Birla Institute of Technology & Science, Pilani
Work-Integrated Learning Programmes Division
First Semester 2014-2015

Comprehensive Examination
(EC-3 Make-up)

Course No. : SS ZG519
Course Title : DATA STRUCTURES & ALGORITHMS DESIGN
Nature of Exam : Open Book — No. of Pages = 1
Weightage : 50% — No. of Questions = 4
Duration : 3 Hours
Date of Exam : 22/11/2014 (FN)

Note:
1. Please follow all the *Instructions to Candidates* given on the cover page of the answer book.
2. All parts of a question should be answered consecutively. Each answer should start from a fresh page.
3. Assumptions made if any, should be stated clearly at the beginning of your answer.

Q.1. Insert the elements A = {33, 22, 11, 43, 53, 63, 73, 96, 95, 94, 83, 82 } sequentially into an empty AVL tree. Write all intermediate steps clearly. [10]

Q.2. G is a an undirected graph with vertex set V(G) = { v1, v2,v3,v4,v5,v6,v7 } and edge Set E(G) = { v1v2, v1v3, v1v4, v2v4, v2v3, v3v4, v3v5, v4v5, v5v6, v5v7, v6v7}.

(a) Give the adjacency list representation of the graph.
(b) Is the given graph, a separable graph ? Justify your claim.
(c) Identify the cut vertices and the blocks of the graph, if they exist.
(d) Apply depth first search algorithm to find the depth first search tree of the graph G.

Write all intermediate steps clearly and write the stack, active vertex at every iteration. Also, make a table of depth first search index of the vertices. [3 + 3 + 6 +10 = 22]

Q.3. Find the minimum spanning tree of the weighted graph Gstarting at vertex ausing Prim's algorithm and find the weight of the minimum spanning tree. [8]
Write all intermediate steps clearly.
Vertex set V(G) = {a, b, c, d, e, f, g, h},
Edge Set E(G) = {ab, ac, bc, bd, cd, cf, de, ef, fg, eg, dg, gh, fh}
Weights = { ab = 2, ac = 4, bc = 1, bd = 3, cd = 5, cf = 4, de = 7, ef = 6, fg = 3, eg = 8, dg = 9, gh = 10 , fh = 12 }

Q.4. G is an undirected graph with vertex set V(G) = {a, b, c, d, e, f, g, h} edge Set E(G) = {ab, ac, bc, bd, cd, cf, de, ef, fg, eg, dg, gf, fh}

(a) Is the graph an Euler graph? Justify your answer.
(b) If the graph is an Euler graph, find the Euler circuit using Fleury's algorithm.
(c) Find a Hamiltonian circuit in the graph. [3 + 4 + 3 = 10]

****

Birla Institute of Technology & Sciences, Pilani
Work-Integrated Learning Programmes Division
First Semester 2014-2015

Comprehensive Examination
(EC-3 Regular)

Course No. : SS ZG519
Course Title : DATA STRUCTURES & ALGORITHMS DESIGN
Nature of Exam : Open Book — No. of Pages = 1
Weightage : 50% — No. of Questions = 5
Duration : 3 Hours
Date of Exam : 08/11/2014 (FN)

Note:
1. Please follow all the *Instructions to Candidates* given on the cover page of the answer book.
2. All parts of a question should be answered consecutively. Each answer should start from a fresh page.
3. Assumptions made if any, should be stated clearly at the beginning of your answer.

Q.1. Illustrate merge sort to sort the array A = {54, 65, 9, 72, 81, 16, 23, 37, 49 } in ascending order. Write all intermediate steps clearly. [8]

Q.2. Build a max heap out of the array A = {8, 20, 9, 4, 15, 10, 7, 22, 3, 12}. Write all intermediate steps clearly. [10]

Q.3. Construct a Huffman tree and a Huffman code for the following data. Clearly show the intermediate steps in the construction of the Huffman tree. [10]

Character	A	B	C	D	E	F
Frequency	0.28	0.21	0.67	0.4	0.22	0.27

Q.4. G is a an undirected graph with vertex set V(G) = { v1, v2,v3,v4,v5,v6,v7 } and edge Set E(G) = { v1v2, v1v3, v1v4, v2v4, v2v3, v3v4, v3v5, v4v5, v5v6, v5v7, v6v7}.
(a) Draw the graph with the given vertex set and edge set.
(b) Apply breadth first search algorithm to find the breadth first search tree of the graph G. Write all intermediate steps clearly and write the queue at every iteration. [2 + 10 = 12]

Q.5. Find the minimum spanning tree of the weighted graph G using Kruskal's algorithm and find the weight of the minimum spanning tree. Write all intermediate steps clearly. [10]
Vertex set V(G) = {a, b, c, d, e, f, g}
Edge Set E(G) = {ab, ac, bc, bd, cd, cf, de, ef, fg, eg, dg}
Weights = { ab = 2, ac = 4, bc = 1, bd = 3, cd = 5, cf = 4, de = 7, ef = 6, fg = 3, eg = 8, dg = 9}

****

innovate achieve lead

What is a program?

- **Algorithm**

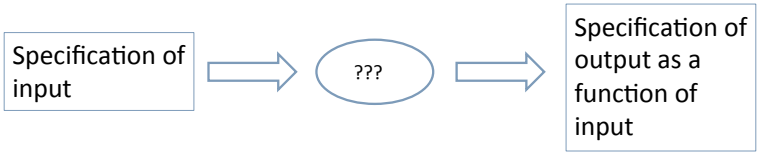
  An algorithm is a step-by-step procedure for solving a problem in a finite amount of time.

- **Data Structures**

  Is a systematic way of organizing and accessing data, so that data can be used efficiently.

  Algorithms + Data Structures = Program

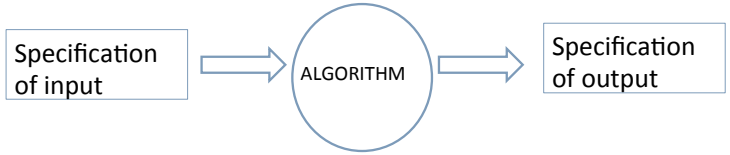
Algorithmic problem



For eg: Sorting of integers

Input Instance	:	8,4 ,5,2,10
Output Instance as a permutation of input	:	2,4,5,8,10

Algorithmic Solution



- Algorithm describes actions on the input instance.
- Infinitely many correct algorithm for the same problem.

Infinite number of input instances satisfying the specification.

Two key points: Repeatable argument & Correctness

What is good algorithm?

- Resources Used
  - Running time
  - Space used
- Resource Usage
  - Measured proportional to (input) size

Measuring the running time

- Write a program implementing the algorithm.
- Run the program with inputs of varying size and composition.
- Use a method like System.currentTimeMillis() to get an accurate measure of the actual running time.

Limitations of experimental studies

- Implementation is a must.
- Execution is possible on limited set of inputs.
- If we need to compare two algorithms we need to use the same environment
  (like hardware, software etc)

Analytical model to analyze algorithm

- Algorithm should be analyzed by using general methodology.
- This approach uses:
  - High level description of the algorithm.
  - Takes into account all possible inputs.
  - Allows one to evaluate the efficiency of any algorithm in a way that is independent of the hardware and the software environment.

Pseudo-code

- A mixture of natural language and high level programming concepts that describes the main ideas behind a generic implementation of a data structure and algorithms.

Algorithm ***arrayMax(A, n)***
Input: **An array *A of n integers***
Output: **The maximum element of *A***
currentMax ←A[0]
for *i ← 1 to n - 1 do*
if *A[i] > currentMax then currentMax ← A[i]*
return *currentMax*

Pseudo-code

- Is structured than usual prose but less formal than a programming language.
- Expressions
  - Use standard mathematical symbols to describe numeric and Boolean expressions.
  - Uses ← for assignment.
  - Use = for the equality relationship.
- Method declaration
  - Algorithm name(param1,param2…)

Assumptions

Individual statement considered as "unit" time

- Not applicable for function calls and loops

Individual variable considered as "unit" storage

Often referred to as "algorithmic complexity"

Complexity Example [1]

Example 1 (Y and Z are input)

```
X = Y * Z;
X = Y * X + Z;
// 2 units of time and 1 unit of storage
// Constant Unit of time and Constant Unit of
   storage
```


Complexity Example [2]

Example 2 (a and N are input)

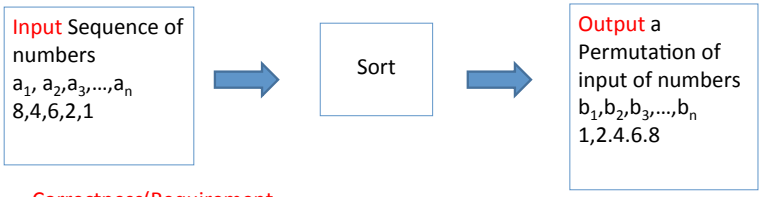
```
j = 0;
while (j < N) do
   a[j] = a[j] * a[j];
   b[j] = a[j] + j;
   j = j + 1;
endwhile;
// 3N + 1 units of time and N+1 units of storage
// time units prop. to N and storage prop. to N
```


Complexity Example [3]

Example 3 (a and N are input)

```
j = 0;
while (j < N) do
   k = 0;
   while (k < N) do
        a[k] = a[j] + a[k];
      k = k + 1;
   endwhile;
   b[j] = a[j] + j;
   j = j + 1;
endwhile;
//??? units of time and ??? units of storage
// time prop. to N^2 and  storage prop. to N
```

Example of sorting



Correctness(Requirement for the output)
For any input algorithm halts with the output:

- $b_1 < b_2 < b_3 < \ldots < b_n$
- $b_1, b_2, b_3, \ldots, b_n$ is a permutation of $a_1, a_2, a_3, \ldots, a_n$

Running time of algorithm depends on

- Number of elements n.
- How (partially)sorted they are.

Order Notation

- Purpose
  - Capture proportionality
  - Machine independent measurement
  - Asymptotic growth
    (i.e. large values of input size N)

Motivation for Order Notation

Examples

- $100 * \log_2 N < N$ for $N > 1000$
- $70 * N + 3000 < N^2$ for $N > 100$
- $10^5 * N^2 + 10^6 * N < 2^N$ for $N > 26$

Asymptotic Analysis

- Goal: To simplify analysis of running time of algorithm .eg $3n^2 = n^2$.
- Capturing the essence: how the running time of the algorithm increases with the size of the input in the limit.

Asymptotic Notation

- The big O notation

  Definition

  Let f and g be functions from the set of integers to the set of real numbers. We say that $f(x)$ is in $O(g(x))$ if there are constants $\boldsymbol{C}$ > and $\boldsymbol{k}$ such that $\boldsymbol{|f(x)| \leq C\,|g(x)|}$, whenever $\boldsymbol{x} >= \boldsymbol{k}$.

- This is read as $f(x)$ is ***big-oh*** of $g(x)$

Note: Pair of C and k is never unique.

Order Notation

Examples

g(n) = 17*N + 5

$\lim_{n\to\infty}$ g(n) / f(n) = c
$\lim_{n\to\infty}$ (17*N +5)/N = 17. The asymptotic complexity is O(N)

g(n) = $5*N^3 + 10*N^2 + 3$
$\lim_{n\to\infty}$ ($5*N^3 + 10*N^2 + 3$) / N^3 = 5. The asymptotic complexity is $O(N^3)$

g(n) = $C1*N^k + C2*N^{k-1} + \ldots + Ck*N + C$
$\lim_{n\to\infty}$ ($C1*N^k + C2*N^{k-1} + \ldots + Ck*N + C$) / N^k = C1.
The asymptotic complexity is $O(N^k)$

$2^N + 4*N^3 + 16$ is $O(2^N)$

5*N*log(N) + 3*N is O(N*log(N))

1789 is O(1)

Linear Search

```
function search(X, A, N)
j = 0;
while (j < N)
      if (A[j] == X)  return j;
     j++;
 endwhile;
return "Not-found";
```


Linear Search - Complexity

Time Complexity

"if" statement introduces possibilities

- Best-case: O(1)
- Worst case: O(N)
- Average case: ???

Binary Search Algorithm

Assume: Sorted Sequence of numbers

```
low = 1;  high = N;
while (low <= high) do
   mid = (low + high) /2;
   if (A[mid] = = x)  return x;
   else if (A[mid] < x) low = mid +1;
   else high = mid – 1;
endwhile;
   return Not-Found;
```

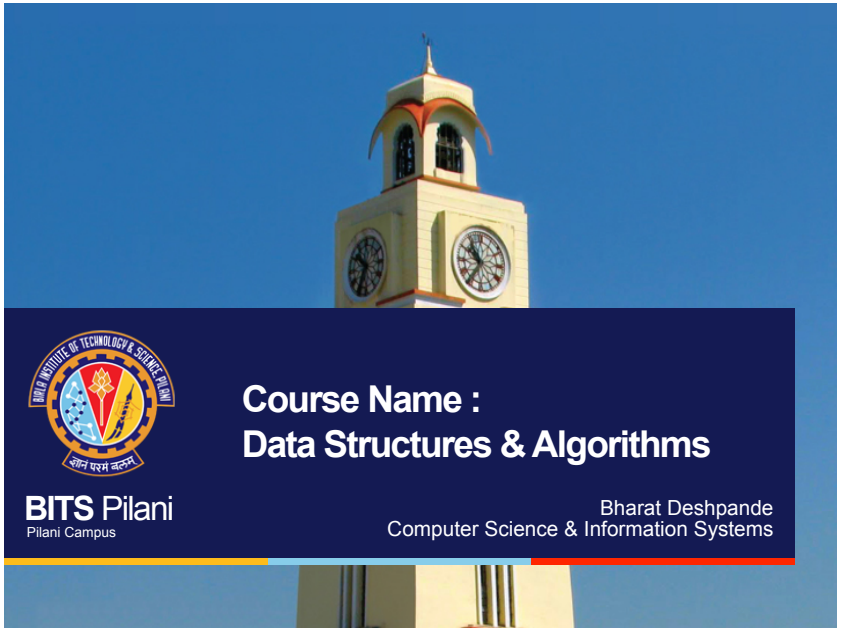

Binary Search - Complexity

- Best Case
  - O(1)
- Worst case:
  - Loop executes until low <= high
  - Size halved in each iteration
  - N, N/2, N/4, … 1
  - How many steps ?

Binary Search - Complexity

- Worst case:
  - K steps such that $2^K = N$

  i.e. $\log_2 N$ steps is O(log(N))



Assumptions

Individual statement considered as “unit” time

- Not applicable for function calls and loops

Individual variable considered as “unit” storage

Often referred to as “algorithmic complexity”

Complexity Example [1]

Example 1 (Y and Z are input)

```
X = Y * Z;
X = Y * X + Z;
// 2 units of time and 1 unit of storage
// Constant Unit of time and Constant Unit of storage
```


Complexity Example [2]

Example 2 (a and N are input)

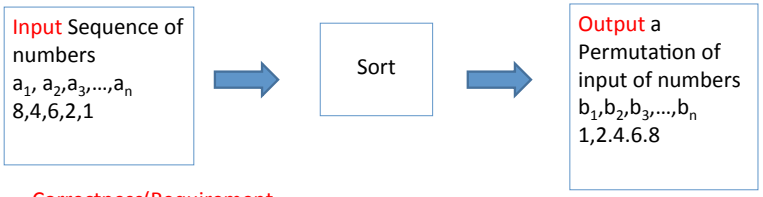
```
j = 0;
while (j < N) do
   a[j] = a[j] * a[j];
    b[j] = a[j] + j;
    j = j + 1;
endwhile;
// 3N + 1 units of time and N+1 units of storage
// time units prop. to N and storage prop. to N
```


Complexity Example [3]

Example 3 (a and N are input)

```
j = 0;
while (j < N) do
    k = 0;
    while (k < N) do
        a[k] = a[j] + a[k];
        k = k + 1;
     endwhile;
    b[j] = a[j] + j;
    j = j + 1;
endwhile;
//??? units of time and ??? units of storage
// time prop. to N^2 and  storage prop. to N
```

Example of sorting



Correctness(Requirement for the output)
For any input algorithm halts with the output:

- $b_1 < b_2 < b_3 < \ldots < b_n$
- $b_1, b_2, b_3, \ldots, b_n$ is a permutation of $a_1, a_2, a_3, \ldots, a_n$

Running time of algorithm depends on

- Number of elements n.
- How (partially)sorted they are.

Order Notation

- Purpose
  - Capture proportionality
  - Machine independent measurement
  - Asymptotic growth
    (i.e. large values of input size N)

Motivation for Order Notation

Examples

- $100 * \log_2 N < N$ for $N > 1000$
- $70 * N + 3000 < N^2$ for $N > 100$
- $10^5 * N^2 + 10^6 * N < 2^N$ for $N > 26$

Asymptotic Analysis

- Goal: To simplify analysis of running time of algorithm .eg $3n^2 = n^2$.
- Capturing the essence: how the running time of the algorithm increases with the size of the input in the limit.

Asymptotic Notation

- The big O notation

  Definition

  Let f and g be functions from the set of integers to the set of real numbers. We say that $f(x)$ is in $O(g(x))$ if there are constants $\boldsymbol{C}$ > and $\boldsymbol{k}$ such that $|\boldsymbol{f(x)}| \leq \boldsymbol{C}\,|\boldsymbol{g(x)}|$, **w**henever $\boldsymbol{x} >= \boldsymbol{k}$.
- This is read as $f(x)$ is ***big-oh*** of $g(x)$

Note: Pair of C and k is never unique.

Order Notation

Examples

g(n) = 17*N + 5

$\lim_{n\to\infty}$ g(n) / f(n) = c
$\lim_{n\to\infty}$ (17*N +5)/N = 17. The asymptotic complexity is O(N)

g(n) = $5*N^3 + 10*N^2 + 3$
$\lim_{n\to\infty}$ $(5*N^3 + 10*N^2 + 3) / N^3 = 5$. The asymptotic complexity is $O(N^3)$

g(n) = $C1*N^k + C2*N^{k-1} + \ldots + Ck*N + C$
$\lim_{n\to\infty}$ $(C1*N^k + C2*N^{k-1} + \ldots + Ck*N + C) / N^k = C1$.
The asymptotic complexity is $O(N^k)$

$2^N + 4*N^3 + 16$ is $O(2^N)$

5*N*log(N) + 3*N is O(N*log(N))

1789 is O(1)

Linear Search

```
function search(X, A, N)
j = 0;
while (j < N)
    if (A[j] == X)  return j;
     j++;
 endwhile;
return "Not-found";
```


Linear Search - Complexity

Time Complexity

"if" statement introduces possibilities

– Best-case: O(1)
– Worst case: O(N)
– Average case: ???

Binary Search Algorithm

Assume: Sorted Sequence of numbers

```
low = 1;  high = N;
while (low <= high) do
   mid = (low + high) /2;
   if (A[mid] = = x)  return x;
   else if (A[mid] < x) low = mid +1;
   else high = mid – 1;
endwhile;
   return Not-Found;
```


Binary Search - Complexity

- Best Case
  - O(1)
- Worst case:
  - Loop executes until low <= high
  - Size halved in each iteration
  - N, N/2, N/4, … 1
  - How many steps ?

Binary Search - Complexity

- Worst case:
  - K steps such that $2^K = N$

  i.e. $\log_2 N$ steps is O(log(N))

Abstract Data Type

- Abstract Data Type

In computer science, an **abstract data type** (**ADT**) is a mathematical model for a certain class of data structures that have similar behavior.

Abstract Data Types (ADTs)

- A method for achieving abstraction for data structures and algorithms
- ADT = model + operations
- Describes what each operation does, but not how it does it
- An ADT is independent of its implementation

Abstract Data Types

- Typical operations on data
  - Add data to a data collection
  - Remove data from a data collection
  - Ask questions about the data in a data collection

Abstract Data Types

- Data abstraction
  - Asks you to think *what* you can do to a collection of data independently of *how* you do it
  - Allows you to develop each data structure in relative isolation from the rest of the solution
  - A natural extension of procedural abstraction

Examples

- Simple ADTs
  - *Stack*
  - *Queue*
  - Vector
  - Lists
  - Sequences
  - Iterators

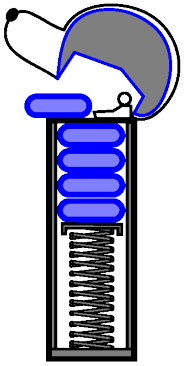
  All these are called Linear Data Structures

Stacks

- A **stack** is a container of objects that are inserted and removed according to the last-in-first-out (**LIFO**) principle.
- Objects can be inserted at any time, but only the last (the most-recently inserted) object can be removed.
- Inserting an item is known as "**pushing**" onto the stack. "**Popping**" off the stack is synonymous with removing an item.

Stacks

- A coin dispenser as an analogy:



Stacks: An Array Implementation

- Create a stack using an array by specifying a maximum size *N* for our stack.
- The stack consists of an N-element array *S* and an integer variable *t*, the index of the top element in array S.

- Array indices start at 0, so we initialize *t* to -1

Stacks: An Array Implementation

- **Pseudo code**

```
Algorithm size()
return t+1

Algorithm isEmpty()
return (t<0)

Algorithm top()
if isEmpty() then
   return Error
return S[t]
```

```
Algorithm push(o)
if size()==N then
   return Error
t=t+1
S[t]=o

Algorithm pop()
 if isEmpty() then
   return Error
 t=t-1

return S[t+1]
```

Stacks: An Array Implementation

The array implementation is simple and efficient (methods performed in O(1)).

Disadvantage

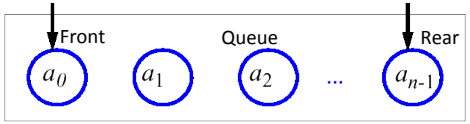
There is an upper bound, *N*, on the size of the stack.

The arbitrary value *N* may be too small for a given application OR

a waste of memory.

Queues

- A queue differs from a stack in that its insertion and removal routines follows the **first-in-first-out** (FIFO) principle.
- Elements may be inserted at any time, but only the element which has been in the queue the longest may be removed.
- Elements are inserted at the **rear** (enqueued) and removed from the **front** (dequeued)



Queues

- The **queue** supports three fundamental methods:
  - **New():***ADT – Creates an empty queue*
  - **Enqueue(S:***ADT,* **o:***element***):***ADT* - Inserts object *o* at the rear of the queue
  - **Dequeue(S:***ADT***):***ADT* - Removes the object from the front of the queue; an error occurs if the queue is empty
  - **Front(S:***ADT***):***element* - Returns, but does not remove, the front element; an error occurs if the queue is empty

Queues: An Array Implementation

- Create a queue using an array in a circular fashion
- A maximum size *N* is specified.
- The queue consists of an N-element array *Q* and two integer variables:
  - *f*, index of the front element (head – for dequeue)
  - *r*, index of the element after the rear one (tail – for enqueue)
  - Initially, f=r=0 and the queue is empty if f=r

Queues

Disadvantage

Repeatedly enqueue and dequeue a single element N times.

Finally, f=r=N.

- No more elements can be added to the queue,

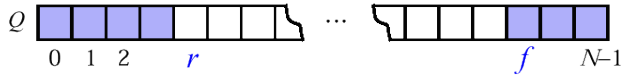
though there is space in the queue.

Solution

Let f and r wraparound the end of queue.

Queues: An Array Implementation

"wrapped around" configuration



- Each time r or f is incremented, compute this increment as (r+1)modN or (f+1)modN

Queues: An Array Implementation

- Pseudo code

```
Algorithm size()
return (N-f+r) mod N

Algorithm isEmpty()
return (f=r)

Algorithm front()
if isEmpty() then
   return Error
return Q[f]
```

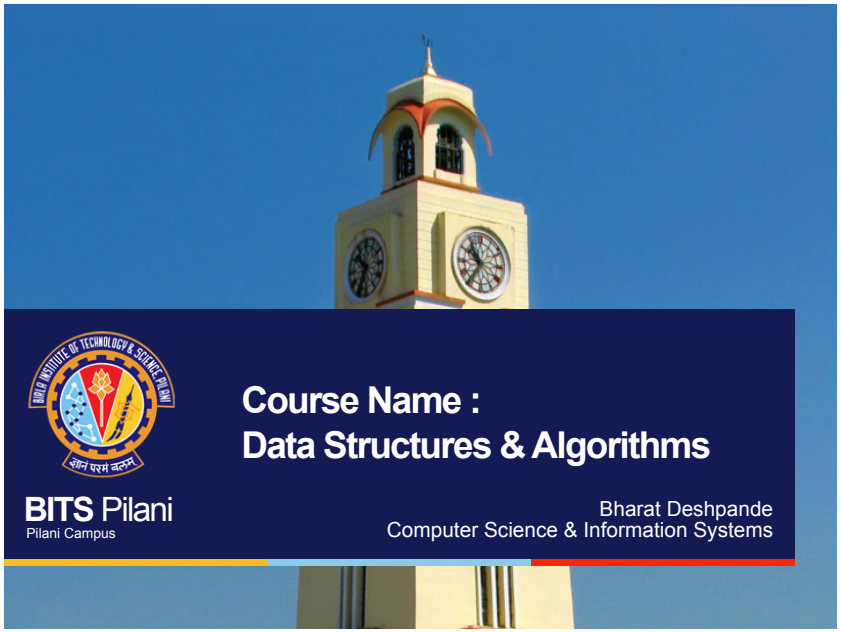
```
Algorithm dequeue()
if isEmpty() then
   return Error
Q[f]=null
f=(f+1)modN

Algorithm enqueue(o)
if size = N - 1 then
   return Error
Q[r]=o
r=(r +1)modN
```


Arrays: pluses and minuses

+ Fast element access.

-- Impossible to resize.

- Many applications require resizing!
- Required size not always immediately available.



Order Notation

- Purpose
  - Capture proportionality
  - Machine independent measurement
  - Asymptotic growth
    (i.e. large values of input size N)

Asymptotic Analysis

- Goal: To simplify analysis of running time of algorithm .eg $3n^2=n^2$.
- Capturing the essence: how the running time of the algorithm increases with the size of the input in the limit.

Asymptotic Notation

- The big O notation

  Definition

  Let f and g be functions from the set of integers to the set of real numbers. We say that $f(x)$ is in $O(g(x))$ if there are constants $\boldsymbol{C}>$ and $\boldsymbol{k}$ such that $|\boldsymbol{f(x)}| \leq \boldsymbol{C}\,|\boldsymbol{g(x)}|$, **w**henever $\boldsymbol{x} >= \boldsymbol{k}$.
- This is read as $f(x)$ is ***big-oh*** of $g(x)$

Note: Pair of C and k is never unique.

Order Notation

Examples

g(n) = 17*N + 5

$\lim_{n\to\infty}$ g(n) / f(n) = c
$\lim_{n\to\infty}$ (17*N +5)/N = 17. The asymptotic complexity is O(N)

g(n) = $5*N^3 + 10*N^2 + 3$
$\lim_{n\to\infty}$ $(5*N^3 + 10*N^2 + 3) / N^3 = 5$. The asymptotic complexity is $O(N^3)$

g(n) = $C1*N^k + C2*N^{k-1} + \ldots + Ck*N + C$
$\lim_{n\to\infty}$ $(C1*N^k + C2*N^{k-1} + \ldots + Ck*N + C) / N^k = C1$.
The asymptotic complexity is $O(N^k)$

$2^N + 4*N^3 + 16$ is $O(2^N)$

5*N*log(N) + 3*N is O(N*log(N))

1789 is O(1)

Linear Search

```
function search(X, A, N)
j = 0;
while (j < N)
     if (A[j] == X)  return j;
      j++;
 endwhile;
return “Not-found”;
```


Linear Search - Complexity

Time Complexity

“if” statement introduces possibilities

– Best-case: O(1)
– Worst case: O(N)
– Average case: ???

Binary Search Algorithm

Assume: Sorted Sequence of numbers

```
low = 1;  high = N;
while (low <= high) do
   mid = (low + high) /2;
   if (A[mid] = = x)  return x;
   else if (A[mid] < x) low = mid +1;
   else high = mid – 1;
endwhile;
   return Not-Found;
```

Binary Search - Complexity

- Best Case
  - O(1)
- Worst case:
  - Loop executes until low <= high
  - Size halved in each iteration
  - N, N/2, N/4, … 1
  - How many steps ?

Binary Search - Complexity

- Worst case:
  - K steps such that $2^K = N$

  i.e. $\log_2 N$ steps is O(log(N))

Abstract Data Type

- Abstract Data Type

In computer science, an **abstract data type** (**ADT**) is a mathematical model for a certain class of data structures that have similar behavior.

Abstract Data Types (ADTs)

- A method for achieving abstraction for data structures and algorithms
- ADT = model + operations
- Describes what each operation does, but not how it does it
- An ADT is independent of its implementation

Abstract Data Types

- Typical operations on data
  - Add data to a data collection
  - Remove data from a data collection
  - Ask questions about the data in a data collection

Abstract Data Types

- Data abstraction
  - Asks you to think *what* you can do to a collection of data independently of *how* you do it
  - Allows you to develop each data structure in relative isolation from the rest of the solution
  - A natural extension of procedural abstraction

Examples

- Simple ADTs
  - *Stack*
  - *Queue*
  - Vector
  - Lists
  - Sequences
  - Iterators

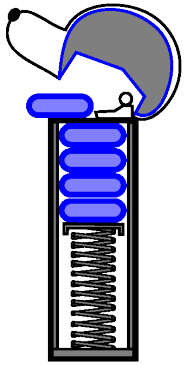
  All these are called Linear Data Structures

Stacks

- A **stack** is a container of objects that are inserted and removed according to the last-in-first-out (**LIFO**) principle.
- Objects can be inserted at any time, but only the last (the most-recently inserted) object can be removed.
- Inserting an item is known as "**pushing**" onto the stack. "**Popping**" off the stack is synonymous with removing an item.

Stacks

- A coin dispenser as an analogy:



Stacks: An Array Implementation

- Create a stack using an array by specifying a maximum size *N* for our stack.
- The stack consists of an N-element array *S* and an integer variable *t*, the index of the top element in array S.

- Array indices start at 0, so we initialize *t* to -1

Stacks: An Array Implementation

- **Pseudo code**

```
Algorithm size()
return t+1

Algorithm isEmpty()
return (t<0)

Algorithm top()
if isEmpty() then
   return Error
return S[t]
```

```
Algorithm push(o)
if size()==N then
   return Error
t=t+1
S[t]=o

Algorithm pop()
 if isEmpty() then
   return Error
 t=t-1

return S[t+1]
```

Stacks: An Array Implementation

The array implementation is simple and efficient (methods performed in O(1)).

Disadvantage

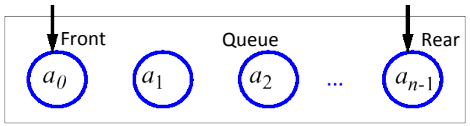
There is an upper bound, *N*, on the size of the stack.

The arbitrary value *N* may be too small for a given application OR

a waste of memory.

Queues

- A queue differs from a stack in that its insertion and removal routines follows the **first-in-first-out** (FIFO) principle.
- Elements may be inserted at any time, but only the element which has been in the queue the longest may be removed.
- Elements are inserted at the **rear** (enqueued) and removed from the **front** (dequeued)



Queues

- The **queue** supports three fundamental methods:
  - **New():***ADT – Creates an empty queue*
  - **Enqueue(S:***ADT,* **o:***element***):***ADT* - Inserts object *o* at the rear of the queue
  - **Dequeue(S:***ADT***):***ADT* - Removes the object from the front of the queue; an error occurs if the queue is empty
  - **Front(S:***ADT***):***element* - Returns, but does not remove, the front element; an error occurs if the queue is empty

Queues: An Array Implementation

- Create a queue using an array in a circular fashion
- A maximum size *N* is specified.
- The queue consists of an N-element array *Q* and two integer variables:
  - *f*, index of the front element (head – for dequeue)
  - *r*, index of the element after the rear one (tail – for enqueue)
  - Initially, f=r=0 and the queue is empty if f=r

Queues

Disadvantage

Repeatedly enqueue and dequeue a single element N times.

Finally, f=r=N.

- No more elements can be added to the queue,

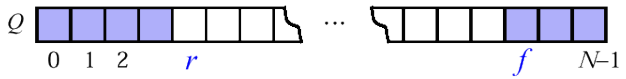
though there is space in the queue.

Solution

Let f and r wraparound the end of queue.

Queues: An Array Implementation

"wrapped around" configuration



- Each time r or f is incremented, compute this increment as (r+1)modN or (f+1)modN

Queues: An Array Implementation

- Pseudo code

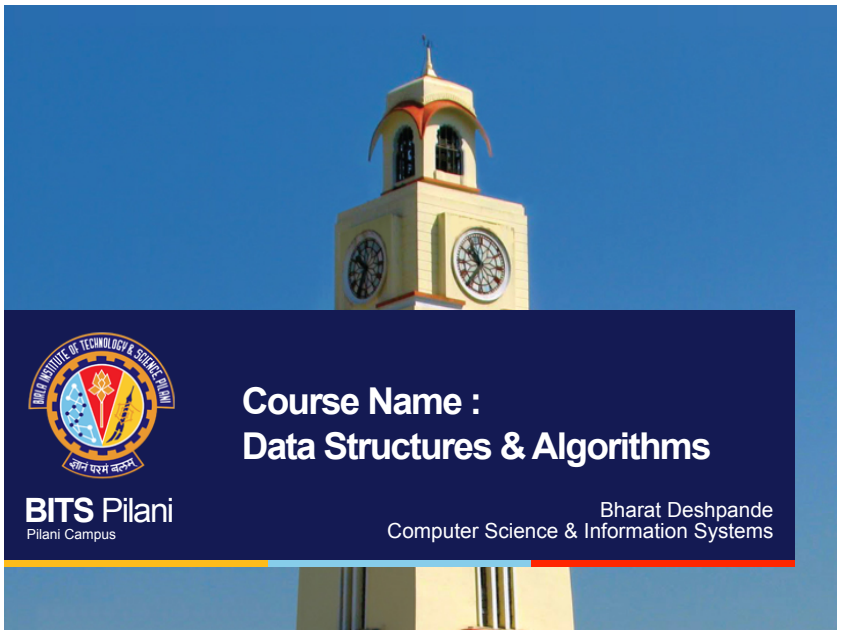
```
Algorithm size()
return (N-f+r) mod N

Algorithm isEmpty()
return (f=r)

Algorithm front()
if isEmpty() then
   return Error
return Q[f]
```

```
Algorithm dequeue()
if isEmpty() then
   return Error
Q[f]=null
f=(f+1)modN

Algorithm enqueue(o)
if size = N - 1 then
   return Error
Q[r]=o
r=(r +1)modN
```


Arrays: pluses and minuses

+ Fast element access.
-- Impossible to resize.

- Many applications require resizing!
- Required size not always immediately available.



Examples

- Simple ADTs
  - *Stack*
  - *Queue*
  - Vector
  - Lists
  - Sequences
  - Iterators

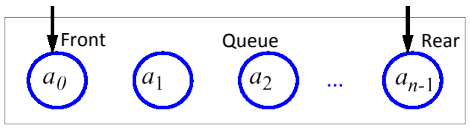
  All these are called Linear Data Structures

Stacks

- A **stack** is a container of objects that are inserted and removed according to the last-in-first-out (**LIFO**) principle.
- Objects can be inserted at any time, but only the last (the most-recently inserted) object can be removed.
- Inserting an item is known as "**pushing**" onto the stack. "**Popping**" off the stack is synonymous with removing an item.

Queues

- A queue differs from a stack in that its insertion and removal routines follows the **first-in-first-out** (FIFO) principle.
- Elements may be inserted at any time, but only the element which has been in the queue the longest may be removed.
- Elements are inserted at the **rear** (enqueued) and removed from the **front** (dequeued)



Queues

- The **queue** supports three fundamental methods:
  - **New():***ADT – Creates an empty queue*
  - **Enqueue(S:***ADT,* **o:***element***):***ADT* - Inserts object *o* at the rear of the queue
  - **Dequeue(S:***ADT***):***ADT* - Removes the object from the front of the queue; an error occurs if the queue is empty
  - **Front(S:***ADT***):***element* - Returns, but does not remove, the front element; an error occurs if the queue is empty

Queues: An Array Implementation

- Create a queue using an array
- A maximum size *N* is specified.
- The queue consists of an N-element array *Q* and two integer variables:
  - *f*, index of the front element (head – for dequeue)
  - *r*, index of the element after the rear one (tail – for enqueue)
  - Initially, f=r=0 and the queue is empty if f=r

Queues

Disadvantage

Repeatedly enqueue and dequeue a single element N times.

Finally, f=r=N.

- No more elements can be added to the queue,

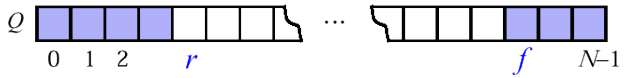
though there is space in the queue.

Solution

Let f and r wraparound the end of queue.

Queues: An Array Implementation

"wrapped around" configuration



- Each time r or f is incremented, compute this increment as (r+1)modN or (f+1)modN

Queues: An Array Implementation

- Pseudo code

```
Algorithm size()
return (N-f+r) mod N

Algorithm isEmpty()
return (f=r)

Algorithm front()
if isEmpty() then
   return Error
return Q[f]
```

```
Algorithm dequeue()
if isEmpty() then
   return Error
Q[f]=null
f=(f+1)modN
```

```
Algorithm enqueue(o)
if size = N - 1 then
   return Error
Q[r]=o
r=(r +1)modN
```

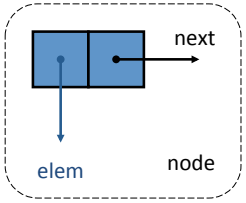
Arrays: pluses and minuses

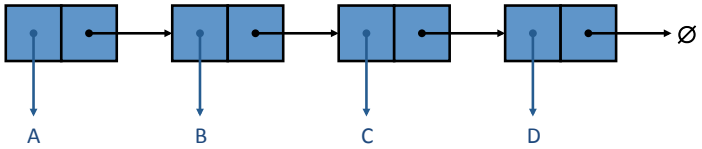
+ Fast element access.
-- Impossible to resize.

- Many applications require resizing!
- Required size not always immediately available.

Singly Linked Lists

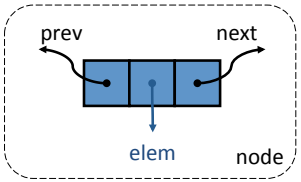
- A singly linked list is a concrete data structure consisting of a sequence of nodes
- Each node stores
  - element
  - link to the next node

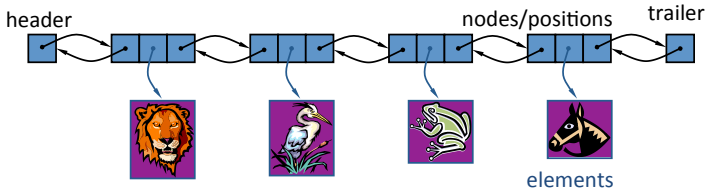




Doubly Linked List

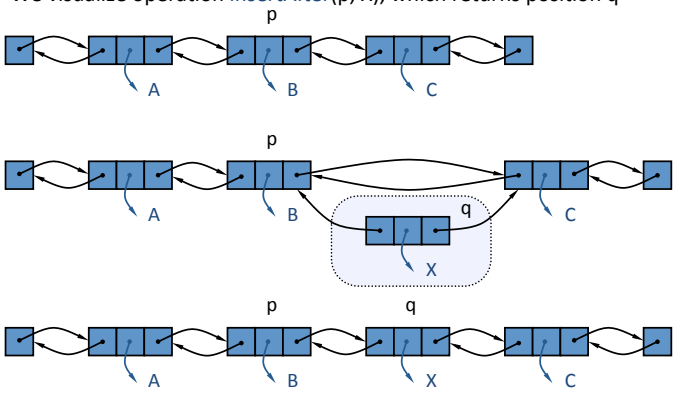
- A doubly linked list is often more convenient!
- Nodes store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes





Insertion

- We visualize operation insertAfter(p, X), which returns position q

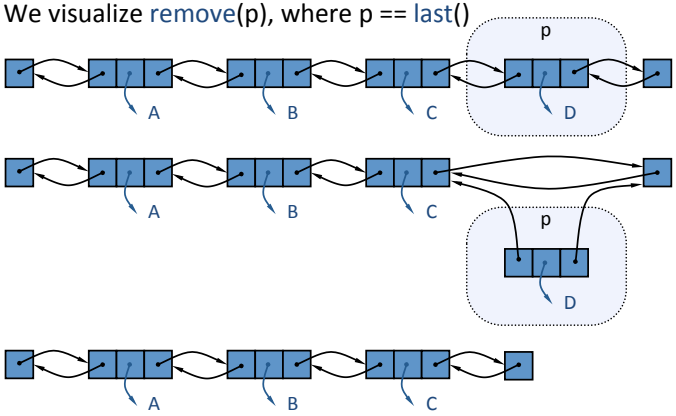


Insertion Algorithm

Algorithm insertAfter(*p,e*):
Create a new node *v*
v.element ⟵ *e*
v.prev ⟵ *p* {link *v* to its predecessor}
v.next ⟵ *p*.next {link *v* to its successor}
(*p*.next).prev ⟵ *v* {link *p*'s old successor to *v*}
p.next ⟵ *v* {link *p* to its new successor, *v*}
return *v* {the position for the element *e*}

Deletion

- We visualize remove(p), where p == last()



Deletion Algorithm

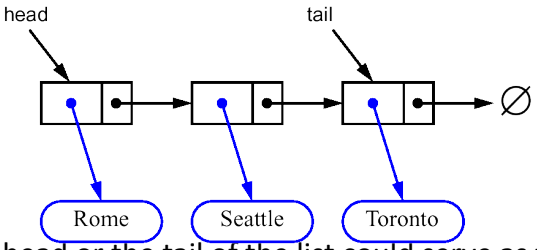
Algorithm remove(*p*):
t ⟵ *p*.element {a temporary variable to hold the return value}
(*p*.prev).next ⟵ *p*.next {linking out *p*}
(*p*.next).prev ⟵ *p*.prev
p.prev ⟵ **null** {invalidating the position *p*}
p.next ⟵ **null**
return *t*

Worst-cast running time

- In a doubly linked list
  + insertion at head or tail is in O(1)
  + deletion at either end is on O(1)
  -- element access is still in O(n)

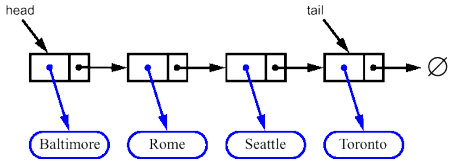
Stacks: Singly Linked List implementation

- Nodes *(data, pointer)* connected in a chain by links

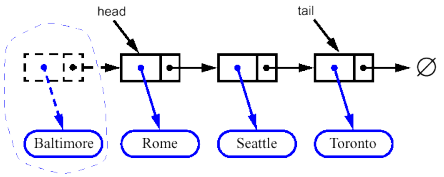


- the head or the tail of the list could serve as the top of the stack

Queues: Linked List Implementation



- Dequeue - advance head reference



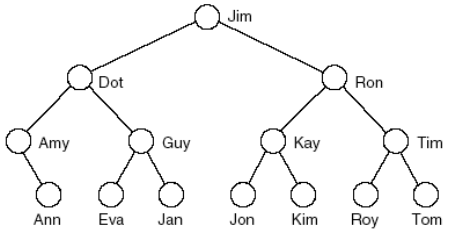
Linear data structures

- Here are some of the data structures we have studied so far:
  - Arrays
  - Singly-linked lists and doubly-linked lists
  - Stacks and queues
  - These all have the property that their elements can be adequately displayed in a straight line
- **How to obtain data structures for data that have nonlinear relationships**

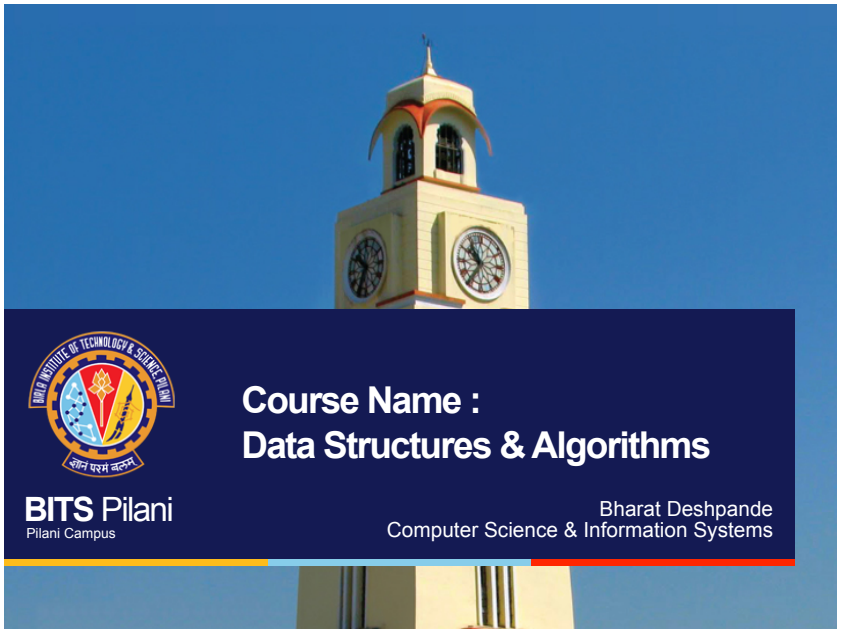
Tree

Tree represents hierarchy.

Examples of trees:
- Directory tree
- Family tree
- Company organization c
-Table of contents



-structure resembles branches of a “tree”, hence the name.



Abstract Data Type

- Abstract Data Type

In computer science, an **abstract data type** (**ADT**) is a mathematical model for a certain class of data structures that have similar behavior.

Abstract Data Types (ADTs)

- A method for achieving abstraction for data structures and algorithms
- ADT = model + operations
- Describes what each operation does, but not how it does it
- An ADT is independent of its implementation

Abstract Data Types

- Typical operations on data
  - Add data to a data collection
  - Remove data from a data collection
  - Ask questions about the data in a data collection

Abstract Data Types

- Data abstraction
  - Asks you to think *what* you can do to a collection of data independently of *how* you do it
  - Allows you to develop each data structure in relative isolation from the rest of the solution
  - A natural extension of procedural abstraction

Examples

- Simple ADTs
  - *Stack*
  - *Queue*
  - Vector
  - Lists
  - Sequences
  - Iterators

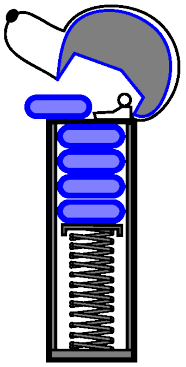
  All these are called Linear Data Structures

Stacks

- A **stack** is a container of objects that are inserted and removed according to the last-in-first-out (**LIFO**) principle.
- Objects can be inserted at any time, but only the last (the most-recently inserted) object can be removed.
- Inserting an item is known as "**pushing**" onto the stack. "**Popping**" off the stack is synonymous with removing an item.

Stacks

- A coin dispenser as an analogy:



Stacks: An Array Implementation

- Create a stack using an array by specifying a maximum size *N* for our stack.
- The stack consists of an N-element array *S* and an integer variable *t*, the index of the top element in array S.

- Array indices start at 0, so we initialize *t* to -1

Stacks: An Array Implementation

- **Pseudo code**

```
Algorithm size()
return t+1

Algorithm isEmpty()
return (t<0)

Algorithm top()
if isEmpty() then
   return Error
return S[t]
```

```
Algorithm push(o)
if size()==N then
   return Error
t=t+1
S[t]=o

Algorithm pop()
 if isEmpty() then
   return Error
 t=t-1

return S[t+1]
```

Stacks: An Array Implementation

The array implementation is simple and efficient (methods performed in O(1)).

Disadvantage

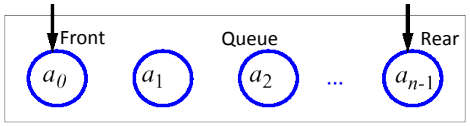
There is an upper bound, *N*, on the size of the stack.

The arbitrary value *N* may be too small for a given application OR

a waste of memory.

Queues

- A queue differs from a stack in that its insertion and removal routines follows the **first-in-first-out** (FIFO) principle.
- Elements may be inserted at any time, but only the element which has been in the queue the longest may be removed.
- Elements are inserted at the **rear** (enqueued) and removed from the **front** (dequeued)



Queues

- The **queue** supports three fundamental methods:
  - **New():***ADT – Creates an empty queue*
  - **Enqueue(S:***ADT,* **o:***element***):***ADT* - Inserts object *o* at the rear of the queue
  - **Dequeue(S:***ADT***):***ADT* - Removes the object from the front of the queue; an error occurs if the queue is empty
  - **Front(S:***ADT***):***element* - Returns, but does not remove, the front element; an error occurs if the queue is empty

Queues: An Array Implementation

- Create a queue using an array
- A maximum size *N* is specified.
- The queue consists of an N-element array *Q* and two integer variables:
  - *f*, index of the front element (head – for dequeue)
  - *r*, index of the element after the rear one (tail – for enqueue)
  - Initially, f=r=0 and the queue is empty if f=r

Queues

Disadvantage

Repeatedly enqueue and dequeue a single element N times.

Finally, f=r=N.

- No more elements can be added to the queue,

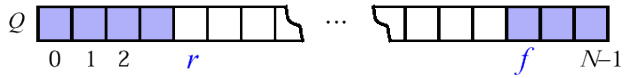
though there is space in the queue.

Solution

Let f and r wraparound the end of queue.

Queues: An Array Implementation

"wrapped around" configuration



- Each time r or f is incremented, compute this increment as (r+1)modN or (f+1)modN

Queues: An Array Implementation

- Pseudo code

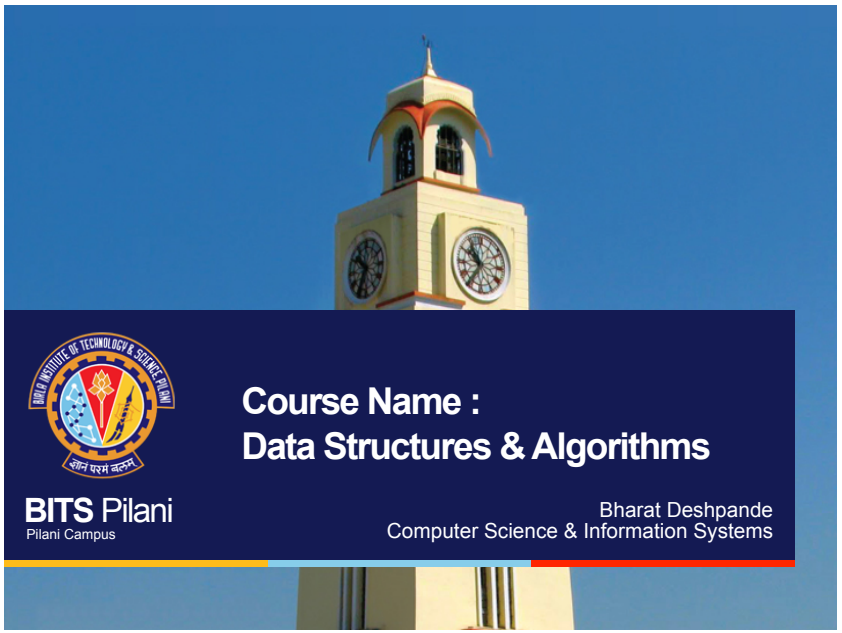
```
Algorithm size()
return (N-f+r) mod N

Algorithm isEmpty()
return (f=r)

Algorithm front()
if isEmpty() then
   return Error
return Q[f]
```

```
Algorithm dequeue()
if isEmpty() then
   return Error
Q[f]=null
f=(f+1)modN

Algorithm enqueue(o)
if size = N - 1 then
   return Error
Q[r]=o
r=(r +1)modN
```


Arrays: pluses and minuses

+ Fast element access.
-- Impossible to resize.

- Many applications require resizing!
- Required size not always immediately available.



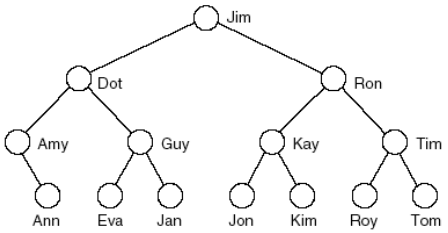
Linear data structures

- Here are some of the data structures we have studied so far:
  - Arrays
  - Singly-linked lists and doubly-linked lists
  - Stacks and queues
  - These all have the property that their elements can be adequately displayed in a straight line
- **How to obtain data structures for data that have nonlinear relationships**

Tree

Tree represents hierarchy.

Examples of trees:
- Directory tree
- Family tree
- Company organization c
-Table of contents

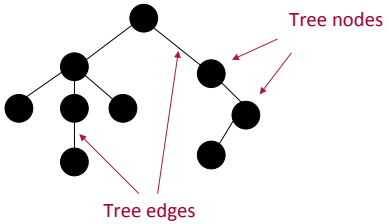


-structure resembles branches of a “tree”, hence the name.

Trees

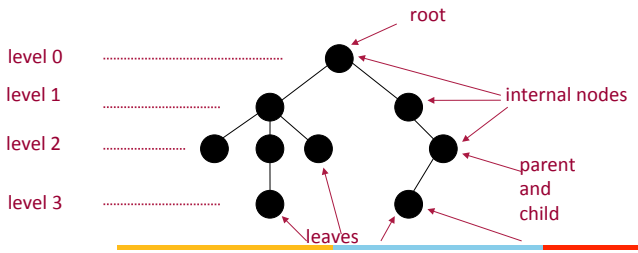
Trees have **nodes**. They also have **edges** that connect the nodes.

- Between two nodes there is *always only one* path.



Trees: More Definitions

- Trees that we consider are rooted. Once the **root** is defined (by the user) all nodes have a specific **level**.
- Trees have **internal** nodes and **leaves**. Every node (except the root) has a **parent** and it also has zero or more **children**.



Tree Terminology (1)

A vertex (or node) is a object that can have a name and can carry other associated information

- The first or top node in a tree is called the root node.
- An edge is a connection between two vertices
- A path in a tree is a list of distinct vertices in which successive vertices are connected by edges in the tree.
- The defining property of a tree is that there is precisely one path connecting any two nodes.
- A disjoint set of trees is called a forest
- Nodes with no children are leaves, terminal or external nodes

Tree Terminology (2)

Child of a node u :- Any node reachable from u by 1 edge.

Parent node :- If b is a child of a, then a is the parent of b.

- All nodes except root have exactly one parent.

Subtree:-any node of a tree, with all of its descendants.

Depth of a node :

- Depth of root node is 0.

-Depth of any other node is 1 greater than depth of its parent.

Tree Terminology (3)

The size of a tree is the number of nodes in it

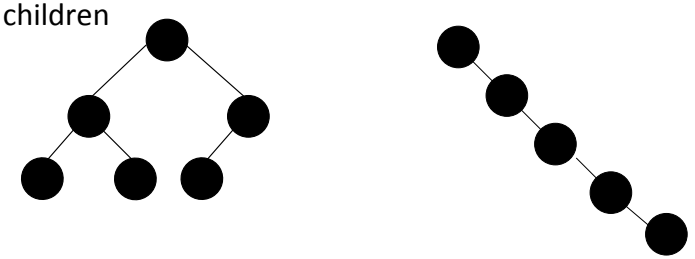
Height : *Maximum of all depths.*

Each node except the root has exactly one node above it in the tree, (i.e. it's parent), and we extend the family analogy talking of children, siblings, or grandparents
Nodes that share parents are called **siblings.**

Binary Trees

Definition: A binary tree is either empty or it consists of a root together with two binary trees called the left subtree and the right subtree.

A **binary tree** is a tree in which each node has *atmost* 2 children



Properties of Binary trees

Proper Binary Tree

- Each internal node has exactly two children

Let

n - number of nodes

n_e – number of external nodes or leaves

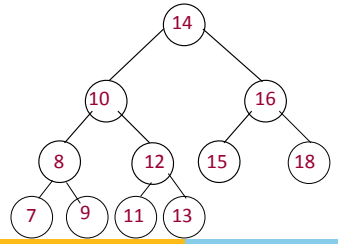
n_i – number of internal nodes

h – height of T, Then the following holds:

- $h+1 <= n_e <= 2^h$
- $h <= n_i <= 2^h - 1$
- $h + 1 <= n <= 2^{h+1} - 1$
- $\log(n+1) - 1 <= h <= (n-1)/2$

Complete Binary Trees

- Nodes in trees can contain **keys** (letters, numbers, etc)
- **Complete binary tree**: A binary tree in which every level, except possibly the deepest, is completely filled. At depth n, the height of the tree, all nodes must be as far left as possible.

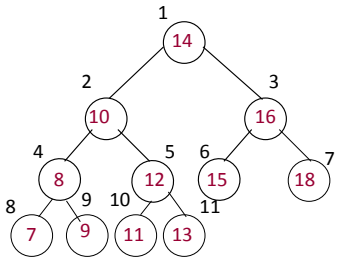


Complete Binary Trees: Array Representation

Complete Binary Trees can be represented in memory with the use of an array A so that all nodes can be accessed in O(1) time:

- Label nodes sequentially top-to-bottom and left-to-right
- Left child of A[i] is at position A[2i]
- Right child of A[i] is at position A[2i + 1]
- Parent of A[i] is at A[i/2]

Complete Binary Trees: Array Representation

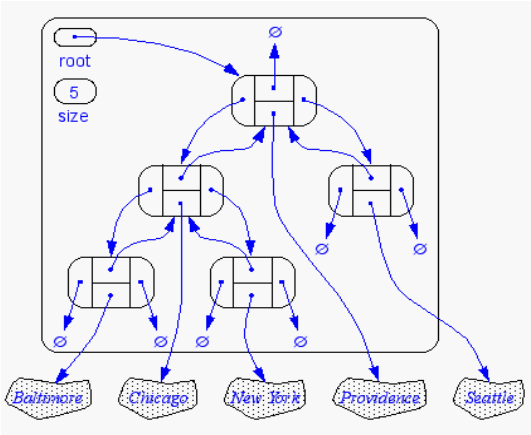


Binary Trees: Linked List Representation

A **Binary tree** is a linked data structure. Each node contains data (including a key and satellite data), and pointers left, right and p.

- ***Left points to the left child of the node.***
- ***Right points to the right child of the node.***
- ***p points to the parent of the node.***
- If a child is missing, the pointer is NIL.
- If a parent is missing, p is NIL.
- The **root** of the tree is the only node for which p is NIL.
- Nodes for which both left and right are NIL are **leaves.**

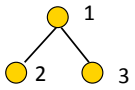
Binary Trees: Linked List Representation



Binary Tree Traversals

- A binary tree is defined recursively: it consists of a root, a left subtree and a right subtree
- To traverse (or walk) the binary tree is to visit each node in the binary tree exactly once.
- Tree traversals are naturally recursive.
- Since a binary tree has three parts, there are six possible ways to traverse the binary tree:
  - root, left, right : preorder (root, right, left)
  - left, root, right: inorder (right, root, left)
  - left, right, root: postorder (right, left, root)

Binary Tree Traversals

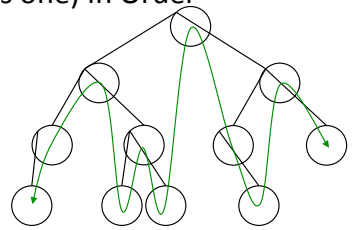


preorder: 1 2 3
inorder: 2 1 3
postorder: 2 3 1

Tree Traversal: InOrder

In-order traversal

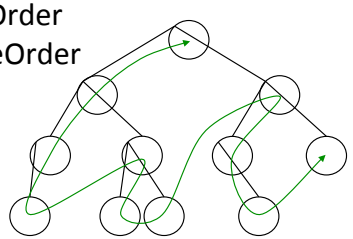
- Visit left subtree (if there is one) In Order
- print the key of the current node
- Visit right subtree (if there is one) In Order



Tree Traversal: PreOrder

Another common traversal is **PreOrder**.
It goes as deep as possible (visiting as it goes) then left to right

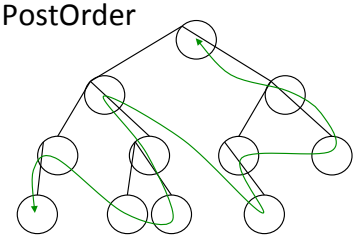
- print root
- Visit left subtree in PreOrder
- Visit right subtree in PreOrder



Tree Traversal: PostOrder

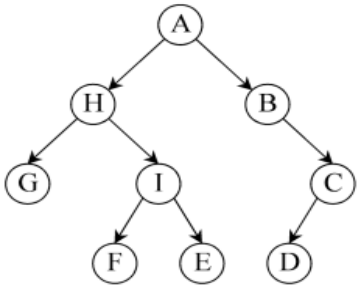
PostOrder traversal also goes as deep as possible, but only visits internal nodes during backtracking.
recursive:

- Visit left subtree in PostOrder
- Visit right subtree in PostOrder
- print root



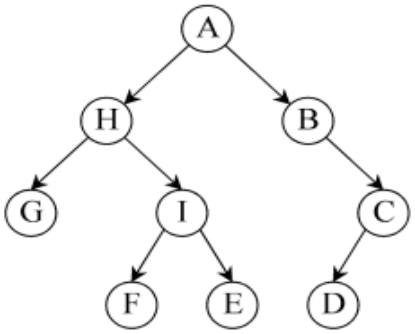
Tree Traversal: examples

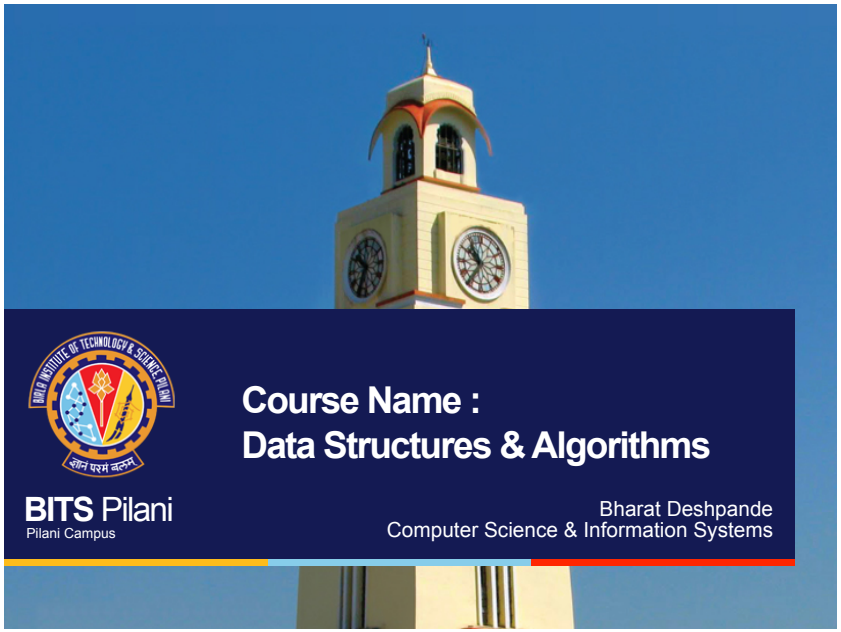
Preorder (DLR) traversal yields: A, H, G, I, F, E, B, C, D
Postorder (LRD) traversal yields: G, F, E, I, H, D, C, B,
In-order (LDR) traversal yields: G, H, F, I, E, A, B, D, C



Tree Traversal: examples

* Preorder (DLR) traversal yields: A, H, G, I, F, E, B, C, D
* Postorder (LRD) traversal yields: G, F, E, I, H, D, C, B, A
* In-order (LDR) traversal yields: G, H, F, I, E, A, B, D, C





Properties of Binary trees

Proper Binary Tree

- Each internal node has exactly two children

Let

n - number of nodes

n_e – number of external nodes or leaves

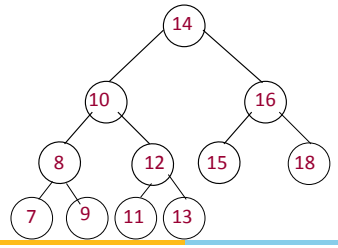
n_i – number of internal nodes

h – height of T, Then the following holds:

- $h+1 <= n_e <= 2^h$
- $h <= n_i <= 2^h - 1$
- $h + 1 <= n <= 2^{h+1} - 1$
- $\log(n+1) - 1 <= h <= (n-1)/2$

Complete Binary Trees

- Nodes in trees can contain **keys** (letters, numbers, etc)
- **Complete binary tree**: A binary tree in which every level, except possibly the deepest, is completely filled. At depth n, the height of the tree, all nodes must be as far left as possible.

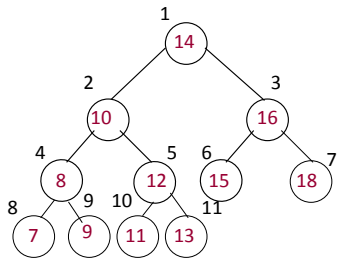


Complete Binary Trees: Array Representation

Complete Binary Trees can be represented in memory with the use of an array A so that all nodes can be accessed in O(1) time:

- Label nodes sequentially top-to-bottom and left-to-right
- Left child of A[i] is at position A[2i]
- Right child of A[i] is at position A[2i + 1]
- Parent of A[i] is at A[i/2]

Complete Binary Trees: Array Representation

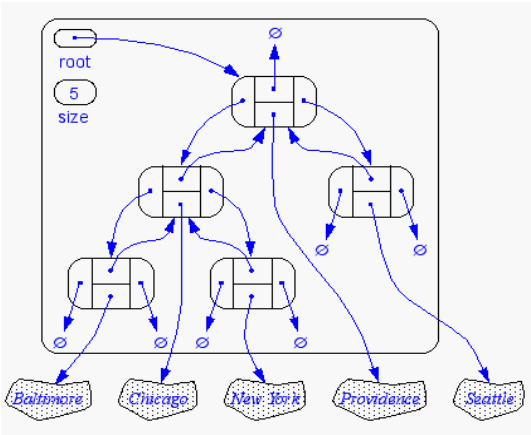


Binary Trees: Linked List Representation

A **Binary tree** is a linked data structure. Each node contains data (including a key and satellite data), and pointers left, right and p.

- ***Left* points to the left child of the node.**
- ***Right* points to the right child of the node.**
- ***p* points to the parent of the node.**
- If a child is missing, the pointer is NIL.
- If a parent is missing, p is NIL.
- The **root** of the tree is the only node for which p is NIL.
- Nodes for which both left and right are NIL are **leaves.**

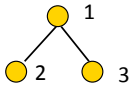
Binary Trees: Linked List Representation



Binary Tree Traversals

- A binary tree is defined recursively: it consists of a root, a left subtree and a right subtree
- To traverse (or walk) the binary tree is to visit each node in the binary tree exactly once.
- Tree traversals are naturally recursive.
- Since a binary tree has three parts, there are six possible ways to traverse the binary tree:
  - root, left, right : preorder (root, right, left)
  - left, root, right: inorder (right, root, left)
  - left, right, root: postorder (right, left, root)

Binary Tree Traversals

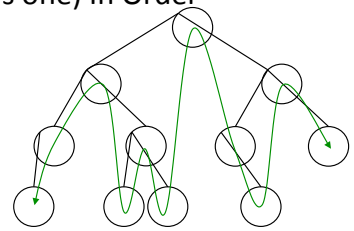


preorder: 1 2 3
inorder: 2 1 3
postorder: 2 3 1

Tree Traversal: InOrder

In-order traversal

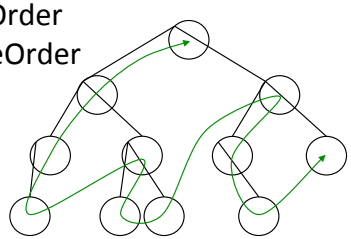
- Visit left subtree (if there is one) In Order
- print the key of the current node
- Visit right subtree (if there is one) In Order



Tree Traversal: PreOrder

Another common traversal is **PreOrder**.
It goes as deep as possible (visiting as it goes) then left to right

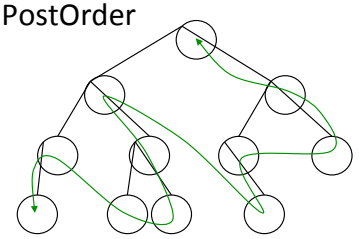
- print root
- Visit left subtree in PreOrder
- Visit right subtree in PreOrder



Tree Traversal: PostOrder

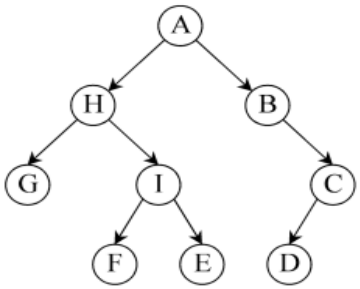
PostOrder traversal also goes as deep as possible, but only visits internal nodes during backtracking.
recursive:

- Visit left subtree in PostOrder
- Visit right subtree in PostOrder
- print root



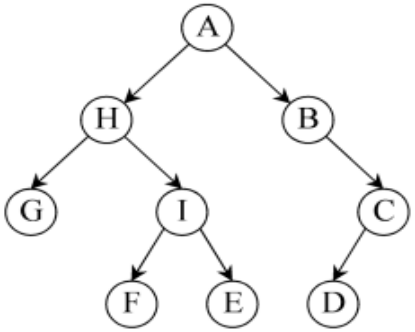
Tree Traversal: examples

Preorder (DLR) traversal yields: A, H, G, I, F, E, B, C, D
Postorder (LRD) traversal yields: G, F, E, I, H, D, C, B,
In-order (LDR) traversal yields: G, H, F, I, E, A, B, D, C



Tree Traversal: examples

* Preorder (DLR) traversal yields: A, H, G, I, F, E, B, C, D
* Postorder (LRD) traversal yields: G, F, E, I, H, D, C, B, A
* In-order (LDR) traversal yields: G, H, F, I, E, A, B, D, C



Sorting Problem

- Input : A sequence of n numbers
  <a1, a2,…..,an>
- Output : A permutation (reordering)
  <a'1, a'2,…..,a'n> of the input sequence such that a'1 <= a'2<=…..<=a'n
- Solutions : Many!
- First Solution : "Insertion Sort"

Insertion Sort

- Big idea:
  - Inserting an element into a sorted list in the appropriate position retains the order.
  - Works the way many people sort a hand of playing cards.
  - Start with an empty left hand and the cards face down on the table.
  - We remove one card from the table and insert it in the correct position in left hand.

Insertion Sort

– To find the correct position for a card, we compare it with each of the cards already in the hand, from right to left.

– Important :

At all times the cards in the left hand are sorted, and these cards were originally the top cards of the pile on the table.

Insertion Sort

Crucial Idea

– Start with a singleton list – sorted trivially.

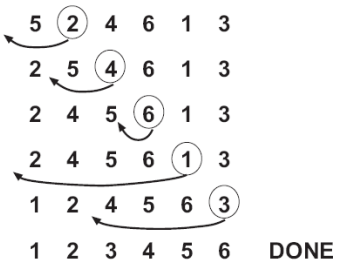
– Repeatedly insert elements – one at a time – while keeping it sorted.

– Initially, x will need to be the second element and a[1] the 'sorted part'.

– Sorted part is extended by first inserting the 2^{nd} element, then the 3^{rd} & so on.

Insertion Sort (Con't)

- Example:



Insertion Sort – Pseudo Code

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
   *insert A[i] into the sorted sequence A[1,..,i-1]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

Insertion Sort - Analysis

- Best Case Analysis
  The best case for insertion sort occurs when the list is already sorted.
  In this case, insertion sort requires n-1 comparisons
  i.e., O(n) complexity.
- Worst Case Analysis
  for each value of i, what is the maximum number of key comparisons possible?
  - Answer: i -1
- Thus, the total time in the worst case is
  T(n) = 1+2+3+……+(n-1)
  = n(n-1)/2
  = O(n^2)

Insertion Sort - Analysis

- Average Case Analysis
  - We assume that all permutations of the keys are equally likely as input.
  - We also assume that the keys are distinct.
  - We first determine how many key comparisons are done on average to insert one new element into the sorted segment.

Insertion Sort – Average Case

- When we deal with entry *i*, how far back must we go to insert it?

  Answer:

  There are *i* possible positions: not moving at all, moving by one position up to moving by *i* – *1* positions.
- Given randomness, these are equally likely.

Insertion Sort – Average Case

Average no. of comparisons

=

$$\frac{1}{i}\sum_{j=1}^{i-1} j + \frac{i-1}{i} = \frac{i-1}{2} + 1 - \frac{1}{i}$$

Total =

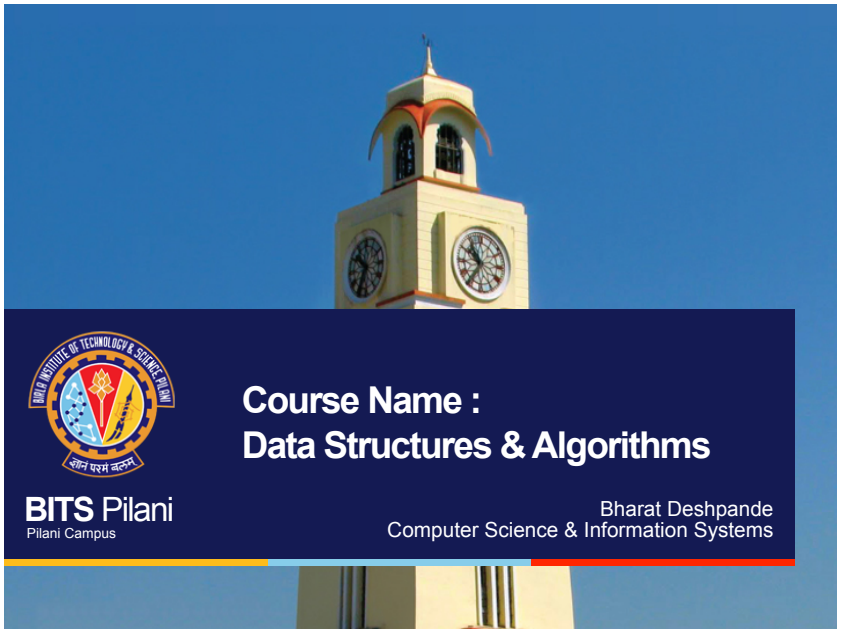
$$\sum_{i=1}^{n-1}\left(\frac{i-1}{2} + 1 - \frac{1}{i}\right) = \frac{(n-1)(n-2)}{4} + n - 1 - \sum_{i=1}^{n-1}\frac{1}{i}$$

Insertion Sort – Average Case

- Well known

$$\sum_{i=1}^{n} \frac{1}{i} \approx \ln n$$

- Thus, the total number of comparisons
  = O(n^2)



Sorting Problem

- Input : A sequence of n numbers
  <a1, a2,…..,an>
- Output : A permutation (reordering)
  <a'1, a'2,…..,a'n> of the input sequence such that a'1 <= a'2<=…..<=a'n
- Solutions : Many!
- First Solution : "Insertion Sort"

Insertion Sort

- Big idea:
  - Inserting an element into a sorted list in the appropriate position retains the order.
  - Works the way many people sort a hand of playing cards.
  - Start with an empty left hand and the cards face down on the table.
  - We remove one card from the table and insert it in the correct position in left hand.

Insertion Sort

– To find the correct position for a card, we compare it with each of the cards already in the hand, from right to left.

– Important :

At all times the cards in the left hand are sorted, and these cards were originally the top cards of the pile on the table.

Insertion Sort

Crucial Idea

– Start with a singleton list – sorted trivially.

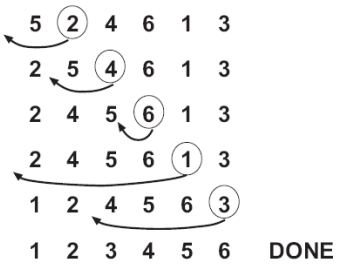
– Repeatedly insert elements – one at a time – while keeping it sorted.

– Initially, x will need to be the second element and a[1] the 'sorted part'.

– Sorted part is extended by first inserting the 2nd element, then the 3rd & so on.

Insertion Sort (Con't)

- Example:



Insertion Sort – Pseudo Code

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
   *insert A[i] into the sorted sequence A[1,..,i-1]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

Insertion Sort - Analysis

- Best Case Analysis
  The best case for insertion sort occurs when the list is already sorted.
  In this case, insertion sort requires n-1 comparisons
  i.e., O(n) complexity.
- Worst Case Analysis
  for each value of i, what is the maximum number of key comparisons possible?
  - Answer: i -1
- Thus, the total time in the worst case is
  T(n) = 1+2+3+……+(n-1)
  = n(n-1)/2
  = O(n^2)

Insertion Sort - Analysis

- Average Case Analysis
  - We assume that all permutations of the keys are equally likely as input.
  - We also assume that the keys are distinct.
  - We first determine how many key comparisons are done on average to insert one new element into the sorted segment.

Insertion Sort – Average Case

- When we deal with entry *i*, how far back must we go to insert it?
  Answer:
  There are *i* possible positions: not moving at all, moving by one position up to moving by *i – 1* positions.
- Given randomness, these are equally likely.

Insertion Sort – Average Case

Average no. of comparisons

$$= \frac{1}{i}\sum_{j=1}^{i-1} j + \frac{i-1}{i} = \frac{i-1}{2} + 1 - \frac{1}{i}$$

Total =

$$\sum_{i=1}^{n-1}\left(\frac{i-1}{2} + 1 - \frac{1}{i}\right) = \frac{(n-1)(n-2)}{4} + n - 1 - \sum_{i=1}^{n-1}\frac{1}{i}$$

Insertion Sort – Average Case

- Well known

$$\sum_{i=1}^{n} \frac{1}{i} \approx \ln n$$

- Thus, the total number of comparisons
  = O(n^2)

Merge Sort

- **Divide:** Divide the n_element sequence to be sorted into two subsequences of n/2 elements each.
- **Conquer:** Sort the two subsequences recursively using merge sort.
- **Combine:** Merge the two sorted subsequences to produce a sorted list.
- The general algorithm for the merge sort is as follows:
- If the list is of size greater than 1, then
  a. Find the mid-position of the list.
  b. Merge sort the first sublist.
  c. Merge sort the second sublist.
  d. Merge the first sublist and the second sublist.

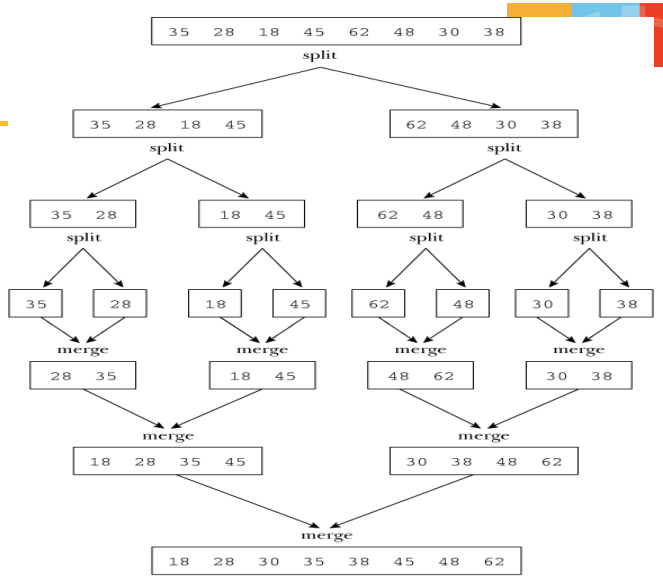


FIGURE 9.22 Merge sort process

Merging two sorted Array

- Once the sublists are sorted, the next step in the merge sort algorithm is to merge the sorted sublists.
- Suppose L_1 and L_2 are two sorted lists as follows:
  - L_1: 2, 7, 16, 35
  - L_2: 5, 20, 25, 40, 50
- Merge L_1 and L_2 into a third list, say L_3.
- The merge process is as follows:
  repeatedly compare, using a loop, the elements of L_1 with the elements of L_2 and copy the smaller element into L_3.

Example

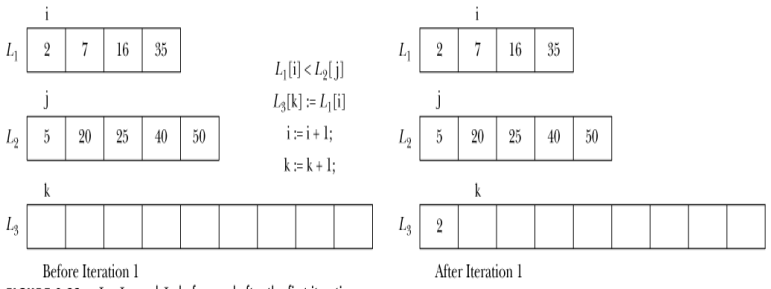


FIGURE 9.23 L_1, L_2, and L_3 before and after the first iteration

- First compare $L_1[1]$ with $L_2[1]$ and see that $L_1[1] < L_2[1]$, so copy $L_1[1]$ into $L_3[1]$

Time : If both the list has n elements each then the merging process takes 2n time in the worst case.

Merge Sort (Complexity)

- Running time analysis:
  - T(n): worst-case running time of merge sort to sort n numbers (assume n is a power of 2)

Running time analysis can be modeled as an recurrence equation:

$T(1) = 1$, if $n=1$

$T(n) = 2T(n/2) + n$, if $n>1$

Merge Sort Complexity (Con't)

- Running time analysis (Con't):

$$
\begin{aligned}
&T(1) = 1 && \textit{Initial condition}\\
&T(n) = 2T(\tfrac{n}{2}) + n\\
&\quad = 2(2T(\tfrac{n}{4}) + \tfrac{n}{2}) + n\\
&\quad = 4T(\tfrac{n}{4}) + 2n\\
&\quad = 4(2T(\tfrac{n}{8}) + \tfrac{n}{4}) + 2n\\
&\quad = 8T(\tfrac{n}{8}) + 3n\\
&\quad \vdots\\
&\quad = 2^k T(\tfrac{n}{2^k}) + kn && \textit{Since } n = 2^k, \textit{ we have } k = log_2 n\\
&\quad = nT(1) + n\ log_2\ n && \textit{Since } T(1) = 1\\
&\quad = n + n\ log_2\ n\\
&\quad = O(n\ log\ n)
\end{aligned}
$$

Design Strategy

Divide and Conquer

- is a general algorithm design paradigm:
  - Divide: divide the input data S in two or more disjoint subsets S_1, S_2, …
  - Recur: solve the subproblems recursively
  - Conquer: combine the solutions for S_1, S_2, …, into a solution for S

- The base case for the recursion are subproblems of constant size
- Analysis can be done using **recurrence equations**

Master Method (Appendix)

- Many divide-and-conquer recurrence equations have the form:

$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$

- **The Master Theorem:**
  1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.

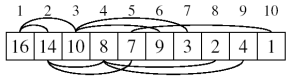
Heap Data Structure

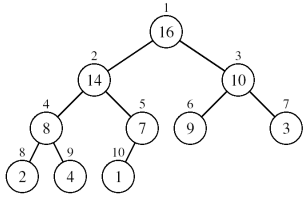
- A **heap** is a complete binary tree with the following two properties:
  - **Structural property:** all levels are full, except possibly the last one, which is filled from left to right
  - **Order (heap) property:** for any node x

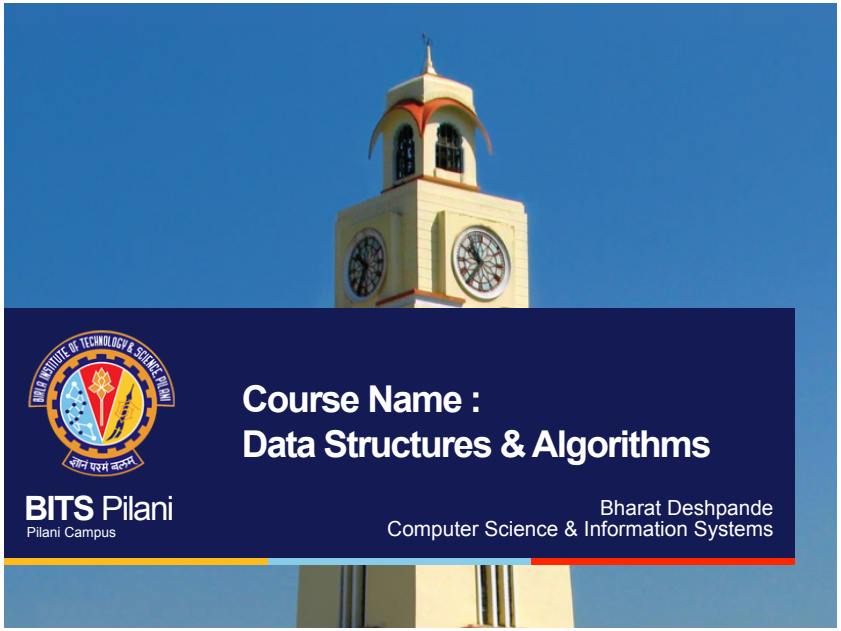
    Parent(x) ≥ x
- From the heap property, it follows that:
- "The root is the maximum element of the heap!"

Heap represented as Array

- A heap can be stored as an array *A*.
- Root of tree is *A*[1]
- Left child of *A*[i] = *A*[2i]
- Right child of *A*[i] = *A*[2i + 1]
- Parent of *A*[i] = *A*[⌊i/2⌋]







Sorting Problem

- Input : A sequence of n numbers
  <a1, a2,…..,an>
- Output : A permutation (reordering)
  <a'1, a'2,…..,a'n> of the input sequence such that a'1 <= a'2<=…..<=a'n
- Solutions : Many!
- First Solution : "Insertion Sort"

Merge Sort

- **Divide:** Divide the n_element sequence to be sorted into two subsequences of n/2 elements each.
- **Conquer:** Sort the two subsequences recursively using merge sort.
- **Combine:** Merge the two sorted subsequences to produce a sorted list.
- The general algorithm for the merge sort is as follows:
- If the list is of size greater than 1, then
  a. Find the mid-position of the list.
  b. Merge sort the first sublist.
  c. Merge sort the second sublist.
  d. Merge the first sublist and the second sublist.

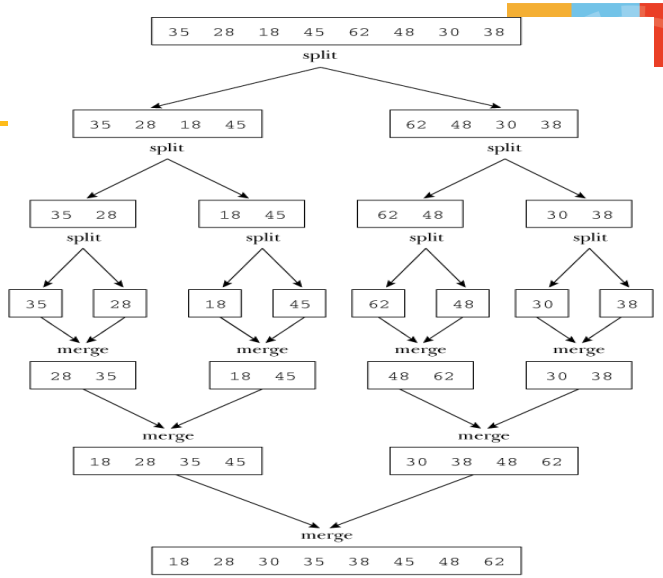


FIGURE 9.22 Merge sort process

Merging two sorted Array

- Once the sublists are sorted, the next step in the merge sort algorithm is to merge the sorted sublists.
- Suppose L_1 and L_2 are two sorted lists as follows:
  - L_1: 2, 7, 16, 35
  - L_2: 5, 20, 25, 40, 50
- Merge L_1 and L_2 into a third list, say L_3.
- The merge process is as follows:
  repeatedly compare, using a loop, the elements of L_1 with the elements of L_2 and copy the smaller element into L_3.

Example

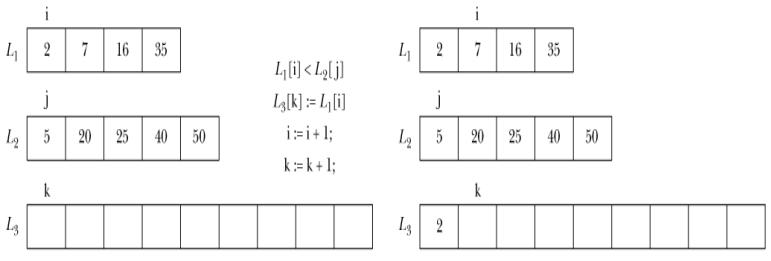


FIGURE 9.23 L_1, L_2, and L_3 before and after the first iteration

- First compare $L_1[1]$ with $L_2[1]$ and see that $L_1[1] < L_2[1]$, so copy $L_1[1]$ into $L_3[1]$

Time : If both the list has n elements each then the merging process takes 2n time in the worst case.

Merge Sort (Complexity)

- Running time analysis:
  - T(n): worst-case running time of merge sort to sort n numbers (assume n is a power of 2)

Running time analysis can be modeled as an recurrence equation:

$T(1) = 1$, if $n=1$

$T(n) = 2T(n/2) + n$, if $n>1$

Merge Sort Complexity (Con't)

- Running time analysis (Con't):

$$
\begin{aligned}
&T(1) = 1 && \textit{Initial condition}\\
&T(n) = 2T(\frac{n}{2}) + n\\
&\quad = 2(2T(\frac{n}{4}) + \frac{n}{2}) + n\\
&\quad = 4T(\frac{n}{4}) + 2n\\
&\quad = 4(2T(\frac{n}{8}) + \frac{n}{4}) + 2n\\
&\quad = 8T(\frac{n}{8}) + 3n\\
&\quad \vdots\\
&\quad = 2^kT(\frac{n}{2^k}) + kn && \textit{Since } n = 2^k, \textit{ we have } k = log_2 n\\
&\quad = nT(1) + n\ log_2\ n && \textit{Since } T(1) = 1\\
&\quad = n + n\ log_2\ n\\
&\quad = O(n\ log\ n)
\end{aligned}
$$

Design Strategy

Divide and Conquer

- is a general algorithm design paradigm:
  - Divide: divide the input data S in two or more disjoint subsets S_1, S_2, …
  - Recur: solve the subproblems recursively
  - Conquer: combine the solutions for S_1, S_2, …, into a solution for S

- The base case for the recursion are subproblems of constant size
- Analysis can be done using **recurrence equations**

Master Method (Appendix)

- Many divide-and-conquer recurrence equations have the form:

$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$

- **The Master Theorem:**
  1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.

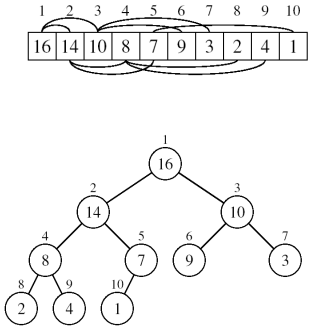
Heap Data Structure

- A **heap** is a complete binary tree with the following two properties:
  - **Structural property:** all levels are full, except possibly the last one, which is filled from left to right
  - **Order (heap) property:** for any node x

    Parent(x) ≥ x
- From the heap property, it follows that:
- The root is the maximum element of the heap!
- All leaf nodes automatically have the heap property!

Heap represented as Array

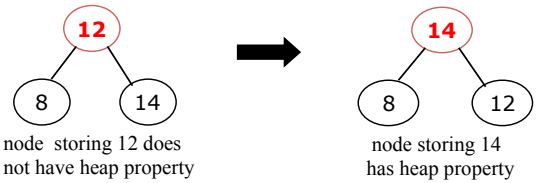
- A heap can be stored as an array *A*.
- Root of tree is A[1]
- Left child of A[i] = A[2i]
- Right child of A[i] = A[2i + 1]
- Parent of A[i] = A[⌊i/2⌋]



- **Height of an heap storing n elements is O(log n)**

Heapify

- Given a node that does not have the heap property, you can give it the heap property by exchanging its value with the value of the larger child



- Notice that the child may have lost the heap property
- Recursively fix the children until all of them satisfy the max-heap property.

Max-Heapify

MaxHeapify(*A*, *i*)
1. *l* ← left(*i*)
2. *r* ← right(*i*)
3. **if** *l* ≤ *heap-size*[*A*] and *A*[*l*] > *A*[*i*]
4. **then** *largest* ← *l*
5. **else** *largest* ← *i*
6. **if** *r* ≤ *heap-size*[*A*] **and** *A*[*r*] > *A*[*largest*]
7. **then** *largest* ← *r*
8. **if** *largest* ≠ *i*
9. **then** exchange *A*[*i*] ↔ *A*[*largest*]
10. *MaxHeapify*(*A*, *largest*)

Assumption:
Left(*i*) and Right(*i*) are max-heaps.

•Running time for MaxHeapify is O(log n)

Building a Heap

- Use MaxHeapify to convert an array *A* into a max-heap.
- **How?**
- Call MaxHeapify on each element in a bottom-up manner.

BuildMaxHeap(*A*)
1. *heap-size*[*A*] ← *length*[*A*]
2. **for** *i* ← ⌊*length*[*A*]/2⌋ **downto** 1
3. **do** *MaxHeapify*(*A*, *i*)

- **Why from i = ⌊length[A]/2⌋ ?**
- **Running time O(nlog n)**

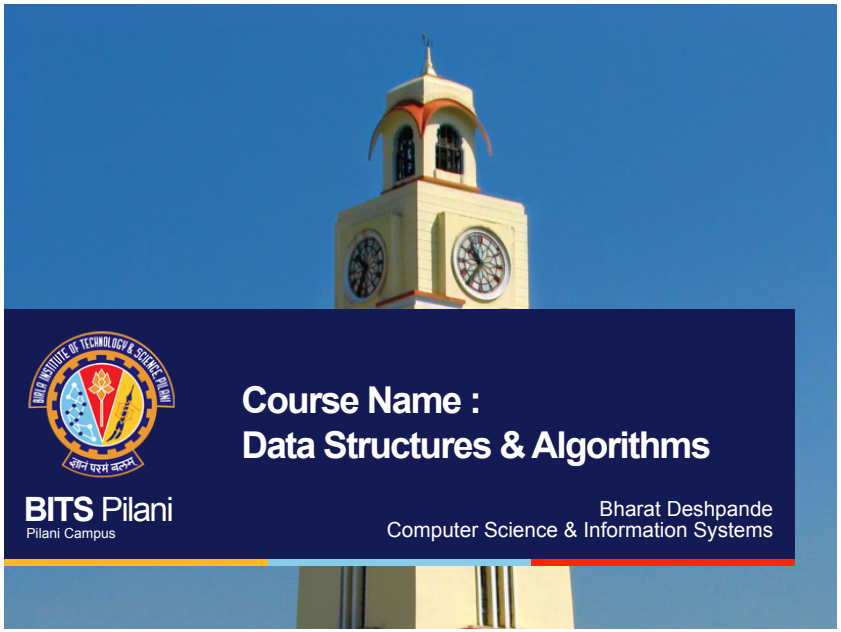
Heap Sort

- Start by building a max-heap on all elements in *A*.
  - Maximum element is in the root, *A*[1].
- Move the maximum element to its correct final position.
  - Exchange *A*[1] with *A*[*n*].
- Discard *A*[*n*] – it is now sorted.
  - Decrement heap-size[*A*].
- Restore the max-heap property on *A*[1..*n*–1].
  - Call *MaxHeapify*(*A*, 1).
- Repeat until heap-size[*A*] is reduced to 2.

Heap Sort

Heap Sort(A)
1. Build-Max-Heap(*A*)
2. **for** *i* ← *length*[*A*] **downto** 2
3. **do** exchange *A*[1] ↔ *A*[*i*]
4. *heap-size*[*A*] ← *heap-size*[*A*] – 1
5. *MaxHeapify*(*A*, 1)

- Build-Max-Heap takes *O*(*nlogn*) and each of the *n-1* calls to Max-Heapify takes time *O*(log *n*).
- Therefore, **T(n) = O(n log n)**



Lower Bound for Sorting

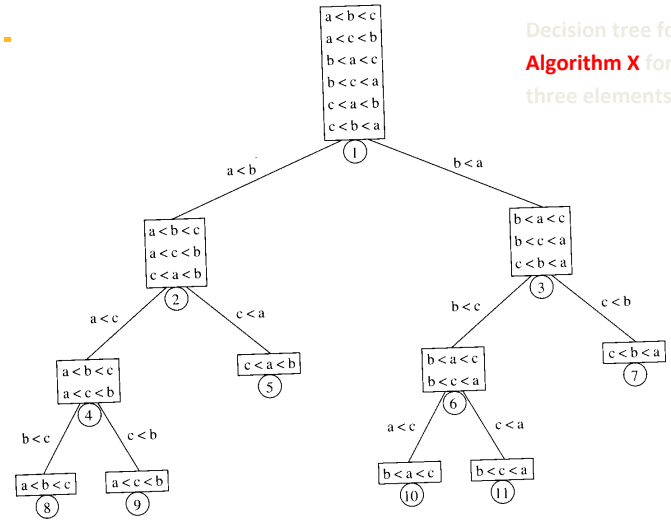
- Merge sort
  - worst-case running time is O(N log N)
- Insertion Sort
- Worst-case running time is $O(N^2)$
- Heap Sort
  - worst-case running time is O(N log N)
- Are there better algorithms?
- **Goal:** Prove that any sorting algorithm based on only comparisons takes at least O(N log N) comparisons in the worst case (worse-case input) to sort N elements.

Lower Bound for Sorting (con't..)

- Suppose we want to sort N distinct elements
- How many possible orderings do we have for N elements?
- We can have N! possible orderings (e.g., the sorted output for a,b,c can be a b c, b a c, a c b, c a b, c b a, b c a.)

Lower Bound for Sorting (con't..)

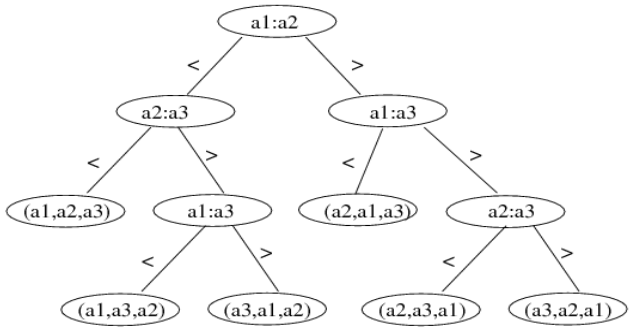
- Any comparison-based sorting process can be represented as a binary decision tree.
  - Each node represents a set of possible orderings, consistent with all the comparisons that have been made
  - The tree edges are results of the comparisons



Decision tree for **Algorithm X** for sorting three elements a, b, c

Lower Bound for Sorting (con't)

- A different algorithm would have a different decision tree
- Decision tree for Insertion Sort on 3 elements:



Lower Bound for Sorting (con't)

- The worst-case number of comparisons used by the sorting algorithm is equal to the depth of the deepest leaf
  - The average number of comparisons used is equal to the average depth of the leaves
- A decision tree to sort N elements must have N! leaves
  - a binary tree of depth d has at most 2^d leaves

  $\Rightarrow$ the tree must have depth at least $\lceil \log_2 (N!) \rceil$
- Therefore, any sorting algorithm based on only comparisons between elements requires at least $\lceil \log_2(N!) \rceil$ comparisons in the worst case.

Lower Bound for Sorting (con't..)

$$\begin{aligned}
\log_2(N!) &= \log(N(N-1)(N-2)\cdots(2)(1)) \\
&= \log N + \log(N-1) + \log(N-2) + \cdots + \log 2 + \log 1 \\
&\geq \log N + \log(N-1) + \log(N-2) + \cdots + \log(N/2) \\
&\geq \frac{N}{2}\log\frac{N}{2} \\
&= \frac{N}{2}\log N - \frac{N}{2} \\
&= \Omega(N \log N)
\end{aligned}$$

- Any sorting algorithm based on comparisons between elements requires Ω(N log N) comparisons.

Linear time sorting

- Can we do better? (linear time algorithm)

Yes, if the input has special structure

- Counting sort, radix sort, Bucket sort

Bucket Sort

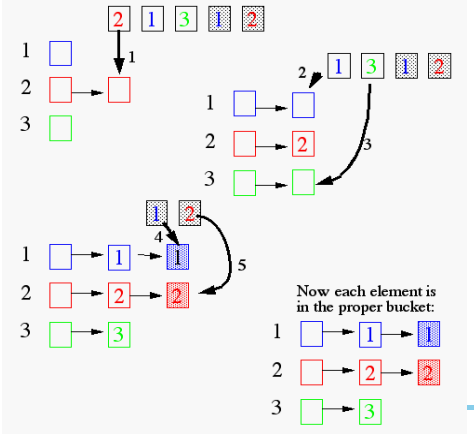
- The bucket sort makes assumptions about the data being sorted
- The n elements to be sorted are integers in the range [0, N-1]
- Consequently, we can achieve better than $\mathbf{O}(n \log n)$ run times

Idea:

- Not based on Comparison
- But using keys as indices into a bucket array.

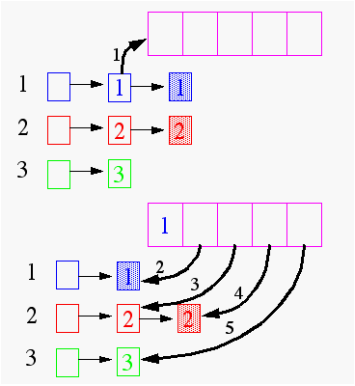
Bucket Sort (Example 1)

Each element of the array is put in one of the N "buckets"

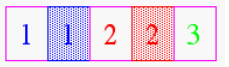


Bucket Sort

Now, pull the elements from the buckets into the array



At last, the sorted array (sorted in a stable way):



Bucket Sort Algorithm

bucketSort(s)
Let B be an array of n lists, each of which is initially empty
for each item x in S do
    let k be the key of x
    remove x from S and insert it at the end of bucket B[k]
for i = 0 to n-1 do
    for each item x in list B[i] do
        remove x from B[i] and insert it at the end of S.

Complexity

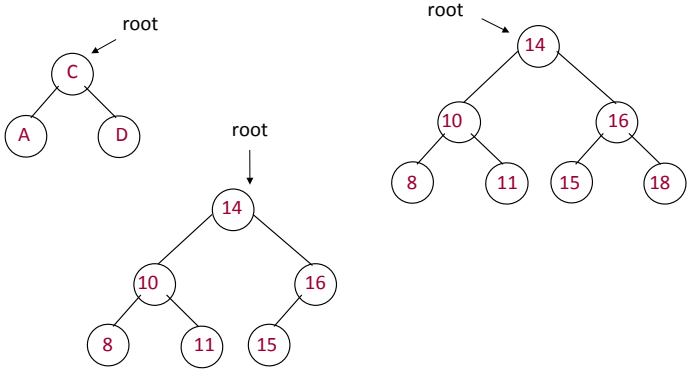
O(n + N)
If N is O(n) then we can sort in O(n) time.

Binary Search Trees

A **Binary Search Tree (BST)** is a binary tree with the following properties:

- The key of a node is *always greater* than the keys of the nodes in its left subtree
- The key of a node is *always smaller* than the keys of the nodes in its right subtree

Binary Search Trees: Examples



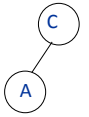
Building a BST

Build a BST from a sequence of nodes read one a time

Example: Inserting **C A B L M** **(in this order!)**

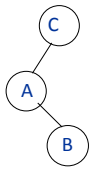
1) Insert C

2) Insert A

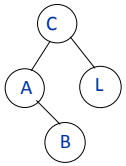


Building a BST

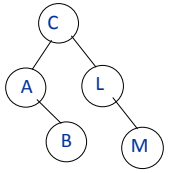
3) Insert B



4) Insert L



5) Insert M

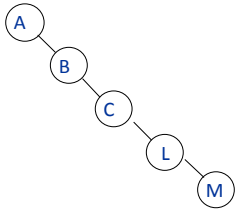


Building a BST

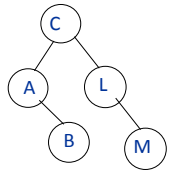
Is there a unique BST for letters A B C L M ?

NO! Different input sequences result in different trees

Inserting: A B C L M

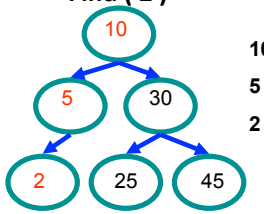


Inserting: C A B L M



Example Binary Searches

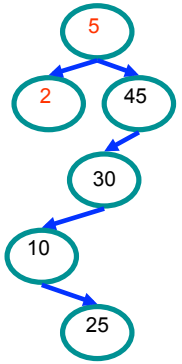
- **Find (2)**



10 > 2, left

5 > 2, left

2 = 2, found



5 > 2, left

2 = 2, found

Recursive Search of Binary Tree

```
Node Find( Node n, Value key) {
    if (n == null)                  // Not found
        return( n );
    else if (n.data == key)         // Found it
        return( n );
    else if (n.data > key)          // In left subtree
        return Find( n.left );
    else                            // In right subtree
        return Find( n.right );
}
```

Complexity of Search

- Running time of searching in a BST is proportional to the height of the tree.

If n is the number of nodes in a BST, then

- **Best Case** – O(logn)
- **Worst Case** – O(n)

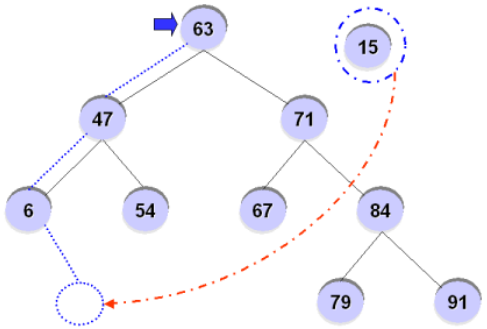
Binary Search Tree - Insertion

Insert Algorithm

- If value we want to insert < key of current node, we have to go to the left subtree
- Otherwise we have to go to the right subtree
- If the current node is empty (not existing) create a node with the value we are inserting and place it here.

Insertion - Example

For example, inserting '15' into the BST?



Binary Search Tree - Deletion

- How do we delete a node form BST?

  Similar to the insert function, after deletion of a node, the property of the BST must be maintained.

Binary Search Tree - Deletion

There are 3 possible cases

Case1 : Node to be deleted has no children

→ **We just delete the node.**

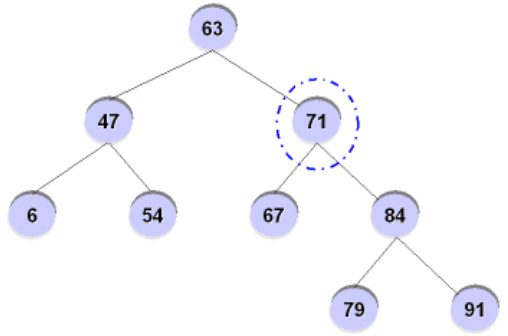
Case2 : Node to be deleted has only one child

→ **Replace the node with its child and make the parent of the deleted node to be a parent of the child of the deleted node**

Case3 : Node to be deleted has two children

Binary Search Tree - Deletion

Node to be deleted has two children



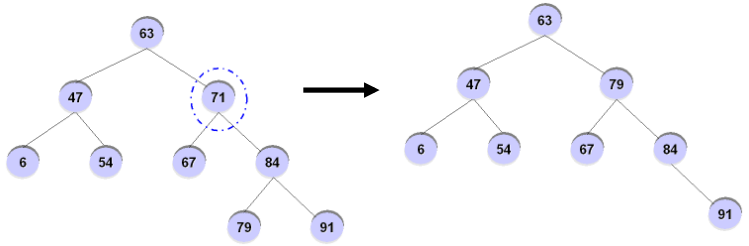
Binary Search Tree - Deletion

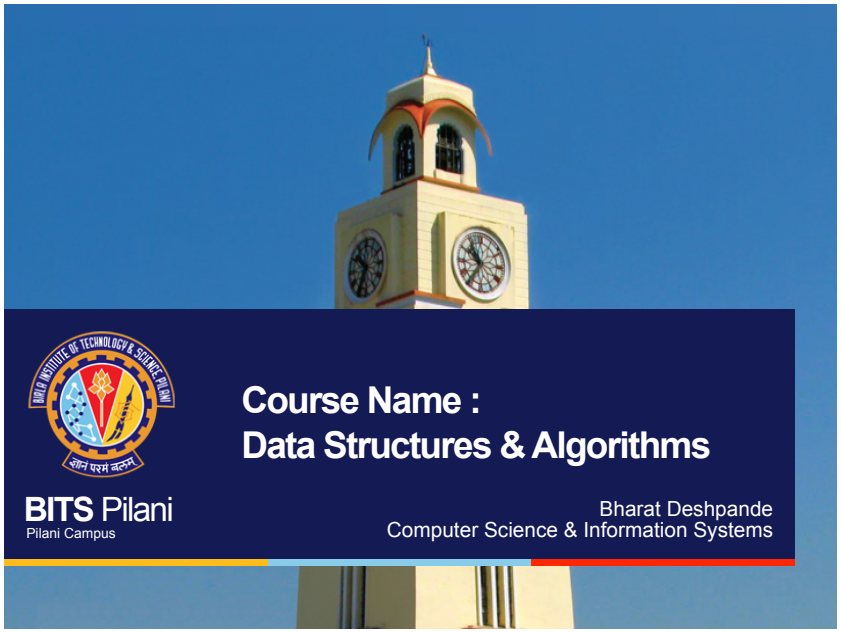
Node to be deleted has two children

Steps:

- Find minimum value of right subtree
- Delete minimum node of right subtree but keep its value
- Replace the value of the node to be deleted by the minimum value whose node was deleted earlier.

Binary Search Tree - Deletion





Binary Search Trees - Disadvantages

- Unbalanced Binary Search Trees are bad
- Worst case Performance is O(n)
- No better than sequential searching.

- We look at correcting this problem.

Height Balance Property

- Height Balance Property

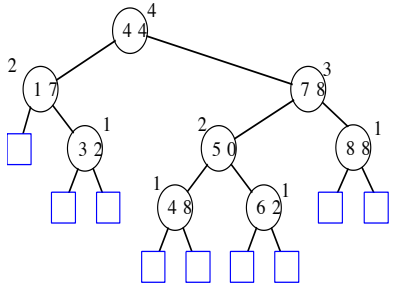
  For every internal node v of a binary search tree, the *heights of the children of v can differ by at most 1*.

- It is a property which characterizes the structure of a BST

Definition: Any Binary Search Tree T that satisfies height balance property is said to be an AVL tree. (Adelson-Velskii & Landis)

AVL Tree

Example



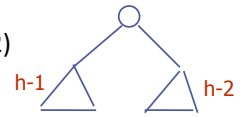
An example of an AVL tree where the heights are shown next to the nodes:

AVL tree

- Simple Observation
  Subtree of an AVL tree is itself an AVL tree.
- Important Consequence
  Keeps the height small

Height of an AVL Tree

- **Fact**: The ***height*** of an AVL tree storing n keys is O(log n).
- **Proof**: Let us bound **n(h):** the minimum number of internal nodes of an AVL tree of height h.
- We easily see that n(1) = 1 and n(2) = 2
- For h > 2, AVL tree with minimum no. of nodes is such that both its subtrees are AVL trees with minimum no. of nodes
- Such an AVL tree contains the root node, one AVL subtree of height n-1 and another of height n-2.
- That is, n(h) = 1 + n(h-1) + n(h-2)



Height of an AVL Tree

- Knowing n(h-1) > n(h-2), we get n(h) > 2n(h-2).
- So n(h) > 2n(h-2), n(h) > 4n(h-4), n(h) > 8n(n-6), …

  by induction,

  $n(h) > 2^i n(h-2i)$

  Choosing i such that h – 2i is either 1 or 2

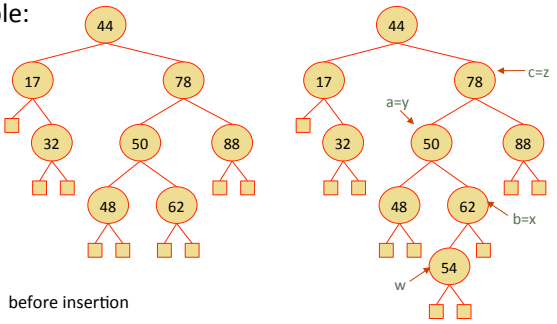
We get: $n(h) > 2^{h/2-1}$

Taking logarithms: h < 2log n(h) +2

Thus the height of an AVL tree is O(log n)

Insertion in an AVL Tree

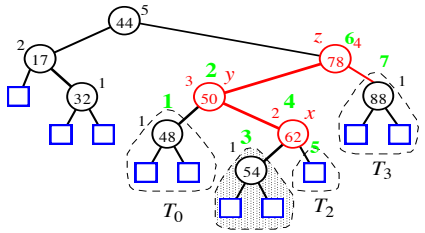
- Insertion is as in a binary search tree
- Always done by expanding an external node.
- Example:



It is no longer balanced

Insertion

- Identifying the node at which height balance property is violated.



Search & Repair strategy

- Let z be the 1st node encountered in going up from the inserted node towards the root at which the height balance property is violated.
- Let y be the child of z with higher height
- Let x be the child of y with higher height
- Trinode Structuring

  Balance the subtree rooted at z.

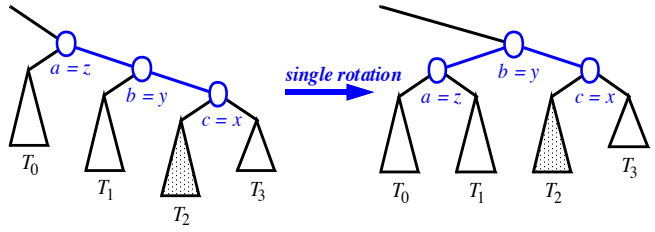
Insertion

- Relabel x, y, z as a, b, c such that a precedes b and b precedes c in an inorder traversal of T.
- There are four possible cases to do this mapping.
- Restructuring (Unified way)

Replace z with the node called b, make the children of this node be a and c while maintaining the inorder relationships of all nodes in T.

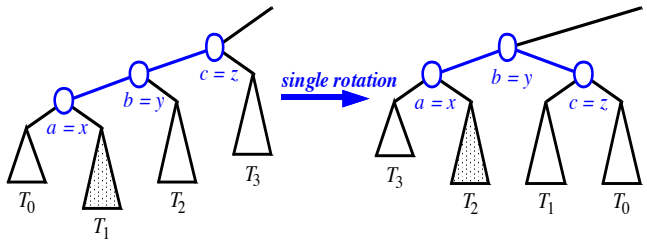
Restructuring (Single Rotation)

- Single Rotations:



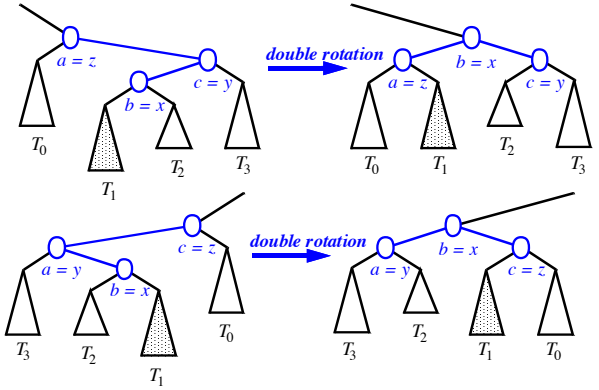
Restructuring (Single Rotation)

- Single Rotations:

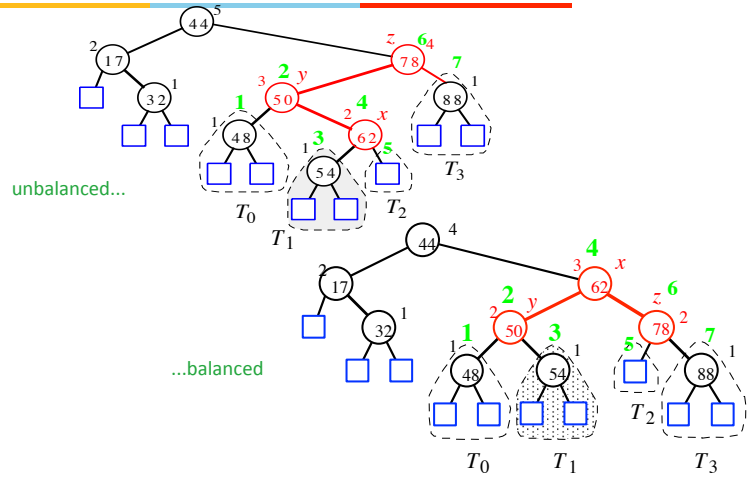


Restructuring (Double Rotation)

- double rotations:



Insertion Example, continued

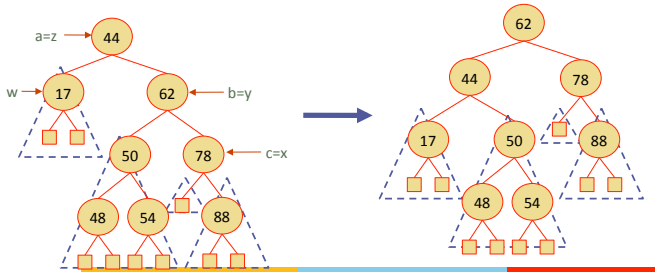


Deletion in an AVL Tree

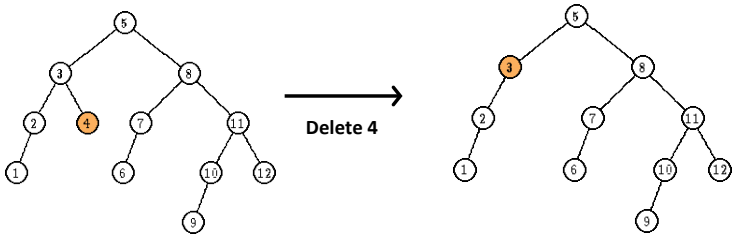
- Let ***z*** be the **first unbalanced** node encountered while travelling up the tree from w. Also,
- let ***y*** be the child of z with the larger height,
- let ***x*** be the child of y defined as follows;
  - If one of the children of y is taller than the other, choose x as the taller child of y.
  - If both children of y have the same height, select x be the child of y on the same side as y (i.e., if y is the left child of z, then x is the left child of y; and if y is the right child of z then x is the right child of y.)

Trinode Restructuring

- We perform restructure(x) to restore balance at z.
- As this restructuring may upset the balance of another node higher in the tree, we must continue checking for balance until the root of T is reached

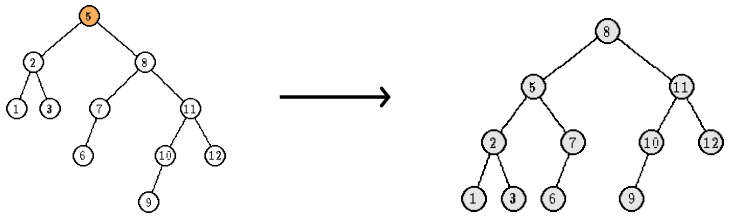


Deletion - Example



Imbalance at 3
Perform rotation with 2

Example – Contd.



Imbalance at 5
Perform rotation with 8

Running Times for AVL Trees

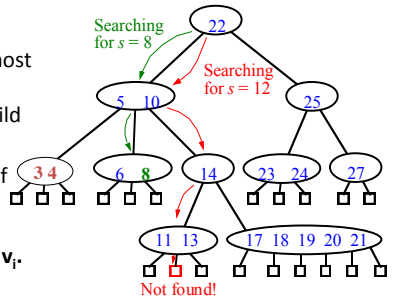
- a single restructure is O(1)
  - using a linked-structure binary tree
- find is O(log n)
  - height of tree is O(log n), no restructures needed
- insert is O(log n)
  - initial find is O(log n)
  - Restructuring up the tree, maintaining heights is O(log n)
- remove is O(log n)
  - initial find is O(log n)
  - Restructuring up the tree, maintaining heights is O(log n)

Multi-Way Search Trees

- Each internal node of a multi-way search tree T:
  - has at least two children, i.e., each internal node is a d-node, d>=2
  - stores a collection of items of the form (k, x), where k is a key and x is an element
  - Each d – node with children v1,…,vd contains d - 1 items, stored in increasing order
  - "contains" 2 pseudo-items: $k_0=-\infty$, $k_d=\infty$. For each item (k, x) stored at a node in the subtree of v rooted at vi, I =1,…,D, we have $k_{i-1} \le k \le k_i$
- Children of each internal node are "between" items
- all keys in the subtree rooted at the child fall between keys of those items

Multi-way Searching

- Similar to binary searching
  - If search key $s<k_1$ search the leftmost child
  - If $s>k_{d-1}$, search the rightmost child
- That's it in a binary tree; what about if **d>2**?
  - Find two keys k_{i-1} and k_i between which s falls, and search the child v_i.



Performance

- **Worst Case** – O(h logd)

  where d is the maximum number of children of any node
- If d is constant, then running time is O(h)
- **Prime Goal**

  Try to keep h as small as possible

(2, 4) or (2, 3, 4) Trees

(2, 4) tree is a multi-way search tree with the following two properties:

Size property:

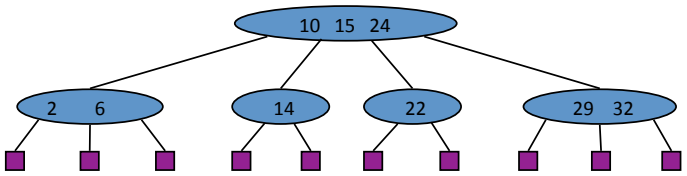
Nodes may contain 1, 2 or 3 items & a node with k items has k + 1 children

Depth property:

All the external nodes have same depth

(2,4) Trees

- Depending on the number of children, an internal node of a (2,4) tree is called a 2-node, 3-node or 4-node



Height of (2, 4) tree

Result:

A multi-way search tree storing n items has n + 1 external nodes.

- Can be proved by induction

Main Result :

Height of a (2, 4) tree storing n items is O(logn)

Proof

Let h be the height of (2, 4) tree T storing n items.

Then the number of external nodes in T is at most 4^h

& the number of external nodes in T is at least 2^h

Therefore,

$2^h \leq n + 1 \leq 4^h$

Taking logarithm in base 2, we get that

$h \leq \log(n+1)$ and $\log(n+1) \leq 2h$, which proves the result.

Insertion in (2, 4) tree

- To insert an item with key k in a (2, 4) tree T
- Assume that tree T has no element with key k.
- Perform search for k.
  This search will terminate at an external node, say z.
- Let v be parent of z
- Insert new into node v and add a new child w(an external node) to v on the left of z.

Observe: Insertion method preserves the depth property.

But it may violate size property.

- If v was previously a 4-node, after insertion will become a 5-node, which is not allowed.
- This violation of size property is called **overflow** at v.
- Overflow needs to be resolved

Insertion in (2, 4) tree

Resolving Overflow

Perform **Split Operation**

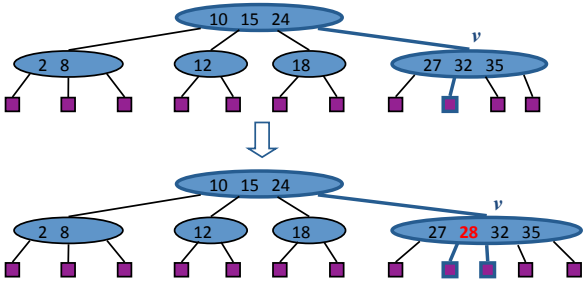
- Let v_1, v_2, v_3 ,v_4 ,v_5 be children of v.
- Let k_1, k_2, k_3, k_4 be the keys stored at v.
- Replace v with two nodes v' and v'', where
- v' is a 3-node with children v_1, v_2, v_3 storing keys k_1, k_2
- v'' is a 2-node with children v_4, v_5 storing key k_4
- If v is a root of T, create a new root u or else u be parent of v.

Insertion in (2, 4) tree

- Insert key $k_3$into u and make v' & v'' children of u, so that if v was ith child of u, then v' & v'' become ith & (i+1)th child of u.
- One split operation takes O(1) time.
- As a consequence of split operation on a node v, a new overflow may occur at the parent u of v.
- In worst case this propagates all the way up to the root, where it is finally resolved.

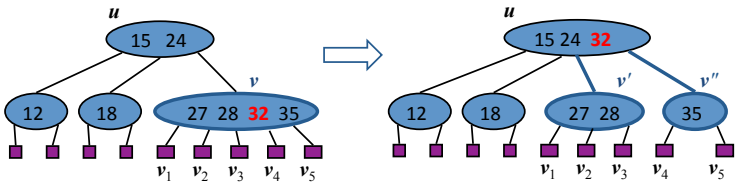
Insertion

- We insert a new item (**k**, **o**) at the parent ***v*** of the leaf reached by searching for **k**
  - We preserve the depth property but
  - We may cause an **overflow** (i.e., node ***v*** may become a 5-node)
- Example: inserting key 28 causes an overflow



Overflow and Split

- We handle an **overflow** at a 5-node ***v*** with a **split operation:**
- The overflow may propagate to the parent node ***u***



Analysis of Insertion

Algorithm insert(k, o)

1. We search for key **k** to locate the insertion node **v**
2. We add the new entry **(k, o)** at node **v**
3. **while** ***overflow***(***v***)
   if ***isRoot***(***v***)
   create a new empty root above ***v***
   v ← ***split***(***v***)

- Let ***T*** be a (2,4) tree with ***n*** items
  - Tree ***T*** has ***O***(log ***n***) height
  - Step 1 takes ***O***(log ***n***) time because we visit ***O***(log ***n***) nodes
  - Step 2 takes ***O***(1) time
  - Step 3 takes ***O***(log ***n***) time because each split takes ***O***(1) time and we perform ***O***(log ***n***) splits
- Thus, an insertion in a (2,4) tree takes ***O*(log *n*)** time

Deletion in (2, 4) tree

To remove an item with key k from a (2, 4) tree T.

- Perform search in T for an item with key k.

Key Point:

Removing item can always be reduced to the case where the item to be removed is stored at a node v whose children are external nodes. (similar to BST)

- Suppose, item k is stored in the ith item at a node z that has only internal node children.
- Find the leftmost internal node v in the subtree rooted at (i+1)th child of z, such that children of v are all external nodes.
- Swap the item to be deleted from z with the first item of v.

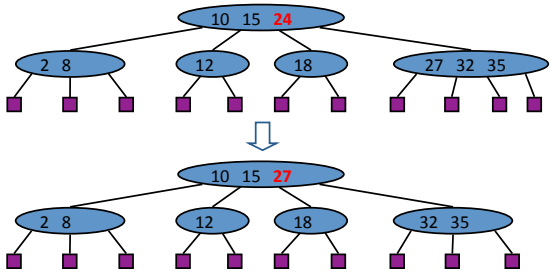
Deletion in (2, 4) tree

- Remove the item k from v and remove the ith external node of v.
- Removal preserves the depth property.
- But size property may get violated

  A 2-node may become a 1-node, which is not allowed.
- This is called **underflow**.
- Underflow needs to be resolved.

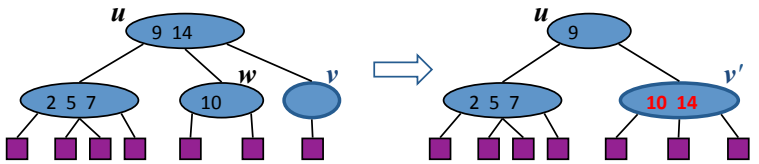
Deletion

- **Example:** to delete key 24, we replace it with 27 (inorder successor**)**



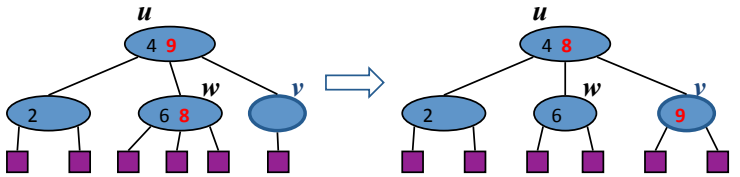
Underflow and Fusion

- Deleting an entry from a node ***v*** may cause an **underflow**, where node ***v*** becomes a 1-node with one child and no keys
- To handle an underflow at node ***v*** with parent ***u***, we consider two cases
- **Case 1:** the adjacent siblings of ***v*** are 2-nodes
  - **Fusion operation:** we merge ***v*** with an adjacent sibling ***w*** and move an entry from ***u*** to the merged node ***v′***
  - After a fusion, the underflow may propagate to the parent ***u***



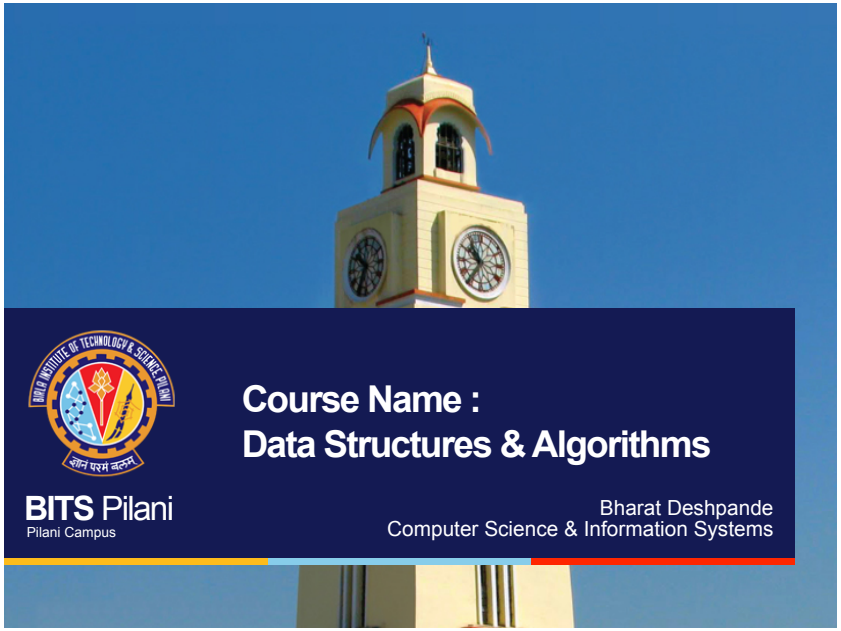
Underflow and Transfer

- **Case 2:** an immediate sibling ***w*** of ***v*** is a 3-node or a 4-node
  - **Transfer operation:**
    1. we move a child of ***w*** to ***v***
    2. we move an item from ***u*** to ***v***
    3. we move an item from ***w*** to ***u***
  - After a transfer, no underflow occurs



Analysis of Deletion

- Let ***T*** be a (2,4) tree with ***n*** items
  - Tree ***T*** has ***O***(log ***n***) height
- In a deletion operation
  - We visit ***O***(log ***n***) nodes to locate the node from which to delete the entry
  - We handle an underflow with a series of ***O***(log ***n***) fusions, followed by at most one transfer
  - Each fusion and transfer takes ***O***(1) time
- Thus, deleting an item from a (2,4) tree takes ***O***(log ***n***) time

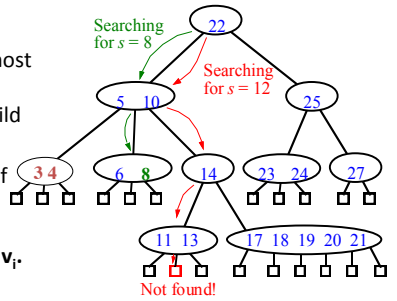


Multi-Way Search Trees

- Each internal node of a multi-way search tree T:
  - has at least two children, i.e., each internal node is a d-node, d>=2
  - stores a collection of items of the form (k, x), where k is a key and x is an element
  - Each d – node with children v1,…,vd contains d - 1 items, stored in increasing order
  - "contains" 2 pseudo-items: $k_0=-\infty$, $k_d=\infty$. For each item (k, x) stored at a node in the subtree of v rooted at vi, I =1,…,D, we have $k_{i-1} \le k \le k_i$
- Children of each internal node are "between" items
- all keys in the subtree rooted at the child fall between keys of those items

Multi-way Searching

- Similar to binary searching
  - If search key $s<k_1$ search the leftmost child
  - If $s>k_{d-1}$, search the rightmost child
- That's it in a binary tree; what about if **d>2**?
  - Find two keys k_{i-1} and k_i between which s falls, and search the child v_i.



Performance

- **Worst Case** – O(h logd)

  where d is the maximum number of children of any node
- If d is constant, then running time is O(h)
- **Prime Goal**

  Try to keep h as small as possible

(2, 4) or (2, 3, 4) Trees

(2, 4) tree is a multi-way search tree with the following two properties:

Size property:

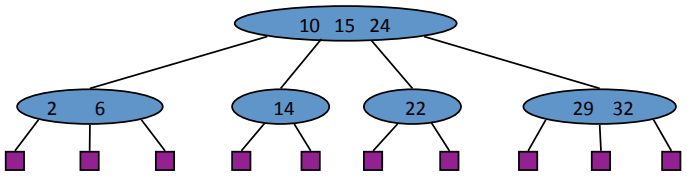
Nodes may contain 1, 2 or 3 items & a node with k items has k + 1 children

Depth property:

All the external nodes have same depth

(2,4) Trees

- Depending on the number of children, an internal node of a (2,4) tree is called a 2-node, 3-node or 4-node



Height of (2, 4) tree

Result:

A multi-way search tree storing n items has n + 1 external nodes.

- Can be proved by induction

Main Result :

Height of a (2, 4) tree storing n items is O(logn)

Proof

Let h be the height of (2, 4) tree T storing n items.

Then the number of external nodes in T is at most 4^h

& the number of external nodes in T is at least 2^h

Therefore,

$2^h \leq n + 1 \leq 4^h$

Taking logarithm in base 2, we get that

$h \leq \log(n+1)$ and $\log(n+1) \leq 2h$, which proves the result.

Insertion in (2, 4) tree

- To insert an item with key k in a (2, 4) tree T
- Assume that tree T has no element with key k.
- Perform search for k.
  This search will terminate at an external node, say z.
- Let v be parent of z
- Insert new into node v and add a new child w(an external node) to v on the left of z.

Observe: Insertion method preserves the depth property.

But it may violate size property.

- If v was previously a 4-node, after insertion will become a 5-node, which is not allowed.
- This violation of size property is called **overflow** at v.
- Overflow needs to be resolved

Insertion in (2, 4) tree

Resolving Overflow

Perform **Split Operation**

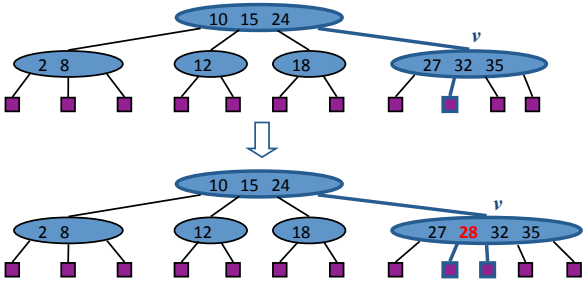
- Let v_1, v_2, v_3 ,v_4 ,v_5 be children of v.
- Let k_1, k_2, k_3, k_4 be the keys stored at v.
- Replace v with two nodes v' and v'', where
- v' is a 3-node with children v_1, v_2, v_3 storing keys k_1, k_2
- v'' is a 2-node with children v_4, v_5 storing key k_4
- If v is a root of T, create a new root u or else u be parent of v.

Insertion in (2, 4) tree

- Insert key $k_3$into u and make v' & v'' children of u, so that if v was ith child of u, then v' & v'' become ith & (i+1)th child of u.
- One split operation takes O(1) time.
- As a consequence of split operation on a node v, a new overflow may occur at the parent u of v.
- In worst case this propagates all the way up to the root, where it is finally resolved.

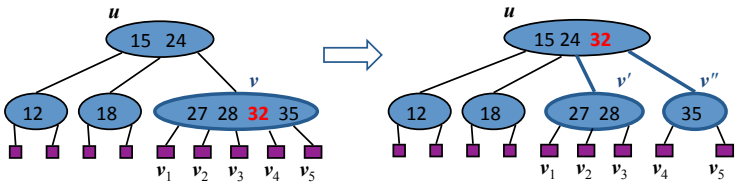
Insertion

- We insert a new item (**k**, **o**) at the parent ***v*** of the leaf reached by searching for **k**
  - We preserve the depth property but
  - We may cause an **overflow** (i.e., node ***v*** may become a 5-node)
- Example: inserting key 28 causes an overflow



Overflow and Split

- We handle an **overflow** at a 5-node ***v*** with a **split operation:**
- The overflow may propagate to the parent node ***u***



Analysis of Insertion

Algorithm insert(k, o)

1. We search for key **k** to locate the insertion node **v**
2. We add the new entry **(k, o)** at node **v**
3. **while** ***overflow***(***v***)
   if ***isRoot***(***v***)
   create a new empty root above ***v***
   v ← ***split***(***v***)

- Let ***T*** be a (2,4) tree with ***n*** items
  - Tree ***T*** has ***O***(log ***n***) height
  - Step 1 takes ***O***(log ***n***) time because we visit ***O***(log ***n***) nodes
  - Step 2 takes ***O***(1) time
  - Step 3 takes ***O***(log ***n***) time because each split takes ***O***(1) time and we perform ***O***(log ***n***) splits
- Thus, an insertion in a (2,4) tree takes ***O*(log *n*)** time

Deletion in (2, 4) tree

To remove an item with key k from a (2, 4) tree T.

- Perform search in T for an item with key k.

Key Point:

Removing item can always be reduced to the case where the item to be removed is stored at a node v whose children are external nodes. (similar to BST)

- Suppose, item k is stored in the ith item at a node z that has only internal node children.
- Find the leftmost internal node v in the subtree rooted at (i+1)th child of z, such that children of v are all external nodes.
- Swap the item to be deleted from z with the first item of v.

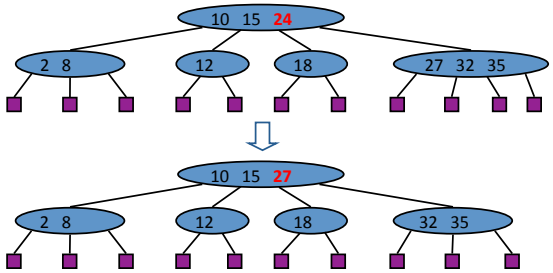
Deletion in (2, 4) tree

- Remove the item k from v and remove the ith external node of v.
- Removal preserves the depth property.
- But size property may get violated

  A 2-node may become a 1-node, which is not allowed.
- This is called **underflow**.
- Underflow needs to be resolved.

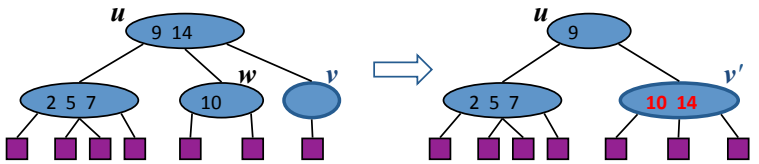
Deletion

- **Example:** to delete key 24, we replace it with 27 (inorder successor**)**



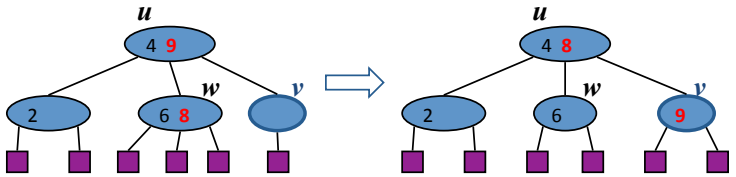
Underflow and Fusion

- Deleting an entry from a node ***v*** may cause an **underflow**, where node ***v*** becomes a 1-node with one child and no keys
- To handle an underflow at node ***v*** with parent ***u***, we consider two cases
- **Case 1:** the adjacent siblings of ***v*** are 2-nodes
  - **Fusion operation:** we merge ***v*** with an adjacent sibling ***w*** and move an entry from ***u*** to the merged node ***v′***
  - After a fusion, the underflow may propagate to the parent ***u***



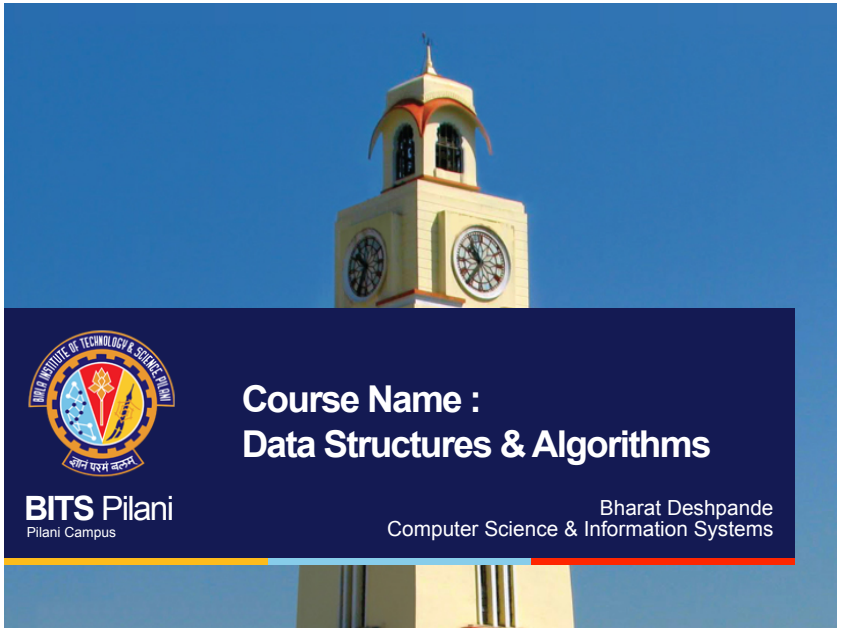
Underflow and Transfer

- **Case 2:** an immediate sibling ***w*** of ***v*** is a 3-node or a 4-node
  - **Transfer operation:**
    1. we move a child of ***w*** to ***v***
    2. we move an item from ***u*** to ***v***
    3. we move an item from ***w*** to ***u***
  - After a transfer, no underflow occurs



Analysis of Deletion

- Let ***T*** be a (2,4) tree with ***n*** items
  - Tree ***T*** has ***O***(log ***n***) height
- In a deletion operation
  - We visit ***O***(log ***n***) nodes to locate the node from which to delete the entry
  - We handle an underflow with a series of ***O***(log ***n***) fusions, followed by at most one transfer
  - Each fusion and transfer takes ***O***(1) time
- Thus, deleting an item from a (2,4) tree takes ***O***(log ***n***) time



(2, 4) or (2, 3, 4) Trees

(2, 4) tree is a multi-way search tree with the following two properties:

Size property:

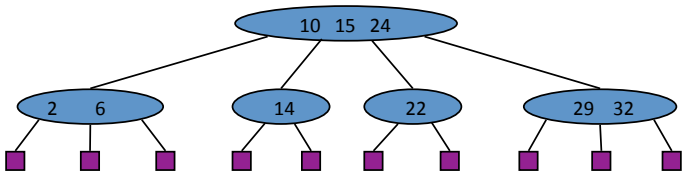
Nodes may contain 1, 2 or 3 items & a node with k items has k + 1 children

Depth property:

All the external nodes have same depth

(2,4) Trees

- Depending on the number of children, an internal node of a (2,4) tree is called a 2-node, 3-node or 4-node



Insertion in (2, 4) tree

- To insert an item with key k in a (2, 4) tree T
- Assume that tree T has no element with key k.
- Perform search for k.
  This search will terminate at an external node, say z.
- Let v be parent of z
- Insert new into node v and add a new child w(an external node) to v on the left of z.

Observe: Insertion method preserves the depth property.

But it may violate size property.

- If v was previously a 4-node, after insertion will become a 5-node, which is not allowed.
- This violation of size property is called **overflow** at v.
- Overflow needs to be resolved

Insertion in (2, 4) tree

Resolving Overflow

Perform **Split Operation**

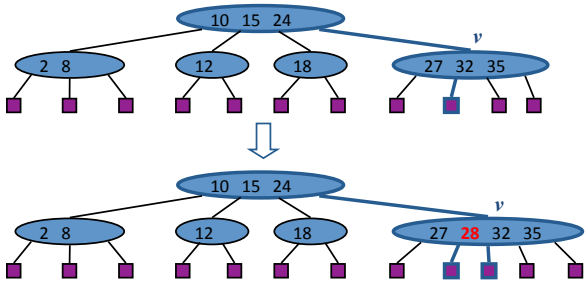
- Let v_1, v_2, v_3, v_4, v_5 be children of v.
- Let k_1, k_2, k_3, k_4 be the keys stored at v.
- Replace v with two nodes v' and v'', where
- v' is a 3-node with children v_1, v_2, v_3 storing keys k_1, k_2
- v'' is a 2-node with children v_4, v_5 storing key k_4
- If v is a root of T, create a new root u or else u be parent of v.

Insertion in (2, 4) tree

- Insert key k_3 into u and make v′ & v″ children of u, so that if v was ith child of u, then v′ & v″ become ith & (i+1)th child of u.
- One split operation takes O(1) time.
- As a consequence of split operation on a node v, a new overflow may occur at the parent u of v.
- In worst case this propagates all the way up to the root, where it is finally resolved.

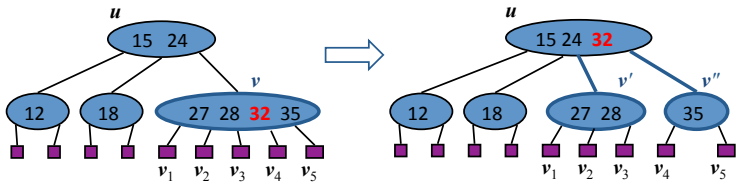
Insertion

- We insert a new item (**k**, **o**) at the parent ***v*** of the leaf reached by searching for **k**
  - We preserve the depth property but
  - We may cause an **overflow** (i.e., node ***v*** may become a 5-node)
- Example: inserting key 28 causes an overflow



Overflow and Split

- We handle an **overflow** at a 5-node ***v*** with a **split operation:**
- The overflow may propagate to the parent node ***u***



Analysis of Insertion

Algorithm insert(k, o)

1. We search for key **k** to locate the insertion node **v**
2. We add the new entry **(k, o)** at node **v**
3. **while** ***overflow***(***v***)
   if ***isRoot***(***v***)
   create a new empty root above ***v***
   v ← ***split***(***v***)

- Let ***T*** be a (2,4) tree with ***n*** items
  - Tree ***T*** has ***O***(log ***n***) height
  - Step 1 takes ***O***(log ***n***) time because we visit ***O***(log ***n***) nodes
  - Step 2 takes ***O***(1) time
  - Step 3 takes ***O***(log ***n***) time because each split takes ***O***(1) time and we perform ***O***(log ***n***) splits
- Thus, an insertion in a (2,4) tree takes ***O(log n)*** time

Deletion in (2, 4) tree

To remove an item with key k from a (2, 4) tree T.

- Perform search in T for an item with key k.

Key Point:

Removing item can always be reduced to the case where the item to be removed is stored at a node v whose children are external nodes. (similar to BST)

- Suppose, item k is stored in the ith item at a node z that has only internal node children.
- Find the leftmost internal node v in the subtree rooted at (i+1)th child of z, such that children of v are all external nodes.
- Swap the item to be deleted from z with the first item of v.

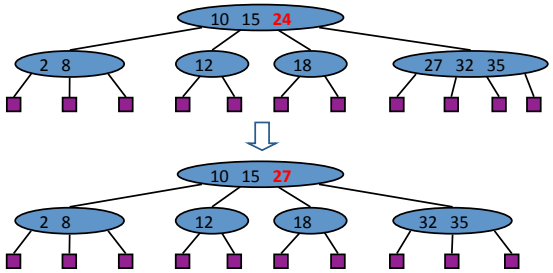
Deletion in (2, 4) tree

- Remove the item k from v and remove the ith external node of v.
- Removal preserves the depth property.
- But size property may get violated

  A 2-node may become a 1-node, which is not allowed.
- This is called **underflow**.
- Underflow needs to be resolved.

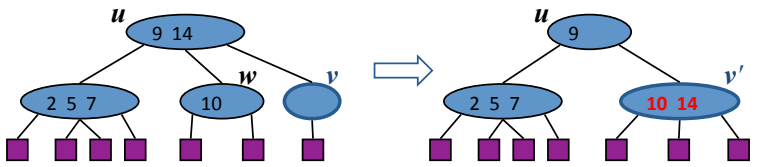
Deletion

- **Example:** to delete key 24, we replace it with 27 (inorder successor)



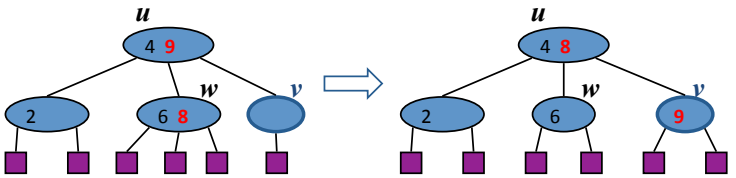
Underflow and Fusion

- Deleting an entry from a node ***v*** may cause an **underflow**, where node ***v*** becomes a 1-node with one child and no keys
- To handle an underflow at node ***v*** with parent ***u***, we consider two cases
- **Case 1:** the adjacent siblings of ***v*** are 2-nodes
  - **Fusion operation:** we merge ***v*** with an adjacent sibling ***w*** and move an entry from ***u*** to the merged node ***v'***
  - After a fusion, the underflow may propagate to the parent ***u***



Underflow and Transfer

- **Case 2:** an immediate sibling ***w*** of ***v*** is a 3-node or a 4-node
  - **Transfer operation:**
    1. we move a child of ***w*** to ***v***
    2. we move an item from ***u*** to ***v***
    3. we move an item from ***w*** to ***u***
  - After a transfer, no underflow occurs



Analysis of Deletion

- Let ***T*** be a (2,4) tree with ***n*** items
  - Tree ***T*** has ***O***(log ***n***) height
- In a deletion operation
  - We visit ***O***(log ***n***) nodes to locate the node from which to delete the entry
  - We handle an underflow with a series of ***O***(log ***n***) fusions, followed by at most one transfer
  - Each fusion and transfer takes ***O***(1) time
- Thus, deleting an item from a (2,4) tree takes ***O***(log ***n***) time

Hash Tables

- **Aim**

To develop a search procedure with running time O(1).
Array is an immediate answer.
Example: Consider a sorted sequence stored in an array.

11	22	28	33	44	49	55	58	66	77	79	88

Focus on finding 44.
Binary search does it in O(logn) time
Can we retrieve faster than binary search?

Hash Tables

If 44 was stored in A[44], search would be done in one step.

- Problem is if keys are very large.

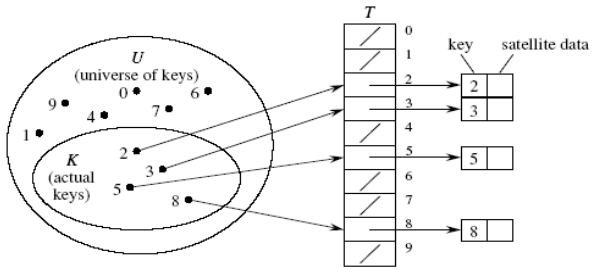
Lot of cells will be empty and so inefficient in terms of space requirement.

- **Answer : Hash Tables**

- Uses an array of size proportional to the number of keys used.
- Instead of using key as an array index, array index is computed from the key.

Array as table

- It is also called Direct-address Hash Table.
  - Each ***slot***, or position, corresponds to a key in U.
  - If there's an element x with key k, then $T[k]$ contains a pointer to x.
  - Otherwise, $T[k]$ is empty, represented by NIL.



Array as table

- Store the records in a huge array where the index corresponds to the key
  - add - very fast *O(1)*
  - delete - very fast *O(1)*
  - search - very fast *O(1)*

Direct Address table

- Disadvantage of Direct Address table
- If the universe U is very large, storing a table of size |U| may be impractical.
- The set K of keys actually stored may be small relative to U, so that most of the space allocated for table is wasted.

Hash function

- In direct addressing, an element with key k is stored in slot k.
- With hashing, the hash function is used to compute the slot h(k)

  $h : U \rightarrow \{0, 1, \ldots, m-1\}$

We say that h(k) is the hash value of k.

- Aim

  To reduce the range of array of indices that needs to be handled

Compression Maps

- Division Method

  h(k) = k mod m

- Certain values of m may not be good:
- Good values for m are prime numbers which are not close to exact powers of 2. For example, if you want to store 2000 elements then m=701 (m = *hash table length*) yields a hash function: h(k) = k mod 701

Compression Maps

- MAD Method (Multiply, Add and Divide)

Define h(k) = (ak +b) mod n,

where n is a prime number and a and b are chosen randomly so that a mod n ≠ 0.

- This function is chosen in order to eliminate repeated patterns.

Hash function

- Problem:

Two keys may hash to the same slot – **collision**

Ideal situation – avoid collision altogether

- But it is generally difficult to design perfect hash. (e.g. when the potential key space is large)

- Solution :

Find effective techniques for resolving collision

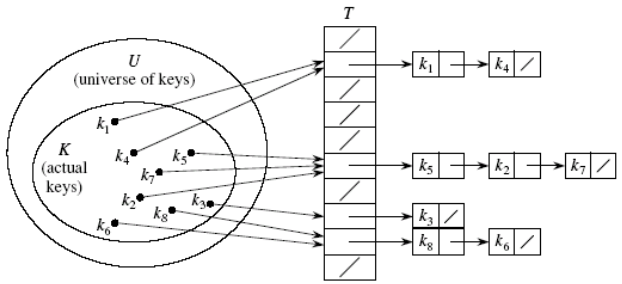
Collision Resolving

- Collision resolution by chaining

In chaining, we put all elements that hash to same slot in a linked list.

- Slot *j* contains a pointer to the head of the list of all stored elements that hash to *j*
- If there are no such elements, slot *j* contains NIL.

Chained Hash Table



Performance Analysis

- Worst Case

All keys are hashed to the same slot, worst cast is O(n) time

- Average Performance

Given a hash table T with m slots that stores n keys,

Define load factor α for T as n/m

- average keys per slot.

Performance Analysis

- **Assume**
  - *Simple uniform hashing*.
    - Any key is equally likely to hash into any of the *m* slots, independent of where any other key hashes to.
  - *O*(1) time to compute *h*(*k*).
- Time to search for an element with key *k* depends linearly on the length of the list *T*[*h*(*k*).
- Expected length of a linked list = load factor = $\alpha = n/m$.
- We consider two cases:

1. Unsuccessful search
2. Successful search

Unsuccessful Search

- **Theorem:**
- An unsuccessful search takes expected time $\Theta(1+\alpha)$.

Proof:

- Any key not already in the table is equally likely to hash to any of the *m* slots.
- To search unsuccessfully for any key *k*, need to search to the end of the list *T*[*h*(*k*)], whose expected length is α.
- Adding the time to compute the hash function, the total time required is $\Theta(1+\alpha)$.

Successful Search

- **Theorem:**
- A successful search takes expected time Θ(1+α).

Proof:

- The probability that a list is searched is proportional to the number of elements it contains.
- Assume that the element being searched for is equally likely to be any of the *n* elements in the table.
- The number of elements examined during a successful search for an element *x* is 1 more than the number of elements examined when the sought for element was inserted.
- The expected length of list to which the ith element is added is (i-1)/m

Expected Cost of a Successful Search

Proof contd.

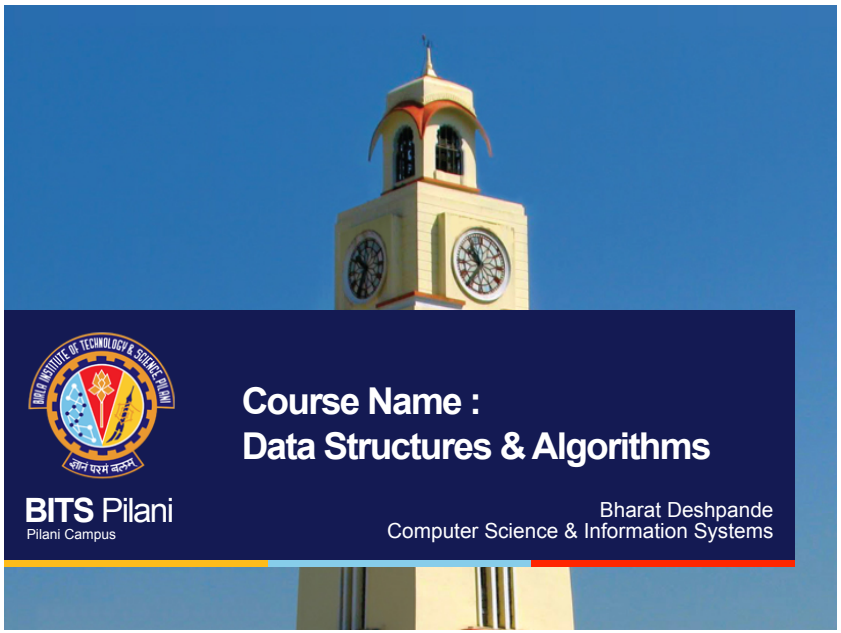
Therefore, the expected no. of elements examined in a successful search is $\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\frac{i-1}{m}\right)\right]$

It can be shown that this is of order Θ(1+α).

Expected Cost – Interpretation

- If $n = O(m)$, then $\alpha = n/m = O(m)/m = O(1)$.

  ⇒ Searching takes constant time on average.
- Insertion is $O(1)$ in the worst case.
- Deletion takes $O(1)$ worst-case time when lists are doubly linked.
- Hence, all dictionary operations take $O(1)$ time on average with hash tables with chaining.



Hash Tables

- **Aim**

To develop a search procedure with running time O(1).
Array is an immediate answer.
Example: Consider a sorted sequence stored in an array.

11	22	28	33	44	49	55	58	66	77	79	88

Focus on finding 44.
Binary search does it in O(logn) time
Can we retrieve faster than binary search?

Hash Tables

If 44 was stored in A[44], search would be done in one step.

- Problem is if keys are very large.

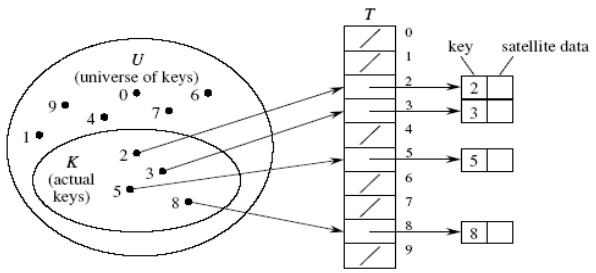
Lot of cells will be empty and so inefficient in terms of space requirement.

- **Answer : Hash Tables**

- Uses an array of size proportional to the number of keys used.
- Instead of using key as an array index, array index is computed from the key.

Array as table

- It is also called Direct-address Hash Table.
  - Each ***slot***, or position, corresponds to a key in U.
  - If there's an element x with key k, then $T[k]$ contains a pointer to x.
  - Otherwise, $T[k]$ is empty, represented by NIL.



Array as table

- Store the records in a huge array where the index corresponds to the key
  - add - very fast *O(1)*
  - delete - very fast *O(1)*
  - search - very fast *O(1)*

Direct Address table

- Disadvantage of Direct Address table

- If the universe U is very large, storing a table of size |U| may be impractical.
- The set K of keys actually stored may be small relative to U, so that most of the space allocated for table is wasted.

Hash function

- In direct addressing, an element with key k is stored in slot k.
- With hashing, the hash function is used to compute the slot h(k)

  $h : U \rightarrow \{0, 1, \ldots, m-1\}$

We say that h(k) is the hash value of k.

- Aim

  To reduce the range of array of indices that needs to be handled

Compression Maps

- Division Method

  $h(k) = k \bmod m$

- Certain values of m may not be good:
- Good values for m are prime numbers which are not close to exact powers of 2. For example, if you want to store 2000 elements then m=701 (m = *hash table length*) yields a hash function: h(k) = k mod 701

Compression Maps

- MAD Method (Multiply, Add and Divide)

Define $h(k) = (ak + b) \bmod n$,

where n is a prime number and a and b are chosen randomly so that $a \bmod n \neq 0$.

- This function is chosen in order to eliminate repeated patterns.

Hash function

- Problem:

Two keys may hash to the same slot – **collision**

Ideal situation – avoid collision altogether

- But it is generally difficult to design perfect hash. (e.g. when the potential key space is large)

- Solution :

Find effective techniques for resolving collision

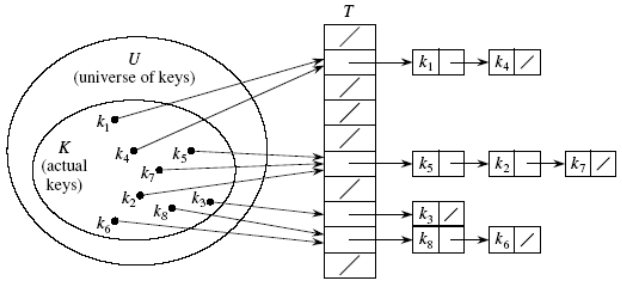
Collision Resolving

- Collision resolution by chaining

In chaining, we put all elements that hash to same slot in a linked list.

- Slot *j* contains a pointer to the head of the list of all stored elements that hash to *j*
- If there are no such elements, slot *j* contains NIL.

Chained Hash Table



Performance Analysis

- Worst Case

All keys are hashed to the same slot, worst cast is O(n) time

- Average Performance

Given a hash table T with m slots that stores n keys,

Define load factor α for T as n/m

- average keys per slot.

Performance Analysis

- **Assume**
  - *Simple uniform hashing*.
    - Any key is equally likely to hash into any of the *m* slots, independent of where any other key hashes to.
  - *O*(1) time to compute *h*(*k*).
- Time to search for an element with key *k* depends linearly on the length of the list *T*[*h*(*k*).
- Expected length of a linked list = load factor = $\alpha = n/m$.
- We consider two cases:

1. Unsuccessful search
2. Successful search

Unsuccessful Search

- **Theorem:**
- An unsuccessful search takes expected time Θ(1+α).

Proof:

- Any key not already in the table is equally likely to hash to any of the *m* slots.
- To search unsuccessfully for any key *k*, need to search to the end of the list *T*[*h*(*k*)], whose expected length is α.
- Adding the time to compute the hash function, the total time required is Θ(1+α).

Successful Search

- **Theorem:**
- A successful search takes expected time Θ(1+α).

Proof:

- The probability that a list is searched is proportional to the number of elements it contains.
- Assume that the element being searched for is equally likely to be any of the *n* elements in the table.
- The number of elements examined during a successful search for an element *x* is 1 more than the number of elements examined when the sought for element was inserted.
- The expected length of list to which the ith element is added is (i-1)/m

Expected Cost of a Successful Search

Proof contd.

Therefore, the expected no. of elements examined in a successful search is $\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\frac{i-1}{m}\right)\right]$

It can be shown that this is of order Θ(1+α).

Expected Cost – Interpretation

- If $n = O(m)$, then $\alpha = n/m = O(m)/m = O(1)$.

  $\Rightarrow$ Searching takes constant time on average.
- Insertion is $O(1)$ in the worst case.
- Deletion takes $O(1)$ worst-case time when lists are doubly linked.
- Hence, all dictionary operations take $O(1)$ time on average with hash tables with chaining.

Open Addressing

An alternative to chaining for handling collisions.

- **Idea:**
  - Store all keys in the hash table itself.
  - Each slot contains either a key or NIL.
  - To ***search*** for key k:
    - Examine slot $h(k)$. Examining a slot is known as a **probe**.
    - If slot $h(k)$ contains key k, the search is successful. If the slot contains NIL, the search is unsuccessful.
    - There's a third possibility: **slot $h(k)$ contains a key that is not k**.
      - Compute the index of some other slot, based on k and which probe we are on.
      - Keep probing until we either find key k or we find a slot holding NIL.
- **Advantages:** Avoids pointers; so can use a larger table.

Probe Sequence

- Sequence of slots examined during a key search constitutes a ***probe sequence***.
- Probe sequence must be a permutation of the slot numbers.
  - We examine every slot in the table, if we have to.
  - We don't examine any slot more than once.
- The hash function is extended to:
  - $h : U \times \{0, 1, \ldots, m-1\} \rightarrow \{0, 1, \ldots, m-1\}$

    probe number — slot number
- $\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$ should be a permutation of $\langle 0, 1, \ldots, m-1 \rangle$.

Insert Operation

Act as though we were searching, and insert at the first NIL slot found

Hash-Insert(T, k)

1. $i \leftarrow 0$
2. **repeat** $j \leftarrow h(k, i)$
3. **if** $T[j]$ = NIL
4. **then** $T[j] \leftarrow k$
5. **return** j
6. **else** $i \leftarrow i + 1$
7. **until** $i = m$
8. **error** "hash table overflow"

Search Operation

The search algorithm for key k probes the same sequence of slots that the insertion algorithm examined when k was inserted.

Hash-Search (T, k)

1. $i \leftarrow 0$
2. **repeat** $j \leftarrow h(k, i)$
3. **if** $T[j] = k$
4. **then return** j
5. $i \leftarrow i + 1$
6. **until** $T[j]$ = NIL **or** $i = m$
7. **return** NIL

Deletion Operation

- Cannot just turn the slot containing the key we want to delete to contain NIL.
- Doing so might make it impossible to retrieve any key k during whose insertion we had probed this slot & found it occupied.
- Use a special value DELETED instead of NIL when marking a slot as empty during deletion.
  - ***Search*** should treat DELETED as though the slot holds a key that does not match the one being searched for.
  - ***Insert*** should treat DELETED as though the slot were empty, so that it can be reused.
- **Disadvantage:** Search time is no longer dependent on α.
  - Hence, chaining is more common when keys have to be deleted.

Probe Sequences

- The ideal situation is ***uniform hashing***:
  - Generalization of simple uniform hashing.
  - Each key is equally likely to have any of the $m!$ permutations of $\langle 0, 1, \ldots, m-1 \rangle$ as its probe sequence.
- It is hard to implement true uniform hashing.
  - Approximate with techniques that at least guarantee that the probe sequence is a permutation of $\langle 0, 1, \ldots, m-1 \rangle$.
- Three commonly used techniques:
  - Linear Probing.
  - Quadratic Probing.
  - Double Hashing.
- All guarantee that $\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$ is a permutation of $\langle 0, 1, \ldots, m-1 \rangle$.
- None of these fulfills assumptions of uniform hashing.
  - Can't produce all $m!$ probe sequences.

Linear Probing

- $h(k, i) = (h'(k)+i) \bmod m$.

key — Probe number — Auxiliary hash function

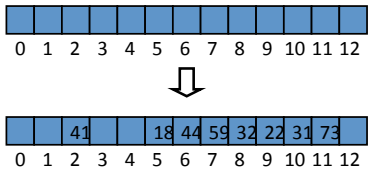
- The initial probe determines the entire probe sequence.
  - $T[h'(k)], T[h'(k)+1], \ldots, T[m-1], T[0], T[1], \ldots, T[h'(k)-1]$
  - Hence, only m distinct probe sequences are possible.
  - Easy to Implement
- Suffers from ***primary clustering***:
  - Long runs of occupied sequences build up.
  - Long runs tend to get longer, since an empty slot preceded by i full slots gets filled next with probability $(i+1)/m$.
  - Hence, average search and insertion times increase.

Linear Probing : Example

Example:

- $h'(x) = x \bmod 13$
- h(x)=(h'(x)+i) mod 13

- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



Quadratic Probing

- $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m \quad c_1 \neq c_2$

key, Probe number, Auxiliary hash function

- The initial probe position is $T[h'(k)]$, later probe positions are offset by amounts that depend on a quadratic function of the probe number *i*.
- Must constrain c_1, c_2, and m to ensure that we get a full permutation of $\langle 0, 1, \ldots, m-1 \rangle$.
- Can suffer from ***secondary clustering***:
  - If two keys have the same initial probe position, then their probe sequences are the same.

Double Hashing

- $h(k, i) = (h_1(k) + i\, h_2(k)) \bmod m$

key, Probe number, Auxiliary hash functions

- Two auxiliary hash functions.
  - h_1 gives the initial probe. h_2 gives the remaining probes.
- Must have $h_2(k)$ relatively prime to m, so that the probe sequence is a full permutation of $\langle 0, 1, \ldots, m-1 \rangle$.
  - Choose m to be a power of 2 and have $h_2(k)$ always return an odd number. Or,
  - Let m be prime, and have $1 < h_2(k) < m$.
- $\Theta(m^2)$ different probe sequences.
  - One for each possible combination of $h_1(k)$ and $h_2(k)$.
  - Close to the ideal uniform hashing.
  - Best method available for open addressing

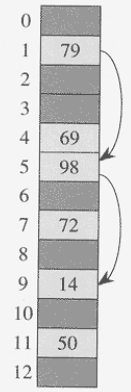


Figure 11.5 Insertion by double hashing. Here we have a hash table of size 13 with $h_1(k) = k \bmod 13$ and $h_2(k) = 1 + (k \bmod 11)$. Since $14 \equiv 1 \pmod{13}$ and $14 \equiv 3 \pmod{11}$, the key 14 is inserted into empty slot 9, after slots 1 and 5 are examined and found to be occupied.

Performance of Open Addressing

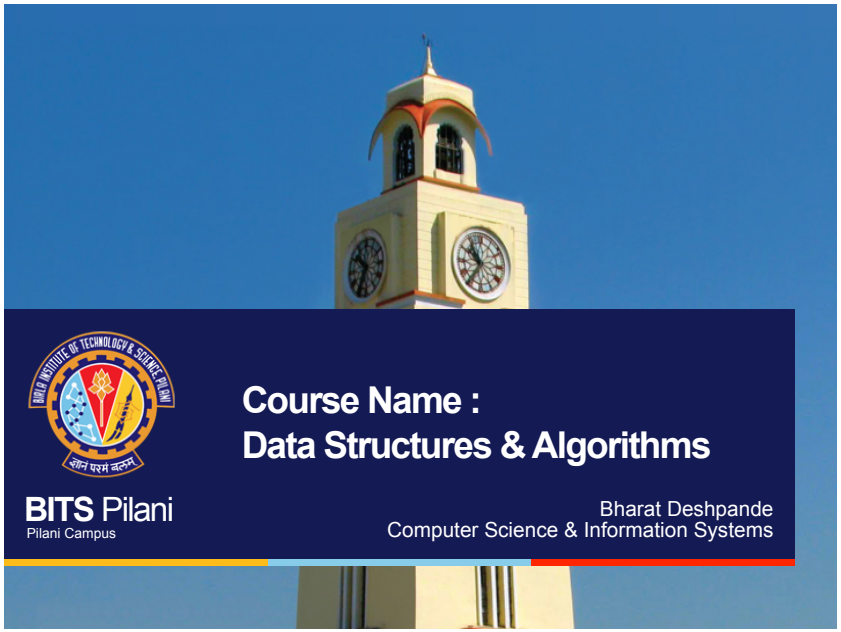
Theorem:

The expected number of probes in an unsuccessful search in an open-address hash table is at most $1/(1-\alpha)$.

Corollary:

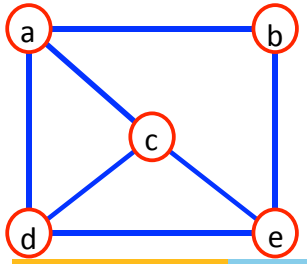
Inserting an element into an open-address table takes $\leq 1/(1-\alpha)$ probes on average.

- If α is a constant, search/insertion takes $O(1)$ time.



What is a Graph?

- A graph G = (V,E) is composed of:
  - V: set of vertices
  - E: set of edges connecting the vertices in V
- An edge e = (u,v) is a pair of vertices
- **Example:**

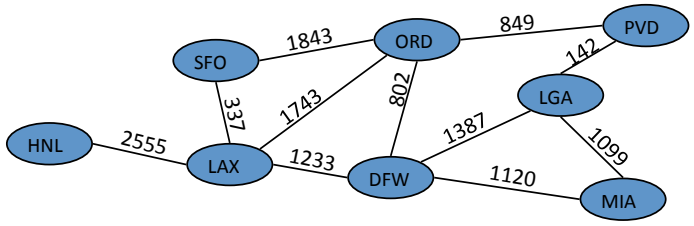


V= {a,b,c,d,e}

E= {(a,b),(a,c),(a,d),
(b,e),(c,d),(c,e),(d,e)}

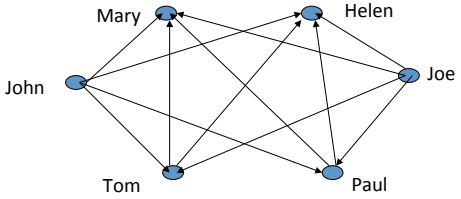
Graph-Example

- Example:
  - A vertex represents an airport and stores the three-letter airport code
  - An edge represents a flight route between two airports and stores the mileage of the route



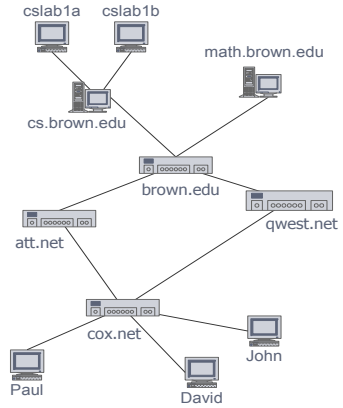
A "Real-life" Example of a Graph

- V=set of 6 people: John, Mary, Joe, Helen, Tom, and Paul, of ages 12, 15, 12, 15, 13, and 13, respectively.
- E ={(x,y) | if x is younger than y}



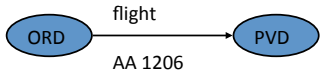
Applications

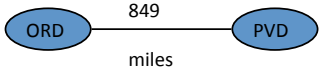
- Electronic circuits
  - Printed circuit board
  - Integrated circuit
- Transportation networks
  - Highway network
  - Flight network
- Computer networks
  - Local area network
  - Internet
  - Web
- Databases
  - Entity-relationship diagram



Edge Types

- Directed edge
  - ordered pair of vertices (u,v)
  - first vertex u is the origin
  - second vertex v is the destination
  - e.g., a flight
- Undirected edge
  - unordered pair of vertices (u,v)
  - e.g., a flight route
- Directed graph (Digraph)
  - all the edges are directed
  - e.g., flight network
- Undirected graph
  - all the edges are undirected
  - e.g., route network





Terminology

- If (v_0, v_1) is an edge in an undirected graph,
  - v_0 and v_1 are adjacent
  - The edge (v_0, v_1) is incident on vertices v_0 and v_1
- If $<v_0, v_1>$ is an edge in a directed graph
  - v_0 is adjacent to v_1, and v_1 is adjacent from v_0
  - The edge $<v_0, v_1>$ is incident on v_0 and v_1

Terminology:

- *Degree of a Vertex*

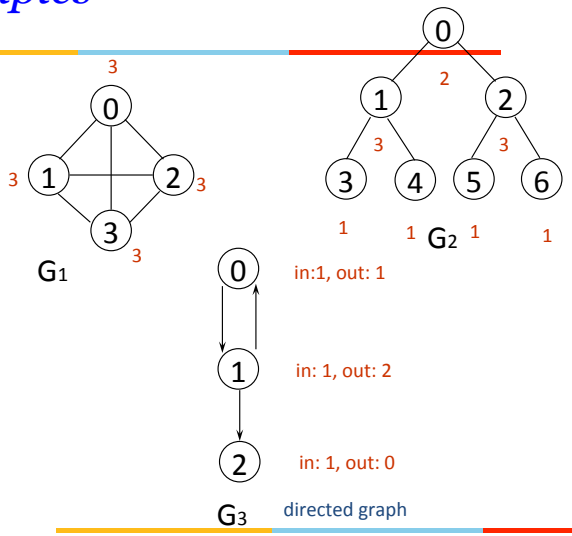
The degree of a vertex is the number of edges incident to that vertex

- For directed graph,
  - the in-degree of a vertex *v* is the number of edges that have *v* as the head
  - the out-degree of a vertex *v* is the number of edges that have *v* as the tail
  - if *di* is the degree of a vertex *i* in a graph *G* with *n* vertices and *e* edges, the number of edges is

$$e = (\sum_{0}^{n-1} d_i) / 2$$

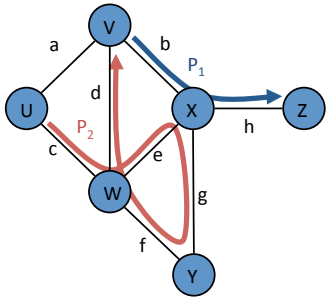
Why? Since adjacent vertices each count the adjoining edge, it will be counted twice

Examples



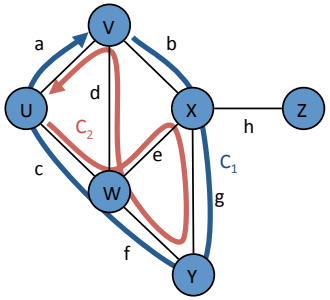
Terminology (cont.)

- **Path**
  - sequence of alternating vertices and edges
  - begins with a vertex
  - ends with a vertex
  - each edge is preceded and followed by its endpoints
- **Simple path**
  - path such that all its vertices and edges are distinct
- **Examples**
  - P_1=(V,b,X,h,Z) is a simple path
  - P_2=(U,c,W,e,X,g,Y,f,W,d,V) is a path that is not simple



Terminology (cont.)

- **Cycle**
  - circular sequence of alternating vertices and edges
  - each edge is preceded and followed by its endpoints
- **Simple cycle**
  - cycle such that all its vertices and edges are distinct
- **Examples**
  - C_1=(V,b,X,g,Y,f,W,c,U,a,↵) is a simple cycle
  - C_2=(U,c,W,e,X,g,Y,f,W,d,V,a,↵) is a cycle that is not simple



Properties

Property 1

$\Sigma_v \deg(v) = 2m$

Proof:

each edge is counted twice

Property 2

In an undirected graph with no self-loops and no multiple edges

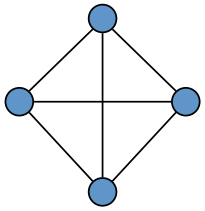
$m \leq n\,(n-1)/2$

Proof:

each vertex has degree at most $(n-1)$

Notation

n	number of vertices
m	number of edges
$\deg(v)$	degree of vertex v

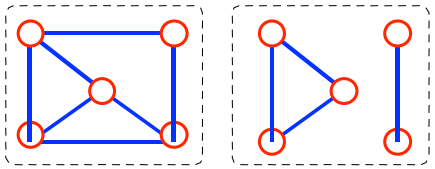


Example

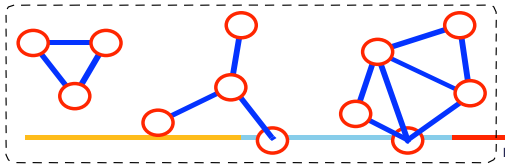
- $n = 4$
- $m = 6$
- $\deg(v) = 3$

Even More Terminology

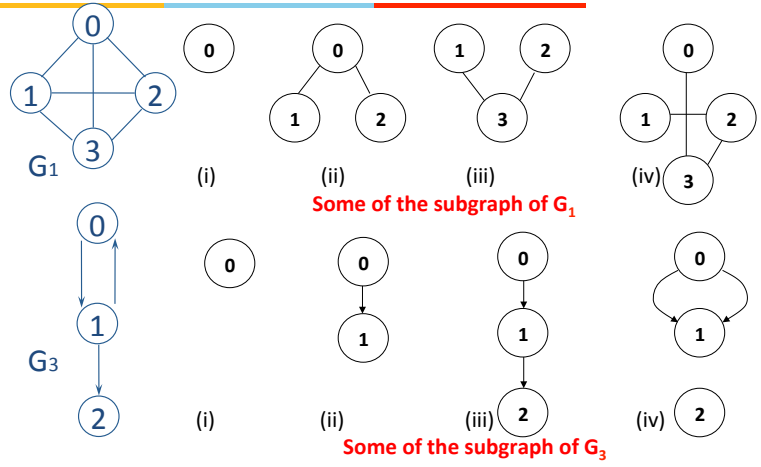
•connected graph: any two vertices are connected by some path



- subgraph: subset of vertices and edges forming a graph
- connected component: maximal connected subgraph. E.g., the graph below has 3 connected components.



Subgraph Examples



Trees

- Tree is a special case of a graph. Each node has zero or more child nodes, which are below it in the tree.
- A tree is a connected acyclic graph
- Already seen.
- Forest - collection of trees

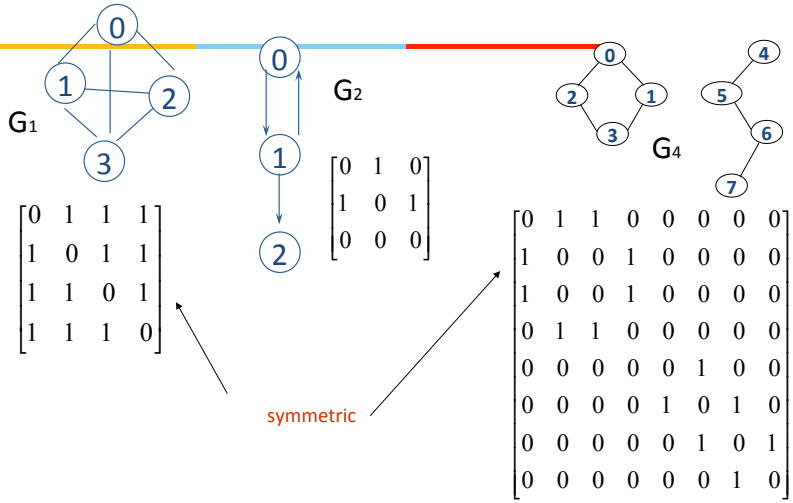
Graph Representations

- Adjacency Matrix
- Adjacency Lists

Adjacency Matrix

- Let G=(V,E) be a graph with n vertices.
- The **adjacency matrix** of G is a two-dimensional n by n array, say adj_mat
- If the edge (v_i, v_j) is in E(G), adj_mat[i][j]=1
- If there is no such edge in E(G), adj_mat[i][j]=0
- The adjacency matrix for an undirected graph is symmetric; the adjacency matrix for a digraph need not be symmetric

Examples for Adjacency Matrix



Merits of Adjacency Matrix

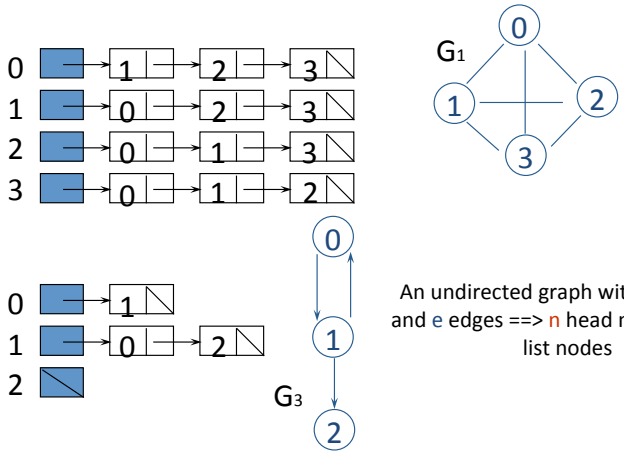
- From the adjacency matrix, to determine the connection of vertices is easy
- The degree of a vertex is $\sum_{j=0}^{n-1} adj\_mat[i][j]$
- For a digraph the row sum is the out_degree, while the column sum is the in_degree

$$ind(vi) = \sum_{j=0}^{n-1} A[j,i] \qquad outd(vi) = \sum_{j=0}^{n-1} A[i,j]$$

Cons : No matter how few edges the graph has, the matrix takes $O(n^2)$ in memory

Adjacency Lists Representation

- A graph of n nodes is represented by a one-dimensional array L of linked lists, where
  - L[i] is the linked list containing all the nodes adjacent from node i.
  - The nodes in the list L[i] are in no particular order



An undirected graph with n vertices and e edges ==> n head nodes and 2e list nodes

Pros and Cons of Adjacency Lists

- Pros
  - Saves on space (memory): the representation takes as many memory words as there are nodes and edge.
- Cons
  - It can take up to O(n) time to determine if a pair of nodes (i,j) is an edge: one would have to search the linked list L[i], which takes time proportional to the length of L[i].

Graph Traversal

- Problem: Search for a certain node or traverse all nodes in the graph.
- Depth First Search
  - Once a possible path is found, continue the search until the end of the path
- Breadth First Search
  - Start several paths at a time, and advance in each one step at a time

Exploring a Labyrinth Without Getting Lost

- A **depth-first search (DFS)** in an undirected graph G is like wandering in a labyrinth with a string and a can of red paint without getting lost.
- We start at vertex ***s***, tying the end of our string to the point and painting ***s*** "visited". Next we label ***s*** as our current vertex called ***u***.
- Now we travel along an arbitrary edge (***u***, ***v***).
- If edge (***u***, ***v***) leads us to an already visited vertex ***v*** we return to ***u***.
- If vertex ***v*** is unvisited, we unroll our string and move to ***v***, paint ***v*** "visited", set ***v*** as our current vertex, and repeat the previous steps.
- The process terminates when our backtracking leads us back to the start index s, and there are no more unexplored edges incident on s.

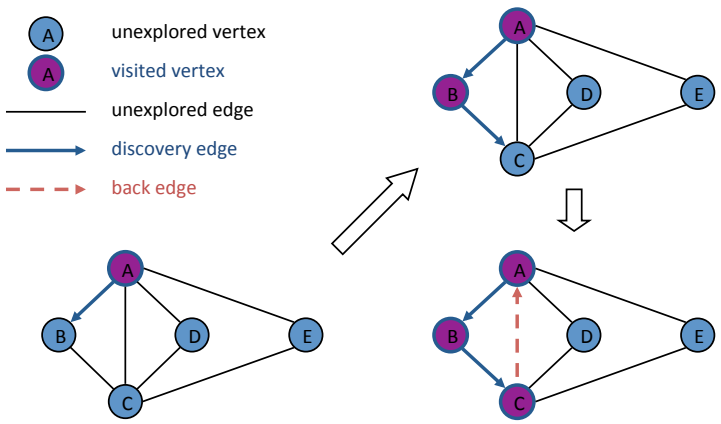
Depth-First Search

Algorithm DFS(v); **Input**: A vertex v in a graph
Output: A labeling of the edges as "discovery" edges and "backedges"
for each edge *e* incident on *v* **do**
if edge *e* is unexplored **then** let *w* be the other endpoint of *e*
if vertex w is unexplored **then** label *e* as a discovery edge
recursively call **DFS(*w*)**
else label *e* as a backedge

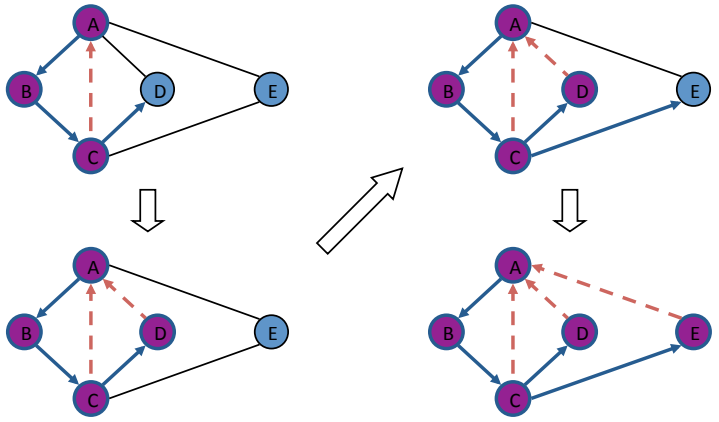
Property 1

DFS*(*G*, *v*)** visits all the vertices and edges in the connected component of ***v

Example



Example (cont.)



Analysis of DFS

- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice
  - once as UNEXPLORED
  - once as VISITED
- Each edge is labeled twice
  - once as UNEXPLORED
  - once as DISCOVERY or BACK
- Method incidentEdges is called once for each vertex
- DFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure

Path Finding

- We can specialize the DFS algorithm to find a path between two given vertices u and z using the template method pattern
- We call $DFS(G, u)$ with u as the start vertex
- We use a stack S to keep track of the path between the start vertex and the current vertex
- As soon as destination vertex z is encountered, we return the path as the contents of the stack

Cycle Finding

- We can specialize the DFS algorithm to find a simple cycle using the template method pattern
- We use a stack S to keep track of the path between the start vertex and the current vertex
- As soon as a back edge (v, w) is encountered, we return the cycle as the portion of the stack from the top to vertex w

Breadth-First Search

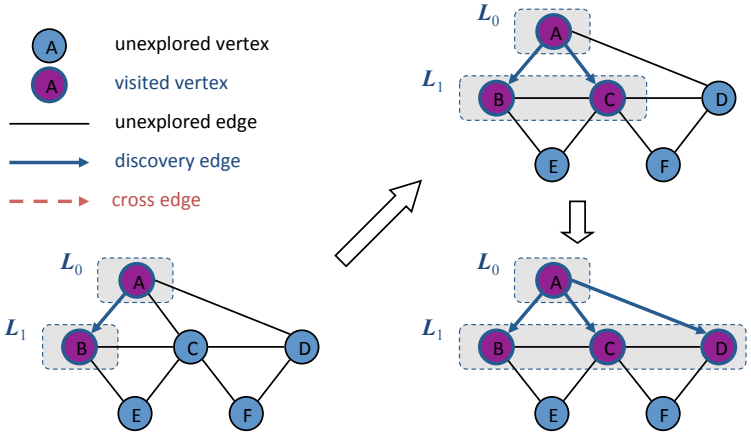
- Instead of going as far as possible, BFS tries to search all paths.
- BFS makes use of a queue to store visited (but not dead) vertices, expanding the path from the earliest visited vertices
- The starting vertex s has level 0, and, as in DFS, defines that point as an "anchor."
- In the first round, the string is unrolled the length of one edge, and all of the nodes that are only one edge away from the anchor are visited.
- These edges are placed into level 1
- In the second round, all the new edges that can be reached by unrolling the string 2 edges are visited and placed in level 2.
- This continues until every vertex has been assigned a level.
- The label of any vertex v corresponds to the length of the shortest path from s to v.

BFS Pseudo-Code

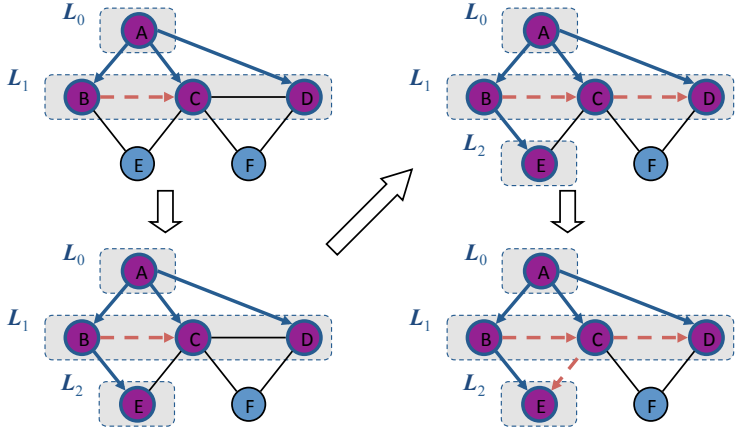
Algorithm BFS(s): Input: A vertex s in a graph
Output: A labeling of the edges as "discovery" edges and "cross edges"

```
initialize container L0 to contain vertex s
i ← 0
while Li is not empty do
        create container Li+1 to initially be empty
        for each vertex v in Li do
          if edge e incident on v do
             let w be the other endpoint of e
             if vertex w is unexplored then
                label e as a discovery edge
             insert w into Li+1
          else label e as a cross edge
             i ← i + 1
```

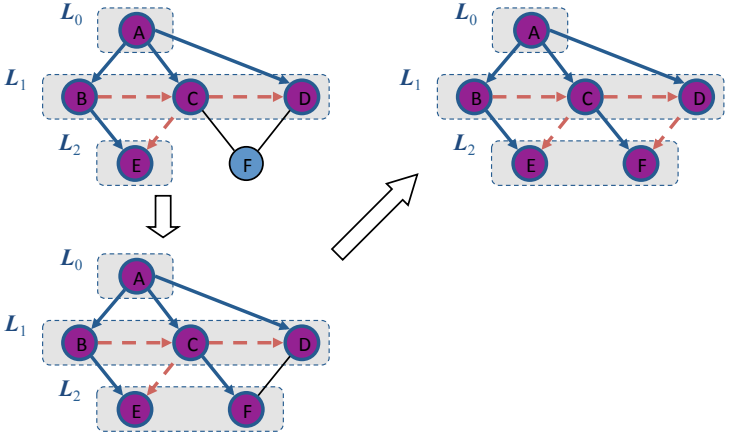

Example



Example (cont.)



Example (cont.)



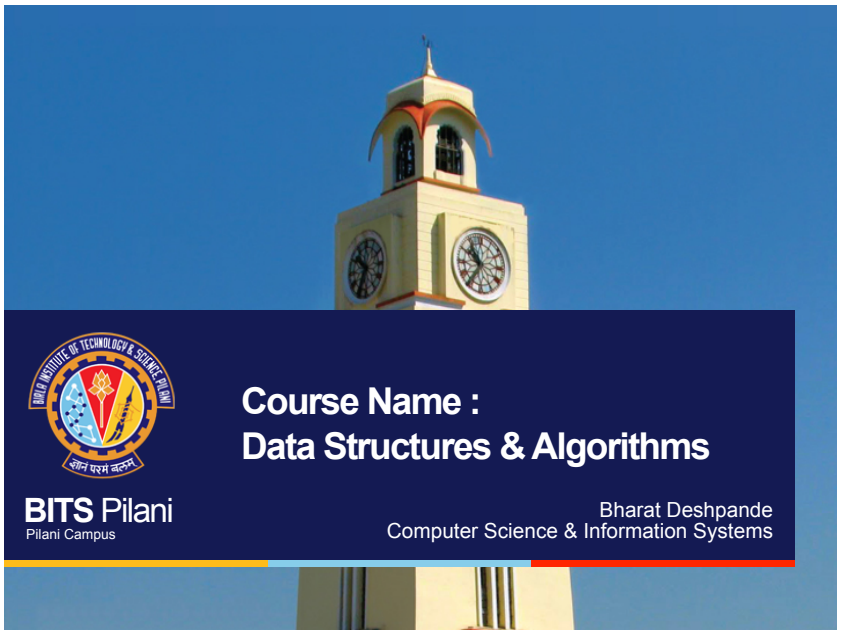
Analysis

- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice
  - once as UNEXPLORED
  - once as VISITED
- Each edge is labeled twice
  - once as UNEXPLORED
  - once as DISCOVERY or CROSS
- Each vertex is inserted once into a sequence L_i
- Method incidentEdges is called once for each vertex
- BFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure

Applications

We can specialize the BFS traversal of a graph G to solve the following problems in $O(n + m)$ time

- Compute the connected components of G
- Find a simple cycle in G, or report that G is a forest
- Given two vertices of G, find a path in G between them with the minimum number of edges, or report that no such path exists



Graph Traversal

- Problem: Search for a certain node or traverse all nodes in the graph.
- Depth First Search
  - Once a possible path is found, continue the search until the end of the path
- Breadth First Search
  - Start several paths at a time, and advance in each one step at a time

Exploring a Labyrinth Without Getting Lost

- A **depth-first search (DFS)** in an undirected graph G is like wandering in a labyrinth with a string and a can of red paint without getting lost.
- We start at vertex ***s***, tying the end of our string to the point and painting ***s*** "visited". Next we label ***s*** as our current vertex called ***u***.
- Now we travel along an arbitrary edge (***u***, ***v***).
- If edge (***u***, ***v***) leads us to an already visited vertex ***v*** we return to ***u***.
- If vertex ***v*** is unvisited, we unroll our string and move to ***v***, paint ***v*** "visited", set ***v*** as our current vertex, and repeat the previous steps.
- The process terminates when our backtracking leads us back to the start index s, and there are no more unexplored edges incident on s.

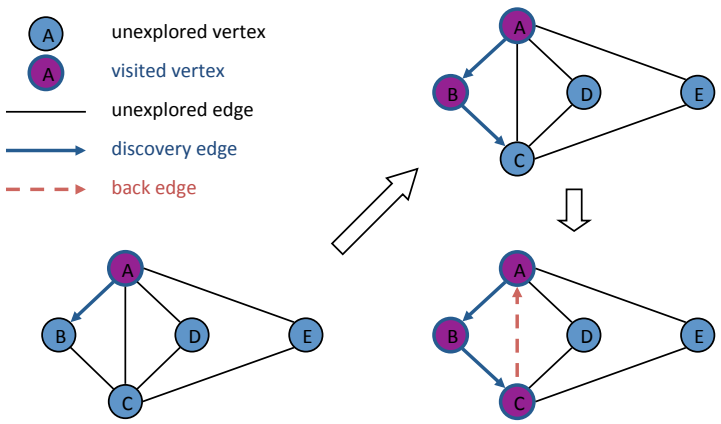
Depth-First Search

Algorithm DFS(v); **Input**: A vertex v in a graph
Output: A labeling of the edges as "discovery" edges and "backedges"
for each edge *e* incident on *v* **do**
if edge *e* is unexplored **then** let *w* be the other endpoint of *e*
if vertex w is unexplored **then** label *e* as a discovery edge
recursively call **DFS(*w*)**
else label *e* as a backedge

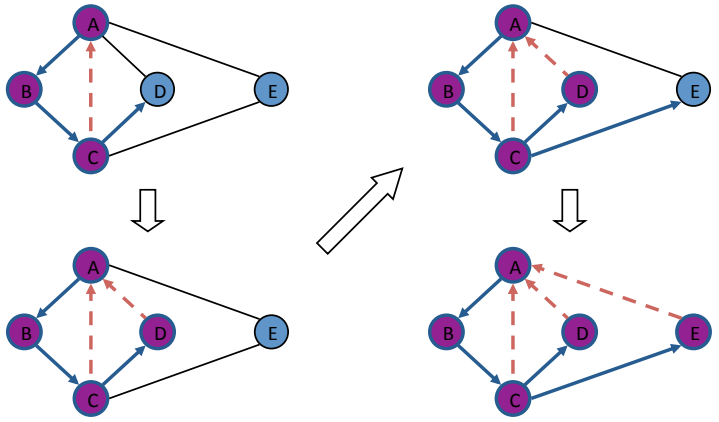
Property 1

DFS*(*G*, *v*)** visits all the vertices and edges in the connected component of ***v

Example



Example (cont.)



Analysis of DFS

- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice
  - once as UNEXPLORED
  - once as VISITED
- Each edge is labeled twice
  - once as UNEXPLORED
  - once as DISCOVERY or BACK
- Method incidentEdges is called once for each vertex
- DFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure

Path Finding

- We can specialize the DFS algorithm to find a path between two given vertices u and z using the template method pattern
- We call $DFS(G, u)$ with u as the start vertex
- We use a stack S to keep track of the path between the start vertex and the current vertex
- As soon as destination vertex z is encountered, we return the path as the contents of the stack

Cycle Finding

- We can specialize the DFS algorithm to find a simple cycle using the template method pattern
- We use a stack S to keep track of the path between the start vertex and the current vertex
- As soon as a back edge (v, w) is encountered, we return the cycle as the portion of the stack from the top to vertex w

Breadth-First Search

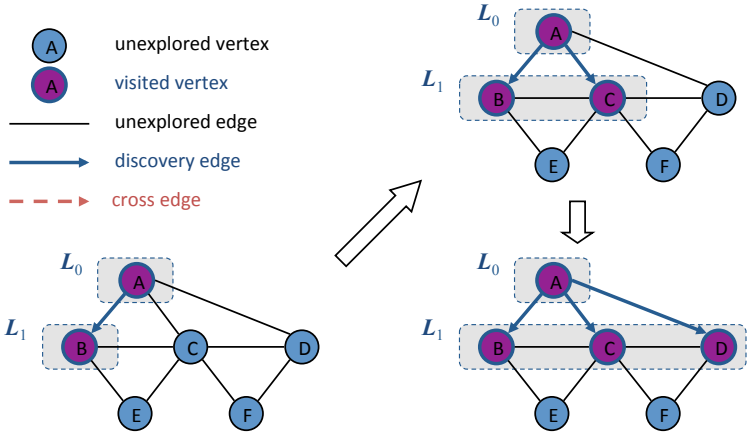
- Instead of going as far as possible, BFS tries to search all paths.
- BFS makes use of a queue to store visited (but not dead) vertices, expanding the path from the earliest visited vertices
- The starting vertex s has level 0, and, as in DFS, defines that point as an "anchor."
- In the first round, the string is unrolled the length of one edge, and all of the nodes that are only one edge away from the anchor are visited.
- These edges are placed into level 1
- In the second round, all the new edges that can be reached by unrolling the string 2 edges are visited and placed in level 2.
- This continues until every vertex has been assigned a level.
- The label of any vertex v corresponds to the length of the shortest path from s to v.

BFS Pseudo-Code

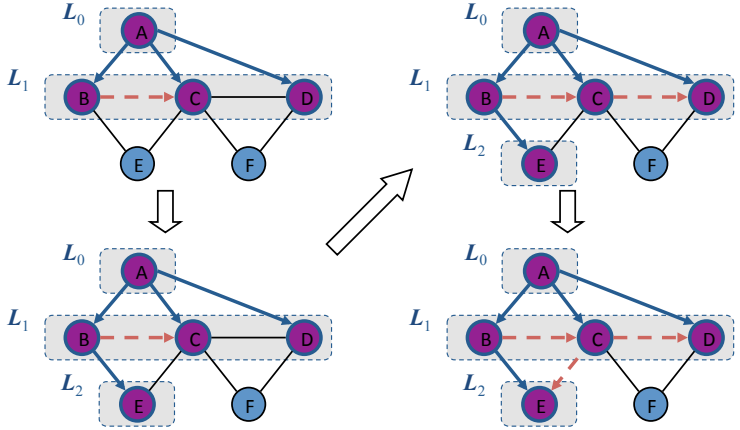
Algorithm BFS(s): Input: A vertex s in a graph
Output: A labeling of the edges as "discovery" edges and "cross edges"

```
initialize container L0 to contain vertex s
i ← 0
while Li is not empty do
        create container Li+1 to initially be empty
        for each vertex v in Li do
          if edge e incident on v do
             let w be the other endpoint of e
             if vertex w is unexplored then
                label e as a discovery edge
             insert w into Li+1
          else label e as a cross edge
             i ← i + 1
```

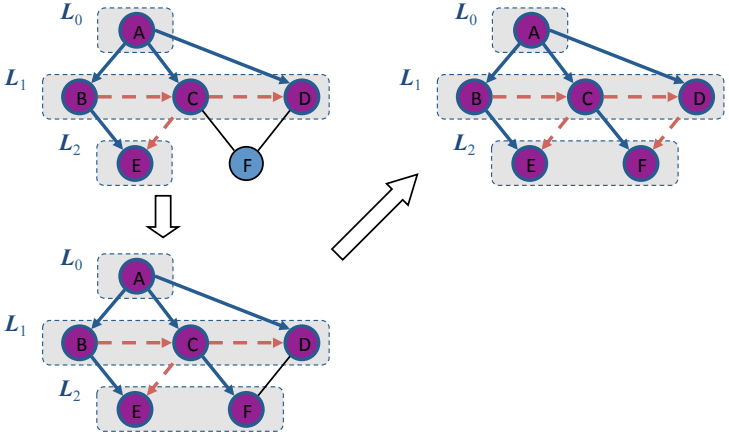

Example



Example (cont.)



Example (cont.)



Analysis

- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice
  - once as UNEXPLORED
  - once as VISITED
- Each edge is labeled twice
  - once as UNEXPLORED
  - once as DISCOVERY or CROSS
- Each vertex is inserted once into a sequence L_i
- Method incidentEdges is called once for each vertex
- BFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure

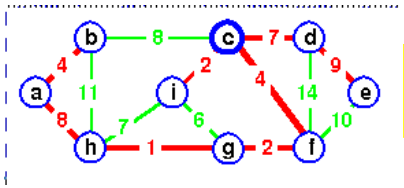
Applications

We can specialize the BFS traversal of a graph G to solve the following problems in $O(n + m)$ time

- Compute the connected components of G
- Find a simple cycle in G, or report that G is a forest
- Given two vertices of G, find a path in G between them with the minimum number of edges, or report that no such path exists

Graphs - Definitions

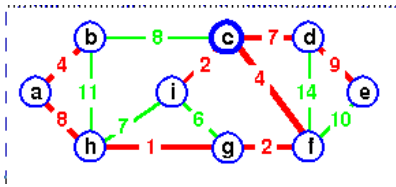
- Spanning Tree
  - A spanning tree is a set of $|V|$-1 edges that connect all the vertices of a graph



The red path connects all vertices, so it's a spanning tree

Graphs - Definitions

- **Minimum Spanning Tree**
  - Generally there is more than one spanning tree
  - If a cost c_{ij} is associated with edge $e_{ij} = (v_i, v_j)$ then the minimum spanning tree is the set of edges E_{span} such that $C = \Sigma (c_{ij} \mid \forall e_{ij} \in E_{span})$ is a minimum



Other ST's can be formed ..
- **Replace 2 with 7**
- **Replace 4 with 11**

The red tree is the Min ST

Growing a MST

- The procedure manages a set A that is always a subset of some MST.
- At each step, an edge (u, v) is added to A such that A ∪ {(u, v)} is also a subset of a MST.
- Edge (u, v) is called a safe edge.
- GENERIC-MST(G)

  A ← ϕ

  while A does not form a spanning tree

  do find an edge (u, v) that is safe for A

  A ← A ∪ {(u, v)}

  return A

- We design two MST algorithms, each of which uses a specific rule to determine the safe edge to be added.

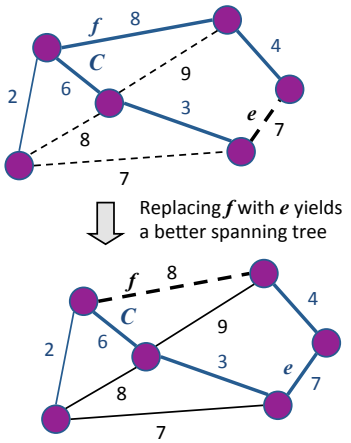
Cycle Property

Cycle Property:

- Let T be a minimum spanning tree of a weighted graph G
- Let e be an edge of G that is not in T and C let be the cycle formed by e with T
- For every edge f of C, $weight(f) \leq weight(e)$

Proof:

- By contradiction
- If $weight(f) > weight(e)$ we can get a spanning tree of smaller weight by replacing e with f



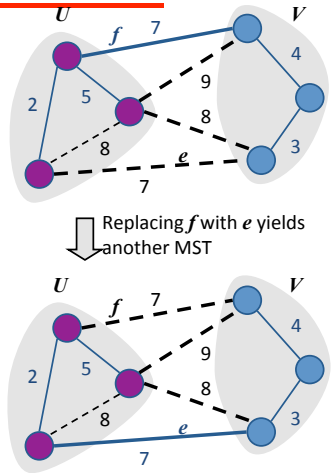
Partition Property

Partition Property:

- Consider a partition of the vertices of G into subsets U and V
- Let e be an edge of minimum weight across the partition
- There is a minimum spanning tree of G containing edge e

Proof:

- Let T be an MST of G
- If T does not contain e, consider the cycle C formed by e with T and let f be an edge of C across the partition
- By the cycle property, $weight(f) \leq weight(e)$
- Thus, $weight(f) = weight(e)$
- We obtain another MST by replacing f with e

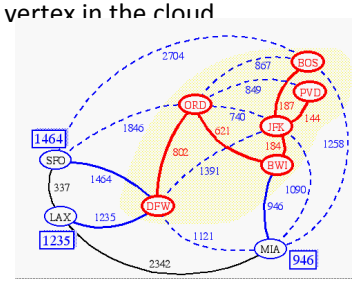


Prim-Jarnik's Algorithm

- Initially discovered in 1930 by Vojtěch Jarník, then rediscovered in 1957 by Robert C. Prim
- We pick an arbitrary vertex s and we grow the MST as a cloud of vertices, starting from s
- We store with each vertex v a label $d(v)$ = the smallest weight of an edge connecting v to a vertex in the cloud

At each step:

- We add to the cloud the vertex u outside the cloud with the smallest distance label
- We update the labels of the vertices adjacent to u

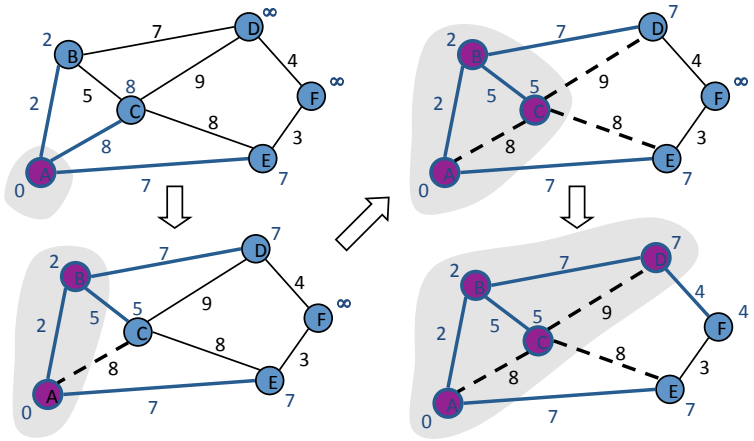


Prim's Algorithm – Pseudo Code

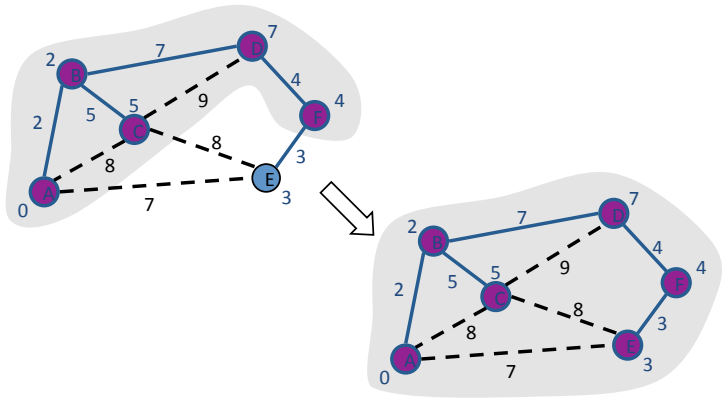
```
MST-PRIM(G, r)
Q ← V[G]  // Q – priority queue initialize
for each u ∈ Q
    do d[u] ← ∞
D[r] = 0
∏[r] ← NIL
while Q ≠ ϕ
   do u ← EXTRACT-MIN(Q)
      for each v ∈ Adj[u]
          do if v ∈ Q and w(u, v) < d[v]
             then ∏[v] ← u
                  d[v] ← w(u,v)
```

The **running time** is $O(m \log n)$

Example



Example (contd.)



Kruskal's Algorithm

- Created in 1957 by Joseph Kruskal

Algorithm

- Starts by choosing an edge in the graph with minimum weight.
- Successively add edges with minimum weight that do not form a cycle with those edges already chosen.
- Terminates after n-1 edges have been selected (n is the number of vertices)

Note

In Kruskal's algorithm the set A is a forest.

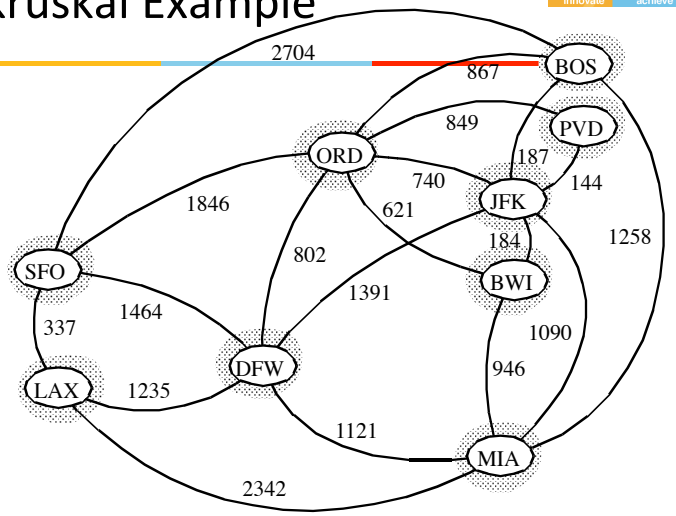
The safe edge added to A is always a least-weight edge in the graph that connects two distinct components.

Kruskal's algorithm – Pseudo code

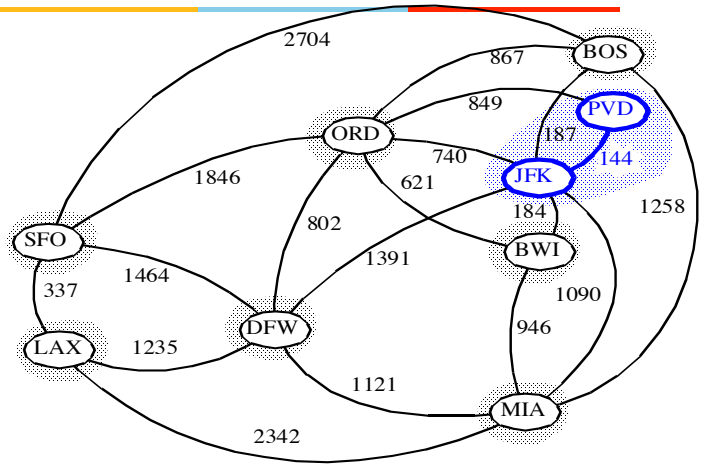
Algorithm Kruskal(*G*)
Input: A weighted graph ***G***.
Output: An MST ***T*** for ***G***.
Let ***P*** be a partition of the vertices of ***G***, where each vertex forms a separate set.
Let ***Q*** be a priority queue storing the edges of ***G***, sorted by their weights
Let ***T*** be an initially-empty tree
while ***Q*** is not empty **do**
  (***u***,***v***) ← ***Q***.removeMinElement()
  if ***P***.find(***u***) != ***P***.find(***v***) **then**
    Add (***u***,***v***) to ***T***
    P.union(***u***,***v***)
return ***T***

Running time: O(m logm)

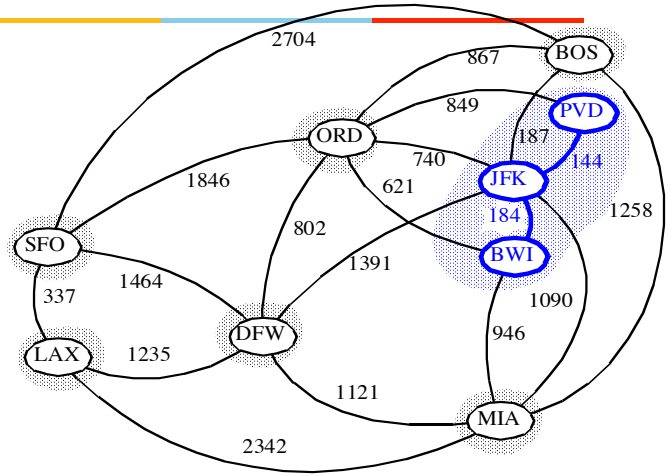
Kruskal Example



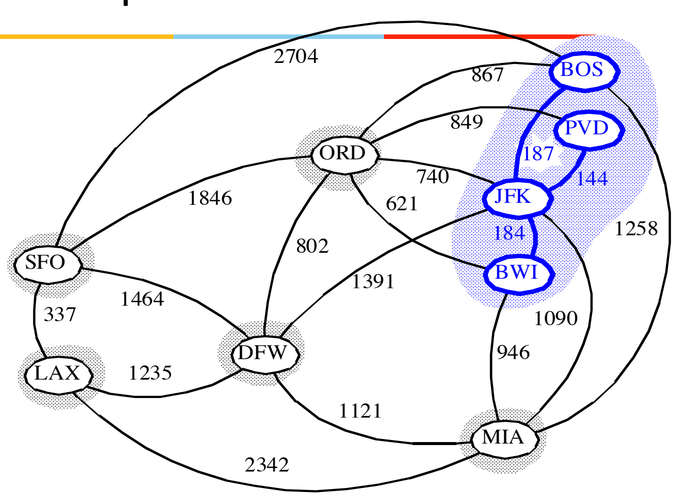
Example



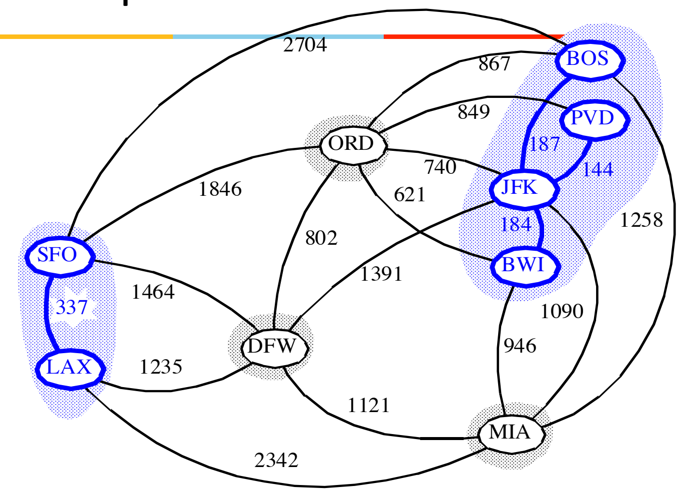
Example



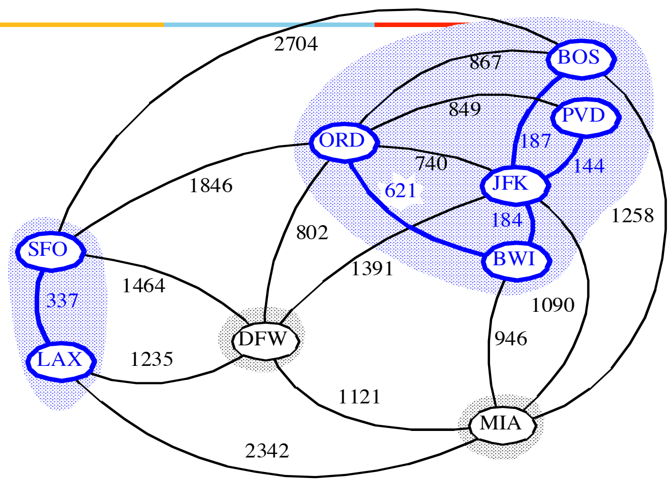
Example



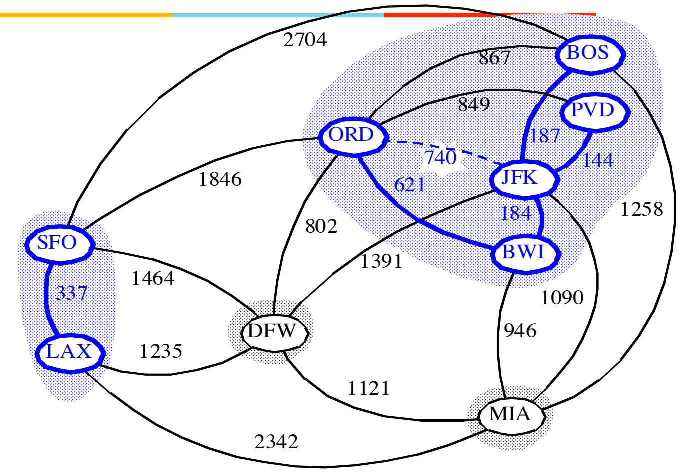
Example



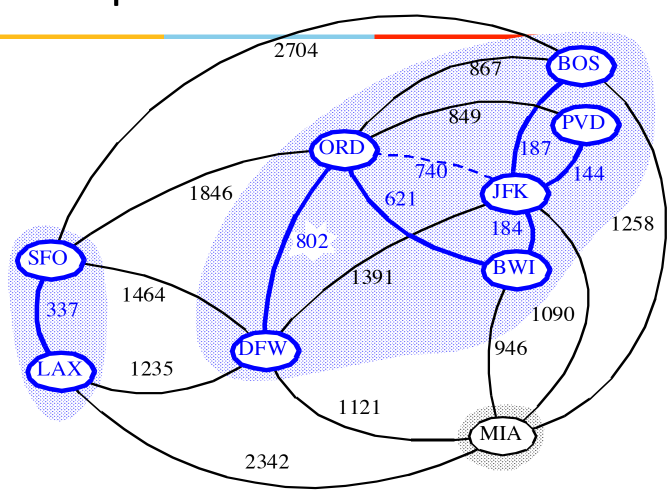
Example



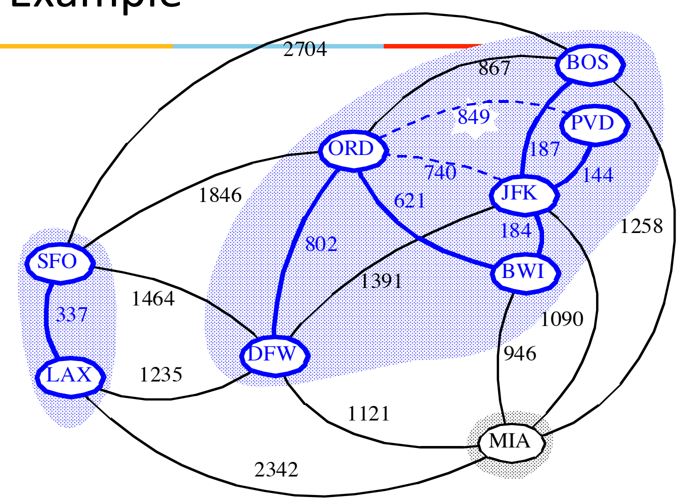
Example



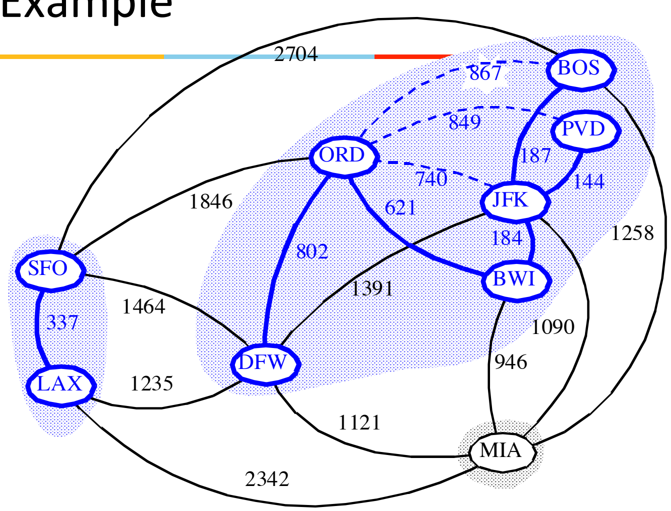
Example



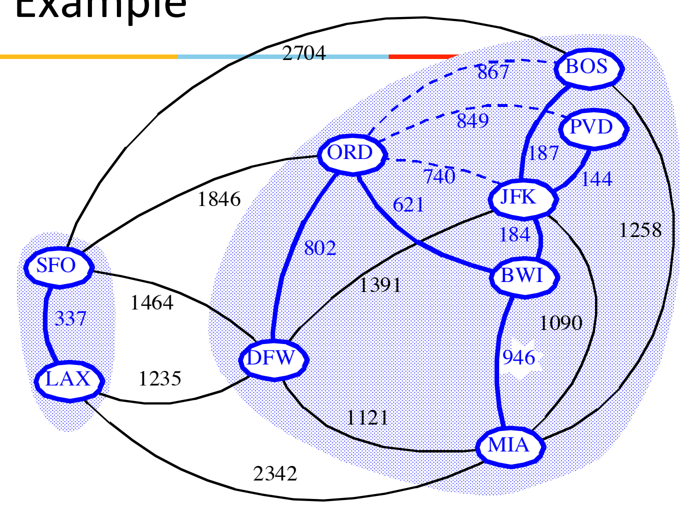
Example



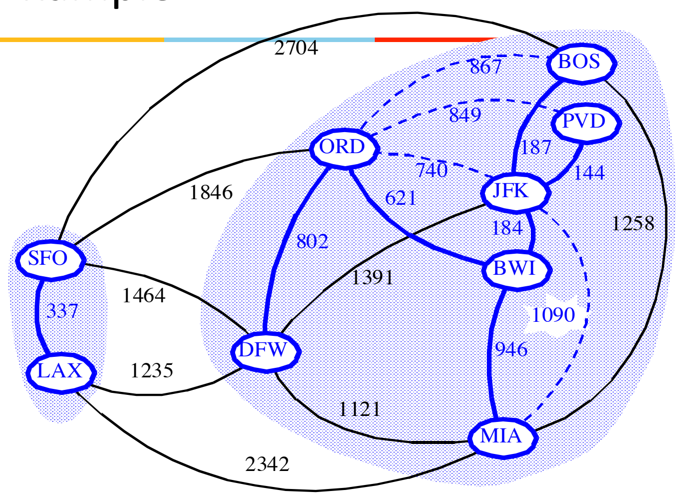
Example



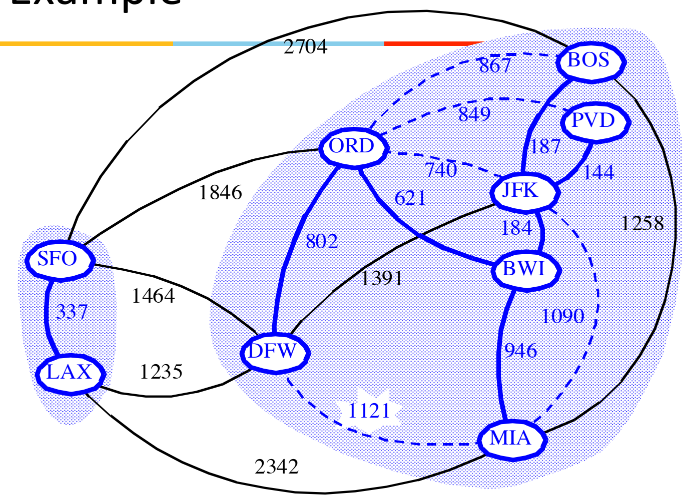
Example



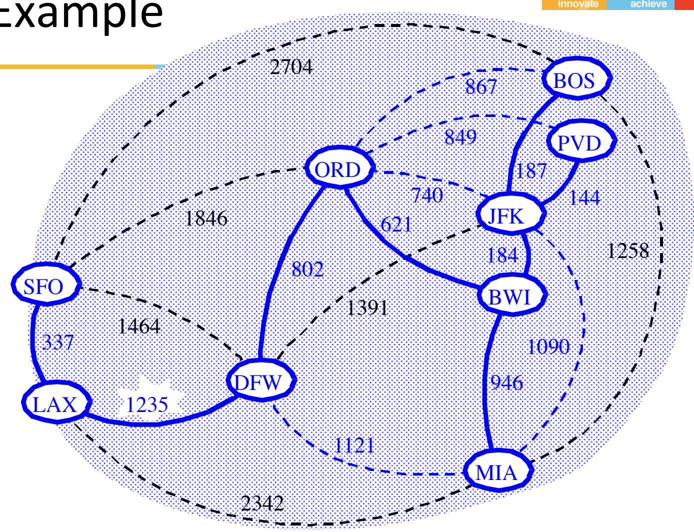
Example

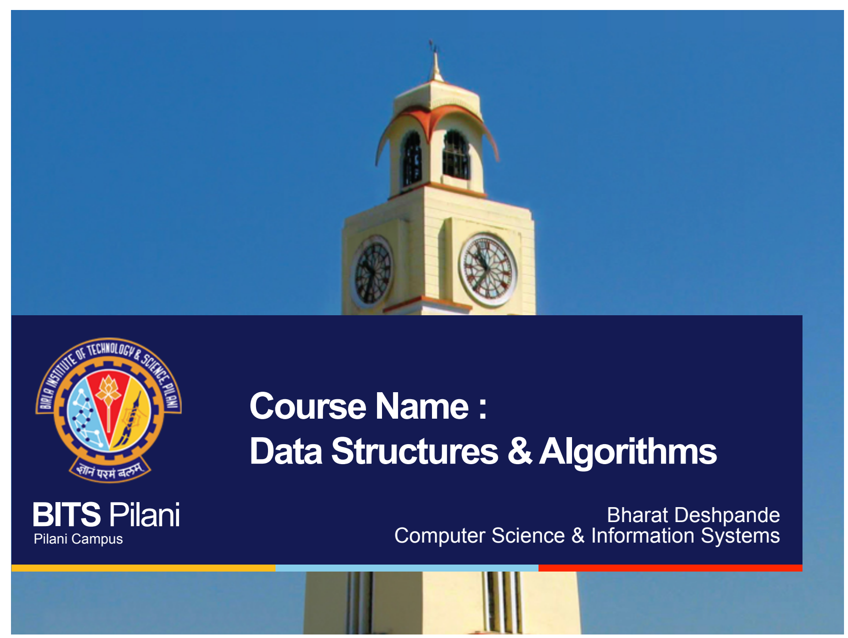


Example



Example





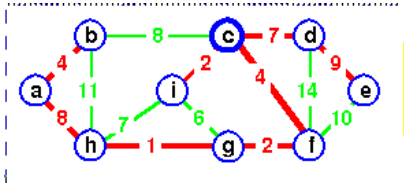
BITS Pilani
Pilani Campus

Course Name :
Data Structures & Algorithms

Bharat Deshpande
Computer Science & Information Systems

Graphs - Definitions

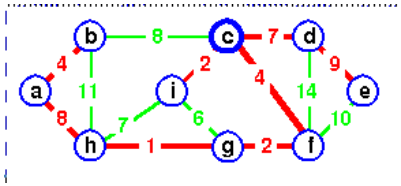
- Spanning Tree
  - A spanning tree is a set of $|V|-1$ edges that connect all the vertices of a graph



The red path connects all vertices, so it's a spanning tree

Graphs - Definitions

- **Minimum Spanning Tree**
  - Generally there is more than one spanning tree
  - If a cost c_{ij} is associated with edge $e_{ij} = (v_i, v_j)$ then the minimum spanning tree is the set of edges E_{span} such that $C = \Sigma (c_{ij} \mid \forall e_{ij} \in E_{span})$ is a minimum



Other ST's can be formed ..
- **Replace 2 with 7**
- **Replace 4 with 11**

The red tree is the Min ST

Growing a MST

- The procedure manages a set A that is always a subset of some MST.
- At each step, an edge (u, v) is added to A such that A ∪ {(u, v)} is also a subset of a MST.
- Edge (u, v) is called a safe edge.
- GENERIC-MST(G)

  A ← ϕ

  while A does not form a spanning tree

  do find an edge (u, v) that is safe for A

  A ← A ∪ {(u, v)}

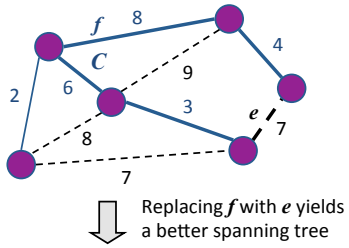
  return A

- We design two MST algorithms, each of which uses a specific rule to determine the safe edge to be added.

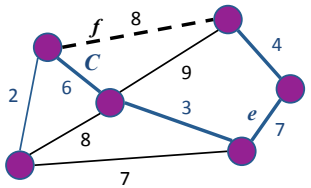
Cycle Property

Cycle Property:
- Let T be a minimum spanning tree of a weighted graph G
- Let e be an edge of G that is not in T and C let be the cycle formed by e with T
- For every edge f of C, $weight(f) \leq weight(e)$

Proof:
- By contradiction
- If $weight(f) > weight(e)$ we can get a spanning tree of smaller weight by replacing e with f



Replacing f with e yields a better spanning tree



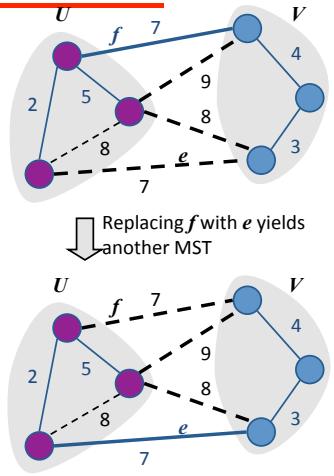
Partition Property

Partition Property:

- Consider a partition of the vertices of G into subsets U and V
- Let e be an edge of minimum weight across the partition
- There is a minimum spanning tree of G containing edge e

Proof:

- Let T be an MST of G
- If T does not contain e, consider the cycle C formed by e with T and let f be an edge of C across the partition
- By the cycle property, $weight(f) \leq weight(e)$
- Thus, $weight(f) = weight(e)$
- We obtain another MST by replacing f with e

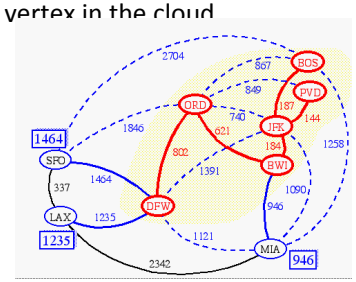


Prim-Jarnik's Algorithm

- Initially discovered in 1930 by Vojtěch Jarník, then rediscovered in 1957 by Robert C. Prim
- We pick an arbitrary vertex s and we grow the MST as a cloud of vertices, starting from s
- We store with each vertex v a label $d(v)$ = the smallest weight of an edge connecting v to a vertex in the cloud

At each step:

- We add to the cloud the vertex u outside the cloud with the smallest distance label
- We update the labels of the vertices adjacent to u

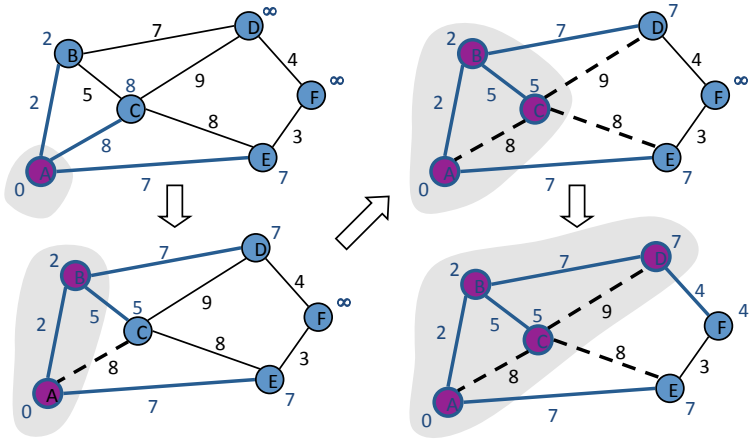


Prim's Algorithm – Pseudo Code

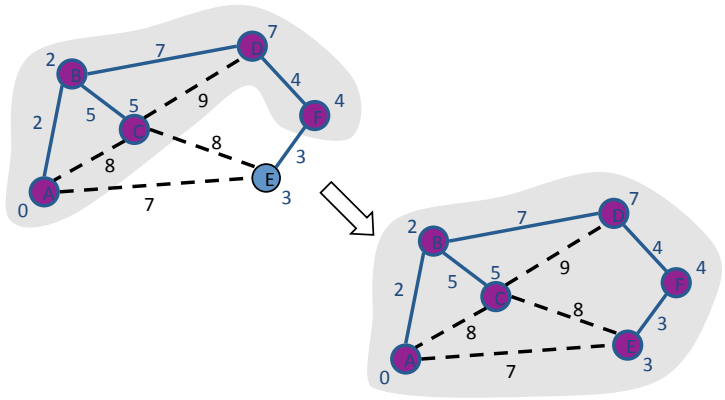
```
MST-PRIM(G, r)
Q ← V[G]  // Q – priority queue initialize
for each u ∈ Q
    do d[u] ← ∞
D[r] = 0
∏[r] ← NIL
while Q ≠ ϕ
   do u ← EXTRACT-MIN(Q)
      for each v ∈ Adj[u]
          do if v ∈ Q and w(u, v) < d[v]
             then ∏[v] ← u
                  d[v] ← w(u,v)
```

The **running time** is $O(m \log n)$

Example



Example (contd.)



Kruskal's Algorithm

- Created in 1957 by Joseph Kruskal

Algorithm

- Starts by choosing an edge in the graph with minimum weight.
- Successively add edges with minimum weight that do not form a cycle with those edges already chosen.
- Terminates after n-1 edges have been selected (n is the number of vertices)

Note

In Kruskal's algorithm the set A is a forest.

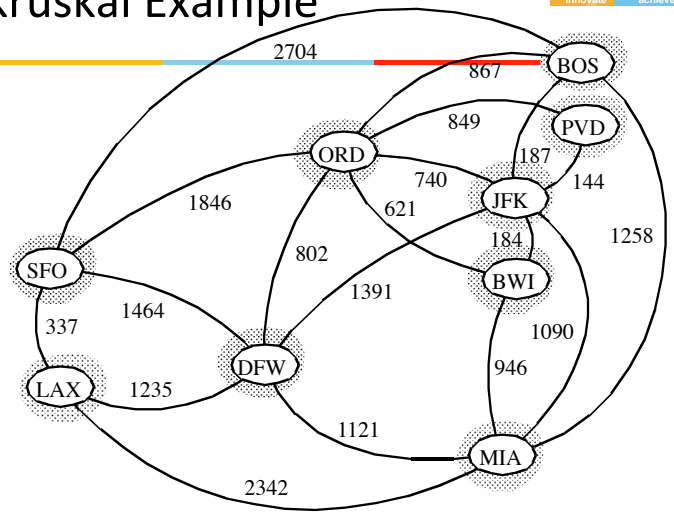
The safe edge added to A is always a least-weight edge in the graph that connects two distinct components.

Kruskal's algorithm – Pseudo code

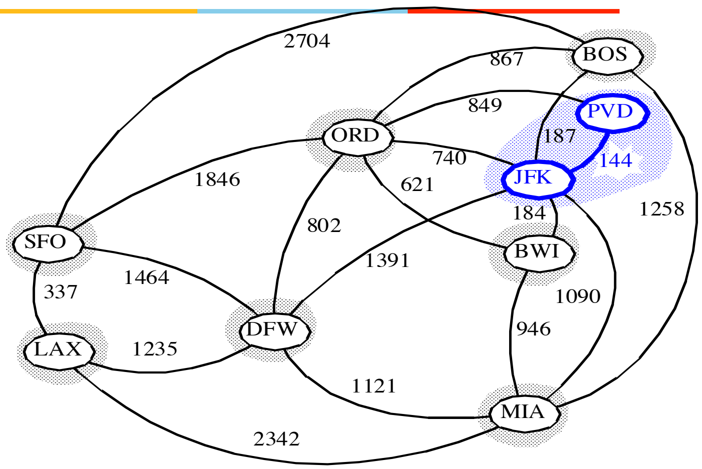
Algorithm Kruskal(*G*)
Input: A weighted graph ***G***.
Output: An MST ***T*** for ***G***.
Let ***P*** be a partition of the vertices of ***G***, where each vertex forms a separate set.
Let ***Q*** be a priority queue storing the edges of ***G***, sorted by their weights
Let ***T*** be an initially-empty tree
while ***Q*** is not empty **do**
  (***u***,***v***) ← ***Q***.removeMinElement()
  if ***P***.find(***u***) != ***P***.find(***v***) **then**
    Add (***u***,***v***) to ***T***
    P.union(***u***,***v***)
return ***T***

Running time: O(m logm)

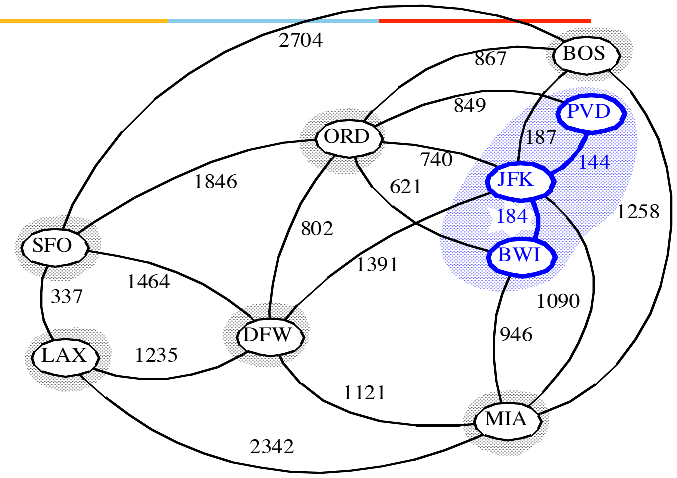
Kruskal Example



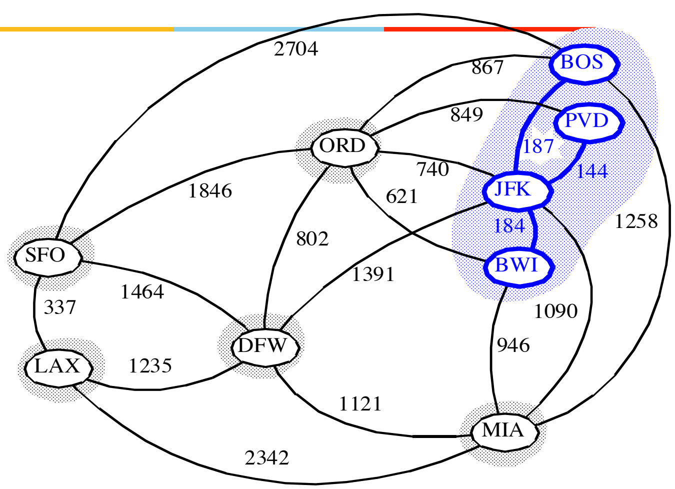
Example



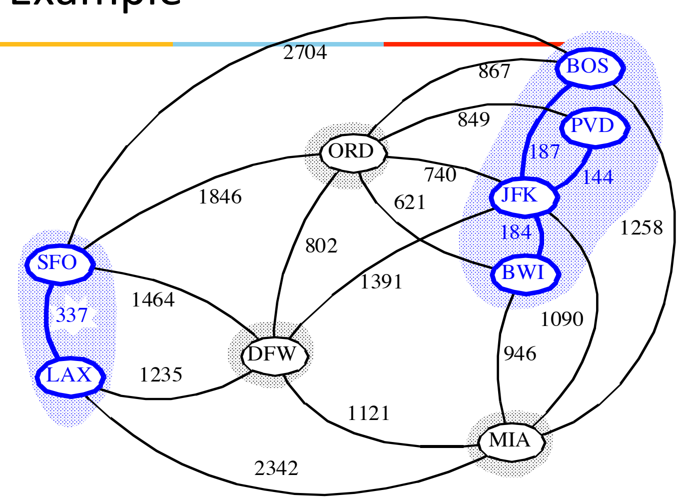
Example



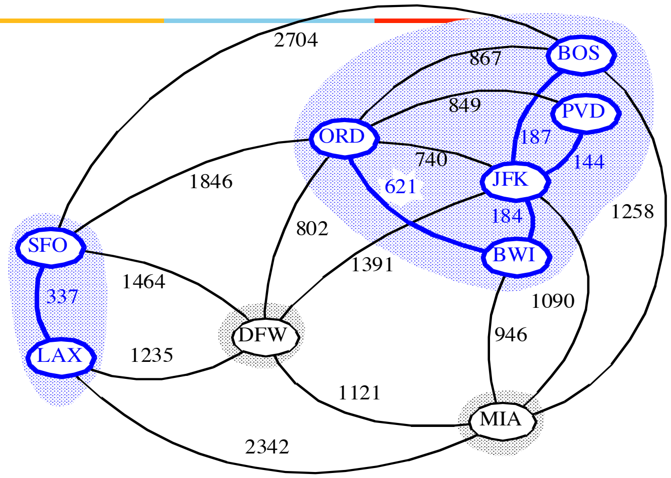
Example



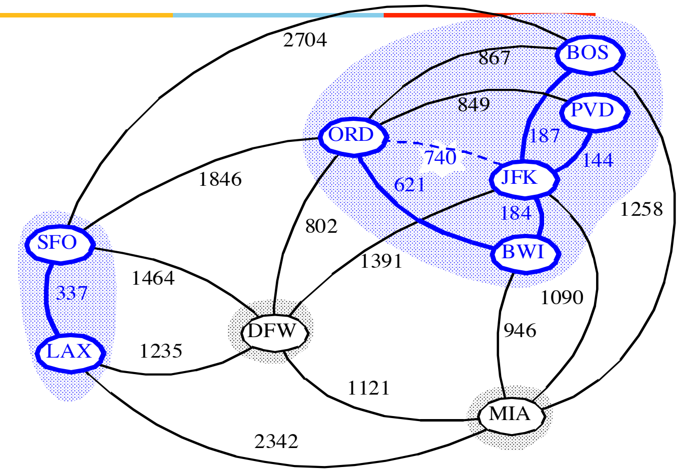
Example



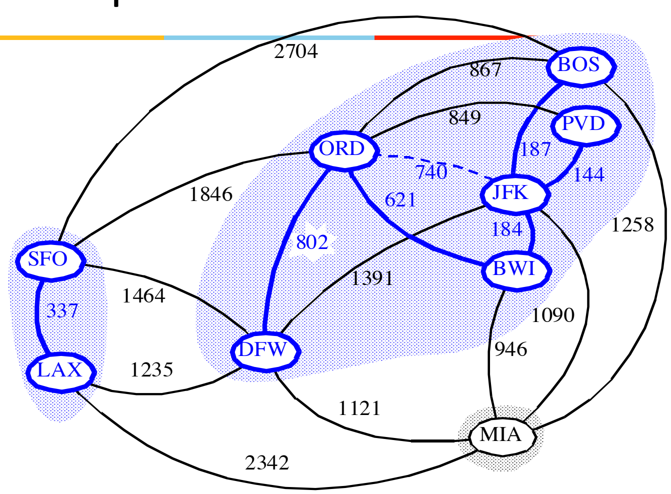
Example



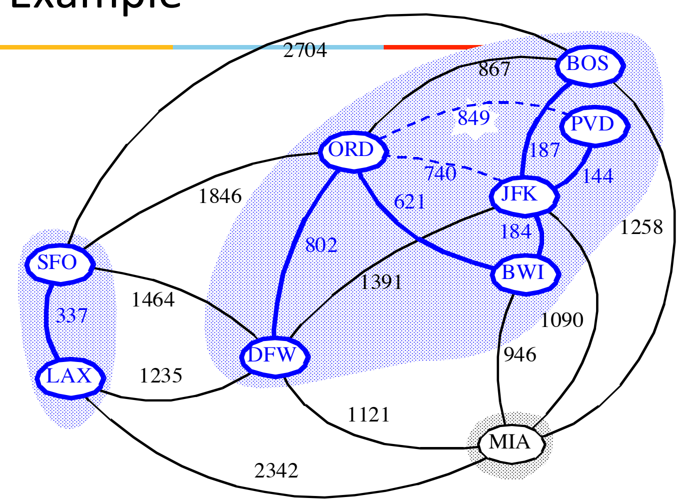
Example



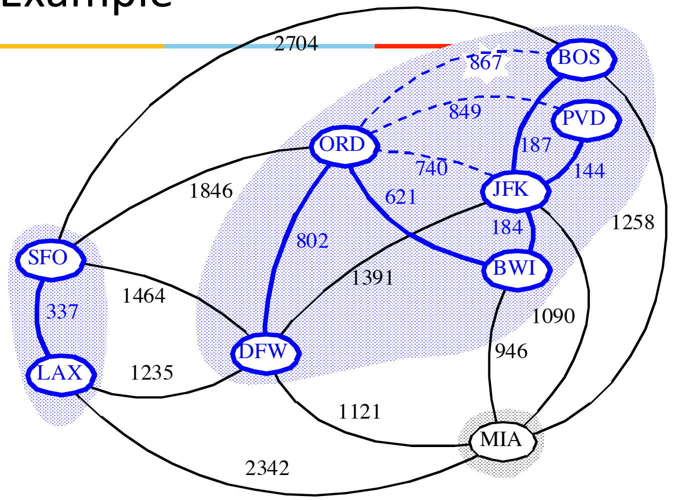
Example



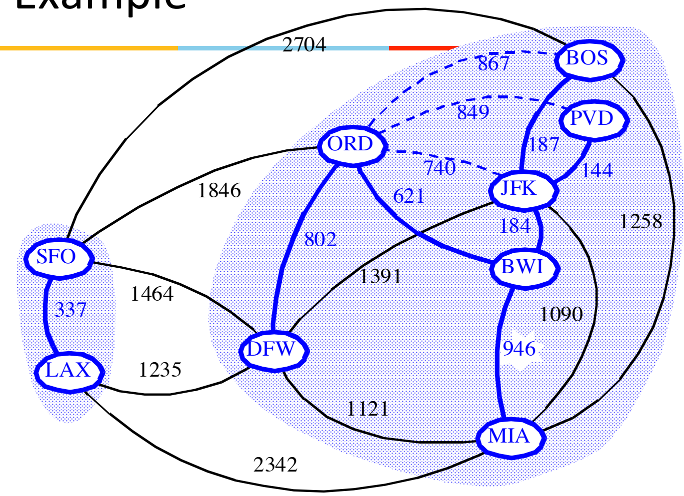
Example



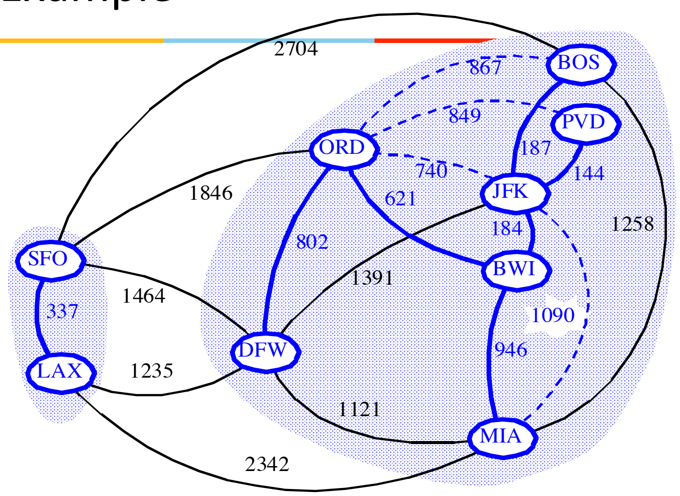
Example



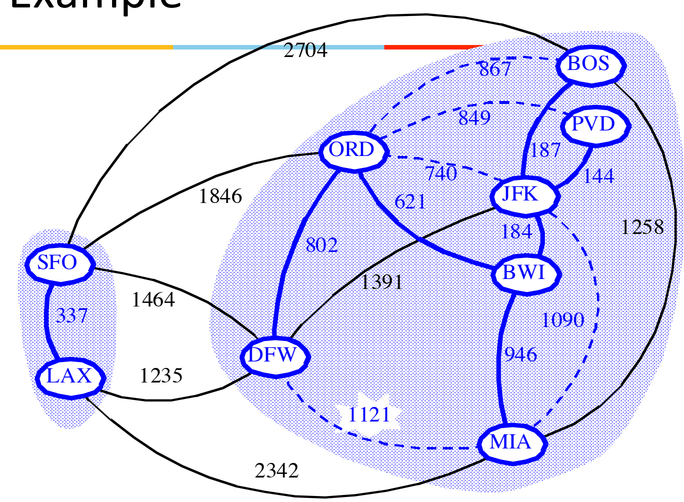
Example



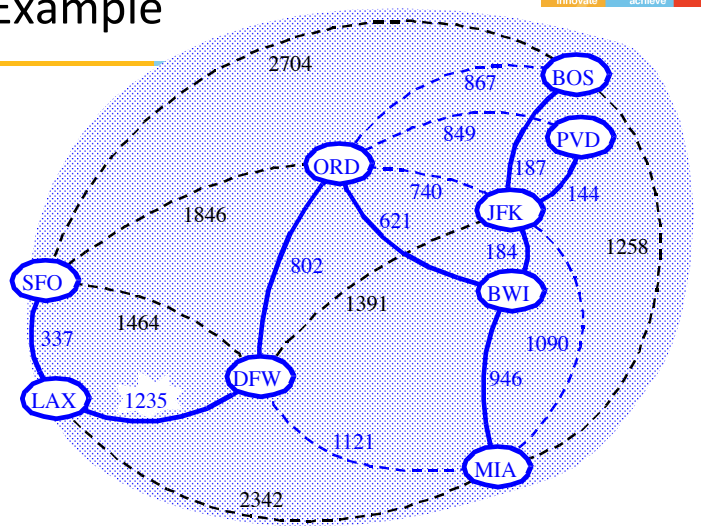
Example

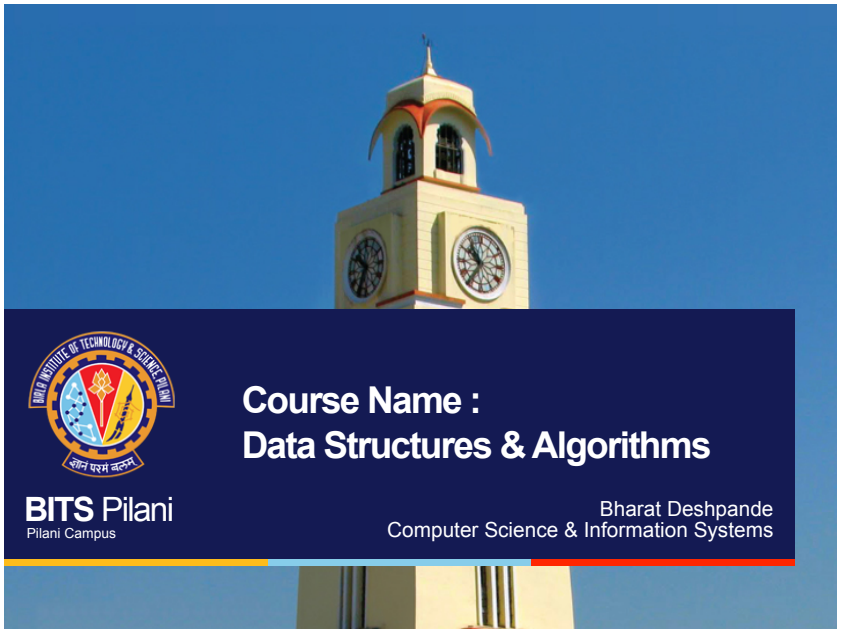


Example



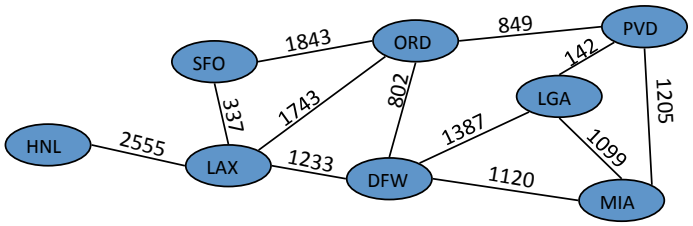
Example





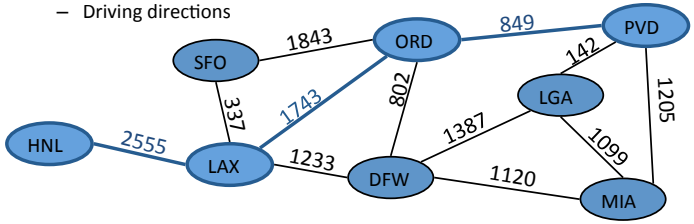
Weighted Graphs

- In a weighted graph, each edge has an associated numerical value, called the **weight** of the edge
- Edge weights may represent, distances, costs, etc.
- **Example:**
  - In a flight route graph, the weight of an edge represents the distance in miles between the endpoint airports



Shortest Paths

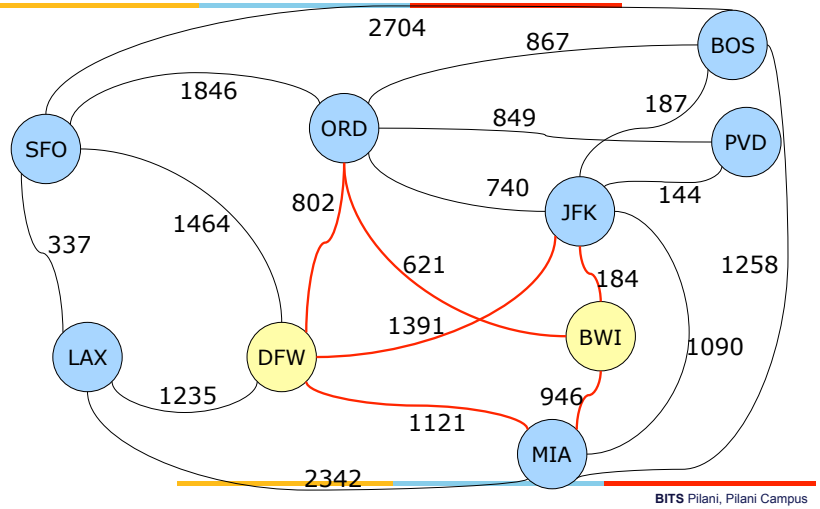
- Given a weighted graph and two vertices ***u*** and ***v***, we want to find a path of minimum total weight between ***u*** and ***v.***
  - Length of a path is the sum of the weights of its edges.
- **Example:**
  - Shortest path between Providence (PVD and Honolulu (HNL)
- **Applications**
  - Internet packet routing
  - Flight reservations
  - Driving directions



Weighted shortest path

- We want to minimize the total mileage.
- Breadth-first search does not work!
  - Minimum number of hops does not mean minimum distance.

Three 2-hop routes to DFW



A greedy algorithm

- Assume that every city is infinitely far away.
  - I.e., every city is ∞ miles away from BWI (except BWI, which is 0 miles away).
  - Now perform something similar to breadth-first search, and *optimistically guess that we have found the best path to each city as we encounter it*.
  - If we later discover we are wrong and find a better path to a particular city, then update the distance to that city.

Intuition behind Dijkstra's algorithm

- For our airline-mileage problem, we can start by guessing that every city is ∞ miles away.
  - Mark each city with this guess.
- Find all cities one hop away from BWI, and check whether the mileage is less than what is currently marked for that city.
  - If so, then revise the guess.
- Continue for 2 hops, 3 hops, etc.

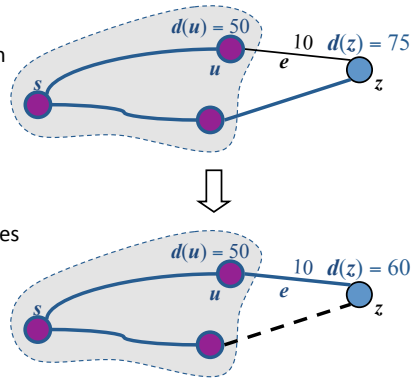
Dijkstra's Algorithm

- **Dijkstra's algorithm** computes the distances of all the vertices from a given start vertex s
- **Assumptions:**
  - the graph is connected
  - the graph is undirected and simple
  - the edge weights are nonnegative
- Algorithm grows a **cloud** of vertices, beginning with s and eventually covering all the vertices
- We store with each vertex v a label $d(v)$ representing the distance of v from s in the subgraph consisting of the cloud and its adjacent vertices
- At each step
  - We add to the cloud the vertex u outside the cloud with the smallest distance label, $d(u)$
  - We update the labels of the vertices adjacent to u

Edge Relaxation

- Update procedure is known as relaxation procedure.
- Consider an edge $e = (u,z)$ such that
  - u is the vertex most recently added to the cloud
  - z is not in the cloud
- The relaxation of edge e updates distance $d(z)$ as follows:

$$d(z) \leftarrow \min\{d(z), d(u) + weight(e)\}$$



Dijkstra's algorithm

- Algorithm initialization:
- Label each node with the distance ∞, except start node, which is labeled with distance 0.
  - D[v] is the distance label for v.

- Put all nodes into a priority queue Q, using the distances as labels.

Edge Relaxation

- While Q is not empty do:
  - u = Q.removeMin
  - for each node z one hop away from u do:
    - if D[u] + miles(u,z) < D[z] then
      - D[z] = D[u] + miles(u,z)
      - change key of z in Q to D[z]

- **Note** use of priority queue allows "finished" nodes to be found quickly (in $O(\log N)$ time).

- Running time is $O(m \log n)$

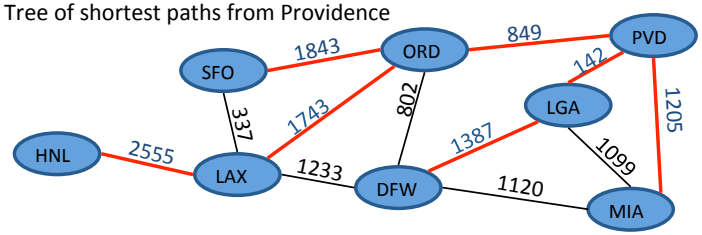
DIJKSTRA'S ALGORITHM - WHY IT WORKS

Property 1: Triangle inequality
If δ(u,v) is the shortest path length between u and v,
δ(u,v) ≤ δ(u,x) + δ(x,v)

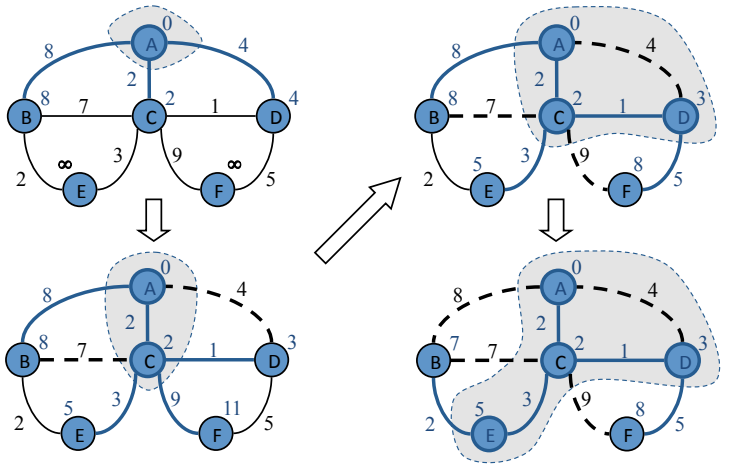
Property 2: A subpath of a shortest path is itself a shortest path

Property 3:
There is a tree of shortest paths from a start vertex to all the other vertices

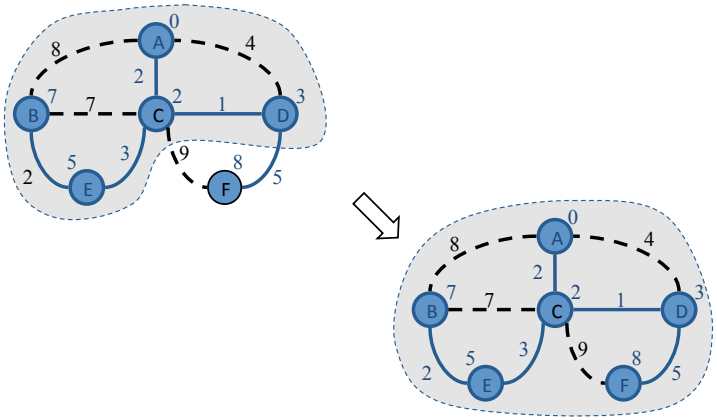
Example:
Tree of shortest paths from Providence



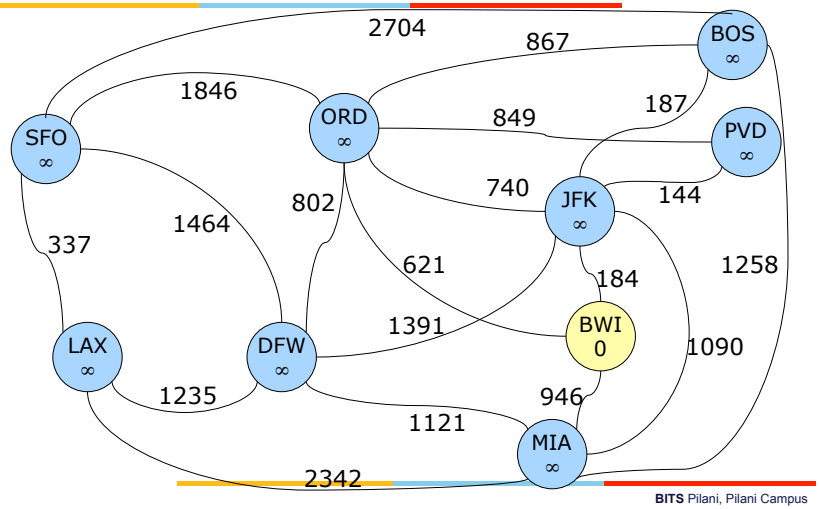
Example



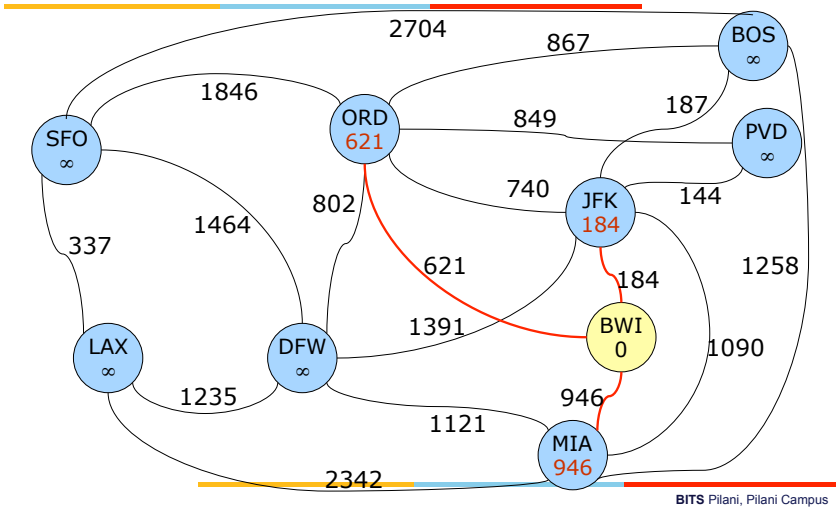
Example (cont.)



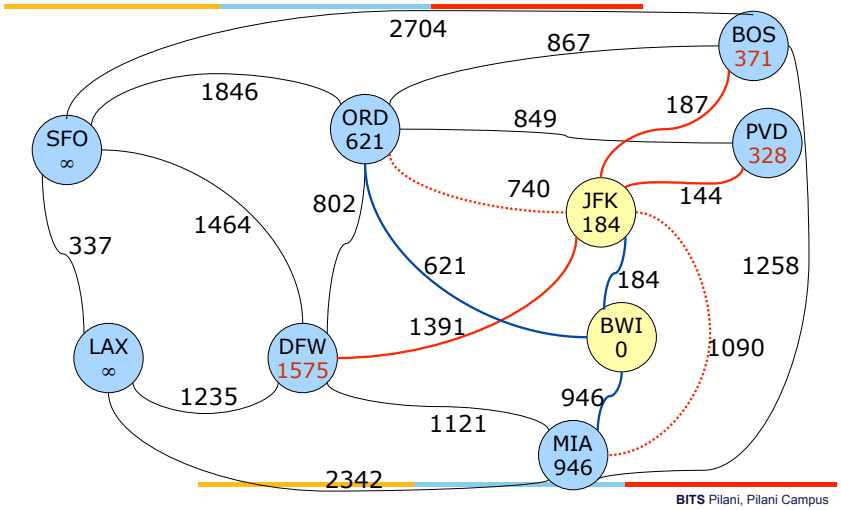
Shortest mileage from BWI



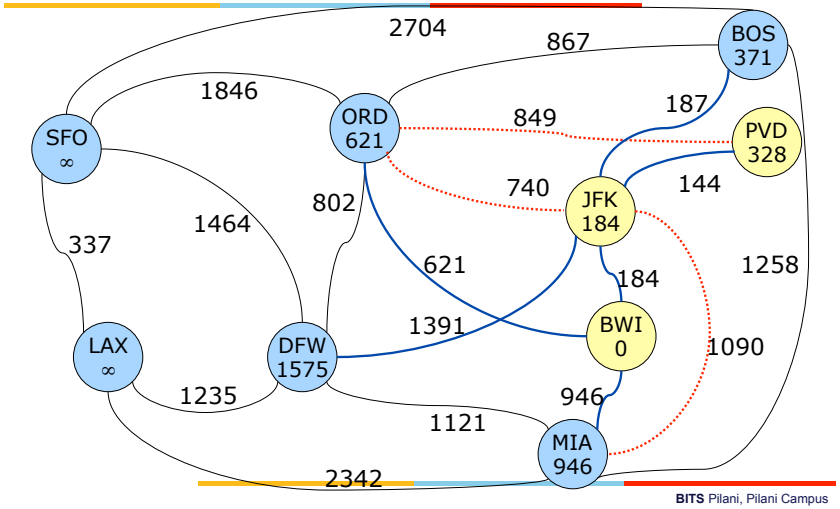
Shortest mileage from BWI



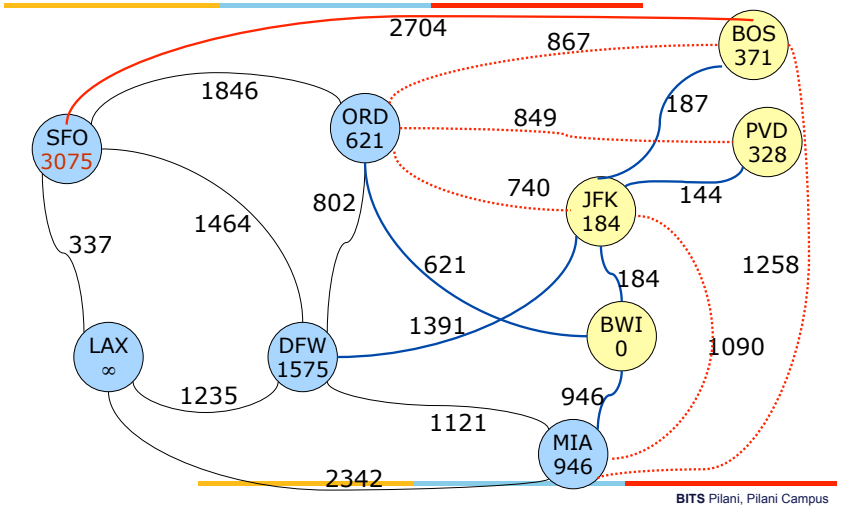
Shortest mileage from BWI



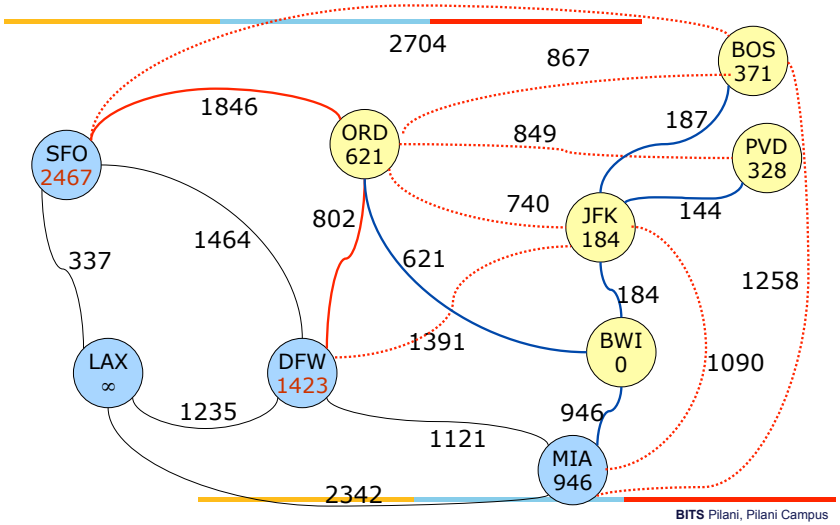
Shortest mileage from BWI



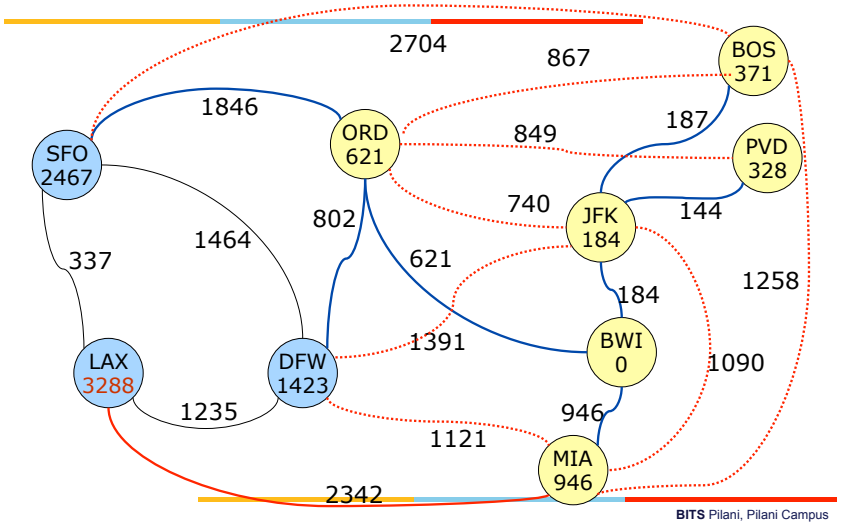
Shortest mileage from BWI



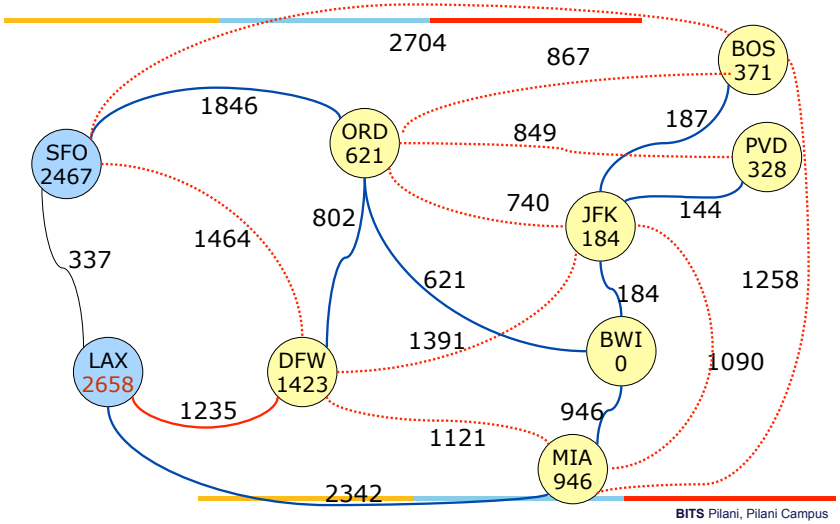
Shortest mileage from BWI



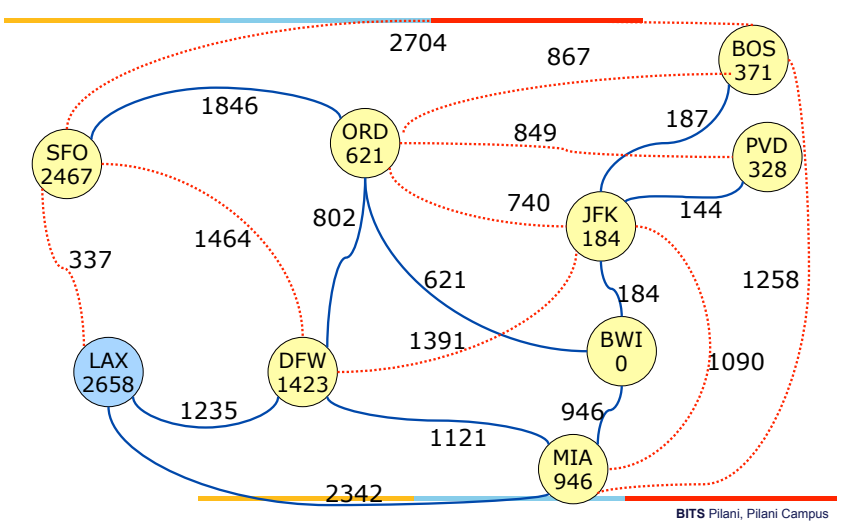
Shortest mileage from BWI



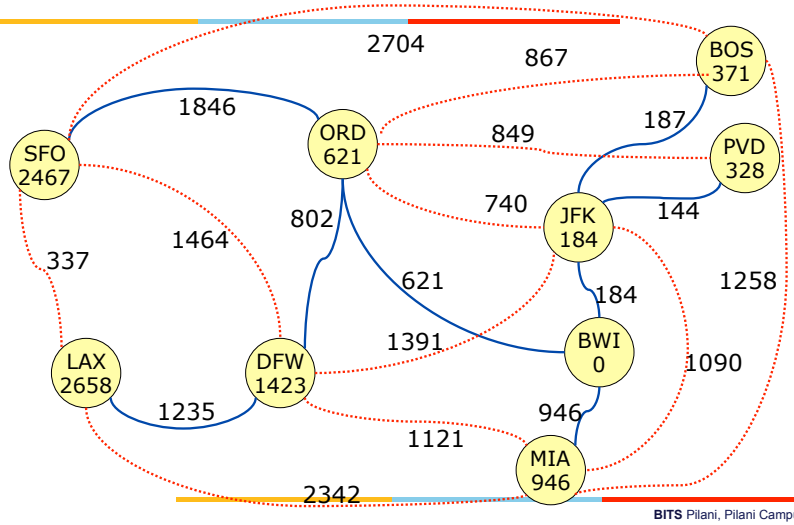
Shortest mileage from BWI



Shortest mileage from BWI

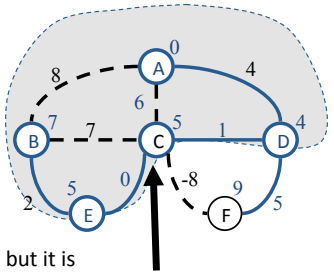


Shortest mileage from BWI



Why It Doesn't Work for Negative-Weight Edges

- Dijkstra's algorithm is based on the greedy method. It adds vertices by increasing distance.
  - If a node with a negative incident edge were to be added late to the cloud, it could mess up distances for vertices already in the cloud.



C's true distance is 1, but it is already in the cloud with d(C)=5!

Bellman-Ford Algorithm

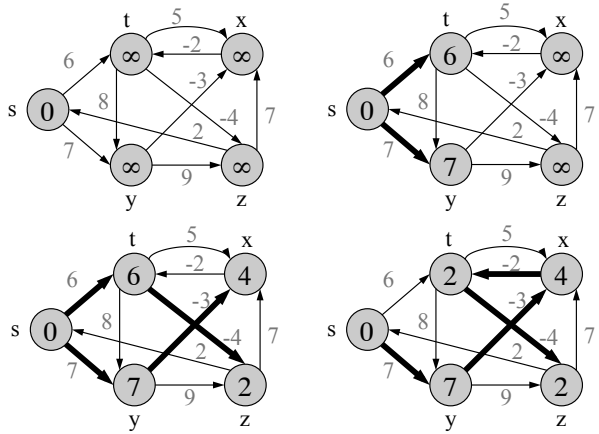
- Works even with negative-weight edges
- Must assume directed edges (for otherwise we would have negative-weight cycles)
- Iteration i finds all shortest paths that use i edges.
- Can be extended to detect a negative-weight cycle if it exists
- Uses edge relaxation as in Dijkstra's but not in conjunction with the greedy method.

Bellman-Ford Algorithm

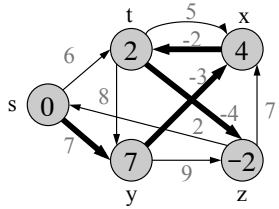
Bellman-Ford(G,s)

```
for each vertex u ∈ V
    d[u] ←∞
   parent[u] ← NIL
d[s] ← 0
for i ← 1 to |V|-1 do
  for each edge (u,v) outgoing from u do
     {Perform the relaxation operation on (u, v)}
      if d[u] + w(u, v) < d[z] then
        d[z] ← d[u] + w(u, v)
if there are no more edges left with potential relaxation operation then
  return the label d[u] of each vertex u
else
  return G contains a negative weight cycle.
```

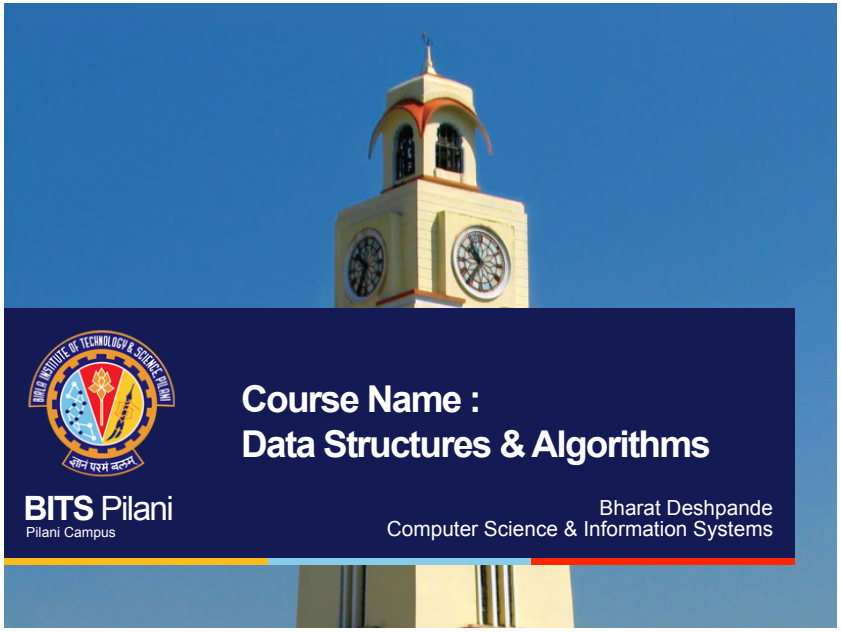
Bellman-Ford Example



Bellman-Ford Example



- Bellman-Ford running time:
  - $(|V|-1)|E| + |E| = \Theta(VE)$



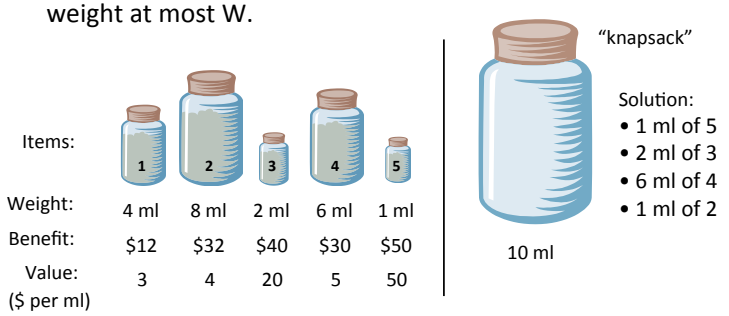
The Greedy Method

- **The greedy method** is a general algorithm design paradigm, built on the following elements:
  - **configurations**: different choices, collections, or values to find
  - **objective function**: a score assigned to configurations, which we want to either maximize or minimize
- It works best when applied to problems with the **greedy-choice** property:
  - a globally-optimal solution can always be found by a series of local improvements from a starting configuration.

The Fractional Knapsack Problem

- Given: A set S of n items, with each item i having
  - b_i - a positive benefit
  - w_i - a positive weight
- **Goal:** Choose items with maximum total benefit but with weight at most W.
- If we are allowed to take fractional amounts, then this is the **fractional knapsack problem**.
  - In this case, we let x_i denote the amount we take of item i
  - Objective: maximize $\sum_{i \in S} b_i(x_i / w_i)$
  - Constraint: $\sum_{i \in S} x_i \leq W$

- Given: A set S of n items, with each item i having
  - b_i - a positive benefit
  - w_i - a positive weight
- Goal: Choose items with maximum total benefit but with weight at most W.



BITS Pilani, Pilani Campus

The Fractional Knapsack Algorithm

- **Greedy choice**: Keep taking item with highest **value** (benefit to weight ratio)
  - Since $\sum_{i \in S} b_i(x_i / w_i) = \sum_{i \in S} (b_i / w_i) x_i$
  - **Run time:** O(n log n).
- **Correctness:** Suppose there is a better solution
  - there are two items i & j such that $x_i < w_i$, $x_j > 0$ and $v_i < v_j$
  - If we substitute some i with j, we get a better solution
  - How much of i: min{w_i-x_i, x_j}
  - Thus, there is no better solution than the greedy one

Algorithm *fractionalKnapsack(S, W)*
Input: set *S* of items w/ benefit b_i and weight w_i; max. weight *W*
Output: amount x_i of each item *i* to maximize benefit w/ weight at most *W*
for *each item i in S*
  $x_i \leftarrow 0$
  $v_i \leftarrow b_i / w_i$ {value}
$w \leftarrow 0$ {total weight}
while $w < W$
  remove item i w/ highest v_i
  $x_i \leftarrow \min\{w_i, W - w\}$
  $w \leftarrow w + \min\{w_i, W - w\}$

BITS Pilani, Pilani Campus

Task Scheduling

- Given: a set T of n tasks, each having:
  - A start time, s_i
  - A finish time, f_i (where $s_i < f_i$)
- **Goal:** Perform all the tasks using a minimum number of "machines."
- Tasks i & j are non-conflicting if $f_i \leq s_j$ or $f_j \leq s_i$
- Two tasks can be scheduled on the same machine if they are non-conflicting

BITS Pilani, Pilani Campus

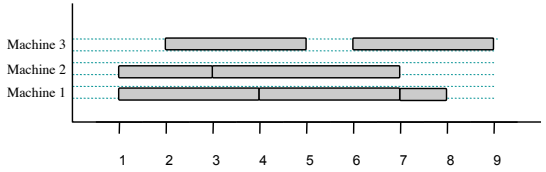
Task Scheduling Algorithm

- **Greedy choice: consider tasks by their start time and use as few machines as possible with this order.**
  - **Run time: O(n log n).**
- **Correctness: Suppose there is a better schedule.**
  - **We can use k-1 machines**
  - **The algorithm uses k**
  - **Let i be first task scheduled on machine k**
  - **Task i must conflict with k-1 other tasks**
  - **All these tasks also conflict with each other**
  - **But that means there is no non-conflicting schedule using k-1 machines**

Algorithm *taskSchedule(T)*
Input: set *T* of tasks w/ start time s_i and finish time f_i
Output: non-conflicting schedule with minimum number of machines
m ← *0* {no. of machines}
while *T is not empty*
remove task i w/ smallest s_i
if *there's a machine j for i* **then**
schedule i on machine j
else
m ← *m* + *1*
schedule i on machine m

Example

- Given: a set T of n tasks, each having:
  - A start time, s_i
  - A finish time, f_i (where $s_i < f_i$)
  - [1,4], [1,3], [2,5], [3,7], [4,7], [6,9], [7,8] (ordered by start)
- Goal: Perform all tasks on min. number of machines



Text Compression

- Given a string X, efficiently encode X into a smaller bit string Y.
- **Easy Solution**

Since there are only 26 characters in English Language & there are 32 bit string of length 5, Each alphabet can be represented by a bit string of length 5.

- This is called a **fixed length code.**

Text Compression

Question

Is it possible to find coding scheme, in which fewer bits are used.

Answer – Variable length codes

- Alphabets that occur more frequently should be encoded using short bit strings &

  rarely occurring alphabets should be encoded using long bit strings

Example

	a	b	c	d	e	f
Frequency	45	13	12	16	9	5
Fixed length	000	001	010	011	100	101
Variable length	0	101	100	111	1101	1100

A file with 100000 character contains only the alphabets a – f.

- Fixed length code requires 300000 bits to code the file.
- Variable length coding requires 224000 bits to code the file.

Prefix Code

- **Important**

In variable length code, some method must be used to determine where the bits for each character start and end.

For Example: If e : 0, a : 1, t : 01

Then 0101 could correspond to eat, tea or eaea.

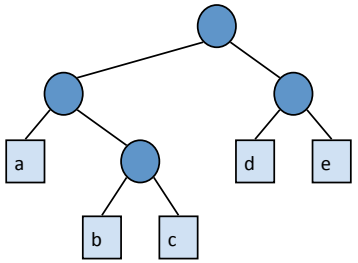
A **prefix code** is a binary code such that no code word is the prefix of another code-word

Encoding Tree

Key: A binary code can be represented by a binary tree.

- Each external node stores a character
- The code word of a character is given by the path from the root to the external node storing the character (0 for a left child and 1 for a right child)

00	010	011	10	11
a	b	c	d	e



The Huffman Coding algorithm-History

- In 1951, David Huffman and his MIT information theory classmates given the choice of a term paper or a final exam
- Huffman hit upon the idea of using a frequency-sorted binary tree and quickly proved this method the most efficient.
- In doing so, the student outdid his professor, who had worked with information theory inventor Claude Shannon to develop a similar code.
- Huffman built the tree from the bottom up instead of from the top down

Huffman Code

- Greedy algorithm that constructs an optimal prefix code.
- Algorithm builds the tree in a bottom up manner.
- C be set of n alphabets and f[c] be the frequency of c ∈ C.
- Algorithm begins with a set of |C| leaves and performs a sequence of |C| - 1 merging operations to create the final tree.
- A priority queue Q keyed on f values, is used to identify the two least frequent objects to merge together.
- The result of the merger of two objects is a new object whose frequency is the sum of two frequencies of the two objects that are merged. The new object is inserted back into Q.

Huffman's Algorithm

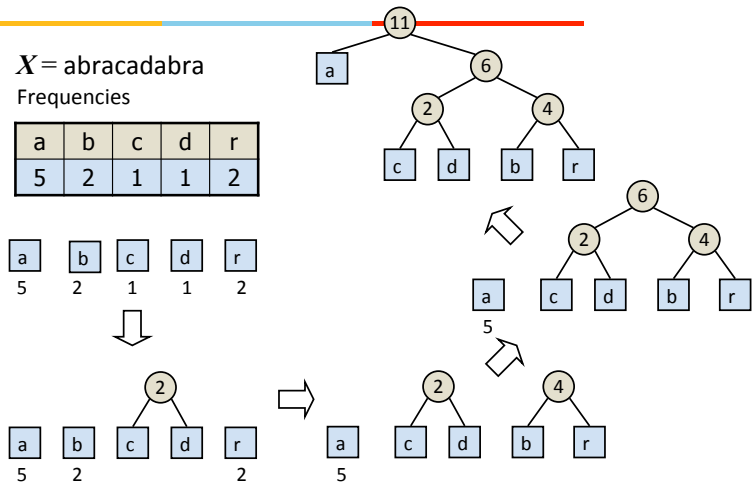
- It runs in time $O(n + d \log d)$, where n is the size of X and d is the number of distinct characters of X
- A heap-based priority queue is used as an auxiliary structure

Algorithm *HuffmanEncoding*(X)
Input string X of size n
Output optimal encoding trie for X
$C \leftarrow$ *distinctCharacters*(X)
computeFrequencies(C, X)
$Q \leftarrow$ new empty heap
for all $c \in C$
  $T \leftarrow$ new single-node tree storing c
  Q.*insert*(*getFrequency*(c), T)
while Q.*size*() > 1
  $f_1 \leftarrow Q$.*minKey*()
  $T_1 \leftarrow Q$.*removeMin*()
  $f_2 \leftarrow Q$.*minKey*()
  $T_2 \leftarrow Q$.*removeMin*()
  $T \leftarrow$ *join*(T_1, T_2)
  Q.*insert*($f_1 + f_2$, T)
return Q.*removeMin*()

Example

X = abracadabra
Frequencies

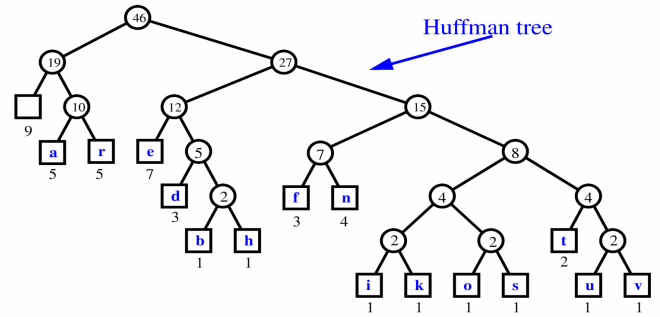
a	b	c	d	r
5	2	1	1	2



Huffman Tree Example

String: **a fast runner need never be afraid of the dark**

Character	a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
Frequency	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1



	PHI
R3	Algorithm Design, Jon Kleinberg, Eva Tardos, First Ed., Pearson

Content Structure

1. Analyzing Algorithms [3 Hours]
   - 1.1. Theoretical Foundation
     - 1.1.1. Algorithms and it's Specification
     - 1.1.2. Random Access Machine Model
     - 1.1.3. Counting Primitive Operations
     - 1.1.4. Notion of best case, average case and worst case
   - 1.2. Characterizing Run Time
     - 1.2.1. Use of asymptotic notation
     - 1.2.2. Big-Oh Notation, Little-Oh, Omega and Theta Notations
   - 1.3. Correctness of Algorithms
   - 1.4. Analyzing Recursive Algorithms
     - 1.4.1. Recurrence relations
     - 1.4.2. Specifying runtime of recursive algorithms
     - 1.4.3. Solving recurrence equations
   - 1.5. Case Study: Analyzing Algorithms

2. Elementary Data Structures [2 hours]
   - 2.1.
   - 2.2. Stacks
     - 2.2.1. Stack ADT and Implementation
     - 2.2.2. Applications
   - 2.3. Queues
     - 2.3.1. Queue ADT and Implementation
     - 2.3.2. Applications
   - 2.4. List
     - 2.4.1. Notion of position in lists
     - 2.4.2. List ADT and Implementation

3. Non-Linear Data Structures [4 hours]
   - 3.1. Trees
     - 3.1.1. Terms and Definition
     - 3.1.2. Tree ADT
     - 3.1.3. Applications
   - 3.2. Binary Trees
     - 3.2.1. Properties
     - 3.2.2. Representations (Vector Based and Linked)
     - 3.2.3. Binary Tree traversal (In Order, Pre Order, Post Order)
     - 3.2.4. Applications
   - 3.3. Heaps
     - 3.3.1. Definition and Properties
     - 3.3.2. Representations (Vector Based and Linked)
     - 3.3.3. Insertion and deletion of elements
     - 3.3.4. Heap implementation of priority queue
     - 3.3.5. Heap sort

4. Dictionaries [as Hash Tables and Search Trees] [3 hours]
   - 4.1. Unordered Dictionary
     - 4.1.1. ADT Specification

Topics Covered

BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI

WORK INTEGRATED LEARNING PROGRAMMES

Digital Learning

Part A: Course Design

Course Title	Data Structures and Algorithms Design
Course No(s)	SSZG519
Credit Units	
Credit Model	
Content Authors	BHARAT M DESHPANDE

Course Objectives

CO1	Introduce mathematical and experimental techniques to analyze algorithms
CO2	Introduce linear and non-linear data structures and best practices to choose appropriate data structure for a given application
CO3	Exposes students to various sorting and searching techniques
CO4	Overview of algorithm design methods such as the greedy method, divide and conquer, dynamic programming, backtracking, and branch and bound
CO5	Introduce complexity classes and ways of classifying problem.

Text Book(s)

T1	Algorithms Design: Foundations, Analysis and Internet Examples Michael T. Goodrich, Roberto Tamassia, 2006, Wiley (Students Edition)

Reference Book(s) & other resources

R1	Data Structures, Algorithms and Applications in C++, Sartaj Sahni, Second Ed, 2005, Universities Press
R2	Introduction to Algorithms, TH Cormen, CE Leiserson, RL Rivest, C Stein, Third Ed, 2009,

6.2.4.2. Vertex Cover

6.2.4.3. Clique and Set-Cover

6.2.4.4. Subset-Sum and Knapsack

6.2.4.5. Hamiltonian Cycle and TSP

Learning Outcomes:

No	Learning Outcomes
LO1	Analyzing algorithms using recurrence relations and expressing it using asymptotic notation.
LO2	Understanding different sorting and searching algorithms with the analysis of best, worst and average cases.
LO4	Understanding of graph algorithms for Reachability and Path Finding
LO3	Students will be able to solve problems using appropriate data structures to implement solutions.
L05	Able to classify problems into complexity classes P and NP and to prove hardness of problems

Part B: Content Development Plan

Academic Term	Second Semester 2015-2016
Course Title	Data Structures and Algorithms Design
Course No	SSZG 519
Content Developer	BHARAT M DESHPANDE

Glossary of Terms:

1. Contact Hour (CH) stands for a hour long live session with students conducted either in a physical classroom or enabled through technology. In this model of instruction, instructor led sessions will be for 20 CH.

a. Pre CH = Self Learning done prior to a given contact hour
b. During CH = Content to be discussed during the contact hour by the course instructor
c. Post CH = Self Learning done post the contact hour

2. RL stands for Recorded Lecture or Recorded Lesson. It is presented to the student through an online portal. A given RL unfolds as a sequences of video segments interleaved with exercises
3. SS stands for Self-Study to be done as a study of relevant sections from textbooks and reference books. It could also include study of external resources.
4. LE stands for Lab Exercises
5. HW stands for Home Work will consists could be a selection of problems from the text.

4.1.2. Applications

4.2. Hash Tables

4.2.1. Notion of Hashing and Collision (with a simple vector based hash table)

4.2.2. Hash Functions

4.2.2.1. Properties

4.2.2.2. Simple hash functions

4.2.3. Methods for Collision Handling

4.2.3.1. Separate Chaining

4.2.3.2. Notion of Load Factor

4.2.3.3. Rehashing

4.2.3.4. Open Addressing [Linear & Quadratic Probing, Double Hash]

4.3. Ordered Dictionary

4.3.1. ADT Specification

4.3.2. Applications

4.4. Binary Search Tree

4.4.1. Motivation with the task of Searching and Binary Search Algorithm

4.4.2. Properties of BST

4.4.3. Searching an element in BST

4.4.4. Insertion and Removal of Elements

4.4.5. Performance

5. Algorithm Design Techniques [6 Hours]

5.1. Greedy Method

5.1.1. Design Principles and Strategy

5.1.2. Fractional Knapsack Problem

5.1.3. Task Scheduling Problem

5.2. Divide and Conquer

5.2.1. Design Principles and Straegy

5.2.2. Analyzing Divide and Conquer Algorithms

5.2.3. Integer Multiplication Problem

5.2.4. Sorting Problem

5.2.4.1. Merge Sort Algorithm

5.2.4.2. Quick Sort Algorithm

5.2.5. Searching Problem and Binary Search Algorithm [Ref to 4.4.1]

5.3. Dynamic Programming

5.3.1. Design Principles and Strategy

5.3.2. Matrix Chain Product Problem

5.3.3. 0/1 Knapsack Problem

5.4. Graph Algorithms

5.4.1. Introduction to Graphs

5.4.2. Prim's and Kruskal's Algorithms [Greedy]

5.4.3. Dijkstra's Algorithm [Greedy]

5.4.4. Bellman-ford Shortest Path Algorithm [Greedy]

5.4.5. All Pair Shortest Path Algorithm [Dynamic Programming]

6. Complexity Classes [4 hours]

6.1. Definition of P and NP classes and examples

6.2. Understanding NP-Completeness

6.2.1. NP-Hardness

6.2.2. Polynomial time reducibility

6.2.3. Cook-Levin theorem

6.2.4. Problems in NP-Complete and using polynomial time reductions

6.2.4.1. CNF-SAT, 3-SAT

Post CH	LE4		
Post CH	QZ4		

Notes:

Contact Hour 5

Time	Type	Sequence	Content Reference
Pre CH	RL1.2		
During CH	CH5	CH5.1 =List: Notion of position in lists CH5.2 = List ADT and Implementation	T1: 2.2
Post CH	SS5		
Post CH	HW5	T1 - R-2.1	
Post CH	LE5		
Post CH	QZ5		

Notes:

Contact Hour 6

Time	Type	Sequence	Content Reference
Pre CH	RL1.3		
During CH	CH6	CH6.1 =Trees: Terms and Definition, CH6.2= Tree ADT, Applications CH6.3= Binary Trees : Terms and Definition, CH6.4= Properties	T1: 2.3.1, 2.3.2
Post CH	SS6		
Post CH	HW6		

Post CH	LE6		
Post CH	QZ6		

Notes:

Contact Hour 7

Time	Type	Sequence	Content Reference
Pre CH	RL1.3		
During CH	CH7	CH7.1 =Binary Trees : Representations (Vector Based and Linked), C H7.2= Binary Tree traversal (In Order, Pre Order, Post Order), CH7.3= Applications	T1: 2.3.3
Post CH	SS7		
Post CH	HW7		
Post CH	LE7		
Post CH	QZ7		

Notes:

Contact Hour 8

Time	Type	Sequence	Content Reference
Pre CH	RL2.2		
During CH	CH8	CH8.1 =Heaps: Definition and Properties, CH8.2 = Representations (Vector Based and Linked), CH8.3 =Insertion and deletion of elements	T1:2.4.1, 2.4.3
Post CH	SS8		

Contact Hour 1

Time	Type	Sequence	Content Reference
Pre CH	RL1.1		
During CH	CH1	CH1.1= Introduction CH1.2 = Algorithms and it's Specification, CH1.3 = Random Access Machine Model, Counting Primitive Operations CH1.4= Notion of best case, average case and worst case	T1: 1.1.1-1.1.3
Post CH	SS1		
Post CH	HW1		
Post CH	LE1		
Post CH	QZ1		

Notes:

Contact Hour 2

Time	Type	Sequence	Content Reference
Pre CH	RL1.1		
During CH	CH2	CH2.1 =Use of asymptotic notation (Big-Oh Notation, Little-Oh, Omega and Theta Notations) CH2.2 = Correctness of Algorithms	T1: 1.2
Post CH	SS2	T1 – 1.3	
Post CH	HW2	T1 - R-1.15, R-1.19	
Post CH	LE2		

Post CH	QZ2		

Notes:

Contact Hour 3

Time	Type	Sequence	Content Reference
Pre CH			
During CH	CH3	CH3.1= Analyzing Recursive Algorithms: CH3.2= Recurrence relations, CH3.3= Specifying runtime of recursive algorithms, CH3.4= Solving recurrence equations	T1: 1.1.4, Class Notes
Post CH	SS3		
Post CH	HW3		
Post CH	LE3		
Post CH	QZ3		

Notes:

Contact Hour 4

Time	Type	Sequence	Content Reference
Pre CH	RL1.2		
During CH	CH4	CH4.1 =Stacks: ADT and Implementation, Applications CH4.2= Queues: Queue ADT and Implementation, Applications	T1: 2.1
Post CH	SS4		
Post CH	HW4		

		Searching and Binary Search Algorithm, CH12.3 = Properties of BST CH12.4 = Searching an element in BST, Insertion and Removal of Elements	
Post CH	SS12	T1: 3.1.6	
Post CH	HW12		
Post CH	LE12		
Post CH	QZ12		

Notes:

Contact Hour 13

Time	Type	Sequence	Content Reference
Pre CH			
During CH	CH13	CH13.1 =CH12.1 =Greedy Method: Design Principles and Strategy, CH12.2 =Fractional Knapsack Problem	T1: 5.1.1
Post CH	SS13		
Post CH	HW13		
Post CH	LE13		
Post CH	QZ13		

Notes:

Contact Hour 14

Time	Type	Sequence	Content Reference
Pre CH			
During CH	CH14	CH14.1 =Greedy Method Task Scheduling Problem	T1:5.1.2
Post CH	SS14		
Post CH	HW14		
Post CH	LE14		
Post CH	QZ14		

Notes:

Contact Hour 15

Time	Type	Sequence	Content Reference
Pre CH	RL2.1		
During CH	CH15	CH15.1 =Divide and Conquer: Design Principles and Strategy, CH15.2 = Analyzing Divide and Conquer Algorithms CH15.3 = Integer Multiplication Problem	T1:5.2.1, 5.2.2
Post CH	SS15		
Post CH	HW15		
Post CH	LE15		
Post CH	QZ15		

Notes:

Contact Hour 16

Time	Type	Sequence	Content Reference
Pre CH	RL2.1,RL2.2		
During CH	CH16	CH16.1 = Merge Sort CH16.2 =Quick Sort	T1:4.1, 4.3
Post CH	HW8		
Post CH	LE8		
Post CH	QZ8		

Notes:

Contact Hour 9

Time	Type	Sequence	Content Reference
Pre CH	RL2.2		
During CH	CH9	CH9.1 =Heaps: Heap implementation of priority queue, CH9.2 = Heap sort	T1: 2.4.2, 2.4.3
Post CH	SS9	T1 - Section 2.4.2 PQ-Sort	
Post CH	HW9		
Post CH	LE9		
Post CH	QZ9		

Notes:

Contact Hour 10

Time	Type	Sequence	Content Reference
Pre CH	RL3.1		
During CH	CH10	CH10.1 =Unordered Dictionary :ADT, Applications CH10.2= Hash Tables: Notion of Hashing and Collision (with a simple vector based hash table) CH10.3 = Hash Functions: Properties, Simple hash functions	T1: 2.5.1, 2.5.2, 2.5.3, 2.5.4
Post CH	SS10		
Post CH	HW10		
Post CH	LE10		
Post CH	QZ10		

Notes:

Contact Hour 11

Time	Type	Sequence	Content Reference
Pre CH	RL3.2		
During CH	CH11	CH11.1 =Methods for Collision Handling: CH11.2 = Notion of Load Factor, CH11.3 = Rehashing, CH11.4 = Open Addressing	T1: 2.5.5
Post CH	SS11		
Post CH	HW11		
Post CH	LE11		
Post CH	QZ11		

Notes:

Contact Hour 12

Time	Type	Sequence	Content Reference
Pre CH	RL4.1		
During CH	CH12	CH12.1= Ordered Dictionary: ADT, Applications CH12.2 = Binary Search Tree: Motivation with the task of	T1: 3.1.1-3.1.5

Post CH	SS20		
Post CH	HW20		
Post CH	LE20		
Post CH	QZ20		

Notes:

Contact Hour 21

Time	Type	Sequence	Content Reference
Pre CH			
During CH	CH21	CH21.1 =Clique and Set-Cover, CH21.2 = Subset-Sum and Knapsack	T1:13.3
Post CH	SS21	Traveling Salesman Problem	
Post CH	HW21		
Post CH	LE21		
Post CH	QZ21		

Notes:

Contact Hour 22

Time	Type	Sequence	Content Reference
Pre CH			
During CH	CH22	CH22.1 = Course Review	
Post CH	SS22		
Post CH	HW22		
Post CH	LE22		
Post CH	QZ22		

Notes:

Evaluation Scheme:
Legend: EC = Evaluation Component; AN = After Noon Session; FN = Fore Noon Session

No	Name	Type	Duration	Weight	Day, Date, Session, Time
EC-1	Quiz-I/ Assignment-I	Online	-	5%	February 1 to 10, 2016
	Quiz-II			5%	March 1 to 10, 2016
	Quiz-III/ Assignment-II			5%	March 25 to April 3, 2016
EC-2	Mid-Semester Test	Closed Book	2 hours	35%	27/02/2016 (AN) 2 PM – 4 PM
EC-3	Comprehensive Exam	Open Book	3 hours	50%	09/04/2016 (AN) 2 PM – 5 PM

Syllabus for Mid-Semester Test (Closed Book): Topics in Session Nos. 1 to 11
Syllabus for Comprehensive Exam (Open Book): All topics (Session Nos. 1 to 22)

Important links and information:

Elearn portal: https://elearn.bits-pilani.ac.in

Students are expected to visit the Elearn portal on a regular basis and stay up to date with the latest announcements and deadlines.

Contact sessions: Students should attend the online lectures as per the schedule provided on the Elearn portal.

Evaluation Guidelines:

1. EC-1 consists of either two Assignments or three Quizzes. Students will attempt them through the course pages on the Elearn portal. Announcements will be made on the portal, in a timely manner.
2. For Closed Book tests: No books or reference material of any kind will be permitted.
3. For Open Book exams: Use of books and any printed / written reference material (filed or bound) is permitted. However, loose sheets of paper will not be allowed. Use of calculators is permitted in all exams. Laptops/Mobiles of any kind are not allowed. Exchange of any material is not allowed.
4. If a student is unable to appear for the Regular Test/Exam due to genuine exigencies, the student should follow the procedure to apply for the Make-Up Test/Exam which will be made available on the Elearn portal. The Make-Up Test/Exam will be conducted only at selected exam centres on the dates to be announced later.

It shall be the responsibility of the individual student to be regular in maintaining the self study schedule as given in the course handout, attend the online lectures, and take all the prescribed evaluation components such as Assignment/Quiz, Mid-Semester Test and Comprehensive Exam according to the evaluation scheme provided in the handout.

Post CH	SS16		
Post CH	HW16		
Post CH	LE16		
Post CH	QZ16		

Notes:

Contact Hour 17

Time	Type	Sequence	Content Reference
Pre CH			
During CH	CH17	CH17.1 =Dynamic Programming: Design Principles and Strategy, CH17.2 = Matrix Chain Product Problem CH17.3 =0/1 Knapsack Problem	T1: 5.3.1, 5.3.2
Post CH	SS17		
Post CH	HW17		
Post CH	LE17		
Post CH	QZ17		

Notes:

Contact Hour 18

Time	Type	Sequence	Content Reference
Pre CH	RL5.1,RL5.2		
During CH	CH18	CH18.1 = Graphs : Terms and Definitions, Properties, CH18.2 = Representations (Edge List, Adjacency list, Adjacency Matrix) CH18.3 = Graph Traversals	T1: 6.1, 6.2,6.3

Post CH	SS18	T1: 6.4	
Post CH	HW18		
Post CH	LE18		
Post CH	QZ18		

Notes:

Contact Hour 19

Time	Type	Sequence	Content Reference
Pre CH	RL 5.4,5.5		
During CH	CH19	CH 19.1 =Single Source Shortest Path algorithm: Dijkstra's Algorithm CH19.2 = Definition of P and NP classes and examples	T1:7.1.1 T1: 13.1
Post CH	SS19	T1. 7.1.2, 7.2.1, 7.3.1, 7.3.2	
Post CH	HW19		
Post CH	LE19		
Post CH	QZ10		

Notes:

Contact Hour 20

Time	Type		Content Reference
Pre CH			
During CH	CH20	CH20.1 =Understanding NP-Completeness: CNF-SAT Cook-Levin theorem CH20.2 =Polynomial time reducibility: CNF-SAT and 3-SAT, Vertex Cover	T 13.2, T 13.3