the color problem (i.e., parent and child painted red) up toward the root, and stopped on the way by a single rotation. Here, in the case of discrepancies which result from the deletion of nodes, the discrepancy of a missing black node may propagate toward the root, and stopped on the way by an application of an appropriate rotation.

<table>
<tr><td>Rb0</td><td>
</td><td>

change of color, sending the deficiency up to the root of the subtree</td></tr>
<tr><td>Rb1</td><td>
</td><td>
</td></tr>
<tr><td></td><td>
</td><td>
</td></tr>
<tr><td></td><td></td><td></td></tr>
</table>

<table>
<tr>
<td>Rb2</td>
<td>
</td>
<td>
</td>
</tr>
<tr>
<td>Rr0</td>
<td>
</td>
<td>

might result in LLb discrepancy of parent and child having both the red color</td>
</tr>
<tr>
<td>Rr1</td>
<td>
</td>
<td>
</td>
</tr>
</table>

<table>
<tr>
<td></td>
<td>
</td>
<td>
</td>
</tr>
<tr>
<td>Rr2</td>
<td>
</td>
<td>
</td>
</tr>
</table>

Similar transformations apply to Lb0, Lb1, Lb2, Lr0, Lr1, and Lr2.

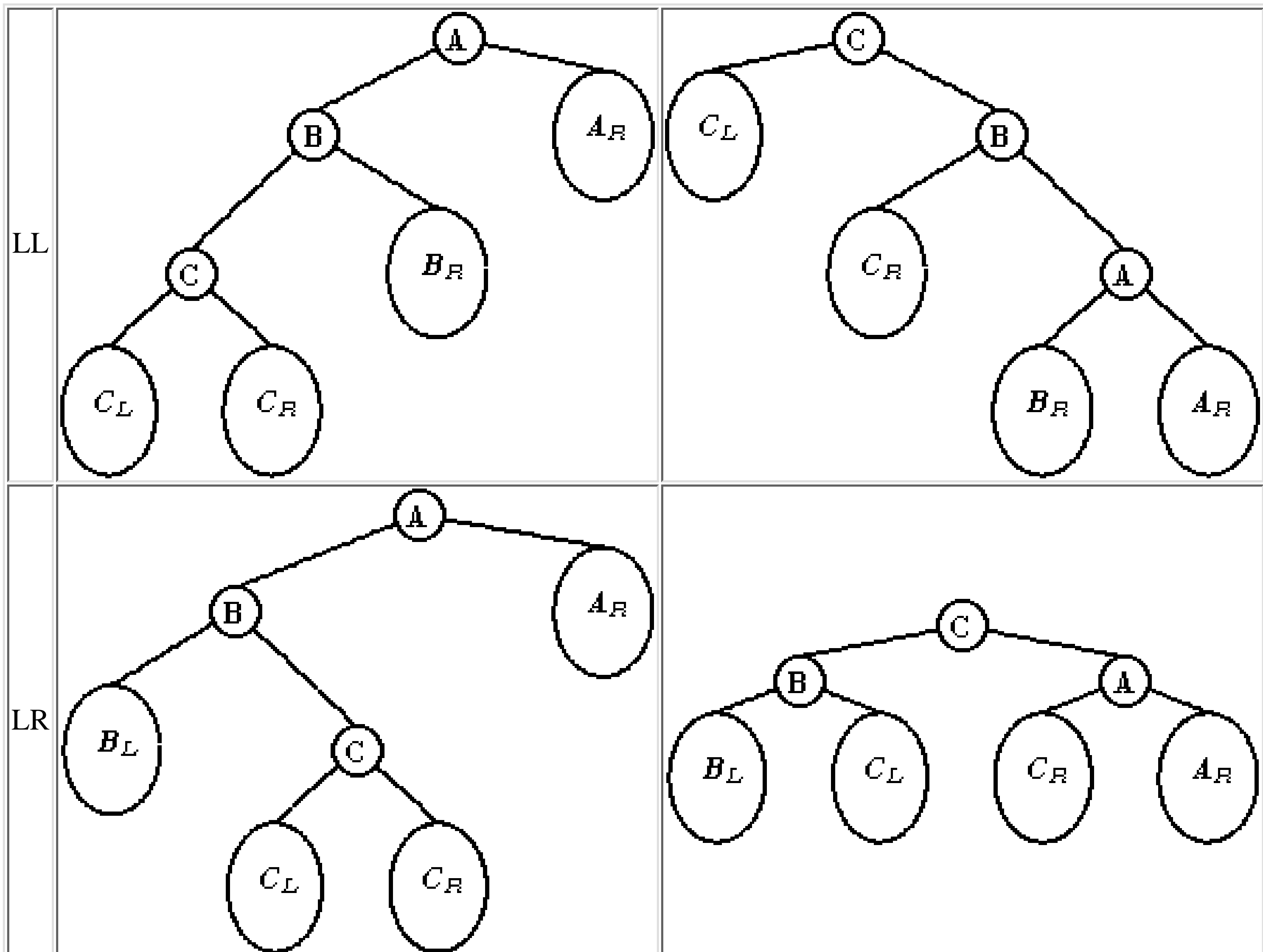
11.4 Demo Applet

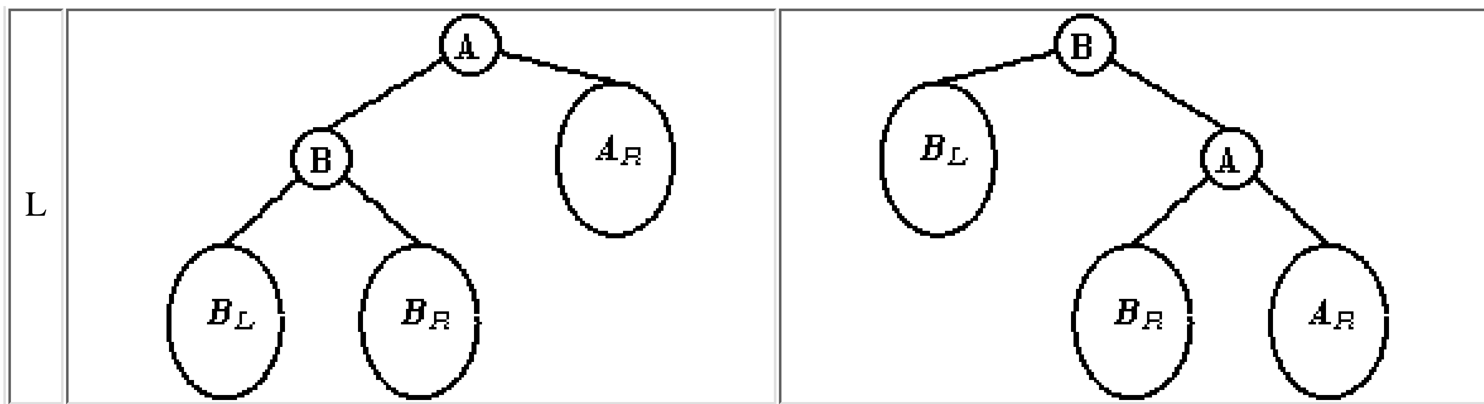
- demo applet for red-black trees
- another demo applet

11.5 Assignment #5 (due Fr, Oct 29)

1. Construct a binary search tree by introducing the following keys in the given order: 4, 3, 2, 1, 11, 8, 5, 6, 7, 9, 10, 12, 13. Color the nodes at odd level with black, and the nodes at even levels with red (the root is assumed to be at level 1). Then repeatedly use the red-black transformations to transform the tree into a red-black tree, while showing all the intermediate trees being created in the process.

   In each stage, the transformation should be conducted at a discrepancy that is farthest from the root. A discrepancy is assumed to be caused by a deletion, when the numbers of black nodes in the different paths are not equal. Otherwise, it is assumed to be caused by an insertion.
2. Can every binary search tree be transformed into a red-black tree by using only red-black tree transformations, assuming the coloring strategy of the previous problem, and that in each stage the transformation should be conducted at a discrepancy that is farthest from the root? Justify your answer.
3. Consider the following operations.





A binary search tree is said to be splayed at a given node, if the node is moved to the root with the above, and symmetric, operations. *Splay trees* (applet), or *or self-adjusting search trees*, are trees which are splayed whenever their nodes are accessed. On the average, they are accessed and modified in O(log n) time.

Construct a binary search tree by introducing the following keys in the given order: 7, 6, 5, 4, 3, 2, 1. Then splay the tree at node 1, and show all the intermediate trees being created in the process. (In the LL and LR cases, 1 is represented by the node C. In the L case it is represented by the node B.)

[next] [prev] [prev-tail] [front] [up]

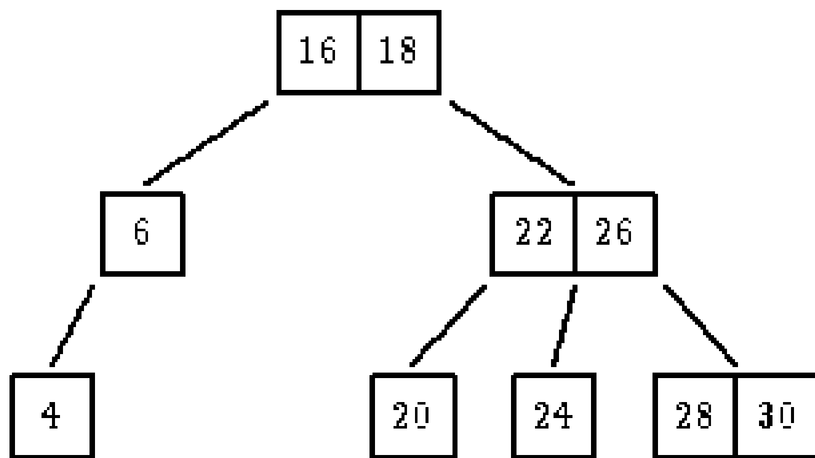
[next] [prev] [prev-tail] [tail] [up]

Chapter 12
Multi-way Trees

12.1 Definition

A m-way search tree is a tree in which

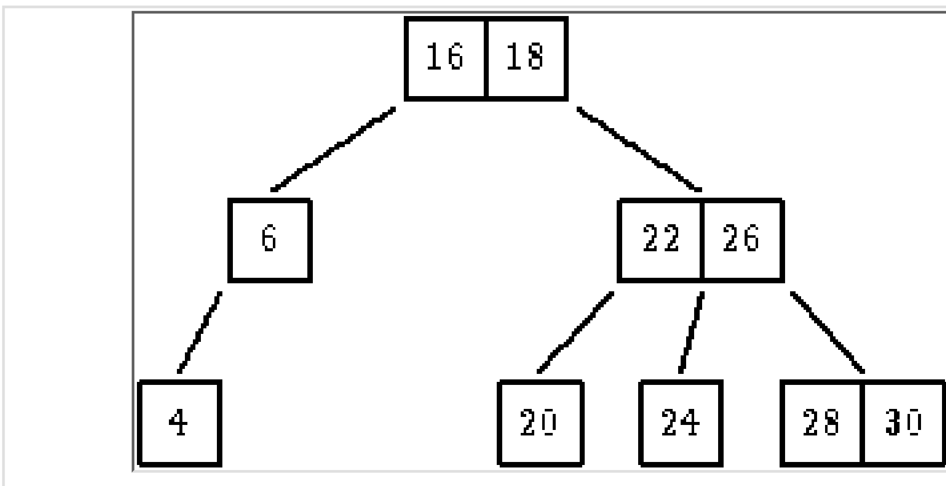
a. The nodes hold between 1 to m-1 distinct keys
b. The keys in each node are sorted
c. A node with k values has k+1 subtrees, where the subtrees may be empty.
d. The i th subtree of a node $[v_1, ..., v_k]$, $0 \leq i \leq k$, may hold only values v in the range $v_i \leq v \leq v_{i+1}$ (v_0 is assumed to equal $-\infty$, and v_{k+1} is assumed to equal infty).

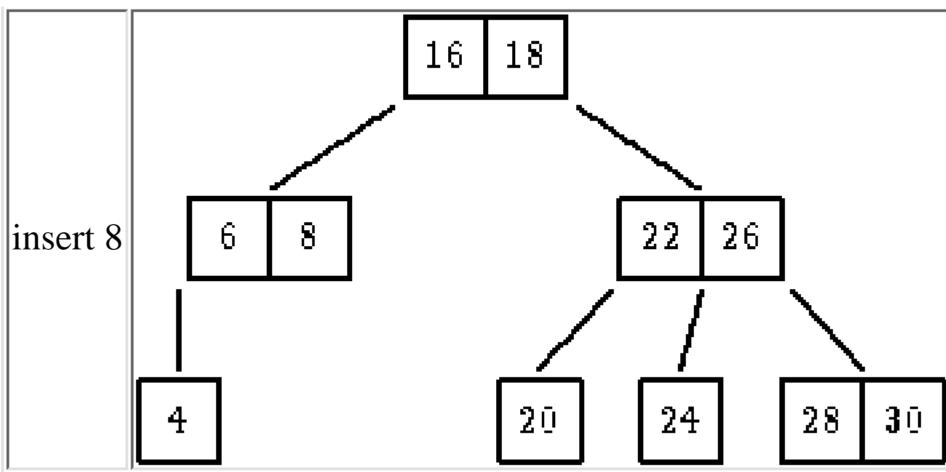


A m-way tree of height h has between h and m^h - 1 keys.

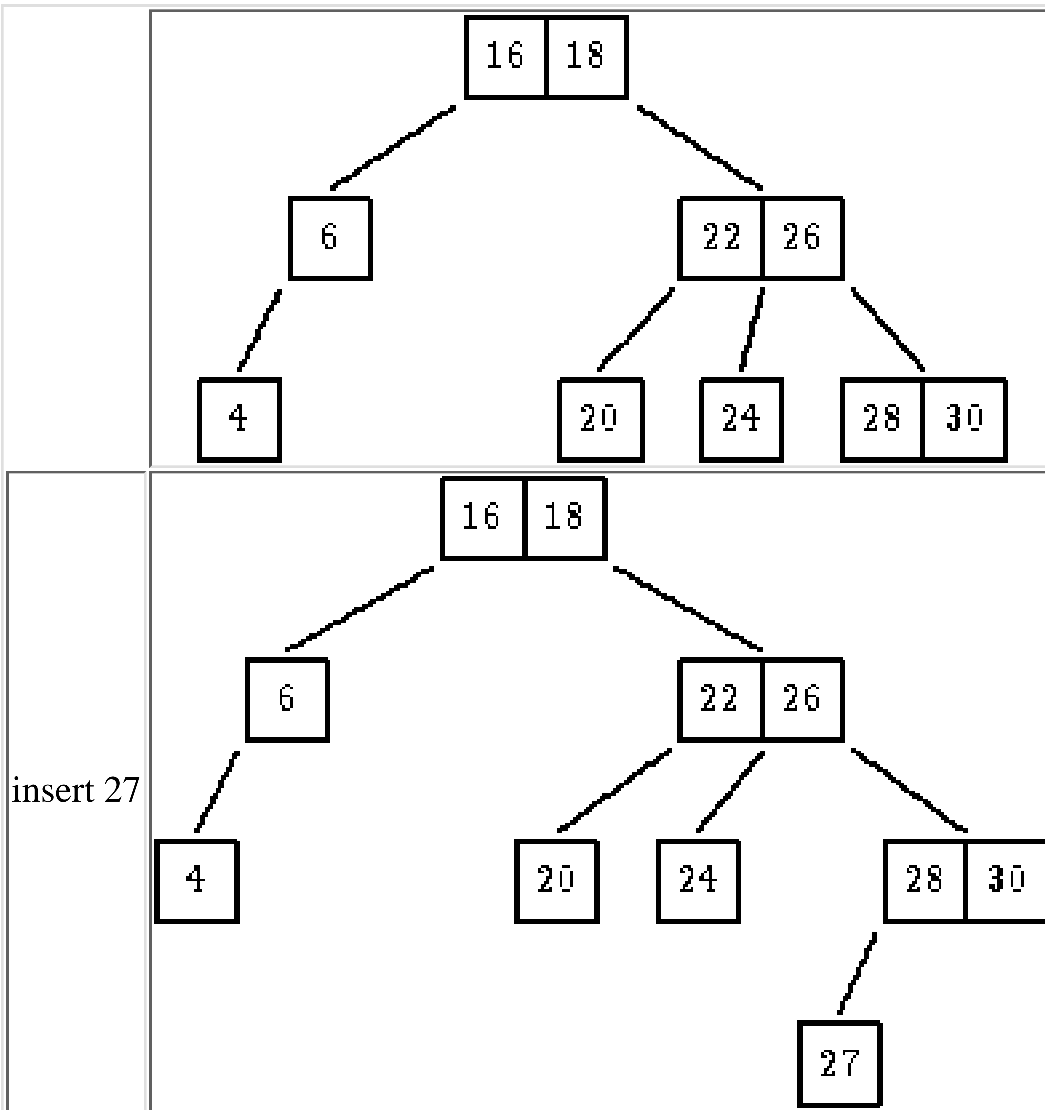
12.2 Insertions

- Search the key going down the tree until reaching an empty subtree
- Insert the key to the parent of the empty subtree, if there is room in the node.



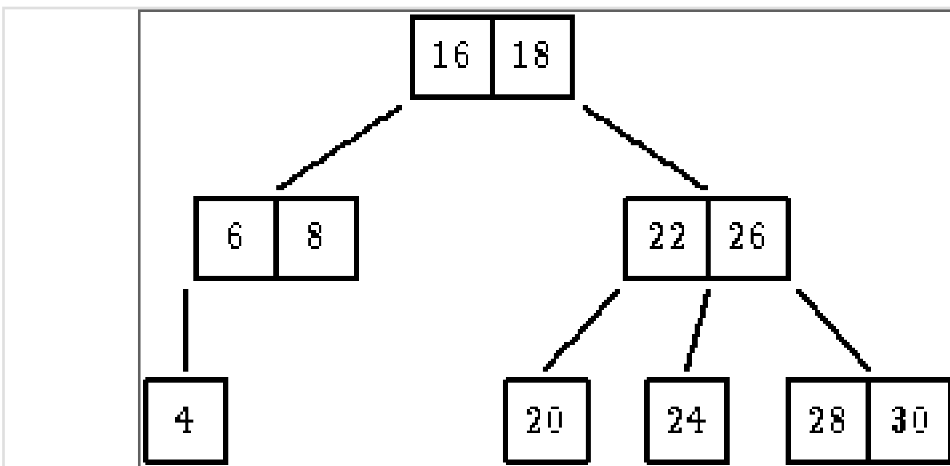


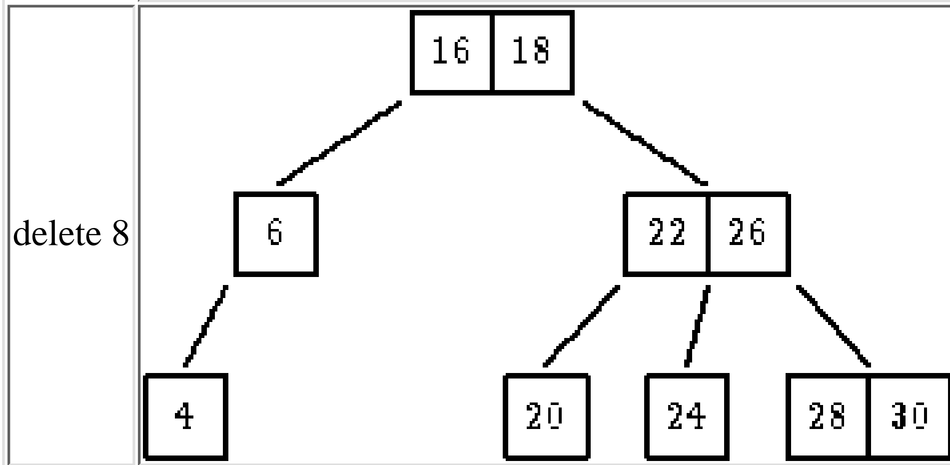
- Insert the key to the subtree, if there is no room in its parent.



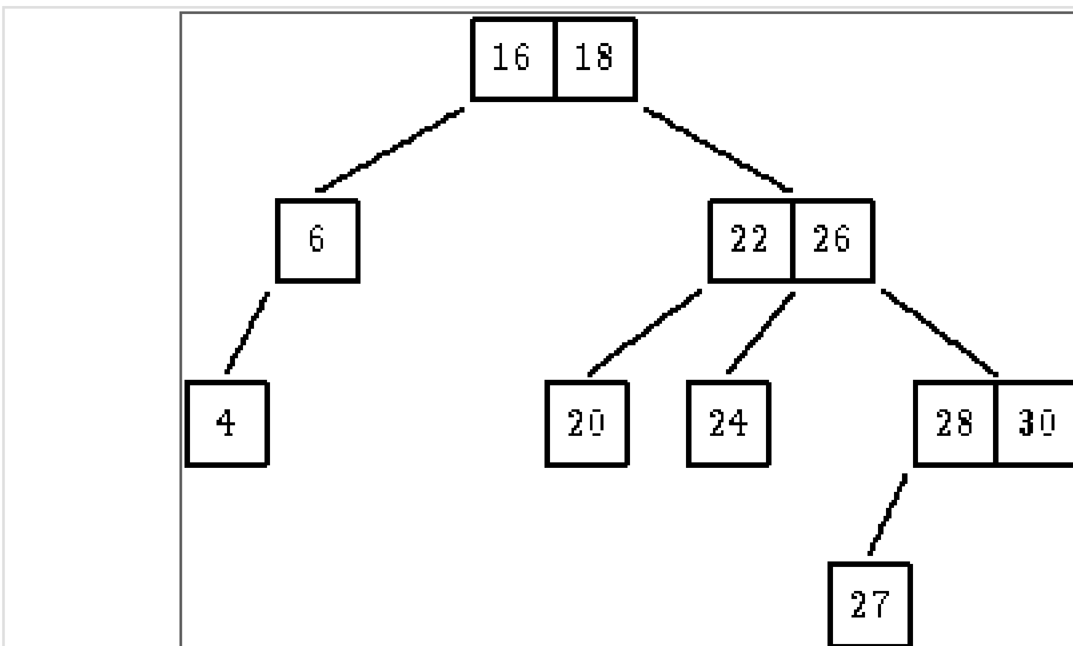
12.3 Deletions

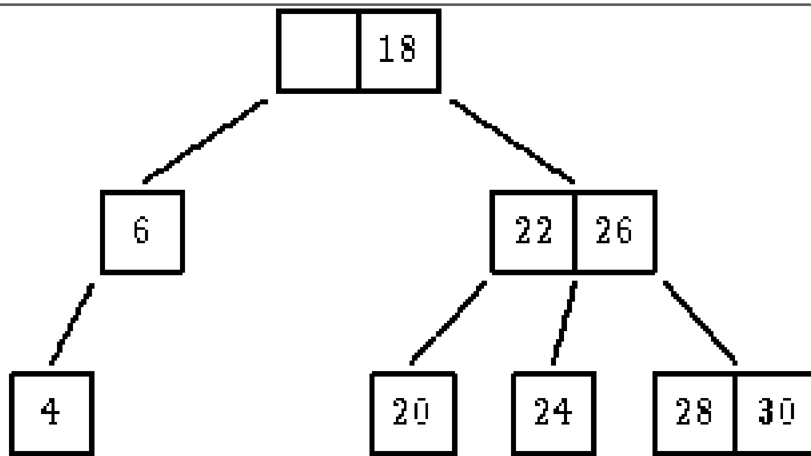
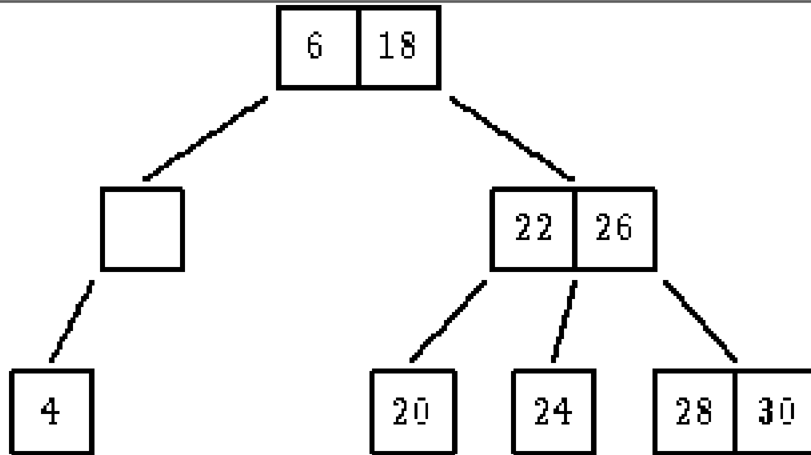
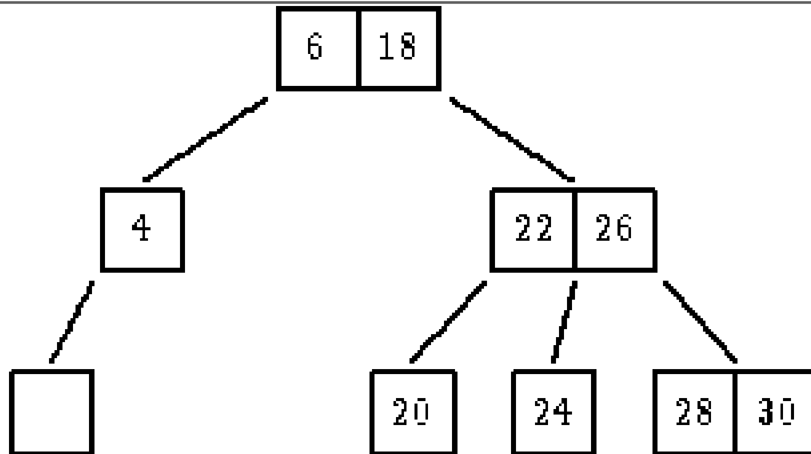
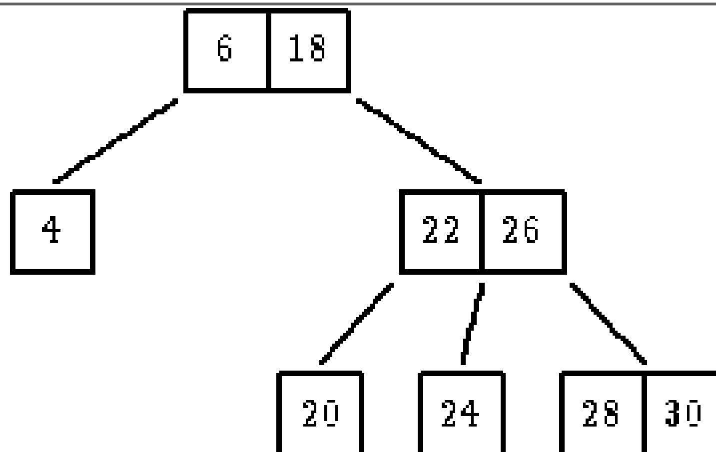
- If the key to be deleted is between two empty subtrees, delete it





- If the key is adjacent to a nonempty subtree, replace it with the largest key from the left subtree or the smallest key from the right subtree



remove 16	
	
	
	

[next] [prev] [prev-tail] [front] [up]

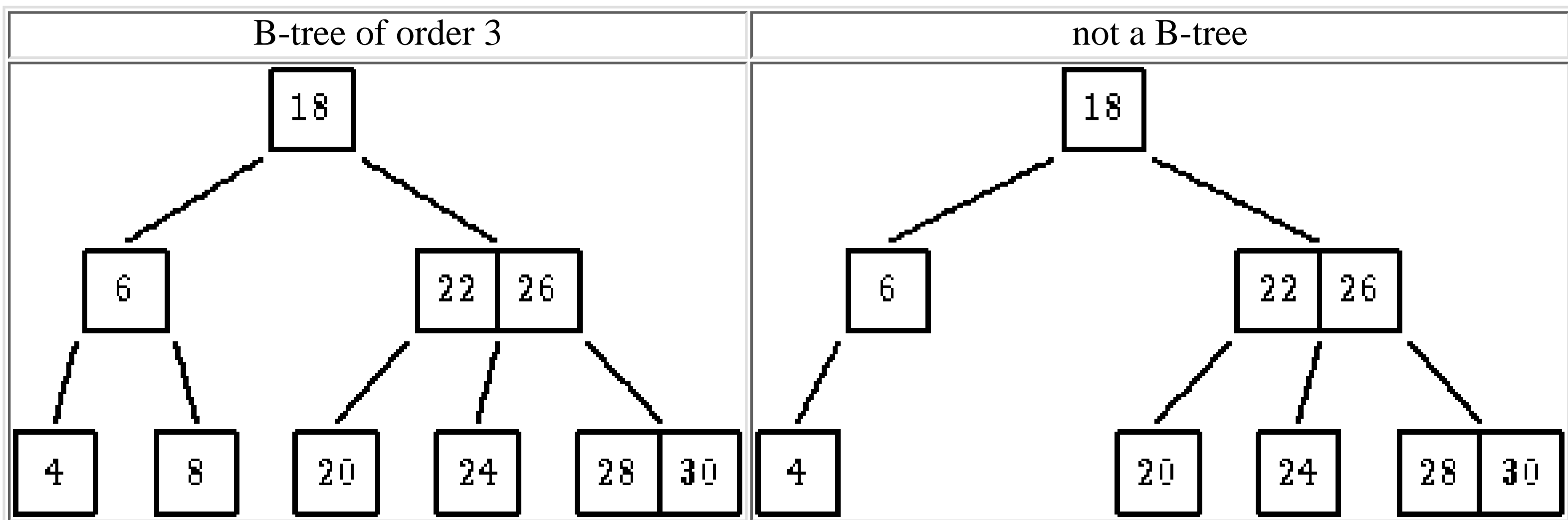
[prev] [prev-tail] [tail] [up]

Chapter 13
B-Trees

13.1 Definition

A m-way tree in which

- The root has at least one key
- Non-root nodes have at least $\lceil m/2 \rceil$ subtrees (i.e., at least $\lfloor (m - 1)/2 \rfloor$ keys)
- All the empty subtrees (i.e., external nodes) are at the same level

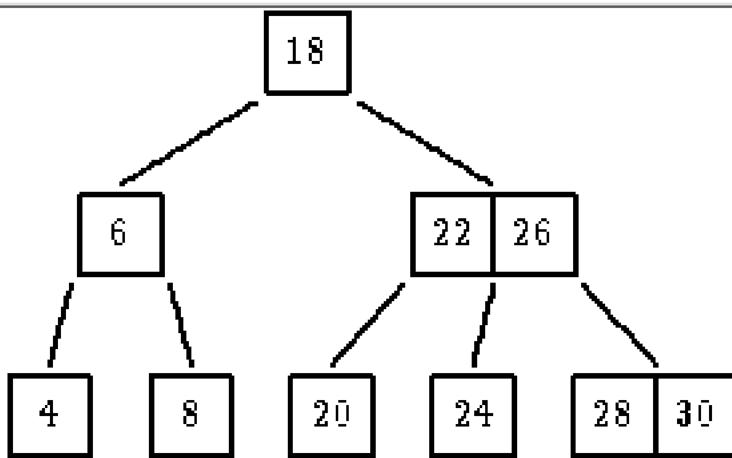


B-trees are especially useful for trees stored on disks, since their height, and hence also the number of disk accesses, can be kept small.

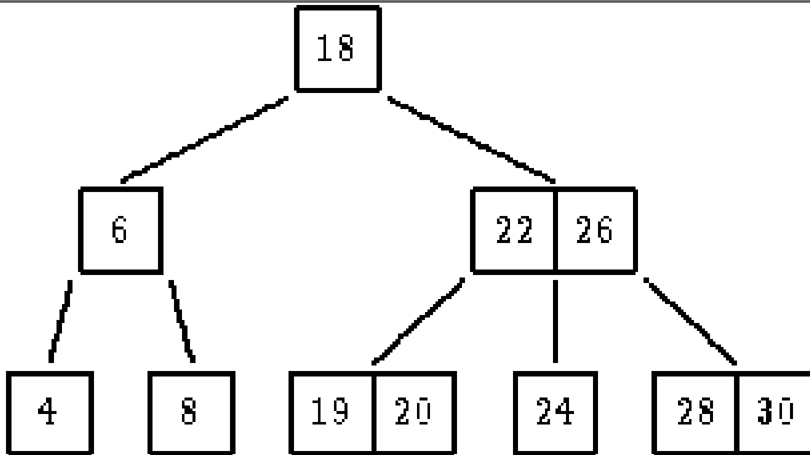
The growth and contraction of m-way search trees occur at the leaves. On the other hand, B-trees grow and contract at the root.

13.2 Insertions

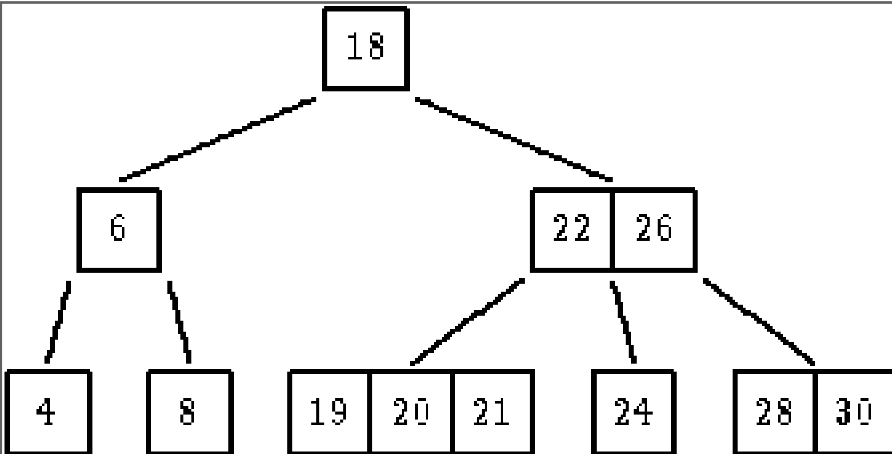
- Insert the key to a leaf
- Overfilled nodes should send the middle key to their parent, and split into two at the location of the submitted key.

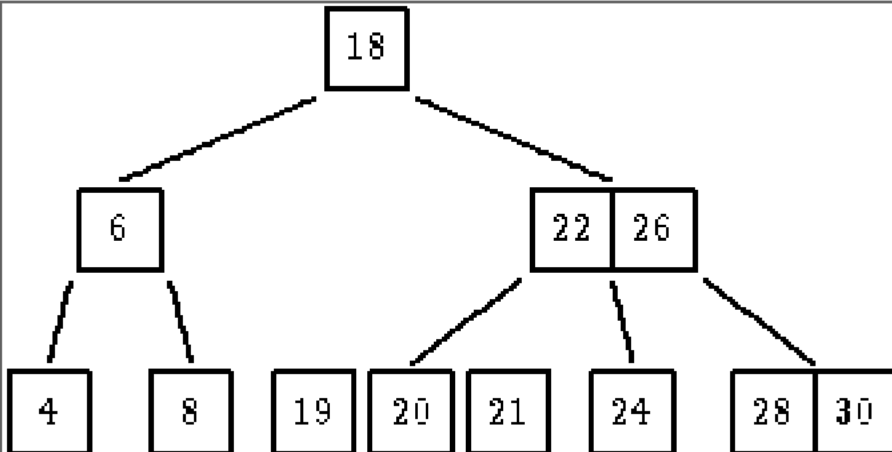


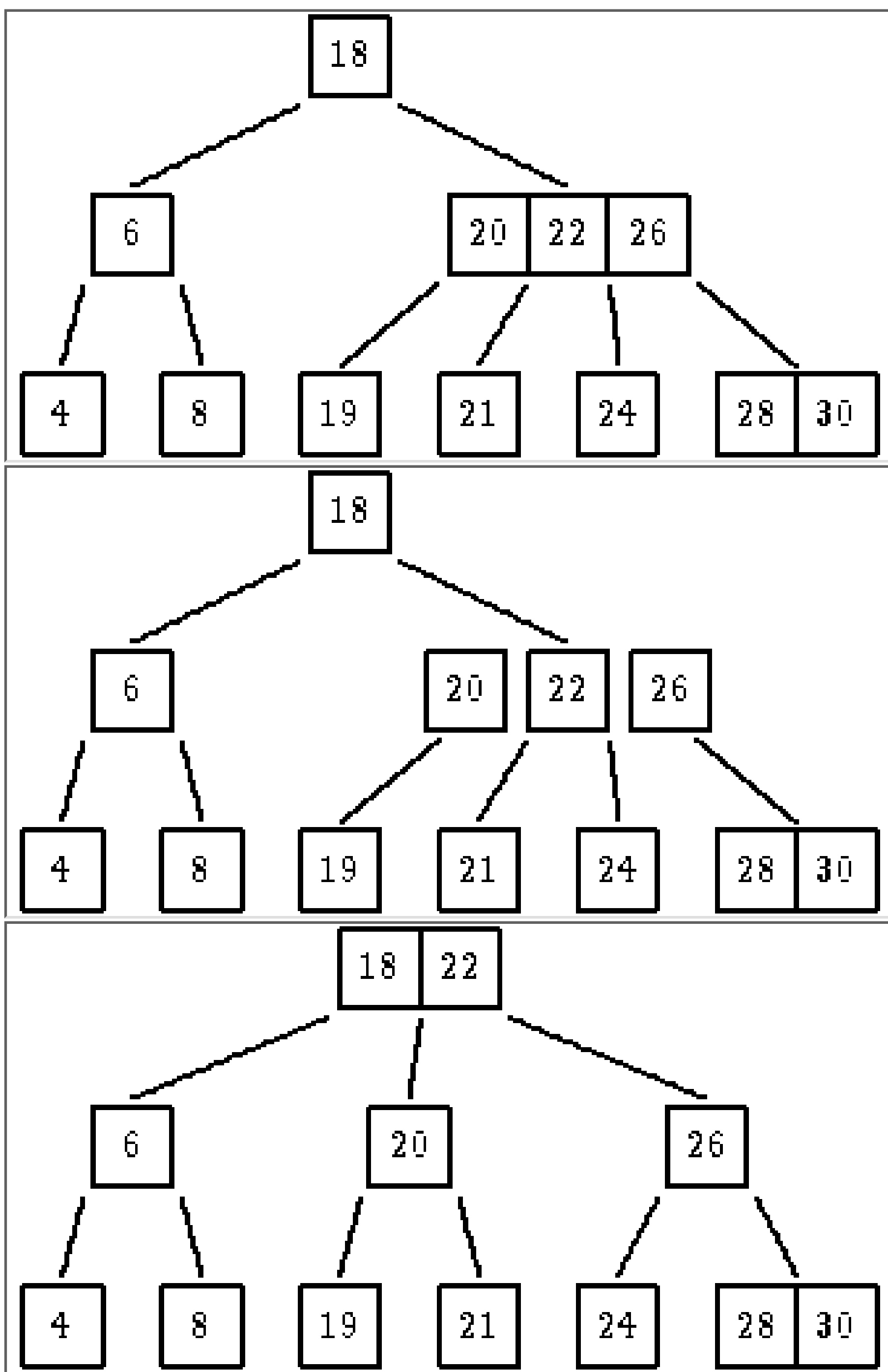
add 19



add 21

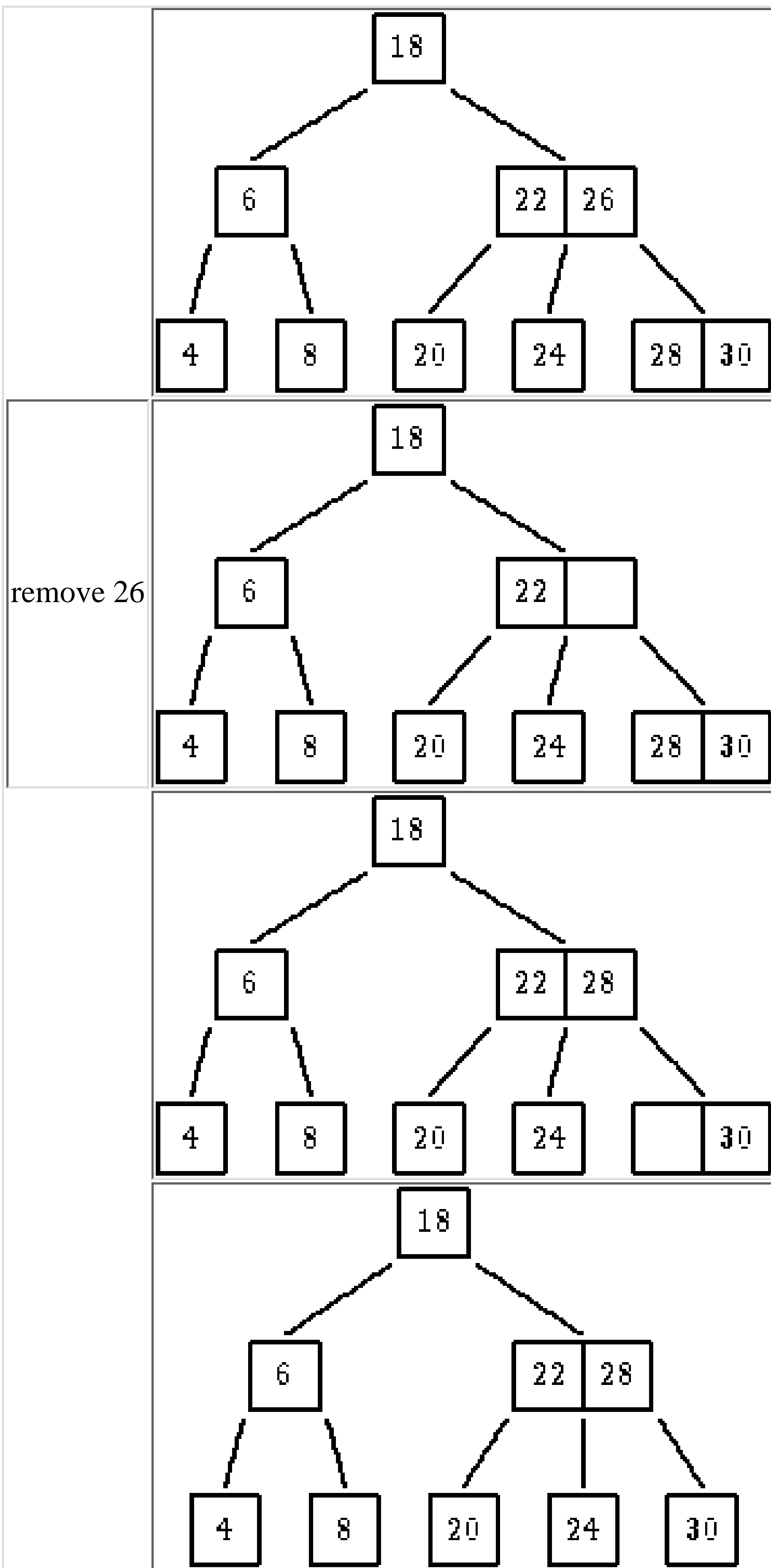




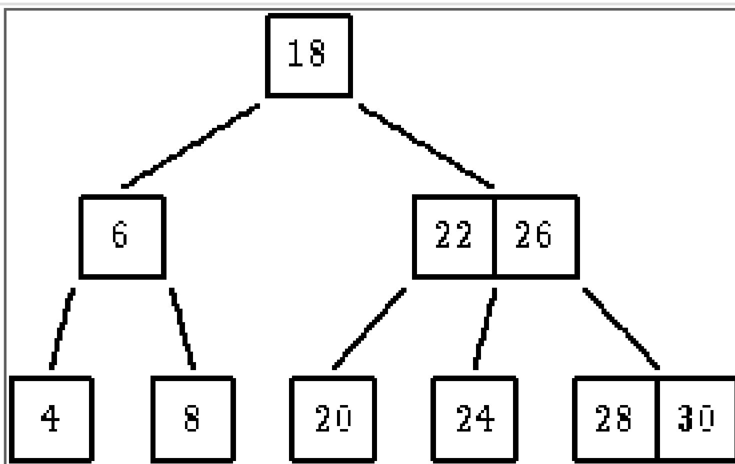


13.3 Deletions

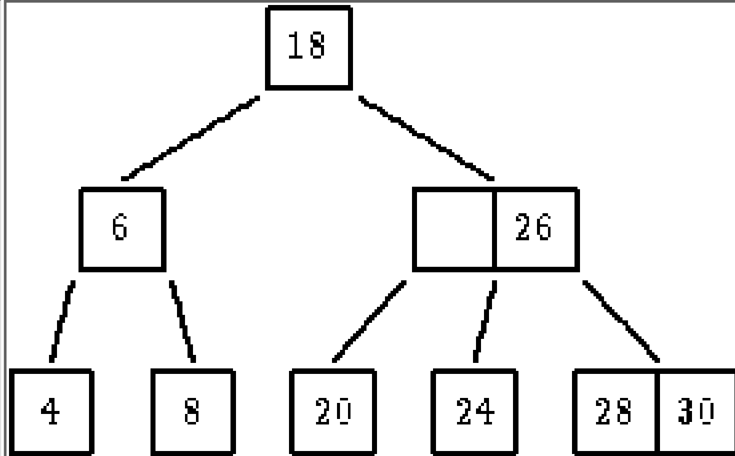
- Key that is to be removed from a node with non-empty subtrees is being replaced with the largest key of the left subtree or the smallest key in the right subtree. (The replacement is guaranteed to come from a leaf.)

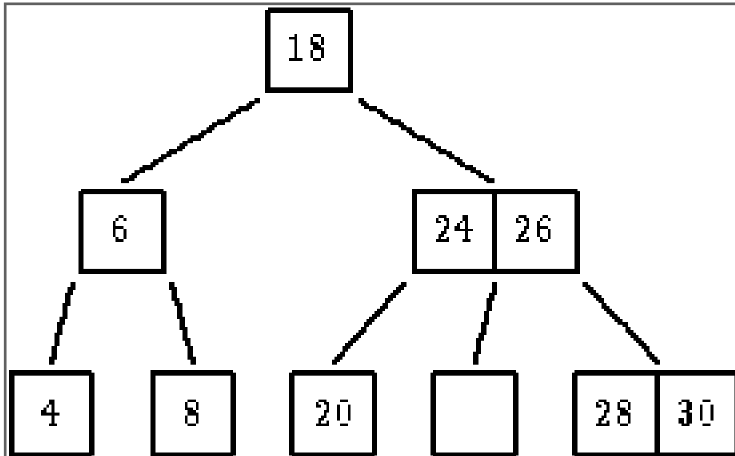


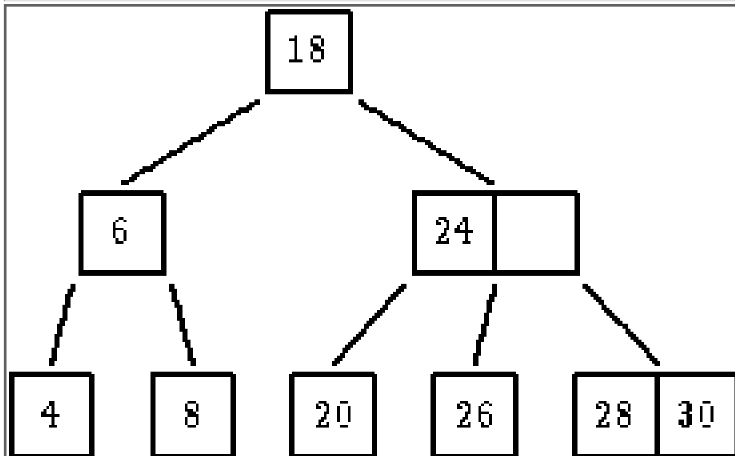
- If a node becomes under staffed, it looks for a sibling with an extra key. If such a sibling exist, the node takes a key from the parent, and the parent gets the extra key from the sibling.

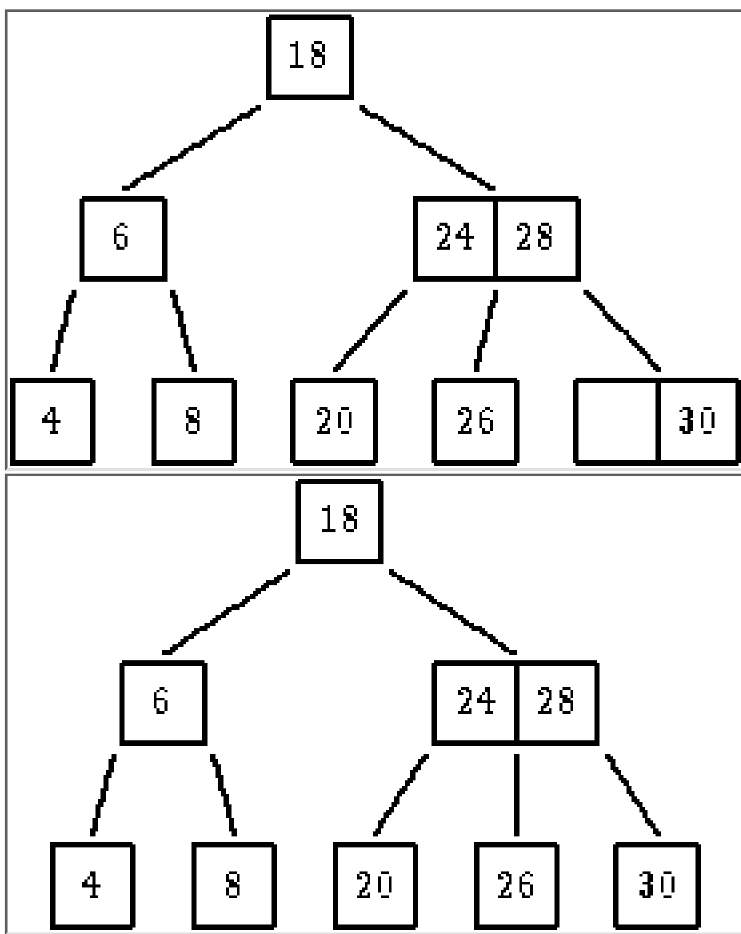


remoce 22

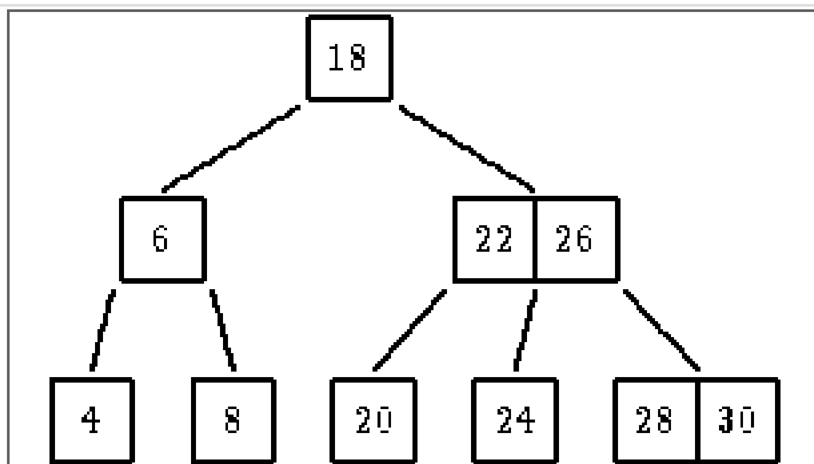


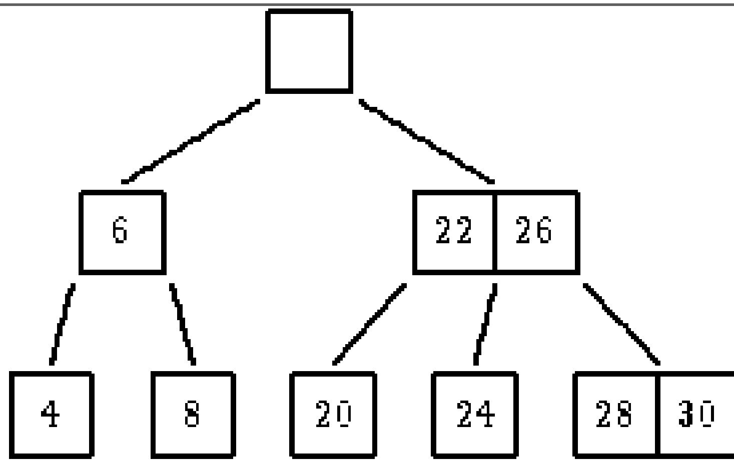
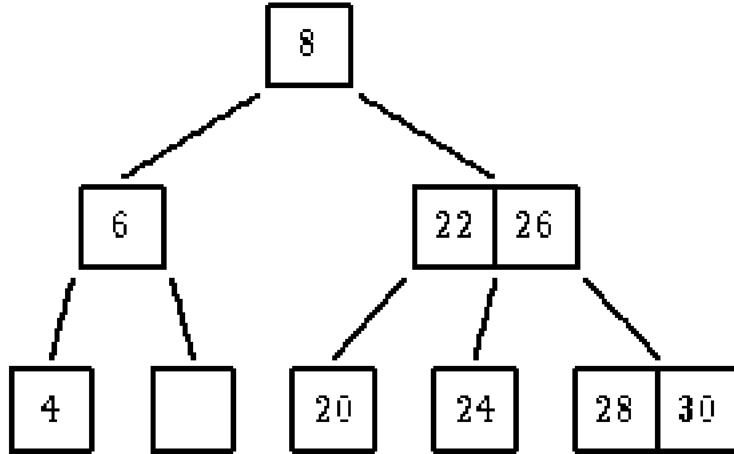
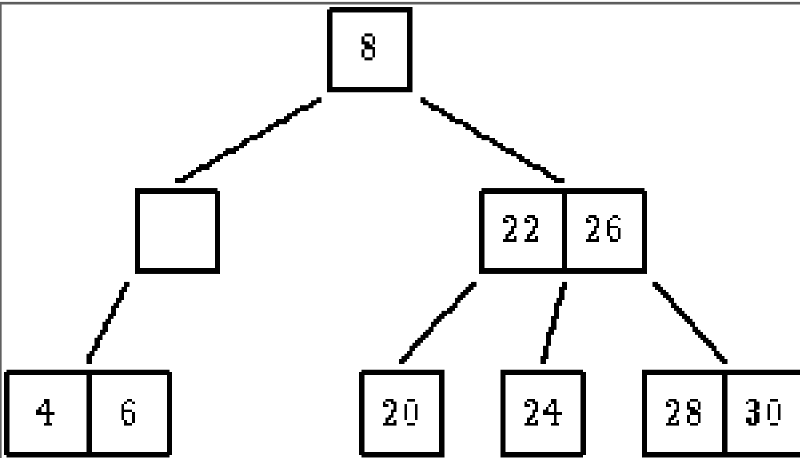
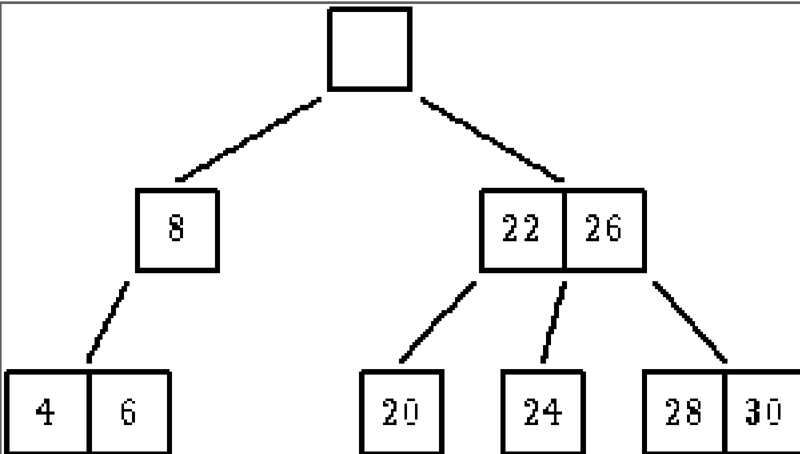


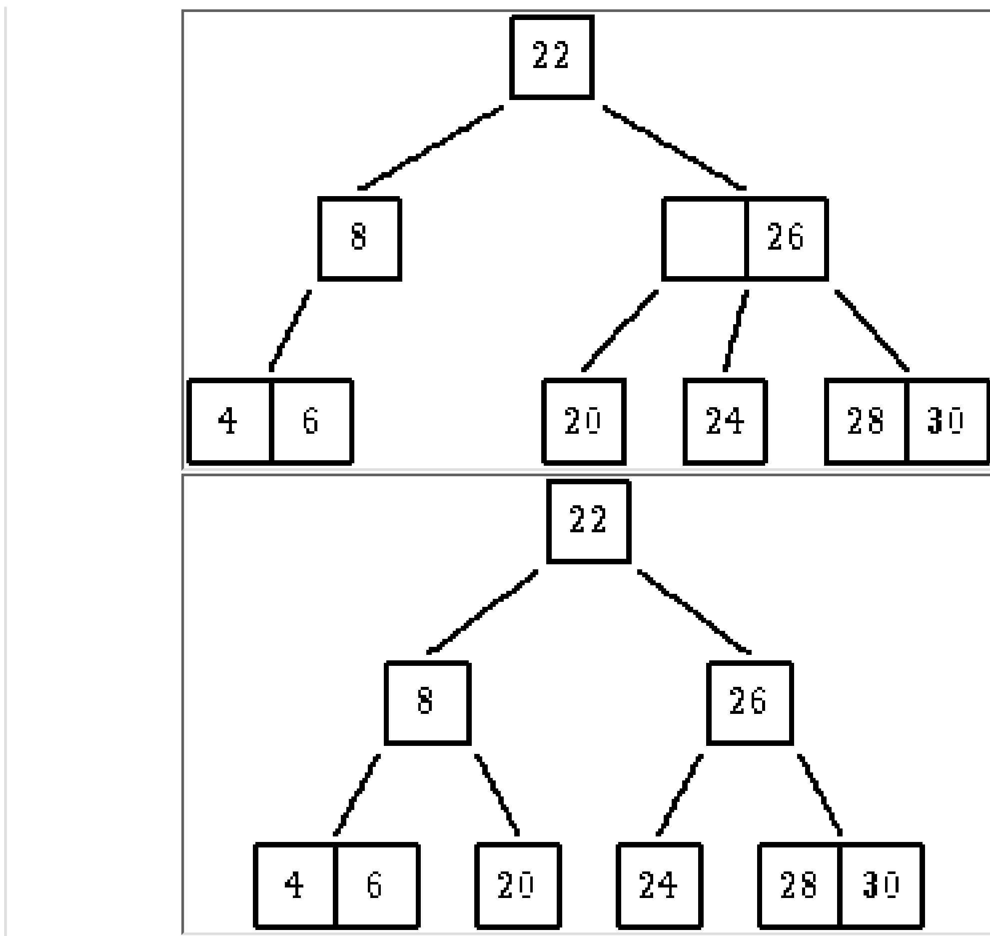




- If a node becomes under staffed, and it can t receive a key from a sibling, the node is merged with a sibling and a key from the parent is moved down to the node.



remove 18	
	
	
	



13.4 Assignment #6 (due We, Nov 3)

a. Consider B-trees of order 3. Starting from an empty tree, show all the intermediate B-trees obtained by the sequence of operations: add key 1, add key 2, add key 3, add key 4, add key 5, add key 6, add key 7, add key 8, add key 9, delete key 1, delete key 2, delete key 3, delete key 4, delete key 5, and delete key 6.
b. Splitting of overflowed nodes may be avoided by redistributing keys to siblings, in a manner similar to that taking place for avoiding merging of underflowed nodes. Repeat the addition portion of the previous problem, utilizing redistribution of keys when possible.
c. Give a procedure that will transform any B-Tree of degree 4 into a Red-Black Tree, and a procedure that will transform any Red-Black Tree into a B-Tree of degree 4.

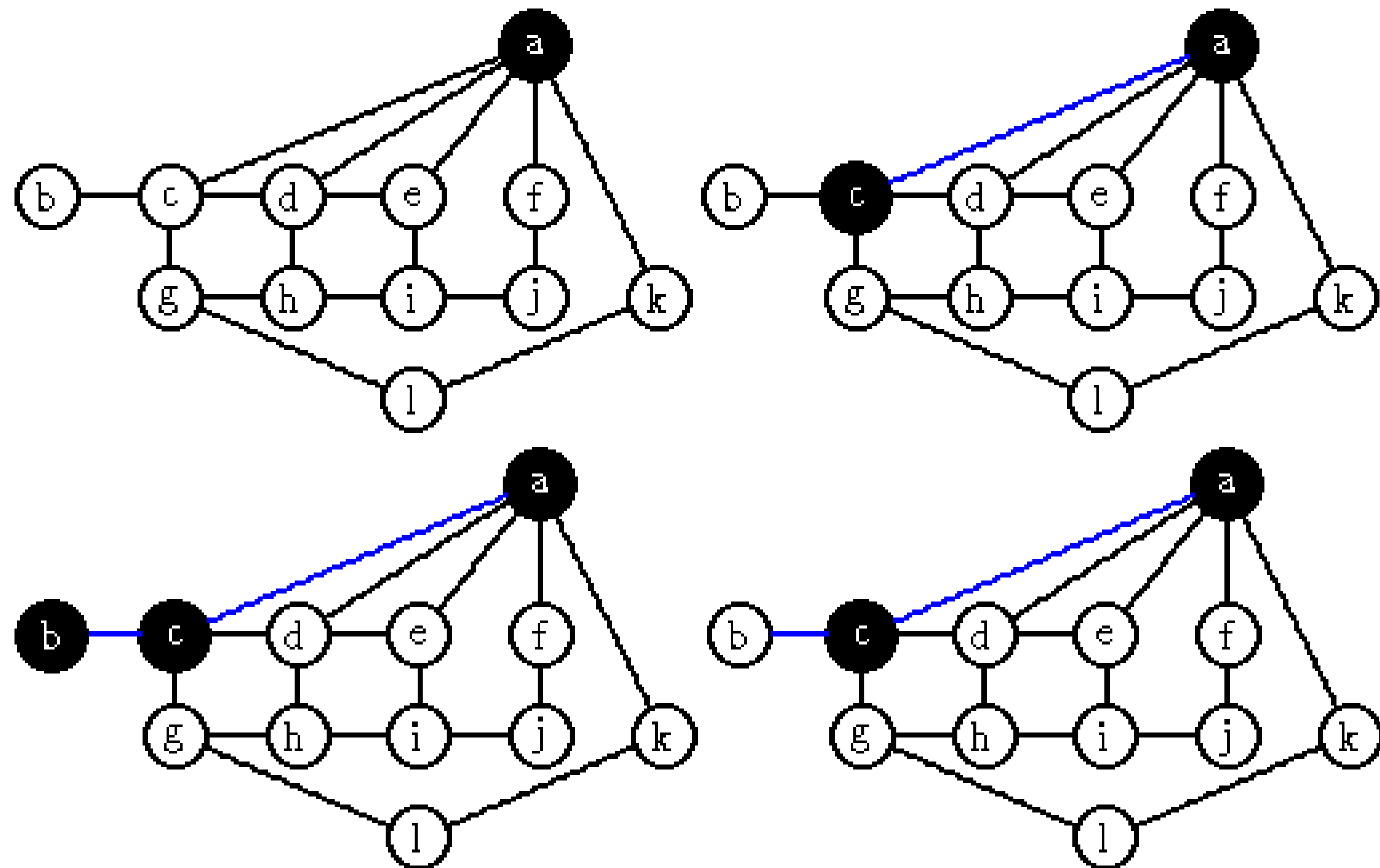
[prev] [prev-tail] [front] [up]

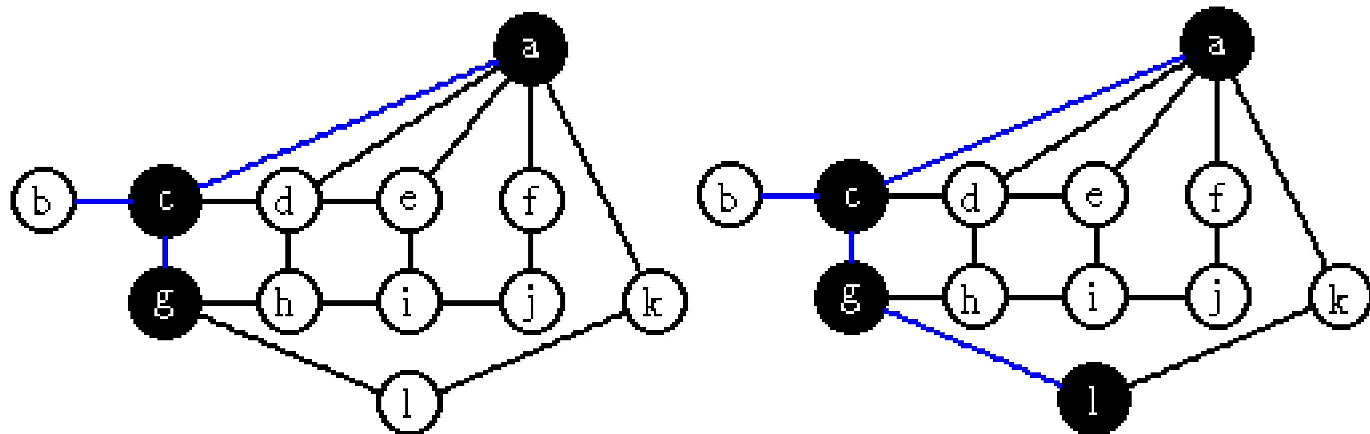
Chapter 14
Graph Traversal

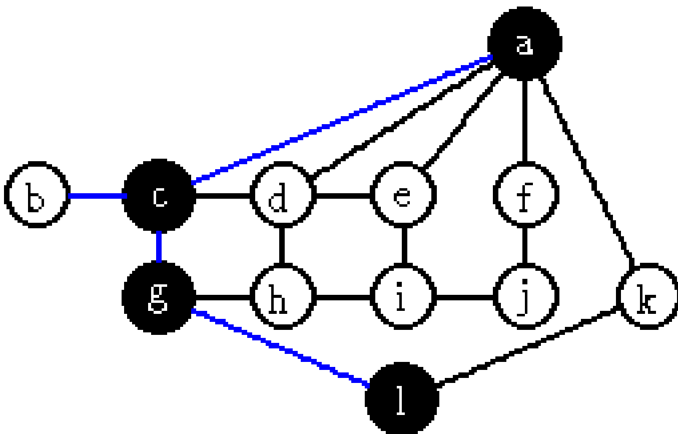
14.1 Depth-First Traversal

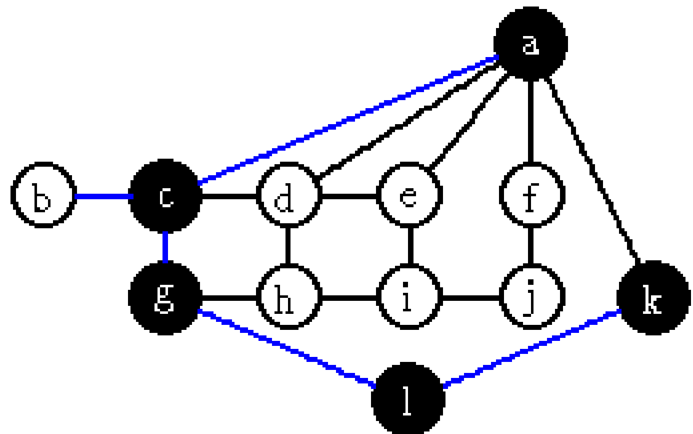
```
algorithm  dft(x)
   visit(x)
   FOR each y such that (x,y) is an edge DO
     IF y was not visited yet THEN
        dft(y)
```

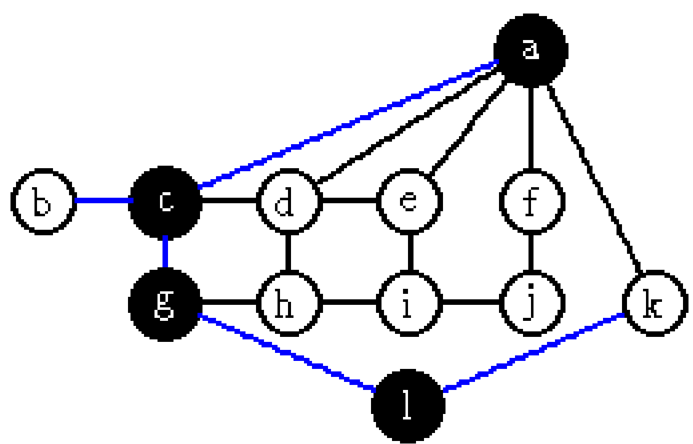
A recursive algorithm implicitly recording a backtracking path from the root to the node currently under consideration

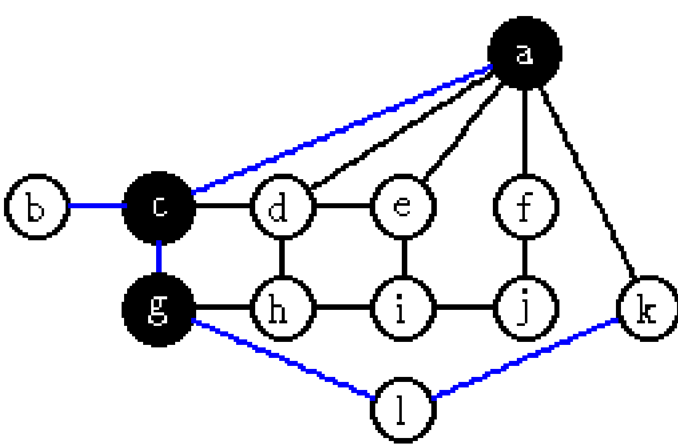


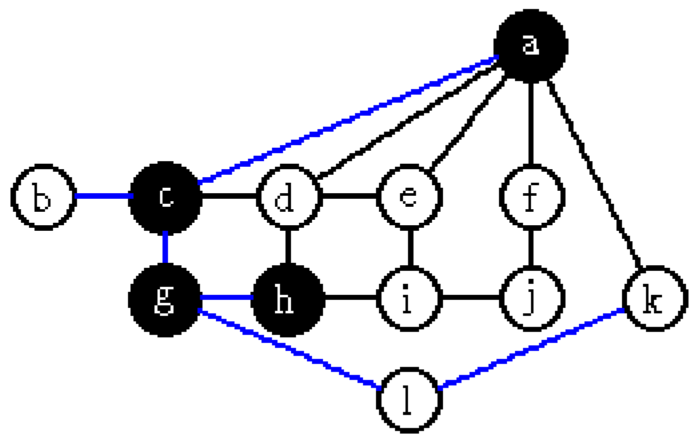

a
b
c
d
e
f
g
h
i
j
k
l

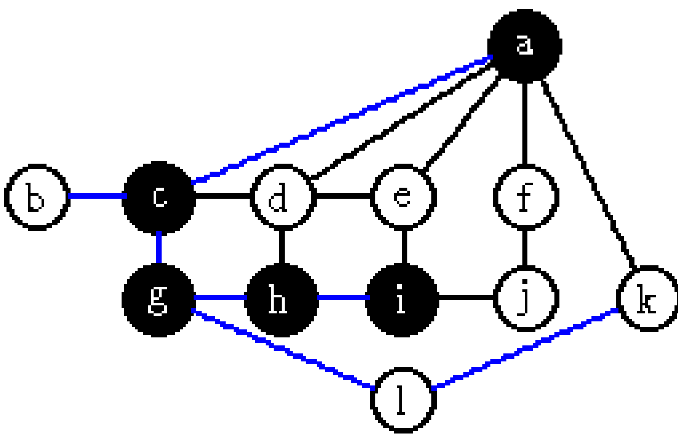

a
b
c
d
e
f
g
h
i
j
k
l

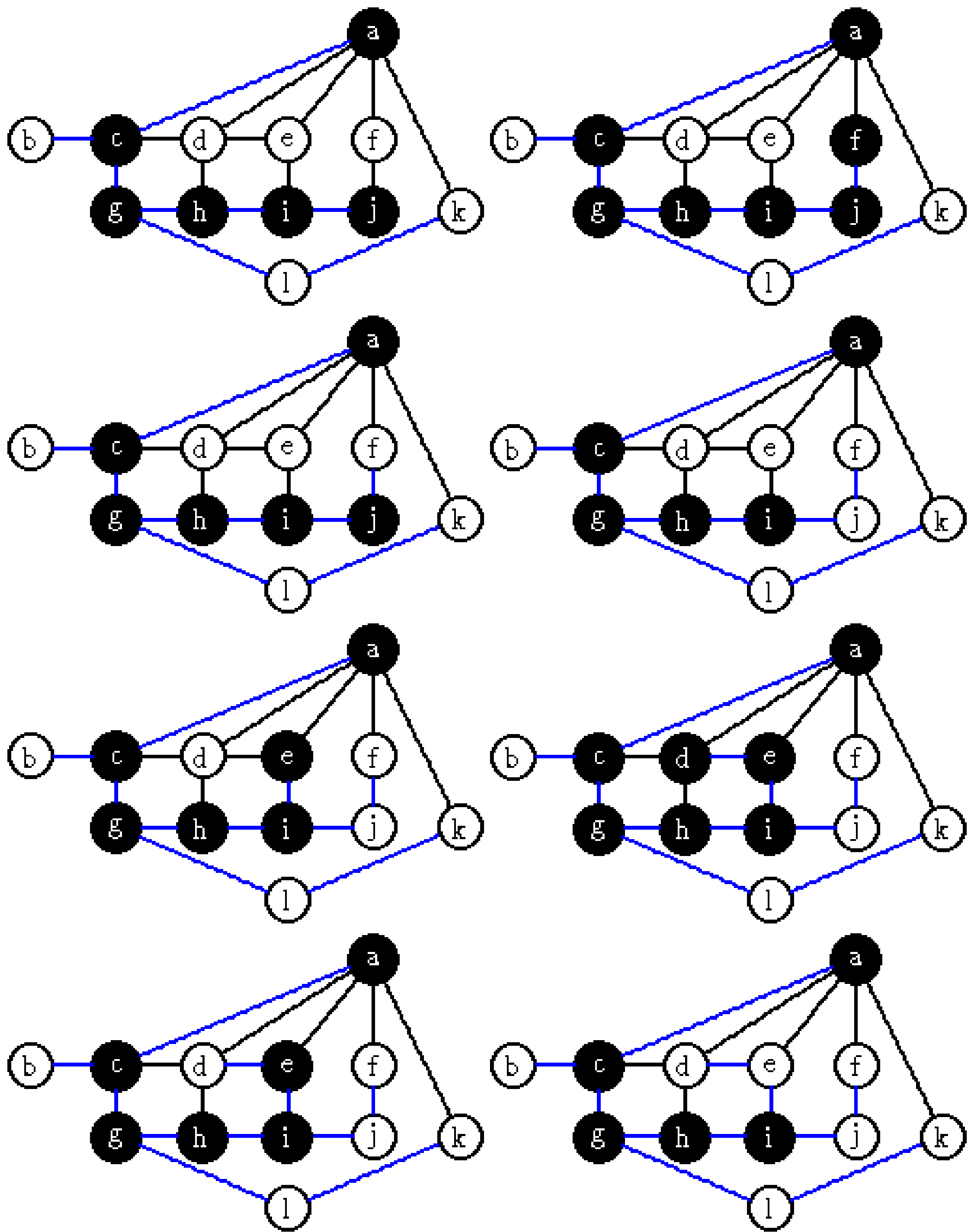

a
b
c
d
e
f
g
h
i
j
k
l


a
b
c
d
e
f
g
h
i
j
k
l


a
b
c
d
e
f
g
h
i
j
k
l


a
b
c
d
e
f
g
h
i
j
k
l

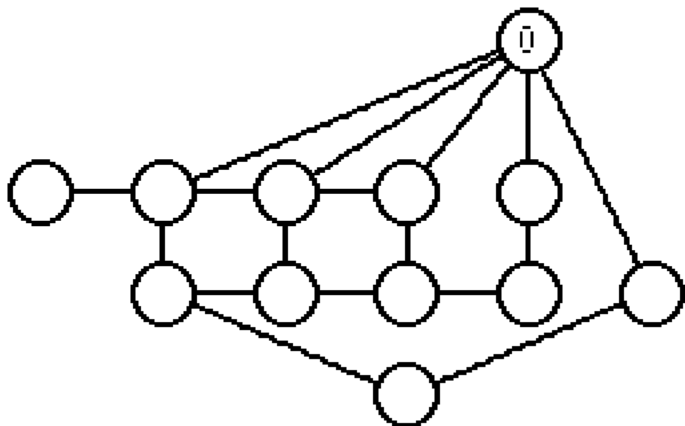

a
b
c
d
e
f
g
h
i
j
k
l

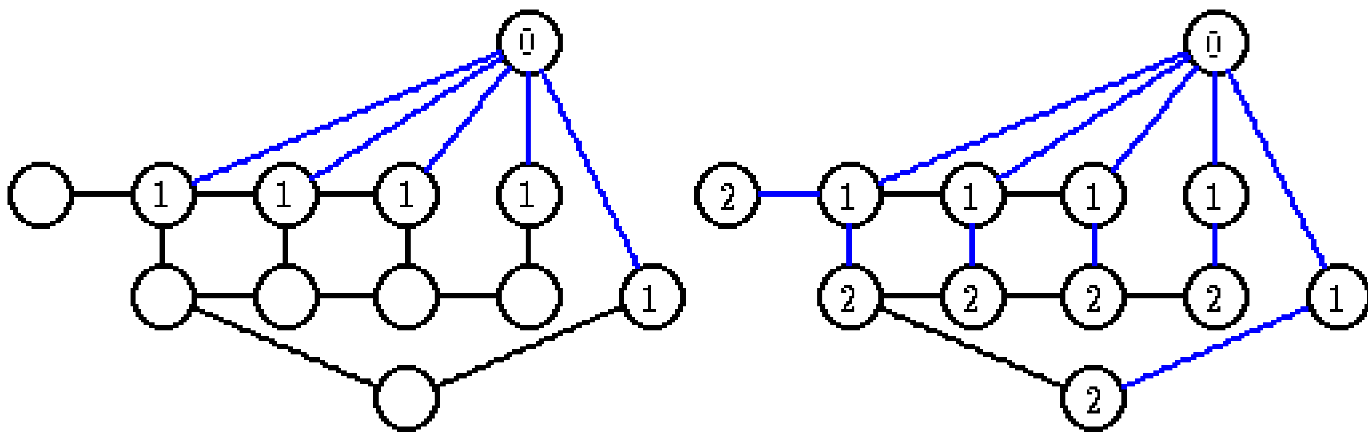


a
b
c
d
e
f
g
h
i
j
k
l

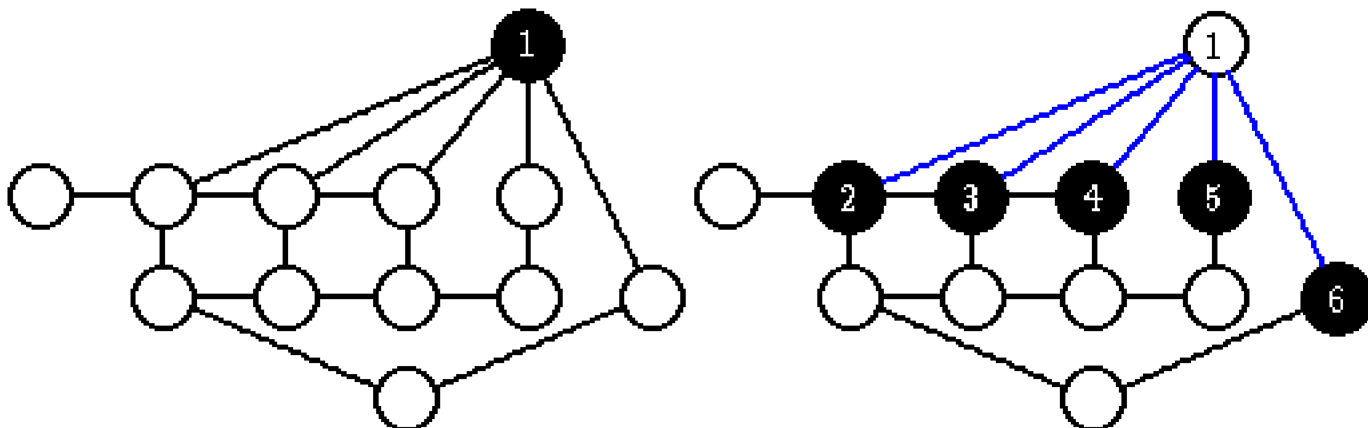
14.2 Breadth-First Traversal

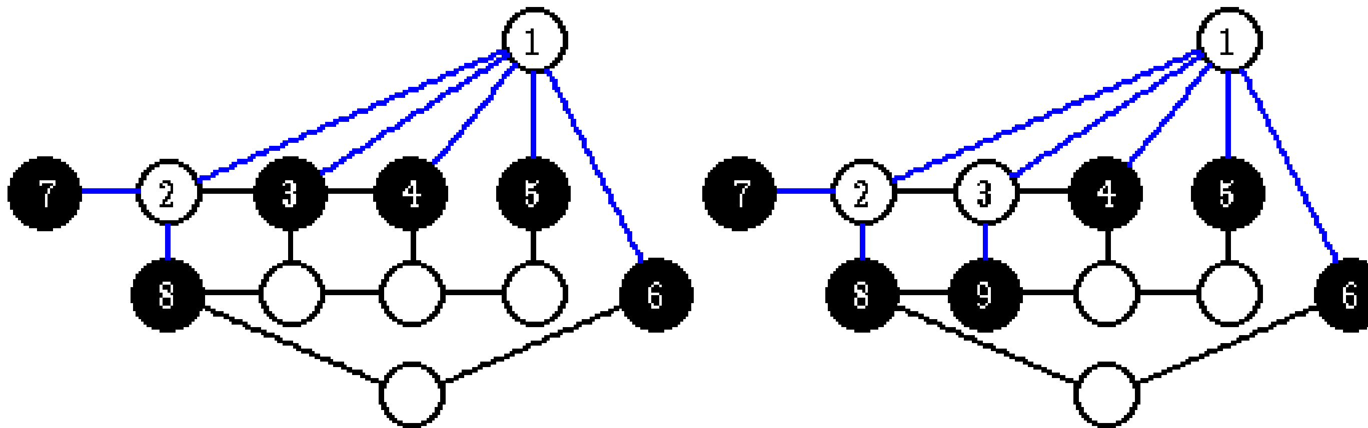
Visit the nodes at level i before the nodes of level i+1.

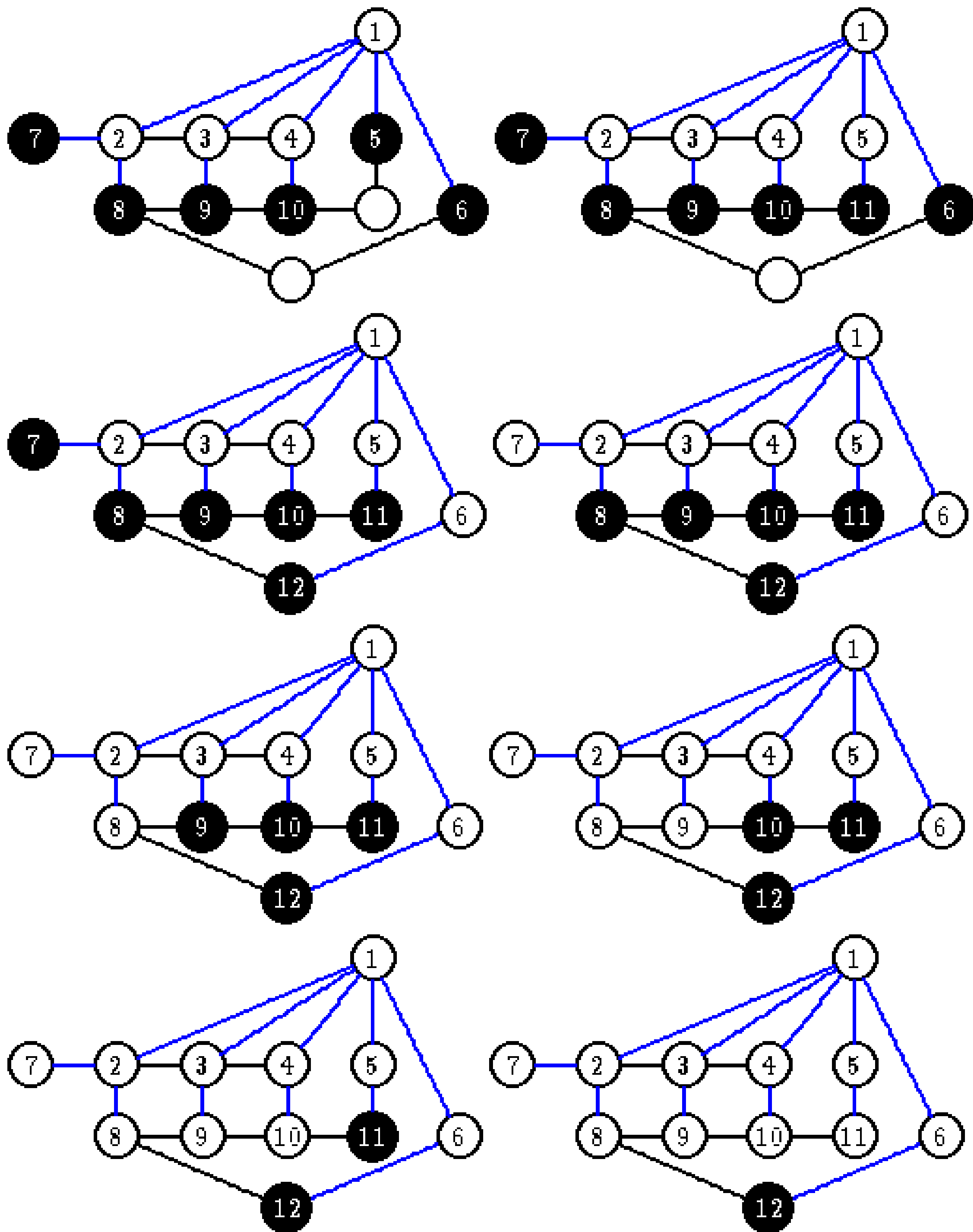




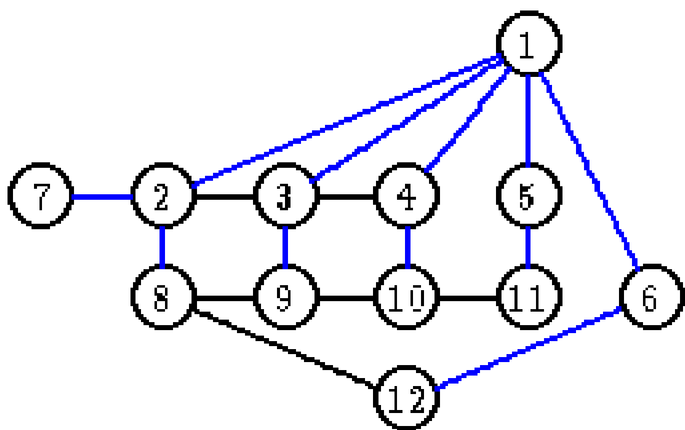

```
visit(start node)
queue <- start node
WHILE queue is nor empty DO
  x <- queue
  FOR each y such that (x,y) is an edge
                    and y has not been visited yet DO
    visit(y)
    queue <- y
  END
END
```







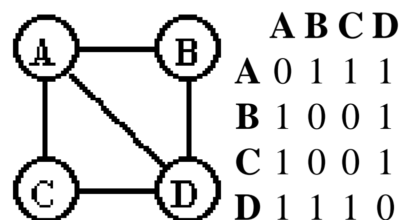
1
7
2
3
4
5
8
9
10
11
6
12



14.3 Representations of Graphs

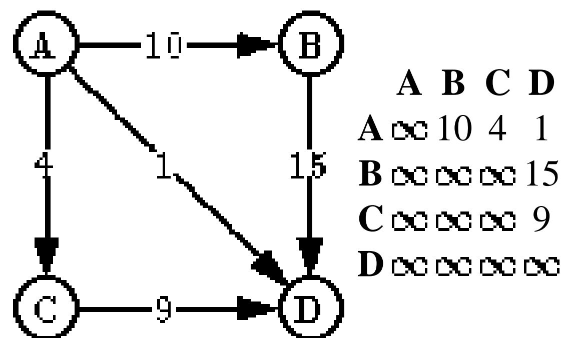
Adjacency Matrices

Graphs G = (V, E) can be represented by **adjacency matrices** $G[v_1..v_{|V|}, v_1..v_{|V|}]$, where the rows and columns are indexed by the nodes, and the entries $G[v_i, v_j]$ represent the edges. In the case of unlabeled graphs, the entries are just boolean values.



	A	B	C	D
A	0	1	1	1
B	1	0	0	1
C	1	0	0	1
D	1	1	1	0

In case of labeled graphs, the labels themselves may be introduced into the entries.

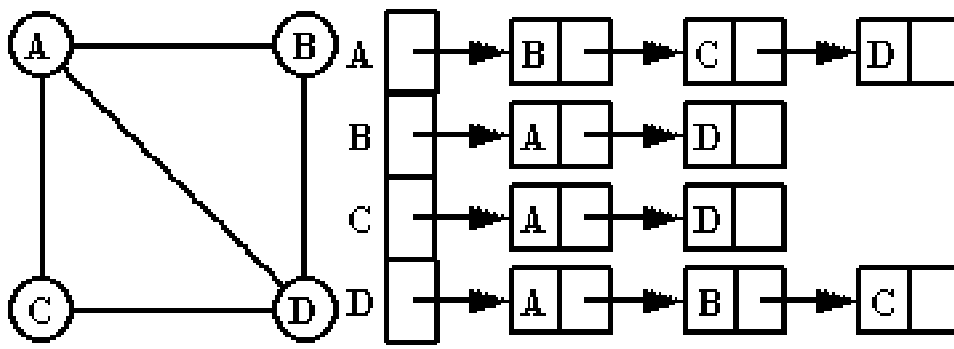


	A	B	C	D
A	∞	10	4	1
B	∞	∞	∞	15
C	∞	∞	∞	9
D	∞	∞	∞	∞

Adjacency matrices require $O(|V|^2)$ space, and so they are space-efficient only when they are dense (that is, when the graphs have many edges). Time-wise, the adjacency matrices allow easy addition and deletion of edges.

Adjacency Lists

A representation of the graph consisting of a list of nodes, with each node containing a list of its neighboring nodes.



This representation takes O(|V | + |E|) space.

14.4 Demo Applets

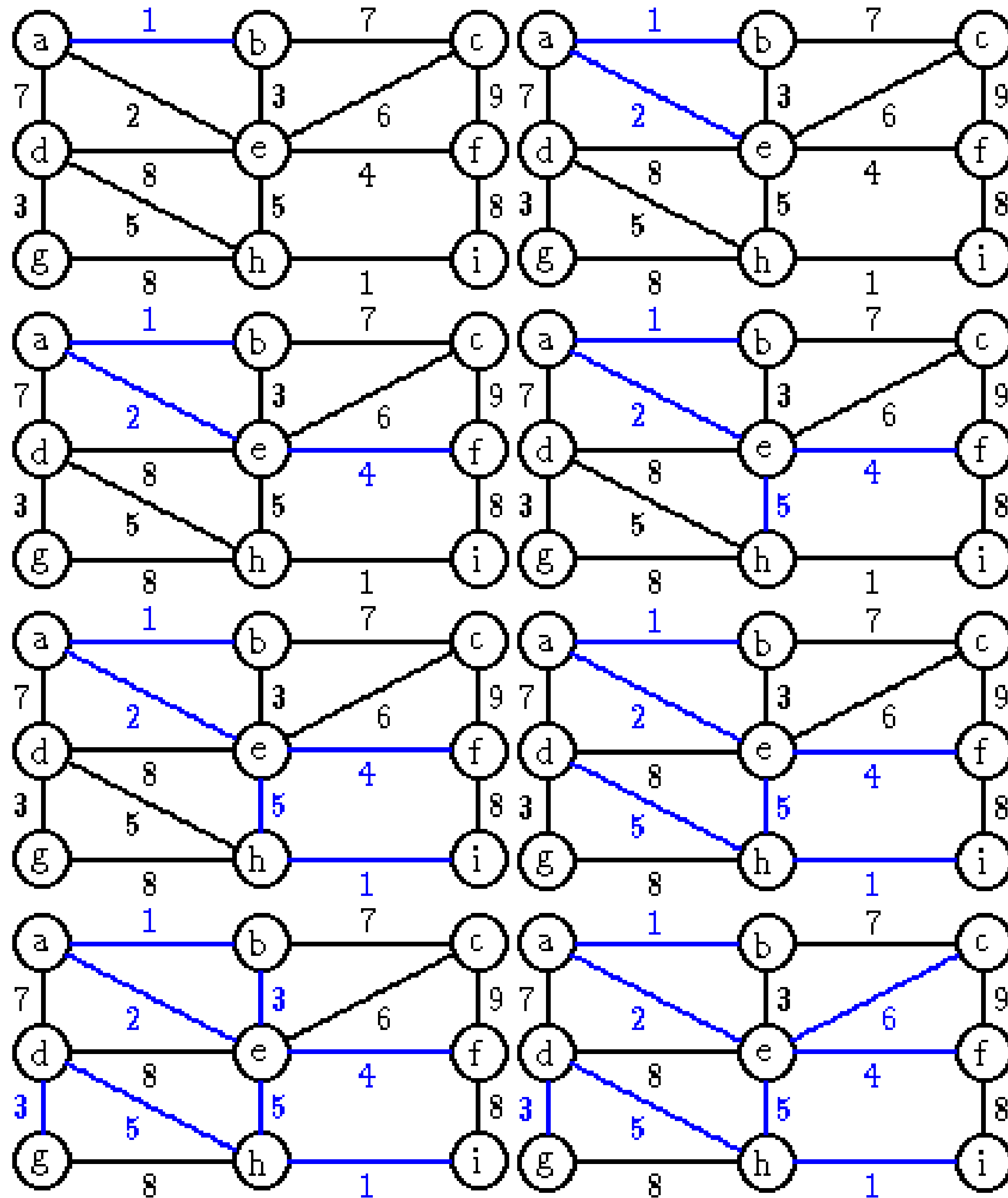
- dft applet
- bft applet

[next] [front] [up]

Chapter 15
Least Cost Spanning Trees

15.1 Prim s Algorithm

At each stage, include the least cost edge whose addition to the selected edges forms a tree.



The approach takes O(|E| log |V |) time for a graph G = (V, E). The time requirement is driven by the algorithm used for selecting the edges to be added in each stage.

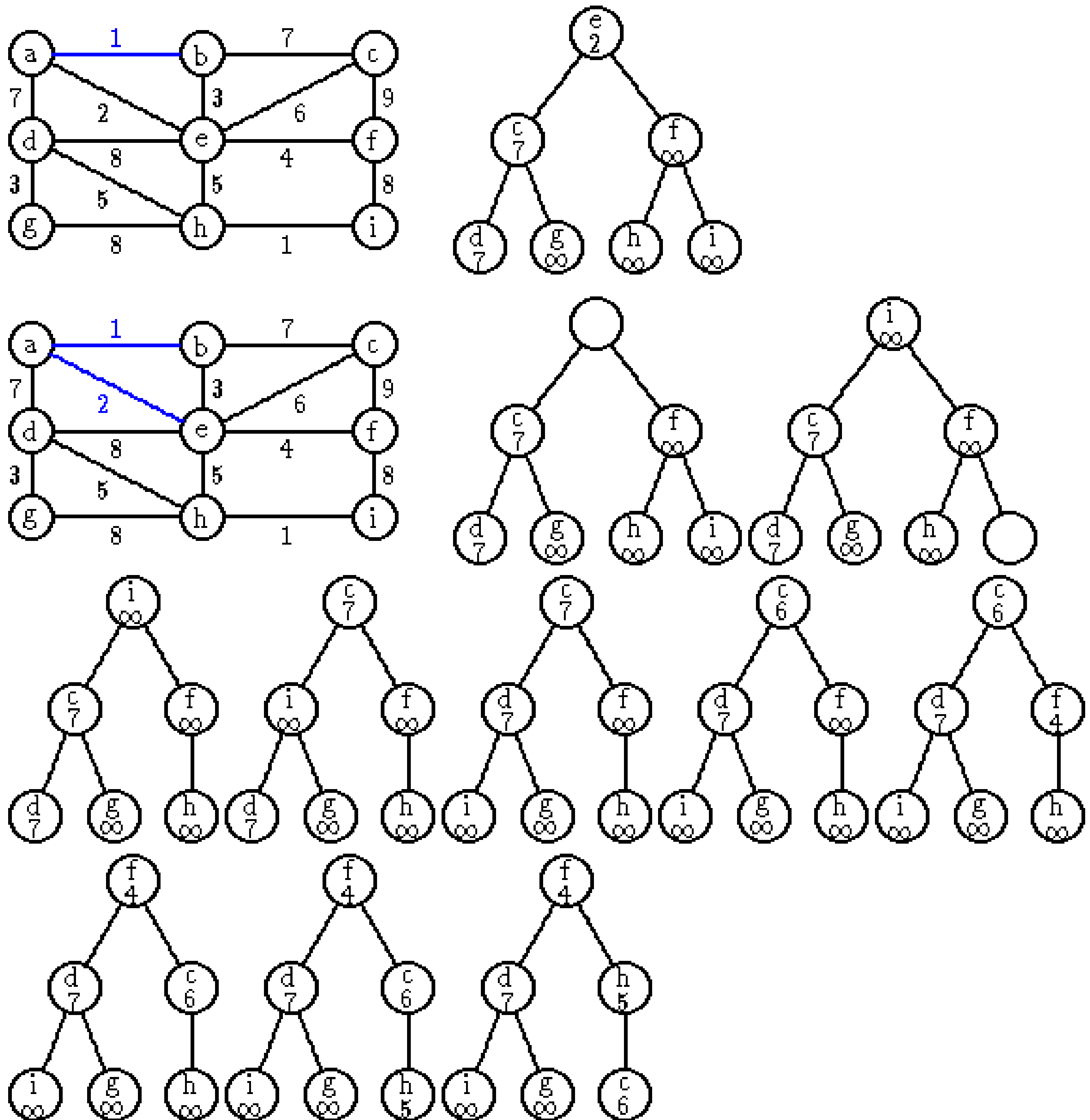
a. Assume a priority queue Q for the nodes of the graph that have not been chosen yet.
b. For the priority evaluation, assign each node in Q the least cost of adding it into V - Q (i.e., the nodes already selected for inclusion the spanning tree T).

c. Initially, the priority queue is a complete tree, and each node carries a priority value ∞ larger than the cost of all the edges.

d. Upon removing a node u from the queue Q, and adding it into the spanning tree T, the priority value is modified for each node v in Q that is adjacent to u. The changing of the priority value of a node may require the shifting of the up or down in the tree.

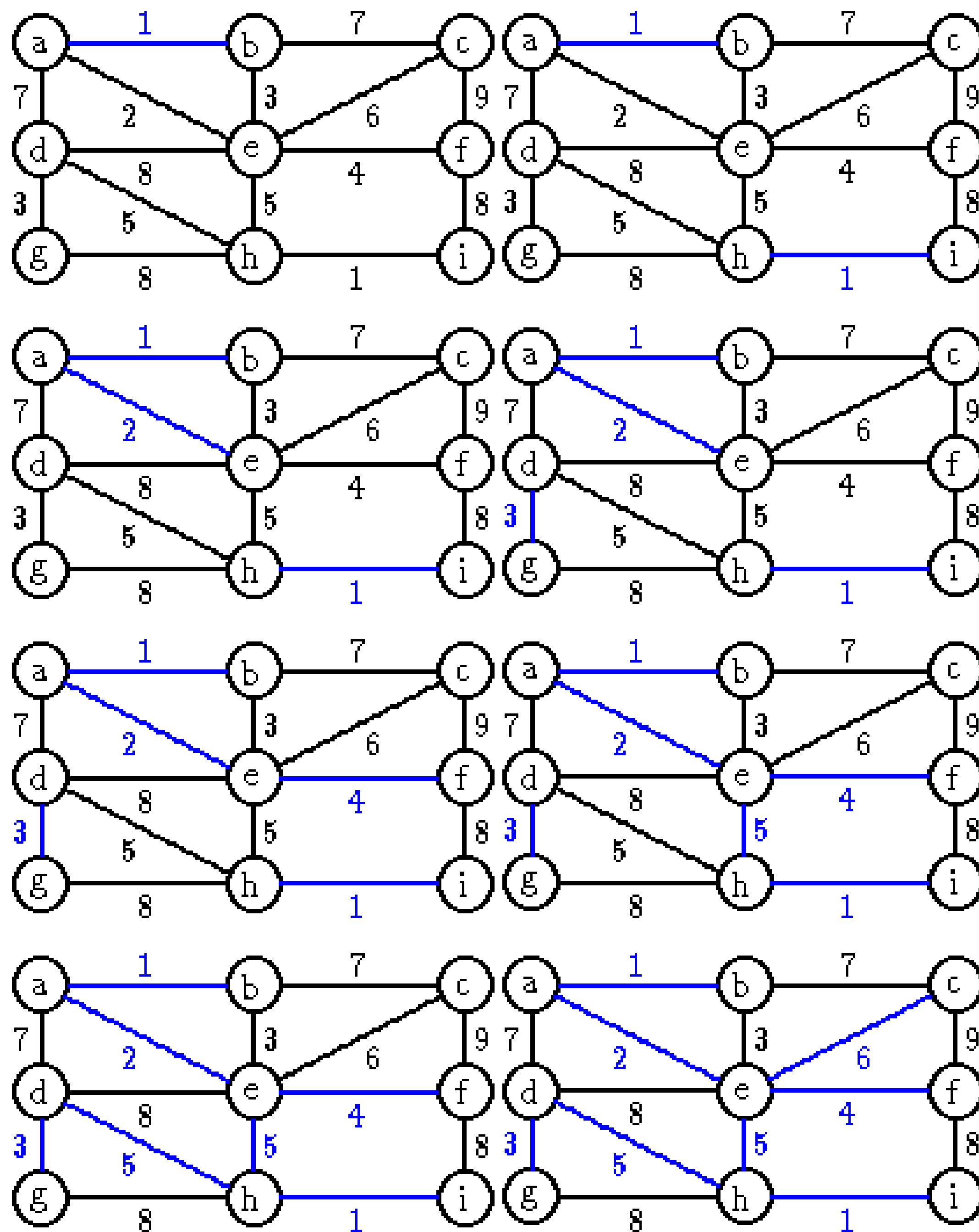
The initialization of the the priority queue takes O(|V |) time. Each deletion of a node from the queue takes O(log |V |) time, and so O(|V | log |V |) time takes to empty the queue. A modification of a priority value of a node in the queue takes O(log |V |) time, and |E| such changes are needed.

Consequently, the algorithm takes O(|V |) + O(|V | log |V |) + O(|E| log |V |) = O(|E| log |V |) time.



15.2 Kruskal s Algorithm

A greedy algorithm: Visit the edges in order of increasing cost. Include just those edges that don t create cycles.

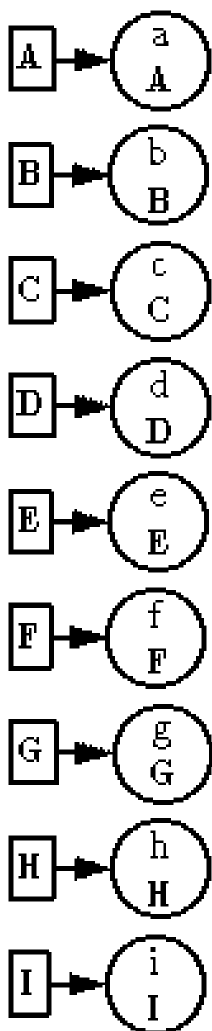


Kruskal s algorithm requires O(|E| log |E|) time for sorting the edges, O(|E|) time to traversing them, and O(|V | log |V |) time for checking the existence of cycles (employing the union-find algorithm below). Hence, the algorithm is of time complexity O(|E| log |E|).

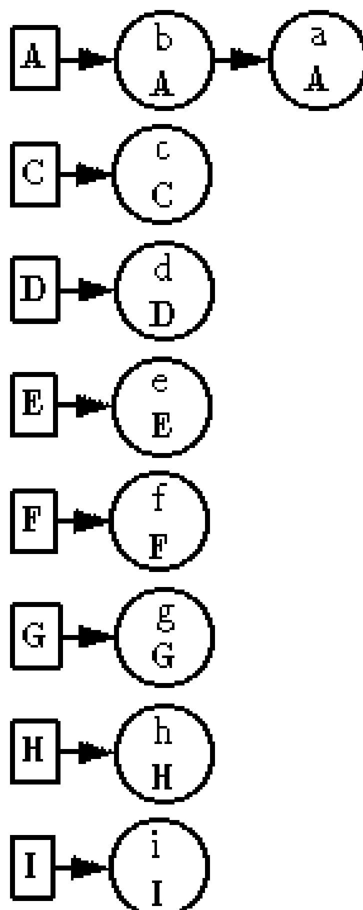
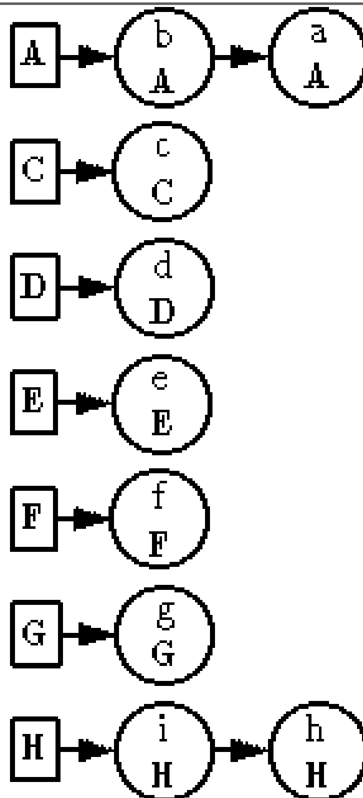
15.3 Union-Find Algorithm

Kruskal s algorithm relies on a union-find algorithm for checking cycles. The union operation unites equivalent sets of elements, and the find operation determines the sets in which the elements reside.

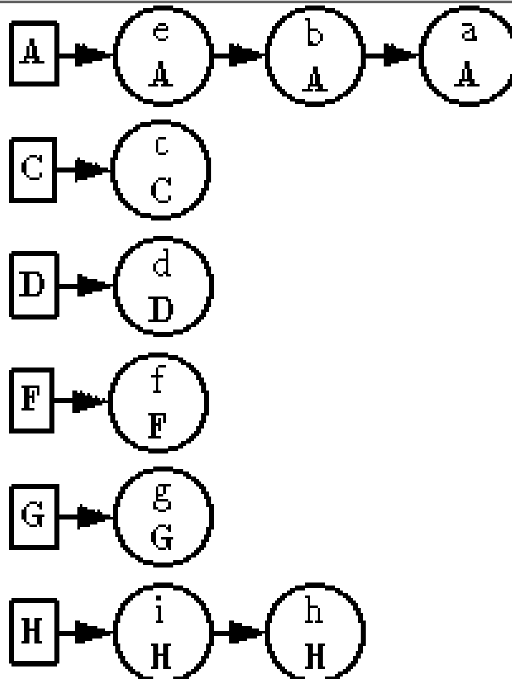
- Components are represented by linked lists, with each element carrying the name of its component. Initially, each of the components consists of a single node.



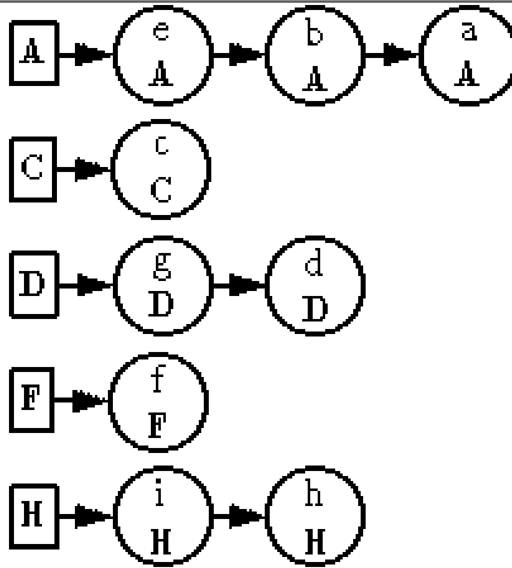
- Merging of components is accomplished by moving the elements of the smaller set into the bigger one. To that end, the shorter linked list is traversed to change the name of the set recorded in the elements, and the the last element in the shorter linked list is set to point to the first element of the longer list.

A ∪ B	
H ∪ I	

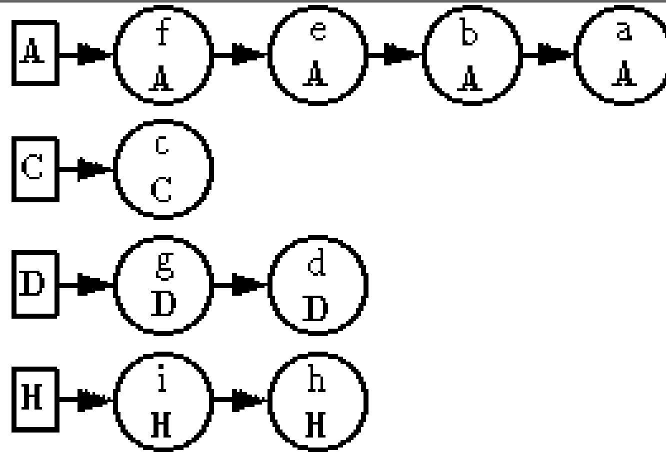
A ∪ C

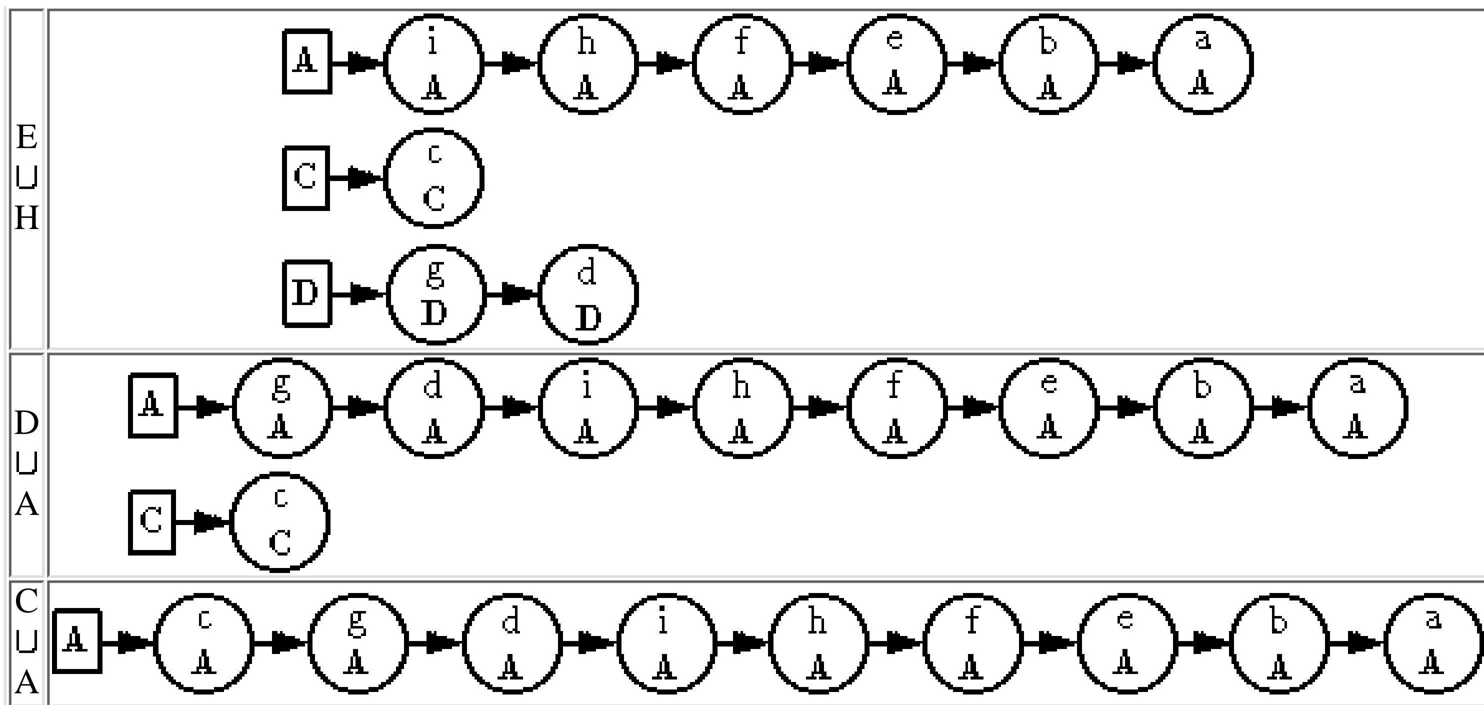


D ∪ G



E ∪ F





- The time to merge two components is linear in the number of elements in the shorter list
- Each node can be visited at most O(log |V |) times, since each merge at least doubles the cardinality of the shorter list. Hence, the algorithm is of time complexity O(|V | log |V |)

15.4 Demo Applets and Resources

- Prim s algorithms
- applet for Kruskal s algorithm
- Kruskal s algorithm
- Kruskal s and Prim s algorithms
- Info about the union-find algorithm

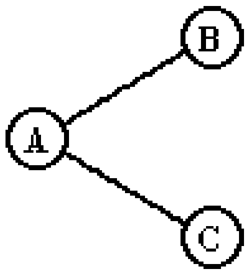
15.5 Assignment #7 (due We, Nov 10)

Write a (real) program that reads the edges of a directed graph, internally stores the graph using the adjacency list approach, and then prints

a. The list of nodes, and the list of nodes that are adjacent to each of these node.
b. An adjacency matrix of the graph.
c. A topologically sorted list of the nodes of the graph, if the graph is acyclic.

   A list of nodes is said to be **topologically sorted**, if node `A` precedes node `B` in the listing whenever there exists an edge from node `A` to node `B`.

   The lists of nodes A,B,C and A,C,B are the only topologically sorted lists for the the following graph. The list B,A,C is an example of one which is not topologically ordered.



Hint: Assume an indegree field in each node.

Your program should be well documented, and accept files of pairs of integers for input. Each pair represents an edge.

Submit a printout of the program, of an input file, and of the corresponding output.

[prev] [prev-tail] [front] [up]

[next] [tail] [up]

Chapter 16
Brute Force Algorithms

These are algorithms that use obvious non-sophisticated approaches to solve the problems in hand. Typically they are useful for small domains, due to large overheads in sophisticated approaches.

16.1 Sequential Search

A search algorithm which traverses all the elements of the given set.

A more sophisticated approach, in cases that many searches are made, is to first sort the sets and then use binary search.

16.2 Hamilton Circuits

A Hamilton Circuit is a closed path in a graph, in which each node appears exactly once. The Hamilton Circuit problem asks whether such a path exists for the given graph.

The Hamilton Circuit problem is an NP-hard problem, and so no polynomial time algorithm is available for it. In particular, a brute force approach of examining all the possible routes requires the inspection of $(n - 1)!$ candidates.

[next] [front] [up]

Chapter 17
Greedy Algorithms

A greedy algorithm repeatedly executes a procedure which tries to maximize the return based on examining local conditions, with the hope that the outcome will lead to a desired outcome for the global problem. In some cases such a strategy is guaranteed to offer optimal solutions, and in some other cases it may provide a compromise that produces acceptable approximations.

Typically, the greedy algorithms employ simple strategies that are simple to implement and require minimal amount of resources.

17.1 Prim s Minimal Spanning Tree Algorithm

Grows a tree by adding in each step a branch with minimal cost. Despite this short sight approach, the outcome is optimal.

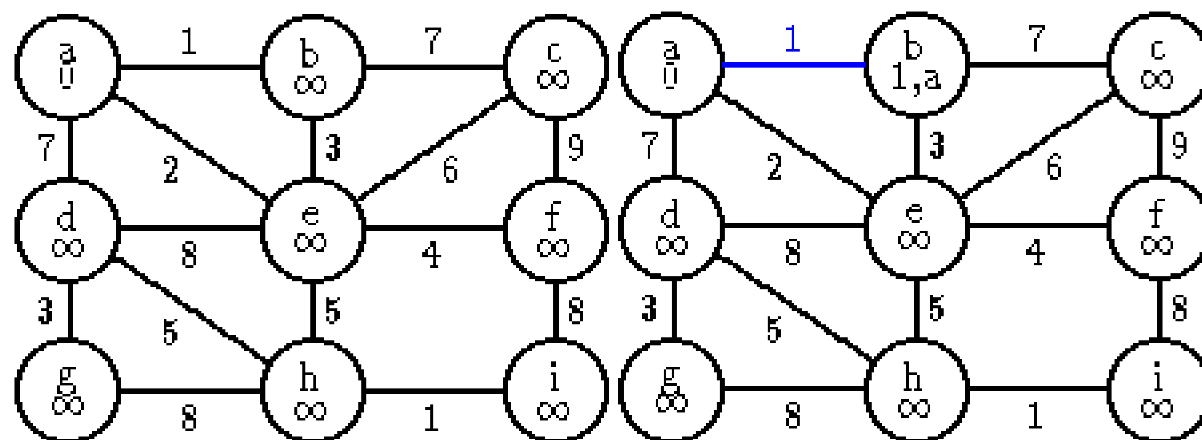
17.2 Kruskal s Minimal Spanning Tree Algorithm

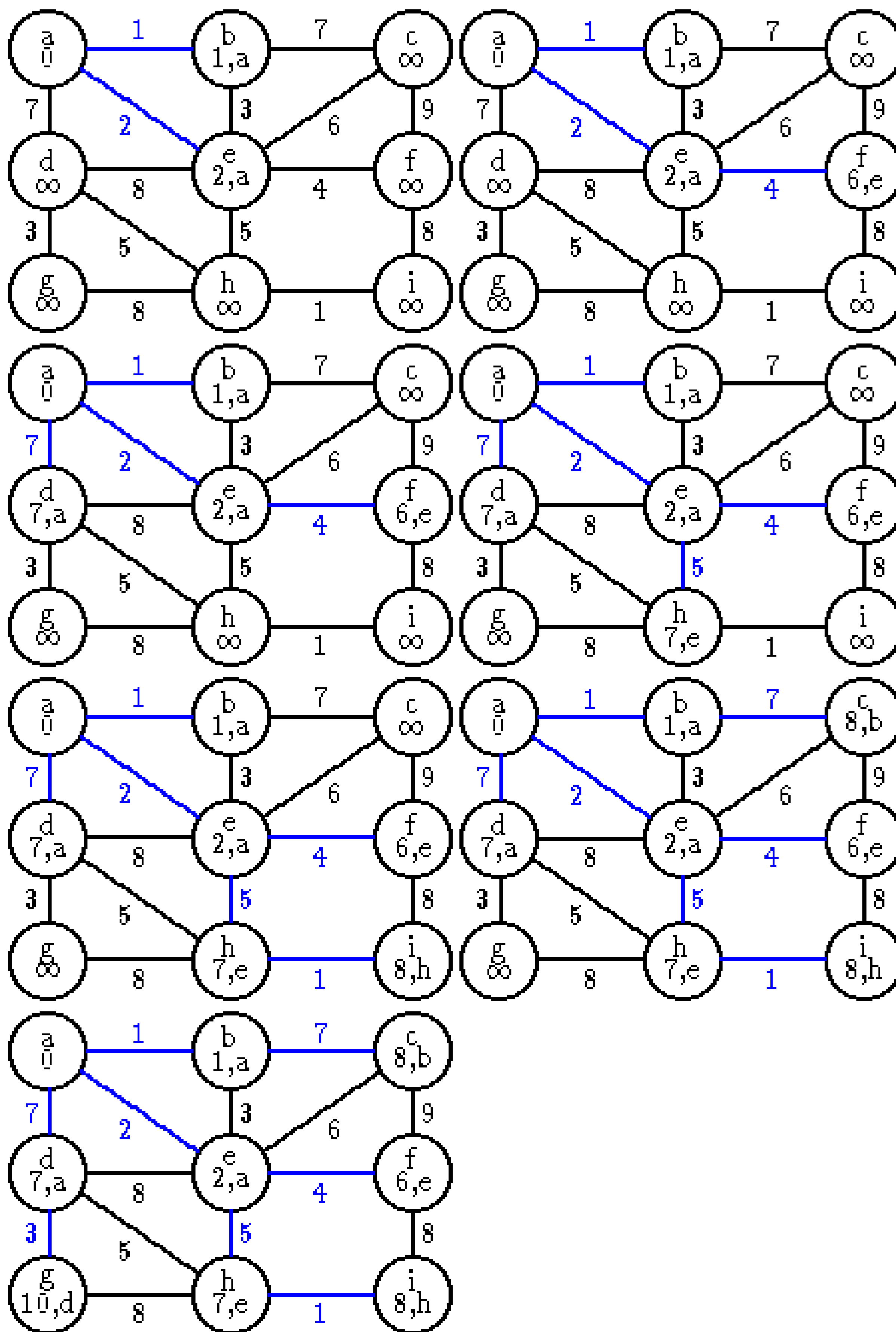
At each stage, the edge with the least cost is processed.

17.3 Dijkstra s Single-Source Shortest Paths Algorithm

Establish the shortest path between a single source node and all of the other nodes in a graph.

Greedy algorithm: Add an edge that connects a new node to those already selected. The path from the source to the new node should be the shortest among all the paths to nodes that have not been selected yet.





The algorithm has time complexity O(|V |^2). (Note the similarity and difference relatively to Prim s algorithm.)

17.4 Coin Change

The problem asks to provide a change for a specified amount, using a minimal number of coins. A greedy algorithm may at each stage employ the criteria of providing the largest available coin which is not greater than the amount still owed.

17.5 Egyptian Fractions

Each fraction can be expressed as a sum of different fractions with unit numerators. Such representations have been used by the ancient Egyptians.

$$\frac{87}{110} = \frac{1}{2} + \frac{1}{5} + \frac{1}{11}$$

A given fraction might have more than one Egyptian representation. A greedy algorithm, for finding such a representation, can at each stage add to the sum the largest unit fraction which does not cause the sum to exceed the fraction. Fibonacci proved that this greedy algorithm is guaranteed to halt.

17.6 Map Coloring

The map coloring problem asks to assign colors to the different regions, where adjacent regions are not allowed to have the same color. There is no general algorithm to assign minimal number of colors, yet each map has an assignment which uses no more than four colors.

A greedy approach repeatedly choose a new color, and assign it to as many regions as possible.

17.7 Voting Districts

Given an integer number N, and a map in which each region is assigned an integer number (= population size), the problem asks to create N district with minimal deviation of population. The regions in each district must be connected.

A greedy algorithm can start by choosing the N biggest regions as cores of the different districts. Then, in each iteration, the largest unassigned region, among those that are adjacent to assigned regions, is assigned to the district with the smallest population.

17.8 Vertex Cover

A vertex cover of a graph G = (V, E) is a subset of nodes, where the nodes of the subset are attached all the edges of the graph. The problem of finding a vertex cover of minimum size is NP-complete, implying the non-existence of efficient algorithms for solving the problem accurately.

A greedy algorithm may provide approximated solutions, by selecting in each stage a vertex which covers the most edges that have not been covered yet.

17.9 0/1 Knapsack

Given a set of N item (v_i, w_i), and a container of capacity C, find a subset of the items that maximizes the value $\sum v_i$ while satisfying the weight constraints $\sum w_i \leq C$. This problem is a NP-hard problem, requiring an exhaustive search over the 2^N possible combinations of items, for determining an exact solution.

A greedy algorithm may consider the items in order of decreasing value-per-unit weight v_i/w_i. Such an approach guarantees a solution with value no worst than 1/2 the optimal solution.

17.10 Demo Applets

- Dijkstra s Algorithm
- Dijkstra s Algorithm

[next] [prev] [prev-tail] [front] [up]

CIS 680: DATA STRUCTURES

Eitan Gurari, Autumn 1999

Data abstraction; introduction to algorithm analysis; data structures and file structures, including lists, trees, and graphs; searching and sorting (OSU Bulletin, OSU Schedule)

TEXT

S. Sahni, *Data Structures, Algorithms, and Applications in C++*, McGraw-Hill, 1998.

lecture notes -- pointers -- sample midterm exam -- sample final exam -- midterm exam -- final exam

PREREQUISITES: 560 and 570; Stat 427 or equiv; and Math 366; or grad standing.

GRADING POLICY

- 25% Homework (10 assignments)
- 30% Midterm exam (Mo, October 25)
- 45% Final exam (Mon, Dec 6, 11:30 - 1:18)

Notes:

a. The exams will be with open notes and open books.
b. No homework will be accepted after the end of class on due date. The assignments are due in class; don t turn them in my office or my mailbox. Also, keep all your graded works until you receive the final grade for the course.
c. Those who graduate this quarter will have their final exam on We, Dec 1, 11:30am-1:20am.
d. Exceptions to the above dates of exams must be arranged with the instructor during the first week of the quarter.

TIME/ROOM MWF, 12:30-1:20, DL 357

INSTRUCTOR Eitan Gurari, Dreese 495, 292-3083; email: gurari@cis.ohio-state.edu; office hours: MF, 11:30-12:20, and by appointment

GRADER Xiaojin Wang, Caldwel 414, 298-4460, xw@cis.ohio-state.edu, Office Hours: Mo, 3:30-5:00, and by appointment.

ASSIGNMENTS #1 (due We, Sept 29)
#2 (due We, Oct 6)
#3 (due We, Oct 13)
#4 (due We, Oct 20)
#5 (due Fr, Oct 29)
#6 (due We, Nov 3)
#7 (due We, Nov 10)

#8 (due We, Nov 17)
#9 (due We, Dec 1)

http://www.cis.ohio-state.edu/~gurari/course/cis680/cis680.html

Chapter 19
Backtracking Algorithms

19.1 Solution Spaces

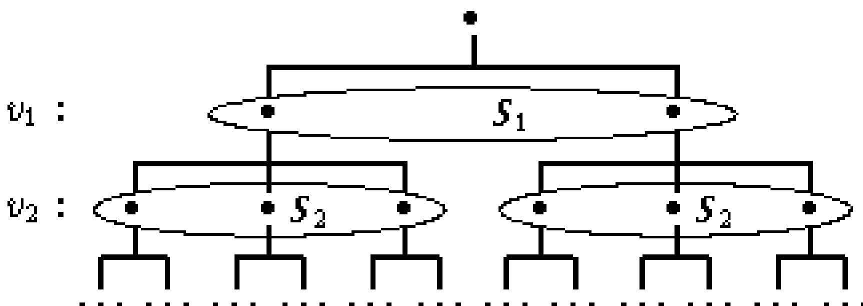
Backtracking is a refinement of the brute force approach, which systematically searches for a solution to a problem among all available options. It does so by assuming that the solutions are represented by vectors $(v_1, ..., v_m)$ of values and by traversing, in a depth first manner, the domains of the vectors until the solutions are found.

When invoked, the algorithm starts with an empty vector. At each stage it extends the partial vector with a new value. Upon reaching a partial vector $(v_1, ..., v_i)$ which can t represent a partial solution, the algorithm backtracks by removing the trailing value from the vector, and then proceeds by trying to extend the vector with alternative values.

```
ALGORITHM try(v1,...,vi)
   IF (v1,...,vi) is a solution THEN RETURN (v1,...,vi)
   FOR each v DO
      IF (v1,...,vi,v) is acceptable vector  THEN
        sol = try(v1,...,vi,v)
        IF sol != () THEN RETURN sol
      END
   END
   RETURN ()
```

If S_i is the **domain** of v_i, then $S_1 \times ... \times S_m$ is the **solution space** of the problem. The **validity criteria** used in checking for acceptable vectors determines what portion of that space needs to be searched, and so it also determines the resources required by the algorithm.

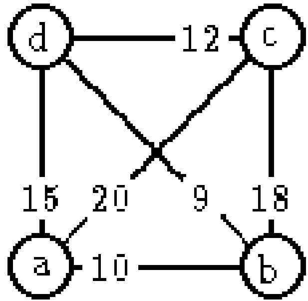
The traversal of the solution space can be represented by a depth-first traversal of a tree. The tree itself is rarely entirely stored by the algorithm in discourse; instead just a path toward a root is stored, to enable the backtracking.



19.2 Traveling Salesperson

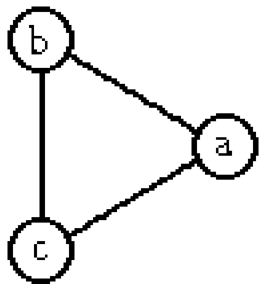
The problem assumes a set of n cities, and a salesperson which needs to visit each city exactly once and return to the base city at the end. The solution should provide a route of minimal length.

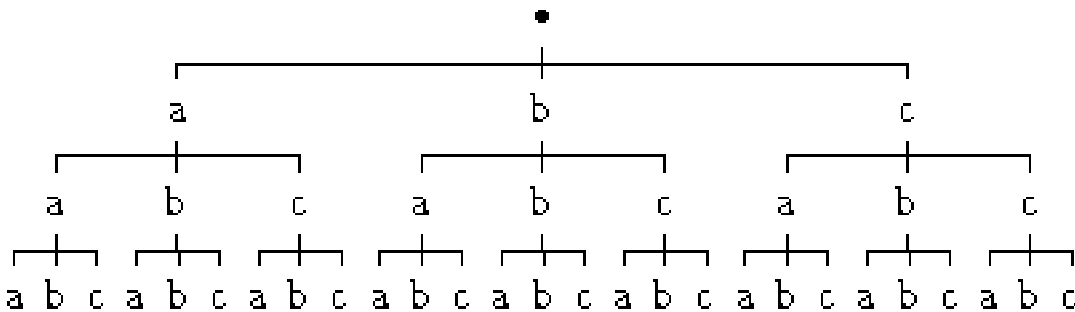
The route (a, b, d, c) is the shortest one for the following one, and its length is 51.



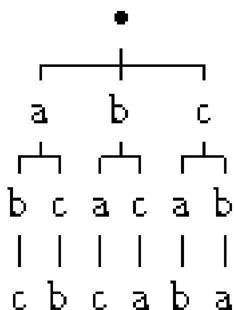
The traveling salesperson problem is an NP-hard problem, and so no polynomial time algorithm is available for it. Given an instance G = (V, E) the backtracking algorithm may search for a vector of cities $(v_1, ..., v_{|V|})$ which represents the best route.

The validity criteria may just check for number of cities in of the routes, pruning out routes longer than $|V|$. In such a case, the algorithm needs to investigate $|V|^{|V|}$ vectors from the solution space.



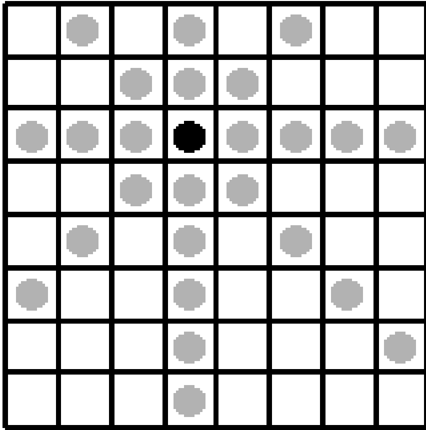


On the other hand, the validity criteria may check for repetition of cities, in which case the number of vectors reduces to $|V|!$.



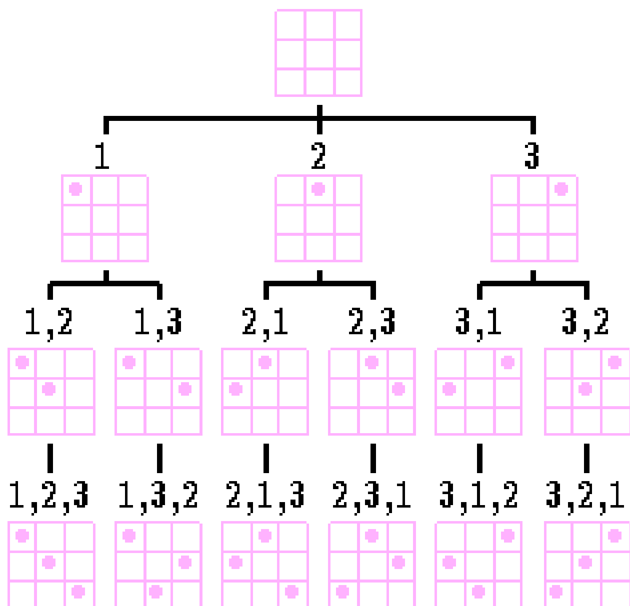
19.3 The Queens Problem

Consider a n by n chess board, and the problem of placing n queens on the board without the queens threatening one another.

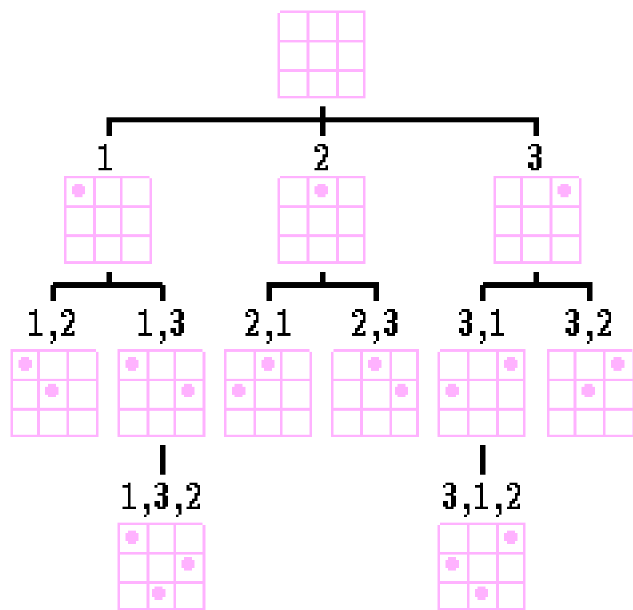


The solution space is $\{1, 2, 3, \ldots, n\}^n$. The backtracking algorithm may record the columns where the different queens are positioned. Trying all vectors $(p_1, \ldots, p_n)$ implies n^n cases. Noticing that all the queens must reside in different columns reduces the number of cases to n!.

For the latter case, the root of the traversal tree has degree n, the children have degree n - 1, the grand children degree n - 2, and so forth.

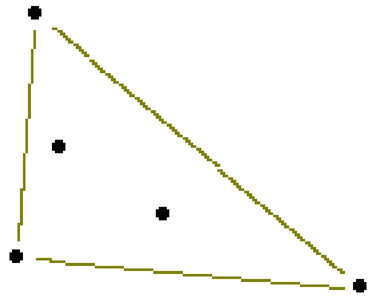


Checking for threatening positions along the way my further reduce the number of visited configurations.

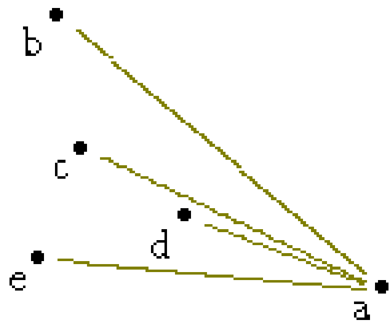


19.4 Convex Hull (Graham s Scan)

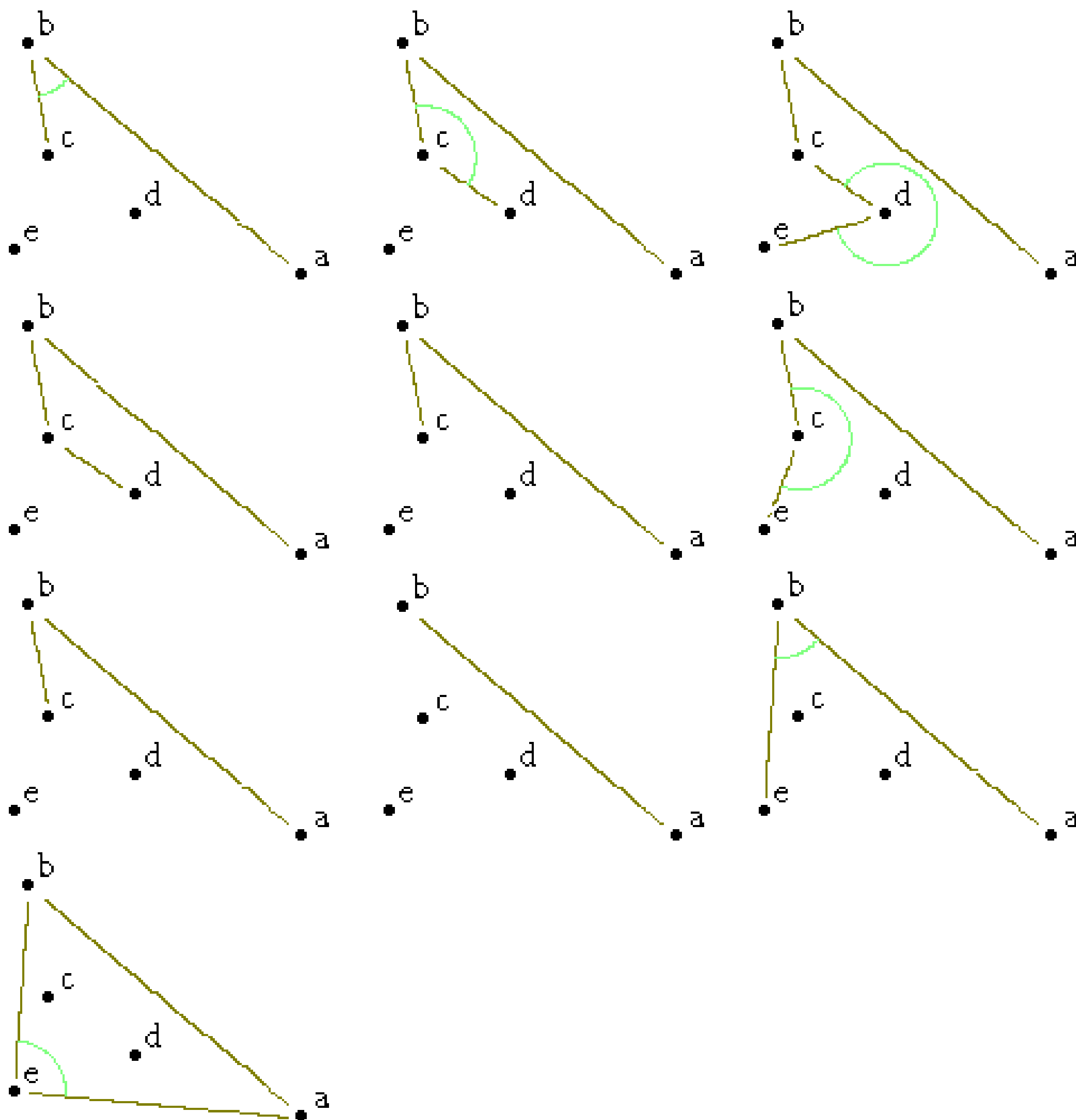
The problem asks to construct the shortest polygon which encloses a given set of points on a plan. Intuitively, the polygon can be determined by placing a rubber around the set.



Determine an extreme point with the largest x coordinate. Sort the points in order of increasing angles, relatively to the extreme point.



Traverse the points in the sorted order, adding them to the partial solution upon making a turn of less than 180 degrees, and backtracking when making a larger turn.

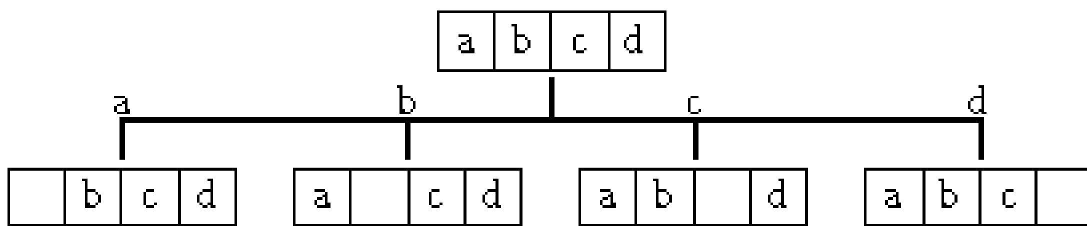


The algorithm requires O(n log n) time for sorting the points, and O(n) time to select an appropriate subset.

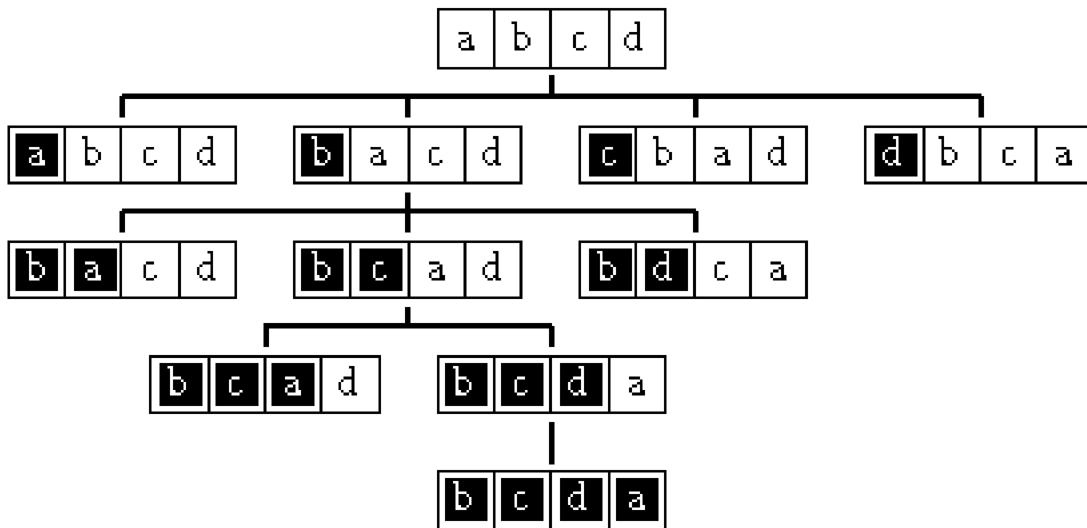
19.5 Generating Permutations

A permutation can be obtained by selecting an element in the given set and recursively permuting the remaining elements.

$$P(a_1, \ldots, a_N) = \begin{cases} a_i, P(a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_N) & \text{if } N > 1 \\ a_N & \text{if } N = 1 \end{cases}$$



At each stage of the permutation process, the given set of elements consists of two parts: a subset of values that already have been processed, and a subset that still needs to be processed. This logical seperation can be physically realized by exchanging, in the i th step, the i th value with the value being chosen at that stage. That approaches leaves the first subset in the first i locations of the outcome.




```
permute(i)
   if i == N  output A[N]
   else
      for j = i to N do
         swap(A[i], A[j])
         permute(i+1)
         swap(A[i], A[j])
```

19.6 Demo Applets

- the queens problem
- 2D convex hulls
- Graham s scan algorithm for convex hulls
- 3D convex hulls

[next] [front] [up]

[next] [tail] [up]

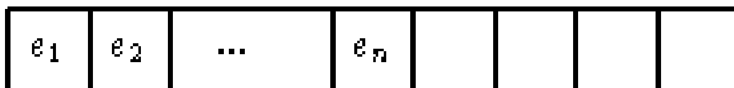
Chapter 2
Lists

2.1 Specification

```
AbstractDataType  LinearList{
  instances
    finite ordered collection of elements
  operations
    create():   create an empty list
    destroy():
    IsEmpty()
    Length():
    Find(k):    find k'th element
    Search(x):  find position of x
    delete():
    insert(x): insert x after the current element element
    output():
}
```

2.2 Array Representation

length 5

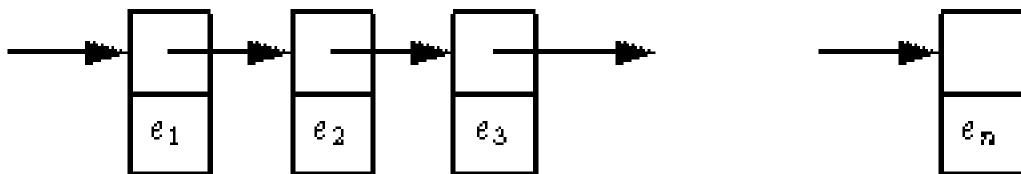


- Operations require simple implementations.
- Insert, delete, and search, require linear time
- Inefficient use of space

2.3 One-way Linked Representation

prev 10

curr 5

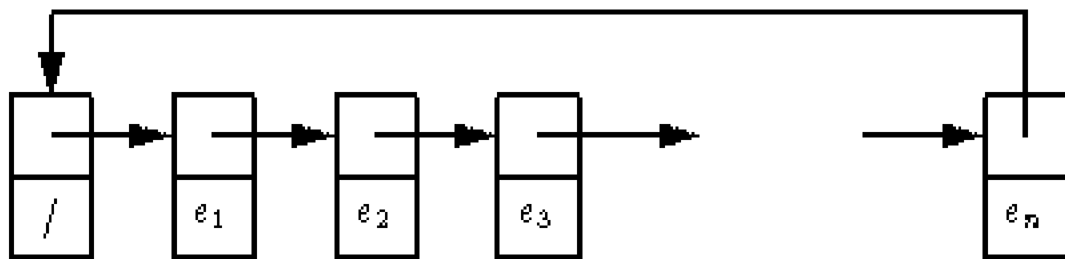


- Insert and delete in O(1) time
- Search in O(n) time
- Memory overhead, but allocated only to entries that are present.

2.4 Circular One-way Linked Representation

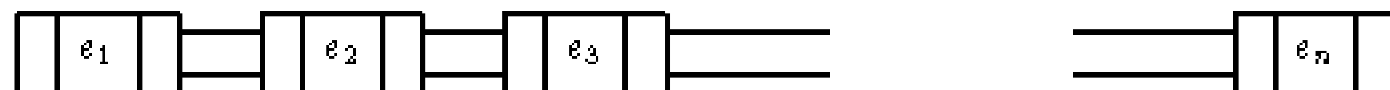
prev 10

curr 5



- The head node, together with the circular representation, simplify the dealing with boundaries conditions

2.5 Doubly Linked Representation



- Insert and delete in (1) time.
- We can also go here for doubly linked list with an additional header node.

[next] [front] [up]

[next] [prev] [prev-tail] [tail] [up]

Chapter 20
Branch-and-Bound Algorithms

A counter-part of the backtracking search algorithm which, in the absence of a cost criteria, the algorithm traverses a spanning tree of the solution space using the breadth-first approach. That is, a queue is used, and the nodes are processed in first-in-first-out order.

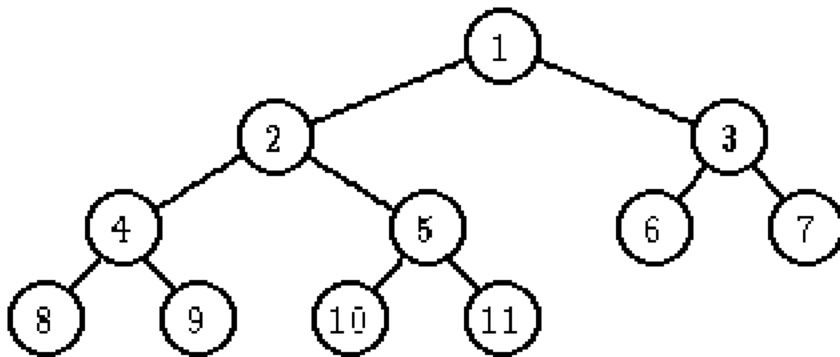
If a cost criteria is available, the node to be expanded next (i.e., the branch) is the one with the best cost within the queue. In such a case, the cost function may also be used to discard (i.e., the bound) from the queue nodes that can be determined to be expensive. A priority queue is needed here.

20.1 Cost-Based Tree Traversal

A function can be considered to be a **tree generator**, if when given a node X and index i it produces the i th child of the node.

The following function produces a complete binary tree of 11 nodes.

$$f(X,i) = \begin{cases} 2X & \text{if } i = 1 \text{ and } 2X < 12 \\ 2X+1 & \text{if } i = 2 \text{ and } 2X+1 < 12 \end{cases}$$



The recursive function provided for deriving permutations is another example of a function that may be used to generate trees.

Besides for a tree generator function, we also need a **cost function** to decide in what order to traverse the nodes when searching for a solution. The algorithm proceeds in the following manner.

1. **Initialization:** The root of the of the tree is declared to be alive.
2. **Visit:** The cost criteria decides which of the live nodes is to process next.
3. **Replacement:** The chosen node is removed from the set of live nodes, and its children are inserted into the set. The children are determined by the tree generator function.
4. **Iteration:** The visitation and replacement steps are repeated until no alive nodes are left.

In the case of **backtracking** the cost criteria assumes a **last-in-first-out** (LIFO) function, which can be realized with a **stack** memory. A **first-in-first-out** cost criteria implies the **FIFO branch-and-bound** algorithm, and it can be realized with **queue** memory. A generalization to **arbitrary** cost criteria is the

basis for the **priority branch-and-bound** algorithm, and a **priority queue** memory can be employed to realize the function.

20.2 Mazes

20.3 Traveling Salesperson Problem

20.4 Job Scheduling

20.5 Integer Linear Programming

[next] [prev] [prev-tail] [front] [up]

[prev] [prev-tail] [tail] [up]

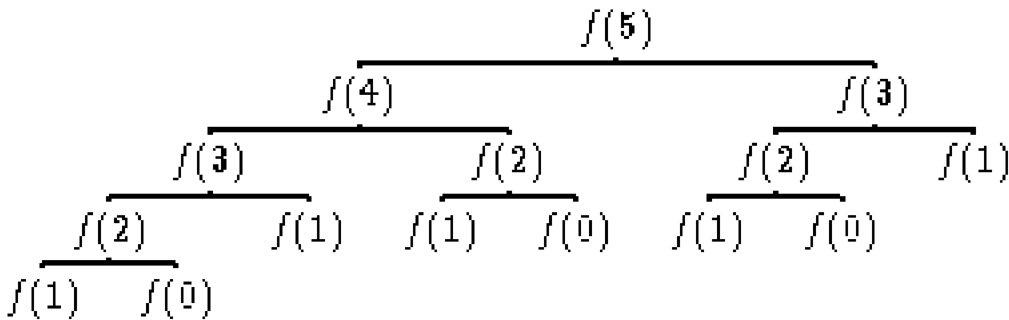
Chapter 21
Dynamic Programming Algorithms

The approach assumes a recursive solution for the given problem, with a bottom-up evaluation of the solution. The subsolutions are recorded (in tables) for reuse.

21.1 Fibonnaci Numbers

$$f(i) = \begin{cases} f(i-1) + f(i-2) & \text{if } i > 1 \\ 1 & \text{otherwise} \end{cases}$$

A top-down approach of computing, say, f(5) is inefficient do to repeated subcomputations.



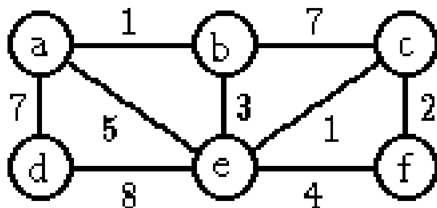
A bottom-up approach computes f(0), f(1), f(2), f(3), f(4), f(5) in the listed order.

21.2 0/1 Knapsack Problem

21.3 All Pairs Shortest Paths (Floyd-Warshall)

A dynamic programming algorithm.

```
FOR k=1 TO n
  FOR i=1 TO n
    FOR j=1 TO n
      c(i,j,k) = min(  c(i,j,k-1),
                       c(i,k,k-1)+c(k,j,k-1)
                    )
```



k = Ø	a b c d e f a 1 ∞ 7 5 ∞ b 7 ∞ 3 ∞ c ∞ 1 2 d 8 ∞ e 4 f
k = {a}	a b c d e f a 1 ∞ 7 5 ∞ b 7,b-a-c ∞,b-a-d 3,b-a-e ∞,b-a-f c ∞,c-a-d 1,c-a-e 2,c-a-f d 8,d-a-e ∞,d-a-f e 4,e-a-f f

k = {a, b}
k = {a, b, c}
k = {a, b, c, d}
k = {a, b, c, d, e}
k = {a, b, c, d, e, f}

21.4 Demo Applets

- Floyd s algorithm

21.5 Assignment #9 (due We, Dec 1)

1. Show the tree of memory states traversed by our recursive permutation algorithm for input a,b,c,d .
2. Provide a recursive backtracking algorithm for the vertex cover problem.
3. Provide a branch-and-bound algorithm for the 0/1 knapsack problem.
4. Provide a dynamic programming algorithm for the coin exchange problem.

[prev] [prev-tail] [front] [up]

[next] [prev] [prev-tail] [tail] [up]

Chapter 3
Stacks

3.1 Specification

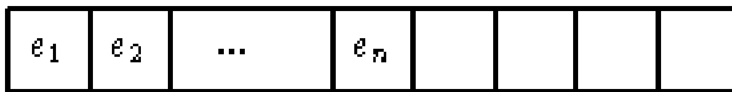
A special case of lists in which restrictions are placed on where insertions and deletions can take place.

```
AbstractDataType Stack{
  instances
     linear list of element, with top and bottom elements
  operations
    Create()
    IsEmpty()
    IsFull()
    Top()
    Add(x)
    Delete()
}
```

3.2 Array Representation

max 10

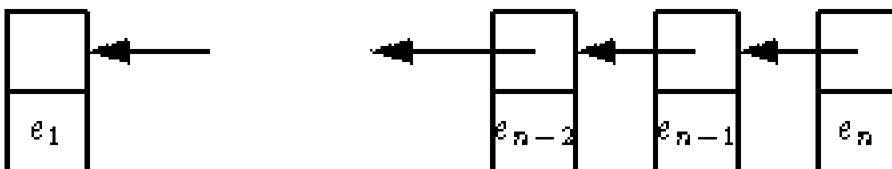
length 5



- Insert and delete in O(1) time.

3.3 One-way Linked Representation

top 5



- Insert and delete in O(1) time

- No advantage to double-linked representation.

[next] [prev] [prev-tail] [front] [up]

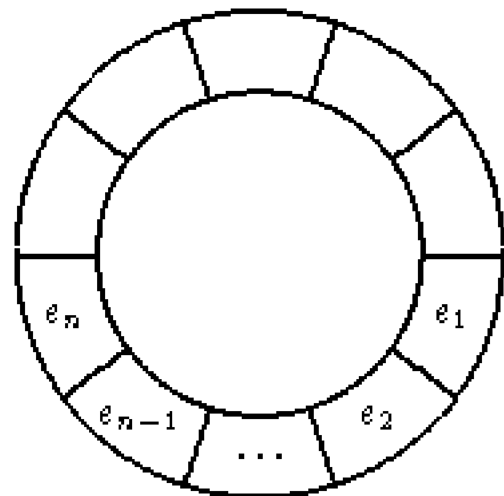
[prev] [prev-tail] [tail] [up]

Chapter 4
Queues

4.1 Specification

```
\Verbatim
AbstractDataType Queue{
  instances
     linear list of element, with top and bottom elements
  operations
    Create()
    IsEmpty()
    IsFull()
    First()
    Last()
    Add(x)
    Delete()
}
```

4.2 Circular Array Representation

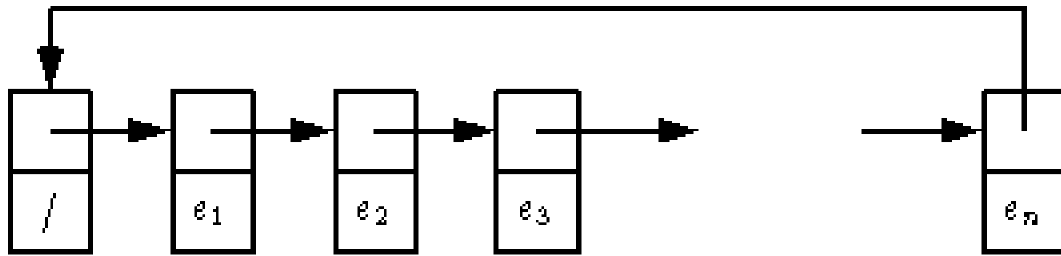


- Insert and delete in O(1) time

4.3 Circular One-way Linked Representation

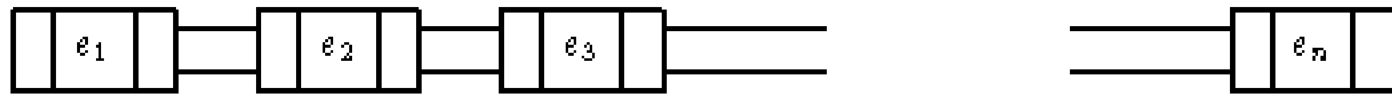
prev 10

curr 5



- Insertion O(1), deletion O(n); or vice-versa

4.4 Doubly Linked Representation



- Insert and delete in (1) time.

[prev] [prev-tail] [front] [up]

[next] [tail] [up]

Chapter 5
General Trees

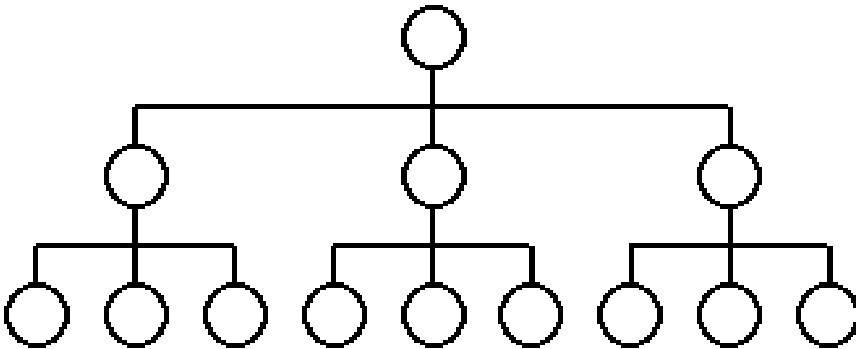
Suitable for representing hierarchies

```
AbstractDataType tree{
  instances
    A set of elements: (1) empty or having a distinguished root element
    (2) each non-root element having exactly one parent element
  operations
    root()
    degree()
    child(k)
}
```

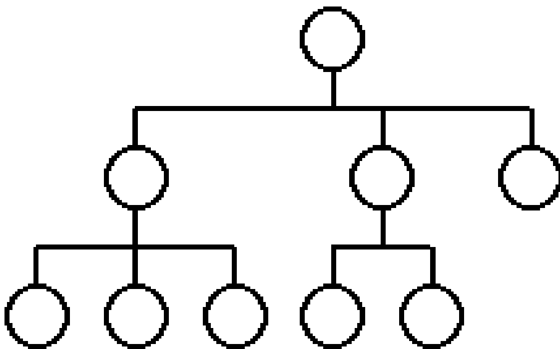
Concepts: tree, subtree, root, leaf, parent, children, level, degree of node, degree of tree, height

5.1 Properties

- **Full tree** A tree with all the leaves at the same level, and all the non-leaves having the same degree.



- **Complete Tree** A full tree in which the last elements are deleted.



- Level h of a full tree has d^{h-1} nodes.
- The first h levels of a full tree have

$$1 + d + d^2 + \cdots + d^{h-1} = \frac{d^h - 1}{d - 1}$$

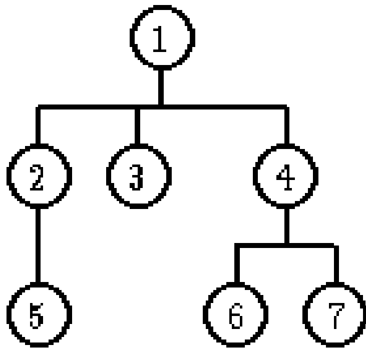
nodes.

- A tree of height h and degree d has at most $d^h - 1$ elements

$$N(h) = \begin{cases} dN(h-1) & \text{if } h > 1 \\ 1 & \text{if } h = 1 \end{cases}$$

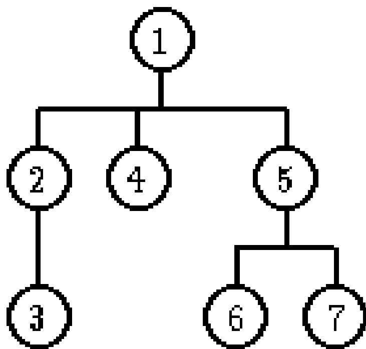
5.2 Traversal

Level order



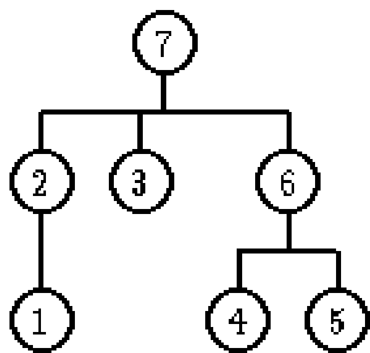

```
x := root()
if( x ) queue (x)
while( queue not empty ){
  x := dequeue()
  visit()
  i=1; while( i <= degree() ){
     queue( child(i) )
  }
}
```

Preorder




```
procedure preorder(x){
  visit(x)
  i=1; while( i <= degree() ){
     preorder( child(i) )
  }
}
```

Postorder




```
procedure postorder(x){
  i=1; while( i <= degree() ){
     postorder( child(i) )
  }
  visit(x)
}
```

Inorder

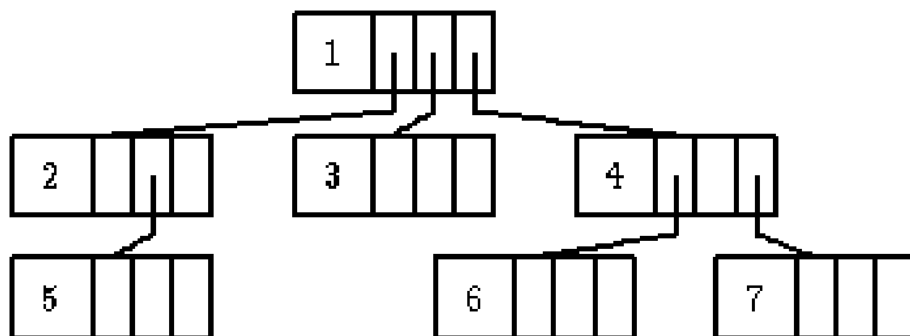
Meaningful just for binary trees.

```
procedure inorder(x){
  if( left_child_for(x) ) { inorder( left_child(x)  ) }
  visit(x)
  if( right_child_for(x) ) { inorder( right_child(x)  ) }
}
```

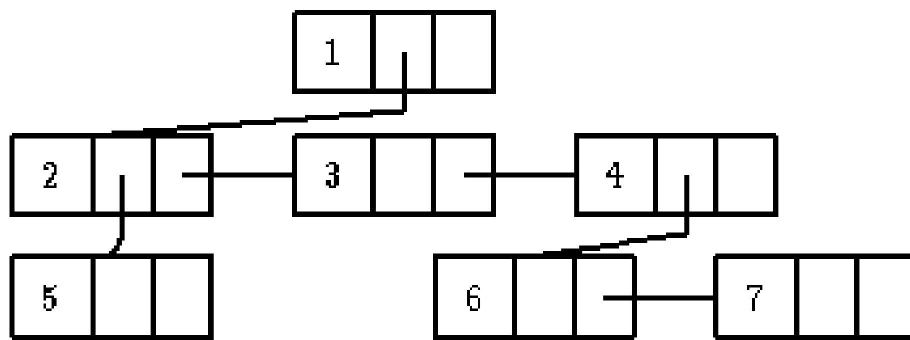
Usages for visit : determine the height, count the number of elements

5.3 Representations

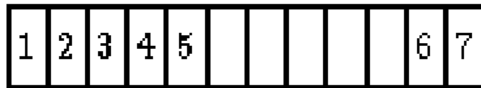
1. Nodes consisting of a data field and k pointers



2. Nodes consisting of w data field and two pointers: a pointer to the first child, and a pointer to the next sibling.



3. A tree of degree k assumes an array for holding a complete tree of degree k, with empty cells assigned for missing elements.



5.4 Demo Applets

binary search tree

5.5 Assignment #2 (due We, Oct 6)

1. Assume binary trees in which the leaf nodes hold integer numbers and the non-leaf nodes hold the binary operations `+`, `-`, `*`, and `/`.
   a. Provide an algorithm that, when given the root of a tree, evaluates the expression represented by the tree.
   b. Provide an algorithm that, when given the root of a tree, outputs a program which evaluates the expression represented by the tree. The program should consist just of instructions of the form `var = integer`, `var = var + var`, `var = var - var`, `var = var * var`, and `var = var / var`, where each `var` stands for a variable name.
2. Assume binary trees in which each node carries a name. Provide an algorithm that, when given the root of a tree, writes the names of the nodes of the tree level-wise from bottom up, and in each level from right to left. Your algorithm should be of time complexity O(n), where n denotes the number of nodes in the trees.

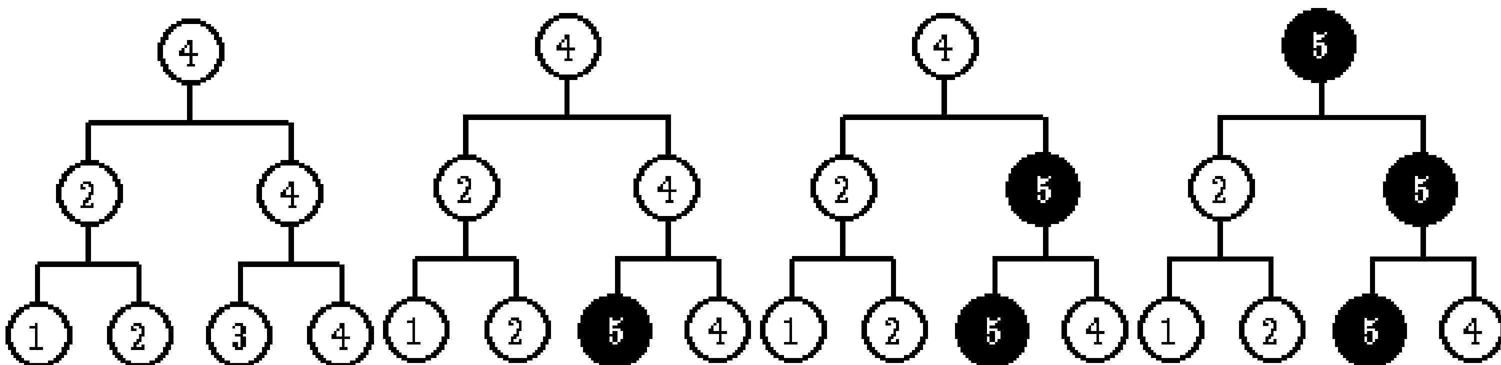
[next] [front] [up]

Chapter 6
Selection Trees

A **selection tree** is a complete binary tree in which the leaf nodes hold a set of keys, and each internal node holds the winner key among its children.

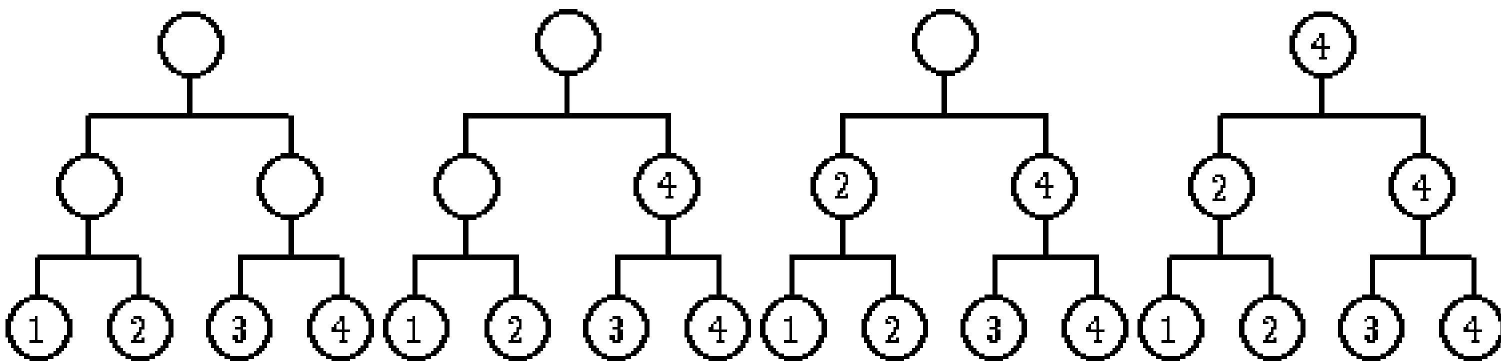
6.1 Modifying a Key

It takes O(log n) time to modify a selection tree in response to a change of a key in a leaf.



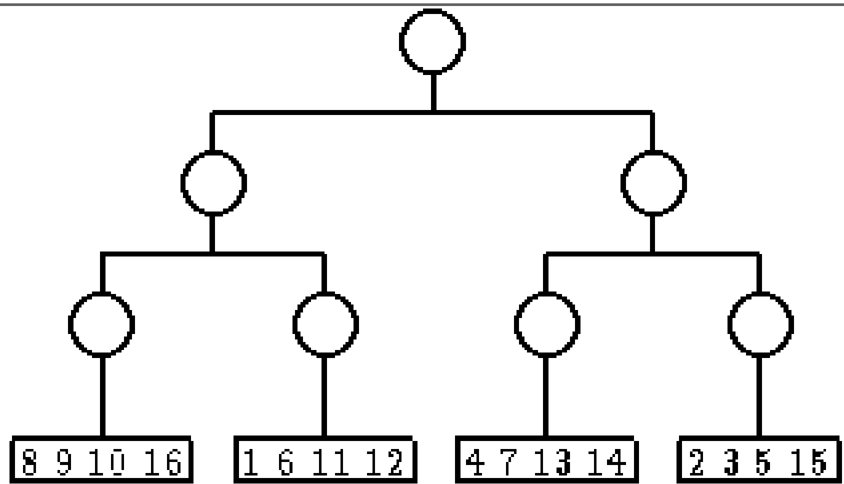
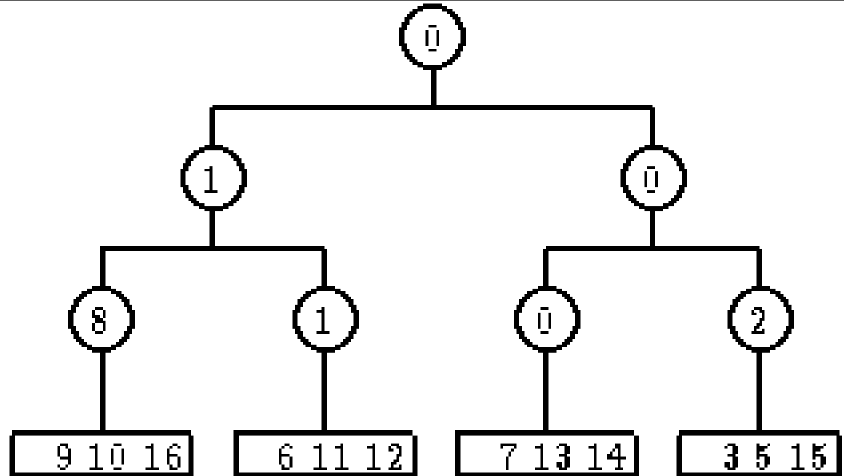
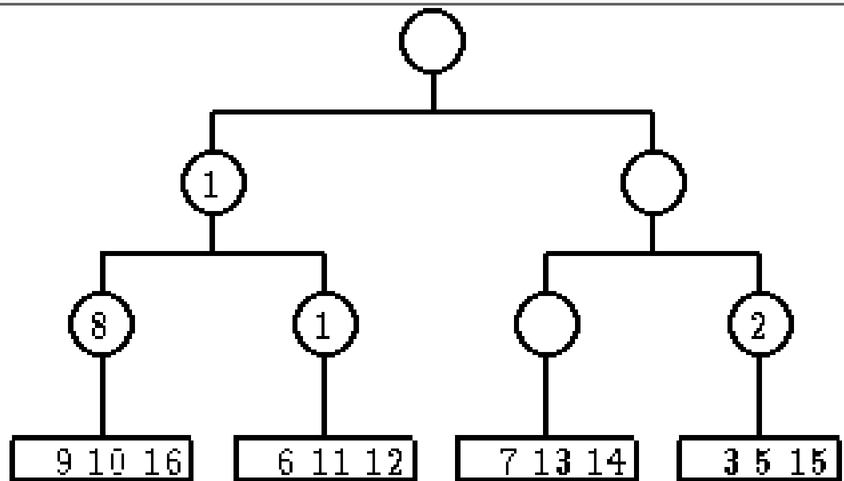
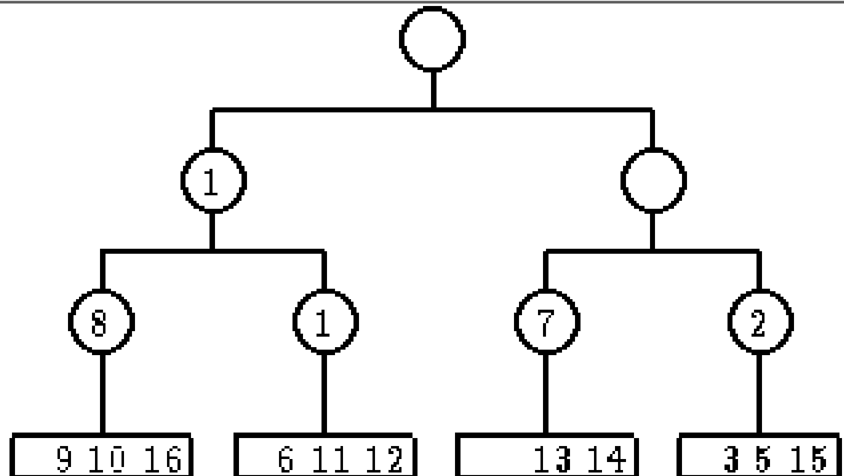
6.2 Initialization

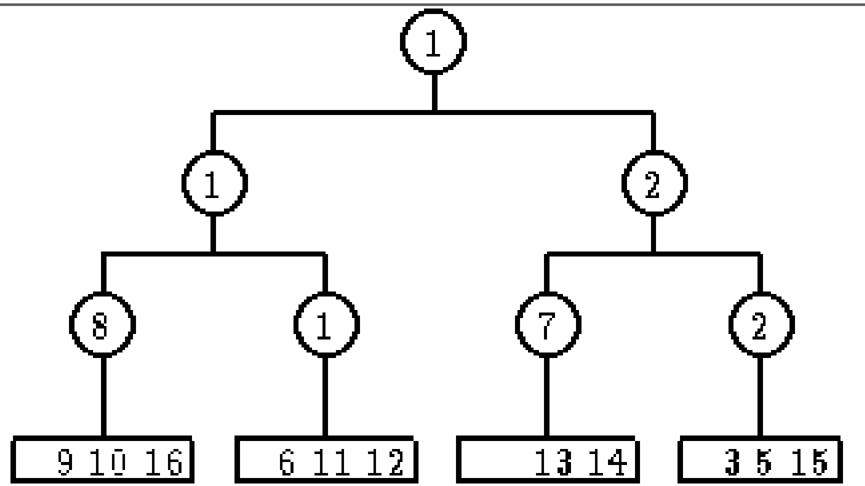
The construction of a selection tree from scratch takes O(n) time by traversing it level-wise from bottom up.



6.3 Application: External Sort

Given a set of n values	16 9 10 8 6 11 12 1 4 7 14 13 2 15 5 3				n = 16
divide it into M chuncks,	16 9 10 8	6 11 12 1	4 7 14 13	2 15 5 3	M = 4
internally sort each chunk,	8 9 10 16	1 6 11 12	4 7 13 14	2 3 5 15	

construct complete binary tree of M leaves with the chunks attached to the leaves.	
Convert the tree into a selection tree with the keys being fed to the leaves from the chunks	
Remove the winner from the tree	
Feed to the empty leaf the next value from its corresponding chunk	

Adjust the selection tree to the change in the leaf	
Repeat the deletion subprocess until all the values are consumed.	

- The algorithm takes $O(M \frac{n}{M} \log \frac{n}{M})$ time to internally sort the elements of the chunks, O(M) to initialize the selection tree, and O(n log M) to perform the selection sort. For M « n the total time complexity is O(n log n).
- To reduce I/O operations, inputs from the chunks to the selection tree should go through buffers.

[prev] [prev-tail] [front] [up]

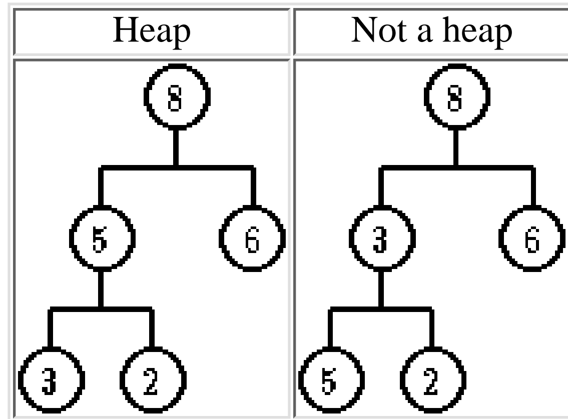
[next] [tail] [up]

Chapter 7
Heaps

A **heap** is a complete tree with an ordering-relation R holding between each node and its descendant.

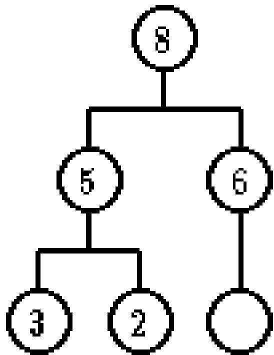
Examples for R: smaller-than, bigger-than

Assumption: In what follows, R is the relation bigger-than , and the trees have degree 2.



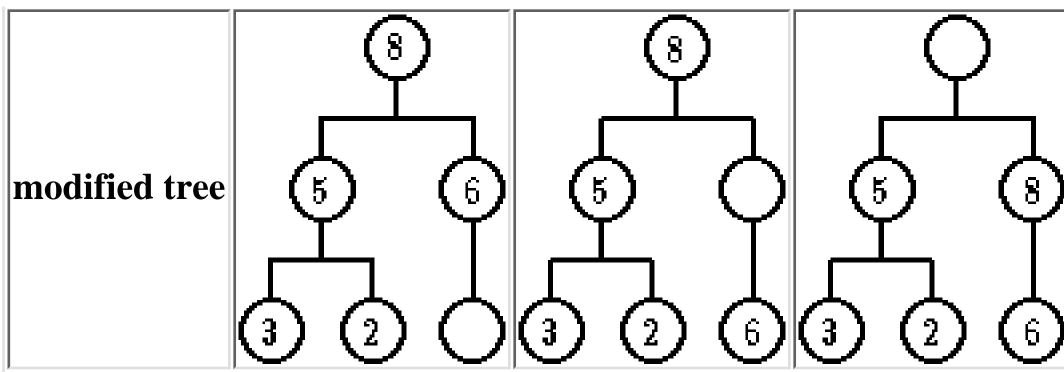
7.1 Adding an Element

1. Add a node to the tree

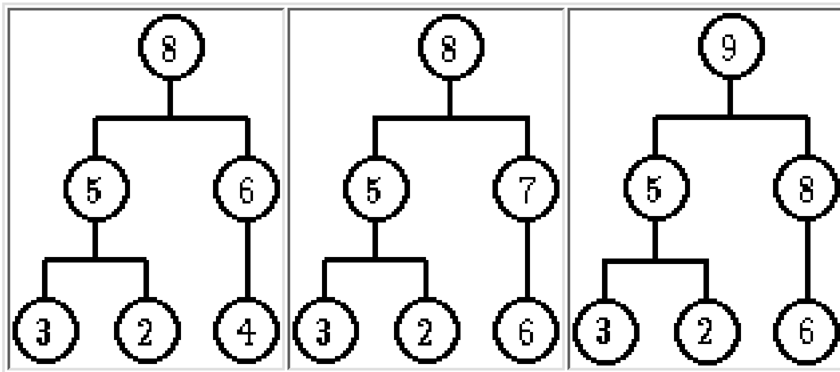


2. Move the elements in the path from the root to the new node one position down, if they are smaller than the new element

new element	4	7	9



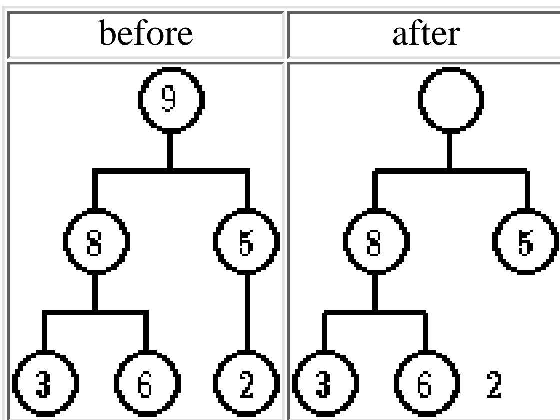
3. Insert the new element to the vacant node



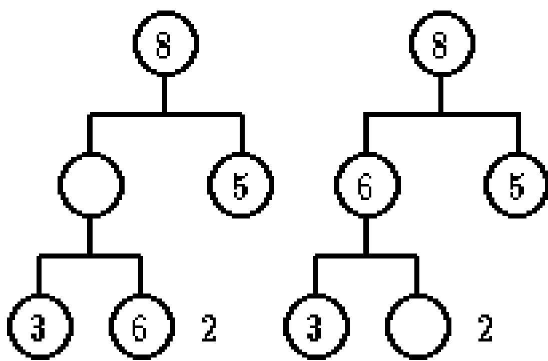
4. A complete tree of n nodes has depth $\lceil \log n \rceil$, hence the time complexity is O(log n)

7.2 Deleting an Element

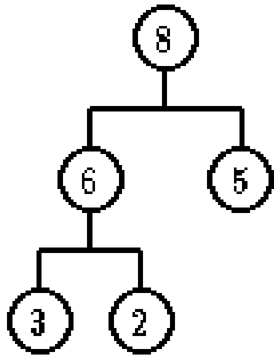
1. Delete the value from the root node, and delete the last node while saving its value.



2. As long as the saved value is smaller than a child of the vacant node, move up into the vacant node the largest value of the children.



3. Insert the saved value into the vacant node



4. The time complexity is O(log n)

7.3 Initialization

Brute Force

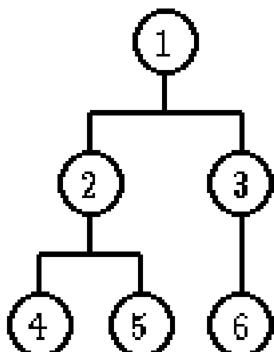
Given a sequence of n values e_1, ..., e_n, repeatedly use the insertion module on the n given values.

- Level h in a complete tree has at most $2^{h-1} = O(2^n)$ elements
- Levels 1, ..., h - 1 have $2^0 + 2^1 + ... + 2^{h-2} = O(2^h)$ elements
- Each element requires O(log n) time. Hence, brute force initialization requires O(n log n) time.

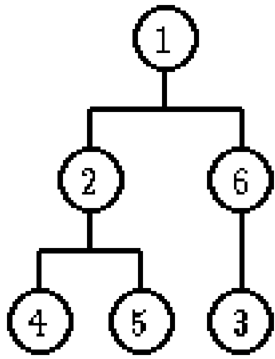
  elements.

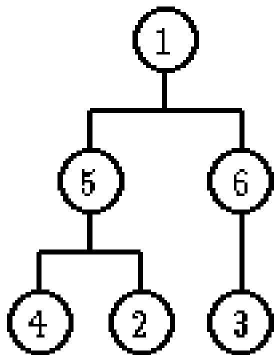
Efficient

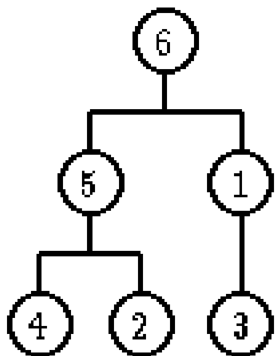
- Insert the n elements e_1, ..., e_n into a complete tree

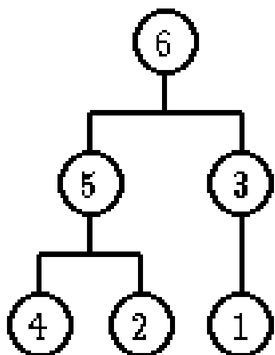


- For each node, starting from the last one and ending at the root, reorganize into a heap the subtree whose root node is given. The reorganization is performed by interchanging the new element with the child of greater value, until the new element is greater than its children.









- The time complexity is $O(0 * (n/2) + 1 * (n/4) + 2 * (n/8) + \ldots + (\log n) * 1) = O(n(0 . 2^{-1} + 1 . 2^{-2} + 2 . 2^{-3} + \ldots + (\log n) . 2^{-\log n})) = O(n)$

  since the following equalities holds. $\sum_{k=1}^{\infty}(k - 1)2^{-k} = 2[\sum_{k=1}^{\infty}(k - 1)2^{-k}] - [\sum_{k=1}^{\infty}(k - 1)2^{-k}] = [\sum_{k=1}^{\infty}k2^{-k}] - [\sum_{k=1}^{\infty}(k - 1)2^{-k}] = \sum_{k=1}^{\infty}[k - (k - 1)]2^{-k} = \sum_{k=1}^{\infty}2^{-k} = 1$

7.4 Applications

Priority Queue A dynamic set in which elements are deleted according to a given ordering-relation.

Heap Sort Build a heap from the given set (O(n)) time, then repeatedly remove the elements from the heap (O(n log n)).

7.5 Implementation

An array. The root is at location 1. Location $i > 1$ is a child of $\lfloor i/2 \rfloor$.

7.6 Demo Applets

- demo of heap

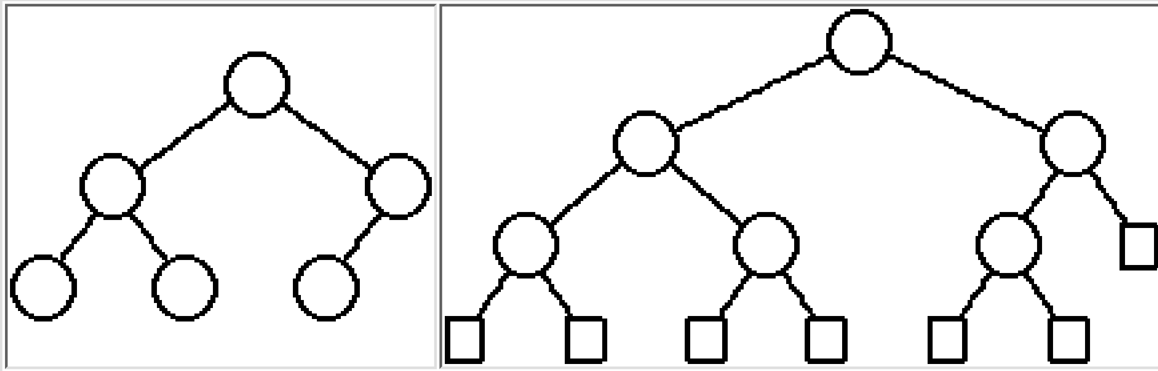
[next] [front] [up]

[prev] [prev-tail] [tail] [up]

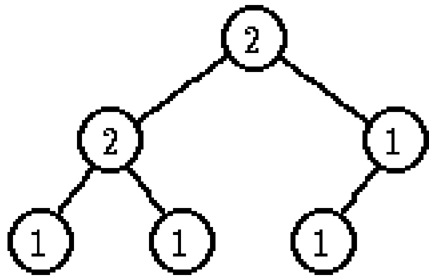
Chapter 8
Height-Biased Leftist Trees

8.1 Definitions

An **external node** is an imaginary node in a location of a missing child.

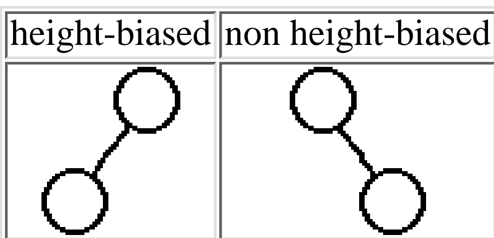


Notation. Let the **s-value** of a node be the shortest distance from the node to an external node.



- An external node has the s-value of 0.
- An internal node has the s-value of 1 plus the minimum of the s-values of its internal and external children.

In a **height-biased leftist tree** the s-value of a left child of a node is not smaller than the s-value of the right child of the node.



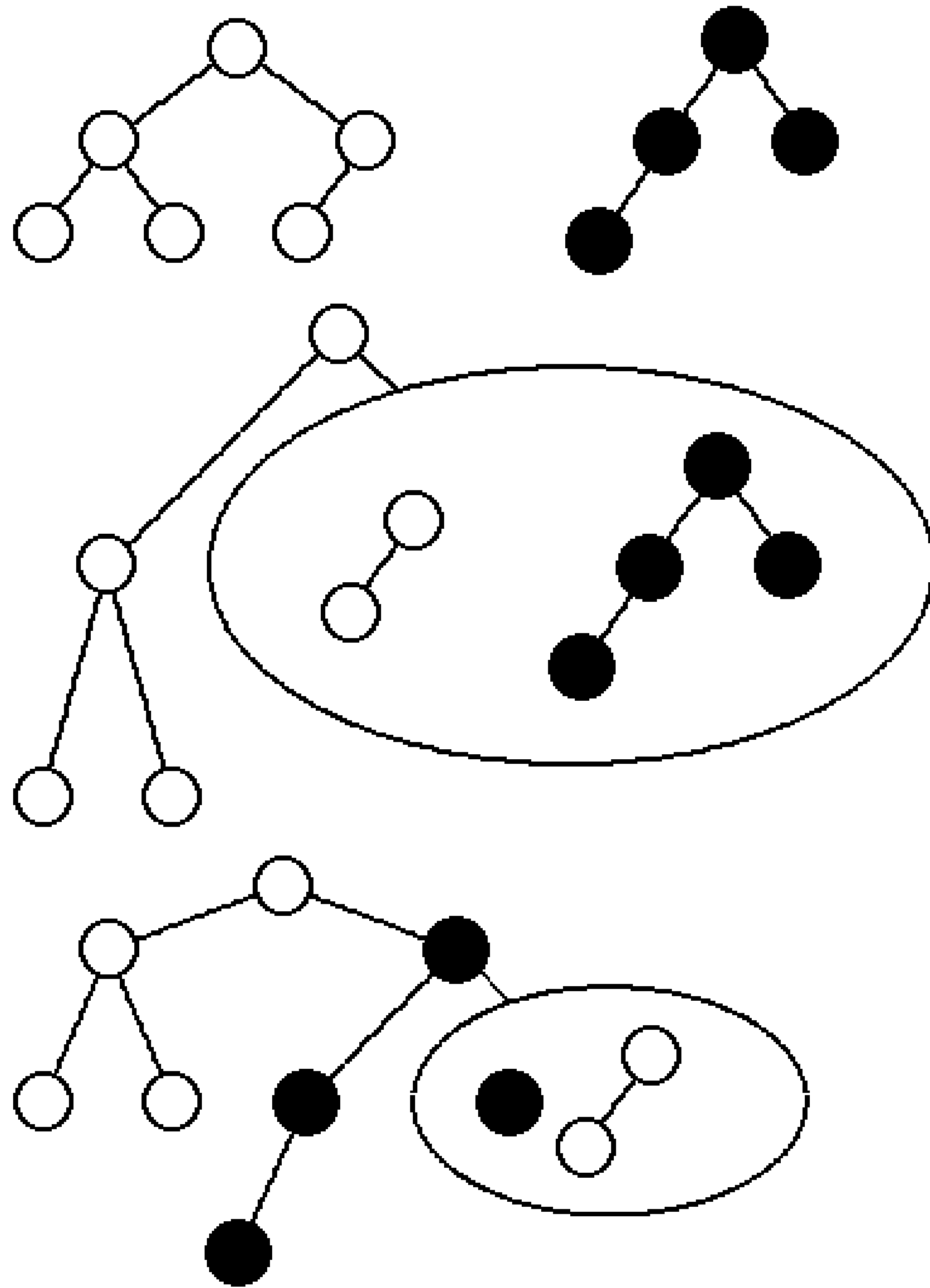
8.2 Merging Height-Biased Leftist Trees

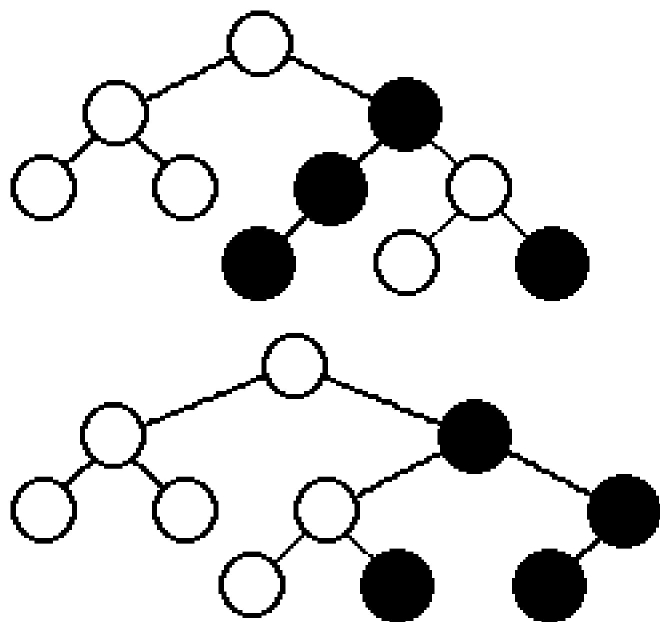
Recursive algorithm

- Consider two nonempty height-biased leftist trees A and B, and a relation (e.g., smaller than) on the values of the keys.

- Assume the key-value of A is not bigger than the key-value of B
- Let the root of the merged tree have the same left subtree as the root of A
- Let the root of the merged tree have the right subtree obtained by merging B with the right subtree of A.
- If in the merged tree the s-value of the left subtree is smaller than the s-value of the right subtree, interchange the subtrees.

For the following example, assume the key-value of each node equals its s-value.





Time complexity

- Linear in the rightmost path of the outcome tree.
- The rightmost path of the the outcome tree is a shortest path
- A shortest path can t contain more than log n nodes.

  Proof If the shortest path can contain more than log n nodes, then the first 1 + log n levels should include $2^0 + 2^1 + ... + 2^{\log n} = 2^{1+\log n} - 1 = 2n - 1$ nodes. In such a case, for n > 1 we end up with n > 2n - 1.

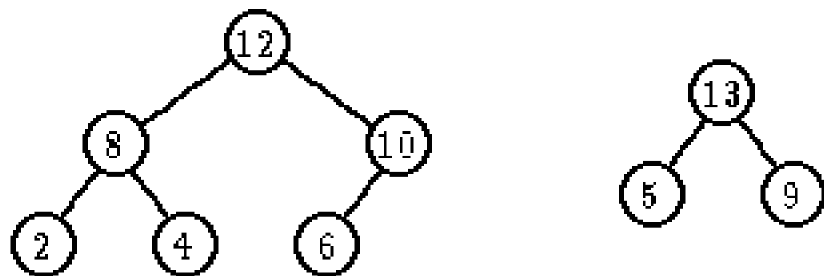
8.3 Application: Priority Queues

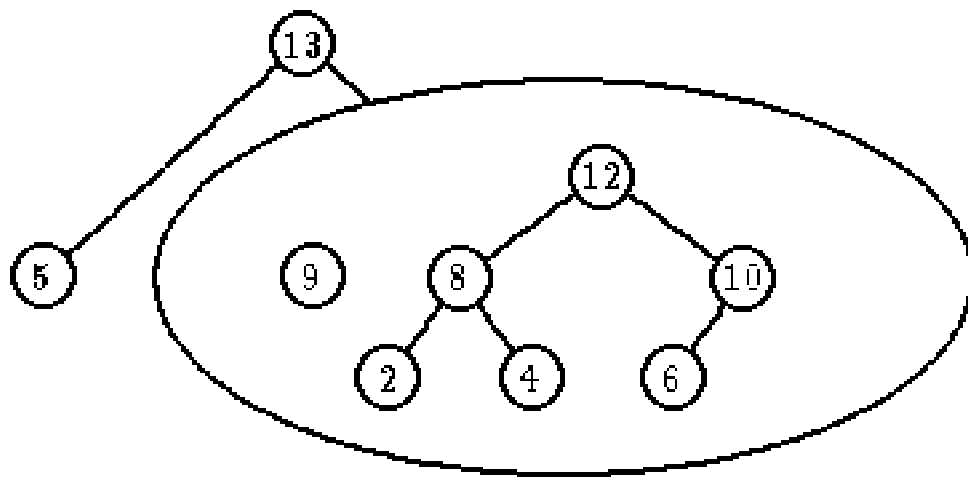
A data type which elements may be added in arbitrary order, but can be removed in order that fits a priority function.

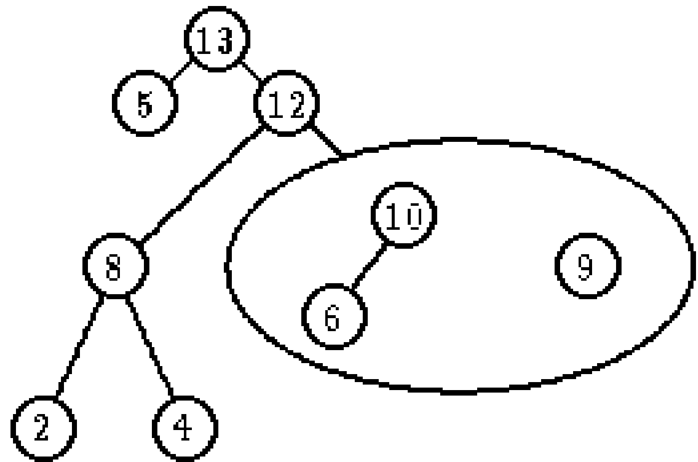
Assumption for the examples of this section: priority goes for the largest value.

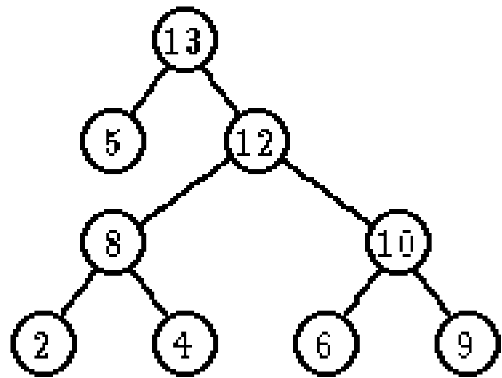
Merge

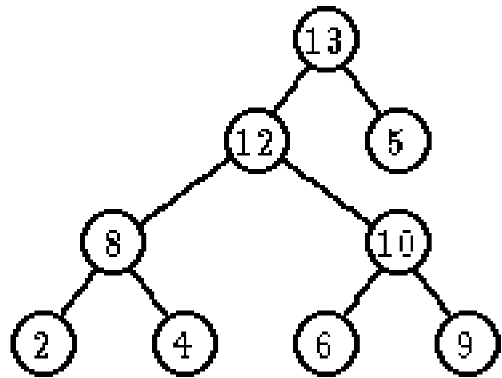
Use general keys, instead of the s-values, for determining the roots.











Time Complexity O(log n)

Add

The added element is treated as a height-biased leftist tree to be merged with the tree representing the queue.

Delete

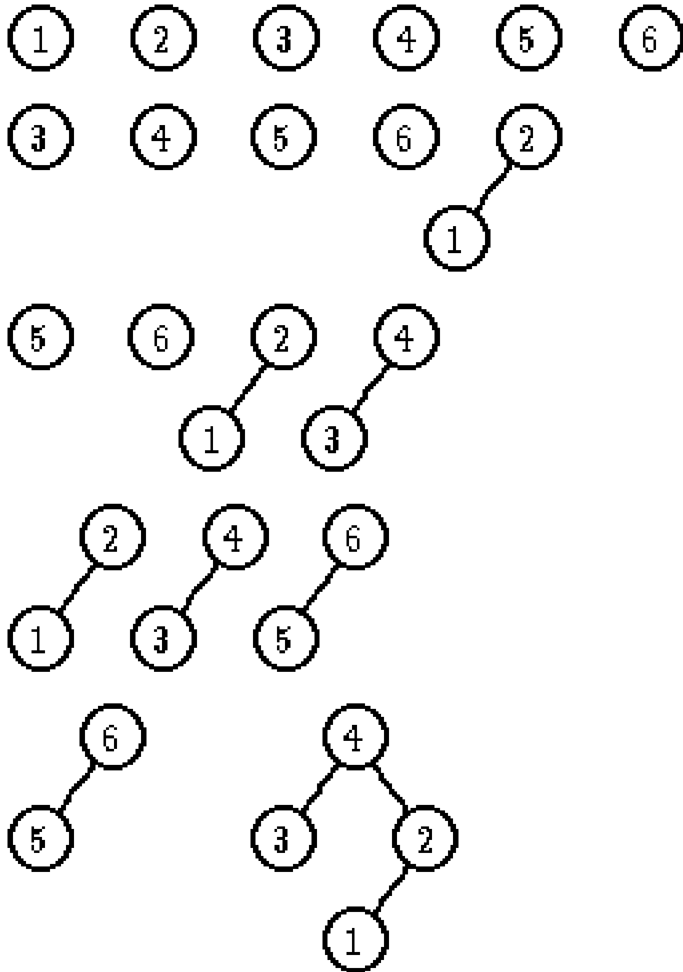
Delete the root then merge the two subtrees.

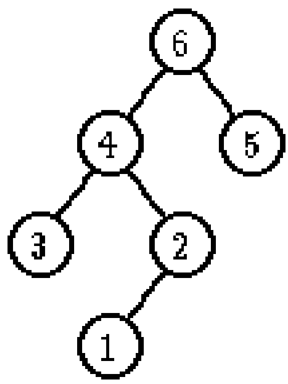
Initialization

Brute Force: Use the add operation repeatedly. Time complexity O(n log n)

Specialized Approach:

1. Insert the given elements into a queue, taking each of them to be a height-biased leftist tree.
2. As long as the queue contains more than one tree, remove the next pair of trees from the queue, merge these tree, and insert the outcome to the queue.





Time complexity: (n/2) . O(1) + (n/4) . O(2) + (n/8) . O(3) + . . . = O(n)

8.4 Weight-Biased Leftist Trees

A variant of the height-biased leftist tree, in which the s-value of a node is the number of nodes in the subtree defined by the node.

8.5 Assignment #3 (due We, Oct 13)

1. Let a min-max heap be a complete binary tree, consisting of min and max nodes satisfying the following conditions.
   a. The root is a min node.
   b. Min nodes can have only max nodes for children, and max nodes can have only min nodes for children.
   c. The descendents of a min node may not hold values smaller than the one associated with the min node.
   d. The descendents of a max node may not hold values greater than the one associated with the max node.

   You have to perform the following activities in the given order.
   i. Draw a min-max tree consisting of 30 nodes
   ii. Add to your tree a new key, smaller than the one in the root. Show the outcome tree, as well as all the intermediate trees.

      Assume a variant of the algorithm for adding values to standard heaps in which, except for the initial step, comparisons and swapping are made with grandparents instead of parents.
   iii. Add to your tree a second value, bigger than those of the children of the root. Use the algorithm of the previous item, and also here draw the intermediate and final trees.
   iv. Delete the min value from the (root of the) tree. Show the outcome tree, as well as all the intermediate trees.

      Assume a variant of the algorithm for deleting values from standard heaps, in which comparisons and swapping are done between nodes and their grandchildren. Local corrections of comparisons and swaps between the grandchildren and their parents, might also be needed.

v. Delete the max value from the (largest child of the root of the) tree. Show all the intermediate and final trees.

2. Show the following weight-biased leftist trees, and all the intermediate trees used to create them.
    a. The initialization tree for the set of values {2, 4, 6, 8, 10, 12, 14}.
    b. The modified tree after adding the value of 17 to the tree in (a).
    c. The modified tree after adding the value 15 to the tree in (b).
    d. The modified tree after deleting the largest value from the tree in (c).
    e. The modified tree after deleting the largest value from the tree in (d).

[prev] [prev-tail] [front] [up]

Chapter 9
Binary Search Trees

9.1 Characteristics

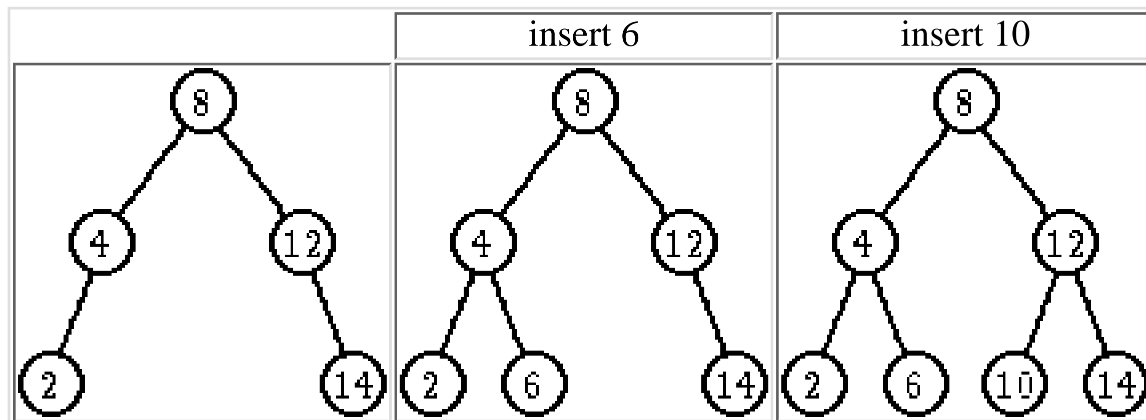
Trees in which the key of an internal node is greater than the keys in its left subtree and is smaller than the keys in its right subtree.

9.2 Search

```
search ( tree,key )
  IF empty tree             THEN return not-found
  IF key == value in root   THEN return found
  IF key  > value in root   THEN search (left-subtree,  key)
                                 search (right-subtree, key)
```

Time: O(depth of tree)

9.3 Insertion

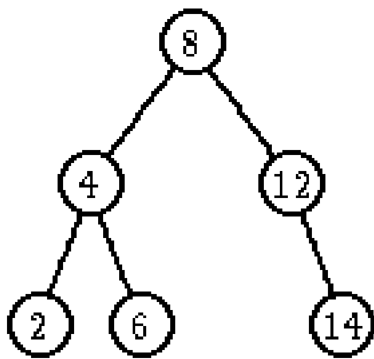


9.4 Deletion

The way the deletion is made depends on the type of node holding the key.

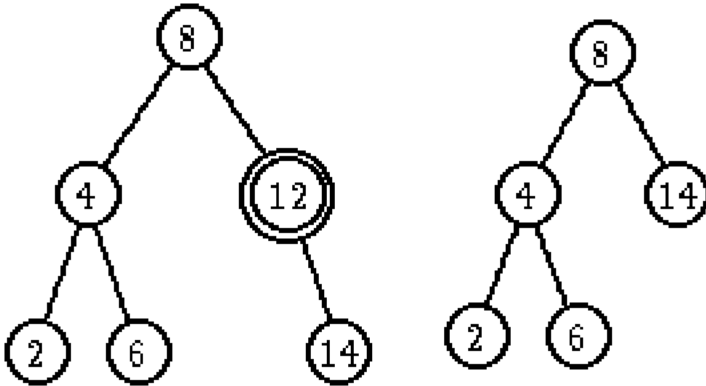
Node of degree 0

Delete the node



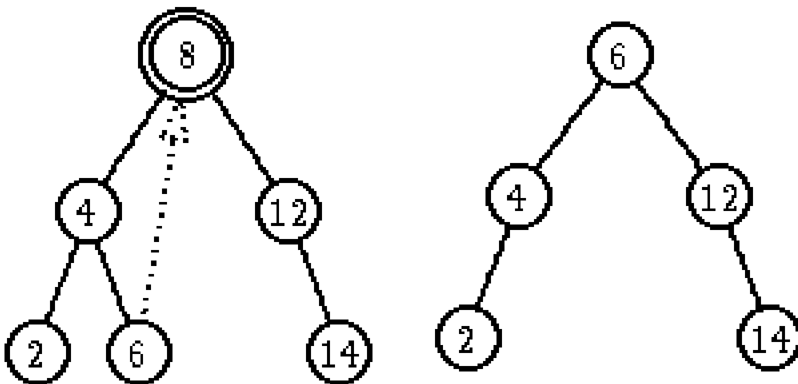
Node of degree 1

Delete the node, while connecting its predecessor to the successor.



Node of degree 2

Replace the node containing the deleted key with the node having the largest key in the left subtree, or with the node having the smallest key in the right subtree.



9.5 Demo Applets

- demo applet binary search trees

[next] [front] [up]

a. Time: $O(1 + 2 + ... + n) = O(n^2)$. Space: $O(1)$

$$
\begin{aligned}
T(n) &= 3T(n-2) \\
&= 3^2T(n-2-2) \\
&= 3^3T(n-2-2-2) \\
&= 3^4T(n-2-2-2-2) \\
&= \dots \\
&= 3^kT(n-k*2) \\
\end{aligned}
$$

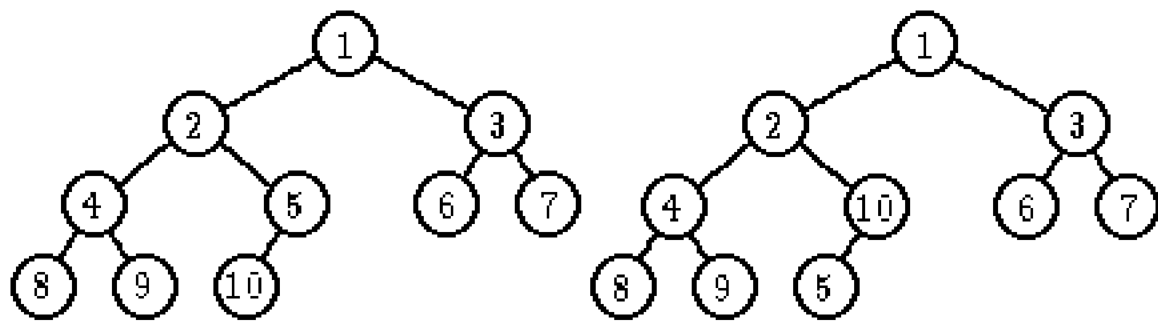
b.

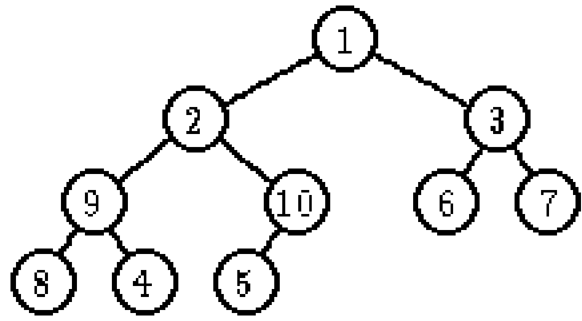
$$
= 3^{\lfloor n/2 \rfloor}
$$

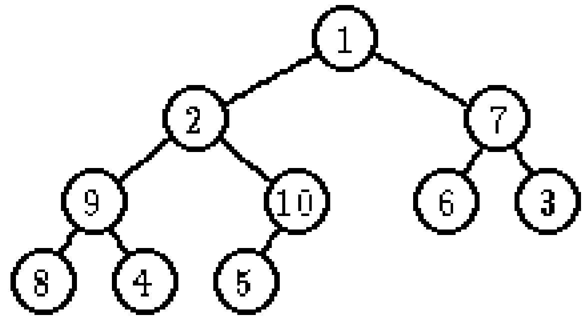
a.
```
FUNCTION leaves( T )
   IF T is empty  THEN return 0
   IF T is a leaf THEN return 1
   IF T has 2 children THEN
      return [ leaves( left(T) ) + leaves( right(T) )]
   IF T has left child THEN return leaves( left(T) )
   return leaves( right(T) )
END
```

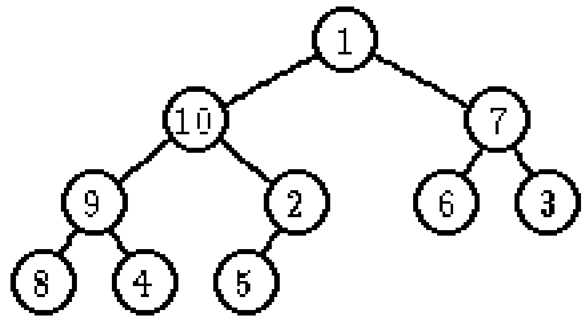
b. O(n)

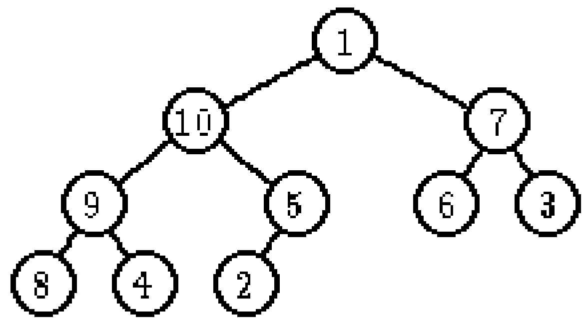
(a) 8. (b) 4. (c) 4. (d) 7. (e) 6.

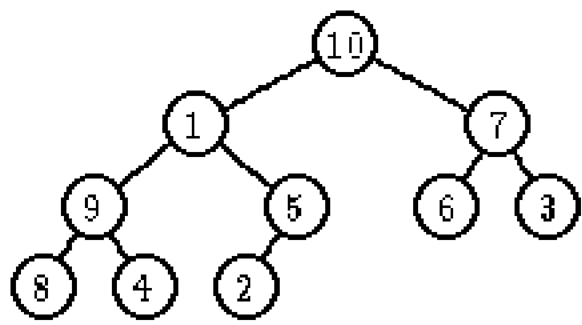

1
2 3
4 5 6 7
8 9 10
1
2 3
4 10 6 7
8 9 5

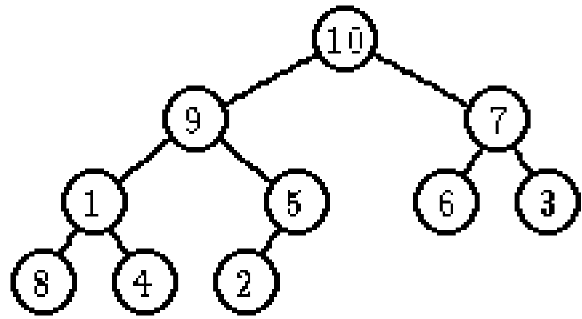

1
2 3
9 10 6 7
8 4 5

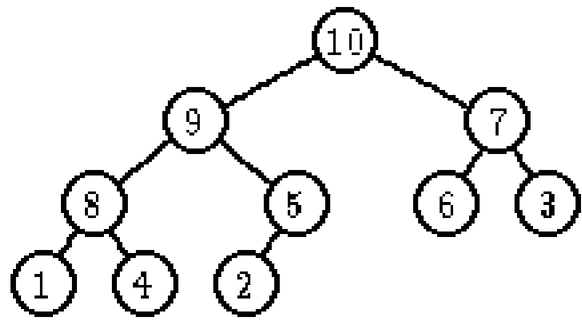

1
2 7
9 10 6 3
8 4 5


1
10 7
9 2 6 3
8 4 5

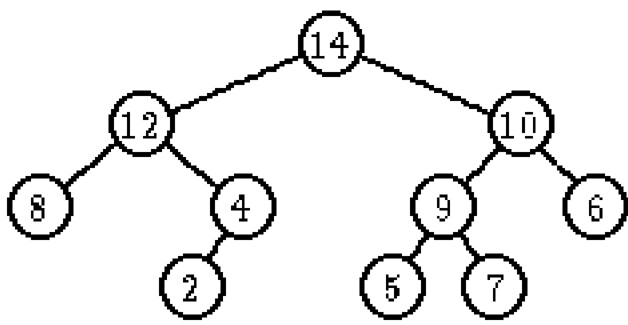

1
10 7
9 5 6 3
8 4 2


10
1
7
9
5
6
3
8
4
2

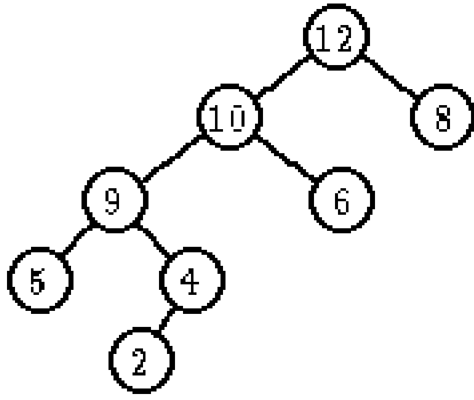

10
9
7
1
5
6
3
8
4
2

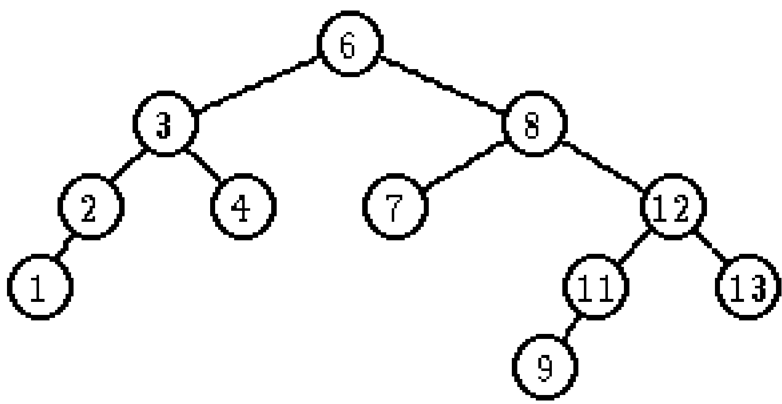

10
9
7
8
5
6
3
1
4
2

a.



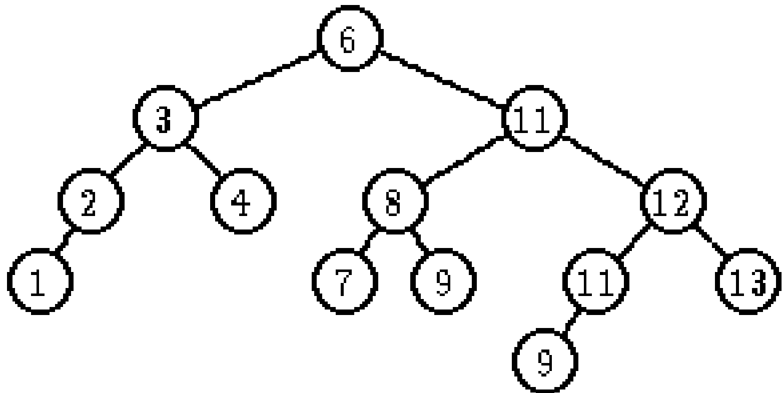
b.



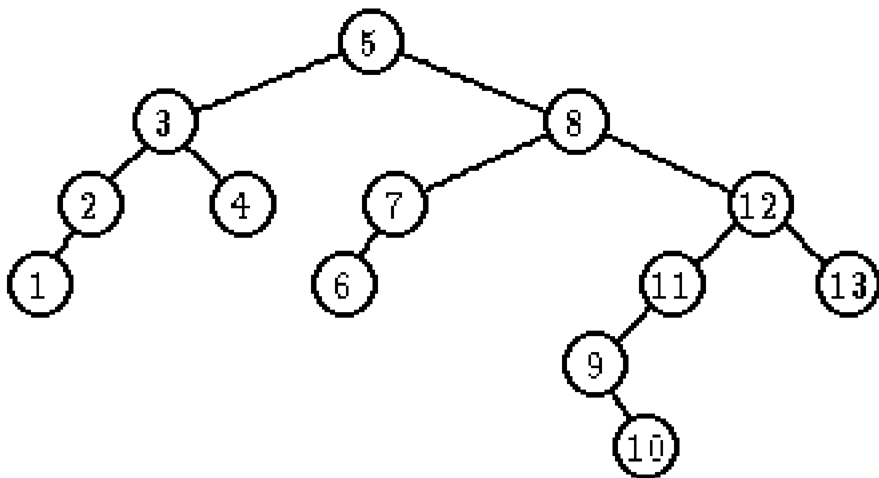


a.

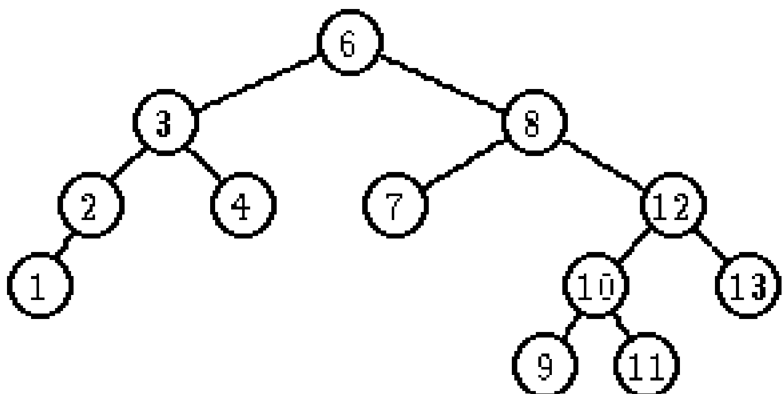
RL transformation at 8:



b.



LR transformation at 11:



a. LLr
b. RLr
c. Lb2
d. Lb0
e. Lr0

a. $T(n) = O(n^6)$, $S(n) = O(n^6)$,
b. $T(n) = O(\log n)$, $S(n) = O(\log n)$,

a.

$$
\begin{aligned}
T(n) &= 2T(n-3) \\
&= 2^2T(n-3-3) \\
&= 2^3T(n-3-3-3) \\
&= 2^4T(n-3-3-3-3) \\
&= \ldots \\
&= 2^kT(n-k*3) \\
&= 2^{n/3}
\end{aligned}
$$

b. bigger

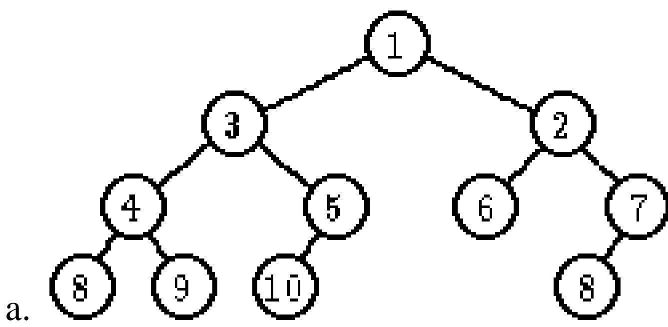
a. True
b. True
c. True
d. False
e. False

a.
```
FUNCTION height( T )
  IF T is empty  THEN return 0
  IF T is a leaf THEN return 1
  RETURN  1 + max( height( left(T) ), leaves( right(T) ) )
END
```
b. O(n)

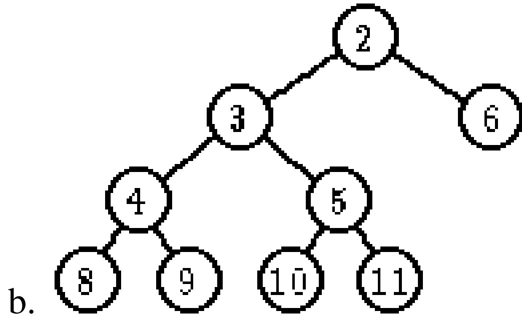
c. T(n)=O(n), S(n)=O(log n)

d. O(n)

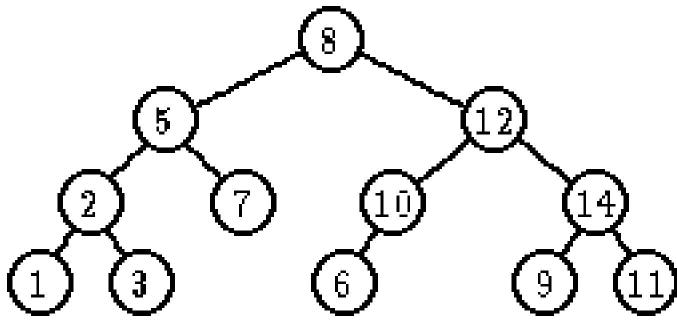
a.



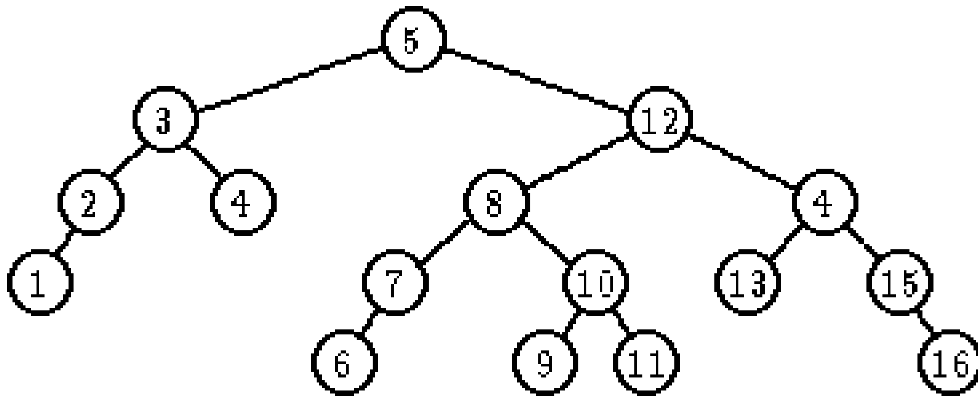
b.



a. Deletion of 4



b. Insertion of 16

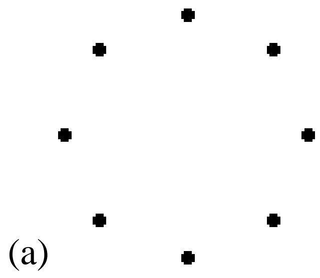


a. LLr
b. RLr
c. Lb2 (18 goes up) or none (22 goes up)
d. none
e. Wrong problem. Should be: Deletion of the key 4.

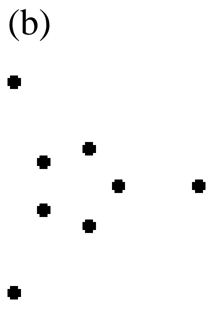
$O(|V|^2)$

(a) 7 (b) 16

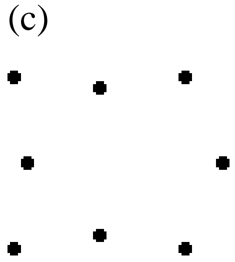
(a)



(b)



(c)



a. Permutation tree of job numbers, where the i th job in a permutation is the assignment provided to worker j
b. Full binary tree, where the i th level determines at what side of the partition the ith node should belong.

(a) $C(n, k) = \begin{cases} C(n-1, k) + C(n-1, k-1) & \text{if } n > k > 1 \\ 1 & \text{if } k = n > 0 \\ n & \text{if } n > k = 1 \end{cases}$

```
FOR i = 1 TO n DO
  FOR j = 1 TO k DO
    IF i > j > 1 THEN C[i,j] = C(i-1,j)+C(i-1,j-1)
    IF i = j > 0 THEN C[i,j] = 1
    IF i > j = 1 THEN C[i,j] = i
```

(b) None (for the boundary conditions in (a)):