Design and Analysis of Algorithms

Module 3: Greedy Method

Module 3: Greedy Method - Outline

- Introduction to Greedy method
  - General method,
  - Coin Change Problem
  - Knapsack Problem
  - Job sequencing with deadlines
- Minimum cost spanning trees:
  - Prim's Algorithm,
  - Kruskal's Algorithm
- Single source shortest paths
  - Dijkstra's Algorithm
- Optimal Tree problem:
  - Huffman Trees and Codes
- Transform and Conquer Approach:
  - Heaps
  - Heap Sort

Module 3: Greedy Method - Outline

- Introduction to Greedy method
  - **General method**
  - Coin Change Problem
  - Knapsack Problem
  - Job sequencing with deadlines
- Minimum cost spanning trees:
  - Prim's Algorithm,
  - Kruskal's Algorithm
- Single source shortest paths
  - Dijkstra's Algorithm
- Optimal Tree problem:
  - Huffman Trees and Codes
- Transform and Conquer Approach:
  - Heaps
  - Heap Sort

Greedy – General Method

- The greedy approach suggests
  - constructing a solution through a sequence of steps
  - each expanding a partially constructed solution obtained so far,
  - until a complete solution to the problem is reached.

- On each step the choice made must be:
  - ***feasible***, i.e., it has to satisfy the problem's constraints
  - ***locally optimal***, i.e., it has to be the best local choice among all feasible choices available on that step
  - ***irrevocable***, i.e., once made, it cannot be changed on subsequent steps of the algorithm

General method

- Given n inputs choose a subset that satisfies some constraints.
- A subset that satisfies the constraints is called a feasible solution.
- A feasible solution that maximises or minimises a given (objective) function is said to be optimal.
- Often it is easy to find a feasible solution but difficult to find the optimal solution.

Greedy Method

```
Algorithm Greedy(a, n)
// a[1 : n] contains the n inputs.
{
    solution := ∅; // Initialize the solution.
    for i := 1 to n do
    {
        x := Select(a);
        if Feasible(solution, x) then
            solution := Union(solution, x);
    }
    return solution;
}
```

Module 3: Greedy Method - Outline

- Introduction to Greedy method
  - General method,
  - **Coin Change Problem**
  - Knapsack Problem
  - Job sequencing with deadlines
- Minimum cost spanning trees:
  - Prim's Algorithm,
  - Kruskal's Algorithm
- Single source shortest paths
  - Dijkstra's Algorithm
- Optimal Tree problem:
  - Huffman Trees and Codes
- Transform and Conquer Approach:
  - Heaps
  - Heap Sort

Coin Change Problem

Problem Statement:

Given coins of several denominations find out a way to give a customer an amount with **fewest** number of coins.

Coin Change Problem

Example: if denominations are 1, 5, 10, 25 and 100 and the change required is 30, the solutions are,

Amount : 30

Solutions :

3 x 10 (3 coins)
6 x 5 (6 coins)
1 x 25 + 5 x 1 (6 coins)
1 x 25 + 1 x 5 (2 coins)

The last solution is the **optimal** one as it gives us change only with 2 coins.

Coin Change Problem

- Solution for coin change problem using greedy algorithm is very intuitive and called as cashier's algorithm.
- Basic principle is:
  - At every iteration for search of a coin, take the largest coin which can fit into remain amount to be changed at that particular time.
  - At the end you will have optimal solution.

Module 3: Greedy Method - Outline

- Introduction to Greedy method
  - General method,
  - Coin Change Problem
  - **Knapsack Problem**
  - Job sequencing with deadlines
- Minimum cost spanning trees:
  - Prim's Algorithm,
  - Kruskal's Algorithm
- Single source shortest paths
  - Dijkstra's Algorithm
- Optimal Tree problem:
  - Huffman Trees and Codes
- Transform and Conquer Approach:
  - Heaps
  - Heap Sort

Example: Your Train breaks down in a desert and you decide to walk to nearest town. You have a rucksack but which objects should you take with you ?

Feasible: Any set of objects is a feasible solution provided that they are not too heavy, fit in the rucksack and will help you survive (these are constraints).

An optimal solution is the one that maximises or minimises something

– One that minimises the weight carried

– One that fills the rucksack completely (maximise)

– One that ensures the most water is taken etc.

Knapsack Problem (Fractional Knapsack Problem)

- Problem description:
  - Pack a knapsack with a capacity of m
  - From a list of n items, we must select the items that are to be packed in the knapsack.
  - Each object i has a weight of w_i and a profit of p_i .

- In a feasible knapsack packing, the sum of the weights packed does not exceed the knapsack capacity.

Knapsack Problem

$$\text{maximize} \sum_{1 \le i \le n} p_i x_i$$

$$\text{subject to} \sum_{1 \le i \le n} w_i x_i \le m$$

$$\text{and } 0 \le x_i \le 1, \quad 1 \le i \le n$$

The profits and weights are positive numbers.

Knapsack Problem

Consider the following instance of the knapsack problem:

n=3, m=20, (p_1, p_2, p_3) =(25, 24, 15), (w_1, w_2, w_3) =(18, 15, 10)

- There are several greedy methods to obtain the feasible solutions. Three are discussed here
  - At each step fill with the object with largest profit
  - At each step fill with the object with smallest weight
  - At each step fill with the object with largest profit/weight ratio

Knapsack Problem

a) At each step fill the knapsack with the object with **largest profit**

n=3, m=20, (p_1, p_2, p_3) =(25, 24, 15), (w_1, w_2, w_3) =(18, 15, 10)

Knapsack Problem

b) At each step fill the object with **smallest weight**

n=3, m=20, (p_1, p_2, p_3) =(25, 24, 15), (w_1, w_2, w_3) =(18, 15, 10)

Knapsack Problem

C) At each step include the object with **maximum profit/weight ratio**

n=3, m=20, $(p_1, p_2, p_3) = (25, 24, 15)$, $(w_1, w_2, w_3) = (18, 15, 10)$

```
void GreedyKnapsack(float m, int n)
// p[1:n] and w[1:n] contain the profits and weights
// respectively of the n objects ordered such that
// p[i]/w[i] >= p[i+1]/w[i+1]. m is the knapsack
// size and x[1:n] is the solution vector.
{
    for (int i=1; i<=n; i++) x[i] = 0.0; // Initialize x.
    float U = m;
    for (i=1; i<=n; i++) {
       if (w[i] > U) break;
       x[i] = 1.0;
       U -= w[i];
    }
    if (i <= n) x[i] = U/w[i];
}
```

Analysis

- Disregarding the time to initially sort the object, each of the above strategies use O(n) time

0/1 Knapsack problem

[0/1 Knapsack] Consider the knapsack problem discussed in this section. We add the requirement that $x_i = 1$ or $x_i = 0$, $1 \leq i \leq n$; that is, an object is either included or not included into the knapsack. We wish to solve the problem

$$\max \sum_{1}^{n} p_i x_i \qquad \text{subject to} \sum_{1}^{n} w_i x_i \leq m \qquad \text{and } x_i = 0 \text{ or } 1,\ 1 \leq i \leq n$$

One greedy strategy is to consider the objects in order of nonincreasing density p_i/w_i and add the object into the knapsack if it fits.

Note: The greedy approach to solve 0/1 knapsack problem does not necessarily yield an optimal solution

Program 6a:
Knapsack 0/1 using Dynamic Programming

```
for (i = 0; i <= n; i++) {
        for (w = 0; w <= C; w++) {
                if (i == 0 || w == 0) {
                        K[i][w] = 0;
                } else if (wt[i] <= w) {
                        K[i][w] = max(p[i] + K[i - 1][w - wt[i]], K[i - 1][w]);
                } else {
                        K[i][w] = K[i - 1][w];
                }
        }
}
```

Total items: 4
Capacity of Knapsack: 5
Items profit: 12 10 20 15
Items weights: 2 1 3 2

Capacity→	0	1	2	3	4	5
Item0 (0, 0):	0 (1)	0 (1)	0 (1)	0 (1)	0 (1)	0 (1)
Item1 (12, 2):	0 (1)	0 (4)	12 (2)	12 (2)	12(2)	12 (2)
Item2 (10, 1):	0 (1)	10 (2)	12 (3)	22 (2)	22 (2)	22 (2)
Item3 (20, 3):	0 (1)	10 (4)	12 (4)	22 (3)	30 (2)	32 (2)
Item4 (15, 2):	0 (1)	10 (4)	15 (2)	25 (2)	30 (3)	37 (2)

Module 3: Greedy Method - Outline

- Introduction to Greedy method
  - General method,
  - Coin Change Problem
  - Knapsack Problem
  - **Job sequencing with deadlines**
- Minimum cost spanning trees:
  - Prim's Algorithm,
  - Kruskal's Algorithm
- Single source shortest paths
  - Dijkstra's Algorithm
- Optimal Tree problem:
  - Huffman Trees and Codes
- Transform and Conquer Approach:
  - Heaps
  - Heap Sort

Job sequencing with deadlines

Problem statement

- Given an array of **n jobs**
- Every job has a **deadline** and
- Every job has a associated **profit** if the job is finished before the deadline.
- Given that every job takes **single** unit of time for processing.
- How the jobs can be scheduled, to maximize total profit if only one job can be scheduled at a time?

Job sequencing with deadlines

Let $n = 4, (p_1, p_2, p_3, p_4) = (100, 10, 15, 27)$ and
$(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$.

Each Job requires 1 unit time for Processing

The feasible solutions and their values are:

	feasible solution	processing sequence	value
1.	(1, 2)	2, 1	110
2.	(1, 3)	1, 3 or 3, 1	115
3.	(1, 4)	4, 1	127
4.	(2, 3)	2, 3	25
5.	(3, 4)	4, 3	42
6.	(1)	1	100
7.	(2)	2	10
8.	(3)	3	15
9.	(4)	4	27

- The greedy strategy to solve job sequencing problem is,
  - "At each time select the job that that satisfies the constraints and gives **maximum profit.**
  - i.e consider the jobs in the non increasing order of the p_i's

```
Algorithm GreedyJob(d, J, n)
// J is a set of jobs that can be completed by their deadlines.
{
     J := {1};
     for i := 2 to n do
     {
          if (all jobs in J ∪ {i} can be completed
               by their deadlines) then J := J ∪ {i};
     }
}
```

```
int JS(int d[], int j[], int n)
// d[i]>=1, 1<=i<=n are the deadlines, n>=1. The jobs
// are ordered such that p[1]>=p[2]>= ... >=p[n]. J[i]
// is the ith job in the optimal solution, 1<=i<=k.
// Also, at termination d[J[i]]<=d[J[i+1]], 1<=i<k.
{
    d[0] = J[0] = 0; // Initialize.
    J[1] = 1; // Include job 1.
    int k=1;
    for (int i=2; i<=n; i++) {
    // Consider jobs in nonincreasing
    // order of p[i]. Find position for
    // i and check feasibility of insertion.
       int r = k;
       while ((d[J[r]] > d[i]) && (d[J[r]] != r)) r--;
       if ((d[J[r]] <= d[i]) && (d[i] > r)) {
          // Insert i into J[].
          for (int q=k; q>=(r+1); q--) J[q+1] = J[q];
          J[r+1] = i; k++;
       }
    }
    return (k);
}
```

i – job number
k – slot number considered
(k++ when job is assigned)
r+1 – slot where i^{th} job is inserted

```
   0  1   2   3   4   5   6   7   8   9
d = {0, 5, 3, 2, 2, 3, 4, 7, 7, 5};
p = {0,30,25,23,20,18,18,16,15,10};
```

Initially : J = 0 1 0 0 0 0 0 0

Iteration i=2: J = 0 2 1 0 0 0 0 0

Iteration i=3: J = 0 3 2 1 0 0 0 0

Iteration i=4: J = 0 3 4 2 1 0 0 0

Iteration i=5: J = 0 3 4 2 1 0 0 0

Iteration i=6: J = 0 3 4 2 6 1 0 0

Iteration i=7: J = 0 3 4 2 6 1 7 0

Iteration i=8: J = 0 3 4 2 6 1 7 8

Iteration i=9: J = 0 3 4 2 6 1 7 8

```
   0  1   2   3   4   5   6   7   8   9
d = {0, 5, 3, 2, 2, 3, 4, 7, 7, 5};
p = {0,30,25,23,20,18,18,16,15,10};
```

i – job number
k – slot number considered
(k++ when job is assigned)
r+1 – slot where i^{th} job is inserted

```
int JS(int d[], int j[], int n)
{
    d[0] = J[0] = 0; // Initialize.
    J[1] = 1; // Include job 1.
    int k=1;
    for (int i=2; i<=n; i++) {
       int r = k;
       while ((d[J[r]] > d[i]) && (d[J[r]] != r)) r--;
       if ((d[J[r]] <= d[i]) && (d[i] > r)) {
          // Insert i into J[].
          for (int q=k; q>=(r+1); q--) J[q+1] = J[q];
          J[r+1] = i; k++;
       }
    }
    return (k);
}
```

J = 0 1 0 0 0 0 0 0

Locates to the $r+1^{th}$ location where i^{th} job can be scheduled

i = 2

while r=1, J[r]=1, d[J[r]]=5 > d[i]=3 && d[J[r]]=5 != r=1 TRUE r--; r=0
while r=0, J[r]=0, d[J[r]]=0 > d[i]=3 && d[J[r]]=0 != r=0 FALSE
if r=0, J[r]=0, d[J[r]]=0 <= d[i]=3 && d[i]=3 > r=0 TRUE
for q=1: J[q]=1 J[q+1]=1
Assign J[r+1]=J[1]=2
k++; k=2 J = 0 2 1 0 0 0 0 0

```
   0  1   2   3   4   5   6   7   8   9
d = {0, 5, 3, 2, 2, 3, 4, 7, 7, 5};
p = {0,30,25,23,20,18,18,16,15,10};
```

i – job number
k – slot number considered
(k++ when job is assigned)
r+1 – slot where i^{th} job is inserted

```
int JS(int d[], int j[], int n)
{
    d[0] = J[0] = 0; // Initialize.
    J[1] = 1; // Include job 1.
    int k=1;
    for (int i=2; i<=n; i++) {
       int r = k;
       while ((d[J[r]] > d[i]) && (d[J[r]] != r)) r--;
       if ((d[J[r]] <= d[i]) && (d[i] > r)) {
          // Insert i into J[].
          for (int q=k; q>=(r+1); q--) J[q+1] = J[q];
          J[r+1] = i; k++;
       }
    }
    return (k);
}
```

J = 0 2 1 0 0 0 0 0

i = 3

while r=2, J[r]=1, d[J[r]]=5 > d[i]=2 && d[J[r]]=5 != r=2 TRUE r--; r=1
while r=1, J[r]=2, d[J[r]]=3 > d[i]=2 && d[J[r]]=3 != r=1 TRUE r--; r=0
while r=0, J[r]=0, d[J[r]]=0 > d[i]=2 && d[J[r]]=0 != r=0 FALSE
if r=0, J[r]=0, d[J[r]]=0 <= d[i]=2 && d[i]=2 > r=0 TRUE
for q=2: J[q]=1 J[q+1]=1
for q=1: J[q]=2 J[q+1]=2
Assign J[r+1]=J[1]=3
k++; k=3 J = 0 3 2 1 0 0 0 0

```
   0  1   2   3   4   5   6   7   8   9
d = {0, 5, 3, 2, 2, 3, 4, 7, 7, 5};
p = {0,30,25,23,20,18,18,16,15,10};
```

i – job number
k – slot number considered
(k++ when job is assigned)
r+1 – slot where i^{th} job is inserted

```
int JS(int d[], int j[], int n)
{
    d[0] = J[0] = 0; // Initialize.
    J[1] = 1; // Include job 1.
    int k=1;
    for (int i=2; i<=n; i++) {
       int r = k;
       while ((d[J[r]] > d[i]) && (d[J[r]] != r)) r--;
       if ((d[J[r]] <= d[i]) && (d[i] > r)) {
          // Insert i into J[].
          for (int q=k; q>=(r+1); q--) J[q+1] = J[q];
          J[r+1] = i; k++;
       }
    }
    return (k);
}
```

J = 0 3 2 1 0 0 0 0

i = 4

while r=3, J[r]=1, d[J[r]]=5 > d[i]=2 && d[J[r]]=5 != r=3 TRUE r--; r=2
while r=2, J[r]=2, d[J[r]]=3 > d[i]=2 && d[J[r]]=3 != r=2 TRUE r--; r=1
while r=1, J[r]=3, d[J[r]]=2 > d[i]=2 && d[J[r]]=2 != r=1 FALSE
if r=1, J[r]=3, d[J[r]]=2 <= d[i]=2 && d[i]=2 > r=1 TRUE
for q=3: J[q]=1 J[q+1]=1
for q=2: J[q]=2 J[q+1]=2
Assign J[r+1]=J[2]=4
k++; k=4 J = 0 3 4 2 1 0 0 0

```
   0  1   2   3   4   5   6   7   8   9
d = {0, 5, 3, 2, 2, 3, 4, 7, 7, 5};
p = {0,30,25,23,20,18,18,16,15,10};
```

i – job number
k – slot number considered
(k++ when job is assigned)
r+1 – slot where i^{th} job is inserted

```
int JS(int d[], int j[], int n)
{
    d[0] = J[0] = 0; // Initialize.
    J[1] = 1; // Include job 1.
    int k=1;
    for (int i=2; i<=n; i++) {
       int r = k;
       while ((d[J[r]] > d[i]) && (d[J[r]] != r)) r--;
       if ((d[J[r]] <= d[i]) && (d[i] > r)) {
          // Insert i into J[].
          for (int q=k; q>=(r+1); q--) J[q+1] = J[q];
          J[r+1] = i; k++;
       }
    }
    return (k);
}
```

J = 0 3 4 2 1 0 0 0

i = 5

while r=4, J[r]=1, d[J[r]]=5 > d[i]=3 && d[J[r]]=5 != r=4 TRUE r--; r=3
while r=3, J[r]=2, d[J[r]]=3 > d[i]=3 && d[J[r]]=3 != r=3 FALSE
if r=3, J[r]=2, d[J[r]]=3 <= d[i]=3 && d[i]=3 > r=3 FALSE

J = 0 3 4 2 1 0 0 0

```
   0  1   2   3   4   5   6   7   8   9
d = {0, 5, 3, 2, 2, 3, 4, 7, 7, 5};
p = {0,30,25,23,20,18,18,16,15,10};
```

i – job number
k – slot number considered
(k++ when job is assigned)
r+1 – slot where i^{th} job is inserted

```
int JS(int d[], int j[], int n)
{
    d[0] = J[0] = 0; // Initialize.
    J[1] = 1; // Include job 1.
    int k=1;
    for (int i=2; i<=n; i++) {
       int r = k;
       while ((d[J[r]] > d[i]) && (d[J[r]] != r)) r--;
       if ((d[J[r]] <= d[i]) && (d[i] > r)) {
          // Insert i into J[].
          for (int q=k; q>=(r+1); q--) J[q+1] = J[q];
          J[r+1] = i; k++;
       }
    }
    return (k);
}
```

J = 0 3 4 2 1 0 0 0

i = 6

while r=4, J[r]=1, d[J[r]]=5 > d[i]=4 && d[J[r]]=5 != r=4 TRUE r--; r=3
while r=3, J[r]=2, d[J[r]]=3 > d[i]=4 && d[J[r]]=3 != r=3 FALSE
if r=3, J[r]=2, d[J[r]]=3 <= d[i]=4 && d[i]=4 > r=3 TRUE
for q=4: J[q]=1 J[q+1]=1
Assign J[r+1]=J[4]=6
k++; k=5 J = 0 3 4 2 6 1 0 0

```
   0  1   2   3   4   5   6   7   8   9
d = {0, 5, 3, 2, 2, 3, 4, 7, 7, 5};
p = {0,30,25,23,20,18,18,16,15,10};
```

i – job number
k – slot number
(k++ when job is assigned)
r+1 – slot where i^{th} job is inserted

```
int JS(int d[], int j[], int n)
{
    d[0] = J[0] = 0; // Initialize.
    J[1] = 1; // Include job 1.
    int k=1;
    for (int i=2; i<=n; i++) {
       int r = k;
       while ((d[J[r]] > d[i]) && (d[J[r]] != r)) r--;
       if ((d[J[r]] <= d[i]) && (d[i] > r)) {
          // Insert i into J[].
          for (int q=k; q>=(r+1); q--) J[q+1] = J[q];
          J[r+1] = i; k++;
       }
    }
    return (k);
}
```

J = 0 3 4 2 6 1 0 0

i = 7

while r=5, J[r]=1, d[J[r]]=5 > d[i]=7 && d[J[r]]=5 != r=5 FALSE
if r=5, J[r]=1, d[J[r]]=5 <= d[i]=7 && d[i]=7 > r=5 TRUE
Assign J[r+1]=J[6]=7
k++; k=6 J = 0 3 4 2 6 1 7 0

```
   0  1   2   3   4   5   6   7   8   9
d = {0, 5, 3, 2, 2, 3, 4, 7, 7, 5};
p = {0,30,25,23,20,18,18,16,15,10};
```

i – job number
k – slot number
(k++ when job is assigned)
r+1 – slot where i^{th} job is inserted

```
int JS(int d[], int j[], int n)
{
    d[0] = J[0] = 0; // Initialize.
    J[1] = 1; // Include job 1.
    int k=1;
    for (int i=2; i<=n; i++) {
       int r = k;
       while ((d[J[r]] > d[i]) && (d[J[r]] != r)) r--;
       if ((d[J[r]] <= d[i]) && (d[i] > r)) {
          // Insert i into J[].
          for (int q=k; q>=(r+1); q--) J[q+1] = J[q];
          J[r+1] = i; k++;
       }
    }
    return (k);
}
```

J = 0 3 4 2 6 1 7 0

i = 8

while r=6, J[r]=7, d[J[r]]=7 > d[i]=7 && d[J[r]]=7 != r=6 FALSE
if r=6, J[r]=7, d[J[r]]=7 <= d[i]=7 && d[i]=7 > r=6 TRUE
Assign J[r+1]=J[7]=8
k++; k=7 J = 0 3 4 2 6 1 7 8

```
   0  1   2   3   4   5   6   7   8   9
d = {0, 5, 3, 2, 2, 3, 4, 7, 7, 5};
p = {0,30,25,23,20,18,18,16,15,10};
```

i – job number
k – slot number
(k++ when job is assigned)
r+1 – slot where i^{th} job is inserted

```
int JS(int d[], int j[], int n)
{
    d[0] = J[0] = 0; // Initialize.
    J[1] = 1; // Include job 1.
    int k=1;
    for (int i=2; i<=n; i++) {
       int r = k;
       while ((d[J[r]] > d[i]) && (d[J[r]] != r)) r--;
       if ((d[J[r]] <= d[i]) && (d[i] > r)) {
          // Insert i into J[].
          for (int q=k; q>=(r+1); q--) J[q+1] = J[q];
          J[r+1] = i; k++;
       }
    }
    return (k);
}
```

J = 0 3 4 2 6 1 7 8

i = 9

while r=7, J[r]=8, d[J[r]]=7 > d[i]=5 && d[J[r]]=7 != r=7 FALSE
if r=7, J[r]=8, d[J[r]]=7 <= d[i]=5 && d[i]=5 > r=7 FALSE
J = 0 3 4 2 6 1 7 8

Analysis

```
int JS(int d[], int j[], int n)
// d[i]>=1, 1<=i<=n are the deadlines, n>=1. The jobs
// are ordered such that p[1]>=p[2]>= ... >=p[n]. J[i]
// is the ith job in the optimal solution, 1<=i<=k.
// Also, at termination d[J[i]]<=d[J[i+1]], 1<=i<k.
{
    d[0] = J[0] = 0; // Initialize.
    J[1] = 1; // Include job 1.
    int k=1;
    for (int i=2; i<=n; i++) {
    // Consider jobs in nonincreasing
    // order of p[i]. Find position for
    // i and check feasibility of insertion.
       int r = k;
       while ((d[J[r]] > d[i]) && (d[J[r]] != r)) r--;
       if ((d[J[r]] <= d[i]) && (d[i] > r)) {
          // Insert i into J[].
          for (int q=k; q>=(r+1); q--) J[q+1] = J[q];
          J[r+1] = i; k++;
       }
    }
    return (k);
}
```

i – slot number
k – job number

Repeats n-1 times

Repeats at most k times

Repeats say, **s** times

$\Theta(ns) = \Theta(n^2)$

Analysis

For JS there are two possible parameters in terms of which its complexity can be measured. We can use n, the number of jobs, and s, the number of jobs included in the solution J. The **while** loop of line 15 in Algorithm 4.6 is iterated at most k times. Each iteration takes $\Theta(1)$ time. If the conditional of line 16 is true, then lines 19 and 20 are executed. These lines require $\Theta(k-r)$ time to insert job i. Hence, the total time for each iteration of the **for** loop of line 10 is $\Theta(k)$. This loop is iterated $n-1$ times. If s is the final value of k, that is, s is the number of jobs in the final solution, then the total time needed by algorithm JS is $\Theta(sn)$. Since $s \leq n$, the worst-case time, as a function of n alone is $\Theta(n^2)$. If we consider the job set $p_i = d_i = n - i + 1$, $1 \leq i \leq n$, then algorithm JS takes $\Theta(n^2)$ time to determine J. Hence, the worst-case computing time for JS is $\Theta(n^2)$. In addition to the space needed for d, JS needs $\Theta(s)$ amount of space for J.

Alternate Method

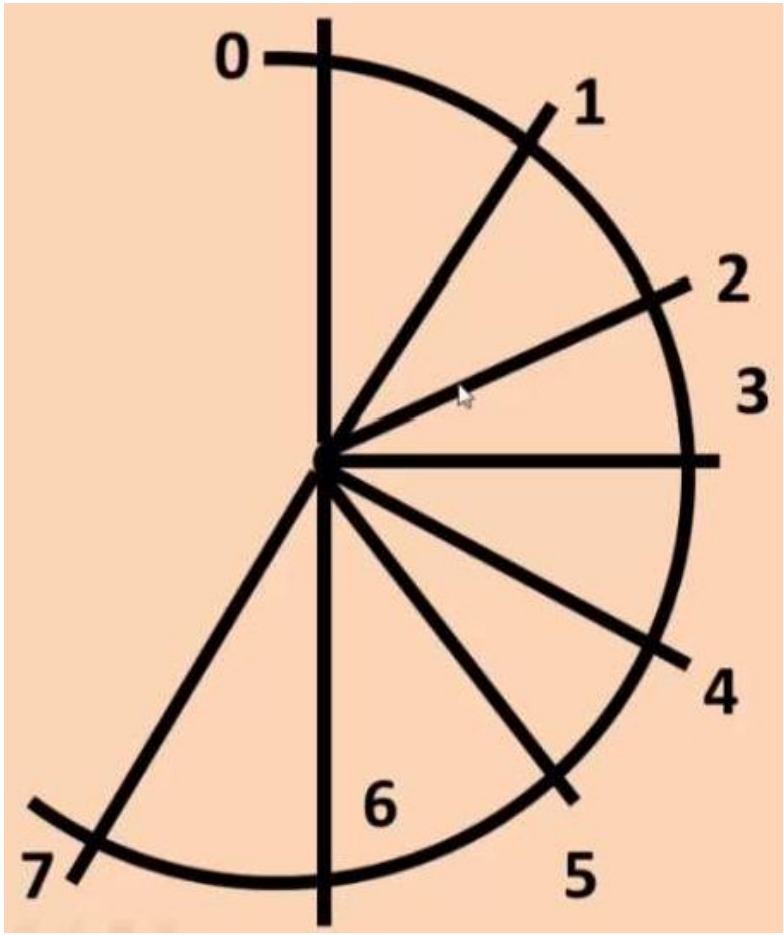
- Alternate Method with efficiency O(n)

1. Sort the jobs so that: p1≥ p2 ≥ p3 ≥ . . . ≥ pn
2. for t = 1:n

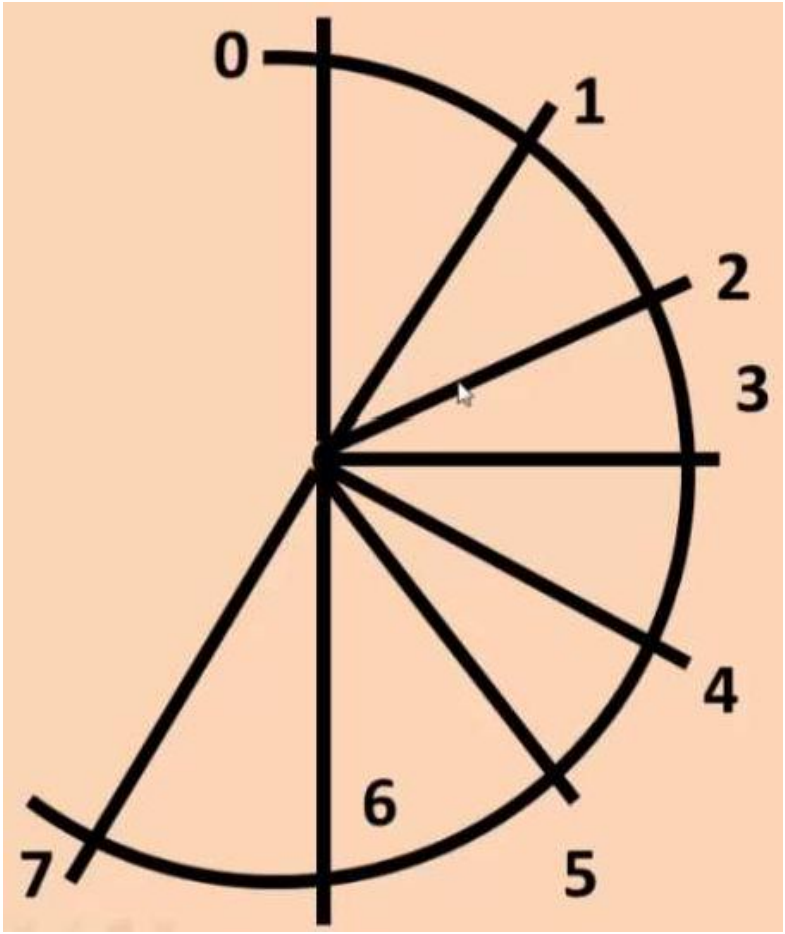
   J(t) =0 (Initialize array J with zeros)
3. for i = 1:n
   1. Schedule job i in the latest possible free slot meeting its deadline;
   2. if there is no such slot, do not schedule i.

Fast Job Scheduling

Tasks	Deadlines	Profit
T1	7	15
T2	2	20
T3	5	30
T4	3	18
T5	4	18
T6	5	10
T7	2	23
T8	7	16
T9	3	25



Tasks	Deadlines	Profit
T3	5	30
T9	3	25
T7	2	23
T2	2	20
T4, T5	3, 4	18, 18
T8	7	16
T1	7	15
T6	5	10



Fast Job Scheduling Algorithm

Example 4.3 Let $n = 5, (p_1, \ldots, p_5) = (20, 15, 10, 5, 1)$ and $(d_1, \ldots, d_5) = (2, 2, 1, 3, 3)$. Using the above feasibility rule, we have

J	assigned slots	job considered	action	profit
$\emptyset$	none	1	assign to [1, 2]	0
{1, 2}	[0, 1], [1, 2]	3	cannot fit; reject	35
{1, 2}	[0, 1], [1, 2]	4	assign to [2, 3]	35
{1, 2, 4}	[0, 1], [1, 2], [2, 3]	5	reject	40

Fast Job Scheduling Algorithm

Example 4.3 Let $n = 5, (p_1, \ldots, p_5) = (20, 15, 10, 5, 1)$ and $(d_1, \ldots, d_5) = (2, 2, 1, 3, 3)$. Using the above feasibility rule, we have

J	assigned slots	job considered	action	profit
$\emptyset$	none	1	assign to [1, 2]	0
{1}	[1, 2]	2	assign to [0, 1]	20
{1, 2}	[0, 1], [1, 2]	3	cannot fit; reject	35
{1, 2}	[0, 1], [1, 2]	4	assign to [2, 3]	35
{1, 2, 4}	[0, 1], [1, 2], [2, 3]	5	reject	40

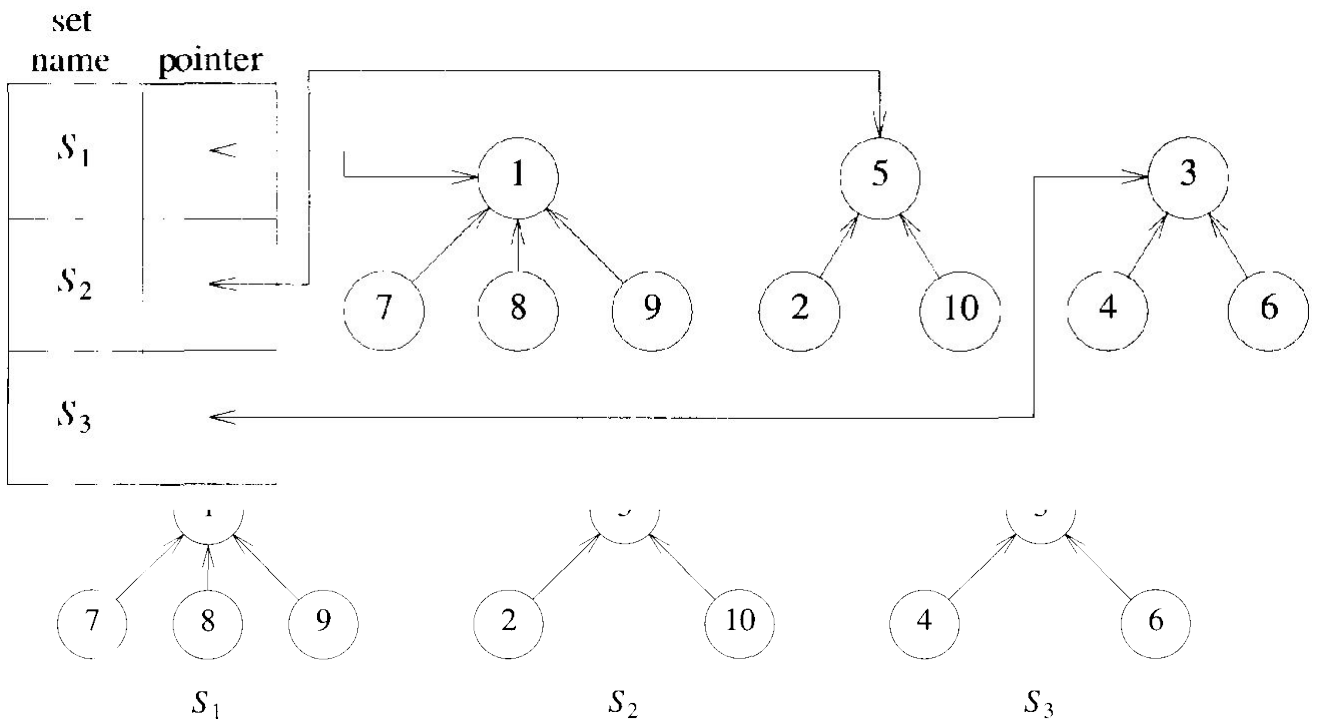
The optimal solution is $J = \{1, 2, 4\}$ with a profit of 40. □

Complexity

O(n)

- If disjoint set union & find algorithm is used

Data Structures for



i	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
p	-1	5	-1	3	-1	3	1	1	1	5

i	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
p	-1	5	-1	3	-1	3	1	1	1	5

Algorithm SimpleUnion(i, j)
{
    $p[i] := j$**;**
}

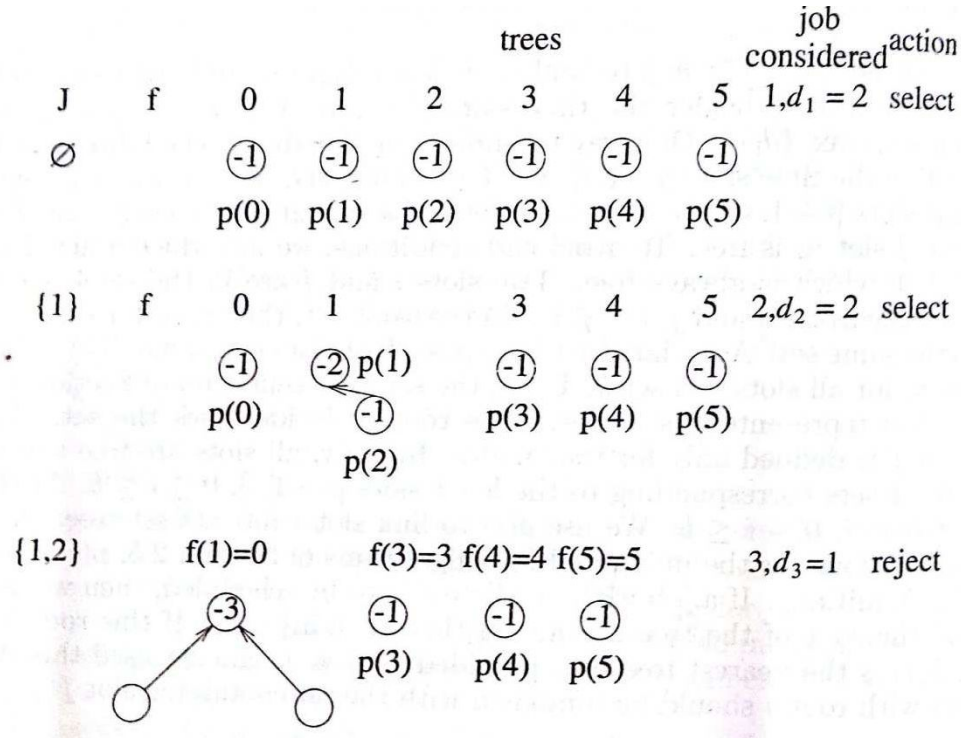
Algorithm SimpleFind(i)
{
    while $(p[i] \geq 0)$ **do** $i := p[i]$**;**
    return i**;**
}

Example 4.3 Let $n = 5, (p_1, \ldots, p_5) = (20, 15, 10, 5, 1)$ and $(d_1, \ldots, d_5) = (2, 2, 1, 3, 3)$. Using the above feasibility rule, we have

J	assigned slots	job considered	action	profit
$\emptyset$	none	1	assign to [1, 2]	0
{1}	[1, 2]	2	assign to [0, 1]	20
{1, 2}	[0, 1], [1, 2]	3	cannot fit; reject	35
{1, 2}	[0, 1], [1, 2]	4	assign to [2, 3]	35
{1, 2, 4}	[0, 1], [1, 2], [2, 3]	5	reject	40

The optimal solution is $J = \{1, 2, 4\}$ with a profit of 40. □

Example 4.4 The trees defined by the $p(i)$'s for the first three iterations in Example 4.3 are shown in Figure 4.4. □



```
Algorithm FJS(d, n, b, j)
// Find an optimal solution J[1 : k]. It is assumed that
// p[1] ≥ p[2] ≥ ··· ≥ p[n] and that b = min{n, max_i(d[i])}.
{
    // Initially there are b + 1 single node trees.
    for i := 0 to b do f[i] := i;
    k := 0; // Initialize.
    for i := 1 to n do
    { // Use greedy rule.
        q := CollapsingFind(min(n, d[i]));
        if (f[q] ≠ 0) then
        {
            k := k + 1; J[k] := i; // Select job i.
            m := CollapsingFind(f[q] − 1);
            WeightedUnion(m, q);
            f[q] := f[m]; // q may be new root.
        }
    }
}
```

Analysis

The fast algorithm appears as FJS (Algorithm 4.7). Its computing time is readily observed to be $O(n\alpha(2n, n))$ (recall that $\alpha(2n, n)$ is the inverse of Ackermann's function defined in Section 2.5). It needs an additional $2n$ words of space for f and p.

Module 3: Greedy Method - Outline

- Introduction to Greedy method
  - General method,
  - Coin Change Problem
  - Knapsack Problem
  - Job sequencing with deadlines
- Minimum cost spanning trees:
  - Prim's Algorithm,
  - Kruskal's Algorithm
- Single source shortest paths
  - Dijkstra's Algorithm
- Optimal Tree problem:
  - Huffman Trees and Codes
- Transform and Conquer Approach:
  - Heaps
  - Heap Sort

Minimum Spanning Tree MST

- A **spanning tree** of a connected graph is its connected acyclic subgraph (i.e., a tree) that contains all the vertices of the graph.
- A **minimum spanning tree** of a weighted connected graph is its spanning tree of the smallest weight, where the weight of a tree is defined as the sum of the **weights** on all its edges.
- The **minimum spanning tree problem** is the problem of finding a minimum spanning tree for a given weighted connected graph.

MST - Example

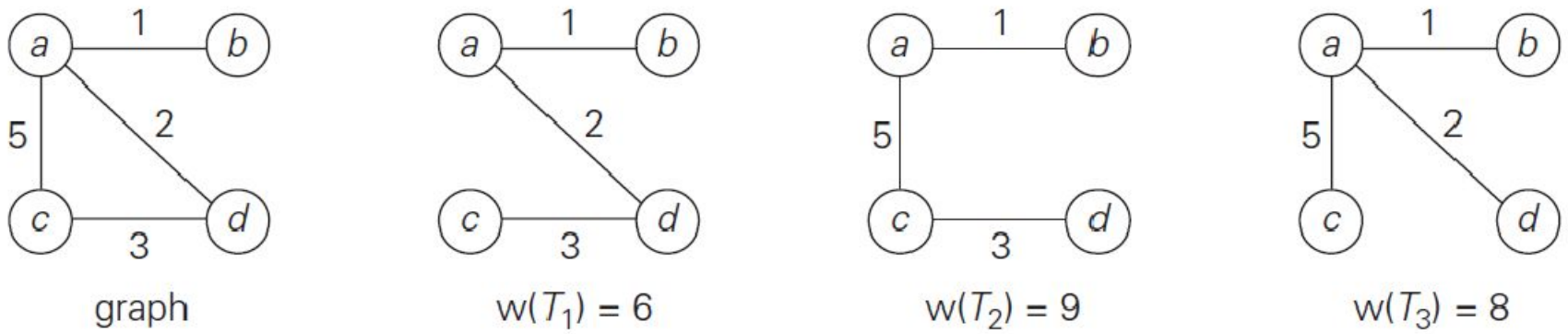
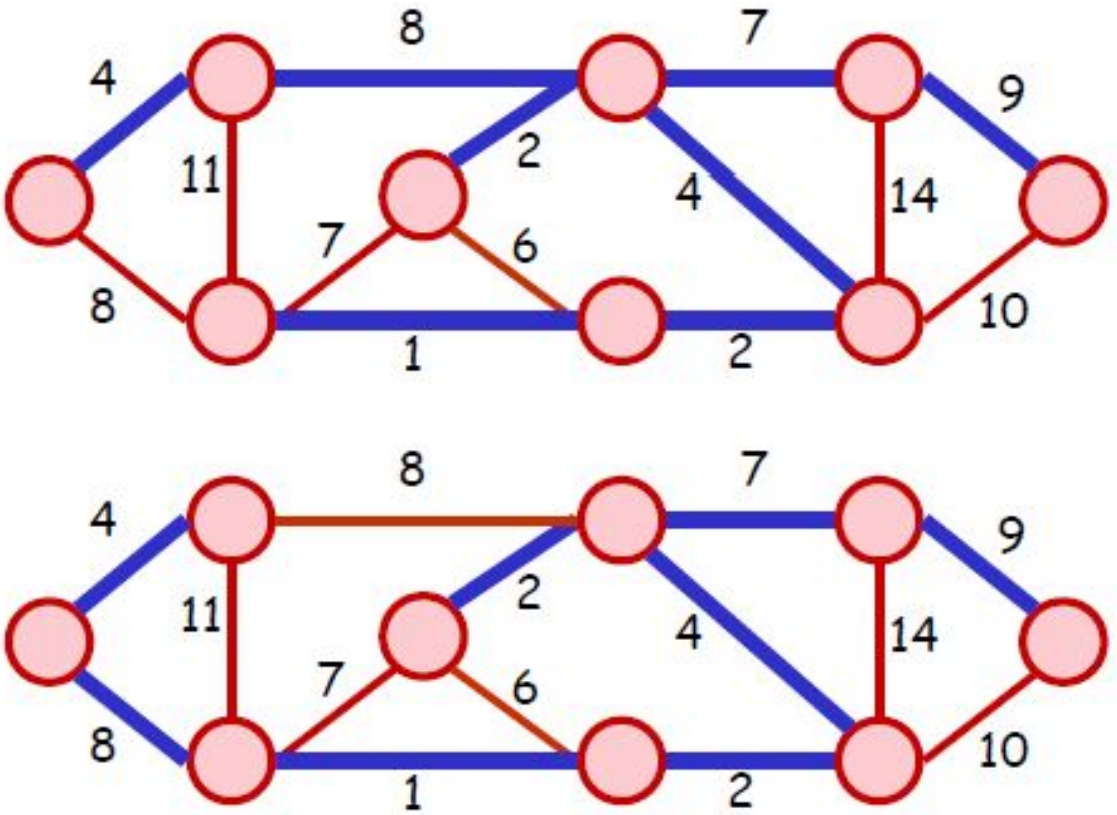


FIGURE 9.2 Graph and its spanning trees, with T_1 being the minimum spanning tree.

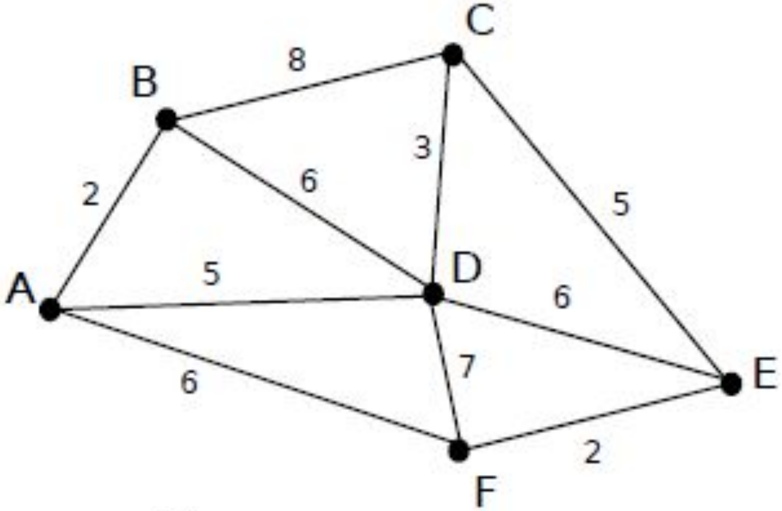
MST of a graph may not be unique

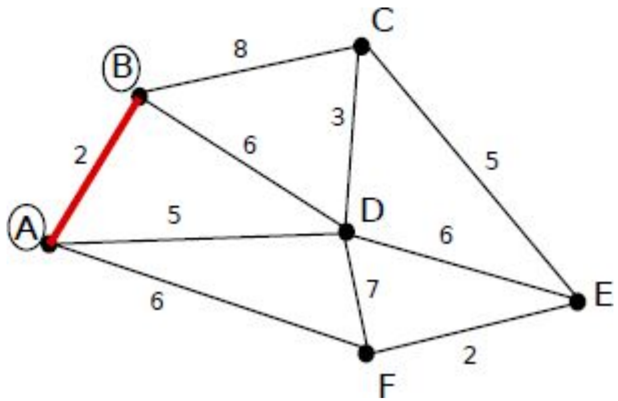


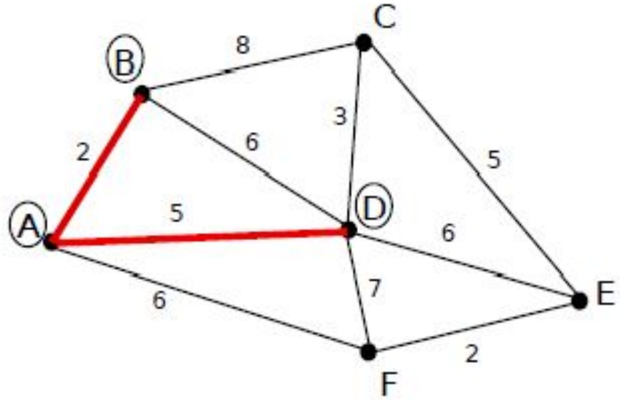
Module 3: Greedy Method - Outline

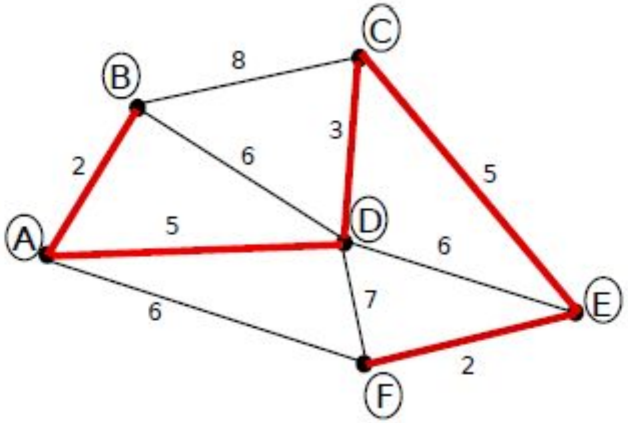
- Introduction to Greedy method
  - General method,
  - Coin Change Problem
  - Knapsack Problem
  - Job sequencing with deadlines
- Minimum cost spanning trees:
  - Prim's Algorithm,
  - Kruskal's Algorithm
- Single source shortest paths
  - Dijkstra's Algorithm
- Optimal Tree problem:
  - Huffman Trees and Codes
- Transform and Conquer Approach:
  - Heaps
  - Heap Sort

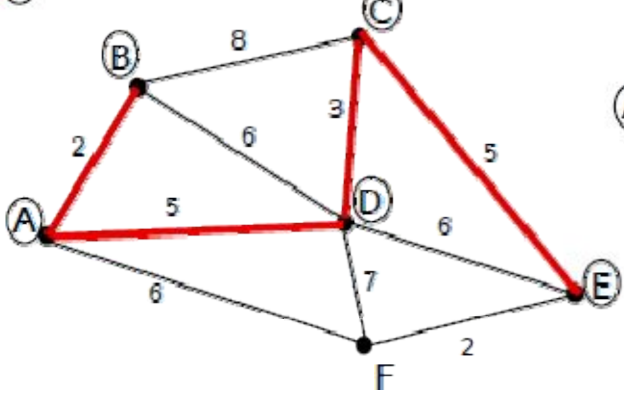
Prims Algorithm-Example

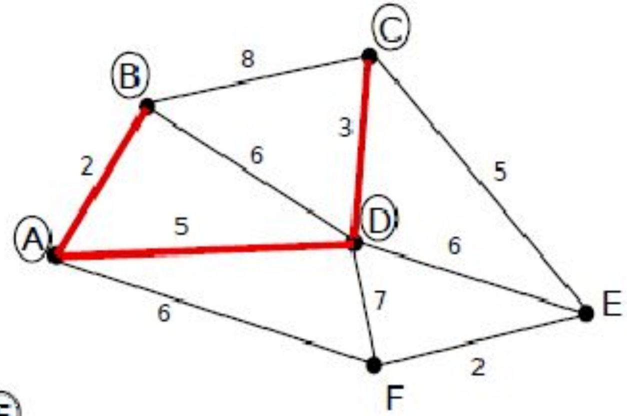




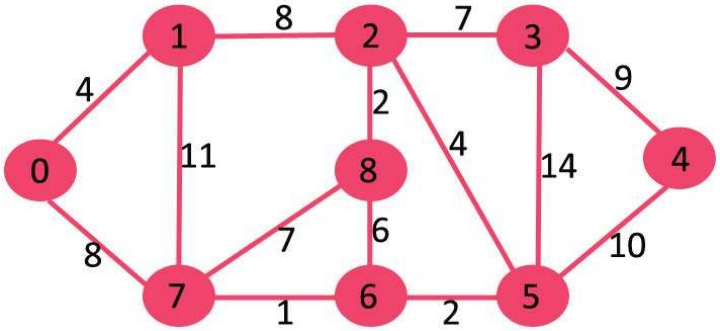


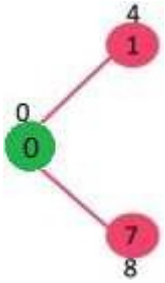


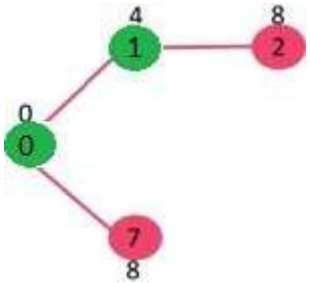


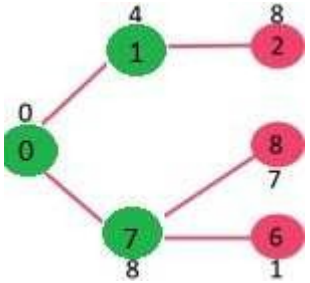


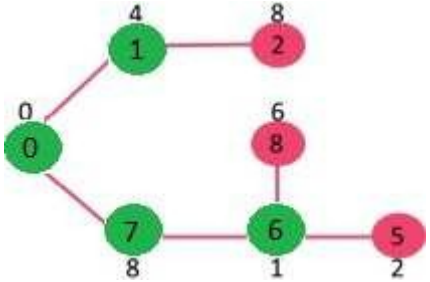
Prim's Algorithm - Example



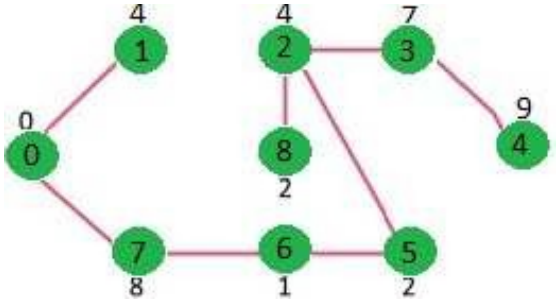




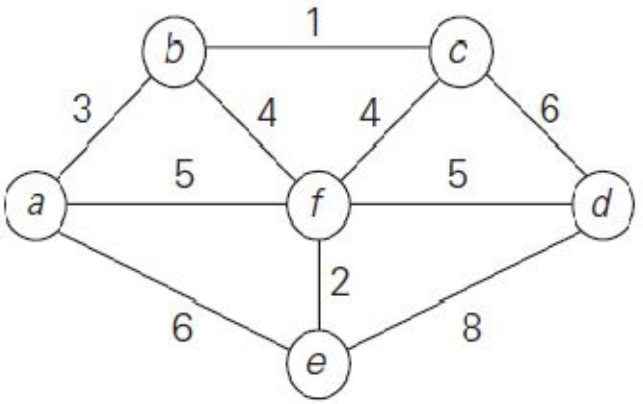


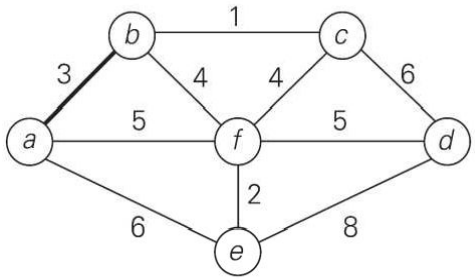
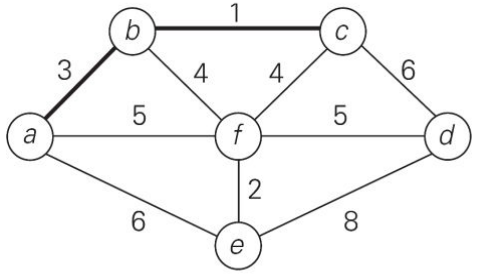
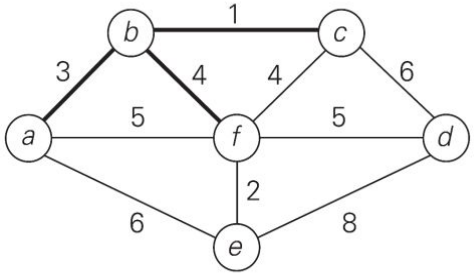


...

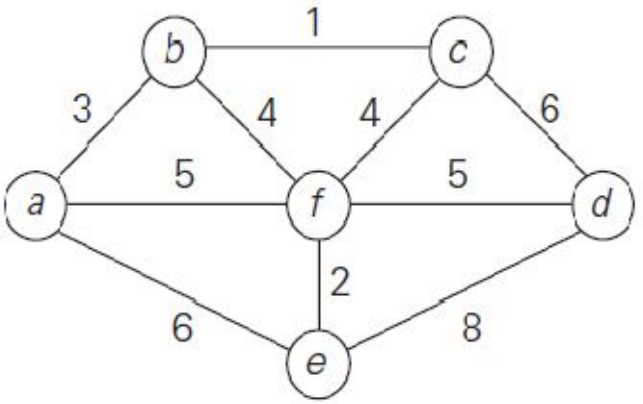


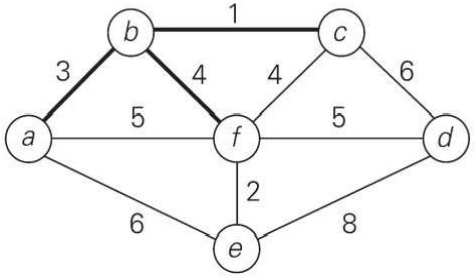
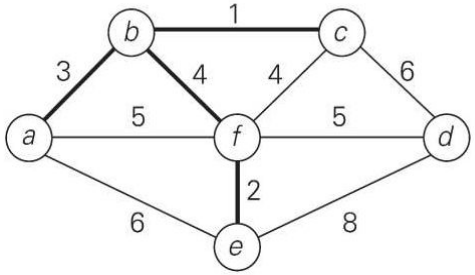
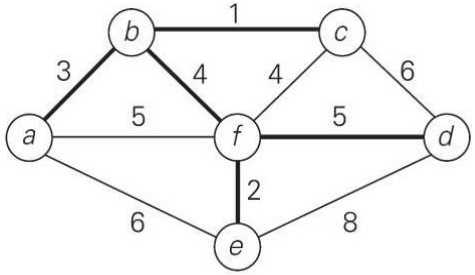
Example from Text book



Tree vertices	*Remaining vertices*	*Illustration*
a(−, −)	**b**(**a**, **3**) c(−, ∞) d(−, ∞) e(a, 6) f(a, 5)	
b(a, 3)	**c**(**b**, **1**) d(−, ∞) e(a, 6) f(b, 4)	
c(b, 1)	d(c, 6) e(a, 6) **f**(**b**, **4**)	

Example from Text book

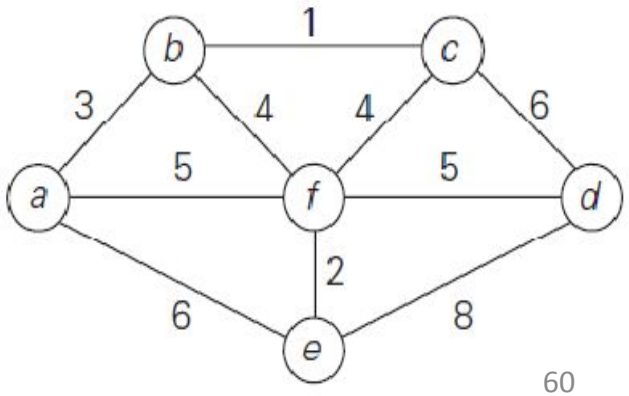


Tree vertices	*Remaining vertices*	*Illustration*
c(b, 1)	d(c, 6) e(a, 6) **f(b, 4)**	
f(b, 4)	d(f, 5) **e(f, 2)**	
e(f, 2)	**d(f, 5)**	
d(f, 5)		

Prim's Algorithm

ALGORITHM *Prim(G)*

//Prim's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E \rangle$
//Output: E_T, the set of edges composing a minimum spanning tree of G
$V_T \leftarrow \{v_0\}$ //the set of tree vertices can be initialized with any vertex
$E_T \leftarrow \emptyset$
for $i \leftarrow 1$ **to** $|V| - 1$ **do**
    find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges (v, u)
    such that v is in V_T and u is in $V - V_T$
    $V_T \leftarrow V_T \cup \{u^*\}$
    $E_T \leftarrow E_T \cup \{e^*\}$
return E_T



Analysis of Efficiency

- Depends on the data structures chosen
  - for the **graph** & for the **priority queue** of the set $V - V_T$ whose vertex priorities are the distances to the nearest tree vertices.
- If graph representation - **weight matrix,**
- priority queue - is implemented as an **unordered array**,

  the running time will be in $\Theta(|V|^2)$.
- On each of the $|V|-1$ iterations,
  - the array implementing the priority queue is traversed to find and delete the minimum and then to update, if necessary, the priorities of the remaining vertices.

Analysis of Efficiency

- We can implement the priority queue as a **min-heap**.
  - A min-heap is a complete binary tree in which every element is less than or equal to its children.
  - Deletion of the smallest element from and insertion of a new element into a min-heap of size n are **O(log n)** operations.
- If a graph is represented as **adjacency lists** and the priority queue is implemented as a **min-heap**,
  - the running time is in **O(|E| log |V|).**

Why running time is in **O(|E| log |V|) ?**

- Algorithm performs |V|−1 deletions of the smallest element and makes |E| verifications
  - and, possibly, changes of an element's priority in a min-heap of size not exceeding |V|.
  - Each of these operations, is a O(log |V|) operation.
- Hence, the running time of this implementation of Prim's algorithm is in

  = (|V| − 1+ |E|) O (log |V |)

  = O(|E| log |V |) because, in a connected graph, |V| − 1≤ |E|.

Module 3: Greedy Method - Outline

- Introduction to Greedy method
  - General method,
  - Coin Change Problem
  - Knapsack Problem
  - Job sequencing with deadlines
- Minimum cost spanning trees:
  - Prim's Algorithm,
  - Kruskal's Algorithm
- Single source shortest paths
  - Dijkstra's Algorithm
- Optimal Tree problem:
  - Huffman Trees and Codes
- Transform and Conquer Approach:
  - Heaps
  - Heap Sort

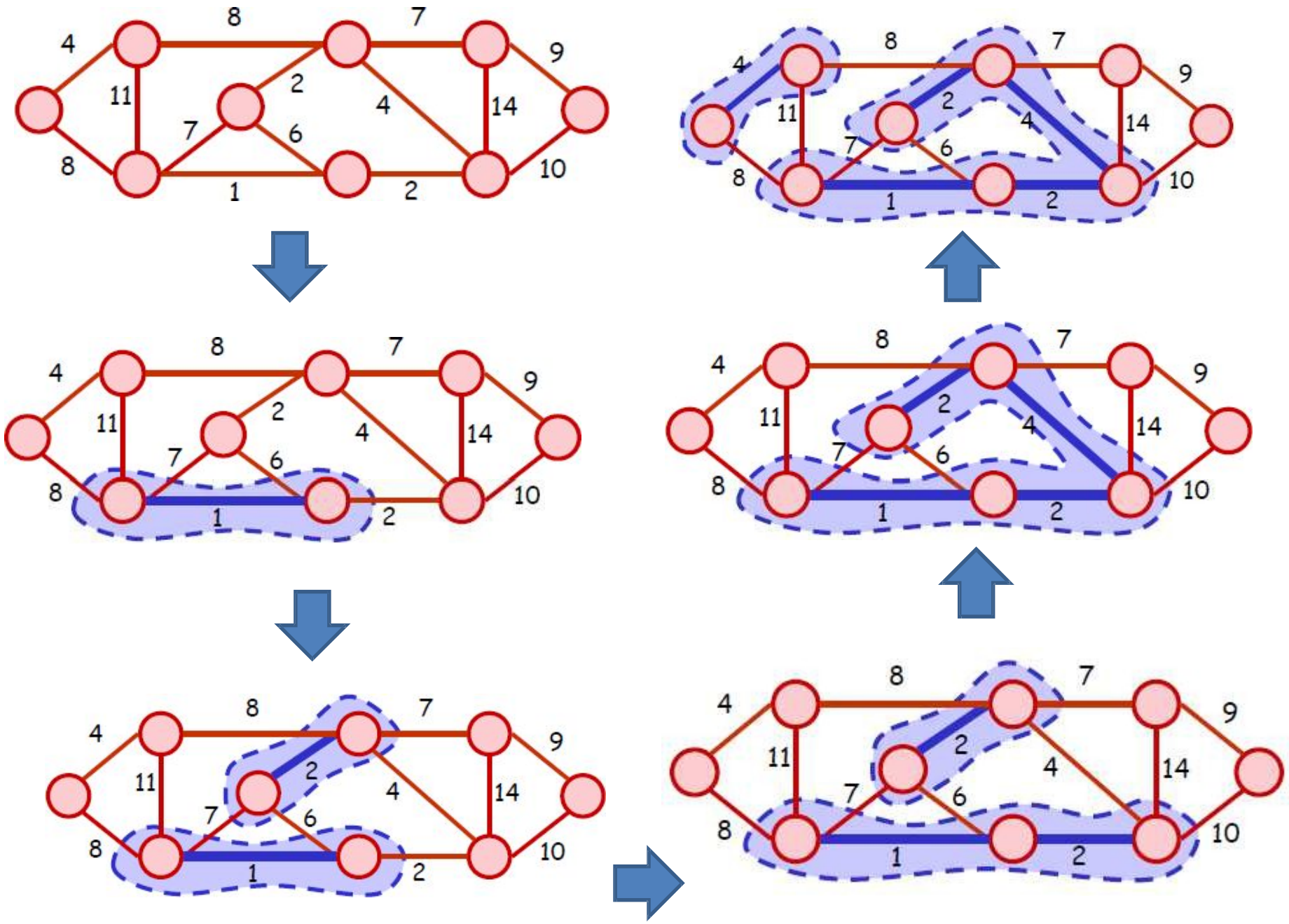
Kruskals Algorithm

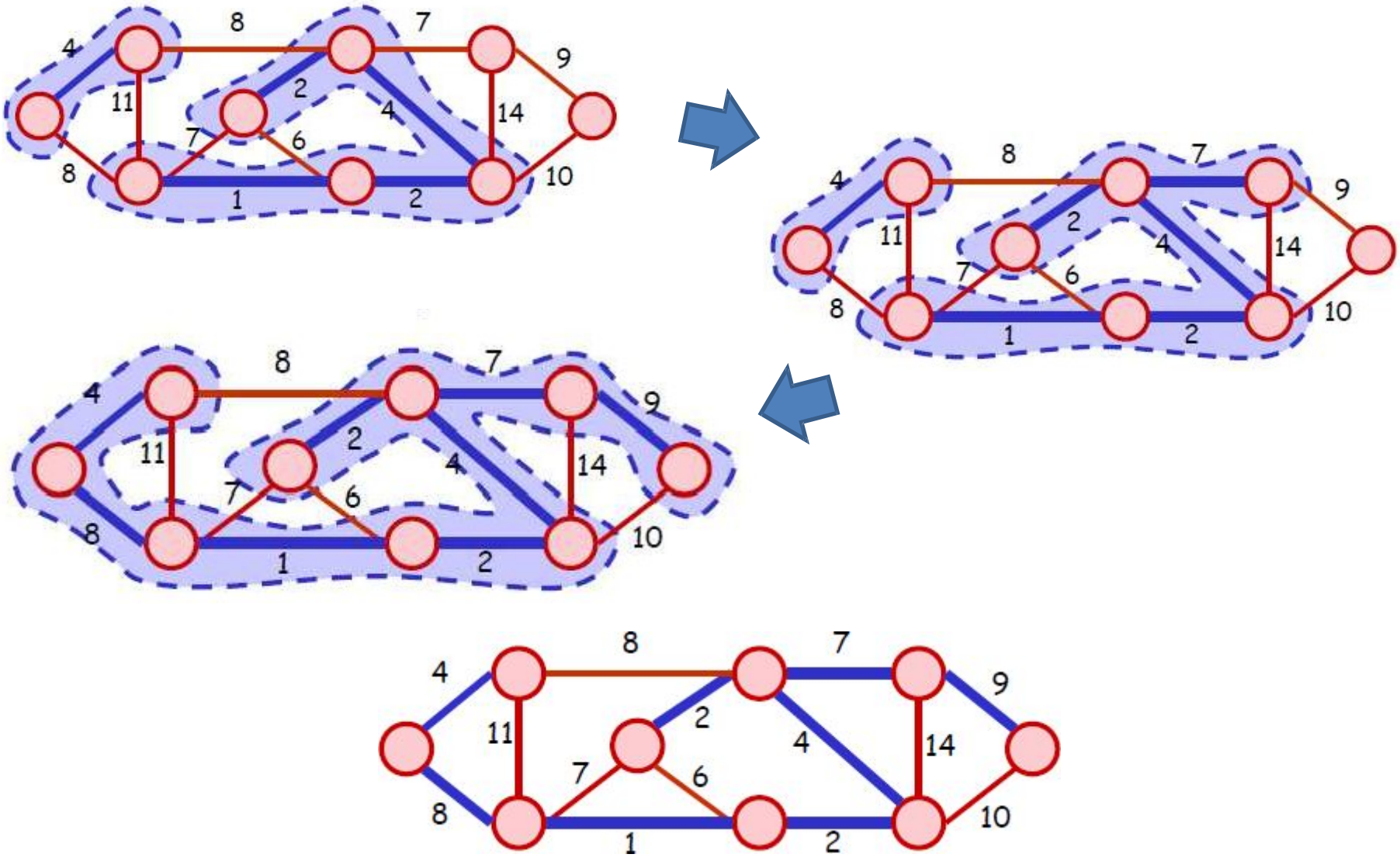
- Algorithm constructs a MST as
  - an expanding sequence of sub graphs,
  - which are always **acyclic**
  - but are not necessarily connected on the intermediate stages of the

Kruskals Algorithm

Working

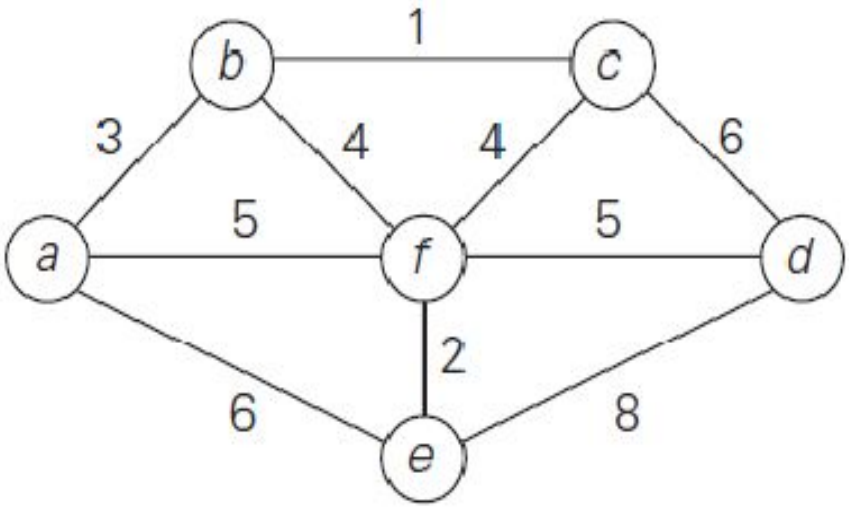
- **Sorting** the graph's edges in **non decreasing** order of their **weights**.
- Then, starting with the empty sub graph,
  - it scans this sorted list **adding the next edge** on the list to the current sub graph if such an inclusion does not create a **cycle** and simply **skipping** the edge otherwise.


4
8
7
9
11
2
4
14
7
6
8
1
2
10

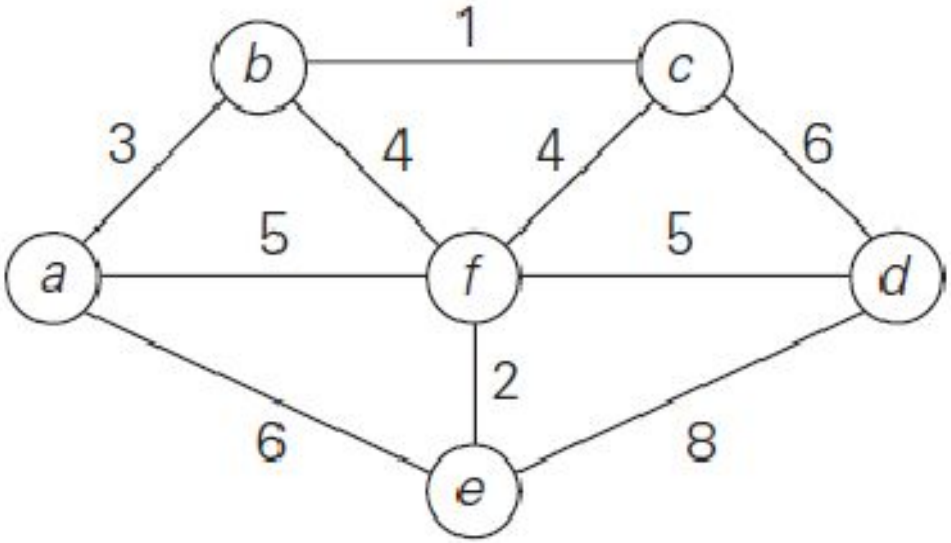

4
8
7
9
11
2
4
14
7
6
8
1
2
10

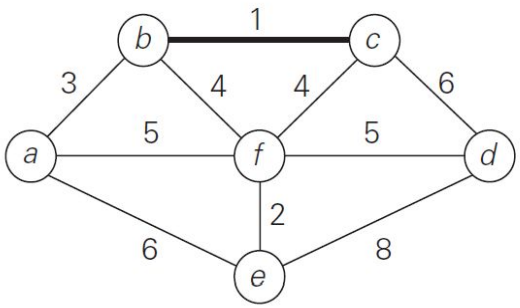
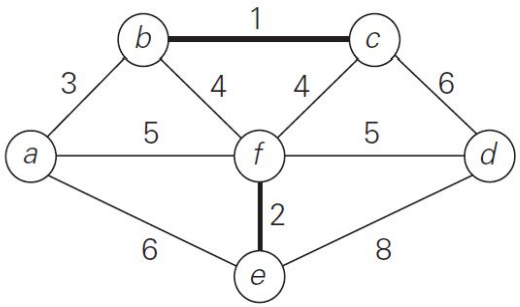
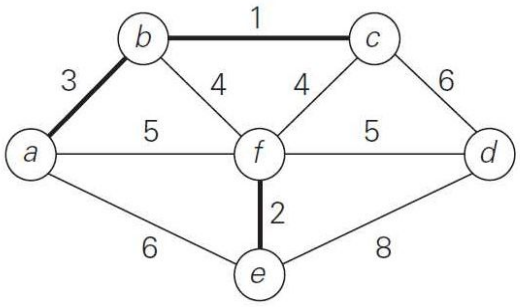
Kruskal Algorithm

- Kruskal's algorithm is **not simpler** because it has to check whether the addition of the next edge to the edges already selected would **create a cycle**.

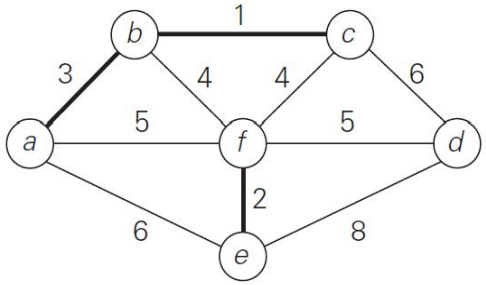


Example from Text book

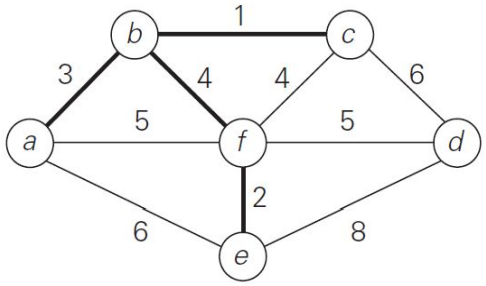


Tree edges	Sorted list of edges	Illustration
	bc ef ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
bc 1	bc **ef** ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
ef 2	bc ef **ab** bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	

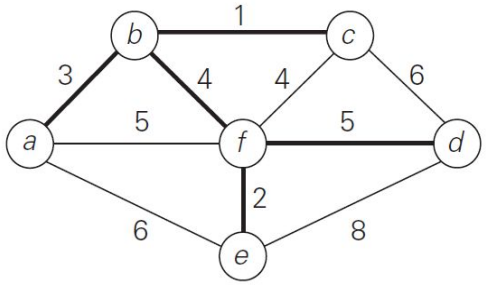
ef	bc	ef	**ab**	bf	cf	af	df	ae	cd	de
2	1	2	3	4	4	5	5	6	6	8



ab	bc	ef	ab	**bf**	cf	af	df	ae	cd	de
3	1	2	3	4	4	5	5	6	6	8



bf	bc	ef	ab	bf	cf	af	**df**	ae	cd	de
4	1	2	3	4	4	5	5	6	6	8



df
5

Kruskal's MST Algorithm

ALGORITHM *Kruskal*(G)

//Kruskal's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E \rangle$
//Output: E_T, the set of edges composing a minimum spanning tree of G
sort E in nondecreasing order of the edge weights $w(e_{i_1}) \le \cdots \le w(e_{i_{|E|}})$
$E_T \leftarrow \varnothing$; $ecounter \leftarrow 0$ //initialize the set of tree edges and its size
$k \leftarrow 0$ //initialize the number of processed edges
while $ecounter < |V| - 1$ **do**
    $k \leftarrow k + 1$
    if $E_T \cup \{e_{i_k}\}$ is acyclic
        $E_T \leftarrow E_T \cup \{e_{i_k}\}$; $ecounter \leftarrow ecounter + 1$
return E_T

Analysis

- The crucial check whether two vertices belong to the same tree can be found out using **union-find algorithms.**
- Efficiency of Kruskal's algorithm is based on the time needed for **sorting the edge weights** of a given graph.
- With an efficient sorting algorithm, the time efficiency of Kruskal's algorithm will be in

O (|E| log |E|).

Module 3: Greedy Method - Outline

- Introduction to Greedy method
  - General method,
  - Coin Change Problem
  - Knapsack Problem
  - Job sequencing with deadlines
- Minimum cost spanning trees:
  - Prim's Algorithm,
  - Kruskal's Algorithm
- Single source shortest paths
  - Dijkstra's Algorithm
- Optimal Tree problem:
  - Huffman Trees and Codes
- Transform and Conquer Approach:
  - Heaps
  - Heap Sort

Single Source Shortest Path

Problem Statement

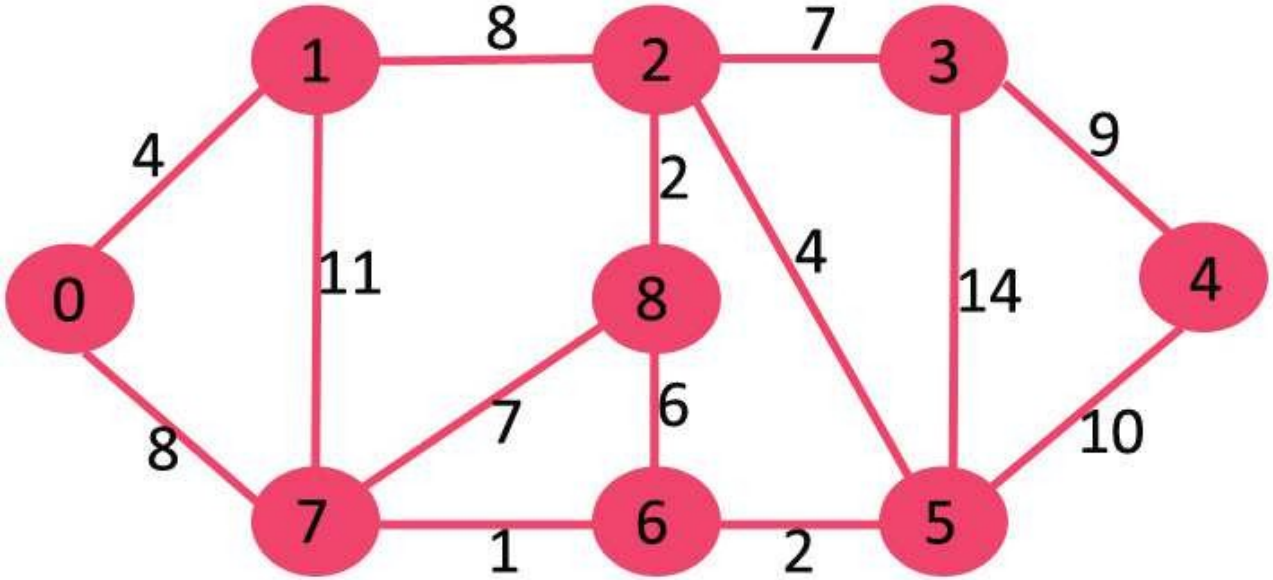
- Given a graph and a source vertex in graph,
- find shortest paths from **source** to **all vertices** in the given graph

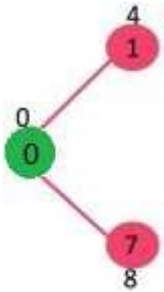
- **Dijkstra's** Algorithm is the best-known algorithm
- It is similar to Prim's algorithm
- This algorithm is applicable to undirected and directed graphs with nonnegative weights only.

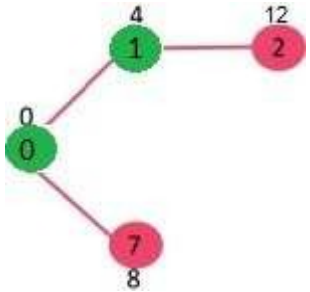
Dijkstra's Algorithm

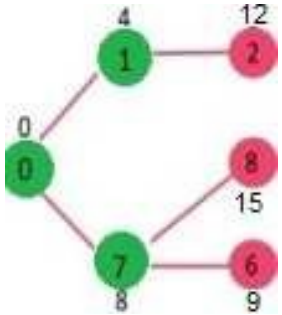
Working

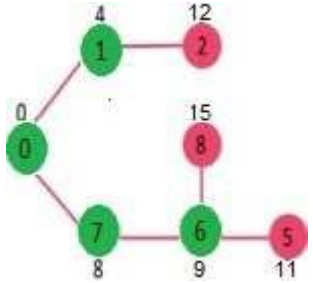
- First, it finds the shortest path from the source to a vertex nearest to it, then to a second nearest, and so on
- In general, before its i^{th} iteration commences, the algorithm has already identified the shortest paths to i-1 other vertices nearest to the source.

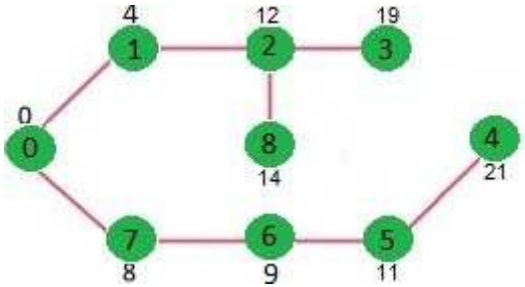

0
1
2
3
4
5
6
7
8
4
8
8
11
7
2
4
14
9
7
6
10
1
2


4
1
0
0
7
8

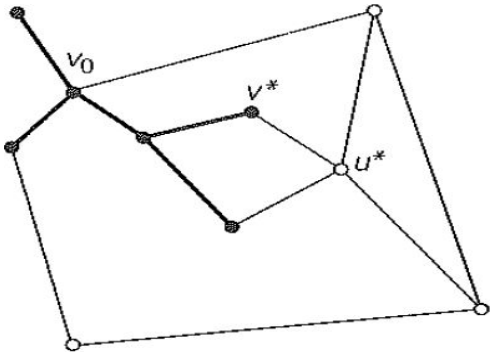

4
12
1
2
0
0
7
8


4
12
1
2
0
0
8
15
7
8
6
9


4
12
1
2
0
0
15
8
7
8
6
9
5
11

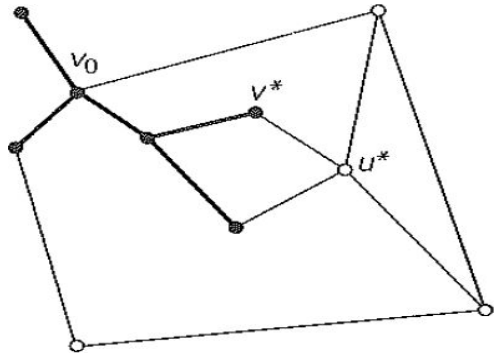

4
12
19
1
2
3
0
0
8
14
4
21
7
8
6
9
5
11

- The source, and the edges of the shortest paths leading to them from the source form a subtree T_i of the given graph shown in the figure.



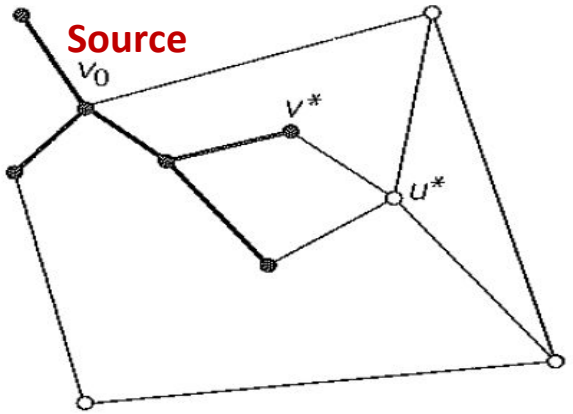
- The next vertex nearest to the source can be found among the vertices adjacent to the vertices of T_i. called as "fringe vertices";
- They are the candidates from which Dijkstra's algorithm selects the next vertex nearest to the source.

- To identify the i^{th} nearest vertex, the algorithm computes, for every fringe vertex *u,*
  - the sum of the distance to the nearest tree vertex ***v*** and the length *d (*shortest path from the source to *v)* (already computed !!!)
- then selects the vertex with the smallest such sum.



- To facilitate the algorithm's operations, we label each vertex with two labels.
  - The numeric label **d** indicates the length of the shortest path from the source to this vertex found by the algorithm so far
  - The other label indicates the parent of the vertex in the tree being constructed.

Vertex (parent, d)

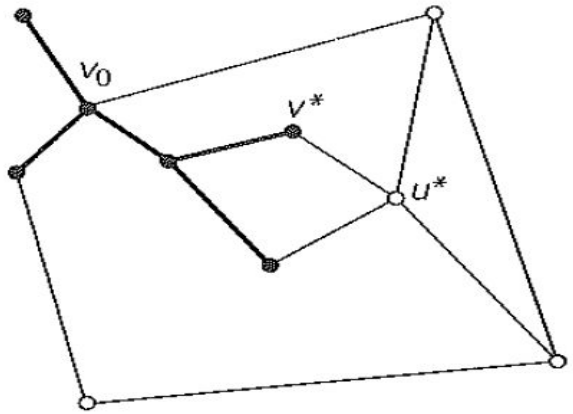


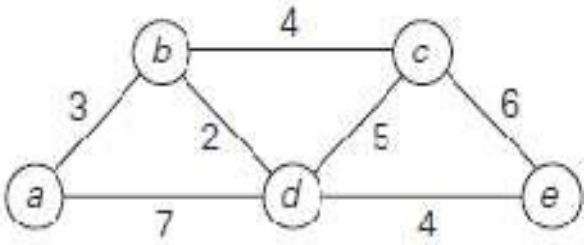
- With such labelling, finding the next nearest vertex u* becomes a simple task of finding a fringe vertex with the smallest d value.

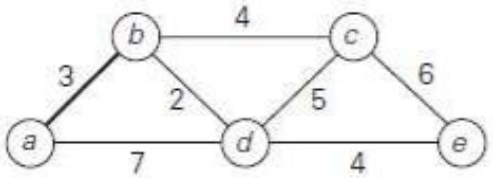
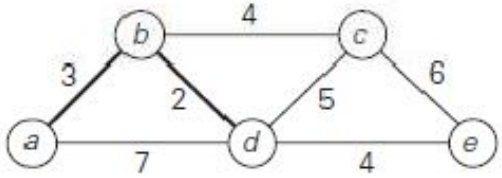
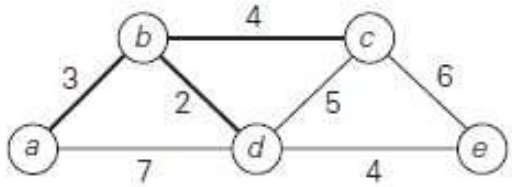
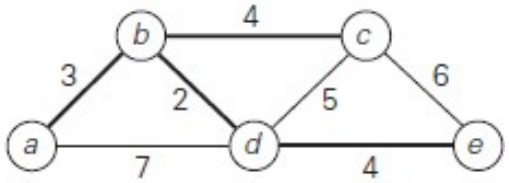
- After we have identified a vertex u* to be added to the tree, we need to perform two operations:
  1. Move *u** from the fringe to the set of tree vertices.
  2. For each remaining fringe vertex u that is connected to u* by an edge of weight w (u*, u) such that

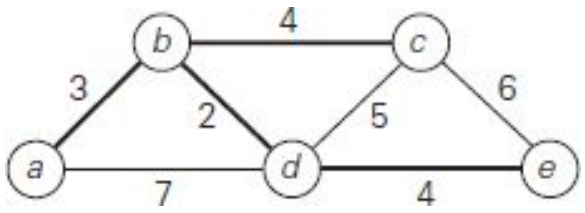
     $d_{u^*} + w(u^*, u) < d_u$,

     update the labels of u by u* and $d_{u^*} + w(u^*, u)$, respectively.





Tree vertices	Remaining vertices	Illustration
a(−, 0)	**b(a, 3)** c(−, ∞) d(a, 7) e(−, ∞)	
b(a, 3)	c(b, 3 + 4) **d(b, 3 + 2)** e(−, ∞)	
d(b, 5)	**c(b, 7)** e(d, 5 + 4)	
c(b, 7)	**e(d, 9)**	
e(d, 9)		



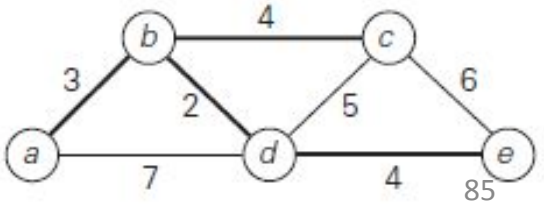
- The shortest paths (identified by following nonnumeric labels backward from a destination vertex in the left column to the source) and their lengths (given by numeric labels of the tree vertices) are as follows:

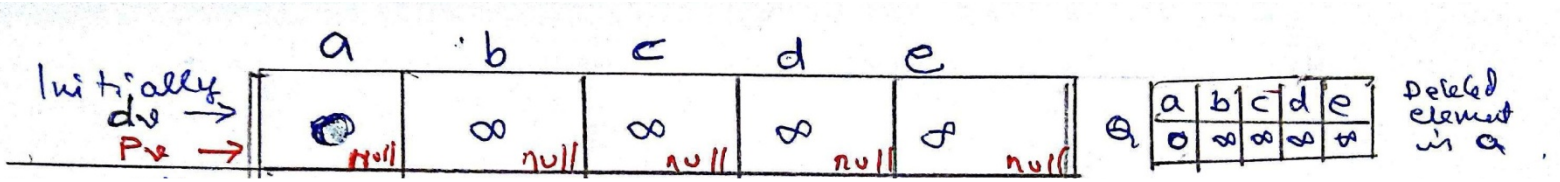
from a to b :	$a - b$	of length 3
from a to d :	$a - b - d$	of length 5
from a to c :	$a - b - c$	of length 7
from a to e :	$a - b - d - e$	of length 9

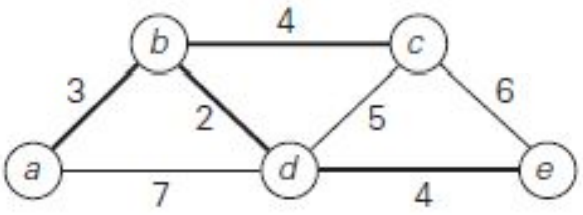
Dijkstra's Algorithm

ALGORITHM *Dijkstra*(G, s)

//Dijkstra's algorithm for single-source shortest paths
//Input: A weighted connected graph $G = \langle V, E \rangle$ with nonnegative weights
// and its vertex s
//Output: The length d_v of a shortest path from s to v
// and its penultimate vertex p_v for every vertex v in V
Initialize(Q) //initialize priority queue to empty
for every vertex v in V
  $d_v \leftarrow \infty$; $p_v \leftarrow$ **null**
  Insert(Q, v, d_v) //initialize vertex priority in the priority queue
$d_s \leftarrow 0$; *Decrease*(Q, s, d_s) //update priority of s with d_s
$V_T \leftarrow \varnothing$
for $i \leftarrow 0$ **to** $|V| - 1$ **do**
  $u^* \leftarrow$ *DeleteMin*(Q) //delete the minimum priority element
  $V_T \leftarrow V_T \cup \{u^*\}$
  for every vertex u in $V - V_T$ that is adjacent to u^* **do**
    if $d_{u^*} + w(u^*, u) < d_u$
      $d_u \leftarrow d_{u^*} + w(u^*, u)$; $p_u \leftarrow u^*$
      Decrease(Q, u, d_u)







...

for $i \leftarrow 0$ **to** $|V| - 1$ **do**

  $u^* \leftarrow DeleteMin(Q)$ //delete the minimum priority element

  $V_T \leftarrow V_T \cup \{u^*\}$

  for every vertex u in $V - V_T$ that is adjacent to u^* **do**

    if $d_{u^*} + w(u^*, u) < d_u$

      $d_u \leftarrow d_{u^*} + w(u^*, u)$; $p_u \leftarrow u^*$

      $Decrease(Q, u, d_u)$

	a	b	c	d	e
Initially d_v → P_v →	0 null	∞ null	∞ null	∞ null	∞ null
$u^*=a$ $V_T=\{a\}$	0 null	Min(0+3, ∞) 3 a	∞ null	Min(0+7, ∞) 7 a	∞ null
$u^*=b$ $V_T=\{a,b\}$	0 null	3 a	Min(3+4, ∞) 7 b	Min(3+2, 7) 5 b	∞ null
$u^*=d$ $V_T=\{a,b,d\}$	0 null	3 a	Min(5+5, 7) 7 b	5 b	Min(5+4, ∞) 9 d
$u^*=c$ $V_T=\{a,b,c,d\}$	0 null	3 a	7 b	5 b	Min(7+6, 9) 9 d
$u^*=e$ $V_T=\{a,b,c,d,e\}$	0 null	3 a	7 b	5 b	9 d

Q

a	b	c	d	e
0	∞	∞	∞	∞

Deleted element is a.

Q

b	c	d	e
3	∞	7	∞

Deleted element b

Q

c	d	e
7	5	∞

Deleted vertex d

Q

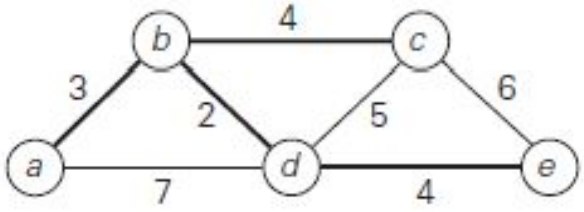
c	e
7	9

Deleted vertex c

Q

e
9

Deleted vertex e.



...

for $i \leftarrow 0$ **to** $|V| - 1$ **do**

  $u^* \leftarrow DeleteMin(Q)$ //delete the minimum priority element

  $V_T \leftarrow V_T \cup \{u^*\}$

  for every vertex u in $V - V_T$ that is adjacent to u^* **do**

    if $d_{u^*} + w(u^*, u) < d_u$

      $d_u \leftarrow d_{u^*} + w(u^*, u)$; $p_u \leftarrow u^*$

      $Decrease(Q, u, d_u)$

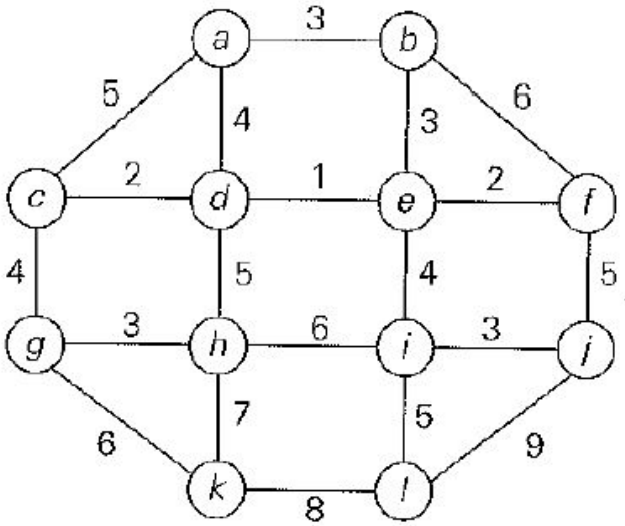
Analysis

- Efficiency is **$\Theta(|V|^2)$** for graphs represented by their ***weight matrix*** and the priority queue implemented as an ***unordered array***.

- For graphs represented by their **adjacency lists** and the priority queue implemented as a **min-heap**, it is in **O (|E| log |V|)**

Home Work

- Apply Dijkstra's Algorithm to the following graph. Consider a as source vertex



Module 3: Greedy Method - Outline

- Introduction to Greedy method
  - General method,
  - Coin Change Problem
  - Knapsack Problem
  - Job sequencing with deadlines
- Minimum cost spanning trees:
  - Prim's Algorithm,
  - Kruskal's Algorithm
- Single source shortest paths
  - Dijkstra's Algorithm
- Optimal Tree problem:
  - Huffman Trees and Codes
- Transform and Conquer Approach:
  - Heaps
  - Heap Sort

Optimal Tree Problem

Background

- Suppose we have to encode a text that comprises characters from some n-character alphabet
- Encode by assigning to each of the text's characters some sequence of bits called the **codeword.**
- There are two types of

  encoding: Fixed-length

  encoding, Variable-length

  encoding

Optimal Tree Problem

Fixed length Coding

- This method assigns to each character a bit string of the same length ***m***.
- Example: ASCII code.

- Drawback?
  - How to overcome the drawback?
  - assigning shorter code-words to more frequent characters and longer code-words to less frequent characters.

Optimal Tree Problem

Variable length Coding

- This method assigns code-words of different lengths to different characters
- Prefix-free codes (prefix codes) are used; Here no codeword is a prefix of a codeword of another character.
- We can simply scan a bit string until we get the first group of bits that is a codeword for some character

Optimal Tree Problem

Variable length Coding

- If we want to create a binary prefix code for some alphabet,
  - it is natural to associate the alphabet's characters with leaves of a binary tree in which all the left edges are labelled by 0 and all the right edges are labelled by 1 (or vice versa).
- Many trees that can be constructed in this manner for a given alphabet
- Example...

Optimal Tree Problem

- For known frequencies of the character occurrences,
- Construction of such a tree that would assign
  - shorter bit strings to high-frequency characters and
  - longer ones to low-frequency characters can be done by the greedy algorithm, invented by **David Huffman.**

Optimal Tree Problem

Variable length Coding

- This method assigns code-words of different lengths to different characters
- Prefix-free codes (prefix codes) are used; Here no codeword is a prefix of a codeword of another character.
- we can simply scan a bit string until we get the first group of bits that is a codeword for some character

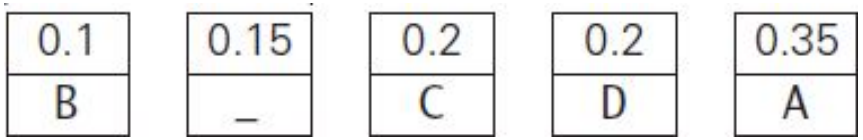
Huffman Trees and Codes

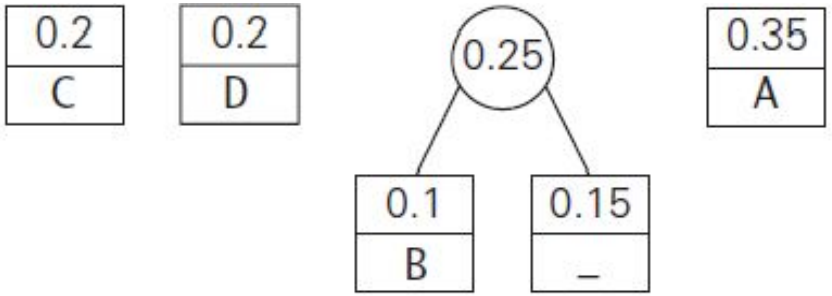
Huffman's Algorithm

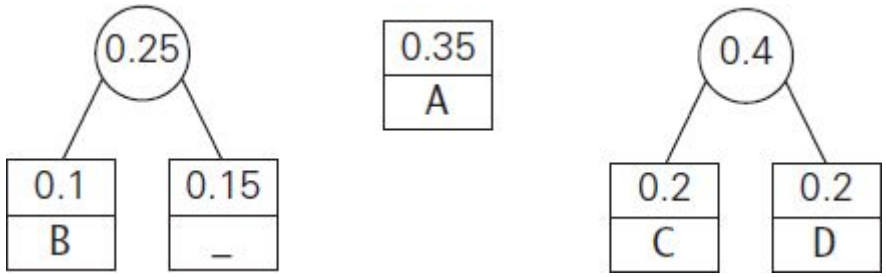
- ***Step 1:*** Initialize *n* one-node trees and label them with the characters of the alphabet. Record the frequency of each character in its tree's root to indicate the tree's *weight.*
- ***Step 2:*** Repeat the following operation until a single tree is obtained.
  - Find two trees with the smallest weight. Make them the left and right subtree of a new tree and record the sum of their weights in the root of the new tree as its weight.
- A tree constructed by the above algorithm is called **a *Huffman tree.*** It defines-in the manner described-a ***Huffman code.***

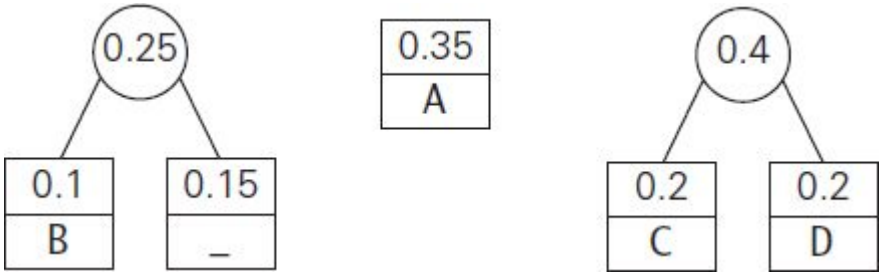
Example

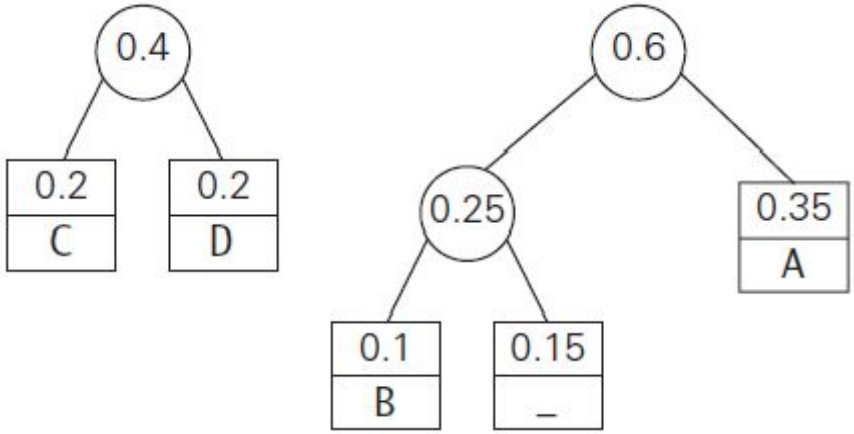
symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15

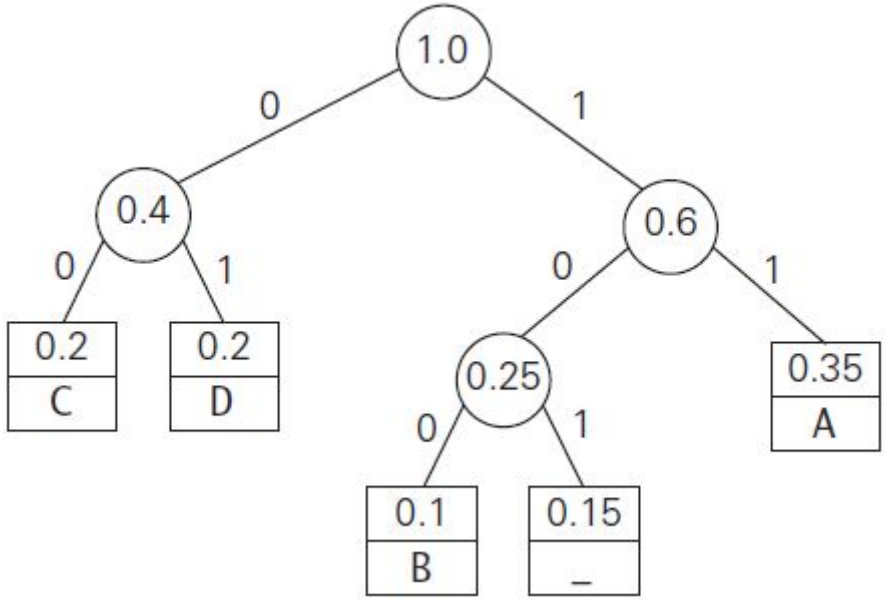






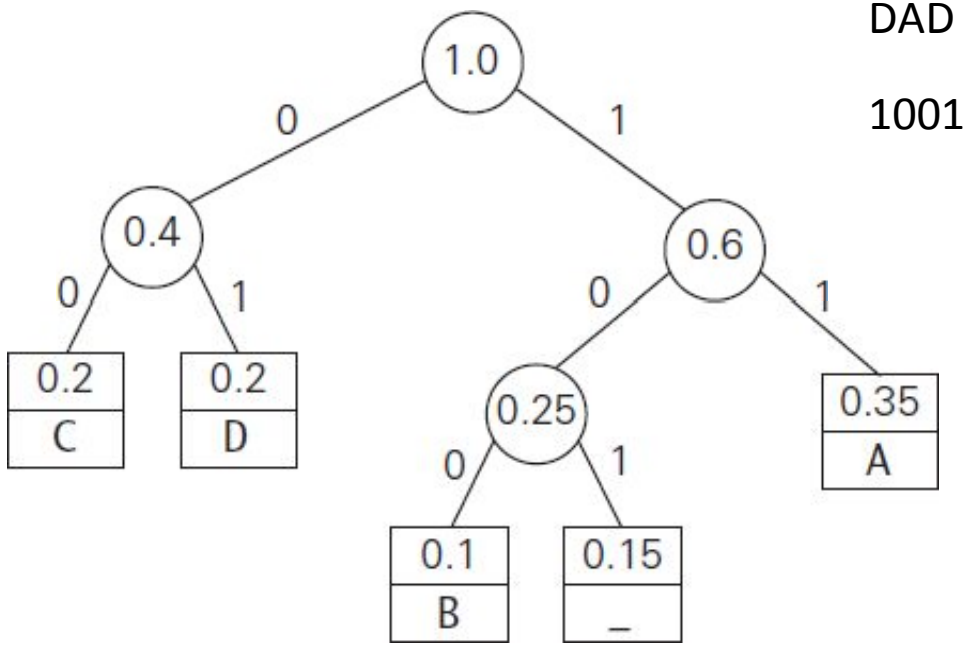

0.25
0.1
B
0.15
_
0.35
A
0.4
0.2
C
0.2
D


0.4
0.2
C
0.2
D
0.6
0.25
0.35
A
0.1
B
0.15
_


1.0
0
1
0.4
0.6
0
1
0
1
0.2
C
0.2
D
0.25
0.35
A
0
1
0.1
B
0.15
_

Huffman codes

- The resulting codewords are as follows:



DAD is encoded as

10011011011101 is decoded as

symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15
codeword	11	100	00	01	101

Analysis

- Average number of bits per symbol in this code

  2 * 0.35 + 3 * 0.1+ 2 * 0.2 + 2 * 0.2 + 3 * 0.15 = 2.25.

- Compression ratio (3−2.25)/3*100 = 25%

  In other words, Huffman's encoding of the above text will use 25% less memory than its fixed-length encoding.

Home work

- Represent the text “ABCDEBCDECDEDEE” using huffman coding.

Module 3: Greedy Method - Outline

- Introduction to Greedy method
  - General method,
  - Coin Change Problem
  - Knapsack Problem
  - Job sequencing with deadlines
- Minimum cost spanning trees:
  - Prim's Algorithm,
  - Kruskal's Algorithm
- Single source shortest paths
  - Dijkstra's Algorithm
- Optimal Tree problem:
  - Huffman Trees and Codes
- Transform and Conquer Approach:
  - Heaps
  - Heap Sort

Transform and Conquer

Two stages

1. The transformation stage - the problem's instance is modified to be, for one reason or another, more amenable to solution.
2. The conquering stage, - it is solved.

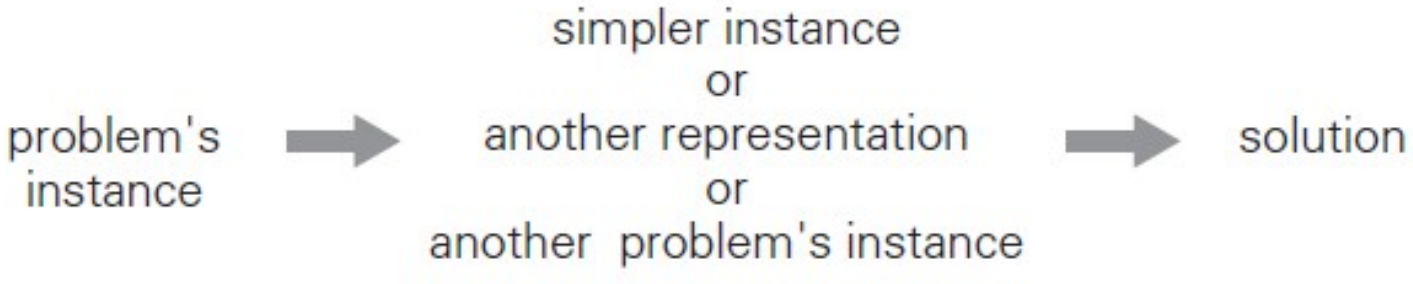


FIGURE 6.1 Transform-and-conquer strategy.

Heap

- **Heap** is a partially ordered data structure that is especially suitable for implementing priority queues.

- **Priority queue** is a multiset of items with an orderable characteristic called an item's **priority**, with the following operations:
  - finding an item with the highest priority
  - deleting an item with the highest priority
  - adding a new item to the multiset

Heap

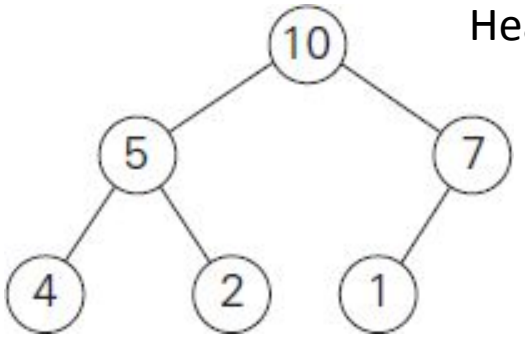
- **Heap** is a partially ordered data structure that is especially suitable for implementing priority queues.

- **Priority queue** is a multiset of items with an orderable characteristic called an item's **priority**, with the following operations:
  - finding an item with the highest priority
  - deleting an item with the highest priority
  - adding a new item to the multiset

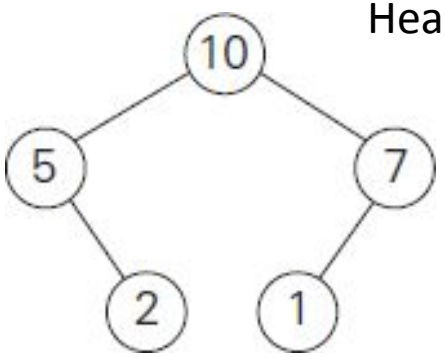
Heap

- Definition: A heap can be defined as a binary tree with keys assigned to its nodes, one key per node, provided the following two conditions are met:
  - The shape property —the binary tree is essentially complete (or simply complete), i.e., all its levels are full except possibly the last level, where only some rightmost leaves may be missing.
  - The parental dominance or heap property —the key in each node is greater than or equal to the keys in its children.

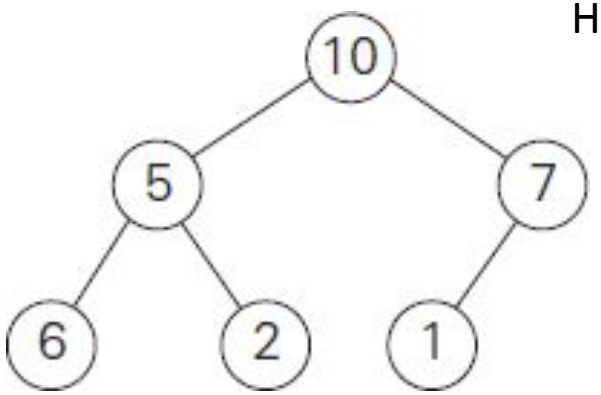
Heap?



Heap?



Heap?

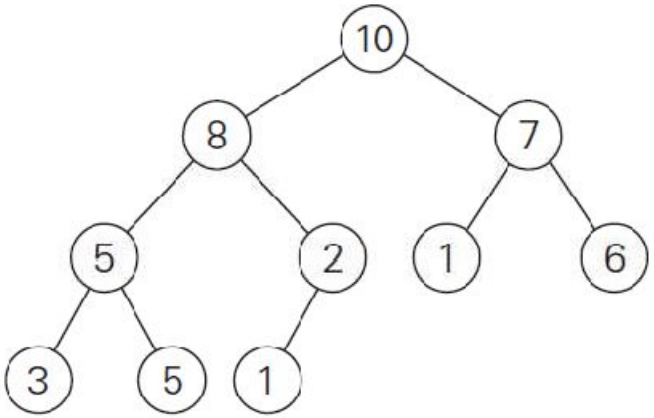


Properties of Heap

1. There exists exactly one essentially complete binary tree with n nodes. Its height is equal to $\lfloor \log_2 n \rfloor$
2. The root of a heap always contains its largest element.
3. A node of a heap considered with all its descendants is also a heap.

Properties of Heap

4. A heap can be implemented as an array by recording its elements in the top down, left-to-right fashion.



the array representation

index	0	1	2	3	4	5	6	7	8	9	10
value		10	8	7	5	2	1	6	3	5	1
		parents					leaves				

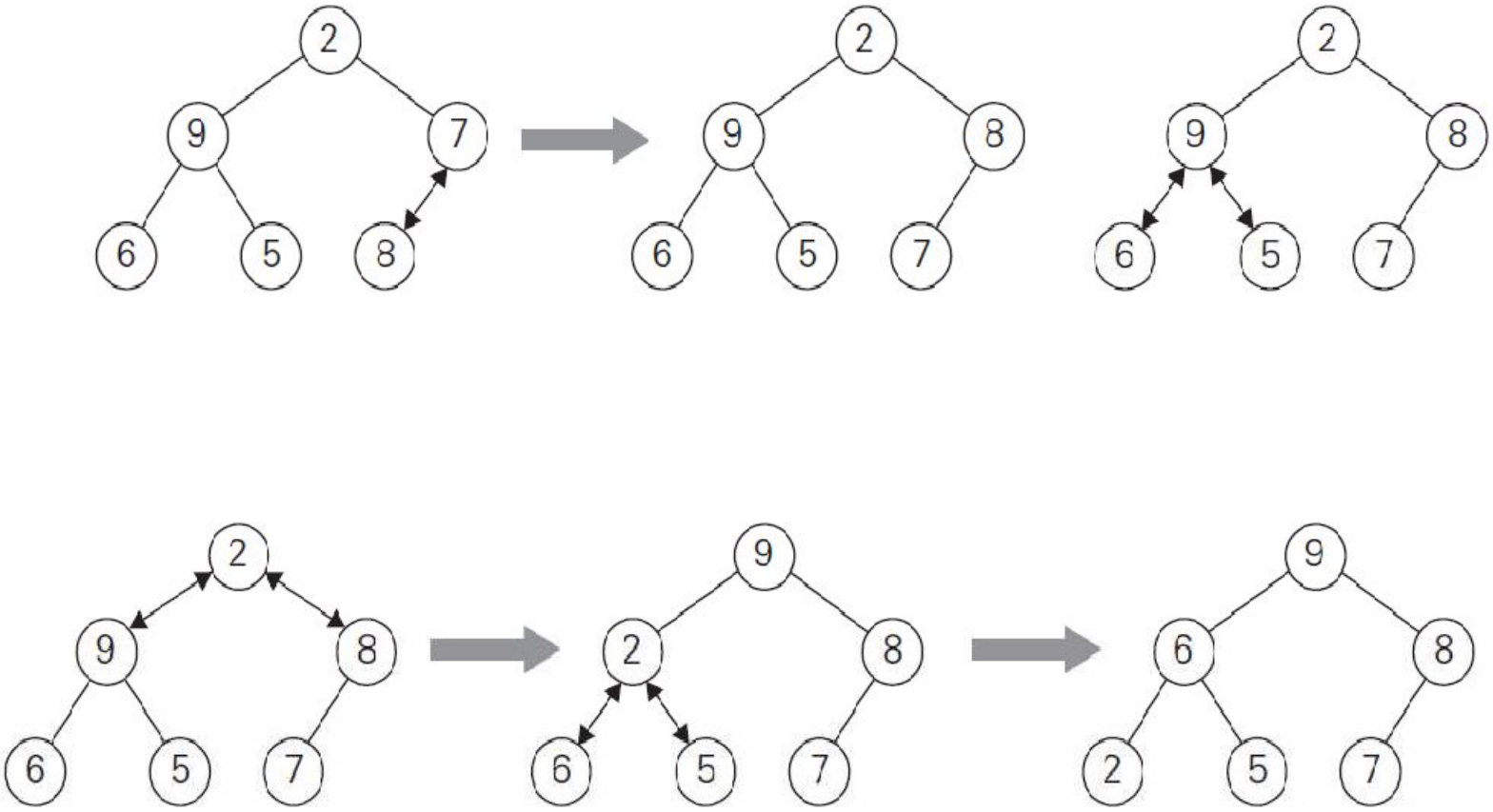
Construction of Heap

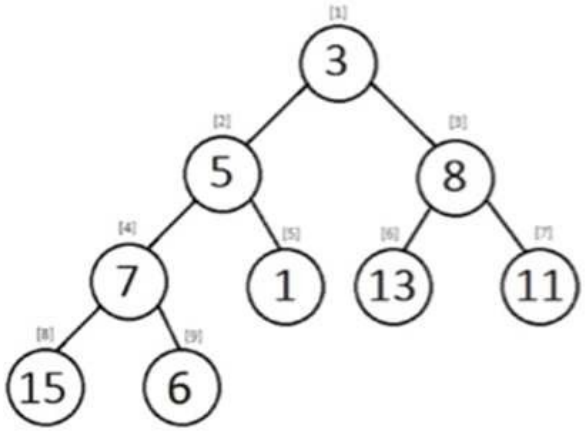
- There are two principal alternatives for constructing Heap.
  1. Bottom-up heap construction
  2. Top-down heap construction

Bottom up heap construction

1. Initialize the essentially complete binary tree with n nodes by placing keys in the order given.
2. Then "heapifies" the tree as follows.
    1. Starting with the last parental node, the check whether the parental dominance holds for the key
        - If it does not, exchange the node's key K with the larger key of its children and checks whether the parental dominance holds for K in its new position.
        - This process continues until the parental dominance for K is satisfied.
    2. After completing the "heapification" of the subtree rooted at the current parental node, proceed to do the same for the immediate predecessor.
    3. The algorithm stops after this is done for the root of the tree.

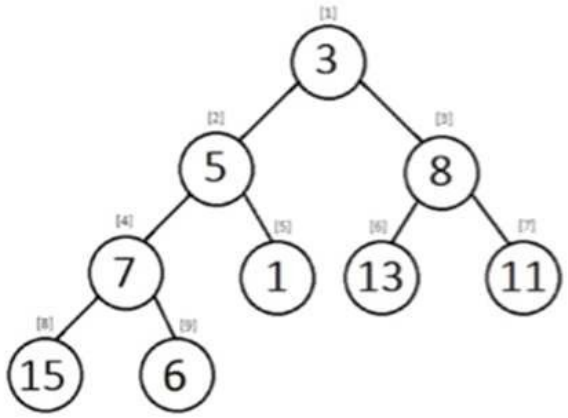
Example from textbook




[1]
3
[2]
5
[3]
8
[4]
7
[5]
1
[6]
13
[7]
11
[8]
15
[9]
6

ALGORITHM *HeapBottomUp*$(H[1..n])$

```
//Constructs a heap from elements of a given array
// by the bottom-up algorithm
//Input: An array H[1..n] of orderable items
//Output: A heap H[1..n]
for i ← ⌊n/2⌋ downto 1 do
    k ← i;   v ← H[k]
    heap ← false
    while not heap and 2 * k ≤ n do
        j ← 2 * k
        if j < n    //there are two children
            if H[j] < H[j + 1]  j ← j + 1
        if v ≥ H[j]
            heap ← true
        else H[k] ← H[j];    k ← j
    H[k] ← v
```



Analysis of Bottom up heap construction

- Assume, for simplicity, that $n = 2^k - 1$ => heap's tree is full.
- Consider Key on level **i** (level i , take the value level 0 to h ; height of the tree)
- For each key, moving to the next level down requires two comparisons
  - one to find the larger child
  - other to determine whether the exchange is required
- The total number of key comparisons involving a key on level i will be 2(h – i).

Analysis of Bottom up heap construction

- Therefore, the total number of key comparisons in the worst case will be

$$C_{worst}(n) = \sum_{i=0}^{h-1} \sum_{\text{level } i \text{ keys}} 2(h-i)$$

$$= \sum_{i=0}^{h-1} 2(h-i)2^i = 2(n - \log_2(n+1)),$$

By mathematical induction $i2^i$

- Thus, with this bottom-up algorithm, a heap of size *n* can be constructed with fewer than ***2n*** comparisons.

Top-down heap construction algorithm

- It constructs a heap by successive insertions of a new key into a previously constructed heap.

1. First, attach a new node with key *K* in it after the last leaf of the existing heap.
2. Then shift *K* up to its appropriate place in the new heap as follows.
   a. Compare *K* with its parent's key: if the latter is greater than or equal to *K,* stop (the structure is a heap);
   b. Otherwise, swap these two keys and compare *K* with its new parent.
   c. This swapping continues until *K* is not greater than its last p a r e n t root.

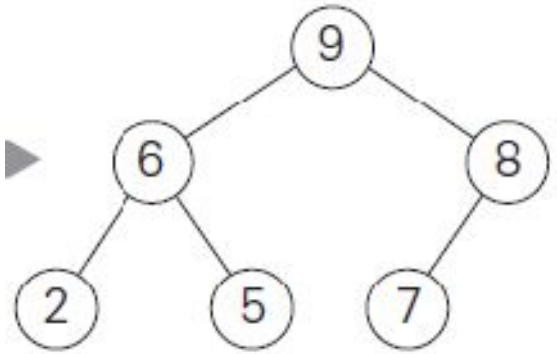
o r it r e a c h e s

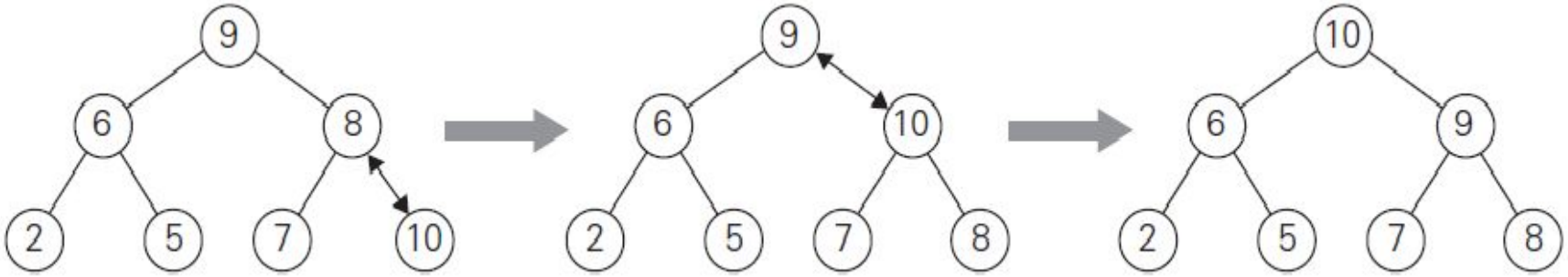
Illustration

- For elements 3, 5, 8, 7, 1, 13, 11, 15, 6 construct heap using top-down approach

Illustration -2

- Inserting a new key (10)





Delete an item from a heap

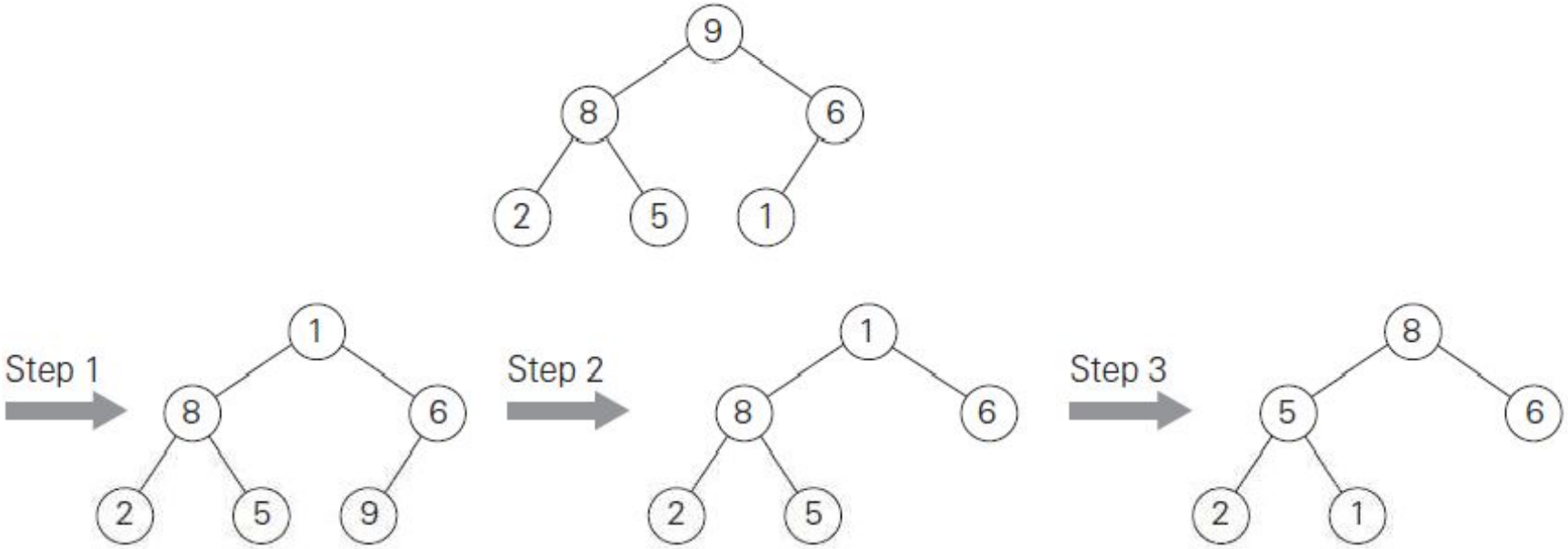
Maximum Key Deletion from a heap

1. Exchange the root's key with the last key *K* of the heap.
2. Decrease the heap's size by 1.
3. "Heapify" the smaller tree by sifting *K* down the tree exactly in the same way we did it in the bottom-up heap construction algorithm.
   That is, verify the parental dominance for

   K: if it holds, we are done;

   if not, swap *K* with the larger of its children and repeat this operation until the parental dominance condition holds for *K* in its new position.

Illustratio



Efficiency

- The efficiency of deletion is determined by
  - the number of key comparisons needed to "heapify" the tree after the swap has been made and the size of the tree is decreased by 1.
  - Since this cannot require more key comparisons than twice the heap's height, the time efficiency of deletion is in O (log n) as well.

Module 3: Greedy Method - Outline

- Introduction to Greedy method
  - General method,
  - Coin Change Problem
  - Knapsack Problem
  - Job sequencing with deadlines
- Minimum cost spanning trees:
  - Prim's Algorithm,
  - Kruskal's Algorithm
- Single source shortest paths
  - Dijkstra's Algorithm
- Optimal Tree problem:
  - Huffman Trees and Codes
- Transform and Conquer Approach:
  - Heaps
  - Heap Sort

Heap Sort

- This is a two-stage algorithm that works as follows.

  Stage 1 (heap construction): Construct a heap for a given array.

  Stage 2 (maximum deletions): Apply the root-deletion operation n−1 times to the remaining heap.

- As a result, the array elements are eliminated in decreasing order.
- Since under the array implementation of heaps an element being deleted is placed last, the resulting array will be exactly the original array sorted in increasing order.

Heap sort - Illustration

Stage 1 (heap construction)

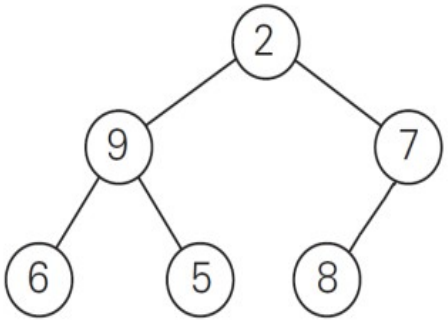
2 9 **7** 6 5 8

2 **9** 8 6 5 7

2 9 8 6 5 7

9 **2** 8 6 5 7

9 6 8 2 5 7



Stage 2 (maximum deletions)

9 6 8 2 5 7

7 6 8 2 5 | **9**

8 6 7 2 5

5 6 7 2 | **8**

7 6 5 2

2 6 5 | **7**

6 2 5

5 2 | **6**

5 2

2 | **5**

2

2, 5, 6, 7, 8, 9

Heapsort - Analysis of efficiency

- Stage-1: heap construction - O(n)
- Stage-2: For the number of key comparisons, C(n), needed for eliminating the root keys from the heaps of diminishing sizes from n to 2, we get the following inequality:

$$C(n) \le 2\lfloor \log_2 (n-1) \rfloor + 2\lfloor \log_2 (n-2) \rfloor + \cdots + 2\lfloor \log_2 1 \rfloor \le 2 \sum_{i=1}^{n-1} \log_2 i$$
$$\le 2 \sum_{i=1}^{n-1} \log_2 (n-1) = 2(n-1) \log_2 (n-1) \le 2n \log_2 n.$$

C(n) $\in$ O(n log n)

- For both stages, we get **O(n) + O(n log n) = O(n log n).**

Analysis

- A more detailed analysis shows that the time efficiency of heapsort is, in fact, in **Θ(n log n)** in both the **worst** and **average** cases.
- Thus, heapsort's time efficiency falls in the same class as that of mergesort.
- heapsort is **in-place**, i.e., it does not require any extra storage.
- Timing experiments on random files show that heapsort runs more slowly than quicksort but can be competitive with mergesort.

Study Tips

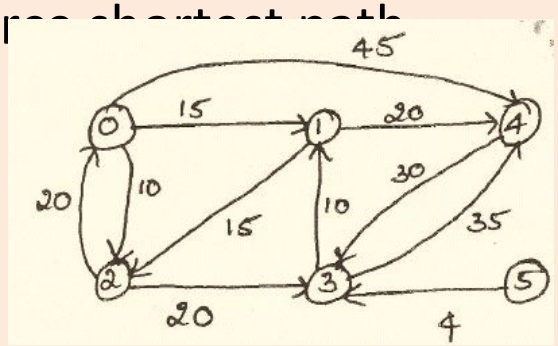
- For all topics study
  - Problem statement
  - Methodology, Algorithm
  - Applying algorithm to given example
  - Analysis
- Solve problems given in Review questions

Summary

- Introduction to Greedy method
  - General method,
  - Coin Change Problem
  - Knapsack Problem
  - Job sequencing with deadlines
- Minimum cost spanning trees:
  - Prim's Algorithm,
  - Kruskal's Algorithm
- Single source shortest paths
  - Dijkstra's Algorithm
    - Optimal Tree problem:
  - Huffman Trees and Codes
- Transform and Conquer Approach:
  - Heaps, Heap Sort

Assignment-3 Due: Within 5 days

1. Apply greedy method to obtain an optimal solution to the knapsack problem given M = 60, (w1, w2, w3, w4, w5) = (5, 10, 20, 30, 40), (p1, p2, p3, p4, p5) = (30, 20, 100, 90, 160). Find the total profit earned.
2. Write Kruskals algorithm to construct MST. Show that the time efficiency is O(|E|log|E|)
3. Apply Dijikstra's algorithm to find single source shortest path for the graph given below. Source vertex is



4. Sort the following lists by heapsort by using the array representation of heaps. 5, 2, 4, 1, 3 (in increasing order)

Extra Byte 3.1: Motu and Patlu

- ***Motu*** and ***Patlu*** are very good friends and they love to eat *ice- creams*. Once they thought of playing a game, so they bought **"n" ice-creams** from the market of *varying heights* (may be same). They arranged the ice-creams in a line in random order. *Motu* starts to eat ice-creams from left to right and *Patlu* from right to left. The heights of the ice-creams are known. ….

  https://www.hackerearth.com/practice/algorithms/greedy/basics- of-greedy-algorithms/practice-problems/algorithm/motu-and-patlu-1-ab612ad8/

Class test -3 Max Marks: 20 Duration: 45 Mins

End of Module-3