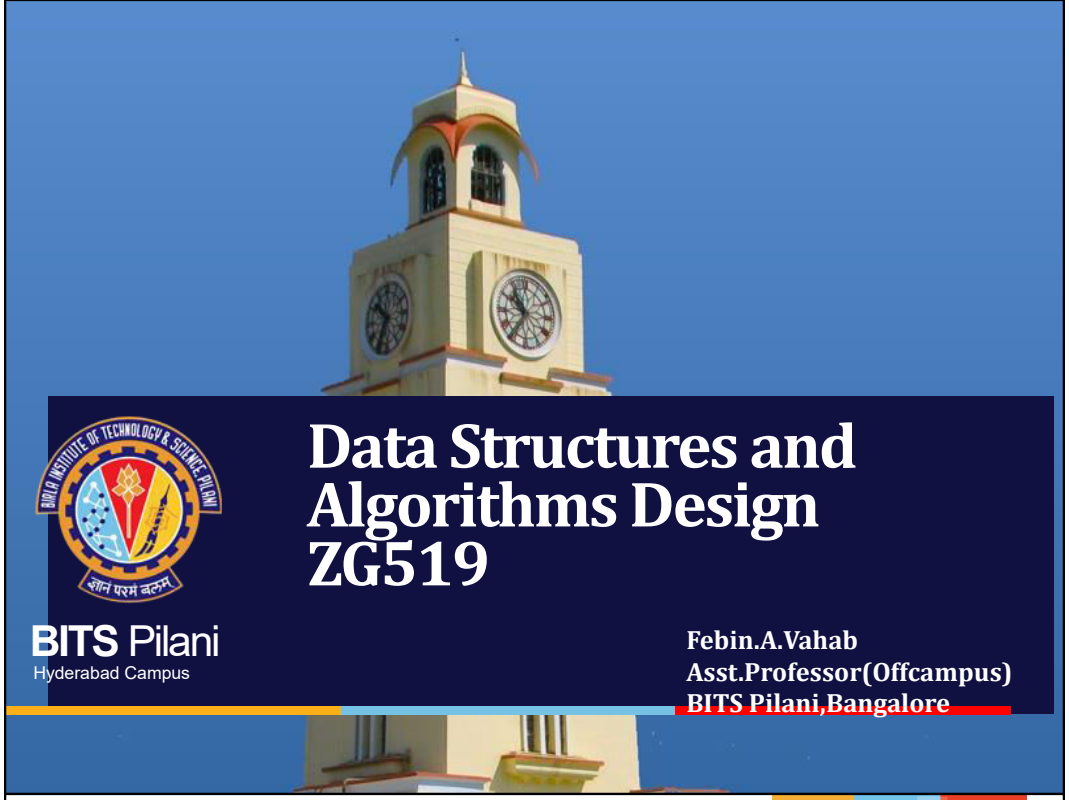


Data Structures and Algorithms Design ZG519

BITS Pilani
Hyderabad Campus

Febin.A.Vahab
Asst.Professor(Offcampus)
BITS Pilani,Bangalore

Course Objectives

No	Objective
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	Teach various dictionary data structures (Lists, Trees, Heaps, Bloom filters) with illustrations on possible representation, various operations and their efficiency
CO4	Exposes students to various sorting and searching techniques
CO5	Discuss in detail various algorithm design approaches (Greedy method, divide and conquer, dynamic programming and map reduce) with appropriate examples, methods to make correct design choice and the efficiency concerns
CO6	Introduce complexity classes , notion of NP-Completeness, ways of classifying problem into appropriate complexity class
CO7	Introduce reduction method to prove a problem's complexity class.

TEXT BOOKS

No	Author(s), Title, Edition, Publishing House
T1	Algorithms Design: Foundations, Analysis and Internet Examples Michael T. Goodrich, Roberto Tamassia, 2006, Wiley (Students Edition).
R1	Data Structures, Algorithms and Applications in Java, Sartaj Sahni, Second Ed, 2005, Universities Press
R2	Introduction to Algorithms, TH Cormen, CE Leiserson, RL Rivest, C Stein, Third Ed,2009, PHI

Topics

- **Analysing Algorithms**
- Elementary Data Structures
- Non-Linear Data Structures
- Dictionaries
  - Ordered/Unordered Dictionaries
  - Hash Tables
- Binary Search Trees
- Algorithm Design Techniques
  - Greedy Method
  - Divide and Conquer
  - Dynamic Programming
  - Graph Algorithms
- Complexity Classes

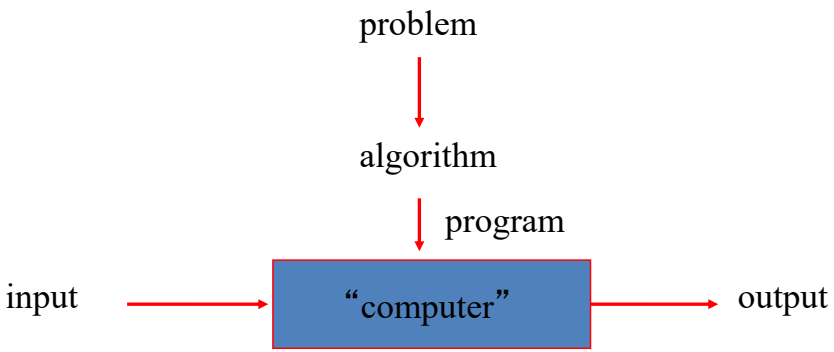
SESSION 1 -PLAN

Contact Sessions(#)	List of Topic Title	Text/Ref Book/external resource
1	Algorithms and it's Specification, Experimental Analysis, Analytical model-Random Access Machine Model, Counting Primitive Operations, Basic Operation method, Analyzing non recursive algorithms, Order of growth	T1: 1.1, 1.2

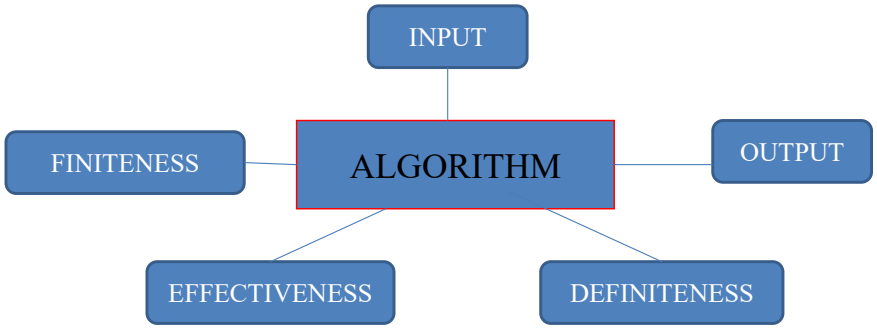
Algorithm

- An algorithm is a **finite sequence of unambiguous, step by step instructions** followed to accomplish a given task.

Notion of algorithm



Properties of Algorithms



Properties of Algorithms

- Finiteness:
  - An algorithm must terminate after finite number of steps.
- Definiteness:
  - The steps of the algorithm must be precisely defined. Each instruction must be clear and unambiguous.
- Effectiveness:
  - The operations of the algorithm must be basic enough to be put down on pencil and paper.
- Input-Output:
  - The algorithm must have certain(zero or more) initial and precise inputs, and outputs that may be generated both at its intermediate and final steps.

EUCLID'S ALGORITHM

ALGORITHM *Euclid(m, n)*

//Computes gcd*(m, n)* by Euclid's algorithm

//Input: Two nonnegative, not-both-zero integers *m* and *n*

//Output: Greatest common divisor of *m* and *n*

Step 1 If *n* = 0, return the value of *m* as the answer and stop; otherwise,

proceed to Step 2.

Step 2 Divide *m* by *n* and assign the value of the remainder to *r*.

Step 3 Assign the value of *n* to *m* and the value of *r* to *n*. Go to Step 1.

Analysis of algorithms

- Design the most efficient algorithm.

- SPACE
- TIME

EFFICENCY

- SPACE
  - Program space
  - Data space
  - Stack space

Analysis of algorithms

- TIME EFFICIENCY
  - Speed of computer
  - Choice of programming language
  - Compiler used
  - ***Choice of algorithm***
  - ***Number(size) of inputs***

Analysis of algorithms

- Experimental Analysis
  - Write a program implementing the algorithm
  - Execute the program on various test inputs and record the actual time spent.
  - Use system calls (Ex: System.currentTimeMillis()) to get an accurate measure of the actual running time.

Analysis of algorithms

- Limitations of experimental studies
  - Implementation is a must.
  - Execution is possible on limited set of inputs.
  - Inputs must be representatives of real time scenarios
  - If we need to compare two algorithms ,we need to use the same environment (like hardware, software etc)

Analysis of algorithms

- Analytical Model
  - 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.
- Find maximum element in an array :

Algorithm *arrayMax(A, n)*

Input: An array *A storing n>=1 integers*

Output: The maximum element in *A*

currentMax ←*A[0]*

for *i* ← *1 to n - 1 do*

if *currentMax <A[i] then*

  currentMax ← *A[i]*

return *currentMax*

Pseudo-code

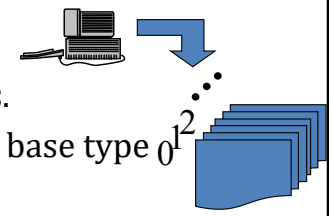
- Expressions
  - Use standard mathematical symbols to describe numeric and Boolean expressions.
  - Uses ← for assignment.(= in Java)
  - Use = for the equality relationship.(== in Java)
- Method declaration
  - Algorithm name(param1,param2…)
- Method returns: **return** value
- Control flow
  - **if** … **then** … [**else** …]
  - **while** … **do** …
  - **repeat** … **until** …
  - **for** … **do** …
  - Indentation replaces braces

Primitive Operations

Primitive operations are basic computations performed by an algorithm

- Assigning a value to a variable
- Calling a method
- Performing arithmetic operation
- Indexing into an array
- Following an object reference
- Returning from a method
- Comparing two numbers

The Random Access Machine (RAM) Model



- **NOT** Random access memory
- A CPU connected to a bank of memory cells.
- Each memory cell stores a word-Value of a base type
  - Number
  - Character string
  - An address etc
- Random Access: Ability of CPU to access an arbitrary memory cell with one *primitive operation*
- The CPU can perform any primitive operation in a *constant number of steps* which do not depend on the size of the input.

The Random Access Machine (RAM) Model

- Approach of simply counting primitive operations.
- Each primitive operation corresponds to constant time instruction.
- *A bound on the number of primitive operations* → *running time of that algorithm.*

Counting Primitive Operations

- Focus on each step of the algorithm and count the primitive operation it takes
- Consider the arrayMax Algorithm.

Algorithm *arrayMax(A, n)*

currentMax ←*A[0]*
for *i* ← *1 to n - 1 do*
if *currentMax <A[i] then*
  currentMax ← *A[i]*
return *currentMax*

- Assigning a value to a variable
- Calling a method
- Performing arithmetic operation
- Indexing into an array
- Following an object reference
- Returning from a method
- Comparing two numbers

Counting Primitive Operations

Statement	# of primitive operations
currentMax* ←*A[0] Indexing into array and assignment	2
for *i* ← *1 to n - 1 do*	
• i initialised to 1	1
i<n is verified , comparing 2 numbers(1 po) The comparison is performed n times(at the end of each iteration of the loop)	n
if currentMax <A[i] Comparison and Indexing	2
currentMax* ← *A[i] Assignment and indexing	2
Counter incremented Summing and assignment	2
Either 4 or 6 primitive operations, each iteration	

Counting Primitive Operations

Statement	# of primitive operations
Body of the loop	n-1
Total	
• If condition satisfied	6(n-1)
• If condition not satisfied	4(n-1)
return *currentMax*	1

The number of primitive operations executed by the algorithm
Atleast (Best case): 2+1+n+4(n-1)+1=5n

Atmost (worst case):2+1+n+6(n-1)+1=7n-2

Basic operation Method

- Identify the basic operation
- Basic operation???
  - Operation that contributes most towards the running time of an algorithm.
  - Statement that executes maximum number of times.

Basic operation Method

- Identify the basic operation
- Obtain the total number of times that operation is executed.

General Plan for Non recursive algorithms

- Decide on parameter n indicating input size
- Identify algorithm's basic operation
- Check whether the number of times the basic operation is executed depends only on the input size n. If it also depends on the type of input, investigate worst, average, and best case efficiency separately.
- Set up summation reflecting the number of times the algorithm's basic operation is executed.
- Simplify summation using standard formulas

Basic operation Method

ArrayMax

Algorithm arrayMax(A, n)
currentMax ←A[0]
for i ← 1 to n - 1 do
if currentMax <A[i] then
  currentMax ← A[i]
return currentMax

Basic operation Method

- **Input Size**: n
- **Basic Operation:**Comparison in the for loop
- Depends on worst case or best case? No, has to go through the entire array
- $T(n)$ = number of comparisons
- $T(n) = \sum_{i=1}^{n} 1 = n\text{-}1$

Basic operation Method

Matrix multiplication

ALGORITHM *MatrixMultiplication*$(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])$

```
//Multiplies two square matrices of order n by the definition-based algorithm
//Input: Two n × n matrices A and B
//Output: Matrix C = AB
for i ← 0 to n − 1 do
    for j ← 0 to n − 1 do
        C[i, j] ← 0.0
        for k ← 0 to n − 1 do
            C[i, j] ← C[i, j] + A[i, k] * B[k, j]
return C
```


Basic operation Method

Matrix multiplication

- **Input Size**: n
- **Basic Operation** :Multiplication
- Depends on worst case or best case? No, has to go through the entire array
- $M(n)$ = number of comparisons

Basic operation Method

Matrix multiplication

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1.$$

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = n^3.$$

Basic operation Method

- Element uniqueness problem

ALGORITHM *UniqueElements*$(A[0..n-1])$

```
//Determines whether all the elements in a given array are distinct
//Input: An array A[0..n − 1]
//Output: Returns "true" if all the elements in A are distinct
//                and "false" otherwise
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] = A[j] return false
return true
```

Basic operation Method

- Element uniqueness problem
- **Input's size:**n, the number of elements in thearray.
- **Algorithm's basic operation:**The comparison of two elements
- The number of element comparisons depends not only on n but also on whether there are equal elements in the array and, if there are, which array positions they occupy.
- We will limit our investigation to the worst case only

Basic operation Method

- **Worst Case of this problem:** The worst case input is an array for which the number of element comparisons is the largest among all arrays of size n
- **Two kinds of worst-case inputs**:
- The algorithm does not exit the loop prematurely - arrays with no equal elements
- The last two elements are the only pair of equal elements

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 \in \quad (n^2).$$

Order of growth

- How the value of 'n' affects the time complexity of the algorithm?

OR

- How time complexity grows wrt 'n'?

Order of growth

- The running time of an algorithm can be

CONSTANT	**1**
LOGARITHMIC	**Log N**
LINEAR	**N**
N Log N	**-**
QUADRATIC	**N^2**
CUBIC	**N^3**
EXPONENTIAL	**2^N**
FACTORIAL	**N!**

Order of growth Exercise

- Compare the order of growth
  - $n(n+1)$ and $200n^2$ --- Same
  - $100n^2$ and $0.01\ n^3$
  - Log n and ln n
  - 2 ^n-1 and 2 ^n

Refer 1.3.2 in text book for Logarithms and Exponents

Order of growth

Consider a program with time complexity $O(n^2)$.

- For the input of size n, it takes 5 seconds.
- If the input size is doubled (2n). –
- **then it takes 20 seconds**.

Consider a program with time complexity O(n).

- For the input of size n, it takes 5 seconds.
- If the input size is doubled (2n). –
- **then it takes 10 seconds.**

Consider a program with time complexity $O(n^3)$.

- For the input of size n, it takes 5 seconds.
- If the input size is doubled (2n) . –
- **then it takes 40 seconds**.

Order of growth

	constant	*logarithmic*	*linear*	*N-log-N*	*quadratic*	*cubic*	*exponential*
n	O(1)	O(log *n*)	O(*n*)	O(*n* log *n*)	O(n^2)	O(n^3)	O(2^n)
1	1	1	1	1	1	1	2
2	1	1	2	2	4	8	4
4	1	2	4	8	16	64	16
8	1	3	8	24	64	512	256
16	1	4	16	64	256	4,096	65536
32	1	5	32	160	1,024	32,768	4,294,967,296
64	1	6	64	384	4,069	262,144	1.84×10^{19}

Exercise

- Which kind of growth best characterizes each of these functions?
- $(3/2)^n$
- 3n
- 1
- (3/2)n
- $2n^3$
- 2^n
- $3n^2$
- 1000

Exercise-Solution

- Which kind of growth best characterizes each of these functions?

	Constant	Linear	Polynomial	Exponential
(3/2)^n				✓
3n		✓		
1	✓			
(3/2)n		✓		
2n^3			✓	
2^n				✓
3n^2			✓	
1000	✓			

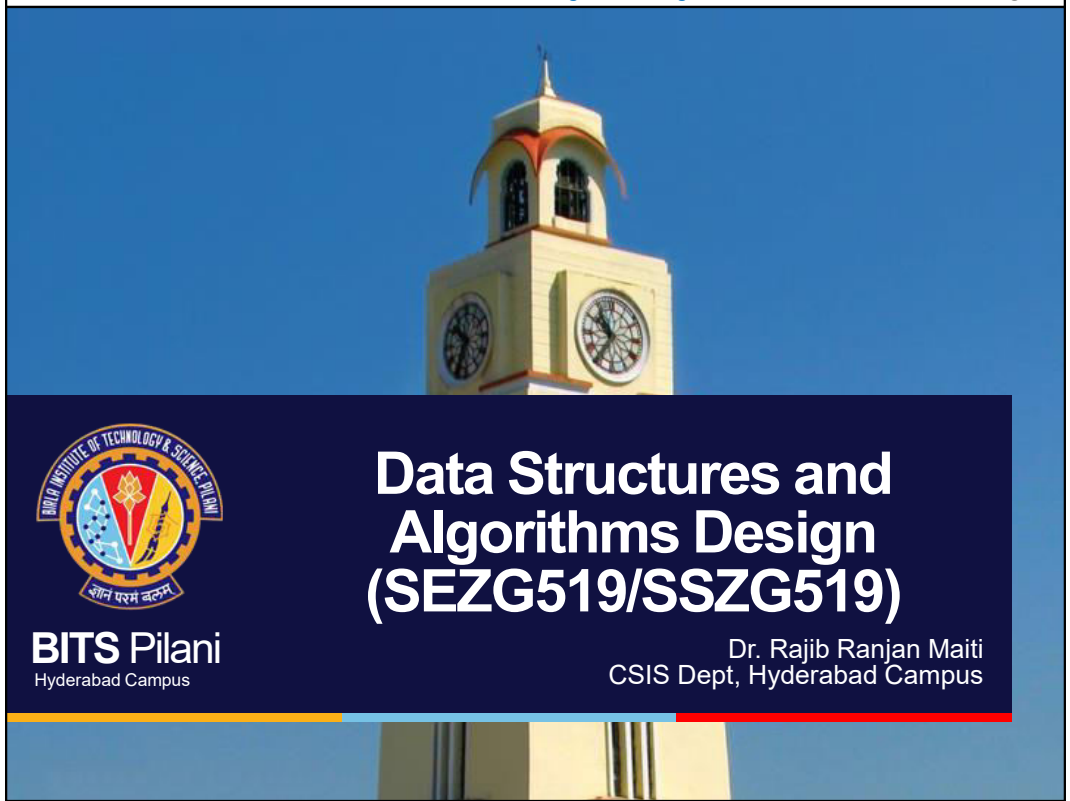
Examples

What is the order of growth of the below function?

```
int fun1(int n)
{
  int count = 0;
  for (int i = 0; i < n; i++)
    for (int j = i; j > 0; j--)
      count = count + 1;
  return count;
}
```

Ans:O(n^2)

THANK YOU!



innovate achieve lead

BITS Pilani
Hyderabad Campus

S1 Algorithms, RAM, Time Complexity, Notation of Correctness

Content of S1

innovate achieve lead

1. Algorithms and it's Specification
2. Random Access Machine Model
3. Notion of best case, average case and worst case
4. Notion of Algorithm Correctness

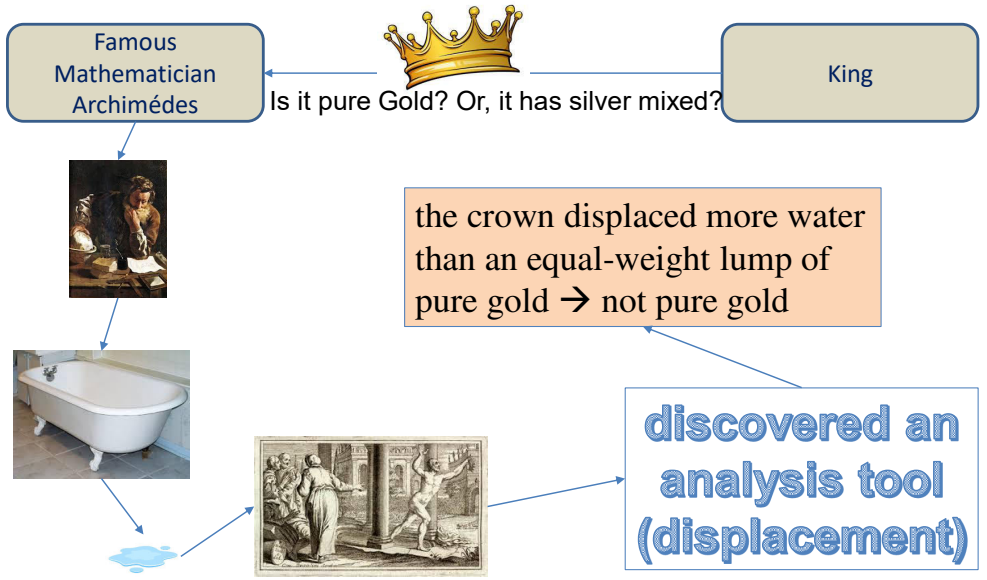
Content of S1

innovate achieve lead

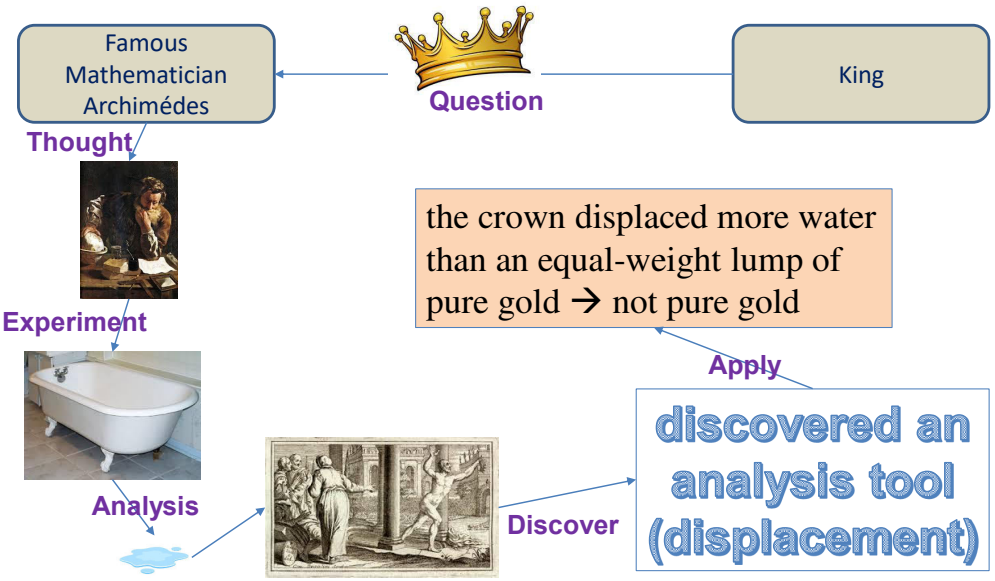
1. **Algorithms and it's Specification**
2. Random Access Machine Model
3. Notion of best case, average case and worst case
4. Notion of Algorithm Correctness

Discovery of A Tool

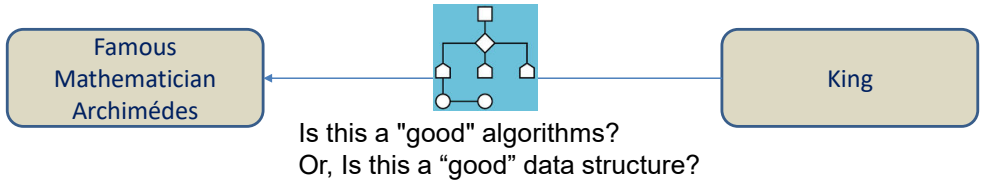
innovate achieve lead



Discovery of A Tool

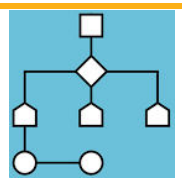


How analogy fits to Algorithm?

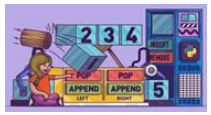


Is this a "good" algorithms?
Or, Is this a “good” data structure?

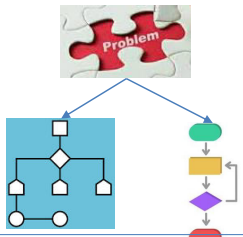
But, what is an algorithm?



An algorithm: a **step-by-step** procedure for performing some task in a **finite amount of time**



A data structure: a systematic way of **organizing** and **accessing** data.

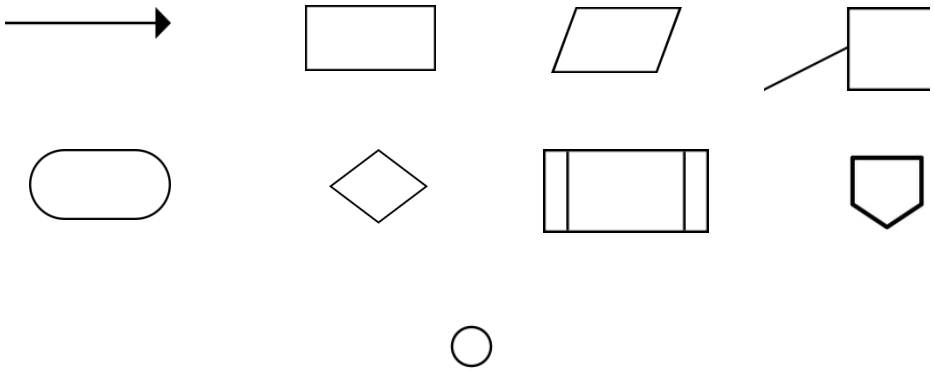


Which one is “good”?
What is the measure of “goodness”?

How to develop an algorithmic thought?

Answer: Flowchart
ANSI set standards for flowcharts and their symbols in the 1960s

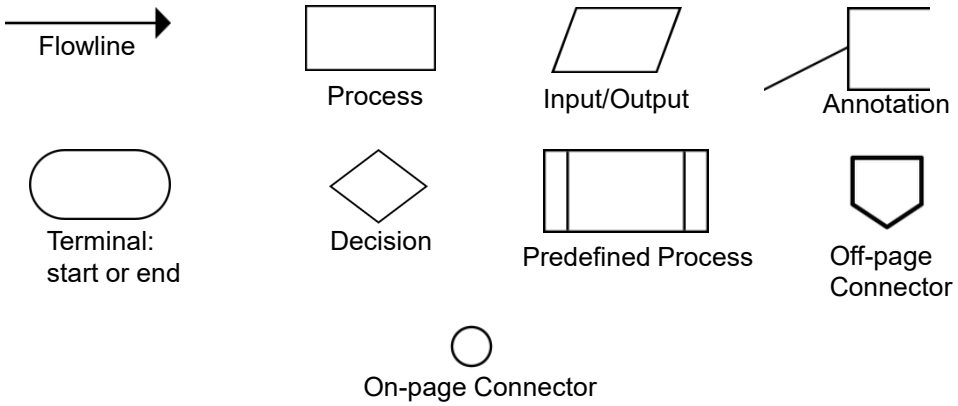
Flowchart
flows from top to bottom and left to right



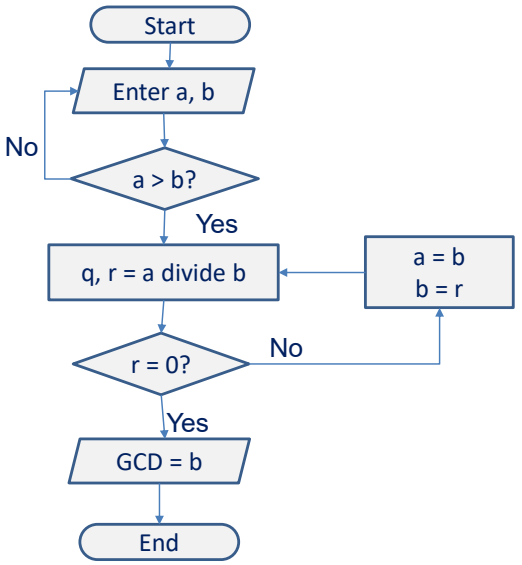
How to develop an algorithmic thought?

Answer: Flowchart
ANSI set standards for flowcharts and their symbols in the 1960s

Flowchart
flows from top to bottom and left to right



Example of Flowchart: GCD



How to Write an Algorithm?

A Computational Problem

- Algorithm
  - Well defined Computational steps
  - Time Complexity: An estimation of execution time based on input size, not based on number of program statements
  - Based on well defined estimated time of individual steps
- Program
  - An exact sequence of statement written in a particular programming language
  - Execution time: depends on instruction set architecture, processor speed, operating system, compiler or interpreter
  - Still get an average run time for the program

Advantage of Run time

When we are interested in an accurate run time measurement, a program would need to be executed and, during execution, exact time for system calls and user functions can be calculated

Obviously, such a time measurement will depend on the language, it library, operating system and so on

However, we can then visualize the results of such experiments by plotting the execution time of each run of the program on y-axis and input size on x-coordinate

An example of run time analysis

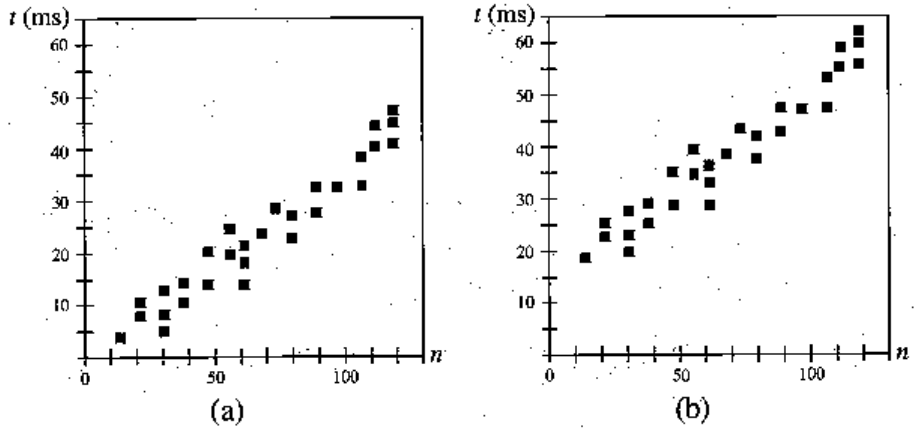


Figure 1.1: Results of an experimental study on the running time of a program. A dot with coordinates (n, t) indicates that on an input of size n, the running time of the program is t milliseconds (ms)
(a) The algorithm executed on a fast computer;
(b) the algorithm executed on a slow computer.

Limitation of Run time of a program

- Experiments can be done only on a limited set of test inputs, and care must be taken to make sure these are representative.
- It is difficult to compare the efficiency of two algorithms unless experiments on their running times have been performed in the same hardware and software environments.
- It is necessary to implement and execute an algorithm in order to study its running time experimentally.

Advantage of Time Complexity of Algorithm

- Takes into account all possible inputs
- Evaluates the relative efficiency of two algorithms independent of the hardware and software environment
- Performs a study on a high-level description of the algorithm without actually implementing or running it

Algorithm ← attach — f(n)

the estimated run time of the algorithm in terms of the input size n

For example, "Algorithm A runs in time proportional to n" → if we were to perform experiments, we would find that the actual running time of algorithm A on any input of size n never exceeds cn, where c is a constant that depends on the hardware and software environment used in the experiment.

Language of Algorithm

- A language for describing algorithms
- A computational model that algorithms execute
- A metric for measuring algorithm running time
- An approach for characterizing running times, including those for recursive algorithms.

Pseudocode

A mixture of natural language and high-level programming constructs that describe the main ideas behind a generic implementation of a data structure or algorithm.

Expression:
1) standard mathematical symbols to express numeric and Boolean expressions
2) left arrow (←) is the assignment operator
3) equal sign (=) is the equality relation in Boolean expressions

Method declarations:
Algorithm name(paraml , param2 ,...)

Decision structures:
1) if condition, then true-actions [else false-actions].

indentation to indicate a block of expressions

Pseudocode

Loops:
1) While-loops: while condition do actions.
2) Repeat-loops: repeat actions until condition.
3) For-loops for variable-increment-definition do actions

indentation to indicate a block of expressions

Array indexing:
1) A[i] represents the ith cell in the array A
2) The cells' of an n-celled array A are indexed from A[0] to A[n-1]

Method calls: object.method(args) (object is optional)
Method returns: return value

Example Pseudocode

Problem: Find maximum number in and array of n numbers

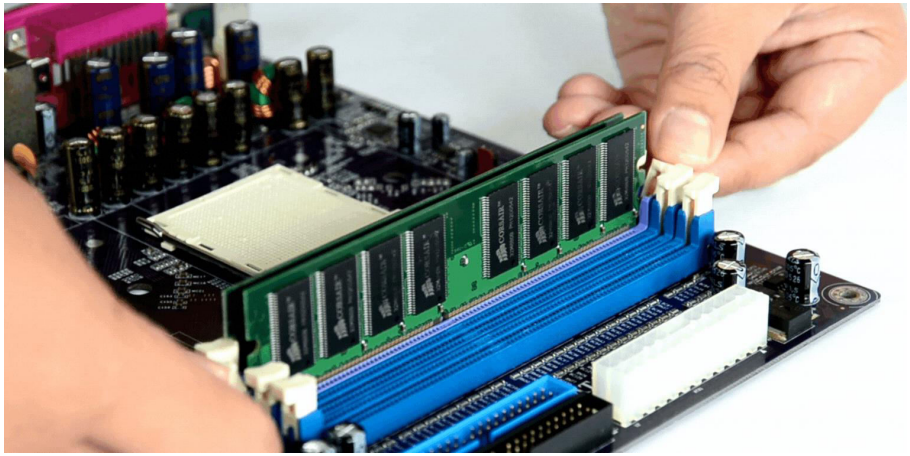
Algorithm arrayMax(A, n):
Input: An array A storing $n \geq 1$ integers.
Output: The maximum element in A.

$currentMax \leftarrow A[0]$
for $i \leftarrow 1$ **to** $n-1$ **do**
  if $currentMax < A[i]$ **then**
    $currentMax \leftarrow A[i]$
return $currentMax$

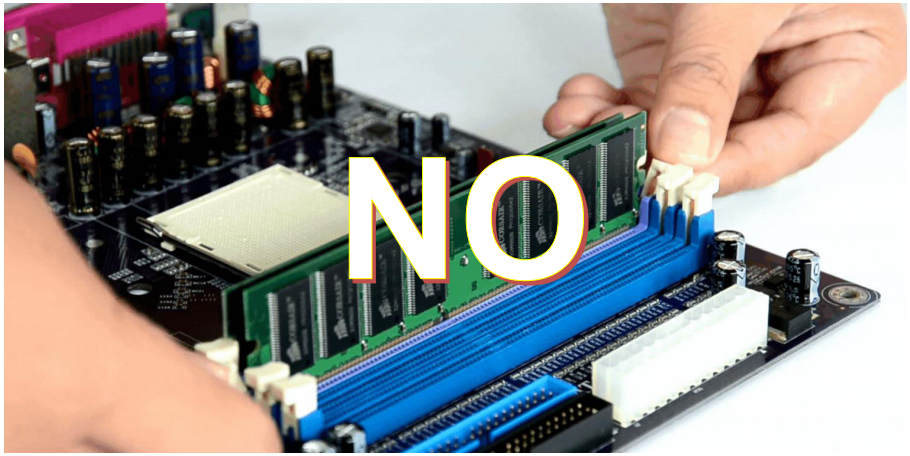
Content of S1

1. Algorithms and it's Specification
2. **Random Access Machine Model**
3. Notion of best case, average case and worst case
4. Notion of Algorithm Correctness

Random Access Machine (RAM) Model



Random Access Machine (RAM) Model



Analysis on Pseudocode

- Experimental analysis is valuable, but has limitations
- If we wish to analyze a particular algorithm without performing experiments on running time, then we can analyze high-level code or pseudo-code
- For that to happen, we must identify high-level primitive operations
- Primitive operations
  - **independent of the programming language**
  - **available in pseudocode**

Analysis on Pseudocode

- What are primitive operations?

1) Assigning a value to variable	5) Calling a method
2) Performing an arithmetic operation	6) Comparing two basic numbers
3) Indexing into an array	7) Following an object reference
4) Returning from a method	

Instead of trying to determine the specific execution time of each primitive operation, we will simply count how many primitive operations are executed.
We shall use this number t as a high-level estimate of the running time of the algorithm

References

1. Algorithms Design: Foundations, Analysis and Internet Examples Michael T. Goodrich, Roberto Tamassia, 2006, Wiley (Students Edition)
2. Data Structures, Algorithms and Applications in C++, Sartaj Sahni, Second Ed, 2005, Universities Press
3. Introduction to Algorithms, TH Cormen, CE Leiserson, RL Rivest, C Stein, Third Ed, 2009, PHI

Theoretical Foundation

Data structure

- Data: value or set of values. E.g. Rohit, 34, {11,33,2,51}, etc.
- Data structure: Logical or mathematical model of particular organization of data is called data structure.
- Array: A list of finite number of similar data elements. A[1], A[2], ..., A[N].
- Linked List: A list linked to one another.
- Stack: A linear list following Last In First Out system (LIFO).
- Queue: A linear list following First In First Out system (FIFO).
- Graph: A non-linear list consisting vertices and edges.
- Tree: A graph without cycles.

Theoretical Foundation

Data structure operations

- Traversing
- Searching
- Inserting
- Deleting
- Sorting
- Merging

Any Question!!

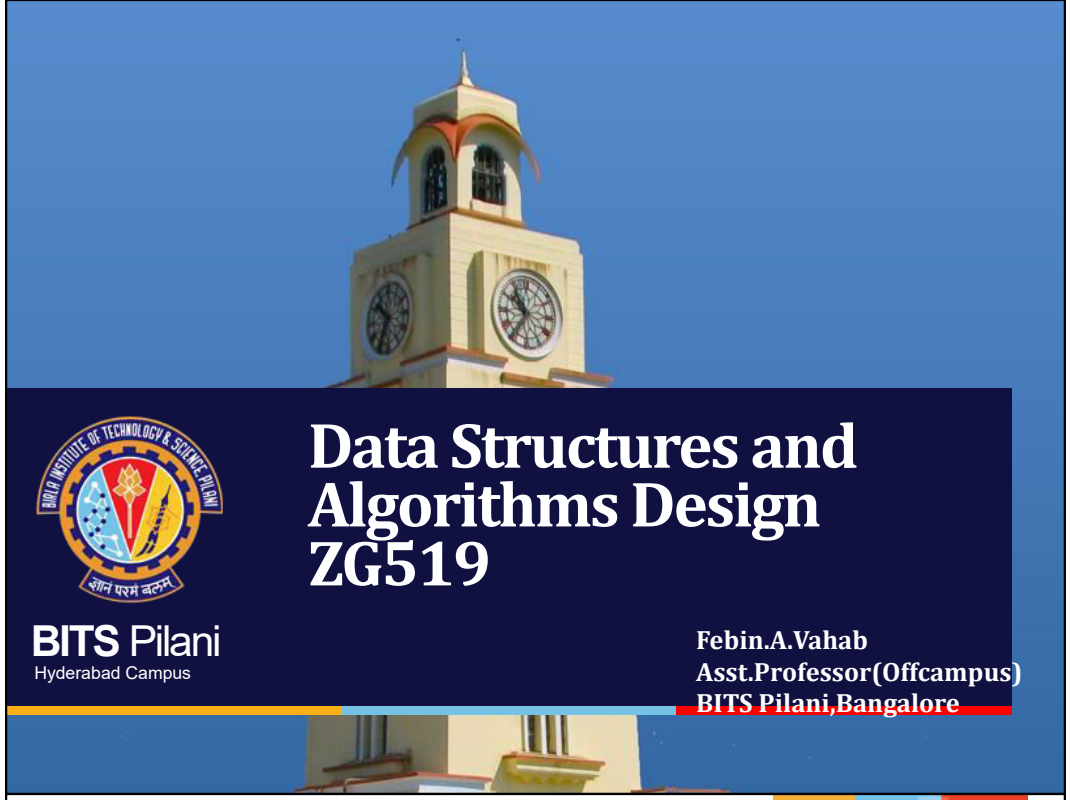
BITS Pilani
Hyderabad Campus

Thank you!!

SESSION 2 -PLAN

innovate achieve lead

Online Sessions(#)	List of Topic Title	Text/Ref Book/external resource
2	Notion of best case, average case and worst case. Use of asymptotic notations- Big-Oh, Omega and Theta Notations. Correctness of Algorithms.	T1: 1.4, 2.1



BITS Pilani
Hyderabad Campus

Data Structures and Algorithms Design ZG519

Febin.A.Vahab
Asst.Professor(Offcampus)
BITS Pilani,Bangalore

Time Complexity- Why should we care?

innovate achieve lead

for i =2 to n-1
If i divides n
n is not prime

---

1 ms for a division
In worst case (n-2) times.

n =11 ------?
n = 101 -------?

for i ← 2 to √n
if i divides n
n is not prime

---

1 ms for a division
In worst case (√n-1) times.

n=11,(3-1) = 2ms
n=101,(√101-1) times = 9ms

Notion of best case and worst case

- Best case: where algorithm takes the least time to execute.
  - In arrayMax ex, occurs when A[0] is the maximum element.
  - T(n)=5n
- Worst case :where algorithm takes maximum time.
  - Occurs when elements are sorted in increasing order so that variable *currentMax* is reassigned at each iteration of the loop.
  - T(n)=7n-2

```
Algorithm arrayMax(A,n)
    currentMax ← A[0]
    for (i =1; i<n; i++)
    if A[i] > currentMax then
    currentMax ← A[i]
    return currentMax
```


Use of asymptotic notation

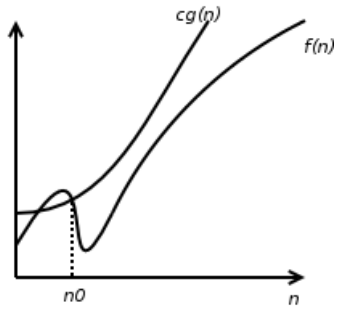
- How the running time of an algorithm increases with the input size, as the size of the input increases without bound?
- Used to compare the algorithms based on the order of growth of their basic operations.

Informal Introduction

- *O(g(n))* is the set of all functions with a lower or same order of growth as *g(n)* (to within a constant multiple, as *n* goes to infinity)
- Ω(*g(n)),* stands for the set of all functions with a higher or same order of growth as *g(n)* (to within a constant multiple, as *n* goes to infinity).
- *Θ(g(n))* is the set of all functions that have the same order of growth as *g(n)* (to within a constant multiple, as *n* goes to infinity).

Big-Oh Notation

- Let f and g be functions from nonnegative numbers to nonnegative numbers. Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there is a real constant c>0 and an integer constant n_0 >=1 such that $f(n) \leq cg(n)$ for every integer $n \geq n_0$



Big-Oh Notation

- Big-Oh notation provides an upper bound on a function to within a constant factor.
- To prove big-Oh, find witnesses, specific values for C and n0, and prove n >= n0 implies f(n) ≤ C ∗ g(n).

One Approach for Finding Witnesses

- *Generate a table for f(n) and g(n) using n = 1, n = 10 and n = 100. [Use values smaller than 10 and 100 if you wish.]*
- *Guess C = ⌈f(1)/g(1)⌉ (or C = ⌈f(10)/g(10)⌉).*
- *Check that f(10) ≤ C ∗ g(10) and f(100) ≤ C ∗ g(100).[If this is not true, f(n) might not be O(g(n)).]*
- *Choose n0 = 1 (or n0 = 10).*
- *Prove that ∀n(n >= n0 → f(n) ≤ C ∗ g(n)).*
- *[It's ok if you end up with a larger, but still constant, value for C.]*

One Approach for Finding Witnesses

- Assume n > 1 if you chose n0 = 1 (or n > 10 if you chose n0 = 10).
- To prove f(n) ≤ C ∗ g(n), you need to find expressions larger than f(n) and smaller than C ∗ g(n).
- If the lowest-order term is negative, just eliminate it to obtain a larger expression.
- Repeatedly use n > k and 2n > 2k and 3n > 3k and so on to "convert" the lowest-order term into a higher-order term.
- Check that your expressions are less than C ∗ g(n) by using n = 100.

Big-Oh Notation Example 1

Show that 3n + 7 is O(n).

- In this case, f(n) = 3n + 7 and g(n) = n.

n	f(n)	g(n)	Ceil(f(n)/g(n))
1	10	1	10
10	37	10	4
100	307	100	4

- This table suggests trying n0 = 1 and c = 10 or
- n0 = 10 and c = 4.
- Proving either one is good enough to prove big-Oh.

Big-Oh Notation Example 1

Try n0 = 1 and c = 10.

- Want to prove ***n > 1 implies 3n + 7 ≤ 10n.***
- Assume n > 1. Want to show ***3n + 7 ≤ 10n.***
- 7 is the lowest-order term, so work on that first.
- **n > 1 implies 7n > 7**, which implies
- 3n + 7 < 3n + 7n = 10n.
- This finishes the proof.

Big-Oh Notation Example 2

- Show that $n^2 + 2n + 1$ is $O(n^2)$.
  - In this case, f(n) = n^2 + 2n + 1 and g(n) = n^2.

n	f(n)	g(n)	Ceil(f(n)/g(n))
1	4	1	4
10	121	100	2
100	10201	10000	2

  - This table suggests trying n0 = 1 and C = 4
  - or n0 = 10 and C = 2.

Big-Oh Notation Example 2

- Try n0 = 1 and c = 4.
  - Want to prove ***n > 1 implies $n^2 + 2n + 1 \leq 4\,n^2$.***
  - Assume n > 1.
  - Want to show $n^2 + 2n + 1 \leq 4\,n^2$.
  - Work on the lowest-order term first.
  - n > 1 implies
  - $n^2 + 2n + 1 < n^2 + 2n + n = n^2 + 3n$
  - Now 3n is the lowest-order term.
  - n > 1 implies 3n > 3 and $3\,n^2 > 3n$, which implies
  - $n^2 + 3n < n^2 + (3n)n$

  $= n^2 + 3\,n^2 = 4\,n^2$. This finishes the proof.

Big-Oh Notation Example 3

- Example
- 2n+10 is O(n)

  ie.

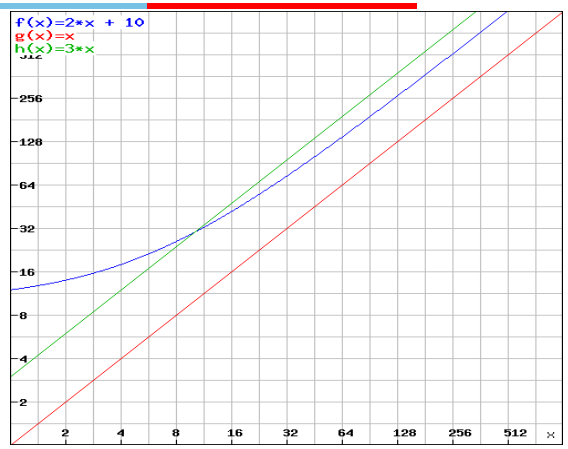
  2n+10 <=c*n

  10/n <=(c-2)

  10/c-2 <=n

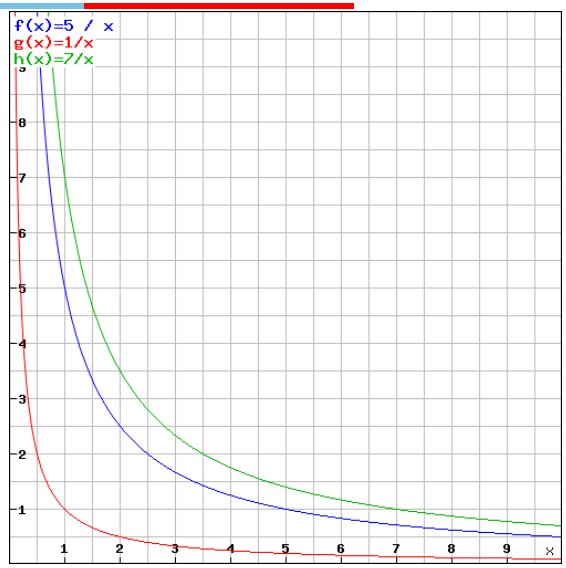
  n >=10/(c-2)

  ie .If c=3,n=10



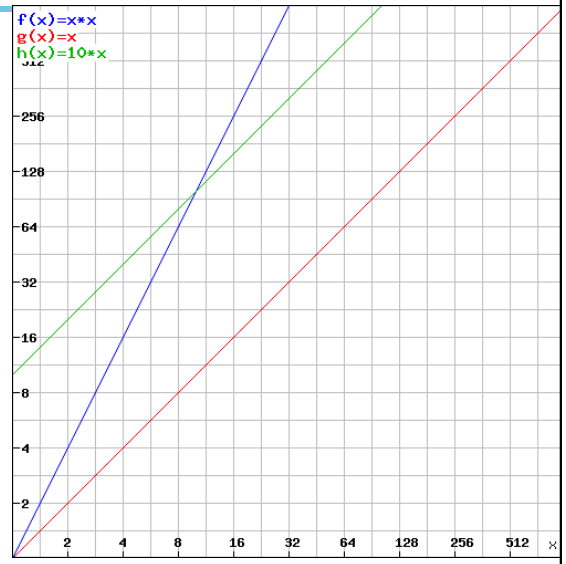
Big-Oh Notation Example 4

- Example
- 5/x is O(1/x)
- 5/x<=c*1/x
- c>=5 for x>=1



Big-Oh Notation Example 5

- Example:
- The function $\boldsymbol{n}^2$ is not $\boldsymbol{O(n)}$
  - $\boldsymbol{n}^2 \leq \boldsymbol{cn}$
  - $\boldsymbol{n} \leq \boldsymbol{c}$
  - The above inequality cannot be satisfied since $\boldsymbol{c}$ must be a positive constant



Big-Oh Notation Example 6

- Show that $8n^3 - 12n^2 + 6n - 1$ is $O(n^3)$.
  - In this case,$f(n) = 8n^3 - 12n^2 + 6n - 1$ and $g(n) = n^3$

n	f(n)	g(n)	Ceil(f(n)/g(n))
1	1	1	1
10	6859	1000	7
100	7880599	1000000	8

  - This table suggests trying n0 = 100 and C = 8.

Big-Oh Notation Example 6

- Try n0 = 100 and c = 8.
  - Want to prove $n > 100$ implies $8n^3 - 12n^2 + 6n - 1 \leq 8n^3$
  - Assume $n > 100$. Want to show $f(n) \leq 8n^3$.
  - The lowest-order term is negative, so eliminate it.
  - $8n^3 - 12n^2 + 6n - 1 < 8n^3 - 12n^2 + 6n$.
  - $n > 100$ implies $n > 6$, $n^2 > 6n$ which implies
  - $8n^3 - 12n^2 + 6n < 8n^3 - 12n^2 + n^2 = 8n^3 - 11n^2$.
  - Now lowest-order term is negative, so eliminate.
  - $n > 100$ implies $8n^3 - 11n^2 \leq 8n^3$.
  - This finishes the proof.

Big-Oh Notation -More examples

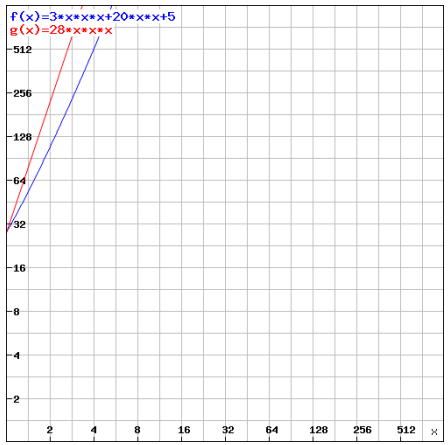
- 7n-2 is O(n)
- $3n^3 + 20n^2 + 5$ is $O(n^3)$
- 3 log n+ log logn is O(log n)

- Solution using one method is given below. Try other one.

Big-Oh Notation -More examples

- 7n-2 is O(n)
  - 7n-2<=cn
  - 7-2/n<=c
  - c>=7-2/n
  - **n0=1 and c=7** is true .
- $3n^3 + 20n^2 + 5$ is $O(n^3)$
  - $3n^3+20n^2+5 <= c.n^3$
  - $3+20/n+5/n^3 <= c$
  - $c >= 3+20/n+5/n^3$

    c>=28 and n0>=1 is true



Big-Oh Notation -More examples

- 3 log n+ log log n is O(log n)
  - 3log n+ log logn< c.log n
  - Let n=8,
  - 3*3+log 3<= 3c
  - 9+1.58 <=3c
  - c >=4

  OR
- 3 log n + log log n <=4 logn , for n >=2 .
  - Note that log log n i s not even defined for n = 1 . That is why we use n >= 2.

Big-Oh Notation

- If $f(n)$ a polynomial of degree d, then $f(n)$ is $O(n^d)$, i.e.,
  1. Drop lower-order terms
  2. Drop constant factors
- Use the smallest possible class of functions
  - Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"
- Use the simplest expression of the class
  - Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"

Big-Oh Notation:Theorem

Let d(n), e(n),f(n),and g(n) be functions mapping nonnegative integers to nonnegative reals. Then

1. If d(n) is O(f(n)), then ad(n) is O(f(n)), for any constant a> 0.
2. If d(n) is O(f(n)) and e(n) is O(g(n)), then d(n) + e(n) is O(f(n) + g(n)).
3. If d(n) is O(f(n)) and e(n) is O(g(n)), then d(n)e(n) is O(f(n)g(n)).
4. If d(n) is O(f(n)) and f(n) is O(g(n)), then d(n) is O(g(n)).
5. n^x is O(a^n) for any fixed x > 0 and a> 1.
6. Log n^x is O(log n) for any fixed x > 0.

Big-Oh Notation:Proof of Theorem

1.If d(n) is O(f(n)), then a*d(n) is O(f(n)) for any constant a>0.

- d(n) ≤ C * f(n) where C is a constant
- a * d(n) ≤ a * C * f(n)
- a * d(n) ≤ C1 * f(n) where a* C = C1
- Therefore a*d(n) = O(f(n))

Big-Oh Notation:Proof of Theorem

2. If d(n) is O(f(n)) and e(n) is O(g(n)), then d(n)+e(n) is O(f(n)+g(n)). The proof will extend to orders of growth

- d(n) ≤ C1 * f(n) for all n ≥ n1 where C1 is a constant
- e(n) ≤ C2 * g(n) all n ≥ n2 where C2 is a constant
- d(n) + e(n) ≤ C1 * f(n) + C2 * g(n)
- ≤ C3 (f(n) + g(n)) where C3=max{C1,C2}
- and n ≥max{n1,n2}

Big-Oh Notation:Proof of Theorem

6. Log n^x is O(log n) for any fixed x > 0.

log n^x <= c.log n

x * log n <=c. log n

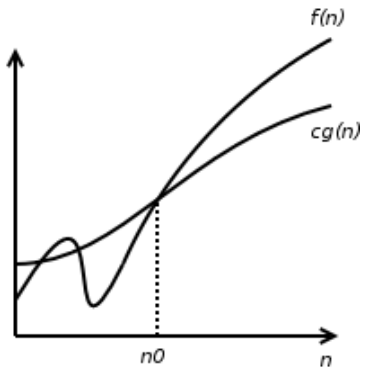
c >= x.

Big-Omega Notation

- The function f(n) is said to be in Ω(g(n)) iff there exists a positive constant c and a positive integer n0 such that

  f(n)>= c.g(n) for all n>=n0.

- Asymptotic lower bound
- n^3 ∈ Ω(n^2)
- n^5+n+3 ∈ Ω(n^4)



Big-Omega Notation

- Big-Omega notation provides a lower bound on a function to within a constant factor.
- To prove big-Omega, find witnesses, specific values for C and n0, and prove n > n0 implies f(n) ≥ C ∗ g(n).

Tricks for Proving Big-Omega

- Assume n > 1 if you chose n0 = 1 (or n > 10 if you chose n0 = 10).
- To prove f(n) ≥ C ∗ g(n), you need to find expressions smaller than f(n) and larger than C ∗ g(n).
- If the lowest-order term is positive, just eliminate it to obtain a larger expression.
- Repeatedly use −n0 > −n and −0.1n0> −0.1n and so on to "convert" the lowest-order term into a higher-order term.
- Check that your expressions are greater than C ∗ g(n) by using n = 100.

Tricks for Proving Big-Omega

- Generate a table for f(n) and g(n). using n = 1, n = 10 and n = 100.[Use values smaller than 10 and 100 if you wish.]
- Guess 1/C = [g(1)/f(1)] (or more likely 1/C = [g(10)/f(10)]).
- Check that f(10) ≥ C ∗ g(10) and f(100) ≥ C ∗ g(100).[If this is not true, f(n) might not be (g(n)).]
- Choose n0 = 1 (or n0 = 10).
- Prove that ∀n(n > n0 → f(n) ≥ C ∗ g(n)).[It's ok if you end up with a smaller, but still positive, value for C.]

Big-Omega Example 1

- Show that 3n + 7 is Ω(n).
  - In this case, f(n) = 3n + 7 and g(n) = n.

n	f(n)	g(n)	Ceil(g(n)/f(n))	C
1	10	1	1	1
10	37	10	1	1
100	307	100	1	1

  - This table suggests trying n0 = 1 and C = 1.
  - Want to prove **n > 1 implies 3n + 7 ≥ n.**
  - n > 1 implies 3n + 7 > 3n > n.

Big-Omega Example 3

- Show that n^2 − 2n + 1 is Ω(n^2).
- In this case, f(n) = n^2 − 2n + 1 and g(n) = n^2.

n	f(n)	g(n)	Ceil(g(n)/f(n))	C
1	0	1	-	-
10	81	100	2	1/2
100	9801	10000	1	1/2

- This table suggests trying n0 = 10 and C = 1/2.

Big-Omega Example 2

- Try n0 = 10 and C = 1/2.
  - Want to prove n > 10 implies n^2 − 2n + 1 ≥ n^2/2.
  - Assume n > 10. Want to show f(n) ≥ n^2/2.
  - The lowest-order term is positive, so eliminate.
  - n^2 − 2n + 1 > n^2 − 2n
  - n > 10 implies −10 > −n, implies −2 > −0.2n.
  - −2 > −0.2n implies n^2 − 2n > n^2 − 0.2n^2 = 0.8n^2.
  - n > 10 implies 0.8n^2 > n^2/2.
  - This finishes the proof.

Big-Omega Example 3

- Show that $n^3/8 - n^2/12 - n/6 - 1$ is $O(n^3)$.
- In this case, $f(n) = n^3/8 - n^2/12 - n/6 - 1$ and $g(n) = n^3$.

n	f(n)	g(n)	Ceil(g(n)/f(n))	C
1	-8	1	-1	-1
10	117.3	1000	9	1/9
100	124,182.3	1000000	9	1/9

- C = −1 is useless, so try n0 = 10 and C = 1/9

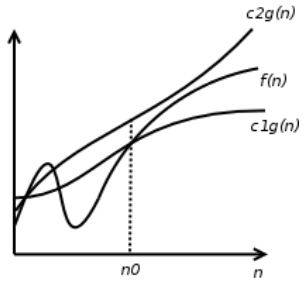
Big-Omega Example 3

- Try n0 = 10 and C = 1/9.
  - Want to prove n > 10 implies $n^3/8 - n^2/12 - n/6 - 1 \geq n^3/9$
  - Assume n > 10, which implies the following:
  - $n^3/8 - n^2/12 - n/6 - 1$
  - $= (3n^3 - 2n^2 - 4n - 24)/24$
  - $> (3n^3 - 2n^2 - 4n - 2.4n)/24$
  - $> (3n^3 - 2n^2 - 7n)/24$
  - $> (3n^3 - 2n^2 - 0.7n^2)/24$
  - $> (3n^3 - 3n^2)/24$
  - $> (3n^3 - 0.3\ n^3)/24$
  - $> (3n^3 - n^3)/24$
  - $= (2n^3)/24 = n^3/12$
  - Ended up with n0 = 10 and C = 1/12, proving
  - n > 10 implies $n^3/8 - n^2/12 - n/6 - 1 \geq n^3/12$

Big-Theta Notation

- The function f(n) is said to be in $\Theta(g(n))$ iff there exists some positive constants c1 and c2 and a non negative integer n0 such that

  c1.g(n)<=f(n)<=c2.g(n) for all n>=n0
- **Asymptotic tight bound**
- an^2+bn+c ∈ Θ(n^2)
- n^2 ∈ Θ(n^2)



Examples – Ω and Θ

- **f(n)=5n^2 .Prove that f(n) is Ω(n)**
  - 5n^2>=c.n
  - c.n<=5n^2
  - c<=5n
  - If n=1,c<=5
  - 5*1<=4*1 hence the proof.

Examples – Ω and Θ

- **f(n)=5n^2 .Prove that f(n) is Ω(n)**
  - 5n^2>=c.n
  - c.n<=5n^2
  - c<=5n
  - If n=1,c<=5
  - 5*1<=4*1 hence the proof.
- Prove that f(n) is Θ(n^2)

Little-Oh and little omega Notation

- f(n) is o(g(n)) (or f(n) ∈ o(g(n))) if for any real constant c > 0, there exists an integer constant n0 ≥ 1 such that
  - ***f(n) < c ∗g(n)*** for every integer n ≥ n0.
- f(n) is ω(g(n)) (or f(n) ∈ ω(g(n))) if for any real constant c > 0, there exists an integer constant n0 ≥ 1 such that
  - ***f(n) > c * g(n)*** for every integer n ≥ n0.

Little-Oh and Little omega Notation

- $12n^2 + 6n$ is $o(n^3)$
- $4n+6$ is $o(n^2)$
- $4n+6$ is $\omega(1)$
- $2n^9 + 1$ is $o(n^{10})$
- n^2 is $\omega(\log n)$

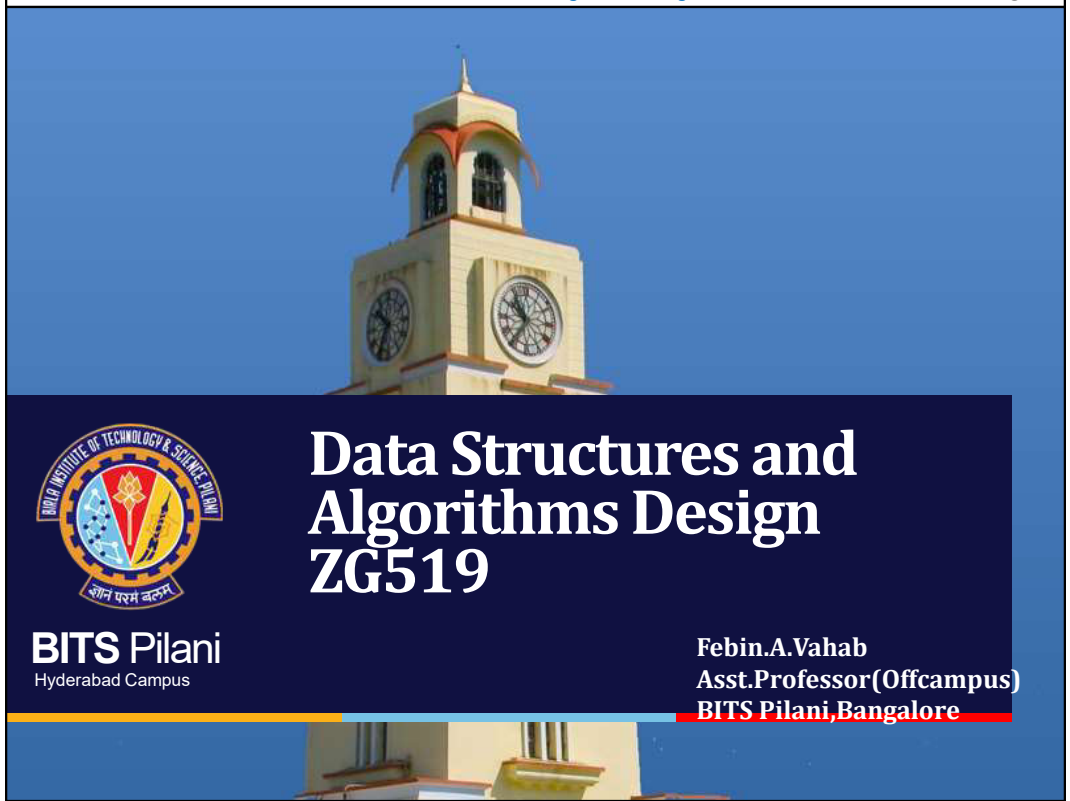
USING LIMITS

Little Oh- $= \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

Little Omega $= \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$

Correctness of algorithm

- An algorithm is said to be correct if, for every input instance, it halts with the correct output.
- When it can be incorrect?
  - Might not halt on all input instances
  - Might halt with an incorrect answer
- Does it makes sense to think of incorrect algorithm?
  - Might be useful if we can control the error rate and can be implemented very fast



SESSION 3 -PLAN

Online Sessions(#)	List of Topic Title	Text/Ref Book/external resource
3	Analyzing Recursive Algorithms: Recurrence relations, Iteration Method, Substitution Method, Recursion Tree, Master Method.	T1: 1.4, 2.1

Analyzing Recursive Algorithms

- Recursive calls:-A procedure P calling itself-calls to P are for solving sub problems of smaller size.
- Recursive procedure call should always define a ***base case***.
- Base case-small enough that it can be solved directly without using recursion.

Analyzing Recursive Algorithms

- Algorithm recursiveMax(A,n)
  // Input : An array A storing n>=1 integers
  //Output: The maximum element in A
  if n = 1 then
  return A[0]
  return max{ recursiveMax(A,n-1),A[n-1]}

Analyzing Recursive Algorithms

- A ***recurrence*** is an equation or inequality that describes a function in terms of its value on smaller inputs.
- ***Recurrence equation***: defines mathematical statements that the running time of a recursive algorithm must satisfy
- Analysis of ***recursiveMax***
  - T(n)-Running time of algorithm on an input size n

$$T(n) = \begin{cases} 3 & \text{if } n=1 \\ T(n-1) + 7 & \text{otherwise} \end{cases}$$

Solving Recurrences

Solving recurrences : Iterative Method

Analyzing Recursive Algorithms- Iterative method

- **General Plan-Iterative Method**
  - Identify the parameter to be considered based on the size of the input.
  - Identify the basic operation in the algorithm
  - Obtain the number of times the basic operation is executed.
  - Obtain an initial condition-base case
  - Obtain a recurrence relation
  - Solve the recurrence relation and obtain the order of growth and express using asymptotic notations.

Analyzing Recursive Algorithms- RecursiveMax

- Analysis of ***recursiveMax***

  T(n)-Running time of algorithm on an input size n

  $$T(n) = \begin{cases} 3 & \text{if } n=1 \\ T(n-1) + 7 & \text{otherwise} \end{cases}$$

```
Algorithm recursiveMax(A,n)
// Input : An array A storing  n>=1 integers
 //Output: The maximum element in A
 if  n = 1 then
 return A[0]
return max{ recursiveMax(A,n-1),A[n-1]}
```


Analyzing Recursive Algorithms- Example **1:-Factorial of a number**

– Algorithm fact(n)

```
//Purpose: Computes factorial of n
//Input: A positive integer n
//Output: factorial of n
If(n=0)
return 1
return n*fact(n-1)
```

Analyzing Recursive Algorithms- Example **1:-Factorial of a number**

- **Analysis**
  - Parameter to be considered –n
  - Basic operation –Multiplication
  - T(n) = 0 if n=0
    1+T(n-1) Otherwise

Time taken to compute fact(n-1)

Time to multiply n*fact(n-1)

Analyzing Recursive Algorithms- Example **1:-Factorial of a number**

- **Solve the recurrence**

$T(n) = T(n-1) + 1$

$[T(n-2) + 1] + 1 = T(n-2) + 2$ substituted $T(n-2)$ for $T(n-1)$

$[T(n-3) + 1] + 2 = T(n-3) + 3$ substituted $T(n-3)$ for $T(n-2)$

.. a pattern evolves

T(n) = 1+T(n-1)

=2+T(n-2)

=3+T(n-3)

=….

=i+T(n-i)

When n=0 T(0)=0,No multiplications

When i=n, T(n) =n+T(n-n)

=n+0

=n **$T(n) \in \Theta(n)$**

Analyzing Recursive Algorithms- Example **2:-Tower of hanoi**

Step 1 – Move n-1 disks from **source** to **temp**

Step 2 – Move n^{th} disk from **source** to **dest**

Step 3 – Move n-1 disks from **temp** to **dest**

Algorithm Hanoi(n, source, dest, temp)

//Input: n :number of disks

//Output :All n disks on dest

If disk = 1

move disk from source to dest

Hanoi(n - 1, source, temp, dest) // Step 1

move nth disk from source to dest // Step 2

Hanoi(n - 1, temp, dest, source) // Step 3

Analyzing Recursive Algorithms- Example **2:-Tower of hanoi**

1.Problem size is n, the number of discs

2. The basic operation is moving a disc from rod to another

3. Base case $M(1) = 1$

4. Recursive relation for moving n discs

$$M(n) = M(n\text{-}1) + 1 + M(n\text{-}1) = 2M(n\text{-}1) + 1$$

Analyzing Recursive Algorithms- Example **2: Tower of hanoi**

Solve using backward substitution

$M(n) = 2M(n\text{-}1) + 1$

$= 2[2M(n\text{-}2) + 1] + 1 = 2^2M(n\text{-}2) + 2+1$

$= 2^2[2M(n\text{-}3) + 1] + 2+1$

$= 2^3M(n\text{-}3) + 2^2 + 2 + 1$

...

$M(n) = 2^iM(n\text{-}i) +$ 2^i-1+2^i-2+....+2^3+2^2+2^1+2^0

$M(n) = 2^iM(n\text{-}i) +$ (2^i-1)/(2-1) It's a GP with a=1,r=2,n=i

$= 2^iM(n\text{-}i) +$ 2^i-1

Analyzing Recursive Algorithms- Example **2:- Tower of hanoi**

When i=n-1

$M(n) = 2^{n-1} M(n-(n-1)) + 2^{n-1} - 1$

$= 2^{n-1} M(1) + 2^{n-1} - 1$

$= 2^{n-1} + 2^{n-1} - 1$

$= 2*2^{n-1} - 1$

$= 2*(2^n / 2) - 1$

$= 2^n - 1$

$\boldsymbol{M(n) \in O(2^n)}$

- Time complexity is exponential
- More computattions even for smaller value of n
- Doesnt necessarily mean algorithm is poor
- Nature of the problem itself is computationally expensive.

Analyzing Recursive Algorithms- Example **3:Exercise**

ALGORITHM *BinRec(n)*

//Input: A positive decimal integer n

//Output: The number of binary digits in n's binary representation

if $n = 1$ **return** 1

else return *BinRec(n/2)* + 1

Let us set up a recurrence and an initial condition for the number of additions $A(n)$ made by the algorithm. The number of additions made in computing

BinRec(n/2) is $A(n/2)$, plus one more addition is made by the algorithm to increase the returned value by 1. This leads to the recurrence

$A(n) = A(n/2) + 1$ for $n > 1$.

$A(1)=0$

Analyzing Recursive Algorithms- Example **3:Exercise**

Base condition A(1)= 0

A(n)=A(n/2)+1

The presence of *n*/2 in the function's argument makes the method of backward substitutions stumble on values of *n* that are not powers of 2. Therefore, the standard approach to solving such a recurrence is to solve it only for $n = 2^k$ and then take advantage of the theorem called the ***smoothness rule,*** which claims that under very broad assumptions the order of growth observed for $n = 2^k$ gives a correct answer about the order of growth for all values of *n.*

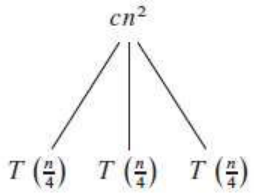
Solving recurrences : Recursion Tree

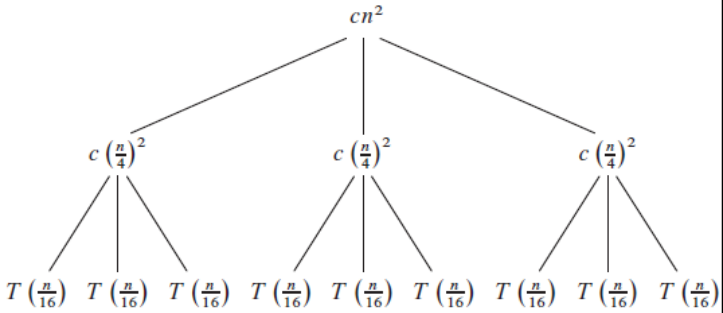
```
Void Test (int n)
{
        if(n>1)
        {
                for (i=1;i<n;i++)
                {
                        stmt;
                }
                Test(n/2);
                Test(n/2);
        }
}
```

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

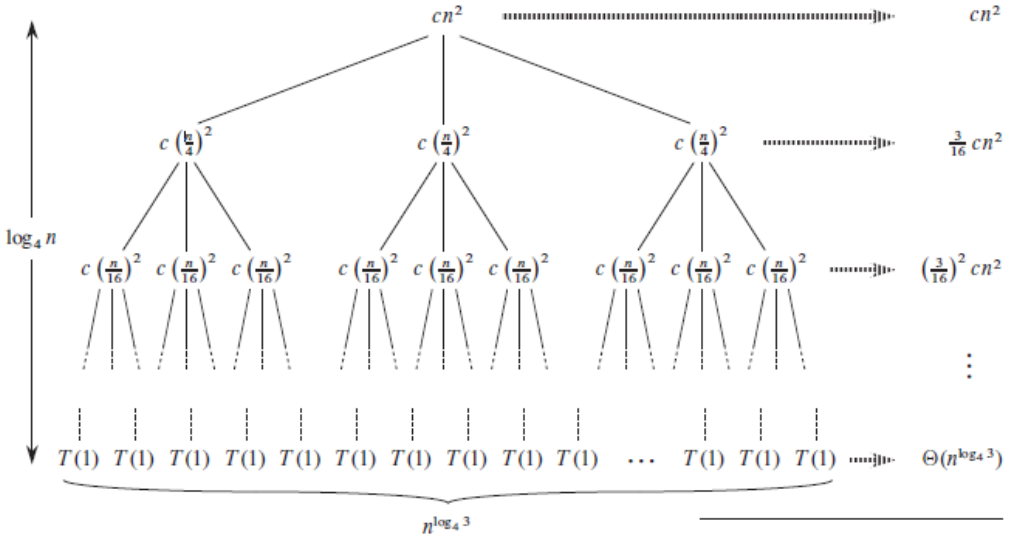
Solving Recurrences-Recursion Tree

- Solve $T(n) = 3T(n/4) + cn^2$.





Solving Recurrences-Recursion Tree



Total: $O(n^2)$

Solving Recurrences-Recursion Tree

$$
\begin{aligned}
T(n) &= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3}) \\
&= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{(3/16)^{\log_4 n} - 1}{(3/16) - 1} cn^2 + \Theta(n^{\log_4 3})
\end{aligned}
$$

Geometric or exponential series

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1}$$

$$
\begin{aligned}
T(n) &= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{1}{1 - (3/16)} cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) \\
&= O(n^2) .
\end{aligned}
$$

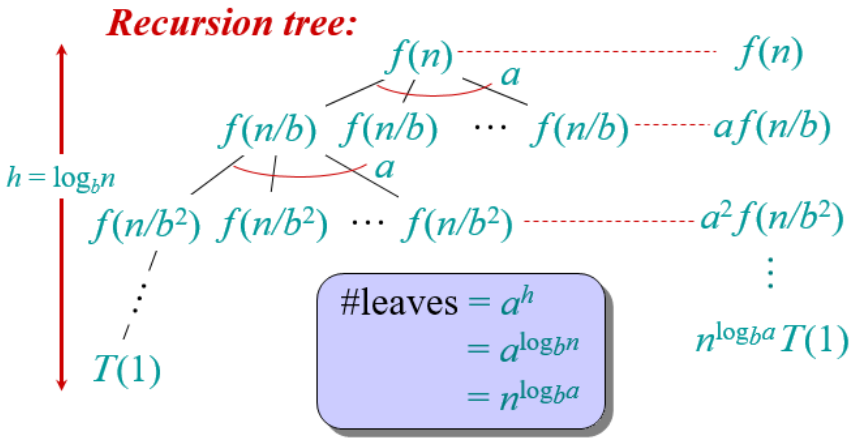
Solving recurrences: Master method Ref: Textbook R2

- The master method applies to recurrences of the form

$$T(n) = a\,T(n/b) + f(n)\,,$$

- where $a \geq 1$, $b > 1$, and f is asymptotically positive.
- (f(n)>0 for n>=n0)

Idea of Master theorem



Solving recurrences: Master method
Ref: Textbook R2

Case 1:

If $f(n)=O(n^{\log_b a-\varepsilon})$, for some constant $\varepsilon > 0$, then $T(n)= \Theta(n^{\log_b a})$

f(n) grows polynomially slower than $n^{\log_b a}$

Case 2:

If $f(n)=\Theta(n^{\log_b a})$, then $T(n)= \Theta(n^{\log_b a} \log n)$

f(n) and $n^{\log_b a}$ grows at similar rates

Case 3:

If $f(n)=\Omega(n^{\log_b a+\varepsilon})$ for some constant $\varepsilon > 0$, and if
$af(n/b)<=cf(n)$ *for some constant $c < 1$ and all sufficiently large n, then $T(n)= \Theta(f(n))$*

f(n) grows polynomially faster than $n^{\log_b a}$

Solving recurrences: Master method
Ref: Textbook R2

Case 2 : (Generalisation):

If there is a constant k >= 0, such that f(n) is $\Theta(n^{\log_b a} \log^k n)$, then T (n) is $\Theta(n^{\log_b a} \log^{k+1} n)$

Example:

$T(n)= 2T(n/2) + n \log n$

$a=2, b=2$ $f(n)=n\log n$

$n^{\log_b a} = n$

f(n) is asymptotically larger than $n^{\log_b a}$, but it is not polynomially larger.

So no standard case of master theorem applies.

It belongs to case 2 general case.

$f(n) = \Theta(n^{\log_b a} \log^k n) = \Theta(n^{\log_b a} \log^1 n)$

So $T(n)= \Theta(n \log^2 n)$

Solving recurrences: Master method

Example 1 : $T(n) = 2T(n/2) + n$

Sol:

Extract a=2, b=2 and f(n) =n

Determine $n^{\log_b a} = n^{\log_2 2} = n^1 = n$

Compare $n^{\log_b a} = n$

$f(n) = n$

Thus case 2: evenly distributed because

$f(n) = \theta(n)$

$T(n) = \theta (n^{\log_b a} \log(n))$

$= \theta (n^1 \log(n))$

$= \theta (n \log n)$

Solving recurrences: Master method

Example 2 : $T(n)=9T(n/3)+n$

a= 9 b=3 and f(n) =n

Determine $n^{\log_b a} = n^{\log_3 9} = n^2$

Compare: $n^{\log_b a} = n^2$

$f(n) = n$

Thus case1; (express f(n) in terms of $n^{\log_b a}$) because $f(n)= O(n^{2-\varepsilon})$

$T(n)= \theta (n^{\log_b a}) = \theta(n^2)$

Solving recurrences: Master method

Example 3 :T(n) = 3T(n/4) + nlogn

a= 3,

b=4,

f(n) = nlogn

Determine; $n^{\log_b a} = n^{\log_4 3}$ $\log_4 3 < 1$

Compare: $n^{\log_b a}$ and f(n)

$n^{\log_4 3} <= nlogn$ f(n) is asymptotically and polynomially larger

Thus case 3, but we have to check the requlrity condition!

The following should be true:

af(n/b) < = cf(n) where c<1

a(n/b) log (n/b) < = cf(n)

=> 3(n/4) log(n/4) <=c n log n

3/4nlog(n/4) <= c .n log n,

this is true for c=3/4 Hence.T(n) = $\theta\ (nlog(n))$

Case Study: Analyzing Algorithms

- **Computing the prefix averages of a sequence of numbers**.

  The ***i***-th prefix average of an array ***X*** is average of the first $(i + 1)$ elements of ***X***:

  $A[i] = (X[0] + X[1] + \ldots + X[i])/(i+1)$
- **Applications**
  - **Runtime analysis example:**

    Two algorithms for prefix averages

Case Study: Analyzing Algorithms

- The following algorithm computes prefix averages in quadratic time by applying the definition

```
Algorithm prefixAverages1(X, n)
    Input array X of n integers
    Output array A of prefix averages of X          #operations
    A ← new array of n integers                     n
    for i ← 0 to n − 1 do                           n
        s ← X[0]                                    n
        for j ← 1 to i do                           1 + 2 + …+ (n − 1)
            s ← s + X[j]                            1 + 2 + …+ (n − 1)
        A[i] ← s / (i + 1)                          n
    return A                                        1
```

Algorithm ***prefixAverages1*** runs in $O(n^2)$ time

Case Study: Analyzing Algorithms

- The following algorithm computes prefix averages in linear time by keeping a running sum

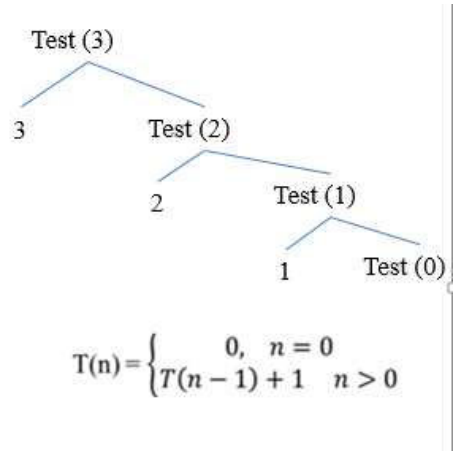
```
Algorithm prefixAverages2(X, n)
    Input array X of n integers
    Output array A of prefix averages of X          #operations
    A ← new array of n integers                     n
    s ← 0                                           1
    for i ← 0 to n − 1 do                           n
        s ← s + X[i]                                n
        A[i] ← s / (i + 1)                          n
    return A                                        1
```

Algorithm ***prefixAverages2*** runs in $O(n)$ time

Homework Problems

Solving recurrences : Iterative Method

```
void test(int n)
{
        if(n>0)
        {
        printf("%d",n);
        test(n-1);
        }
}
```



$$T(n) = \begin{cases} 0, & n = 0 \\ T(n-1) + 1 & n > 0 \end{cases}$$

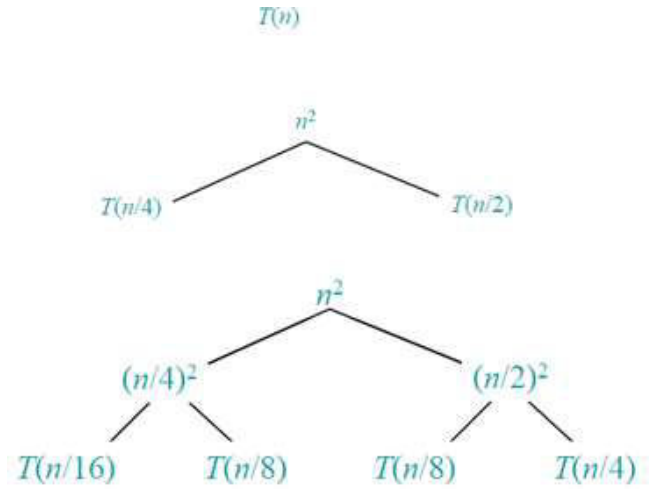
Solving recurrences : Iterative Method

```
Void Test (int n)
{
        if(n>1)
        {
                for (i=1;i<n;i++)
                {
                        stmt;
                }
                Test(n/2);
                Test(n/2);
        }
}
```

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

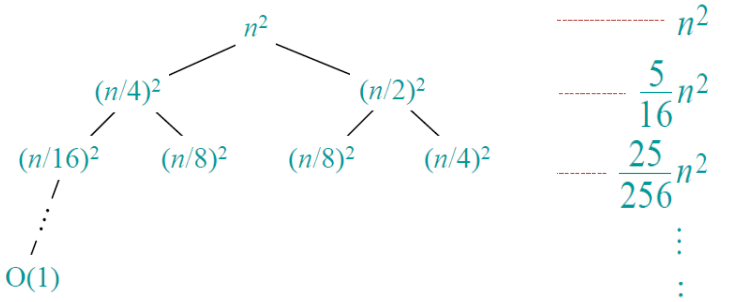
Solving Recurrences-Recursion Tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



Solving Recurrences-Recursion Tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



$$\text{Total} = n^2\left(1 + \frac{5}{16} + \left(\frac{5}{16}\right)^2 + \left(\frac{5}{16}\right)^3 + \cdots\right)$$
$$= O(n^2) \quad \textit{geometric series}$$

Master method Problems

- T(n)=9T(n/3)+n
- T(n)=T(2n/3)+1
- T(n)=3T(n/4)+nlogn
- T(n)=2T (n/2)+ n lg n
- T (n)=8T(n/2)+$\Theta(n^2)$

THANK YOU!

BITS Pilani
Hyderabad Campus

BITS Pilani
Hyderabad Campus

S2 Characterizing Time Complexity, Asymptotic Notation, Recurrence Relation, Master Theorem

Content of S2

1. Characterizing Time Complexity
    1. Use of Asymptotic Notation
    2. Big-Oh, Big-Omega, Theta Notations
2. Analyzing Recursive Algorithm
    1. Recurrence Relation
    2. Runtime of Recursive Algorithm
    3. Master Theorem

Analyzing Algorithm

→Used to mean the prediction of **resource** consumption

→But, **what** is the **resource**?

Analyzing Algorithm

→Used to mean the prediction of **resource** consumption

→But, **what** is the **resource**?

Primarily i) **memory**, ii) communication **bandwidth**, iii) computer **hardware**

But, most often we are interested in **computational time**

Which computer should be taken as a base case or standard?

- **Random Access Machine (RAM)** model of a computer

Random Access Machine Model

Instructions in RAM that takes **one unit of time**

1) **Arithmetic:** Add, Sub, Mul, Div, Rem, Floor, Ceil
2) **Data movement:** Load, Store, Copy
3) **Control:** Subroutine call, Return, Conditional and Unconditional Branch

Data Types in RAM (**fixed size**, like 8 bit or 16 bit or 32 bit)

1) Integer
2) Float

RAM model: What is not an instruction?

1) “Sort” – even if, in some computer, sort can be done in one instruction
2) “exponentiation” – x^y

→ there may be many algorithms to compute x^y, but it is not a single instruction if y is a variable or a large integer

→ But, x^k is a single instruction, where **k is a constant and very small**

RAM model: memory hierarchy

We do not consider any complex memory hierarchy, like having cache or virtual memory.

RAM model: memory hierarchy

We do not consider any complex memory hierarchy, like having cache or virtual memory.

Simplicity of RAM model

→ Though simple, but an excellent predictor of performance on actual computer

→Though simple, exact prediction can still be challenging

→Often, it would require tools like combinatorics, probability theory, algebraic dexterity and the ability to identify the most significant terms in a formula

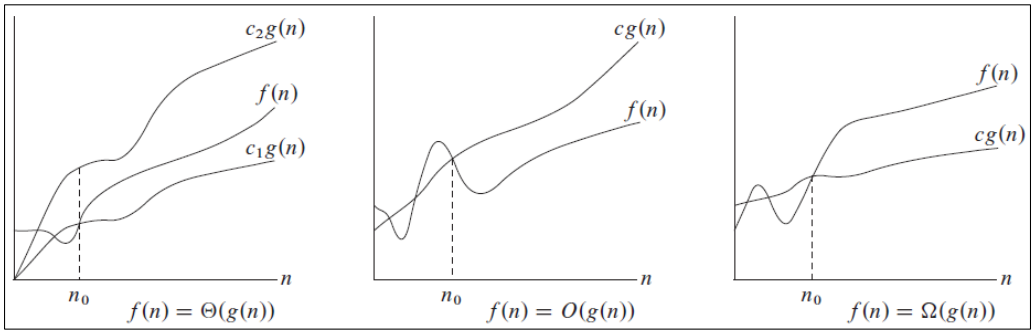
Content of S2

1. **Characterizing Time Complexity**
   1. **Use of Asymptotic Notation**
   2. Big-Oh, Big-Omega, Theta Notations
2. Analyzing Recursive Algorithm
   1. Recurrence Relation
   2. Runtime of Recursive Algorithm
   3. Master Theorem

Characterizing Time Complexity

Big-Oh Notation, Omega and Theta Notations:

- Asymptotic notation primarily describes the running times of algorithms, i.e., time complexity



Content of S2

1. **Characterizing Time Complexity**
   1. Use of Asymptotic Notation
   2. **Big-Oh, Big-Omega, Theta Notations**
2. Analyzing Recursive Algorithm
   1. Recurrence Relation
   2. Runtime of Recursive Algorithm
   3. Master Theorem

Characterizing Time Complexity

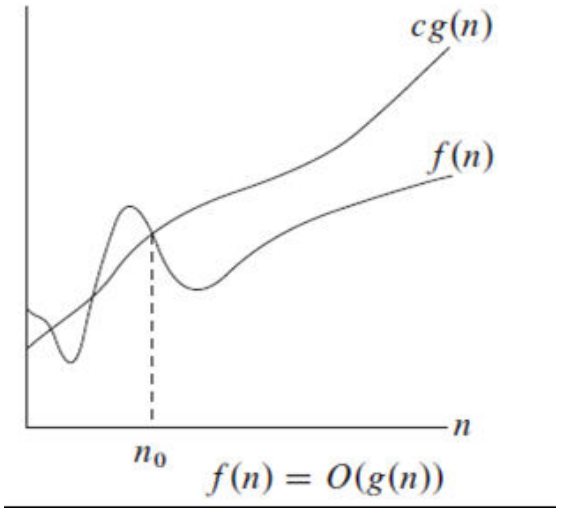
Big-Oh Notation: f(n) = O(g(n)).

- g(n) is an asymptotically upper bound for f(n).

$O(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0\}$.

- f(n) = Θ(g(n)) implies that f(n) = O(g(n)),
  i.e., Θ(g(n)) ϵ O(g(n))

Graphical representation of Big-O



Example: Time Complexity Big-O

Ex-1 f(n) =2n+2
2n+2 ≤ 10n, where n≥1
Here, c = 10, **g(n) = n**
f(n) = O(g(n)) = O(n).

Ex-2 f(n) =2n+2
2n+2 ≤ $10n^2$, where n≥1
Here, c = 10, **g(n) = n^2**
f(n) = O(g(n)) = $O(n^2)$.

Ex-3 f(n) =2n+2
2n+2 ≤ $10n^3$, where n≥1
Here, c = 10, g(n) = n^3
f(n) = O(g(n)) = $O(n^3)$.

Ex-4 f(n) =$2n^2$+5
$2n^2+5 \leq 2n^2+5n^2 = 7n^2$, where n≥1
Here, c = 7, g(n) = n^2
f(n) = O(g(n)) = $O(n^2)$.

$O(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0\}$.

Example: Time Complexity Big-O

Ex-5 f(n) =7n-2
Here, c = 7, n >=1
→ 7n - 2 ≤ cn, **g(n) = n**
f(n) = O(g(n)) = O(n).

Ex-6 f(n) = $20n^3$ + 10nlogn + 5
Here, c = 35, **g(n) = n^3**
f(n) = O(g(n)) = $O(n^3)$.

Ex-7 f(n) = 3logn + loglogn
Here, c = 4, **g(n) = logn**
f(n) = O(g(n)) = O(logn).

Ex-8 f(n) = 2^{100}
Here, c = 2^{100}, **g(n) = 1**
f(n) = O(g(n)) = O(1).

Ex-9 f(n) = 5/n
Here, c = **5**, **g(n) =** 1/n
f(n) = O(g(n)) = O(1/n).

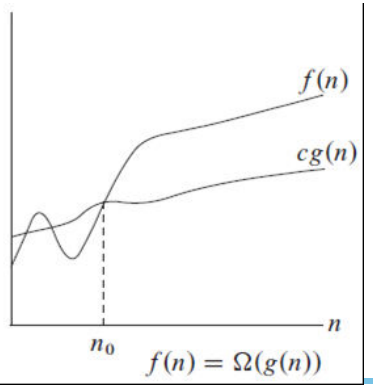
$O(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0\}$.

Time Complexity: Big-Omega

Omega Notation: f(n) = Ω(g(n)).

- g(n) is an asymptotically lower bound for f(n).

$\Omega(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0\}$.



Example: Omega Notation

Ex-1 f(n) =2n+2
2n+2 ≥ 2n, where n≥1
Here, c = 2, **g(n) = n**
f(n) = Ω(g(n)) = Ω(n)

Ex-2 f(n) =2n+2
2n+2 ≥ $\sqrt{n}$, where n≥1
Here, c = 1, **g(n) = $\sqrt{n}$**
f(n) = Ω(g(n)) = Ω($\sqrt{n}$)

Ex-3 f(n) =2n+2
2n+2 ≥ log n, where n≥1
Here, c = 1, **g(n) = log n**
f(n) = Ω(g(n)) = Ω(log n)

Ex-4 f(n) =$2n^2$+5
$2n^2+5 \geq 2n^2$, where n≥1
Here, c = 2, **g(n) = n^2**
f(n) = Ω(g(n)) = $\Omega(n^2)$.

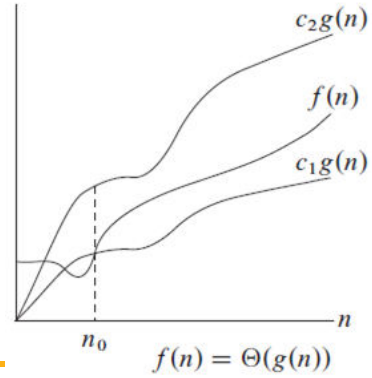
$\Omega(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0\}$.

Characterizing Run Time

Theta Notation: f(n) = Θ(g(n)).

- g(n) is an asymptotically tight bound for f(n).

$\Theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2, and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0\}$.[1]



Example: Theta Notation

Ex-1 f(n) = $\frac{n^2}{2} - \frac{n}{2}$

$\frac{n^2}{4} \le \frac{n^2}{2} - \frac{n}{2} \le \frac{n^2}{2}$, where **n≥2**

$c_1 = \frac{1}{4}$, $c_2 = \frac{1}{2}$, **g(n) = n^2**

f(n) = Θ(g(n)) = Θ(n^2).

Ex-2 f(n) =$6n^3$ **{≠ Θ(n^2), why?}**

$c_1n^2 \le 6n^3 \le c_2n^2$, where n≥1

There exists no c_2 that implies $6n^3 \le c_2n^2$

$\Theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2, and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0\}$.[1]

Time Complexity: Little-Oh, Little-omega

o-notation:

$o(g(n)) = \{f(n)$: for any positive constant $c > 0$, there exists a constant $n_0 > 0$ such that $0 \le f(n) < cg(n)$ for all $n \ge n_0\}$.

Time Complexity: Little-Oh, Little-omega

ω-notation:

$\omega(g(n)) = \{f(n)$: for any positive constant $c > 0$, there exists a constant $n_0 > 0$ such that $0 \le cg(n) < f(n)$ for all $n \ge n_0\}$.

Notation Summary

Because these properties hold for asymptotic notations, we can draw an analogy between the asymptotic comparison of two functions f and g and the comparison of two real numbers a and b:

$f(n) = O(g(n))$ is like $a \leq b$,
$f(n) = \Omega(g(n))$ is like $a \geq b$,
$f(n) = \Theta(g(n))$ is like $a = b$,
$f(n) = o(g(n))$ is like $a < b$,
$f(n) = \omega(g(n))$ is like $a > b$.

Properties of Time Complexity

- Comparison

Transitivity:

$f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$ imply $f(n) = \Theta(h(n))$,
$f(n) = O(g(n))$ and $g(n) = O(h(n))$ imply $f(n) = O(h(n))$,
$f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$ imply $f(n) = \Omega(h(n))$,
$f(n) = o(g(n))$ and $g(n) = o(h(n))$ imply $f(n) = o(h(n))$,
$f(n) = \omega(g(n))$ and $g(n) = \omega(h(n))$ imply $f(n) = \omega(h(n))$.

Reflexivity:

$f(n) = \Theta(f(n))$,
$f(n) = O(f(n))$,
$f(n) = \Omega(f(n))$.

Summary of Properties

- Comparison

Symmetry:

$f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$.

Transpose symmetry:

$f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$,
$f(n) = o(g(n))$ if and only if $g(n) = \omega(f(n))$.

Content of S2

1. Characterizing Time Complexity
   1. Use of Asymptotic Notation
   2. Big-Oh, Big-Omega, Theta Notations
2. **Analyzing Recursive Algorithm**
   1. **Runtime of Recursive Algorithm**
   2. Recurrence Relation
   3. Master Theorem

Analyzing Recursive Algorithms

Algorithm recursiveMax(A, n):
Input: An array A storing $n \geq 1$ integers.
Output: The maximum element in A.

if $n = 1$ **then**
  return $A[0]$
return max{recursiveMax$(A, n-1), A[n-1]$}

Analyzing Recursive Algorithms

Algorithm recursiveMax(A, n):
Input: An array A storing $n \geq 1$ integers.
Output: The maximum element in A.

if $n = 1$ **then**
  return $A[0]$
return max{recursiveMax$(A, n-1), A[n-1]$}

$$T(n) = \begin{cases} 2, if\ n = 1 \\ T(n-1) + 4, otherwise \end{cases}$$

Analyzing Recursive Algorithms

Algorithm recursiveMax(A, n):
Input: An array A storing $n \geq 1$ integers.
Output: The maximum element in A.

1 **if** $n = 1$ **then**
1 +1 **return** $A[0]$
return max{recursiveMax$(A, n-1), A[n-1]$}
1 + 1 + T(n-1) + 1 + 1 + 1

$$T(n) = \begin{cases} 3, if\ n = 1 \\ T(n-1) + 6, otherwise \end{cases}$$

Content of S2

1. Characterizing Time Complexity
   1. Use of Asymptotic Notation
   2. Big-Oh, Big-Omega, Theta Notations
2. **Analyzing Recursive Algorithm**
   1. Runtime of Recursive Algorithm
   2. **Recurrence Relation**
   3. Master Theorem

Recurrence Relation

Def^n: A recurrence relation is an equation that defines a sequence based on a rule that gives the next term as a function of the previous term(s).

Mathematically, $x_{n+1} = f(x_n)$: a simple recurrence relation, also called as first order recurrence relation.

Recurrence Relation

Def^n: A recurrence relation is an equation that defines a sequence based on a rule that gives the next term as a function of the previous term(s).

Mathematically, $x_{n+1} = f(x_n)$: a simple recurrence relation, also called as first order recurrence relation.

Example of first order recurrence relation:

1) $x_{n+1}=2-x_{n/2}$

Recurrence Relation

Def^n: A recurrence relation is an equation that defines a sequence based on a rule that gives the next term as a function of the previous term(s).

Mathematically, $x_{n+1} = f(x_n)$: a simple recurrence relation, also called as first order recurrence relation.

Example of first order recurrence relation:

1) $x_{n+1}=2-x_{n/2}$

A second order recurrence relation depends just on x_n and x_{n-1} and is of the form $x_{n+1}=f(x_n,x_{n-1})$

Example: $x_{n+1}=x_n+x_{n-1}$

Content of S2

1. Characterizing Time Complexity
   1. Use of Asymptotic Notation
   2. Big-Oh, Big-Omega, Theta Notations
2. **Analyzing Recursive Algorithm**
   1. Runtime of Recursive Algorithm
   2. Recurrence Relation
   3. **Master Theorem**

Analyzing Recursive Algorithms

Solving recurrence equations

1. Master Theorem for Dividing Functions

$$T(n) = aT(\frac{n}{b}) + g(n)$$

where g(n) is $O(n^k \log^p n)$, where p and k are integers.

a) $a < b^k$: if $p < 0$, then $T(n) = O(n^k)$

Analyzing Recursive Algorithms

Solving recurrence equations

1. Master Theorem for Dividing Functions

$$T(n) = aT(\frac{n}{b}) + g(n)$$

where g(n) is $O(n^k \log^p n)$, where p and k are integers.

a) $a < b^k$: if $p < 0$, then $T(n) = O(n^k)$

if $p \geq 0$, then $T(n) = O(n^k \log^p n)$

Analyzing Recursive Algorithms

Solving recurrence equations

1. Master Theorem for Dividing Functions

$$T(n) = aT(\frac{n}{b}) + g(n)$$

where g(n) is $O(n^k \log^p n)$

a) $a < b^k$: if $p < 0$, then $T(n) = O(n^k)$

if $p \geq 0$, then $T(n) = O(n^k \log^p n)$

b) $a = b^k$: if $p > -1$, then $T(n) = O(n^k \log^{p+1} n)$

if $p = -1$, then $T(n) = O(n^k \log \log n)$

if $p < -1$, then $T(n) = O(n^k)$

Analyzing Recursive Algorithms

Solving recurrence equations

1. Master Theorem for Dividing Functions

$$T(n) = aT(\frac{n}{b}) + g(n)$$

where g(n) is $O(n^k \log^p n)$

a) $a < b^k$: if $p < 0$, then $T(n) = O(n^k)$

if $p \geq 0$, then $T(n) = O(n^k \log^p n)$

b) $a = b^k$: if $p > -1$, then $T(n) = O(n^k \log^{p+1} n)$

if $p = -1$, then $T(n) = O(n^k \log \log n)$

if $p < -1$, then $T(n) = O(n^k)$

c) $a > b^k$: $T(n) = O(n^{\log_b a})$

Solution using Master Theorem

g(n) is $O(n^k \log^p n)$

Ex-1 $T(n) = 4T(\frac{n}{2})+n$,
a = 4, b = 2, k = 1, p = 0.
$a = 4, b^k = 2 \rightarrow a > b^k$
$T(n) = O(n^{\log_2 4}) = O(n^2)$

Ex-2 $T(n) = 8T(\frac{n}{2})+n^2$,
a = 8, b = 2, k = 2, p = 0.
$a = 8, b^k = 4 \rightarrow a > b^k$
$T(n) = O(n^{\log_2 8}) = O(n^3)$

Ex-3 $T(n) = 8T(\frac{n}{2})+n \log n$,
a = 8, b = 2, k = 1, p = 1.
$a = 8, b^k = 2 \rightarrow a > b^k$
$T(n) = O(n^{\log_2 8}) = O(n^3)$

Solution using Master Theorem

Ex-4 $T(n) = 2T(\frac{n}{2})+n$,
a = 2, b = 2, k = 1, p = 0.
$a = 2, b^k = 2 \rightarrow a = b^k$
$T(n) = O(n^k \log^{p+1} n)$
$= O(n \log n)$

Ex-5 $T(n) = 4T(\frac{n}{2})+n^2$,
a = 4, b = 2, k = 2, p = 0.
$a = 4, b^k = 4 \rightarrow a = b^k$
$T(n) = O(n^k \log^{p+1} n)$
$= O(n^2 \log n)$

Ex-6 $T(n) = 4T(\frac{n}{2})+n^2 \log n$,
a = 4, b = 2, k = 2, p = 1.
$a = 4, b^k = 4 \rightarrow a = b^k$
$T(n) = O(n^k \log^{p+1} n)$
$= O(n^2 \log^2 n)$

Solution using Master Theorem

Ex-7 $T(n) = 2T(\frac{n}{2})+\frac{n}{\log n}$,
a = 2, b = 2, k = 1, p = -1.
$a = 2, b^k = 2 \rightarrow a = b^k$
$T(n) = O(n^k \log \log n)$
$= O(n \log \log n)$

Ex-8 $T(n) = T(\frac{n}{2})+n^2$,
a = 1, b = 2, k = 2, p = 0.
$a = 1, b^k = 4 \rightarrow a < b^k$
$T(n) = O(n^k \log^p n)$
$= O(n^2)$

Ex-9 $T(n) = 2T(\frac{n}{2})+n^2 \log^2 n$,
a = 2, b = 2, k = 2, p = 2.
$a = 2, b^k = 4 \rightarrow a < b^k$
$T(n) = O(n^k \log^p n)$
$= O(n^2 \log^2 n)$

Master Theorem for Decreasing Functions

$$T(n) = aT(n-b) + g(n)$$

where g(n) is $O(n^k)$

a) $a < 1 : T(n) = O(n^k)$
b) $a = 1 : T(n) = O(n^{k+1})$
c) $a > 1 : T(n) = O(n^k a^{n/b})$

Solution using Master Theorem

Ex-1 T(n) = T(n-1)+1,
a = 1, b = 1, k = 0.

T(n) = $O(n^{k+1})$ = O(n)

Ex-3 T(n) = 2T(n-1)+1,
a = 2, b = 1, k = 0.

T(n) = $O(n^k a^{n/b})$
= $O(2^n)$

Ex-2 T(n) = T(n-1)+n,
a = 1, b = 1, k = 1.

T(n) = $O(n^{k+1})$ = $O(n^2)$

Ex-4 T(n) = 2T(n-1)+n,
a = 2, b = 1, k = 1.

T(n) = $O(n^k a^{n/b})$
= $O(n2^n)$

Correctness of Algorithms

- An algorithm is said to be correct
  - if, for every input instance, it halts with the correct output.
- We say that a correct algorithm
  - solves the given computational problem.
- An incorrect algorithm
  - might not halt at all on some input instances, or
  - it might halt with an incorrect answer.

Some Mathematics

Ordering Functions by Their Growth Rates

n	$\log n$	$\sqrt{n}$	n	$n\log n$	n^2	n^3	2^n
2	1	1.4	2	2	4	8	4
4	2	2	4	8	16	64	16
8	3	2.8	8	24	64	512	256
16	4	4	16	64	256	4,096	65,536
32	5	5.7	32	160	1,024	32,768	4,294,967,296
64	6	8	64	384	4,096	262,144	1.84×10^{19}
128	7	11	128	896	16,384	2,097,152	3.40×10^{38}
256	8	16	256	2,048	65,536	16,777,216	1.15×10^{77}
512	9	23	512	4,608	262,144	134,217,728	1.34×10^{154}
1,024	10	32	1,024	10,240	1,048,576	1,073,741,824	1.79×10^{308}

$1 < \log n < \text{sqrt}(n) < n < n \log n < n^2 < n^3 < \ldots < 2^n < 3^n < n^n$

Some Mathematics

- $\sum_{i=0}^{n} a^i = 1 + a + \cdots + a^n = \frac{1-a^{n+1}}{1-a}$
- $log_b a = c \; if \; a = b^c$
- $log_b ac = log_b a + log_b c$
- $log_b (a/c) = log_b a - log_b c$
- $log_b a^c = c log_b a$
- $log_b a = log_c a / log_c b$
- $b^{log_c a} = a^{log_c b}$

Case Studies: Analyzing Algorithms

innovate achieve lead

```
Ex-1
#include <stdio.h>
void main(){
    int n=10;
    int a[n];
    a[3]=5;
    printf("%d",a[3]);
}
```

T(n) = 1+(1+1) + (1+1) → T(n) = O(1)

```
Ex-2
#include <stdio.h>
void main(){
    int n; scanf(“%d”,&n);
    int a[n];
    for(int i=0;i<n;i++)
        scanf("%d",&a[i]);
    for(int i=0;i<n;i++)
        printf("%d",a[i]);
}
```

T(n) = 2+(1+ (n+1) + 2(n)) + 2n + (1+ (n+1) + 2(n)) + 2n = 10n + 6
→ T(n) = O(n)

Case Studies: Analyzing Algorithms

innovate achieve lead

```
Ex-3
#include <stdio.h>
void main(){
 int n; scanf(“%d”,&n);
    int a[n];
    for(int i=0;i<n;i++)
        scanf("%d",&a[i]);
    for(int i=0;i<n;i++)
        for(int j=0;j<n;j++)
            printf("%d",a[i]);
}
```

T(n) = 2+(1+ (n+1) + 2(n)) + 2n + (1+ (n+1) + 2(n)) + n (1+ (n+1) + 2(n))
= $3n^2 + 10n + 6$
→ T(n) = $O(n^2)$

```
Ex-4
#include <stdio.h>
void main(){
 int n; scanf(“%d”,&n);
    int a[n];
    for(int i=0;i<n;i++)
        scanf("%d",&a[i]);
    for(int i=0;i<n;i++)
        for(int j=0;j<n/2;j++)
            printf("%d",a[i]);
}
```

T(n) = 2+(1+ (n+1) + 2(n)) + 2n + (1+ (n+1) + 2(n)) + n (1+ (n+1)/2 + 2(n/2))
→ T(n) = $O(n^2)$

Case Studies: Analyzing Algorithms

innovate achieve lead

```
Ex-5
int findMinimum(int array[]) {
    int min = array[0];
    for(int i = 1; i < n; i++){
        if (array[i] < min) {
            min = array[i];
        }
    }
    return min;
}
```

T(n) = O(n)

Case Studies: Analyzing Algorithms

innovate achieve lead

```
Ex-6
void fun(int n){
        if(n<=0)
                return;
        printf("%d",n);
        fun(n-1);
}
```

T(n) = T(n-1) + 2 → T(n) = O(n)
Master Theorem for Decreasing Functions

```
Ex-7
void fun(int n){
        if(n<=0)
                return;
        printf("%d",n);
        fun(n/2);
}
```

T(n) = T(n/2) + 2 → T(n) = O(log n)
Master Theorem for Dividing Functions

Case Studies: Analyzing Algorithms

```
Ex-8
void fun(int n){
    if(n<=0)___1
        return;
    for(int i=0;i<k';i++) ___(k'+1)
        fun(n-1); ___k'*T(n-1)
}
```

(T(n) = 1 + (k'+1) + (k'(T(n-1)) = k'*T(n-1) + (k' + 2) ➔ T(n) depends on value of k' (Master Theorem for Decreasing Functions))*

```
Ex-9
void fun(int n){
    if(n>1){ ___1
        for(int i=0;i<n;i++) ___(n+1)
            printf("%d",i); ___n
        fun(n/2); ___T(n/2)
        fun(n/2); ___T(n/2)
        }
}
```

T(n) = 1 + (n+1) + n + 2T(n/2) = 2T(n/2) + (2n + 2)
a = 2, b= 2, k = 1, p = 0. O(n log n) as per Master Theorem for Dividing Functions

Any Question!!

References

1. Algorithms Design: Foundations, Analysis and Internet Examples Michael T. Goodrich, Roberto Tamassia, 2006, Wiley (Students Edition)
2. Data Structures, Algorithms and Applications in C++, Sartaj Sahni, Second Ed, 2005, Universities Press
3. Introduction to Algorithms, TH Cormen, CE Leiserson, RL Rivest, C Stein, Third Ed, 2009, PHI

T(n) = 2T(n/2) + 2

This will be third case of master's theorem or the second case?

Calculation:

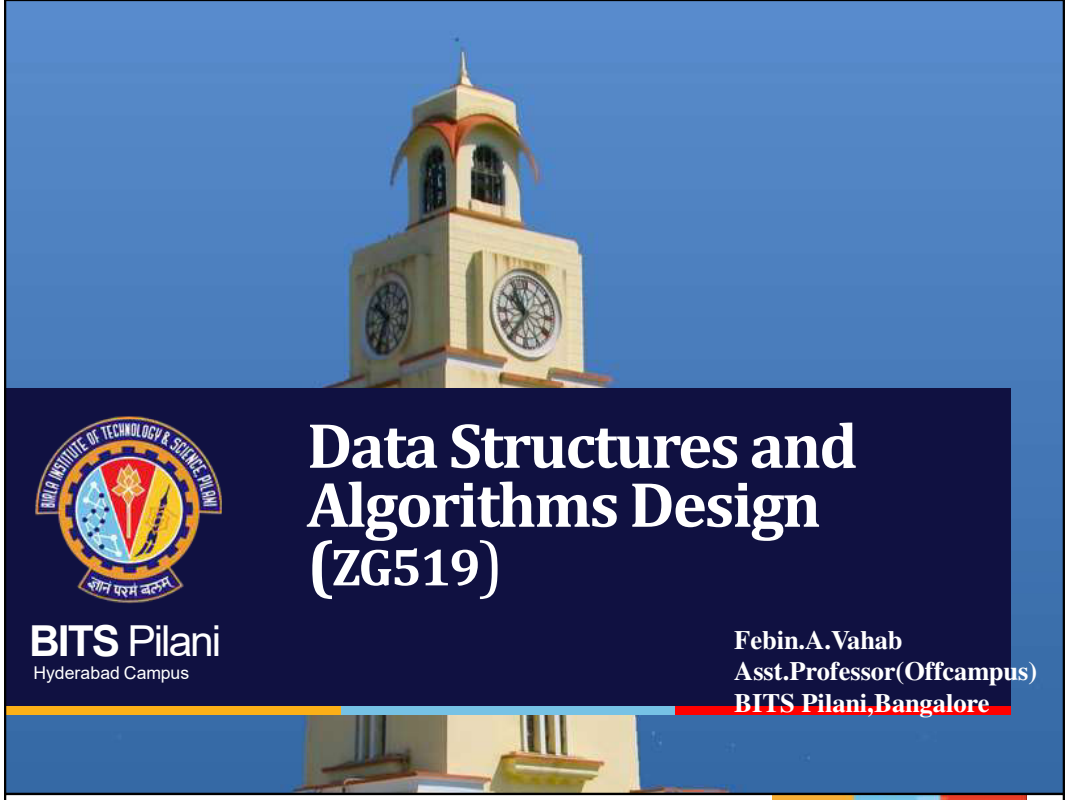
f(n) = 2

Number of leaves = n ^ logb(a) = n ^ log(1) = n ^ 0 = 1

f(n) > n^(logb(a)): Third case. Right?

The answer will be Theta(2)?

has context menu



Data Structures and Algorithms Design (ZG519)

BITS Pilani
Hyderabad Campus

Febin.A.Vahab
Asst.Professor(Offcampus)
BITS Pilani,Bangalore

SESSION 3 -PLAN

Online Sessions(#)	List of Topic Title	Text/Ref Book/external resource
3	Stack ADT and Implementation. Queue ADT and Implementation, Applications	T1: 2.1

Abstract Data Type

- In computer science, an **abstract data type** (**ADT**) is a type (or class) for objects whose behavior is defined by a set of value and a set of operations.

Abstract Data Type

- 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

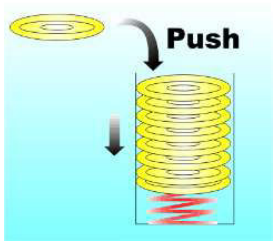
Abstract Data Type

- 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

- Stack "S" is a abstract data type supporting following methods:
  - ***Push(o)***- insert object o at the top of the stack
  - ***Pop()-*** remove from the stack and return the top object on the stack. Error occurs for empty stack.
  - Supporting methods
    - ***Size()-*** return the number of objects in the stack
    - ***isEmpty()-*** return Boolean indicating if stack is empty
    - ***Top()-*** return value of top object on the stack. Error occurs for empty stack.

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 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]
```

```
Algorithm size()
return t+1

Algorithm isEmpty()
if t<0 return

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


Stacks: An Array Implementation

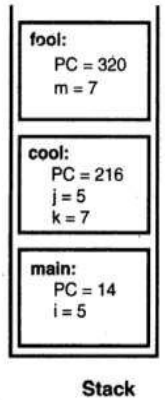
- The array implementation is simple and efficient (**methods performed in O(1))/constant time.**

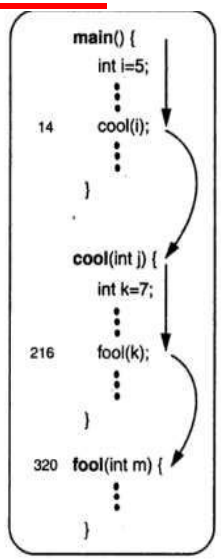
Disadvantage

- 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.

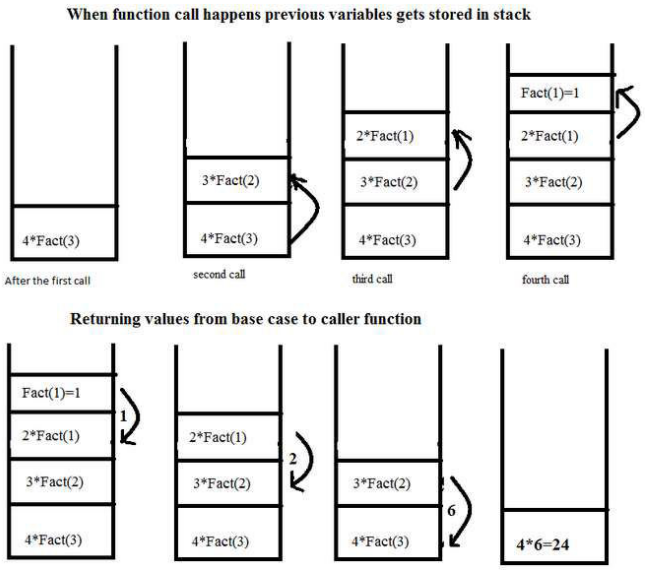
Stacks: Applications

- Procedure calls
- Recursion





Stacks: Applications



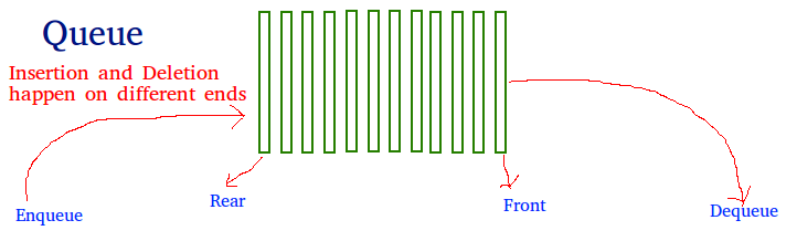
Stacks: HW-Stock Span problem

- Given a list of prices of a single stock for N number of days,find the stock span for each day.
- ***Stock span is the number of consecutive days prior to the current day when the price of the stock was less than or equal to the prices at the current day***

Day	Price
1	200
2	120
3	140
4	160
5	130

Queue ADT

- 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)



Queue ADT

- The queue ADT supports the following two fundamental methods:
  - ***enqueue(o):*** Insert object o at the rear of the queue.
  - ***dequeue():*** Remove and return from the queue the object at the front;an error occurs if the queue is empty.

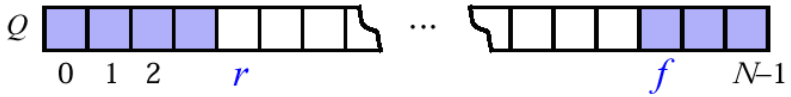
  Additionally, the queue ADT includes the following supporting methods:
  - ***size():*** Return the number of objects in the queue.
  - ***isEmpty():*** Return a Boolean value indicating whether queue is empty.
  - ***front():*** Return, but do not remove, the front object in the queue;an error occurs if the queue is empty

Queues: An Array Implementation

- A maximum size *N* is specified.
- The queue consists of an N-element array *Q* and two integer variables:
  - *f*, is an index to the cell of Q storing the first element of the queue which is the next candidate to be removed by a dequeue operation, unless the queue is empty (in which case f = r)
  - *r* , is an index to the next available array cell in Q.
  - Initially, f=r=0 and the queue is empty if f=r

Queues: An Array Implementation

- The "normal" configuration with f<=r

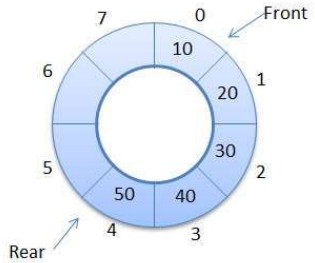


- The "wrapped around" configuration with r < f.

The cells storing queue elements are highlighted.

Queues: An Array Implementation

- Circular Array
  - r=(r+1)mod N
  - f=(f+1)mod N
- When array is full f=r (Same as array empty condition)



Queues: An Array Implementation

- Queue is empty: $f = r$
- When r reaches and overlaps with f, the queue is full: $r = f$

- To distinguish between empty and full states, we impose a constraint: ***Q* can hold at most *N* – 1 objects** (one cell is wasted). So r never overlaps with f, except when the queue is empty.

Queues: An Array Implementation

- Queue Operations

```
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
```

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

```
Algorithm isEmpty()
if(f=r)return
```

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

- Each method runs in O(1) time.

Queues: Application

- Multiprogramming

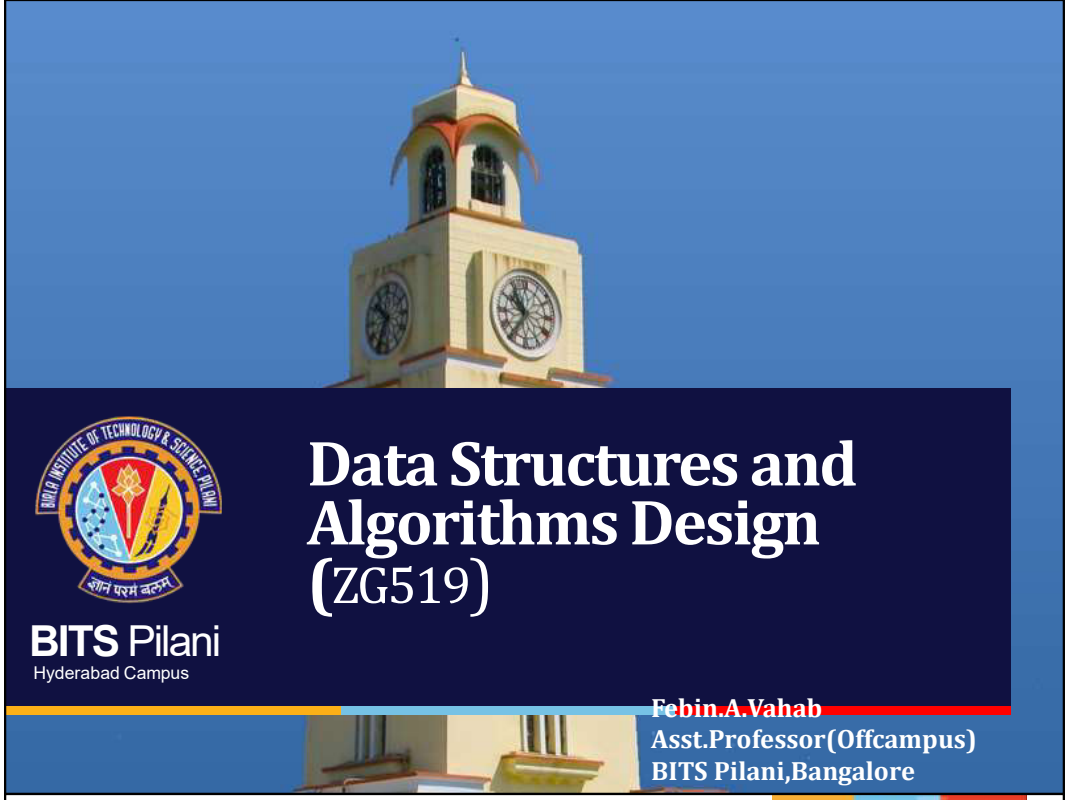
Queues: HW

- Write an Algorithm to implement Queue using Stack(s).Discuss different approaches.

THANK YOU!

BITS Pilani
Hyderabad Campus

innovate achieve lead



Data Structures and Algorithms Design (ZG519)

BITS Pilani
Hyderabad Campus

Febin.A.Vahab
Asst.Professor(Offcampus)
BITS Pilani,Bangalore

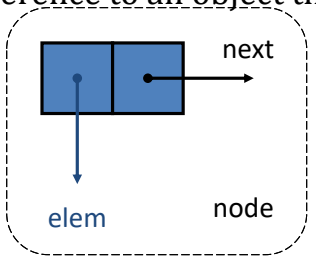
innovate achieve lead

SESSION 4 -PLAN

Sessions(#)	List of Topic Title	Text/Ref Book/external resource
4	Lists- Notion of position in lists, List ADT and Implementation. Sets- Set ADT and Implementation Trees: Terms and Definition, Tree ADT, Applications Binary Trees : Terms and Definition, Properties, Properties, Representations (Vector Based and Linked), Binary Tree traversal (In Order, Pre Order, Post Order), Applications	T1: 2.2, 4.2

SINGLY LINKED LIST

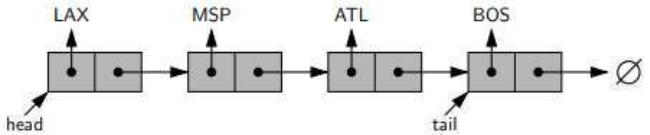
- A singly linked list is a concrete data structure consisting of a sequence of nodes
- Each node stores reference to an object that is a reference to an
  - element
  - link to the next node



- The first and last node of a linked list usually are called the ***head*** and ***tail*** of the list, respectively

Singly Linked List

- Moving from one node to another by following a next reference is known as ***link hopping*** or ***pointer hopping-Traversing the list***
- The order of elements is determined by the chain of ***next*** links going from each node to its successor in the list
- We do not keep track of any index numbers for the nodes in a linked list. So we cannot tell just by examining a node if it is the second, fifth, or twentieth node in the list

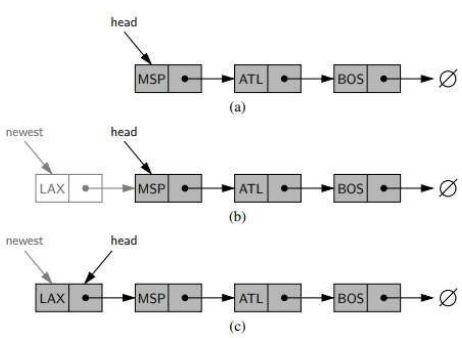


Example of a singly linked list whose elements are strings indicating airport codes

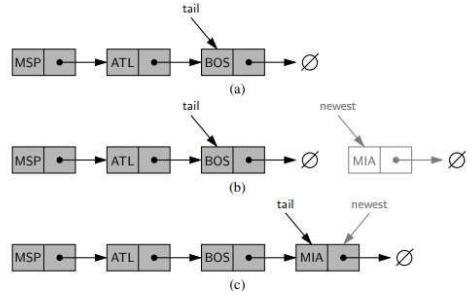
Singly Linked List Implementation

- To implement a singly linked list, we define a Node class

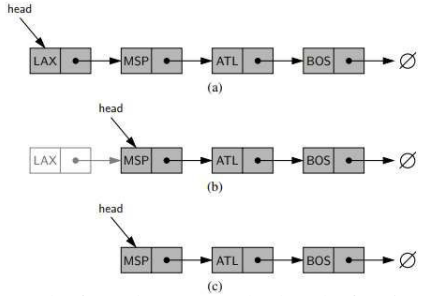
```
class _Node:
  """Lightweight, nonpublic class for storing a singly linked node."""
  __slots__ = '_element', '_next'          # streamline memory usage

  def __init__(self, element, next):       # initialize node's fields
    self._element = element                # reference to user's element
    self._next = next                      # reference to next node
```

Insertion of an element at the head of a singly linked list



Insertion at the tail of a singly linked list



Removal of an element at the head of a singly linked list

Singly Linked List Operations-Node Based

- *Not easy to delete the tail node /Insertbefore operations*
- *Start from the head of the list and search all the way through the list.*
- *Such link hopping operations could take a long time.*

SLL-Node Based-insertBefore

```
insertBefore(n, o):
    if n == first then
        insertFirst(o)
    else
        m = first
        while m.next != n do
            m = m.next
        done
        insertAfter(m, o)
```

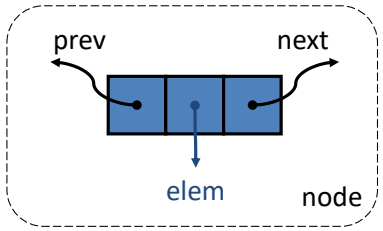
SLL-Node Based- Performance

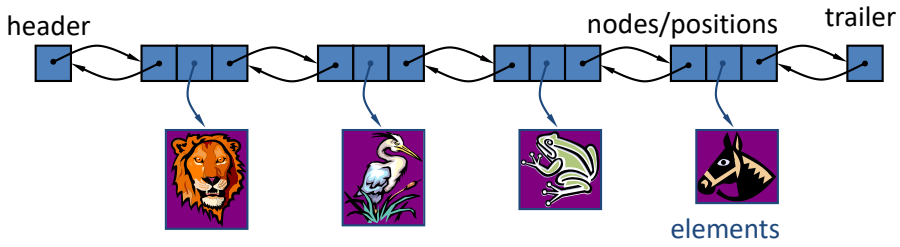
Operation	Worst case Complexity
size, isEmpty	$O(1)$ [1]
first, last, after	$O(1)$ [2]
before	$O(n)$
replaceElement, swapElements	$O(1)$
insertFirst, insertLast	$O(1)$ [2]
insertAfter	$O(1)$
insertBefore	$O(n)$
remove	$O(n)$ [3]

[1] size needs $O(n)$ if we do not store the size on a variable.
[2] last and insertLast need $O(n)$ if we have no variable *last*.
[3] remove(n) runs in best case in $O(1)$ if $n == first$.

DOUBLY LINKED LIST

- Nodes store:
  - element
  - link to the previous node -*prev*
  - link to the next node-*next*
- Special trailer and header nodes



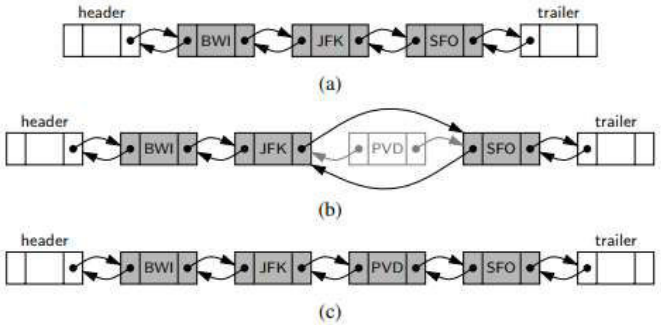


Doubly Linked List Implementation

- To implement a Doubly linked list, we define a Node class

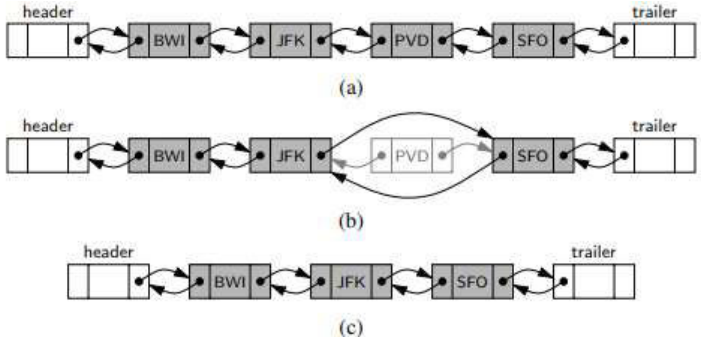
```
class _Node:
  """Lightweight, nonpublic class for storing a doubly linked node."""
  __slots__ = '_element', '_prev', '_next'  # streamline memory

  def __init__(self, element, prev, next):          # initialize node's fields
    self._element = element                         # user's element
    self._prev = prev                               # previous node reference
    self._next = next                               # next node reference
```


Doubly Linked List Implementation



Adding an element to a doubly linked list with header and trailer sentinels: (a) before the operation; (b) after creating the new node; (c) after linking the neighbors to the new node

Doubly Linked List Implementation



Removing the element PVD from a doubly linked list: (a) before the removal; (b) after linking out the old node; (c) after the removal (and garbage collection)

DLL –Node based-Operations

Algorithm removeLast():
  if size = 0 **then**
    Indicate an error: the list is empty
  $v \leftarrow$ trailer.getPrev() {last node}
  $u \leftarrow v$.getPrev() {node before the last node}
  trailer.setPrev(u)
  u.setNext(trailer)
  v.setPrev(**null**)
  v.setNext(**null**)
  size = size − 1

Algorithm addFirst(v):
  $w \leftarrow$ header.getNext() {first node}
  v.setNext(w)
  v.setPrev(header)
  w.setPrev(v)
  header.setNext(v)
  size = size + 1

Algorithm addAfter(v,z):
  $w \leftarrow v$.getNext() {node after v}
  z.setPrev(v) {link z to its predecessor, v}
  z.setNext(w) {link z to its successor, w}
  w.setPrev(z) {link w to its new predecessor, z}
  v.setNext(z) {link v to its new successor, z}
  size $\leftarrow$ size + 1

DLL –Node Based-Performance

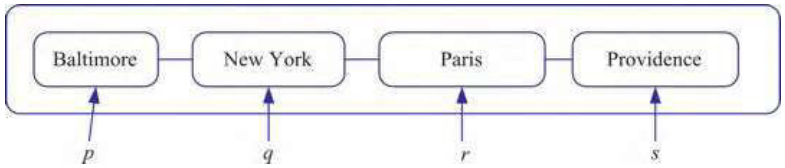
Operation	Worst case Complexity
size, isEmpty	$O(1)$
first, last, after	$O(1)$
before	$O(1)$
replaceElement, swapElements	$O(1)$
insertFirst, insertLast	$O(1)$
insertAfter	$O(1)$
insertBefore	$O(1)$
remove	$O(1)$

LIST -ADT

- One of the great benefits of a linked list structure is that it is possible to perform O(1)-time insertions and deletions at arbitrary positions of the list, as long as we are given a reference to a relevant node of the list.
- Such direct use of nodes would violate the object-oriented design principles of abstraction and encapsulation
- It will be simpler for users of our data structure if they are not bothered with unnecessary details of our implementation
- We ensure that users cannot invalidate the consistency of a list by mismanaging the linking of nodes
- For these reasons, instead of relying directly on nodes, we introduce an independent ***position abstraction*** to denote the location of an element within a list, and then a complete positional list ADT that can encapsulate a doubly linked list

POSITION -ADT

- "place" of an element relative to others in the list.
- In this framework, we view a list as a collection of elements that stores each element at a position and that keeps these positions arranged in a linear order



A node list. The positions in the current order are *p*, *q*, *r*, and *s*.

POSITION ADT

- The positions are arranged in a linear order
- A position is itself an abstract data type that supports the following simple method
  - **element(): Return the element stored at this position**
- A position is always defined relatively
- A position p will always be "after" some position q and "before" some position s (unless p is the first or last position).
- The position *p* does not change even if we replace or swap the element *e* stored at *p* with another element
- They are viewed internally by the linked list as nodes, but from the outside, they are viewed only as positions
- We can give each node *v* instance variables prev and next that respectively refer to the predecessor and successor nodes of *v*

LIST - ADT

- Container of elements that stores each element at a position
- Using the concept of position to encapsulate the idea of "node" in a list, the following methods can be defines for a list
- ***L.first ( ):***Return the position of the first element of S; an error occurs if S is empty.
- ***L.last()*** :Return the position of the last element of S; an error occurs if S is empty.
- ***L.isFirst(p)*** :Return a Boolean value indicating whether the given position is the first one in the list.
- ***L.islast (p)***
- ***L.before(p)*** :Return the position of the element of S preceding the one at position p; an error occurs if p is the first position.
- ***L.after (p) ,L.size(), L.isEmpty()***

LIST – ADT-Update methods

- ***L.replace(p, e)*** : Replace the element at position p with e, returning the element formerly at position p.
- ***L.swap (p , q)*** : Swap the elements stored at positions p and q, so that the element that is at position p moves to position q and the element that is at position q moves to position p.
- ***L.add_First(e)*** : Insert a new element e into S as the first element.
- ***L.add_last(e)*** : Insert a new element e into S as the last element.
- ***L.add_before (p, e)*** : Insert a new element e into S before position p in S; an error occurs if p is the first position.
- ***L.add_after(p, e)*** : Insert a new element e into S after position p in S; an error occurs if p is the last position.

LIST ADT Operation-Position Based

Operation	Return Value	L
L.add_last(8)	p	8_p
L.first( )	p	8_p
L.add_after(p, 5)	q	$8_p, 5_q$
L.before(q)	p	$8_p, 5_q$
L.add_before(q, 3)	r	$8_p, 3_r, 5_q$
r.element( )	3	$8_p, 3_r, 5_q$
L.after(p)	r	$8_p, 3_r, 5_q$
L.before(p)	None	$8_p, 3_r, 5_q$
L.add_first(9)	s	$9_s, 8_p, 3_r, 5_q$
L.delete(L.last( ))	5	$9_s, 8_p, 3_r$
L.replace(p, 7)	8	$9_s, 7_p, 3_r$

LIST – ADT: Linked List Implementation

LIST - ADT-Doubly linked list Implementation

- The nodes of the linked list implement the position ADT, defining a method element ( ) , which returns the element stored at the node.
- The nodes themselves act as positions.

LIST - ADT-Doubly linked list Implementation

- **INSERTION**

Algorithm *add_after(p,e):*

//Inserting an element e after a position p in a linked list.

Create a new node *v*

v.element ←*e*

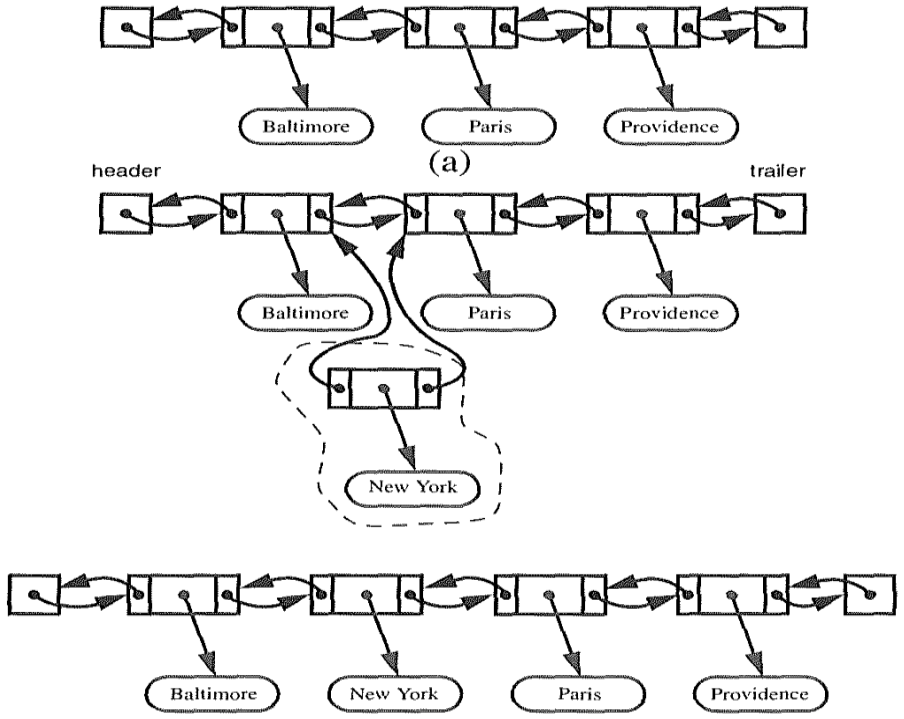
v.prev← *p* {link *v* to its predecessor}

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*}



LIST - ADT-Doubly linked list Implementation

- **DELETION**
  Algorithm delete(p):
  $t \leftarrow p$.element {a temporary variable to hold the return value}
  (p.prev).next ←p.next {linking out p}
  (p.next).prev ←p.prev
  p.prev←**nul** {invalidating the position p}
  p.next←**null**
  return t

- link the two neighbours of p to refer to one another as new neighbours-linking out p.

LIST - ADT linked list Implementation

- In the implementation of the List ADT by means of a linked list
  - All the operations of the List ADT run in $O(1)$ time
  - Operation element() of the Position ADT runs in $O(1)$ time

VECTOR-ADT

A Vector stores a list of elements:

- Access via rank/index.

Accessor methods:
elementAtRank(r),
Update methods:
replaceAtRank(r, o),
insertAtRank(r, o),
removeAtRank(r)
Generic methods:
size(), isEmtpy()

Here r is of type integer, n, m are nodes, o is an object (data).

VECTOR-ADT

Method	Time
size()	$O(1)$
isEmpty()	$O(1)$
elemAtRank(r)	$O(1)$
replaceAtRank(r, e)	$O(1)$
insertAtRank(r, e)	$O(n)$
removeAtRank(r)	$O(n)$

Iterators ADT

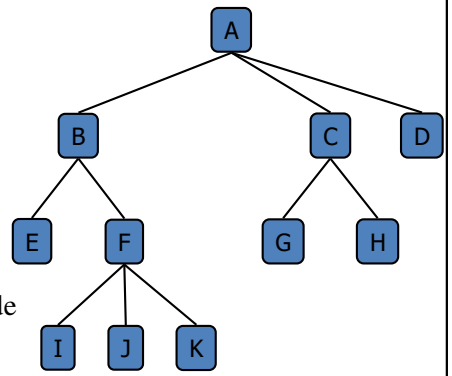
- Iterator allows to traverse the elements in a list or set.
- Iterator ADT provides the following methods:
  - object(): returns the current object
  - hasNext(): indicates whether there are more elements
  - nextObject(): goes to the next object and returns it

TREES

- In computer science, a tree is an abstract model of a hierarchical structure
- A tree consists of nodes with a parent-child relation
- **Applications:**
  - Organization charts
  - File systems
  - Compiler Design/Text processing (syntax analysis & to display the structure of a sentence in a language)
  - Searching Algorithms
  - Evaluating a mathematical expression.

TREE Terminologies

- ***Root:*** node without parent (A)
- ***Internal node***: node with at least one child (A, B, C, F)
- ***External node*** (a.k.a. leaf): node without children (E, I, J, K, G, H, D)
- ***Ancestors of a node***: parent, grandparent, grand-grandparent, etc.
- ***Depth of a node***: number of ancestors
- ***Height of a tree***: maximum depth of any node (3)
- ***Descendant of a node***: child, grandchild, grand-grandchild, etc.
- ***Degree of a node***: the number of its children



TREE ADT-Depth and Height

- Let v be a node of a tree T.
- The depth of v is the number of ancestors of v, excluding v itself
- If v is the root, then the depth of v is 0.
- Otherwise, the depth of v is one plus the depth of the parent of v.

Algorithm depth(T, v):
if T. isRoot(v) then
return 0
else
return 1 + depth (T, T. parent(v))

- The running time of algorithm depth (T, v) is $O(1 + d_v)$, where d_v denotes the depth of the node v in the tree T.
- In the worst case, the depth algorithm runs in O (n) time, where n is the total number of nodes in the tree T

Algorithm depth(T, v):
if T. isRoot(v) then
return 0
else
return 1 + depth (T, T. parent(v)

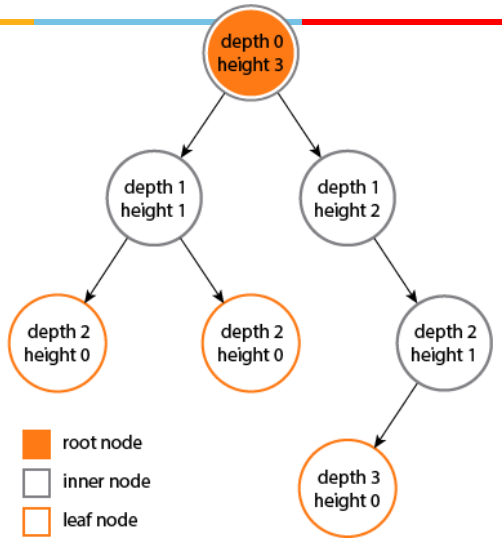
TREE ADT-Depth and Height

- The ***height of a tree*** T is equal to the maximum depth of an external node of T.
- If v is an external node, then the height o f v is 0.
- Otherwise, the height of v is one plus the maximum height of a child of v.

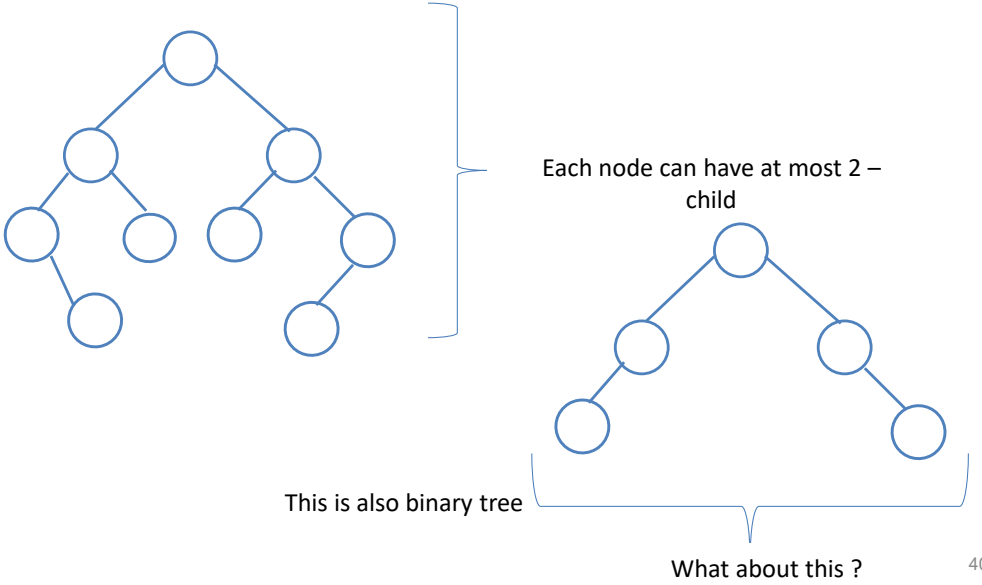
Algorithm height(T, v) :

```
if T. isExternal (v) then
return 0
else
h = 0
for each w ∈ T.children (v) do
h = max(h, height(T, w))
return 1 + h
```

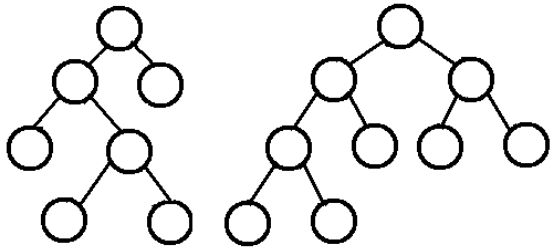

TREE ADT-Depth and Height



Binary Tree



Complete and Full(Proper) Binary tree



A **full binary tree** (sometimes proper **binary tree** or 2-**tree**) is a **tree** in which every node other than the leaves has two children

In Complete binary tree All levels except possibly the last are completely filled and all nodes are as left as possible.

TREE ADT

- We use **Positions** to abstract nodes
- Generic methods:
  - ***integer size():***Return the number of nodes in the tree.
  - ***boolean isEmpty()*** :
  - ***ObjectIterator elements():***Return an iterator of all the elements stored at nodes of the tree.
  - ***positionIterator positions():***Return an iterator of all the nodes of the tree
- Accessor methods:
  - ***position root():***Return the root of the tree
  - ***position parent(p):***Return the parent of node v; error if v is root.
  - ***positionIterator children(p):***Return an iterator of the children of node v.

Back

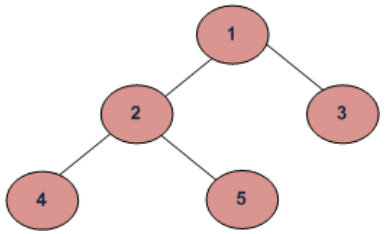
TREE ADT

- Query methods:
  - ***boolean isInternal(p):***Test whether node v is internal.
  - ***boolean isExternal(p):***Test whether node v is external
  - ***boolean isRoot(p):***Test whether node v is the root.
- Update methods:
  - ***swapElements(v, w):***Swap the elements stored at the nodes v and w.
  - ***object replaceElement(v, e):***Replace with e and return the element stored at node v

Assumptions

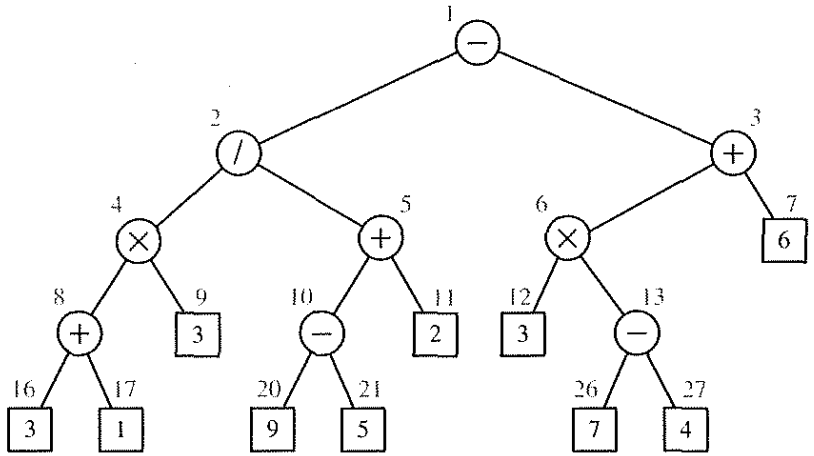
- The accessor methods root() and parent(v) take $O(1)$ time.
- The query methods isInternal(v), isExternal(v), and isRoot(v) take $O(1)$ time, as well.
- The accessor method children(v) takes $O(c_v)$ time, where c_v is the number of children of v.
- The generic methods swapElements(v,w) and replaceElement(v,e) take $O(1)$ time.
- The generic methods elements() and positions(), which return iterators, take $O(n)$ time, where n is the number of nodes in the tree.
- For the iterators returned by methods elements(), positions(), and children(v), the methods hasNext(), nextObject() or nextPosition() take $O(1)$ time each.

TREE Traversals



(a) Inorder (Left, Root, Right) : 4 2 5 1 3
(b) Preorder (Root, Left, Right) : 1 2 4 5 3
(c) Postorder (Left, Right, Root) : 4 5 2 3 1

TREE Traversals-Qn1-HW



QN-2-Class Discussion

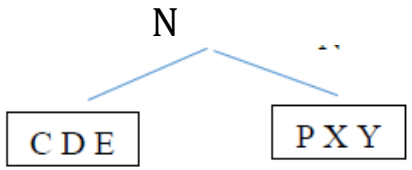
- Find the preorder traversal for the binary tree using
- Inorder traversal : C D E N P X Y
- Postorder traversal: D C E P Y X N
- Also construct the tree.

Qn2:Traversal-ANSWER

- Inorder traversal -C D E N P X Y
- Post order traversal - D C E P Y X N
- We first find the last node in postorder. The last node is “N”, we know this value is root as root always appear in the end of postorder traversal.
- We search “N” in inorder to find left and right subtrees of root. Everything on left of “N” in inorder is in left subtree and everything on right is in right subtree.

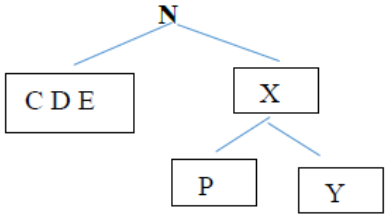
Qn2:Traversal-ANSWER

- Inorder traversal -C D E N P X Y
- Post order traversal - D C E P Y X N
-



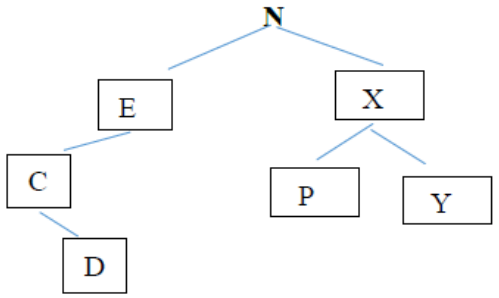
Qn2:Traversal-ANSWER

- Inorder traversal -C D E N P X Y
- Post order traversal - D C E P Y X N
- We recur the above process for following
- Recur for in = {P,X,Y} and post = {P,Y,X}
- Make the created tree as right child of root



Qn2:Traversal-ANSWER

- Inorder traversal -C D E N P X Y
- Post order traversal - D C E P Y X N
- Recur for in = {C,D,E} and post = {D,C,E}



Qn2:Traversal-ANSWER

- Inorder traversal -C D E N P X Y
- Post order traversal - D C E P Y X N
-
- **ANSWER: N E C D X P Y**

QN-3-HW

- Suppose that you are using a doubly linked list, maintaining a reference to the first and last node in the list, along with its size, to implement the operations below
- **addFirst(item):** prepend the item to the beginning of the list
- **get(i)** :return the item at position i in the list
- **set(i, item)** :replace position i in the list with the item
- **removeLast()** delete and return the item at the end of the list

- What is the worst-case running time of each of the operation above?

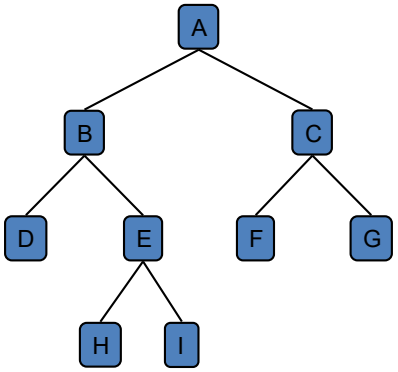
QN-4-HW

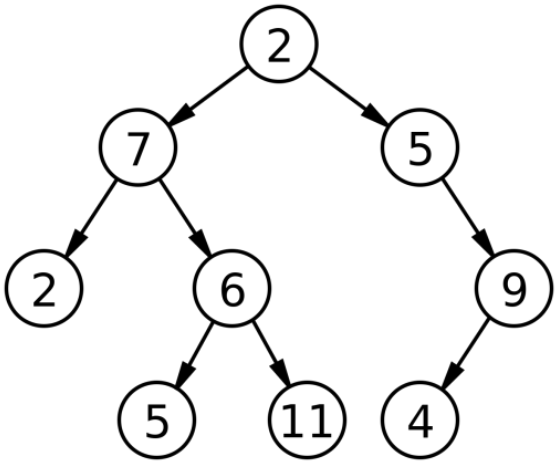
- We can construct a full binary tree from pre order and post order traversals. Discuss with an example.

Binary Trees

- A binary tree is a tree with the following properties:
  - Each internal node has two children
  - Each internal node has **ATMOST** two children
  - The children of a node are an ordered pair
- We call the children of an internal node -**left child and right child**
- Alternative recursive definition: a binary tree is either
  - a tree consisting of a single node, or
  - a tree whose root has an ordered pair of children, each of which is a binary tree

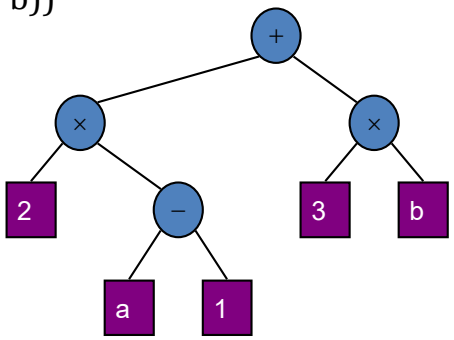
Binary Trees





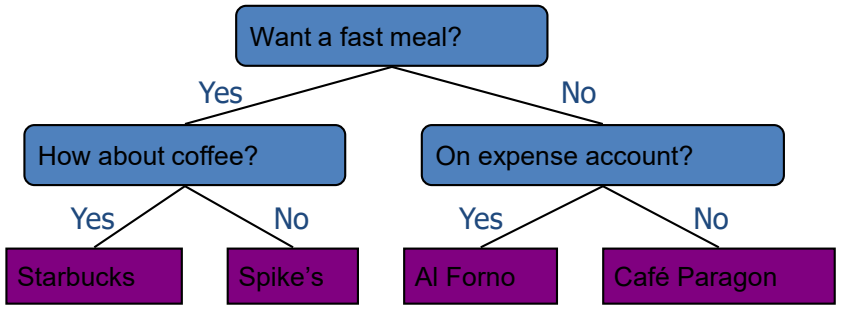
Arithmetic Expression Tree

- Binary tree associated with an arithmetic expression
  - internal nodes: operators
  - external nodes: operands
- Example: arithmetic expression tree for the expression $((2 \times (a - 1)) + (3 \times b))$



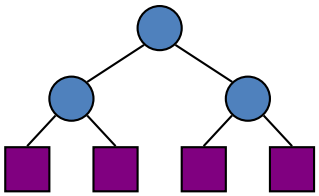
Decision Tree

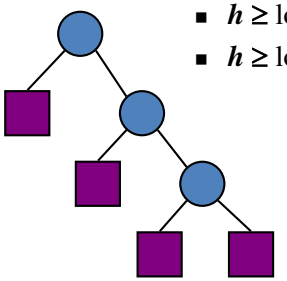
- Binary tree associated with a decision process
  - internal nodes: questions with yes/no answer
  - external nodes: decisions
- Example: dining decision



Properties of (proper)Binary Trees

- Notation
  - ***n*** number of nodes
  - ***e*** number of external nodes
  - ***i*** number of internal nodes
  - ***h*** height





- Properties:
  - $e = i + 1$
  - $n = 2e - 1$
  - $h \leq i$
  - $h \leq (n - 1)/2$
  - $e \leq 2^h$
  - $h \geq \log_2 e$
  - $h \geq \log_2 (n + 1) - 1$

Properties of Binary Trees

Let T be a (proper) binary tree with n nodes, and let h denote the height of T. Then T has the following properties:

- The number of external nodes in T is at least h + 1 and at most 2^h .
- The number of internal nodes in T is at least h and at most $2^h - 1$.
- The total number of nodes in T is at least 2h + 1 and at most $2^{h+1} - 1$.
- The height of T is at least log(n + 1) -1 and at most (n - 1) /2, that is, log(n + 1) - 1 <=h <= (n - 1)/2.
- The number of external nodes is 1 more than the number of internal nodes.

BinaryTree ADT-Methods

- The BinaryTree ADT extends the Tree ADT, i.e., it inherits all the methods of the Tree ADT

Additional methods:

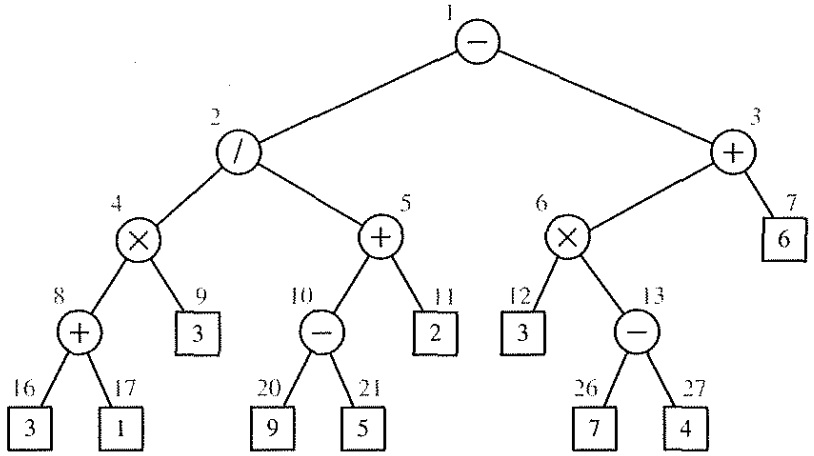
- ***position leftChild(v)*** Return the left child of v; an error condition occurs if v is an external node.
- ***position rightChild(v)***
- ***position sibling(p)***

- Update methods may be defined by data structures implementing the BinaryTree ADT

Binary Trees-Representations Vector Based

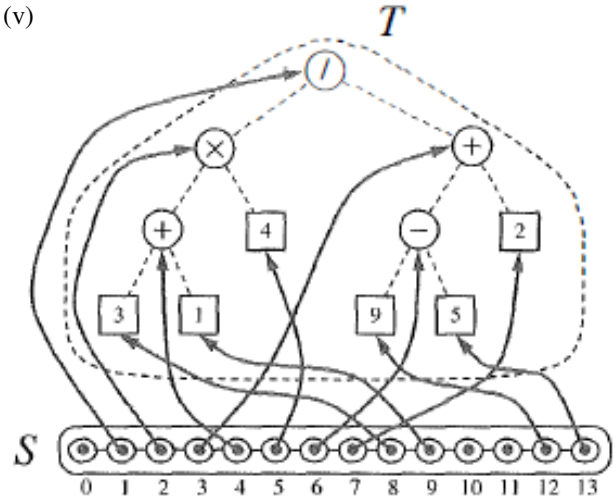
- For every node v of T , let p(v) be the integer defined as follows.
- If v is the root of T, then p(v) = 1 .
- If v is the left child of node u, then p(v) = 2p(u) .
- If v is the right child of node u, then p(v) = 2p (u) + 1 .
- ***p is known as a level numbering***

Binary Trees-Representations Vector Based



Binary Trees-Representations- Vector Based

Node v of T is associated with the element of S at rank p (v)



Binary Trees-Representations Vector Based

- Let n be the number of nodes of T, and let P_M be the maximum value of p (v) over all the nodes of T .
- Vector S has size $N = P_M + 1$ since the element of S at rank 0 is not associated with any node of T .
- Vector S will have, in general, a number of empty elements that do not refer to existing nodes of T.
- These empty slots could correspond to empty external nodes or even slots where descendants of such nodes would go
- In the worst case, $N = 2^{(n+1)/2}$

- //n is the number of nodes of T,N is the vector size

Binary Trees-Representations Vector Based

- Operation Time
  - positions, elements O(n)
  - swap Elements, replaceElement O(1)
  - root, parent, children O(1)
  - leftChild, rightChild, sibling O (1)
  - isInternal , isExternal, isRoot O(1)

Methods

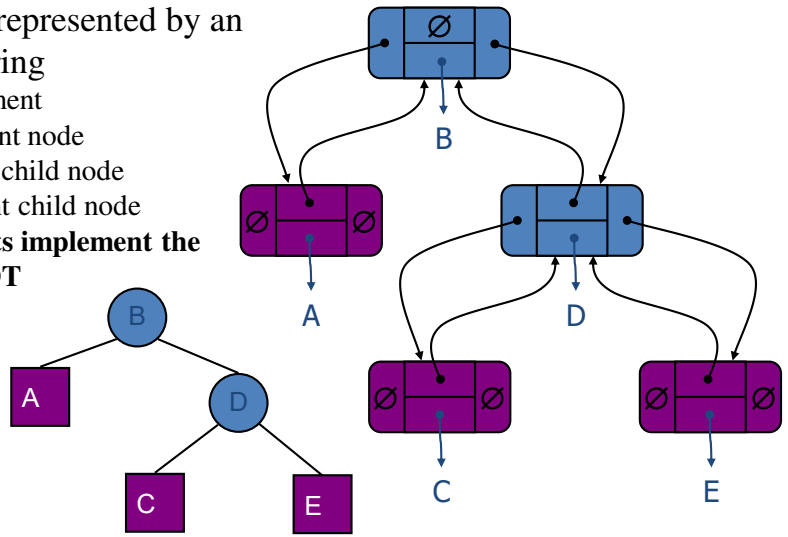
  - Fast and Simple
  - Space inefficient if the height of the tree is large

Binary Trees-Representations Linked Structure

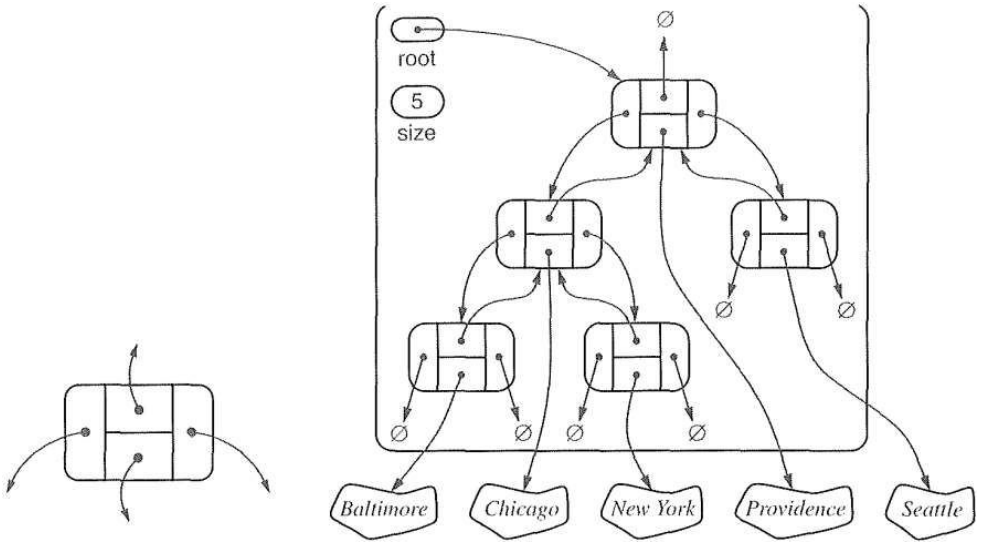
A node is represented by an object storing

- Element
- Parent node
- Left child node
- Right child node

Node objects implement the Position ADT



Binary Trees-Representations Linked Structure



Binary Trees-Representations Linked Structure

- Operation Time
- Size()
- isEmpty()
- swapElements(v,w)
- replaceElement(v,e)

O(1)

- Positions()
- Elements()

O(n)

Binary Trees-Applications

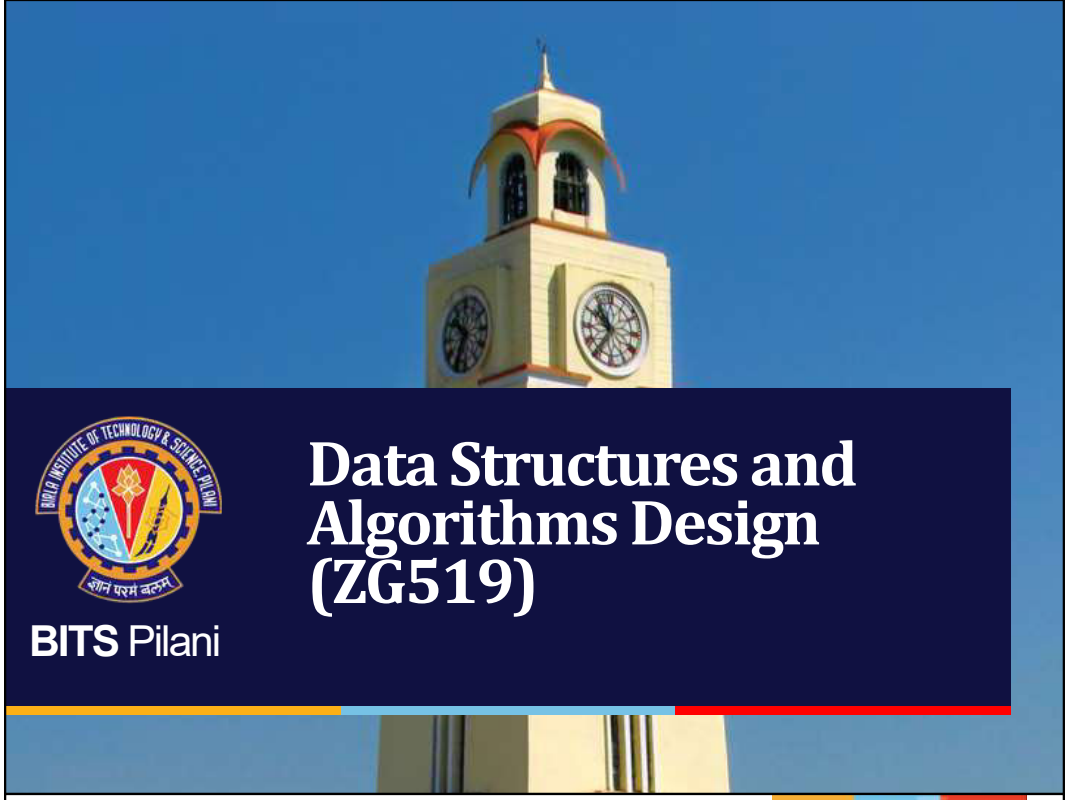
- Data Base indexing
- BSP in Video Games
- Path finding algorithms in AI applications
- Huffman coding
- Heaps
- SET AND MAP IN C++
- Syntax tree

BITS Pilani
Hyderabad Campus

THANK YOU!

SESSION 4 -PLAN

Contact Sessions(#)	List of Topic Title	Text/Ref Book/external resource
4	Heaps - Definition and Properties, Representations (Vector Based and Linked), Insertion and deletion of elements	T1: 2.4



BITS Pilani

Data Structures and Algorithms Design (ZG519)

The Heap Data Structure

- The heap is a binary tree T that stores a collection of keys at its nodes and that satisfies two additional properties: a relational property defined in terms of the way keys are stored in T and a structural property defined in terms of T itself.
- We assume that a **total order relation** on the keys is given, for example, by a comparator.

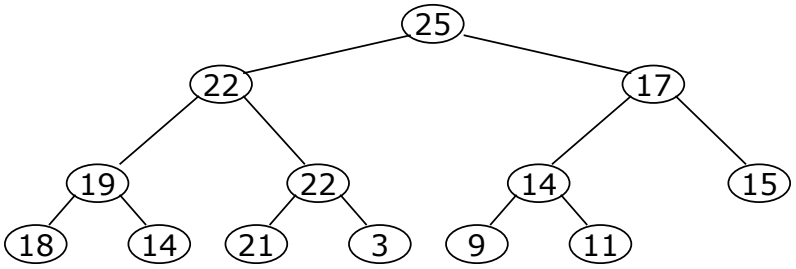
Total order relation

- Comparison rule is defined for every pair of keys and it must satisfy the following properties:
  - **Reflexive property**: $k \leq k$.
  - **Antisymmetric property**: if $k_1 \leq k_2$ and $k_2 \leq k_1$, then $k_1 = k_2$.
  - **Transitive property**: if $k_1 \leq k_2$ and $k_2 \leq k_3$, then $k_1 \leq k_3$.
- Any comparison rule <= that satisfies these three properties will never lead to a comparison contradiction.

The Heap Data Structure:

- A heap is a binary tree storing elements at its nodes and satisfying the following properties:
  - **Heap-Order property (Relational property):**
    - for every node v other than the root, ***key***(***v***) ≥ ***key***(***parent***(***v***)) –Min heap
    - ***key***(***v***)<= ***key***(***parent***(***v***))-Max heap
- The keys encountered on a path from the root to an external node of T are in non decreasing /non increasing order.
- A minimum/max key is always stored at the root of T.

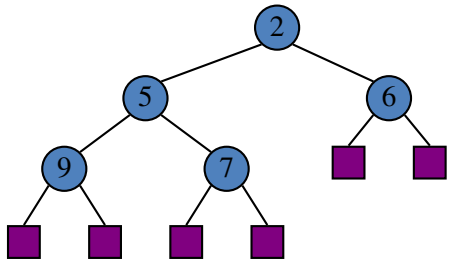
The Heap Data Structure



The Heap Data Structure

Structural property:

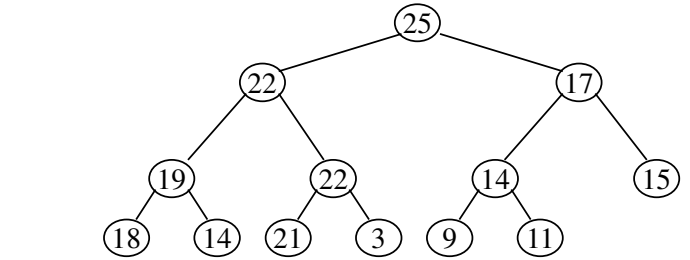
- Heap data structure is always a complete binary tree



Vector-based Heap Implementation

- We can represent a heap with n keys by means of a vector of length $n + 1$
- For the node at rank i
  - the left child is at rank $2i$
  - the right child is at rank $2i + 1$
- Links between nodes are not explicitly stored
- In other words the parent of a node stored in i^{th} location is at floor$[i/2]$
- The cell at rank 0 is not used

Vector-based Heap Implementation

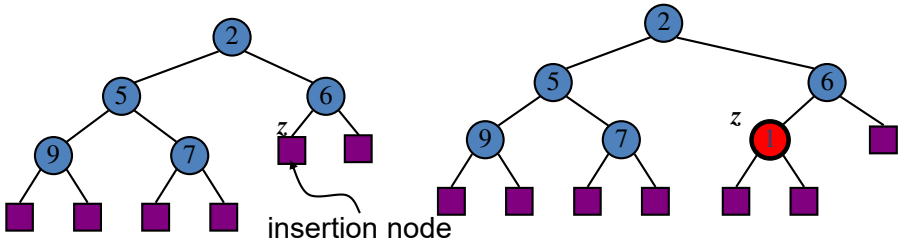


0	1	2	3	4	5	6	7	8	9	10	11	12	13
	25	22	17	19	22	14	15	18	14	21	3	9	11

- the left child is at rank 2i
- the right child is at rank 2i + 1
  - Example: the children of node 3 (17) are 6 (14) and 7 (15)
  - Note that when the heap T is implemented with a vector, the index of the last node is always equal to n

Insertion into a Heap

- Method insertItem corresponds to the insertion of a key k to the heap
- The insertion algorithm consists of three steps
  - Find the insertion node z (the new last node)
  - Store k at z and expand z into an internal node
  - Restore the heap-order property (discussed next)

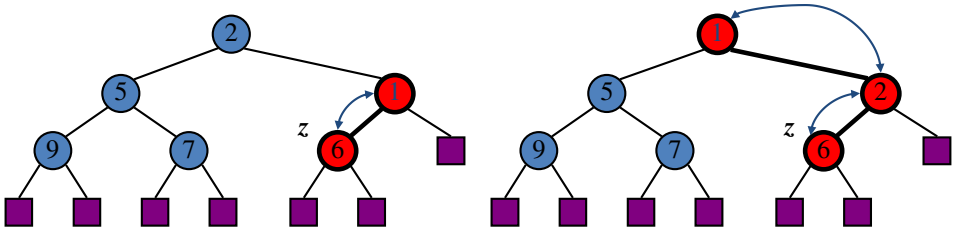


  - External nodes are represented a

Upheap

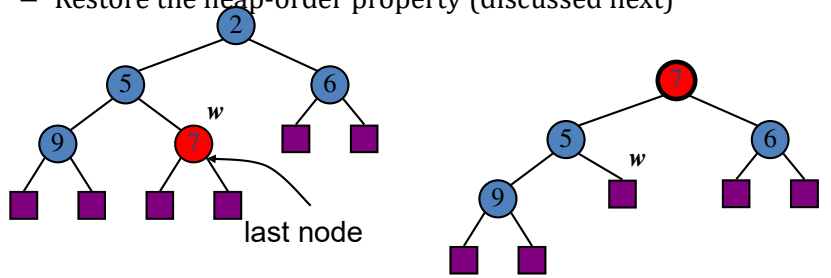
- After the insertion of a new key k, the heap-order property may be violated
- Algorithm upheap restores the heap-order property by swapping k along an upward path from the insertion node
- Upheap terminates when the key k reaches the root or a node whose parent has a key smaller than or equal to k
- Since a heap has height $O(\log n)$, upheap runs in $O(\log n)$ time

Upheap



Removal from a Heap

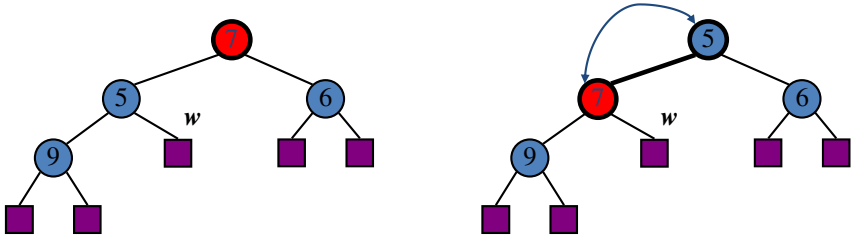
- Method removeMin corresponds to the removal of the root key from the (min) heap
- The removal algorithm consists of three steps
  - Replace the root key with the key of the last node ***w***
  - Compress ***w*** and its children into a leaf
  - Restore the heap-order property (discussed next)



Downheap

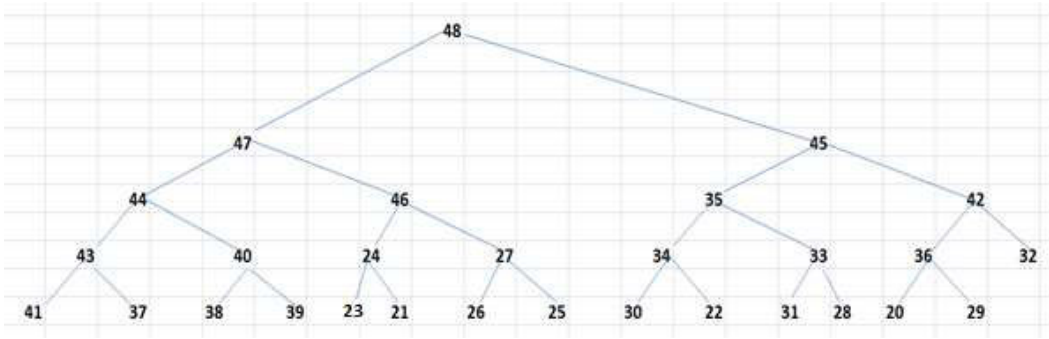
- After replacing the root key with the key ***k*** of the last node, the heap-order property may be violated
- Algorithm downheap restores the heap-order property by swapping key ***k*** along a downward path from the root
- Downheap terminates when key ***k*** reaches a leaf or a node whose children have keys greater than or equal to ***k***
- Since a heap has height ***O***(log ***n***), downheap runs in ***O***(log ***n***) time

Downheap



QN-1

- Consider the following binary heap.



QN-1 Contd.

- Suppose that the last operation performed in the binary heap above was inserting the key x. Find all possible values of x. Give suitable explanations
- Suppose that you delete the maximum key from the binary heap above. Find all keys that are involved in one (or more) comparisons. Show the comparisons.

Solution:

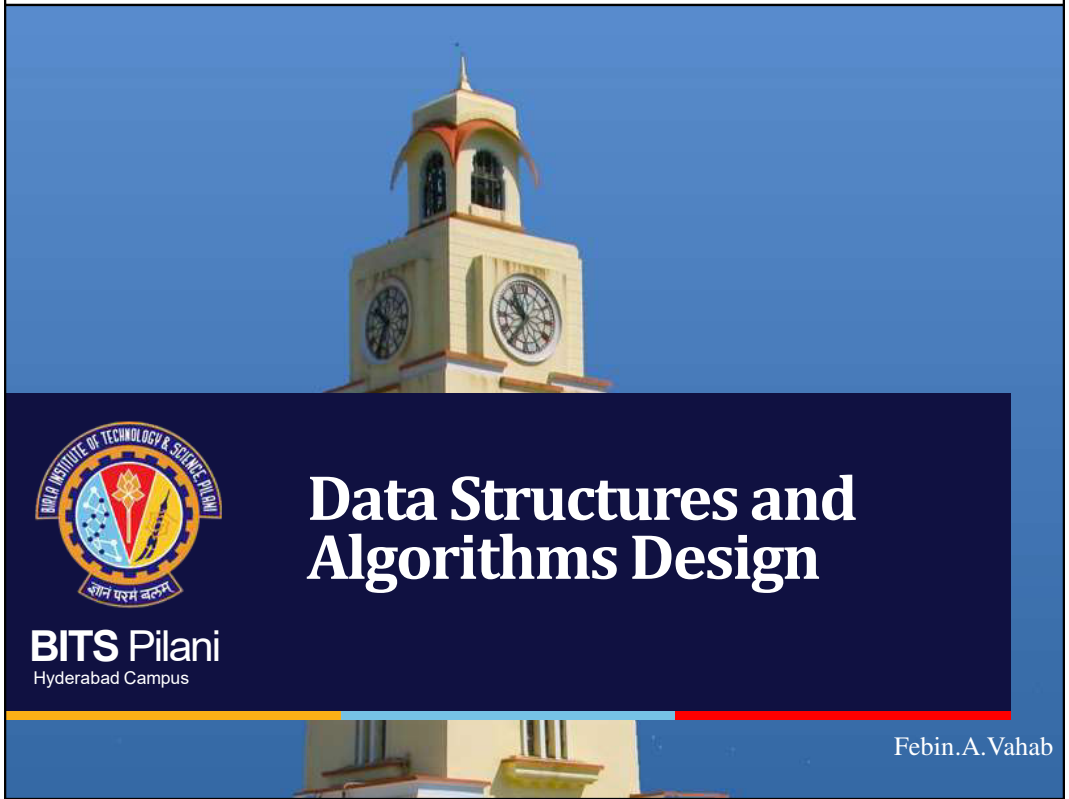
- 29 36 42 45 .To insert a node in a binary heap, we place it in the next available leaf node and swim it up. Thus, 29 36 42 45, and 48 are the only keys that might move. But, the last inserted key could not have been 48, because, then, 45 would have been the old root (which would violate heap order because the left child of the root is 47).
- 24 ,27,29,44,45,46,47 The compares are 45-47,29-47, 44-46, 29-46 ,24-27, 29-27

QN-2

- Write an algorithm to find the **kth largest** element of an array using a
  - **Min** Heap.
  - **Max** Heap
- What is the time complexity in both cases if the array is **unsorted**?

QN-2-Solution

- **For Min Heap**: Create a min heap of size k by taking k elements of the array. Now we compare the root with the remaining elements of the array. If the element is greater than the root, then replace root with this element and min_heapify .Iterate this for all elements of the array. The root of the heap will be the kth largest element.
- Time Complexity: O(k + (n-k)Logk)



QN-3

- **For Max Heap**: Create a max heap of size n by taking all elements of the array. Now remove the root k-1 times and re-heapify. The root of the heap now will be the kth largest element.
- Time Complexity: for building a Max Heap – O(n), for deleting the root k-1 times – O((k-1)log n), so total is O(n + (k-1)logn)

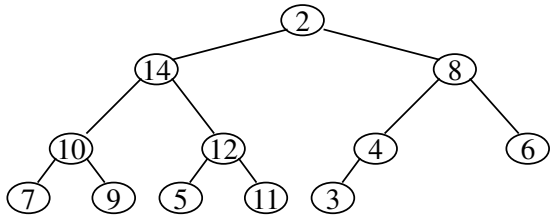
SESSION 4 -PLAN

Contact Sessions(#)	List of Topic Title	Text/Ref Book/external resource
4	Heap implementation of priority queue, Heap sort	T1: 2.4 T2:6

Heapify

- Before discussing the method for building heap from an arbitrary complete binary tree, we discuss a simpler problem.
- Let us consider a binary tree in which left and right subtrees of the root satisfy the heap property but not the root.

  See the following fig

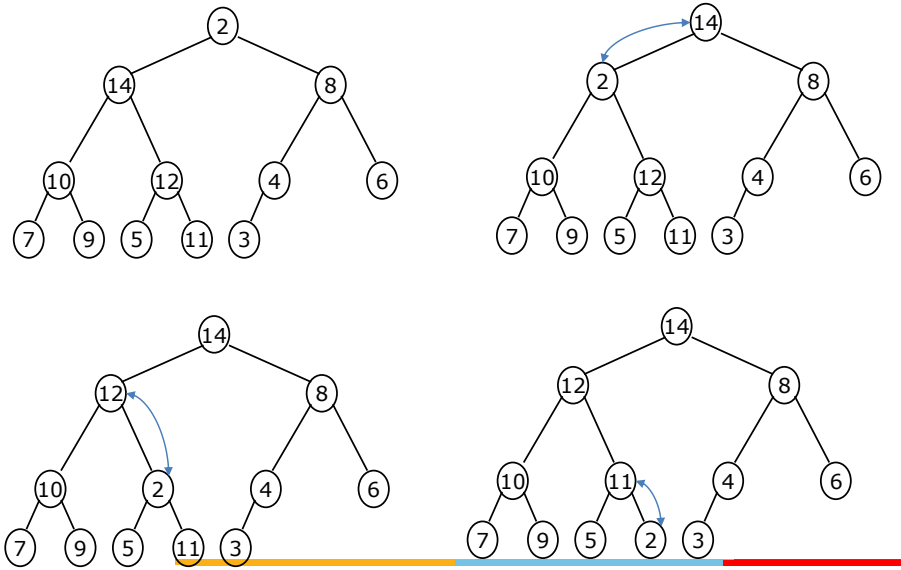


- Now the Question is how to transform the above tree into a Heap?

Solution for the Question

- Swap the root and left child of root, to make the root satisfy the heap property.
- Then check the subtree rooted at left child of the root is heap or not. If it is we are done, if not repeat the above action of swapping the root with the maximum of its children.
- That is, push down the element at root till it satisfies the heap property.
- The following sequence of fig depicts the **heapification** process

Heapify



Max-Heapify

```
MAX-HEAPIFY(A, i)
1  l = LEFT(i)
2  r = RIGHT(i)
3  if l ≤ A.heap-size and A[l] > A[i]
4       largest = l
5  else largest = i
6  if r ≤ A.heap-size and A[r] > A[largest]
7       largest = r
8  if largest ≠ i
9       exchange A[i] with A[largest]
10      MAX-HEAPIFY(A, largest)
```

Time complexity????

O(log n) (Height of the tree)

Building a MAX heap

- By means of n successive insert item operations we can construct a heap storing n key-element pairs in O (n log n) time
- Example

Building a Max Heap-Bottom Up

Build-Max-Heap(A)

- // Input: A: an (unsorted) array
- // Output: A modified to represent a heap.
- // Running Time: O(n) where n = length[A]

heap-size[A] ← *length[A]*

for i ← *floor(length[A]/2) downto 1*

Max-Heapify(A, i)

Why start at n/2?

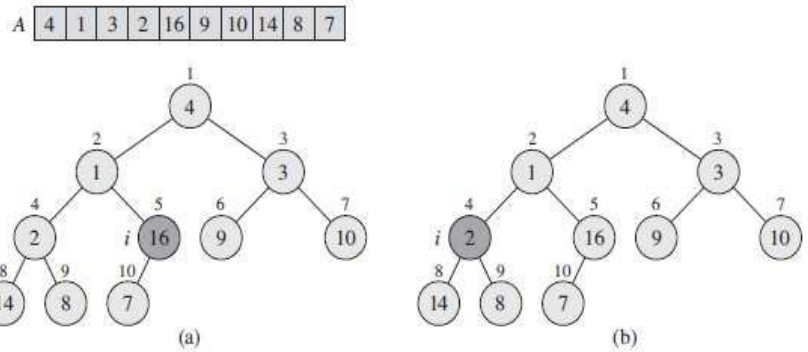
– Because elements A[n/2 + 1 … n] are all leaves of the tree

– 2i > n, for i > n/2 + 1

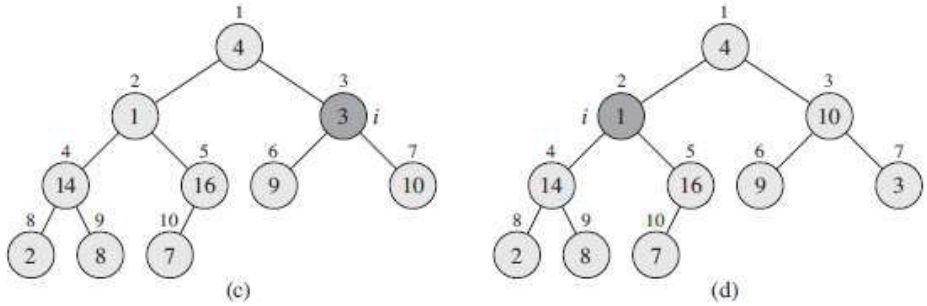
Building a Max Heap-Bottom Up-Steps

- Step 0: Initialize the structure with keys in the order given
- Step 1: Starting with the last (rightmost) parental node, fix the heap rooted at it, if it doesn't satisfy the heap condition: keep exchanging it with its largest child until the heap condition holds
- Step 2: Repeat Step 1 for the preceding parental node

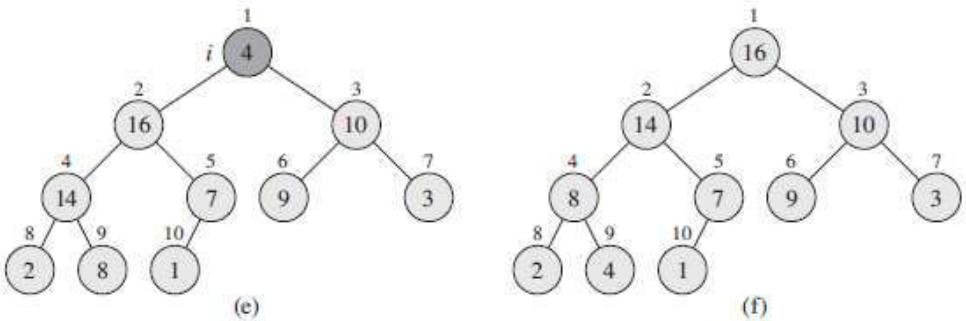
Building a Max Heap-Bottom Up



Building a Max Heap-Bottom Up



Building a Max Heap-Bottom Up



Building a Max Heap-Bottom Up-Analysis

- The time for MAX-HEAPIFY to run at a node varies with the height of the node in the tree
- The heights of most nodes are small
- Our tighter analysis relies on the properties that an
  - n-element heap has height lg n
  - There are at most $\mathbf{n/2^{(h+1)}}$ nodes at any height h
  - ie
  - Max_Heapify takes O(1) for time for nodes that are one level above the leaves, and in general, O(L) for the nodes that are L levels above the leaves.

Building a Max Heap-Bottom Up-Analysis

- The total cost of BUILD-MAX-HEAP

$$T(n) = \sum_{h=0}^{logn} n/2^{h+1} * O(h)$$

$$= O(n/2 * \sum_{h=0}^{logn} h/2^h)$$

$$<= O(n/2 * \sum_{h=0}^{inf} h/2^h)$$

$$<= O(n * [1 + 1/2 + 1/4 + ...]) \; infinte \; GP$$

$$<= O(n * [1/(1 - (1/2))]$$

$$<= O(2n)$$
$$<= O(n))$$

Building a Max Heap-Bottom Up-Analysis

Running Time of Build-Max-Heap

Trivial Analysis: Each call to Max-Heapify requires log(n) time, we make n such calls ⇒ O(n log n).

Tighter Bound: Each call to Max-Heapify requires time O(h) where h is the height of node i. Therefore running time is

$\sum_{h=0}^{\log n} \frac{n}{(2^{h+1})} * O(h) = \mathbf{O(n)}$

Priority Queue ADT

- A priority queue is a container of elements, each having an associated key that is provided at the time the element is inserted.
- The name "priority queue" comes from the fact that keys determine the "priority" used to pick elements to be removed
- An item is a pair (key, element)
- Main methods of the Priority Queue ADT
  - insertItem(k, o)
    inserts an item with key k and element o
  - removeMin()
    removes the item with smallest key and returns its element

Priority Queue ADT

- Additional methods
  - minKey()
    returns, but does not remove, the smallest key of an item
  - minElement()
    returns, but does not remove, the element of an item with smallest key
  - size(), isEmpty()
- Applications:
  - Standby flyers
  - Auctions
  - Stock market

Priority Queue ADT -Total Order Relation

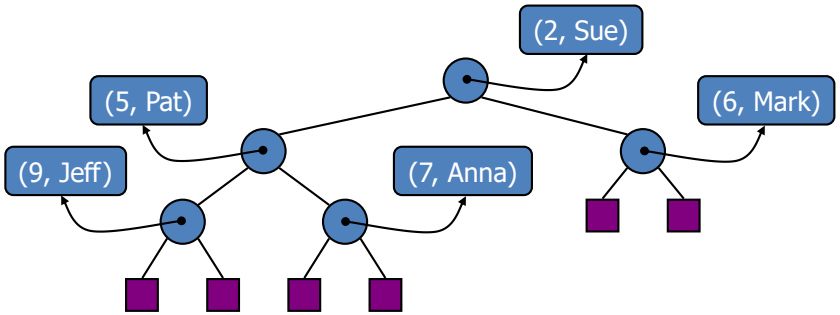
- Keys in a priority queue can be arbitrary objects on which an order is defined.
- Two distinct items in a priority queue can have the same key
- Total order relation

Sorting with a Priority Queue

- We can use a priority queue to sort a set of comparable elements
- Given a collection C of n elements that can be compared according to a total order relation, and we want to rearrange them in increasing order (or at least in nondecreasing order if there are ties).
  - Insert the elements into a PQ one by one with a series of **insertItem(e, e)** operations
  - Remove the elements in sorted order with a series of **removeMin()** operations putting them back into C
- The running time of this sorting method depends on the priority queue implementation

Heaps and Priority Queues

- We can use a heap to implement a priority queue
- A (key, element) item is stored at each internal node
- **Keep track of the position of the last node**



Heap-Sort

Heap-Sort(A)

// Input: A: an (unsorted) array

// Output: A modified to be sorted from smallest to largest

// Running Time: O(n log n) where n = length[A]

Build-Max-Heap(A)

for i = length[A] downto 2

exchange A[1] and A[i]

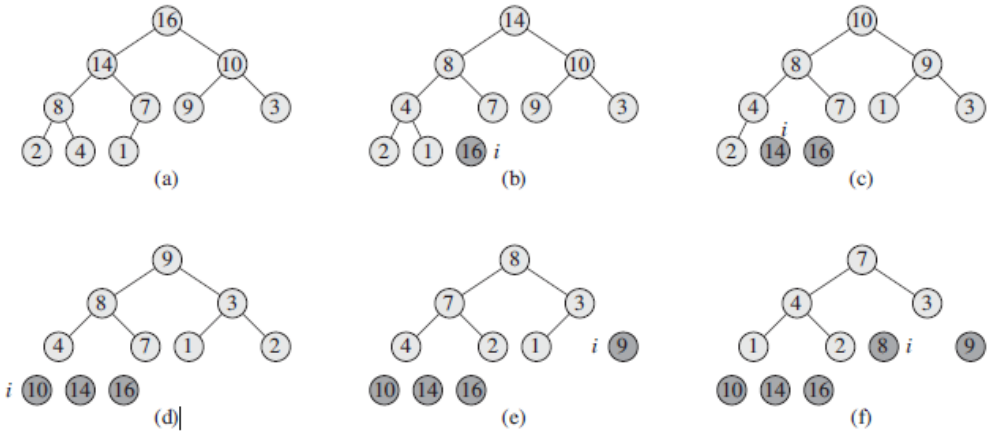
heap-size[A] ← heap-size[A] – 1

Max-Heapify(A, 1)

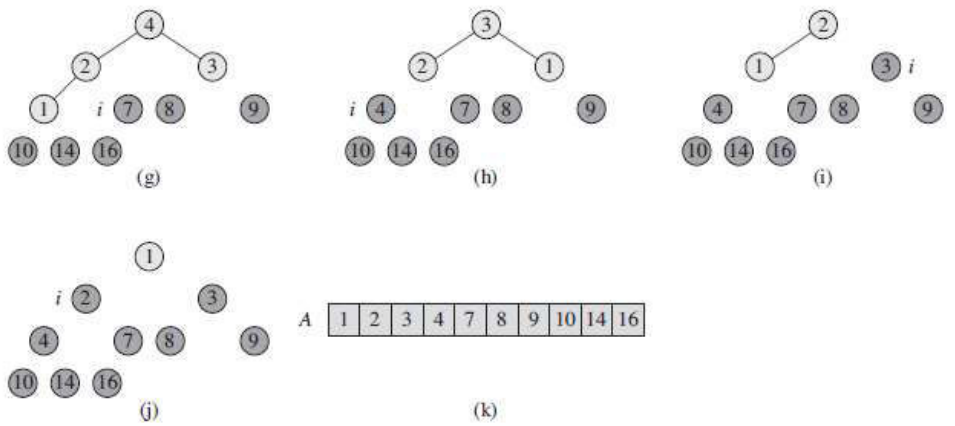
Heap Sort-Steps

- Given an array of n element, first we build the heap
- The largest element is at the root, but its position in sorted array should be at the end. So swap the root with the last.
- We have placed the highest element in its correct position we left with an array of n-1 elements. Repeat the same of these remaining n-1 element to place the next largest elements in its correct position.
- Repeat the above step till an elements are placed in their correct positions.

Heap Sort-Example



Heap Sort-Example



Heap-Sort

- Consider a priority queue with ***n*** items implemented by means of a heap
  - the space used is ***O***(***n***)
  - methods insertItem and removeMin take ***O***(log ***n***) time
  - methods size, isEmpty, minKey, and minElement take time ***O***(1) time
- Using a **heap-based priority queue**, we can sort a sequence of ***n*** elements in ***O***(***n*** log ***n***) time
- The resulting algorithm is called **heap-sort**
- Heap-sort is much faster than quadratic sorting algorithms, such as insertion-sort and selection-sort

Heap-Algorithms

Parent(A, i)

// Input: A: an array representing a heap, i: an array index
// Output: The index in A of the parent of i
// Running Time: O(1)
if i == 1 return NULL
return floor(i/2)

Heap-Algorithms

Left(A, i)

// Input: A: an array representing a heap, i: an array index
// Output: The index in A of the left child of i
// Running Time: O(1)
if $2 * i \leq$ *heap-size[A]*
return $2 * i$
else return NULL

Heap-Algorithms

Right(A, i)

// Input: A: an array representing a heap, i: an array index

// Output: The index in A of the right child of i

// Running Time: O(1)

if $2 * i + 1 \leq$ *heap-size[A]*

return $2 * i + 1$

else return NULL

Heap-Algorithms

Heap-Increase-Key(A, i, key)

// Input: A: an array representing a heap, i: an array index, key: a new key greater than A[i]

// Output: A still representing a heap where the key of A[i] was increased to key

// Running Time: O(log n) where n =heap-size[A]

if key < A[i]

    error("New key must be larger than current key")

A[i] ← key

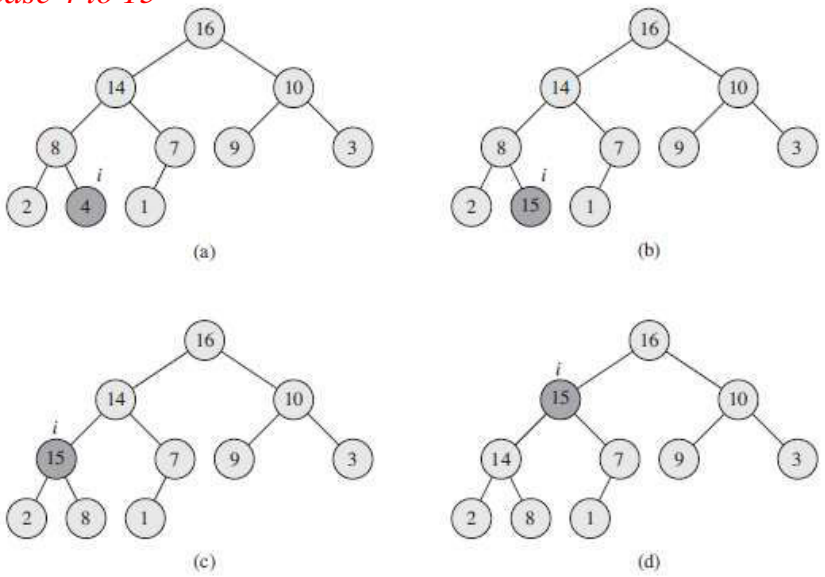
while i > 1 and A[Parent(i)] < A[i]

    exchange A[i] and A[Parent(i)]

    i ← Parent(i)

Heap-Increase-Key

increase 4 to 15



Heap-Algorithms

Heap-Extract-Max(A)

// Input: A: an array representing a heap

// Output: The maximum element of A and A as a heap with this element removed

// Running Time: O(log n) where n =heap-size[A]

max ← A[1]

A[1] ← A[heap-size[A]]

heap-size[A] ←heap-size[A] – 1

Max-Heapify(A, 1)

return max

Heap-Algorithms

Max-Heap-Insert(A, key)

// Input: A: an array representing a heap, key: a key to insert

// Output: A modified to include key

// Running Time: O(log n) where n =heap-size[A]

heap-size[A] ←heap-size[A] + 1

A[heap-size[A]] ← −∞

Heap-Increase-Key(A[heap-size[A]], key)

HW

Max-Heapify and Build-Max-Heap

Demonstrate how Build-Max-Heap turns it into a heap

7	9	3	5	6	8	13	32	12	22

HW

Heap-Increase-Key

Show what happens when you use HeapIncrease-Key to increase key 5 to 33

HW

Heap-Sort

Given the heap show how you use it to sort .Explain why the runtime of this algorithm is O(n log n)

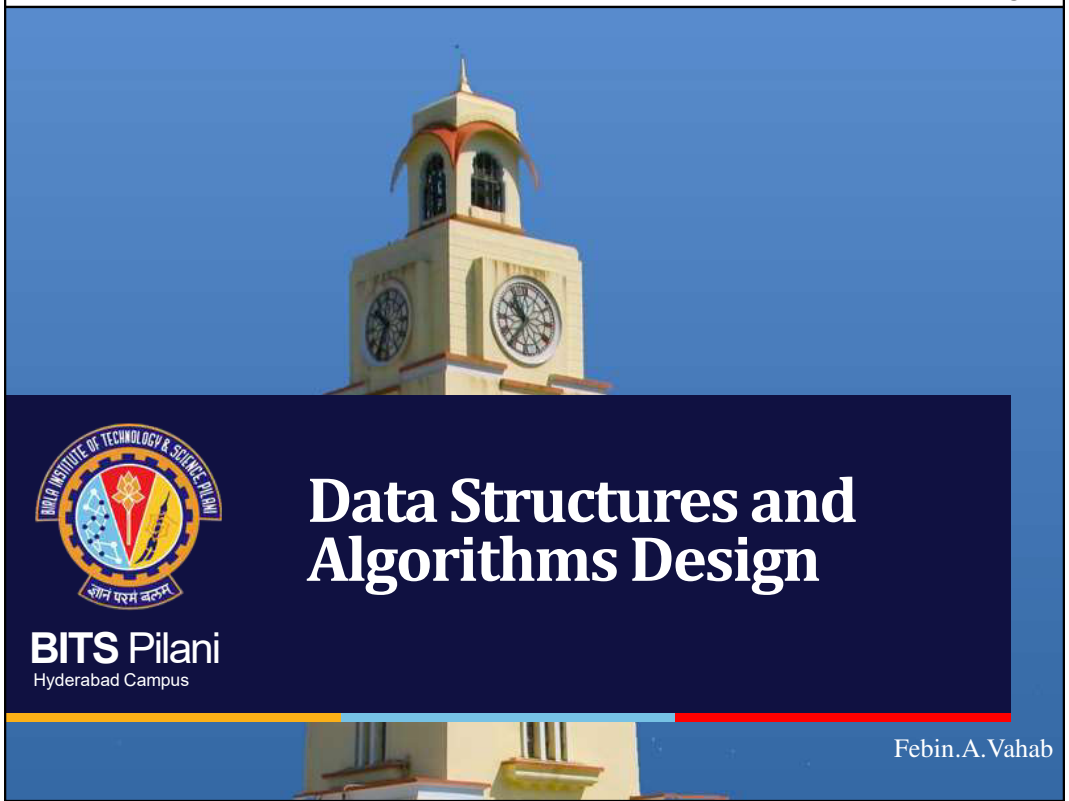
HW

Group 4:

Priority Queue implementation using an Unordered Sequence (implemented using an array/linked list)

Comparison of Different Priority Queue Implementations

Method	Unsorted Sequence	Sorted Sequence	Heap
size, isEmpty, key, replaceElement	$O(1)$	$O(1)$	$O(1)$
minElement, min, minKey	$O(n)$	$O(1)$	$O(1)$
insertItem, insert	$O(1)$	$O(n)$	$O(\log n)$
removeMin	$O(n)$	$O(1)$	$O(\log n)$
remove	$O(1)$	$O(1)$	$O(\log n)$
replaceKey	$O(1)$	$O(n)$	$O(\log n)$



Data Structures and Algorithms Design

BITS Pilani
Hyderabad Campus

Febin.A.Vahab

SESSION 5 -PLAN

Sessions(#)	List of Topic Title	Text/Ref Book/external resource
5	Unordered Dictionary :ADT, Applications Hash Tables: Notion of Hashing and Collision (with a simple vector based hash table)Hash Functions: Properties, Simple hash functions Methods for Collision Handling: Separate Chaining, Notion of Load Factor, Rehashing, Open Addressing [Linear; Quadratic Probing, Double Hash]	T1: 2.5

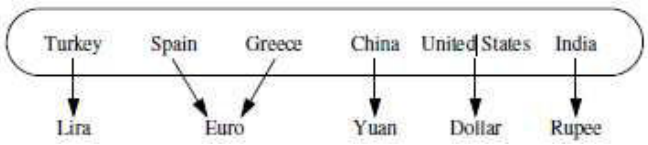
Dictionary ADT

- The dictionary ADT models a searchable collection of key-element items.
- A dictionary stores **key-element pairs (k , e),** which we call **items**, where k is the key and e is the element
- The main operations of a dictionary are searching, inserting, and deleting items
- A key is an identifier that is assigned by an application or user to an associated element.
- *Multiple items with the same key are allowed*
- In cases when keys are unique, the key associated with an object can be viewed as an "address" for that object in memory.

The Dictionary ADT

- For example, in a dictionary storing student records (such as the student's name, address, and course grades), the key might be the **student's ID number. (**we would probably want to disallow two students having the same ID).

The Unordered Dictionary ADT



From countries (the keys) to their units of currency (the values).

- Dictionaries use an array-like syntax for indexing such as **currency[Greece]** to access a value associated with a given key **currency[Greece] = New value** to remap it to a new value.
- Unlike a standard array, indices for a dictionary need not be consecutive nor even numeric.

Applications

- The domain-name system (DNS)maps a host name, such as www.bits-pilani.ac.in,to an Internet-Protocol (IP) address.
- A social media site typically relies on a (nonnumeric) username as a key that can be efficiently mapped to a particular user's associated information.
- A computer graphics system may map a color name, such as turquoise , to the triple of numbers that describes the color's RGB (red-green-blue) representation,such as (64,224,208).
- Python uses a dictionary to represent each namespace, mapping an identifying string, such as pi , to an associated object, such as 3.14159.

Applications-HW

- Counting Word Frequencies
  - Consider the problem of counting the number of occurrences of words in a document.
  - A dictionary is an ideal data structure to use here, for we can use words as keys and word counts as values.

Try implementing this using Python Dictionary class.!!!

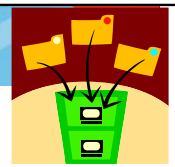
Dictionary ADT methods:

- Dictionary ADT methods:
  - **findElement(k)**: if the dictionary has an item with key k, returns its element, else, returns the special element NO_SUCH_KEY
  - **insertItem(k, e):** Insert an item with element e and key k into D
  - **removeElement(k)**: if the dictionary has an item with key k, removes it from the dictionary and returns its element, else returns the special element NO_SUCH_KEY
  - **size(), isEmpty()**
  - **keys(), elements()-Iterators**

Dictionary ADT methods:

- Special element (NO_SUCH_KEY)is known as a sentinel.
- If we wish to store an item 'e' in a dictionary so that the item is itself its own key, then we would insert e with the method call insertltem(e, e) .
- *findAIIElements(k) - which returns an iterator of all elements with key equal to k*
- *removeAIIElements (k) , which removes from D all the items with key equal to k*

Log Files/Unordered Sequence Implementation



- A log file is a dictionary implemented by means of an **unsorted sequence**
- Often called **audit trail**
- We store the items of the dictionary in a sequence (based on an array or list to store the key-element pairs), in arbitrary order
- The space required for a log file is O(n), since the array data structure can maintain its memory usage to be proportional to its size.

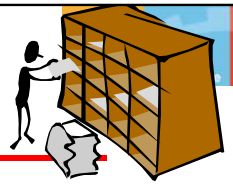
Log Files

- Performance:
  - **Insertion?** insert Item (k, e)
  - **insertItem takes O(1) time** since we can insert the new item at the end of the sequence
  - **Search?? Removal??**
  - **[findElement(k), removeElement(k)]**
  - findElement and removeElement **take O(n) time** since in the worst case (the item is not found) we traverse the entire sequence to look for an item with the given key
- The log file is effective only for dictionaries of small size or for dictionaries on which insertions are the most common operations, while searches and removals are rarely performed
- (e.g., historical record of logins to a workstation)

Dictionary Implementation using Hash Tables

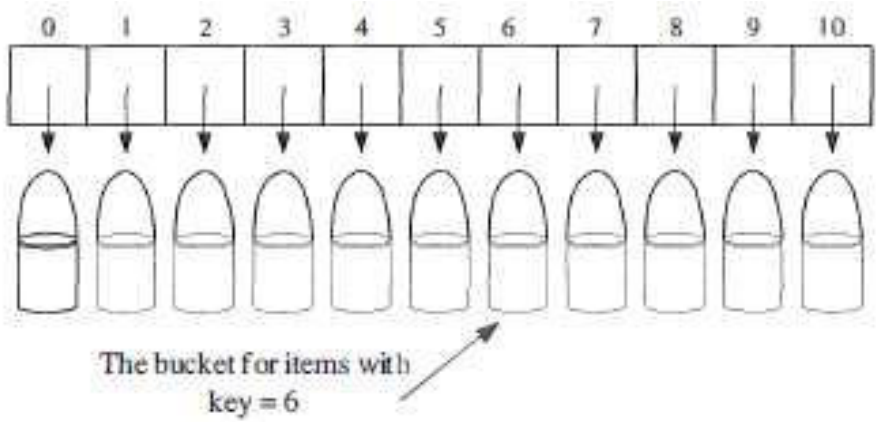
- A hash table for a given key type consists of
  - Array (called table) of size N-(Bucket Array)
  - Hash function h
- **When implementing a dictionary with a hash table, the goal is to store item (k, e) at index i = h(k)**

Bucket Arrays



- A bucket array for a hash table is an array A of size N, where each cell of A is thought of as a "bucket" (that is, a container of key-element pairs)
- Integer N defines the capacity of the array.
- **An element e with key k is simply inserted into the bucket A [k]** .
- Any bucket cells associated with keys not present in the dictionary are assumed to hold the special NO_SUCH _KEY object.

Bucket Arrays



Bucket Arrays

- **If keys are not unique**, then two different elements may be mapped to the same bucket in A .
- A ***collision*** has occurred.
- If each bucket of A can store only a single element, then we cannot associate more than one element with a single bucket
- ⇑ problem in the case of collisions.
- There are ways of dealing with collisions
- The best strategy is to try to avoid them in the first place

Bucket Arrays-Analysis

- **If keys are unique**, then collisions are not a concern, and searches, insertions, and removals in the hash table take
- **worst-case time O(1) .**
- **Uses O(N) space**

Dictionary Implementation using Hash Tables-Motivation

- The bucket array requires keys be **unique integers** in the **range [0 ,N - 1]** , which is often not the case
- There are two challenges in extending this framework to the more general setting
- **What can we do if we have at most 100 entries with integer keys but the keys are in range 0 to 1,000,000,000 ?**
- **What can we do if keys are not integers ?**

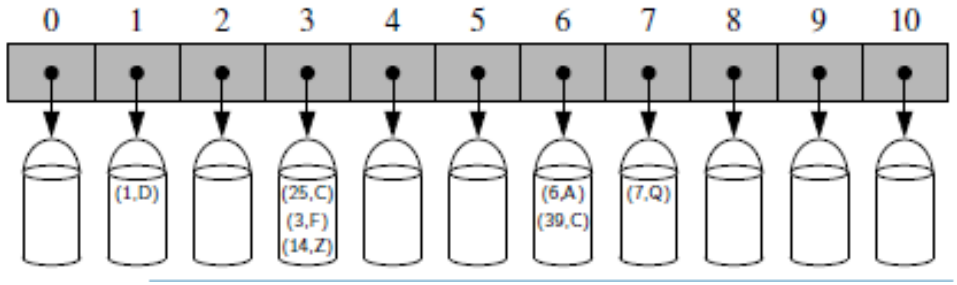
Bucket Arrays-Analysis

- What we can do???
- Define the hash table data structure to consist of a bucket array together with a "good" mapping from our keys to **1. integers,**
  2.in the range [0, N - 1]

Hash Functions

- A hash function h maps keys of a given type to integers in a fixed interval [0, N - 1]
- Now, bucket array method can be applied to arbitrary keys
- Example:
  $h(x) = x \bmod N$
  is a hash function for integer keys
- The integer h(x) is called **the hash value of key x.**

Bucket Arrays with a hash function



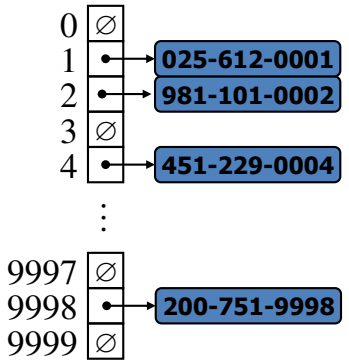
A bucket array of capacity 11 with items (1,D), (25,C), (3,F), (14,Z), (6,A), (39,C), and (7,Q), using a simple hash function

Hash Functions

- **Main Idea**
- Use the hash function value, h(k) , as an index to bucket array, A, instead of the key k (which is most likely inappropriate for use as a bucket array index).
- That is, store the item(k, e) in the bucket **A [h(k)]** .
- A hash function is "good" if it maps the keys in our dictionary so as to **minimize collisions** as much as possible.

Hash Tables-Example

- We design a hash table for a dictionary storing items (SSN, Name), where SSN (social security number) is a nine-digit positive integer
- Hash table uses an array of size $N = 10{,}000$ and the hash function $h(x)$ = last four digits of x



Evaluation of a hash function, h(k)

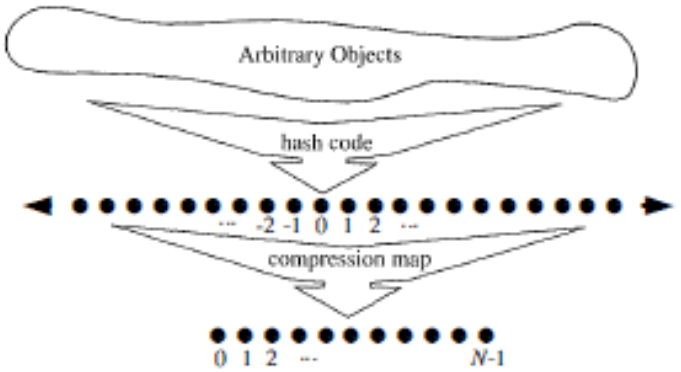
- A hash function ,h(k) , is usually specified as the composition of two functions:

  Hash code map:
  h1: keys → integers[mapping the key k to an integer]

  Compression map:
  h2: integers → [0, N - 1]

  [mapping the hash code to an integer within the range of indices of a bucket array]

Evaluation of a hash function



Evaluation of a hash function, h(k)

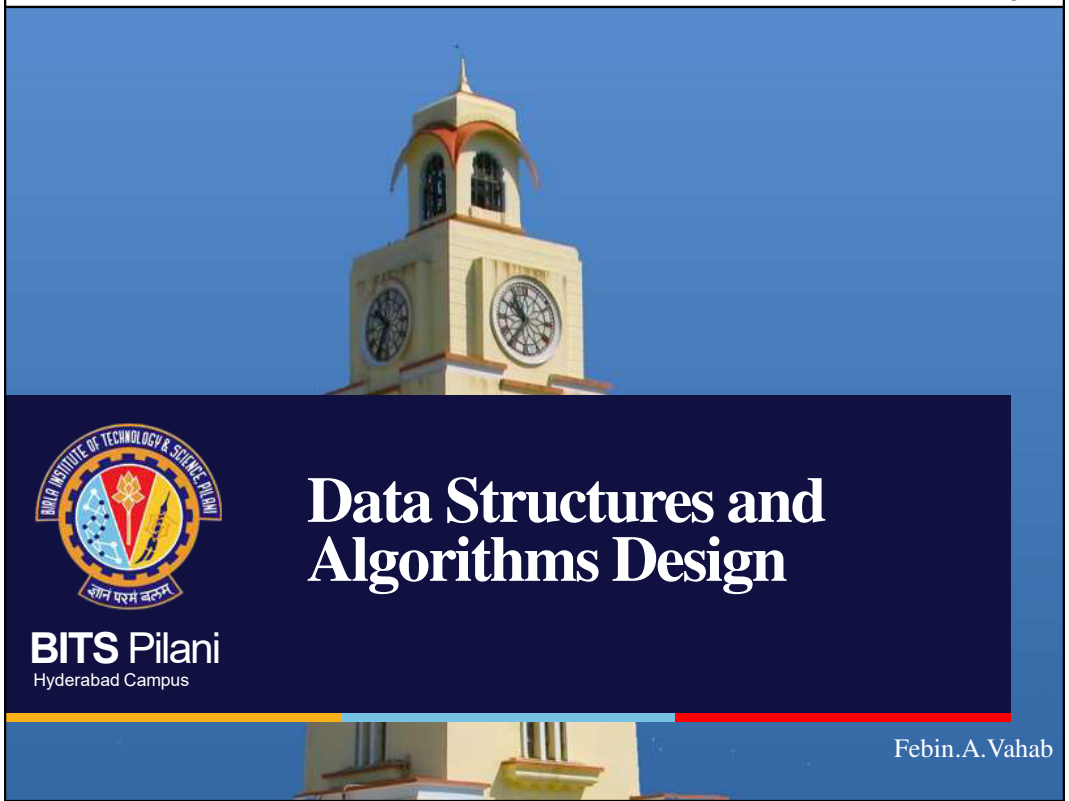
- The hash code map is applied first, and the compression map is applied next on the result, i.e., $\boldsymbol{h}(\boldsymbol{x}) = \boldsymbol{h}_2(\boldsymbol{h}_1(\boldsymbol{x}))$
- The goal of the hash function is to "disperse" the keys in an apparently random way.

Hash code in python

```
In [12]: print('Hash for 220 is:', hash(220))

         # hash for decimal
         print('Hash for 220.34 is:',hash(220.34))

         # hash for string
         print('Hash for Data is:', hash('Data'))

         Hash for 220 is: 220
         Hash for 220.34 is: 783986623132664028
         Hash for Data is: 2539907043859605924

In [13]: id(220.34)

Out[13]: 3207284319792

In [14]: id('Data')

Out[14]: 3207240693944
```

Compression Maps

- **Division:**
  - **Let $y=h_1(x)$//integer hash code for a key object k**
  - $h_2(y) = y \bmod N$
  - The size N of the hash table is usually chosen to be a prime
  - **The reason has to do with number theory and is beyond the scope of this course!!!**

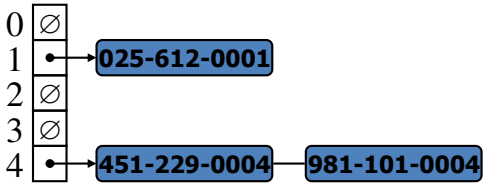


SESSION 5 -PLAN

Sessions(#)	List of Topic Title	Text/Ref Book/external resource
5	Methods for Collision Handling: Separate Chaining, Notion of Load Factor, Rehashing, Open Addressing [Linear; Quadratic Probing, Double Hash]	T1: 2.5

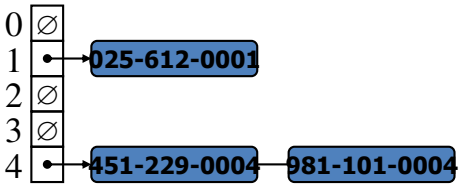
Collision- Handling Schemes

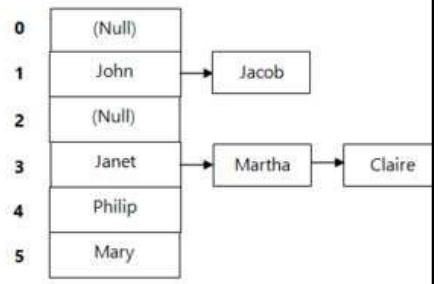
- Collisions occur when different elements are mapped to the same cell
- **Separate Chaining**: let each cell in the table point to a linked list of elements that map there



Collision- Handling Schemes

- Chaining is simple, but requires additional memory outside the table

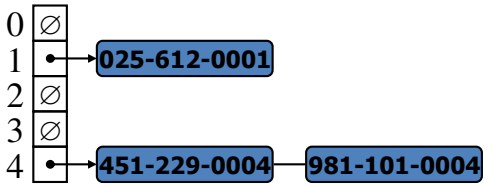




Separate Chaining

- A simple and efficient way for dealing with collisions is to have each bucket A [i] store a reference to a list that stores all the items that our hash function has mapped to the bucket A [i]
- Fundamental dictionary operations
- findElement (k) :

B ←A[h (k)]
if B is empty then
return NO_SUCH_K EY
else
{ search for the key k in the sequence for this bucket}
return B.findElement(k)



Separate Chaining

- Fundamental dictionary operations
- insertItem(k, e) :

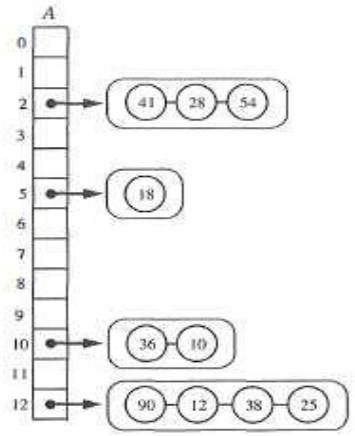
if A [h (k)] is empty then
Create a new initially empty, sequence-based dictionary B
A [h (k)] ←B
else
B ← A[h (k)]
B . insertItem(k, e)

Separate Chaining

- Fundamental dictionary operations
- removeElement (k) :

B ←A[h (k)]
if B is empty then
return NO_SUCH _K EY
else
return B. removeElement(k)

Separate Chaining

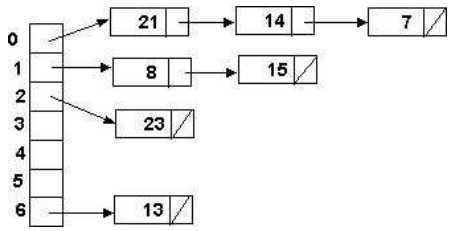


Example of a hash table of size 13, storing 10 integer keys, with collisions resolved by the chaining method. The compression map in this case is h (k) = k mod 1 3 .

Separate chaining

- **Example:** Load the keys **23, 13, 21, 14, 7, 8, and 15** , in this order, in a hash table of size **7** using separate chaining with the hash function: **h(key) = key % 7**

h(23) = 23 % 7 = 2
h(13) = 13 % 7 = 6
h(21) = 21 % 7 = 0
h(14) = 14 % 7 = 0 collision
h(7) = 7 % 7 = 0 collision
h(8) = 8 % 7 = 1
h(15) = 15 % 7 = 1 collision



Separate Chaining

- A good hash function will try to minimize collisions as much as possible, which will imply that most of our buckets are either empty or store just a single entry.
- Assume we use a good hash function to index the *n* entries of our map in a bucket array of capacity *N*, we expect each bucket to be of size *n/N(average)*
- This value is called the **load factor** of the hash table
- Should be bounded by a small constant, preferably below 1

Separate Chaining

- For a good hash function,the expected running time of operations findElement, insertItem, removeElement in a dictionary implemented using hash table which uses separate chaining to resolve collisions is O(n/N).
- Thus we can expect the standard dictionary operations to run in O(1)expected time provided we know that n is O(N)

Rehashing

- Mostly load factor ,0.75 is common
- Whenever we add elements we need to increase the size of our bucket array and change our compression map to match this new size, in order to keep the load factor below the specified constant.
- Moreover, we must then insert all the existing hash-table elements into the new bucket array using the new compression map. Such a size increase and hash table rebuild is called **rehashing**
- A good choice is to rehash into an array roughly double the size of the original array, choosing the size of the new array to be a prime number

Open Addressing

- **Open addressing**: the colliding item is placed in a different cell of the table
- This approach saves space because no auxiliary structures are employed, but it requires a bit more complexity to deal with collisions

Linear Probing

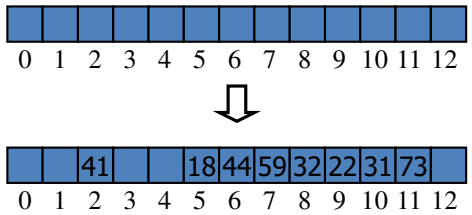
- Linear probing handles collisions by placing the colliding item in the next (circularly) available table cell
- Each table cell inspected is referred to as a "probe"
- Colliding items lump together, causing future collisions to cause a longer sequence of probes

Linear Probing

- In this method, if we try to insert an entry (k, v) into a bucket $A[i]$ that is already occupied, where $i = h(k)$, then we try next at $A[(i+1) \bmod N]$.
- This process will continue until we find an empty bucket that can accept the new entry.

Linear Probing

- **Example**: An insertion into a hash table using linear probing to resolve collisions.
  - h(x) = x mod 13
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



Linear Probing

- **Example**
  - We want to add the following (phone, address) entries to an addressBook with size 101:
  - addressBook.add("869-1264", "8-128");
  - addressBook.add("869-8132", "9-101");
  - addressBook.add("869-4294", "8-156");
  - addressBook.add("869-2072", "9-101");

The hash function is $h(k) = (k \% 10000) \% 101$

All of the above keys (phone numbers) map to index 52. By linear probing, all entries will be put to indices 52 - 55

Search with Linear Probing

- Consider a hash table A that uses linear probing
- **findElement(k)**
  - We start at cell h(k)
  - We probe consecutive locations until one of the following occurs
    - An item with key k is found, or
    - An empty cell is found, or
    - N cells have been unsuccessfully probed

Search with Linear Probing

Algorithm *findElement*(*k*)
  $i \leftarrow h(k)$
  $p \leftarrow 0$
  repeat
    $c \leftarrow A[i]$
    if $c = \emptyset$
      return *NO_SUCH_KEY*
    else if *c.key* () = *k*
      return *c.element*()
    else
      $i \leftarrow (i + 1) \bmod N$
    $p \leftarrow p + 1$
  until $p = N$
  return *NO_SUCH_KEY*

Updates with Linear Probing

- To handle insertions and deletions, we introduce special object, called AVAILABLE, which replaces deleted elements

removeElement(k)

- We search for an item with key k
- If such an item (k, e) is found, we replace it with the special item AVAILABLE and we return element e
- Else, we return NO_SUCH_KEY

Updates with Linear Probing

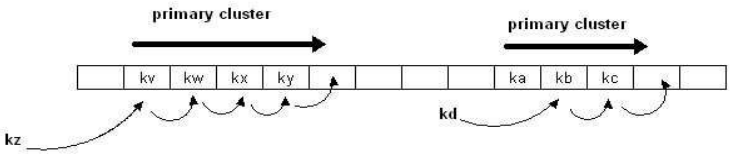
- **insertItem(*k*, *e*)**
  - We throw an exception if the table is full
  - We start at cell ***h*(*k*)**
  - We probe consecutive cells until one of the following occurs
    - A cell ***i*** is found that is either empty or stores ***AVAILABLE***, or
    - ***N*** cells have been unsuccessfully probed
  - We store item (***k, e***) in cell ***i***

Problem with Linear Probing

- Primary Clustering

Problem with Linear Probing

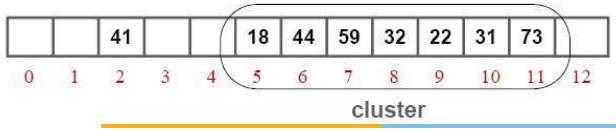
- Linear probing is subject to a primary clustering phenomenon.
- Elements tend to cluster around table locations that they originally hash to.
- Primary clusters can combine to form larger clusters. This leads to long probe sequences and hence deterioration in hash table efficiency.



Problem with Linear Probing

Example of a primary cluster: Insert keys: **18, 41, 22, 44, 59, 32, 31, 73**, in this order, in an originally empty hash table of size **13**, using the hash function **h(key) = key % 13** and **c(i) = i**:

h(18) = 5
h(41) = 2
h(22) = 9
h(44) = 5+1
h(59) = 7
h(32) = 6+1+1
h(31) = 5+1+1+1+1+1
h(73) = 8+1+1+1



Quadratic probing

- This open addressing strategy involves iteratively trying the buckets

  $A[(i + f(j)) \bmod N]$,

  for $j = 1, 2, ...$, where $f(j) = j^2$, until finding an empty bucket
- This strategy may not find an empty slot even when the array is not full.(If N is not chosen as a prime)
- This strategy may not find an empty slot, if the bucket array is at least half full.

Example

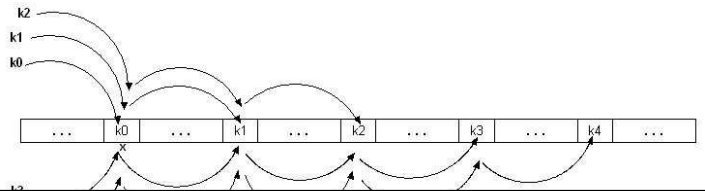
Insert the elements 76,40,48,5 and 55, N=7, h(k)=k mod N

- 76%7=6
- 40%7=5
- 48
  - h(k)=k mod N
  - =48%7=6 --collision
  - $h_1(k)=h(k) +i+ i^2$
  - = (6+1+1)mod 7 =1

0	1	2	3	4	5	6
	48			5	40	76

Quadratic probing -Secondary Clusters

- Quadratic probing is better than linear probing because it eliminates primary clustering.
- However, it may result in **secondary clustering**: if **h(k1) = h(k2)** the probing sequences for **k1** and **k2** are exactly the same. This sequence of locations is called a secondary cluster.
- Secondary clustering is less harmful than primary clustering because secondary clusters do not combine to form large clusters.
-



Linear Vs Quadratic probing

- An advantage of linear probing is that it can reach every location in the hash table.
- This property is important since it guarantees the success of the *insertItem* operation when the hash table is not full.
- Quadratic probing can only guarantee a successful *insertItem* operation when the hash table is at most half full.

Double Hashing

- Double hashing uses a secondary hash function h',
- If h maps some key k to a bucket $A[i]$, with $i = h(k)$, that is already occupied, then we iteratively try the bucket
  - **$A[(i + f(j)) \bmod N]$ next,**
  - **for $j = 1, 2, 3, \ldots$, where $f(j) = j*h'(k)$**
- The secondary hash function cannot have zero values
- The table size N must be a prime to allow probing of all the cells
- Choose a secondary hash function that will attempt to minimize clustering as much as possible

Double Hashing

- Common choice of compression map for the secondary hash function:

  $h_2(k) = q - k \bmod q$

  where
  - $q < N$
  - q is a prime
- The possible values for $h_2(k)$ are

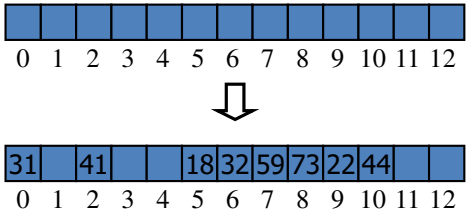
  $1, 2, \ldots, q$

Example of Double Hashing

- Consider a hash table storing integer keys that handles collision with double hashing
  - $N = 13$
  - $h(k) = k \bmod 13$
  - $h'(k) = 7 - k \bmod 7$
- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

Example of Double Hashing

k	$h(k)$	$h'(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		



Performance of Hashing

- In the worst case, searches, insertions and removals on a hash table take $O(n)$ time
- The worst case occurs when all the keys inserted into the dictionary collide
- The load factor $= n/N$ affects the performance of a hash table
- Assuming that the hash values are like random numbers, it can be shown that the expected number of probes for an insertion with open addressing is

  1 / (1 – ***load factor***) [*R2:Section 11.4,Theorem 11.6,11.8*]

Performance of Hashing

- The expected running time of all the dictionary ADT operations in a hash table is $O(1)$
- In practice, hashing is very fast provided the load factor is not close to 100%
- Applications of hash tables:
  - small databases
  - compilers
  - browser caches

Open Addressing

- **Advantages of Open addressing:**
  - All items are stored in the hash table itself. There is no need for another data structure.
  - Open addressing is more efficient storage-wise.
- **Disadvantages of Open Addressing:**
  - The keys of the objects to be hashed must be distinct.
  - Dependent on choosing a proper table size.
  - Requires the use of a three-state (Occupied, Empty, or Deleted) flag in each cell.

Open Addressing

- In general, primes give the best table sizes.
- With any open addressing method of collision resolution,

  as the table fills, there can be a severe degradation in the table performance.
- Load factors between 0.6 and 0.7 are common.
- Load factors > 0.7 are undesirable.
- The search time depends only on the load factor, *not* on the table size.

Separate chaining Vs Open Addressing

Separate Chaining has several advantages over open addressing:

- Collision resolution is simple and efficient.
- The hash table can hold more elements without the large performance deterioration of open addressing (The load factor can be 1 or greater)
- The performance of chaining declines much more slowly than open addressing.
- Deletion is easy - no special flag values are necessary.

Separate chaining Vs Open Addressing

- Table size need not be a prime number.
- The keys of the objects to be hashed need not be unique.

Separate chaining Vs Open Addressing

Disadvantages of Separate Chaining:

- It requires the implementation of a separate data structure for chains, and code to manage it.
- The main cost of chaining is the extra space required for the linked lists.

Exercises-1

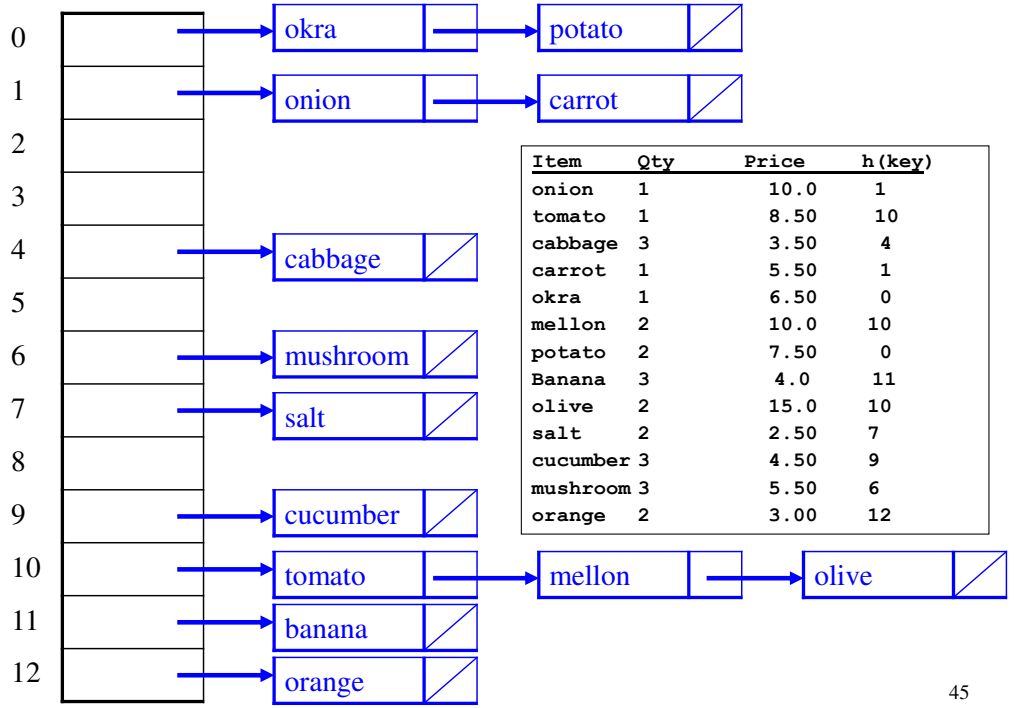
- Use the hash function **hash** to load the following commodity items into a hash table of size **13** using separate chaining:

onion	1	10.0
tomato	1	8.50
cabbage	3	3.50
carrot	1	5.50
okra	1	6.50
mellon	2	10.0
potato	2	7.50
Banana	3	4.00
olive	2	15.0
salt	2	2.50
cucumber	3	4.50
mushroom	3	5.50
orange	2	3.00

Exercises-I Solution

character	a	b	c	e	g	h	i	k	l	m	n	o	p	r	s	t	u	v
ASCII code	97	98	99	101	103	104	105	107	108	109	110	111	112	114	115	116	117	118

hash(onion) = (111 + 110 + 105 + 111 + 110) % 13 = 547 % 13 = 1
hash(salt) = (115 + 97 + 108 + 116) % 13 = 436 % 13 = 7
hash(orange) = (111 + 114 + 97 + 110 + 103 + 101)%13 = 636 %13 = 12



Item	Qty	Price	h(key)
onion	1	10.0	1
tomato	1	8.50	10
cabbage	3	3.50	4
carrot	1	5.50	1
okra	1	6.50	0
mellon	2	10.0	10
potato	2	7.50	0
Banana	3	4.0	11
olive	2	15.0	10
salt	2	2.50	7
cucumber	3	4.50	9
mushroom	3	5.50	6
orange	2	3.00	12

Exercises-II

Example:

Perform the operations given below, in the given order, on an initially empty hash table of size **13** using linear probing with **c(i) = i** and the hash function: **h(key) = key % 13**:

insert(18), insert(26), insert(35), insert(9), find(15), find(48), delete(35), delete(40), find(9), insert(64), insert(47), find(35)

- The required probe sequences are given by:

$$h_i(key) = (h(key) + i) \% 13 \qquad i = 0, 1, 2, \ldots, 12$$

Exercises-II

OPERATION	PROBE SEQUENCE	COMMENT
insert(18)	$h_0(18) = (18 \% 13) = 5$	SUCCESS
insert(26)	$h_0(26) = (26 \% 13) = 0$	SUCCESS
insert(35)	$h_0(35) = (35 \% 13) = 9$	SUCCESS
insert(9)	$h_0(9) = (9 \% 13) = 9$	COLLISION
	$h_1(9) = (9+1) \% 13 = 10$	SUCCESS
find(15)	$h_0(15) = (15 \% 13) = 2$	FAIL because location 2 has **Empty** status
find(48)	$h_0(48) = (48 \% 13) = 9$	COLLISION
	$h_1(48) = (9 + 1) \% 13 = 10$	COLLISION
	$h_2(48) = (9 + 2) \% 13 = 11$	FAIL because location 11 has **Empty** status
withdraw(35)	$h_0(35) = (35 \% 13) = 9$	SUCCESS because location 9 contains 35 and the status is **Occupied** The status is changed to **Deleted**; but the key 35 is not removed.
find(9)	$h_0(9) = (9 \% 13) = 9$	The search continues, location 9 does not contain 9; but its status is **Deleted**
	$h_1(9) = (9+1) \% 13 = 10$	SUCCESS
insert(64)	$h_0(64) = (64 \% 13) = 12$	SUCCESS
insert(47)	$h_0(47) = (47 \% 13) = 8$	SUCCESS
find(35)	$h_0(35) = (35 \% 13) = 9$	FAIL because location 9 contains 35 but its status is **Deleted**

Index	Status	Value
0	O	26
1	E	
2	E	
3	E	
4	E	
5	O	18
6	E	
7	E	
8	O	47
9	D	35
10	O	9
11	E	
12	O	64

Exercises-III

- Suppose you are given an array A[1 : n] stored in read-only memory from which you want to sample ***k*** elements **uniformly** at random *without replacement* (so all of the sampled elements are distinct). Show how to do this, in O(n) expected time and O(k) space using an ADT covered in the contact sessions, and **do not** assume that the elements of ***A*** are *integer-valued*.

Exercises-IV

- Given a **hash table** of size **7** and hash function ***h(x) =x mod 7***, show the final table after inserting the following elements in the table ***19, 26, 13, 48, 17*** for each of the cases
- i. When ***linear probing*** is used
- ii. When ***double hashing*** is used with a second function **g(x) = 5 – (x mod 5)**

Exercise-V

- Example: Load the keys **23, 13, 21, 14, 7, 8, and 15**, in this order, in a hash table of size **7** using quadratic probing and the hash function: **h(key) = key % 7**
- The required probe sequences are given by:

$$h_i(key) = (h(key) + i^2) \% 7 \quad i = 0, 1, 2, 3$$

Exercise-V-Solution

$h_0(23) = (23 \% 7) \% 7 = 2$
$h_0(13) = (13 \% 7) \% 7 = 6$
$h_0(21) = (21 \% 7) \% 7 = 0$
$h_0(14) = (14 \% 7) \% 7 = 0$ collision
$h_1(14) = (0 + 1^2) \% 7 = 1$
$h_0(7) = (7 \% 7) \% 7 = 0$ collision
$h_1(7) = (0 + 1^2) \% 7 = 1$ collision
$h_2(7) = (0 + 2^2) \% 7 = 4$
$h_0(8) = (8 \% 7) \% 7 = 1$ collision
$h_1(8) = (1 + 1^2) \% 7 = 2$ collision
$h_2(8) = (1 + 2^2) \% 7 = 5$
$h_0(15) = (15 \% 7) \% 7 = 1$ collision
$h_1(15) = (1 + 1^2) \% 7 = 2$ collision
$h_2(15) = (1 + 2^2) \% 7 = 5$ collision
$h_3(15) = (1 + 3^2) \% 7 = 3$

0	21
1	14
2	23
3	15
4	7
5	8
6	13

Exercise-VI

Load the keys **18, 26, 35, 9, 64, 47, 96, 36, and 70** in this order,in an empty hash table of size **13**

(a) using double hashing with the first hash function: **h(key) = key % 13** and the second hash function: **h_p(key) = 1 + key % 12**

(b) using double hashing with the first hash function: **h(key) = key % 13** and the second hash function: **h_p(key) = 7 - key % 7**

Show all computations.

Exercise-VI-Solution

$h_0(18) = (18\%13)\%13 = 5$
$h_0(26) = (26\%13)\%13 = 0$
$h_0(35) = (35\%13)\%13 = 9$
$h_0(9) = (9\%13)\%13 = 9$ collision
$h_p(9) = 1 + 9\%12 = 10$
$h_1(9) = (9 + 1*10)\%13 = 6$
$h_0(64) = (64\%13)\%13 = 12$
$h_0(47) = (47\%13)\%13 = 8$
$h_0(96) = (96\%13)\%13 = 5$ collision
$h_p(96) = 1 + 96\%12 = 1$
$h_1(96) = (5 + 1*1)\%13 = 6$ collision
$h_2(96) = (5 + 2*1)\%13 = 7$
$h_0(36) = (36\%13)\%13 = 10$
$h_0(70) = (70\%13)\%13 = 5$ collision
$h_p(70) = 1 + 70\%12 = 11$
$h_1(70) = (5 + 1*11)\%13 = 3$

hi(key) = [h(key) + i*hp(key)]% 13
h(key) = key % 13
hp(key) = 1 + key % 12

Exercise-VI-Solution

0	1	2	3	4	5	6	7	8	9	10	11	12
26			70		18	9	96	47	35	36		64

Data Structures and Algorithms Design

BITS Pilani
Hyderabad Campus

Febin.A.Vahab

CONTACT SESSION 6 -PLAN

Contact Sessions(#)	List of Topic Title	Text/Ref Book/external resource
6	Graphs - Terms and Definitions, Properties, Representations (Edge List, Adjacency list, Adjacency Matrix), Graph Traversals (Depth First and Breadth First Search)	T1: 6.1, 6.2, 6.3

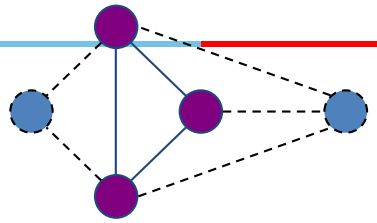
Depth-First Search

- Definitions
  - Subgraph
  - Connectivity
  - Spanning trees and forests
- Depth-first search
  - Algorithm
  - Example
  - Properties
  - Analysis
- Applications of DFS
  - Cycle finding
  - Path finding

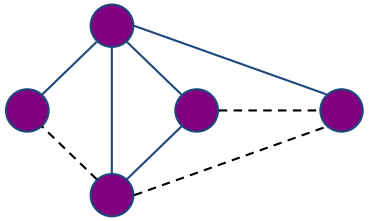
SUBGRAPHS

- Subgraphs
- A subgraph S of a graph G is a graph such that
  - The vertices of S are a subset of the vertices of G
  - The edges of S are a subset of the edges of G
- A spanning subgraph of G is a subgraph that contains all the vertices of G

SUBGRAPHS



Subgraph

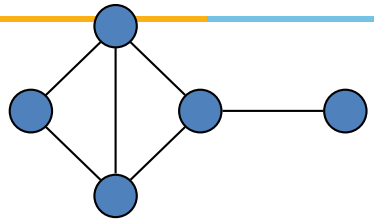


Spanning subgraph

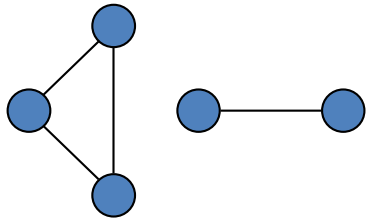
Connectivity

- A graph is **connected** if there is a path between every pair of vertices
- A connected component of a graph G is a maximal connected subgraph of G
- A directed graph G is **strongly connected** if:
  - For any two vertices u and v:
  - There is a directed path u → v, and
  - There is a directed path v → u

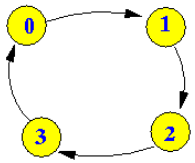
Connected graph



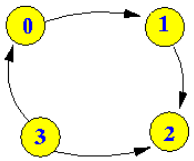
Connected graph



Non connected graph with two connected components



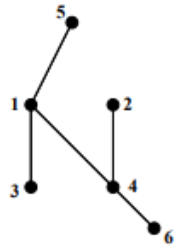
Strongly Connected

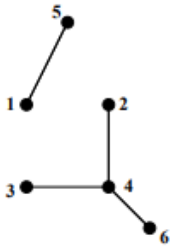


Not Strongly Connected

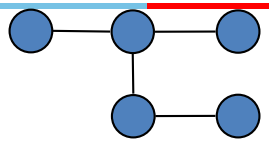
Trees and Forests

- A tree is a connected graph with no cycles.
- A forest is a graph with each connected component a tree

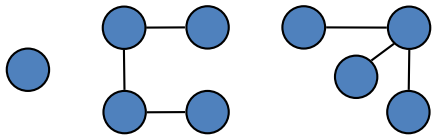




Trees and Forests



Tree

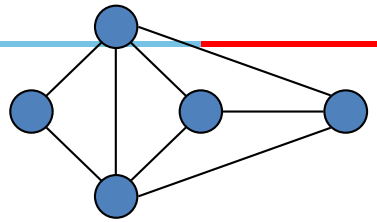


Forest

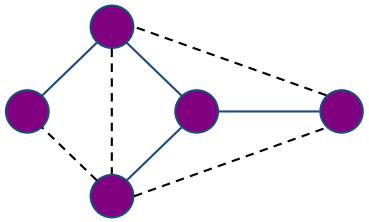
Spanning Trees

- A spanning tree of a connected graph is a spanning subgraph that is a tree:
- which includes all of the vertices of *G*, with minimum possible number of edges

Spanning Tree



Graph



Spanning tree

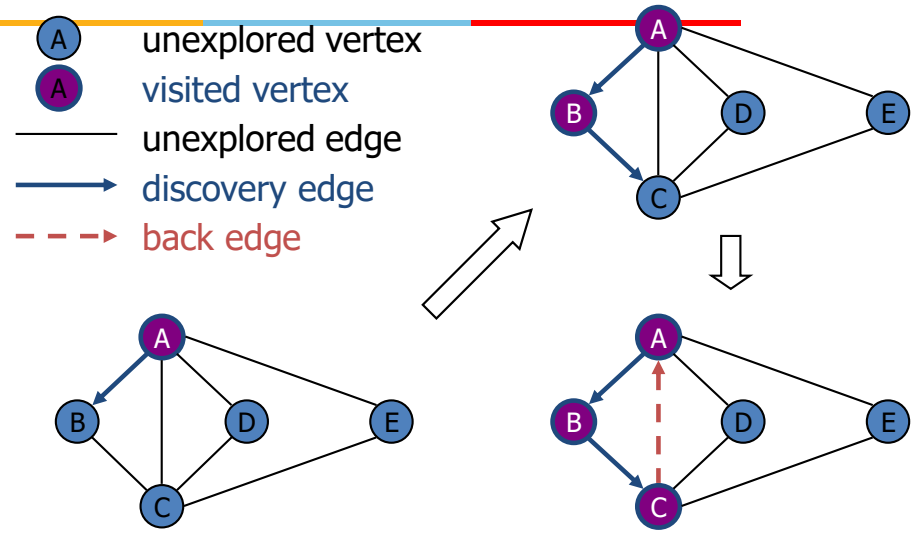
Subgraphs,trees-Example

- Perhaps the most talked about graph today is the Internet, which can be viewed as a graph whose vertices are computers and whose (undirected) edges are communication connections between pairs of computers on the Internet.
- The computers and the connections between them in a single domain, like http://www.bits-pilani.ac.in/ , form a subgraph of the Internet. If this subgraph is connected, then two users on computers in this domain can send e-mail to one another without having their information packets ever leave their domain.
- Suppose the edges of this subgraph form a spanning tree. This implies that, even if a single connection goes down (for example, because someone pulls a communication cable out of the back of a computer in this domain), then this subgraph will no longer be connected.

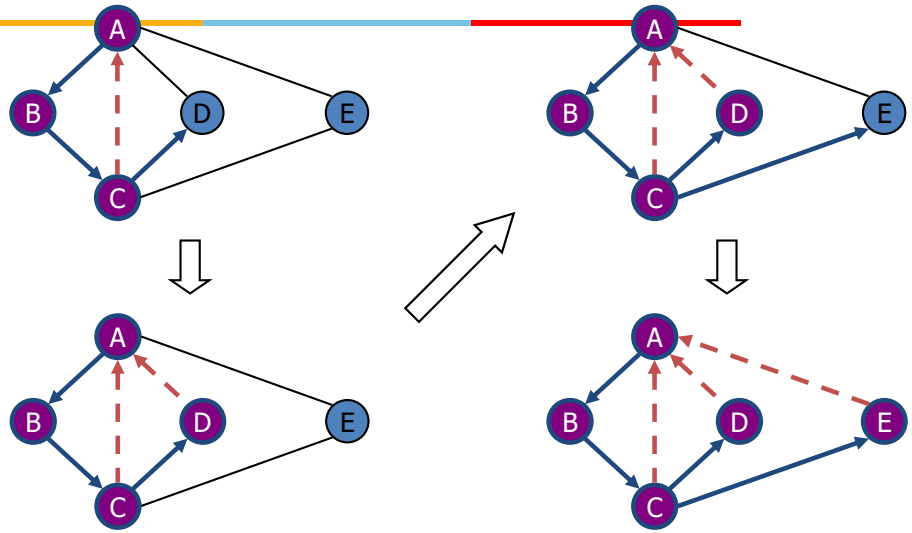
Depth-First Search

- Depth-first search (DFS) is a general technique for traversing a graph
- Search "deeper" in the graph whenever possible
- Explores edges out of the most recently discovered vertex that still has unexplored edges leaving it.
- Once all of v's edges have been explored, the search "backtracks" to explore edges leaving the vertex from which v was discovered.
- This process continues until we have discovered all the vertices that are reachable from the original source vertex.
- If any undiscovered vertices remain, then depth-first search selects one of them as a new source, and it repeats the search from that source.
- The algorithm repeats this ntire process until it has discovered every vertex

Depth-First Search



Depth-First Search



Depth-First Search

- The DFS algorithm is similar to a classic strategy for exploring a maze
  - We mark each intersection, corner and dead end (vertex) visited
  - We mark each corridor (edge) traversed
  - We keep track of the path back to the entrance (start vertex) by means of a rope (recursion stack)

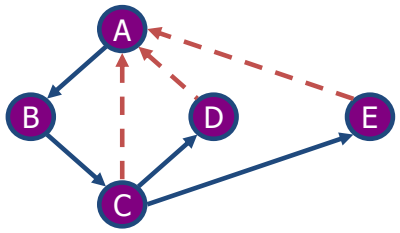
Depth-First Search-Properties

Property 1

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

Property 2

The discovery edges labeled by ***DFS*(*G*, *v*)** form a spanning tree of the connected component of ***v***



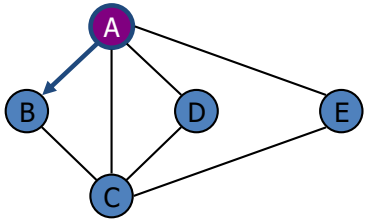
Depth-First Search

- The algorithm uses a mechanism for setting and getting "labels" of vertices and edges

Algorithm ***DFS*(*G*)**
  Input graph ***G***
  Output labeling of the edges of ***G*** as discovery edges and back edges
  for all ***u*** ∈ ***G.vertices*()**
    ***setLabel*(*u*, *UNEXPLORED*)**
  for all ***e*** ∈ ***G.edges*()**
    ***setLabel*(*e*, *UNEXPLORED*)**
  for all ***v*** ∈ ***G.vertices*()**
    if ***getLabel*(*v*) = *UNEXPLORED***
      ***DFS*(*G*, *v*)**

Depth-First Search

Algorithm ***DFS*(*G*, *v*)**
  Input graph ***G*** and a start vertex ***v*** of ***G***
  Output labeling of the edges of ***G*** in the connected component of ***v*** as discovery edges and back edges
  ***setLabel*(*v*, *VISITED*)**
  for all ***e*** ∈ ***G.incidentEdges*(*v*)**
    if ***getLabel*(*e*) = *UNEXPLORED***
      w ← ***G.opposite*(*v*,*e*)**
      if ***getLabel*(*w*) = *UNEXPLORED***
        ***setLabel*(*e*, *DISCOVERY*)**
        ***DFS*(*G*, *w*)**
      else
        ***setLabel*(*e*, *BACK*)**



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
  - Recall that $\sum_v \deg(v) = 2m$

Depth-First Search

- A DFS traversal of a graph G
  - Visits all the vertices and edges of G
  - Determines whether G is connected
  - Computes the connected components of G
  - Computes a spanning tree of G
  - Computing a cycle in G, or reporting that G has no cycles
  - Find and report a path between two given vertices

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

Path Finding

```
Algorithm pathDFS(G, v, z)
    setLabel(v, VISITED)
    S.push(v)
     if v = z
         return S.elements()
     for all e ∈ G.incidentEdges(v)
         if getLabel(e) = UNEXPLORED
           w ← opposite(v, e)
           if getLabel(w) = UNEXPLORED
                      setLabel(e, DISCOVERY)
                      S.push(e)
                      pathDFS(G, w, z)
                      S.pop()                    { e gets popped }
           else
                      setLabel(e, BACK)
    S.pop()                                      { v gets popped }
```


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

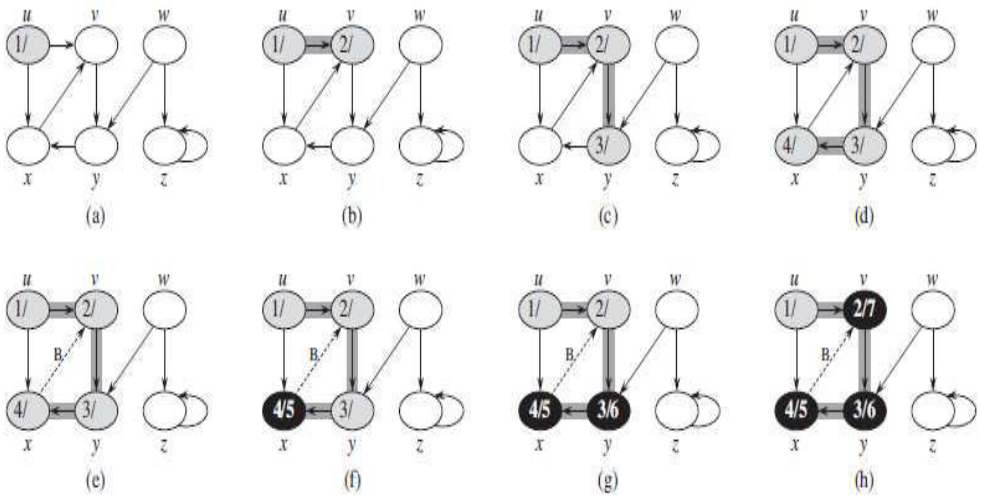
Cycle Finding

```
Algorithm cycleDFS(G, v, z)
    setLabel(v, VISITED)
    S.push(v)
     for all  e ∈ G.incidentEdges(v)
         if  getLabel(e) = UNEXPLORED
            w ← opposite(v,e)
            S.push(e)
            if getLabel(w) = UNEXPLORED
                        setLabel(e, DISCOVERY)
                       pathDFS(G, w, z)
                       S.pop()
            else
                       C ← new empty stack
                       repeat
                                  o ← S.pop()
                                  C.push(o)
                       until o = w
                       return C.elements()
     S.pop()
```

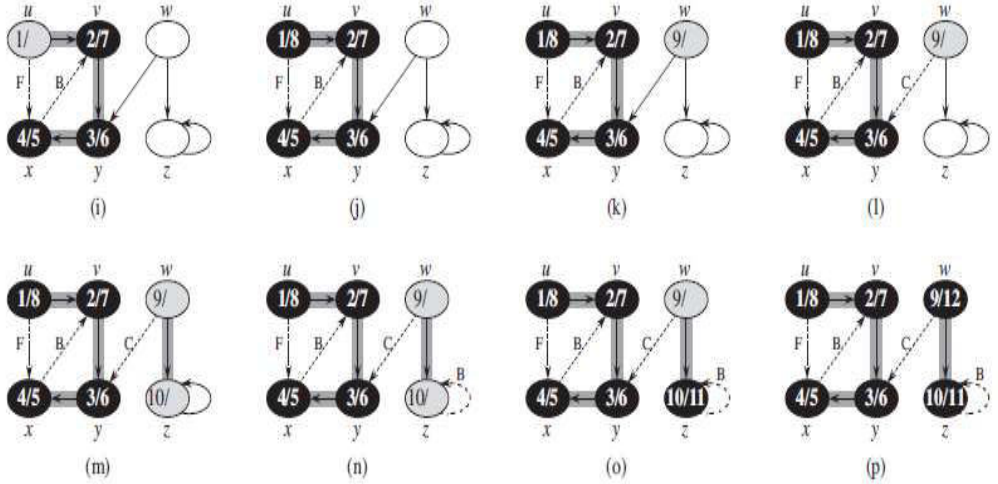

DFS:R2-Chapter 22

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)

DFS-VISIT(G, u)
1  time = time + 1                  // white vertex u has just been discovered
2  u.d = time
3  u.color = GRAY
4  for each v ∈ G.Adj[u]            // explore edge (u, v)
5      if v.color == WHITE
6          v.π = u
7          DFS-VISIT(G, v)
8  u.color = BLACK                  // blacken u; it is finished
9  time = time + 1
10 u.f = time
```

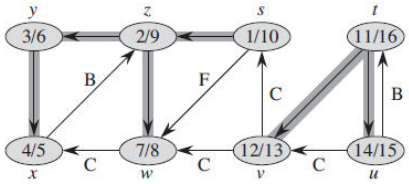

DFS:R2-Chapter 22

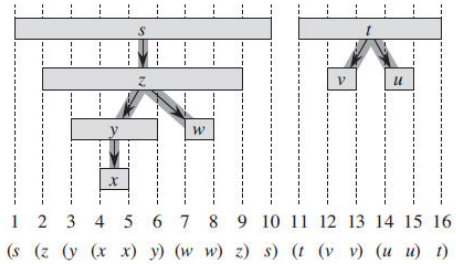


DFS:R2-Chapter 22



DFS:R2-Chapter 22 Paranthesis Structure





Connected components

How can DFS be used to find the connected components of a graph!

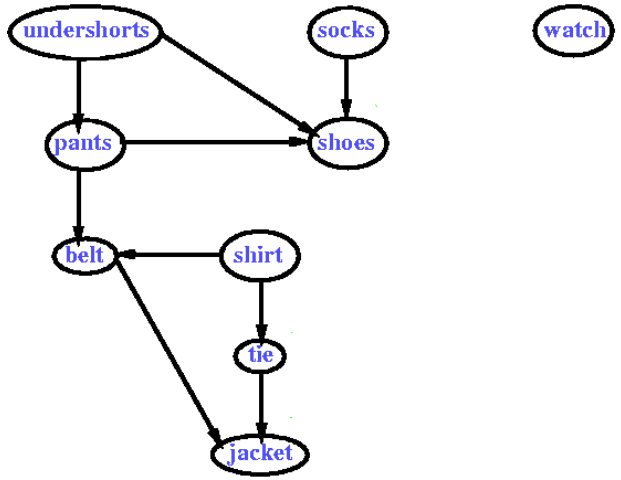
Can you implement it????
What will be the time complexity?

Connected components

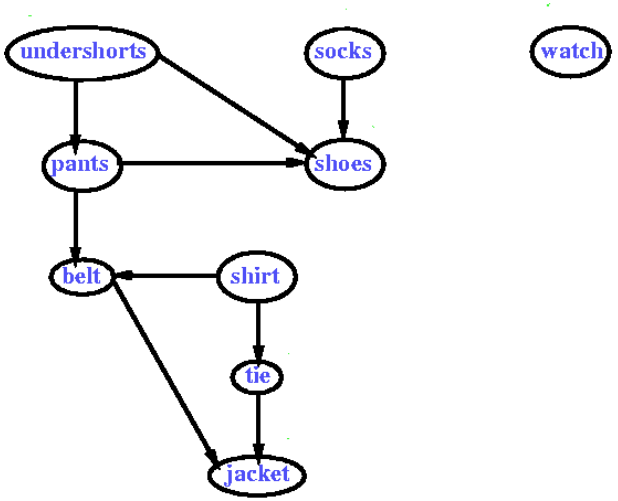
How can DFS be used to check whether a graph is connected or not?

Can you implement it????
What will be the time complexity?

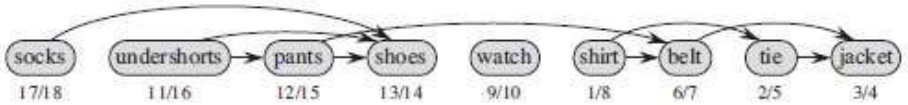
DFS for Toplogical Sort



DFS for Toplogical Sort



DFS for Toplogical Sort-Result



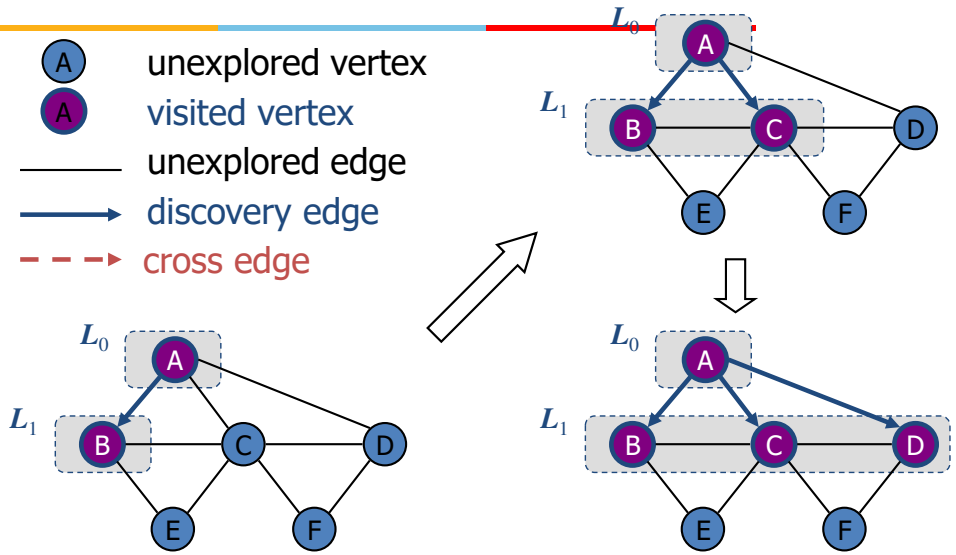
Breadth-first search

- Algorithm
- Example
- Properties
- Analysis
- Applications

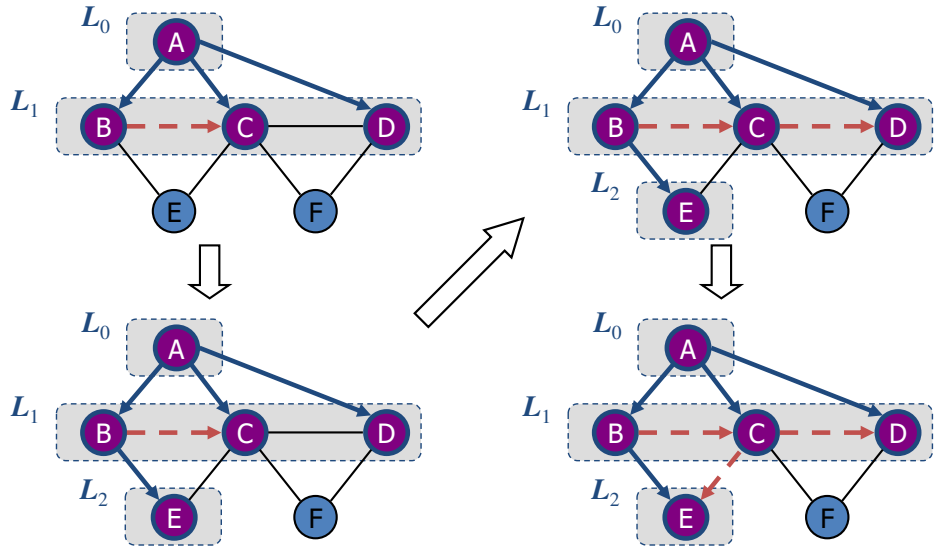
Breadth-first search

- Breadth-first search (BFS) is a general technique for traversing a graph
- A BFS traversal of a graph G
  - discovers all vertices at distance k from s before discovering any vertices at distance k + 1.
- For any vertex v reachable from vertex s, the simple path in the breadth-first tree from s to v corresponds to a "**shortest path**" from s to v in G, that is, a path containing the smallest number of edges.

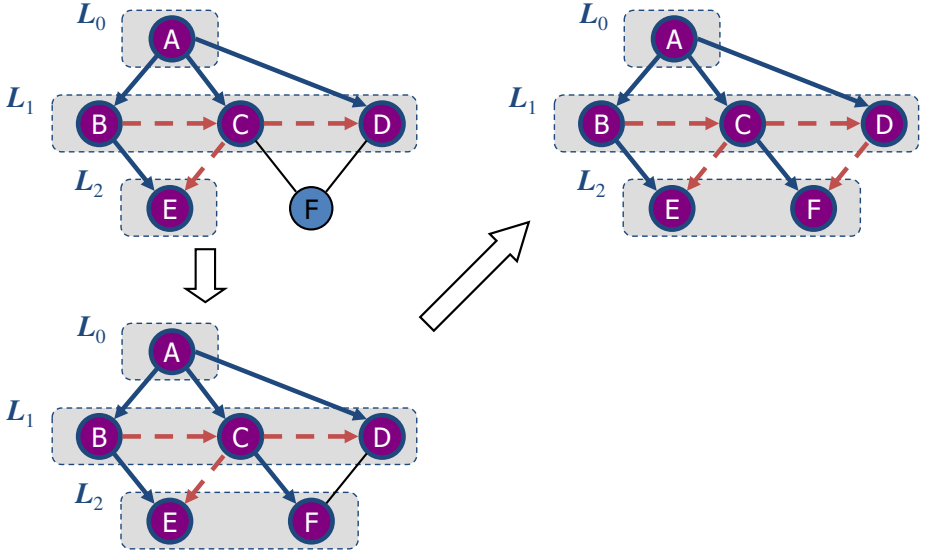
Breadth-first search



Breadth-first search



Breadth-first search



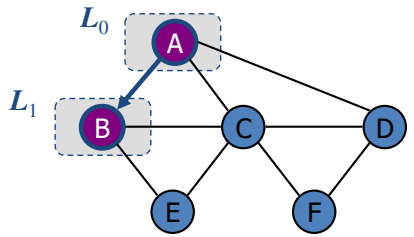
Breadth-first search

- The algorithm uses a mechanism for setting and getting "labels" of vertices and edges

Algorithm *BFS*(*G*)
  Input graph *G*
  Output labeling of the edges and partition of the vertices of *G*
  for all *u* ∈ *G.vertices*()
    setLabel(*u*, *UNEXPLORED*)
  for all *e* ∈ *G.edges*()
    setLabel(*e*, *UNEXPLORED*)
  for all *v* ∈ *G.vertices*()
    if *getLabel*(*v*) = *UNEXPLORED*
      BFS(*G*, *v*)

Breadth-first search – Algorithm *BFS*(*G*, *s*)

```
L0 ← new empty sequence
L0.insertLast(s)
setLabel(s, VISITED)
i ← 0
while ¬Li.isEmpty()
    Li+1 ← new empty sequence
    for all v ∈ Li.elements()
      for all e ∈ G.incidentEdges(v)
      if getLabel(e) = UNEXPLORED
                w ← opposite(v,e)
                if getLabel(w) = UNEXPLORED
                        setLabel(e, DISCOVERY)
                        setLabel(w, VISITED)
                        Li+1.insertLast(w)
                else
                        setLabel(e, CROSS)
i ← i +1
```



Breadth-first search – Algorithm *BFS*(*G*, *s*)

- We use auxiliary space to label edges, mark visited vertices, and store containers associated with levels.
- That is, the containers L0, L 1 , L2, and so on, store the nodes that are in level 0, level 1 , level 2, and so on.

Properties

Notation

G_s: connected component of s

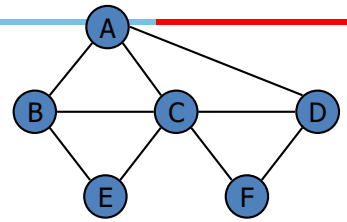
Property 1

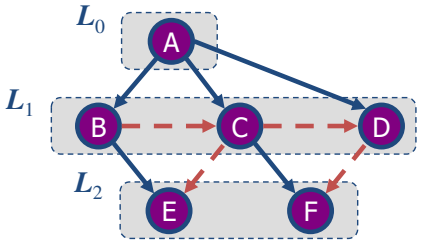
BFS(*G*, *s*) visits all the vertices and edges of G_s

Property 2

The discovery edges of a connected component labeled by ***BFS***(*G*, *s*) form a spanning tree T_s of G_s

Properties





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
  - Recall that $\sum_v \deg(v) = 2m$

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
  - Compute a spanning forest 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

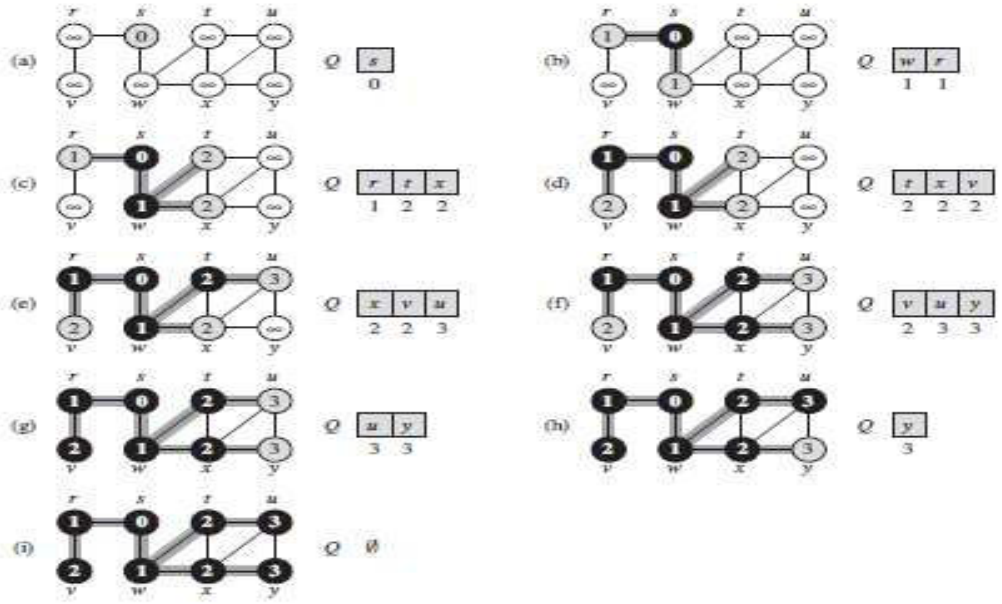
Facebook as Graph

- Traversal: go to 'Friends' to display all your friends (like G.Neighbors)
- BFS: the tabs are a queue - open all friends profiles in new tabs, then close current tab and go to the next one
- DFS: the history is a stack - open the first hot friend profile in the same window; when hitting a dead end, use back button

BFS-CLRS

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2      u.color = WHITE
 3      u.d = ∞
 4      u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)
12      for each v ∈ G.Adj[u]
13          if v.color == WHITE
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)
18      u.color = BLACK
```

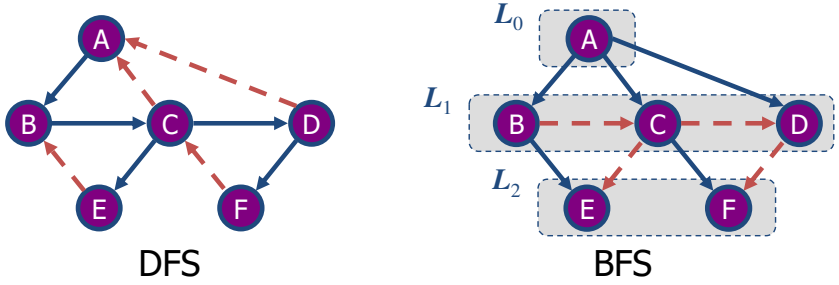
BFS-CLRS



DFS vs. BFS

Application	DFS	BFS
Spanning forest, connected components, paths, cycles	Y	Y
Shortest Paths		Y

DFS vs. BFS



DFS vs. BFS

Back edge (v,w)

- w is an ancestor of v in the tree of discovery edges

Cross edge (v,w)

- w is in the same level as v or in the next level in the tree of discovery edges

THANK YOU!

BITS Pilani
Hyderabad Campus

Depth-First Search

```
Algorithm DFS(G)
    Input graph G
    Output labeling of the edges of G
        as discovery edges and
        back edges
    for all u ∈ G.vertices()
        setLabel(u, UNEXPLORED)
    for all e ∈ G.edges()
        setLabel(e, UNEXPLORED)
    for all v ∈ G.vertices()
        if getLabel(v) = UNEXPLORED
            DFS(G, v)
```

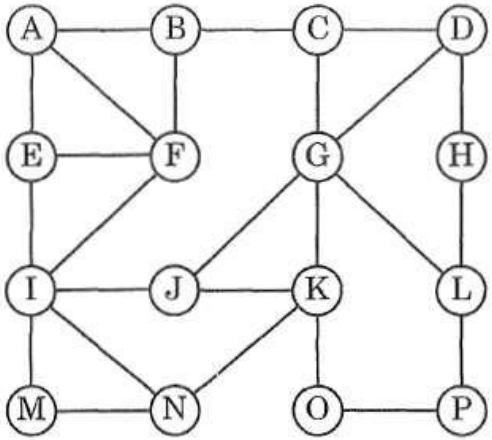
```
Algorithm DFS(G, v)
    Input graph G and a start vertex v of G
    Output labeling of the edges of G
        in the connected component of v
        as discovery edges and back edges
    setLabel(v, VISITED)
    for all e ∈ G.incidentEdges(v)
        if getLabel(e) = UNEXPLORED
            w ← opposite(v,e)
            if getLabel(w) = UNEXPLORED
                setLabel(e, DISCOVERY)
                DFS(G, w)
            else
                setLabel(e, BACK)
```

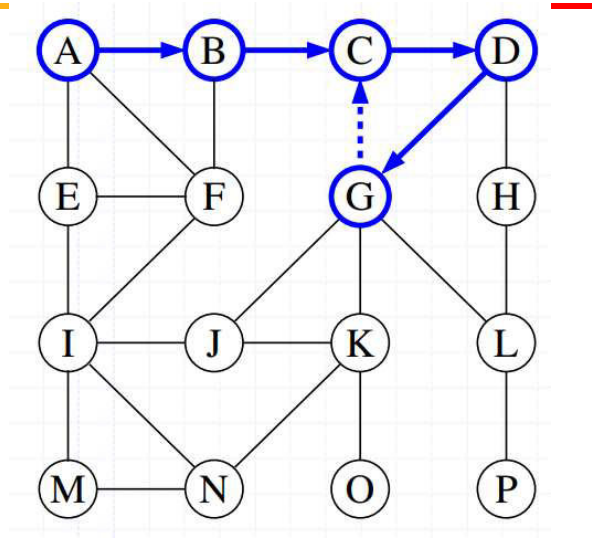

Data Structures and Algorithms Design

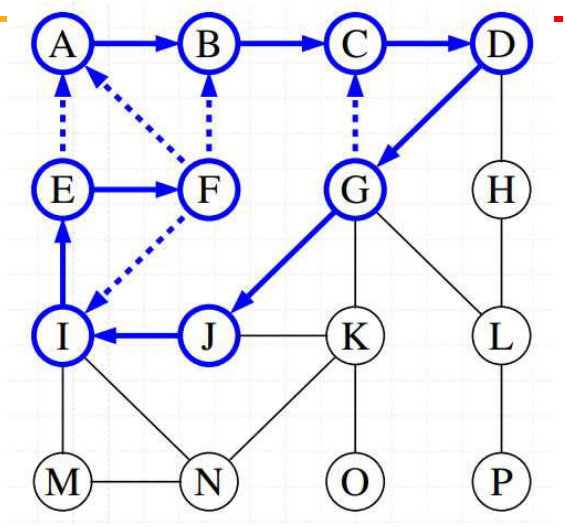
BITS Pilani
Hyderabad Campus

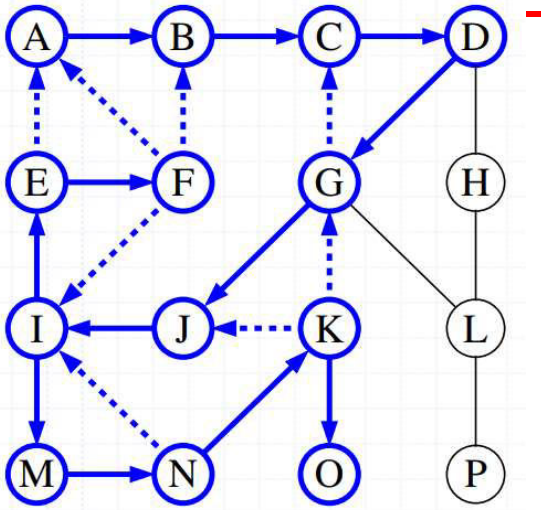
Febin.A.Vahab

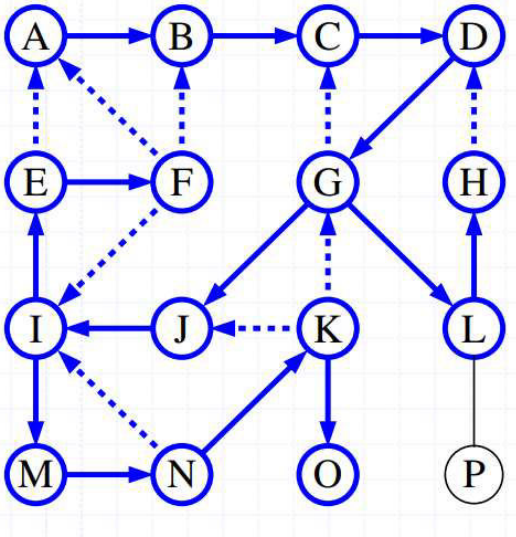
Example 1-Solved

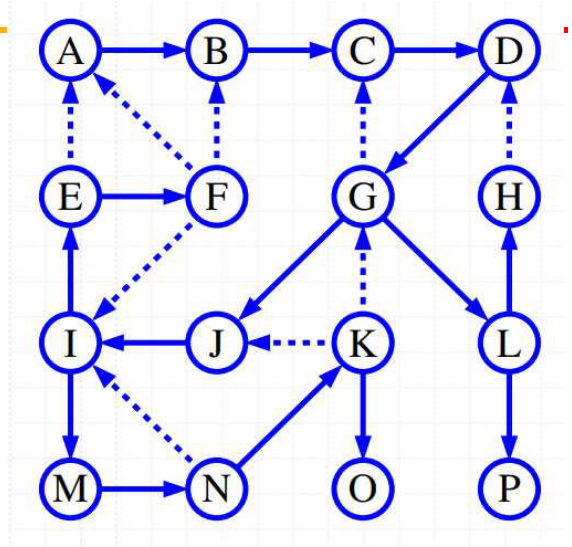



A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P

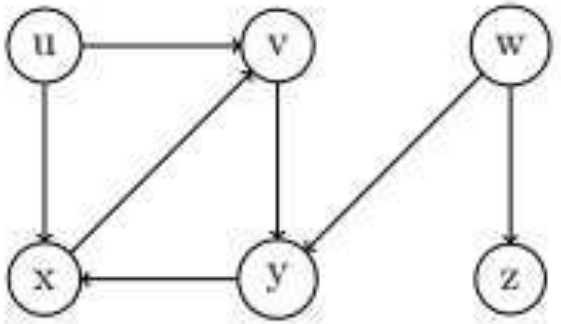

A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P


A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P

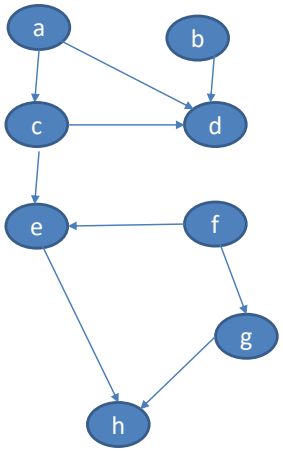

A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P

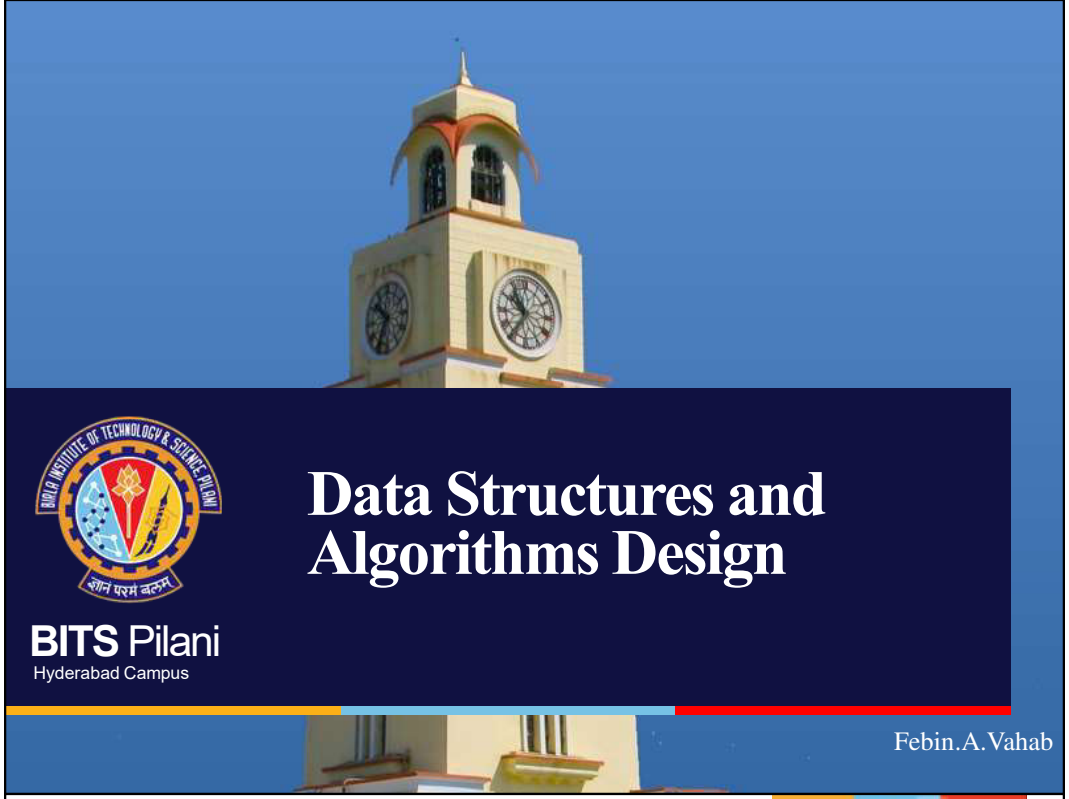


Example 2-Directed graph-Find the DFS tree-Discuss in Canvas



Example 3-Directed graph-Find the connected Components-Discuss in Canvas





Data Structures and Algorithms Design

BITS Pilani
Hyderabad Campus

Febin.A.Vahab

CONTACT SESSION 6 -PLAN

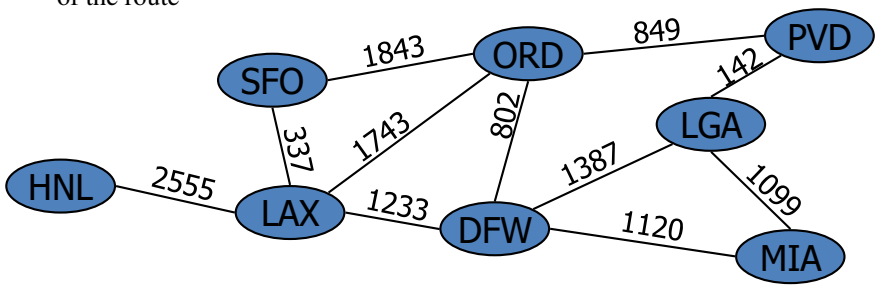
Contact Sessions(#)	List of Topic Title	Text/Ref Book/external resource
6	Graphs - Terms and Definitions, Properties, Representations (Edge List, Adjacency list, Adjacency Matrix), Graph Traversals (Depth First and Breadth First Search)	T1: 6.1, 6.2, 6.3

Graphs

- Graphs
  - Definition
  - Applications
  - Terminology
  - Properties
  - ADT
- Data structures for graphs
  - Edge list structure
  - Adjacency list structure
  - Adjacency matrix structure

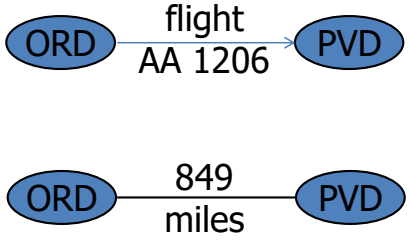
Graphs

- A graph is a pair (***V, E***), where
  - ***V*** is a set of nodes, called vertices
  - ***E*** is a collection of pairs of vertices, called edges
  - Vertices and edges are **positions** and store elements
- 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



Graphs

- 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
  - all the edges are directed
  - e.g., flight network
- Undirected graph
  - all the edges are undirected
  - e.g., route network

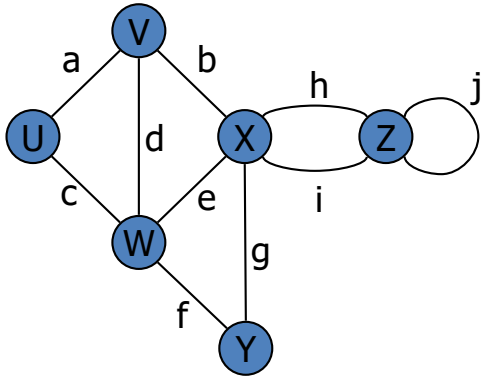


Graphs-Applications

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

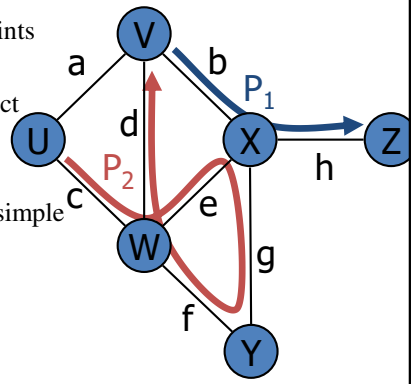
Graphs-Terminology

- End vertices (or endpoints) of an edge
  - U and V are the endpoints of a
- Edges incident on a vertex
  - a, d, and b are incident on V
- Adjacent vertices
  - U and V are adjacent
- Degree of a vertex
  - X has degree 5
- Parallel edges
  - h and i are parallel edges
- Self-loop
  - j is a self-loop

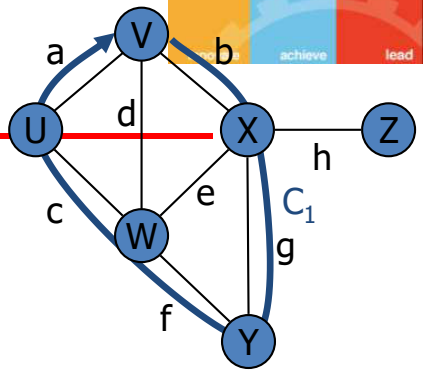


Graphs-Terminology

- 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

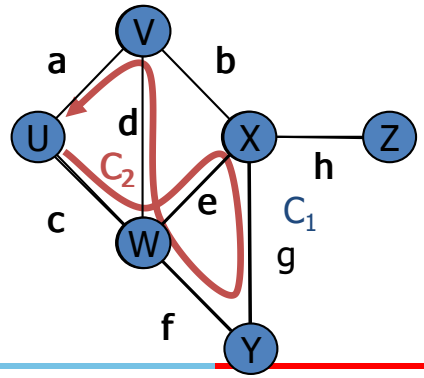


Graphs-Terminology



- 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



Graph-Example 1

We can visualize collaborations among the researchers of a certain discipline by constructing a graph whose vertices are associated with the researchers themselves, and whose edges connect pairs of vertices associated with researchers who have coauthored a paper or book.

Is it a directed or undirected graph?

Such edges are **undirected** because coauthorship is a symmetric relation; that is, if A has coauthored something with B, then B necessarily has coauthored something with A .

Graph-Example 2

- We can associate with an object-oriented program a graph whose vertices represent the classes defined in the program, and whose edges indicate inheritance between classes. There is an edge from a vertex v to a vertex u if the class for v extends the class for u.
- Is it a directed or undirected graph?
- Such edges are directed because the inheritance relation only goes in one direction (that is, it is asymmetric).

Graph-Example 3

- A city map can be modelled by a graph whose vertices are intersections or dead ends, and whose edges are stretches of streets without intersections.
- Directed or undirected?
- This graph has **both undirected edges**, which correspond to stretches of two-way streets, **and directed edges**, which correspond to stretches of one-way streets. Thus, a graph modelling a city map is a mixed graph

Graph-Example 4

- Physical examples of graphs are present in the electrical wiring and plumbing networks of a building.
- Such networks can be modelled as graphs, where each connector, fixture, or outlet is viewed as a vertex, and each uninterrupted stretch of wire or pipe is viewed as an edge.
- Such graphs are actually components of much larger graphs, namely the local power and water distribution networks.
- Depending on the specific aspects of these graphs that we are interested in, we may consider their edges as undirected or directed, for, in principle, water can flow in a pipe and current can flow in a wire in either direction.

Graph-Example 5-Path and Cycle

Given a graph G representing a city map, we can model a couple driving from their home to dinner at a recommended restaurant as traversing a path though G.

If they know the way, and don 't accidentally go through the same intersection twice, then they traverse a simple path in G.

Likewise, we can model the entire trip the couple takes, from their home to the restaurant and back, as a cycle.

If they go home from the restaurant in a completely different way than how they went, not even going through the same intersection twice, then their entire round trip is a simple cycle.

Finally, if they travel along one-way streets for their entire trip, then we can model their night out as a directed cycle.

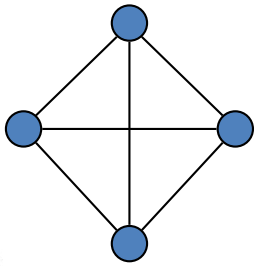
Graphs-Properties

Property 1

If G is a graph with m edges, then

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

Proof: each edge is counted twice



Property 2

Let G be a simple graph with n vertices and m edges.

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

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

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

Property 3

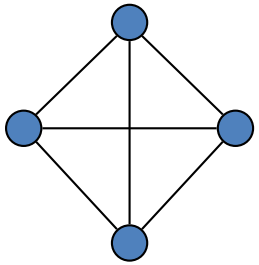
If G is a directed graph with m edges, then

$\sum_{v \in G} indeg(v) = \sum_{v \in G} outdeg(v) = m$

Graphs-Properties

Example

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



Graphs-ADT

- A graph is a positional container of elements that are stored at the graph's vertices and edges
- Vertices and edges
  - are positions
  - store elements
- **Accessor methods** Complexity
  - **incidentEdges(v)**:Return an iterator of the edges incident upon v
  - endVertices(e):Return an array of size 2 storing the end vertices of e.
  - degree(v):
  - adjacentVertices(v):Return an iterator of the vertices adjacent to v.
  - opposite(v, e):Return the endpoint of edge e distinct from v.
  - **areAdjacent(v, w):**Return whether vertices v and w are adjacent

Graphs-ADT

Methods Dealing with Directed Edges

- directed Edges(): Return an iterator of all directed edges.
- undirected Edges(): Return an iterator of all undirected edges.
- destination(e): Return the destination of the directed edge e.
- origin (e): Return the origin of the directed edge e.
- isDirected(e): Return true if and only if the edge e is directed.

Graphs-ADT

Update methods

- **insertVertex(o):**Insert and return a new vertex storing the object o
- **insertEdge(v, w, o):**Insert and return an undirected edge between vertices v and w, storing the object o.
- insertDirectedEdge(v, w, o)
- **removeVertex(v)**:Remove vertex v and all its incident edges.
- **removeEdge(e)**
- Generic methods
  - numVertices()
  - numEdges()
  - vertices():Return an iterator of the vertices of G.
  - edges()

Graphs-ADT

Also supports

- size()
- isEmpty ( )
- elements ( )
- positions()
- replaceElement(p, o)
- swapElements (p , q)

where p and q denote positions, and o denotes an object (that is, an element)

Data Structure for Graphs

- Edge list structure
- Adjacency list structure
- Adjacency matrix

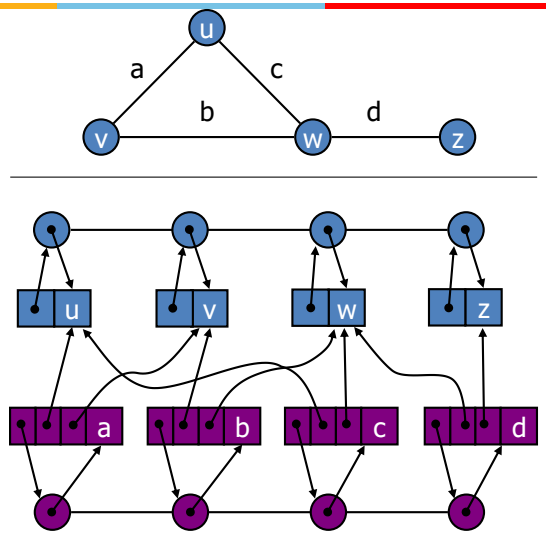
Edge List Structure

- Vertex object
  - **Element, o**
- Edge object
  - **element**
  - **origin vertex object**
  - **destination vertex object**
- Vertex sequence
  - sequence of vertex objects
- Edge sequence
  - sequence of edge objects

Time Complexity

Methods	Edge List-Time Complexity
incidentEdges(v)	O(m)
areAdjacent(v, w)	O(m)
insertVertex(o)	O(1)
insertEdge(v, w, o)	O(1)
removeVertex(v)	O(m)
removeEdge(e)	O(1)

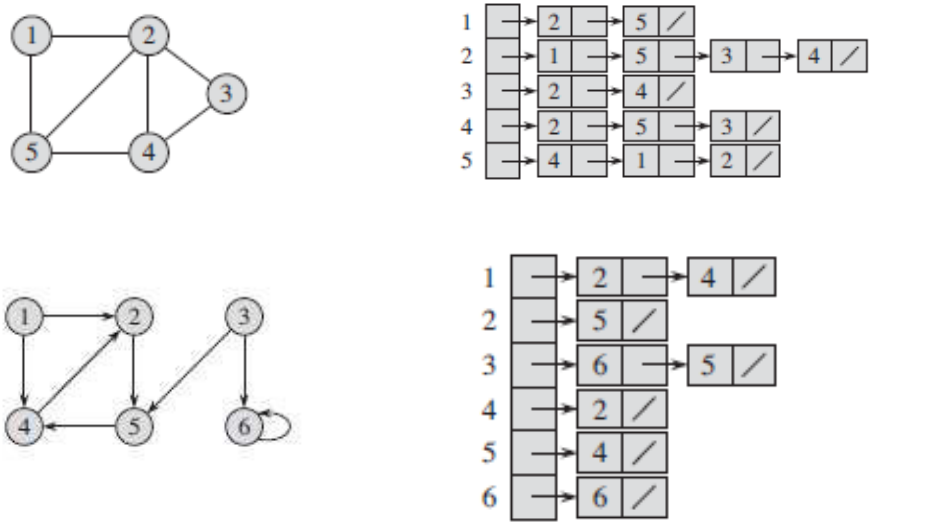
Edge List Structure



Adjacency List Structure

- Extends edge list structure
- Add extra information that supports direct access to the incident edges (and thus to the adjacent vertices) of each vertex
- For each vertex v, store a reference to a list of the vertices adjacent to it.

Adjacency List



Time Complexity

	Edge List	Adjacency List
incidentEdges(v)	m	deg(v)
areAdjacent(v, w)	m	min(deg(v),deg(w))
insertVertex(o)	1	1
insertEdge(v, w, o)	1	1
removeVertex(v)	m	deg(v)
removeEdge(e)	1	1

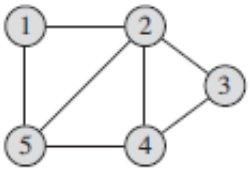
Adjacency Matrix Structure

- The ***adjacency-matrix representation*** of a graph G =(V,E), the vertices are numbered 1,2,…,|V| in some arbitrary manner. Then the adjacency-matrix representation of a graph G consists of a |V| X |V| matrix

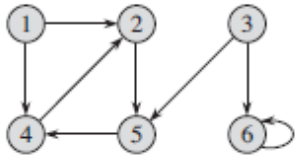
$$A = (a_{ij}) \text{ such that}$$

$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

Adjacency Matrix



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0



	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Asymptotic Performance

	Edge List	Adjacency List	Adjacency Matrix
incidentEdges(v)	m	deg(v)	n
areAdjacent(v, w)	m	min(deg(v),deg(w))	1
insertVertex(o)	1	1	n^2
insertEdge(v, w, o)	1	1	1
removeVertex(v)	m	deg(v)	n^2
removeEdge(e)	1	1	1

Graph ADT

Data Structures and Algorithms Design

BITS Pilani
Hyderabad Campus

Febin.A.Vahab

SESSION 6 -PLAN

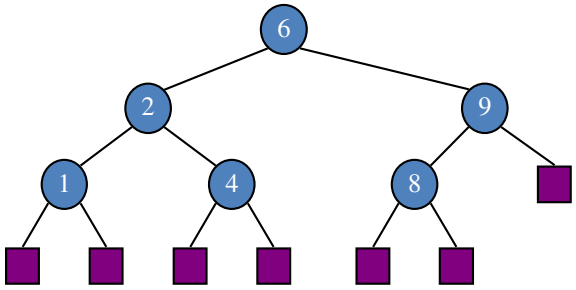
Online Sessions(#)	List of Topic Title	Text/Ref Book/external resource
6	Binary Search Tree - Motivation with the task of Searching and Binary Search Algorithm, Properties of BST, Searching an element in BST, Insertion and Removal of Elements,	T1: 3.1

Binary Search Tree

- A binary search tree is a binary tree storing keys (or key-element pairs) at its internal nodes and satisfying the following property
  - Let u, v, and w be three nodes such that u is in the left subtree of v and w is in the right subtree of v. We have $key(u) \leq key(v) \leq key(w)$

Binary Search Tree

- An inorder traversal of a binary search trees visits the keys in increasing order

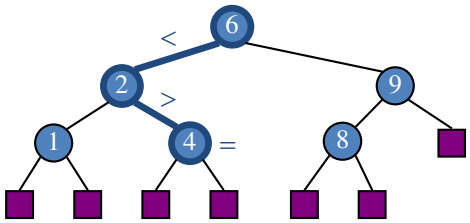


Binary Search Tree- Search

- To search for a key ***k***, we trace a downward path starting at the root
- The next node visited depends on the outcome of the comparison of ***k*** with the key of the current node
- If we reach a leaf, the key is not found and we return NO_SUCH_KEY
- Example: findElement(4)

- External nodes do not store items

Binary Search Tree- Search



Binary Search Tree- Search

Algorithm *findElement*(*k*, *v*)

- Input: A search key k, and a node v of a binary search tree T
- Output: A node w of the subtree T (v) of T rooted at v, such that either w is an internal node storing key k or w is the external node where an item with key k would belong if it existed

```
if T.isExternal (v)
    return NO_SUCH_KEY
 if k < key(v)
    return findElement(k, T.leftChild(v))
 else if k = key(v)
    return element(v)
 else { k > key(v) }
    return findElement(k, T.rightChild(v))
```

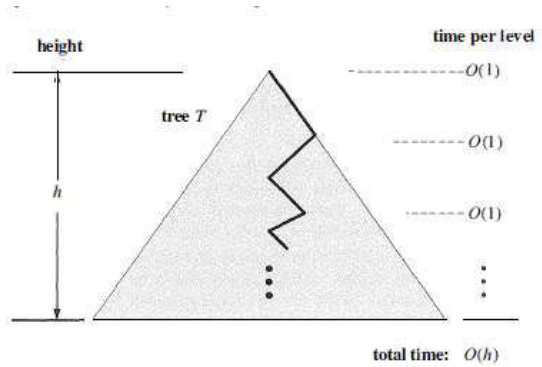

Analysis of Binary Tree Searching

- The binary tree search algorithm executes a constant number of primitive operations for each node it traverses in the tree.
- Each new step in the traversal is made on a child of the previous node.
- That is, the binary tree search algorithm is performed on the nodes of a path of T that starts from the root and goes down one level at a time.
- Thus, the number of such nodes is bounded by h + 1, where h is the height of T.

Analysis of Binary Tree Searching

- In other words, since we spend O(1) time per node encountered in the search, method findElement (or any other standard search operation) runs in O(h) time, where h is the height of the binary search tree T used to implement the dictionary D.
- ie. The running time of searching in a binary search tree T is proportional to the height of T . The height of a tree with n nodes can be **O(log n)**

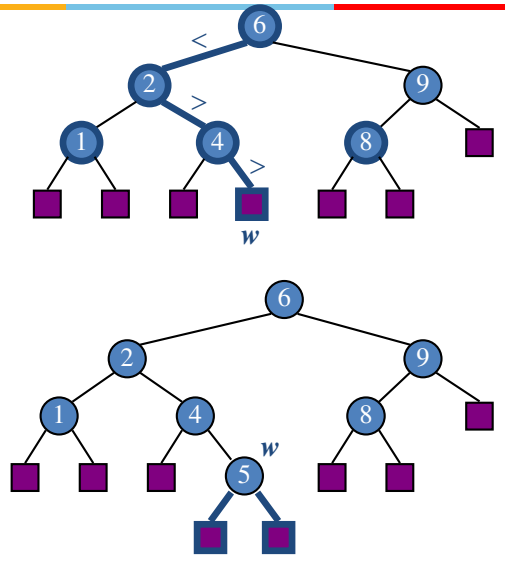
Analysis of Binary Tree Searching



Binary Search Tree- Insertion

- To perform operation insertItem(k, o), we search for key k
- Assume k is not already in the tree, and let w be the leaf reached by the search
- We insert k at node w and expand w into an internal node
- Example: insert 5

Binary Search Tree- Insertion

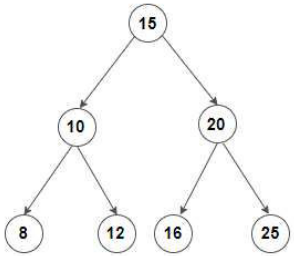


In-order Successor and Predecessor

- In a Binary Search Tree, the successor of a given key is the smallest number which is larger than the key.
- In the same way, a predecessor is the largest number which is smaller than the key.
- If X has two children then its in-order predecessor is the maximum value in its left subtree and its in-order successor the minimum value in its right subtree.

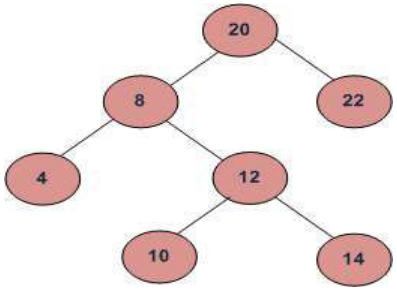
Predecessor

- Inorder predecessor of 8 doesn't exist
- Inorder predecessor of 20 is 16
- Inorder predecessor of 12 is 10



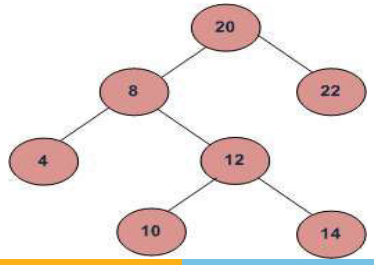
Predecessor

- Inorder predecessor of a node is a node with maximum value in its left subtree. i.e left subtree's right most child.
- If left subtree doesn't exists, then predecessor is one of the ancestors. Travel up using the parent pointer until you see a node which is right child of it's parent. The parent of such a node is the predecessor.



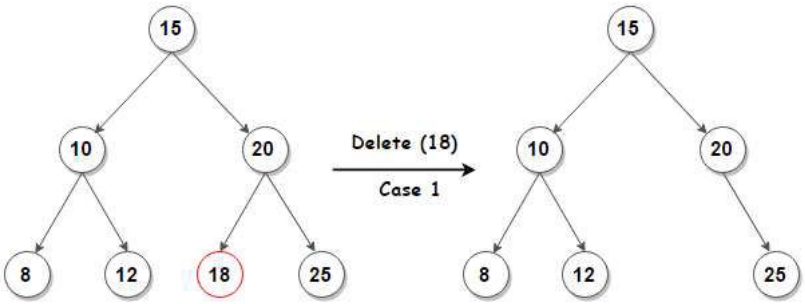
Successor

- If right subtree of node is not NULL, then succ lies in right subtree.Go to right subtree and return the node with minimum key value in right subtree.i.e **right subtree's left most child**.
- If right subtree of node is NULL, then succ is one of the ancestors. Travel up using the parent pointer until you see a node which is left child of it's parent. The parent of such a node is the succ.



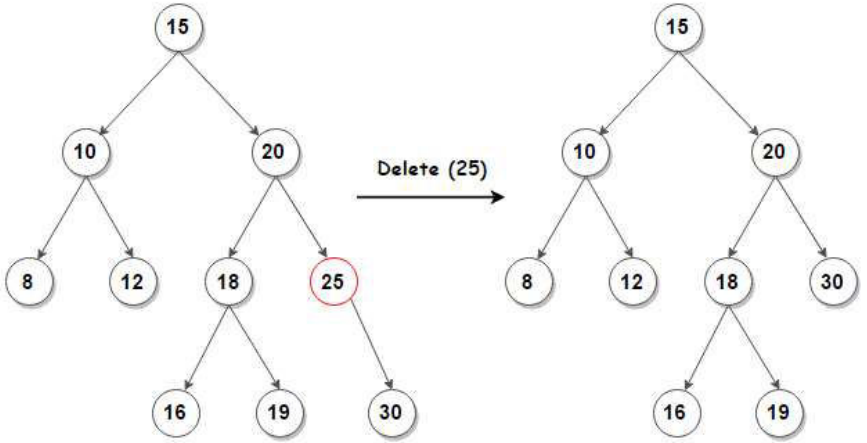
Binary Search Tree-Deletion

- Deleting a node with no Children

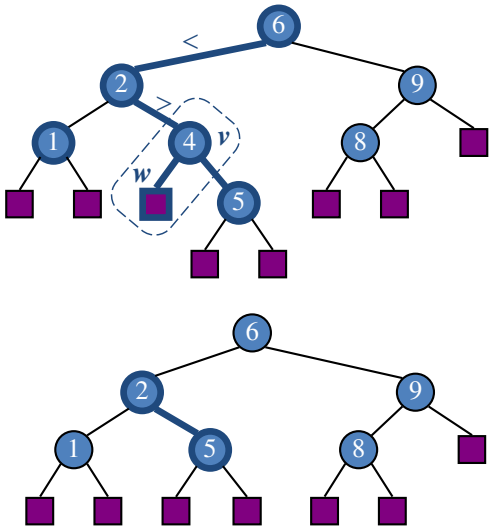


Binary Search Tree-Deletion

- **Deleting a node with 1 child**: Remove the node and replace it with its child

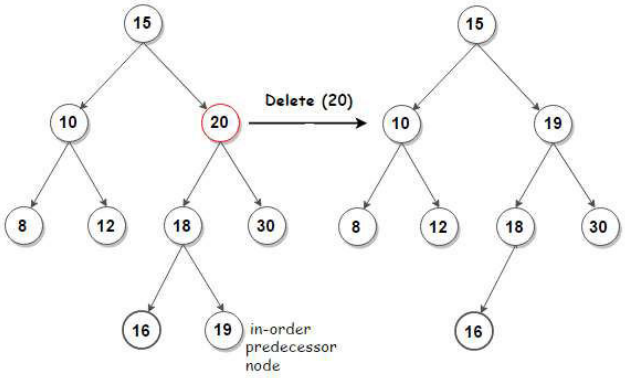


Binary Search Tree-Deletion



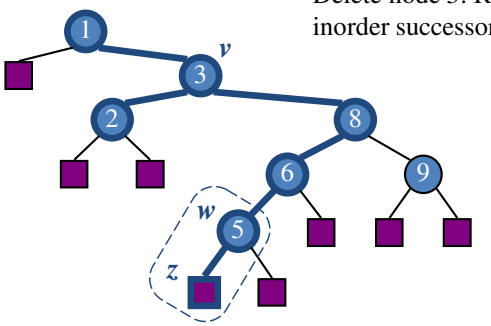
Binary Search Tree-Deletion

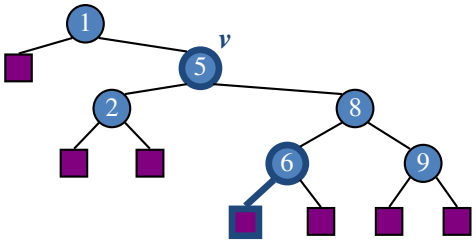
- Deleting a node with 2 children: Replace the node with its inorder successor(Predecessor)



Binary Search Tree-Deletion

Delete node 3: Replace the node with its inorder successor ,node 5

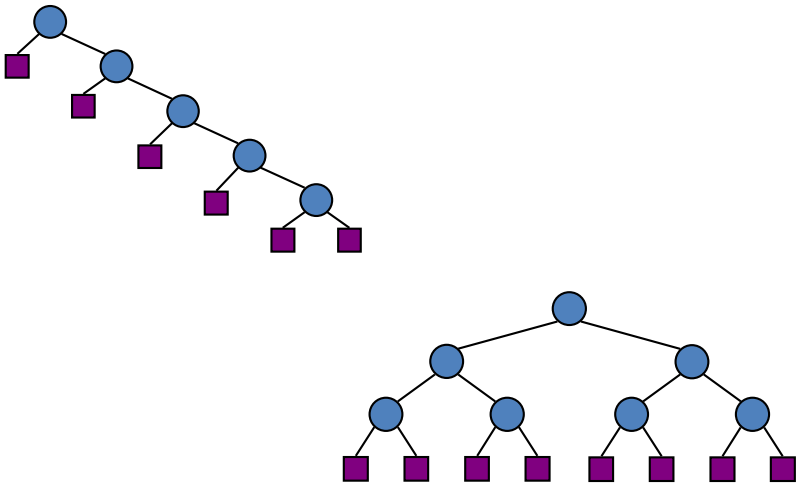




Performance

- Consider a BST with n items and height h
- The space used is O(n)
- Methods findElement, insertItem and removeElement take O(h) time
- The height h is O(n) in the worst case and O(log n) in the best case

Binary Search Tree



Balanced tree

- The worst-case performance, a BST achieves for various operations is linear time, which is no better than the performance of sequence-based dictionary implementations (such as log files and lookup tables).
- A simple way of correcting this problem is balanced binary search tree.

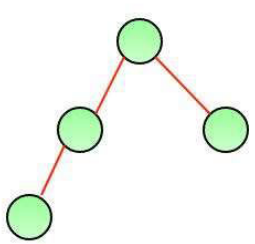
Balanced tree

- A balanced tree is a tree where every leaf is “not more than a certain distance” away from the root than any other leaf.
- Add a rule to the binary search tree definition that will maintain a logarithmic height for the tree
- Height-balance property

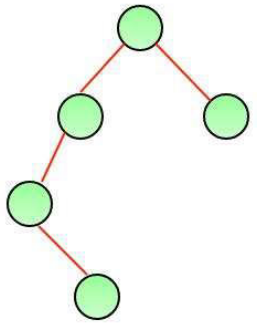
Balanced tree

- Height-Balance Property:

For every internal node v of T, the heights of the children of v can differ by at most 1 .



A height balanced tree

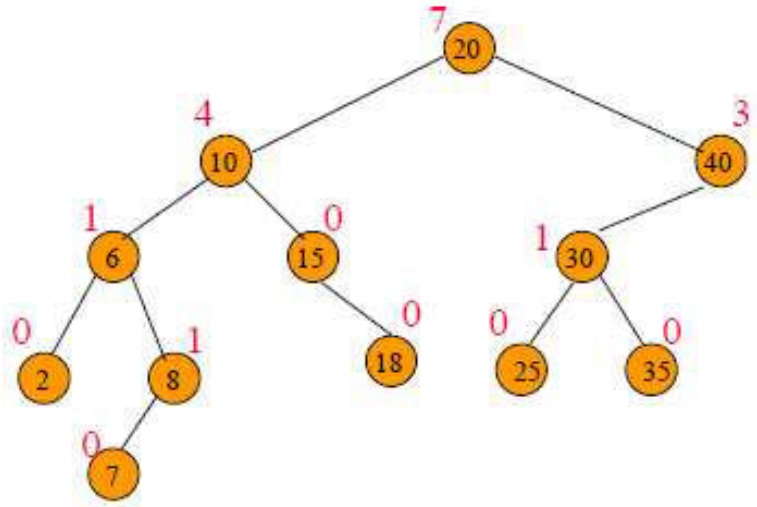


Not a height balanced tree

Leftsize

- Binary search tree.
- Each node has an additional field.
- leftSize = number of nodes in its left subtree

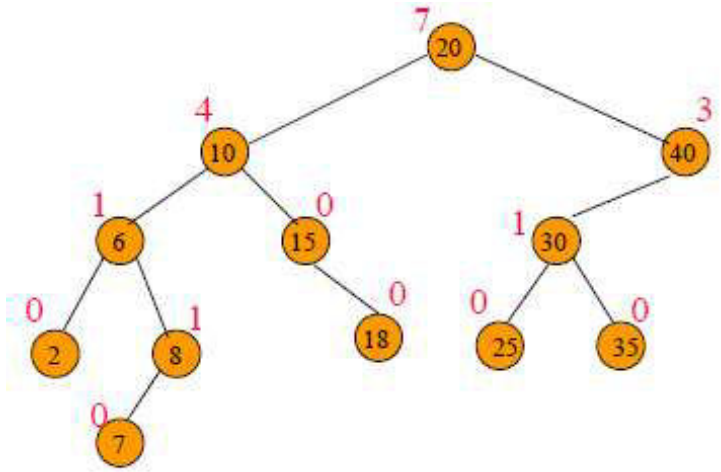
Leftsize



Rank

- Rank of an element is its position in inorder traversal (inorder = ascending key order).
- [2,6,7,8,10,15,18,20,25,30,35,40]
- rank(2) = 0
- rank(15) = 5
- rank(20) = 7
- **leftSize(x) = rank(x) with respect to elements in subtree rooted at x**

Rank



sorted list = [2,6,7,8,10,15,18,20,25,30,35,40]

Find k-th smallest element in BST

- The idea is to maintain rank of each node.
- We can keep track of elements in a subtree of any node while building the tree.
- Since we need K-th smallest element, we can maintain number of elements of left subtree in every node.

Find k-th smallest element in BST

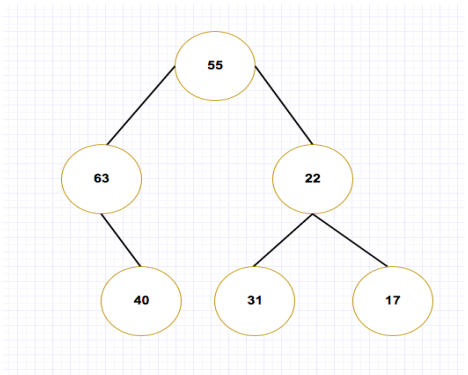
- Assume that the root is having N nodes in its left subtree.
  - If K = N + 1, root is K-th node.
  - If K > N, we continue our search in the right subtree for the (K – (N + 1))-th smallest element.
  - Else we will continue our search (recursion) for the Kth smallest element in the left subtree of root.
  - Note that we need the count of elements in left subtree only.
- Time complexity: O(h) where h is height of tree.

Find k-th smallest element in BST

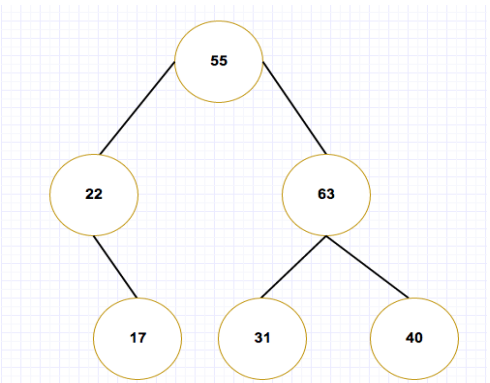
1. *start*
2. ***if K = root.leftElements + 1***
    1. *root node is the K th node.*
    2. *goto stop*
3. ***else if K > root.leftElements***
    1. *K = K - (root.leftElements + 1)*
    2. *root = root.right*
    3. *goto start*
4. ***else***
    1. *root = root.left*
    2. *goto start*
5. *stop*

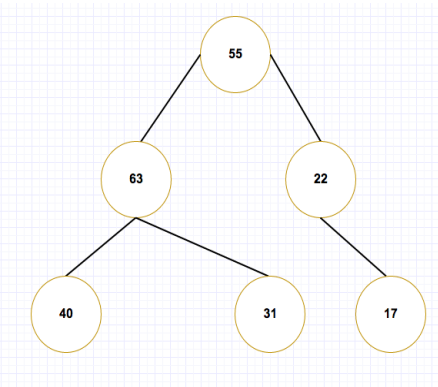
Qn:

- Suppose the keys 55,63,31,17,22,40 are inserted into a binary tree in that order. Which of the following is the BST that is formed?

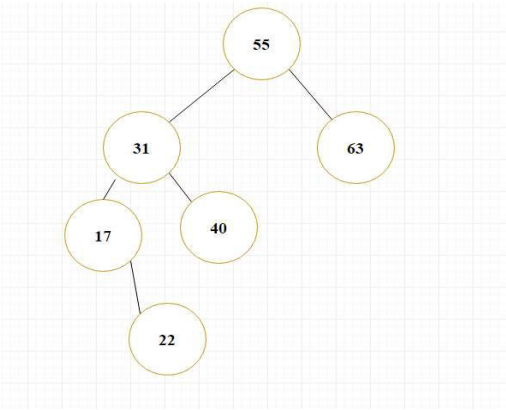


Qn :





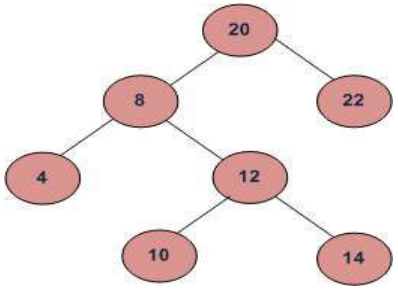
Qn :



Qn:LCA

- Given a binary search tree and two values say n1 and n2, write a program to find the least common ancestor. You may assume that both the values exist in the tree and n1<n2.

Qn:LCA-ANSWER



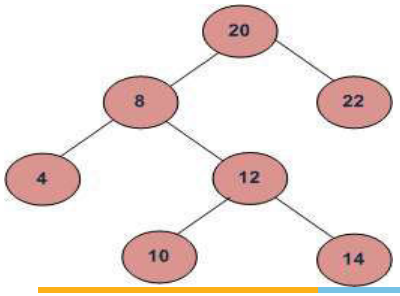
LCA of 10 and 14 is 12

LCA of 14 and 8 is 8

LCA of 10 and 22 is 20

Qn:LCA-ANSWER

- Let T be a rooted tree. The lowest common ancestor between two nodes n1 and n2 is defined as the lowest node in T that has both n1 and n2 as descendants (where we allow a node to be a descendant of itself).
- The LCA of n1 and n2 in T is the shared ancestor of n1 and n2 that is located farthest from the root.



Qn:LCA-ANSWER

- We can solve this problem using BST properties. We can **recursively traverse** the BST from root.
- The main idea of the solution is, while traversing from top to bottom, the first node n we encounter with value between n1 and n2, i.e., n1 < n < n2 or same as one of the n1 or n2, is LCA of n1 and n2 (assuming that n1 < n2).
- So just recursively traverse the BST, if node's value is greater than both n1 and n2 then our LCA lies in left side of the node, if it's is smaller than both n1 and n2, then LCA lies on right side. Otherwise root is LCA (assuming that both n1 and n2 are present in BST)

Qn:LCA-ANSWER

- Time complexity of above solution is O(h) where h is height of tree.

Data Structures and Algorithms Design

BITS Pilani
Hyderabad Campus

Febin.A.Vahab

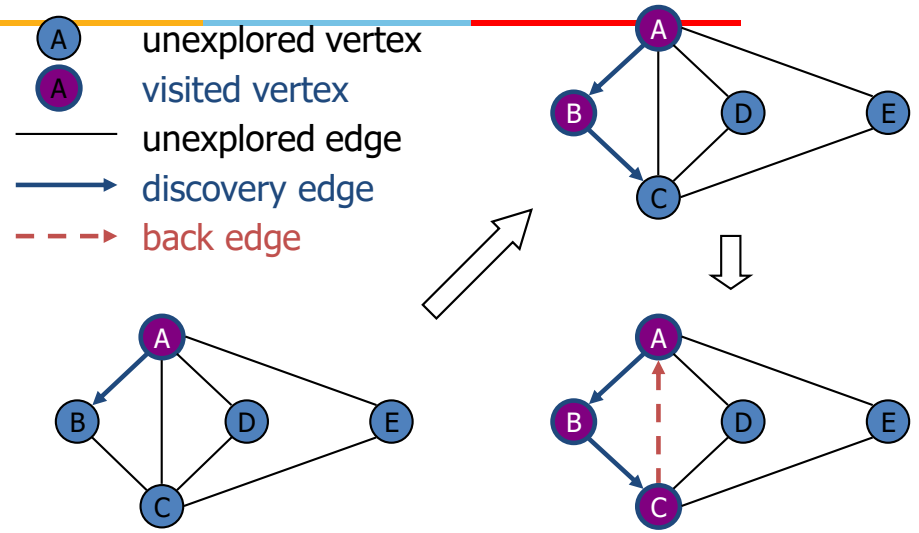
Depth-First Search

- Definitions
  - Subgraph
  - Connectivity
  - Spanning trees and forests
- Depth-first search
  - Algorithm
  - Example
  - Properties
  - Analysis
- Applications of DFS
  - Cycle finding
  - Path finding

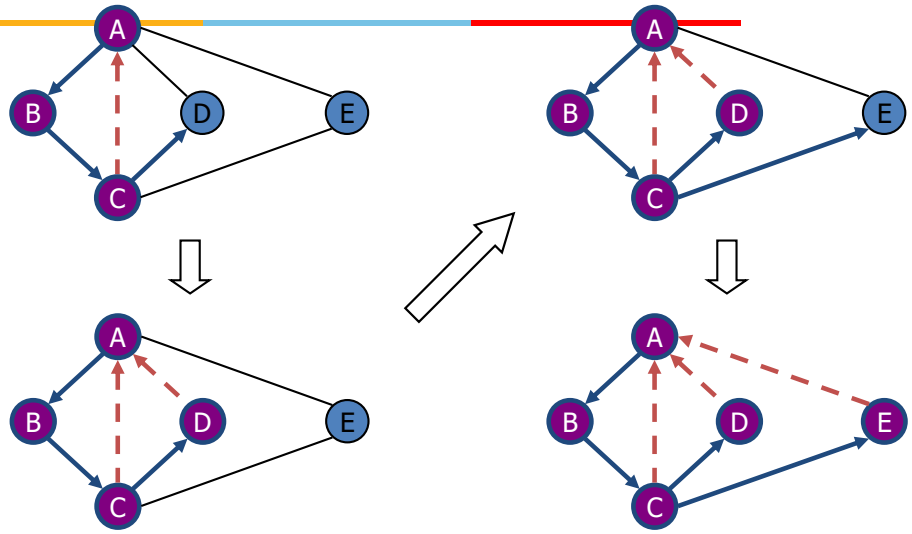
Depth-First Search

- Depth-first search (DFS) is a general technique for traversing a graph
- Search “deeper” in the graph whenever possible
- Explores edges out of the most recently discovered vertex that still has unexplored edges leaving it.
- Once all of v’s edges have been explored, the search “backtracks” to explore edges leaving the vertex from which v was discovered.
- This process continues until we have discovered all the vertices that are reachable from the original source vertex.
- If any undiscovered vertices remain, then depth-first search selects one of them as a new source, and it repeats the search from that source.
- The algorithm repeats this ntire process until it has discovered every vertex

Depth-First Search



Depth-First Search



Depth-First Search

- The DFS algorithm is similar to a classic strategy for exploring a maze
  - We mark each intersection, corner and dead end (vertex) visited
  - We mark each corridor (edge) traversed
  - We keep track of the path back to the entrance (start vertex) by means of a rope (recursion stack)

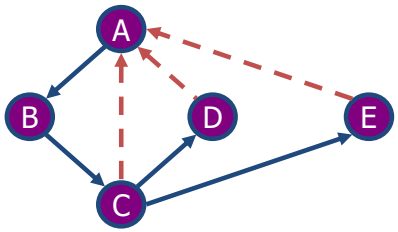
Depth-First Search-Properties

Property 1

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

Property 2

The discovery edges labeled by ***DFS*(*G, v*)** form a spanning tree of the connected component of ***v***



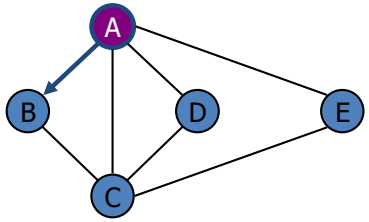
Depth-First Search

- The algorithm uses a mechanism for setting and getting "labels" of vertices and edges

Algorithm *DFS*(*G*)
  Input graph *G*
  Output labeling of the edges of *G* as discovery edges and back edges
  for all *u* ∈ *G.vertices*()
    setLabel(*u, UNEXPLORED*)
  for all *e* ∈ *G.edges*()
    setLabel(*e, UNEXPLORED*)
  for all *v* ∈ *G.vertices*()
    if *getLabel*(*v*) = *UNEXPLORED*
      DFS(*G, v*)

Depth-First Search

Algorithm *DFS*(*G, v*)
  Input graph *G* and a start vertex *v* of *G*
  Output labeling of the edges of *G* in the connected component of *v* as discovery edges and back edges
  setLabel(*v, VISITED*)
  for all *e* ∈ *G.incidentEdges*(*v*)
    if *getLabel*(*e*) = *UNEXPLORED*
      w ← *G.opposite*(*v,e*)
      if *getLabel*(*w*) = *UNEXPLORED*
        setLabel(*e, DISCOVERY*)
        DFS(*G, w*)
      else
        setLabel(*e, BACK*)



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
  - Recall that $\sum_v \deg(v) = 2m$

Depth-First Search

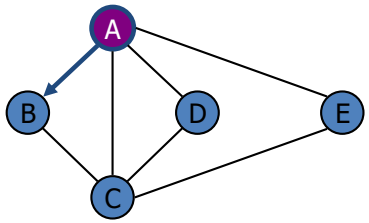
- A DFS traversal of a graph G
  - Visits all the vertices and edges of G
  - Determines whether G is connected
  - Computes the connected components of G
  - Computes a spanning tree of G
  - Computing a cycle in G, or reporting that G has no cycles
  - Find and report a path between two given vertices

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

Path Finding

```
Algorithm pathDFS(G, v, z)
    setLabel(v, VISITED)
    S.push(v)
     if v = z
         return S.elements()
     for all e ∈ G.incidentEdges(v)
         if getLabel(e) = UNEXPLORED
           w ← opposite(v, e)
           if getLabel(w) = UNEXPLORED
                        setLabel(e, DISCOVERY)
                        S.push(e)
                        pathDFS(G, w, z)
                        S.pop()                    { e gets popped }
           else
                        setLabel(e, BACK)
    S.pop()                                        { v gets popped }
```



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***

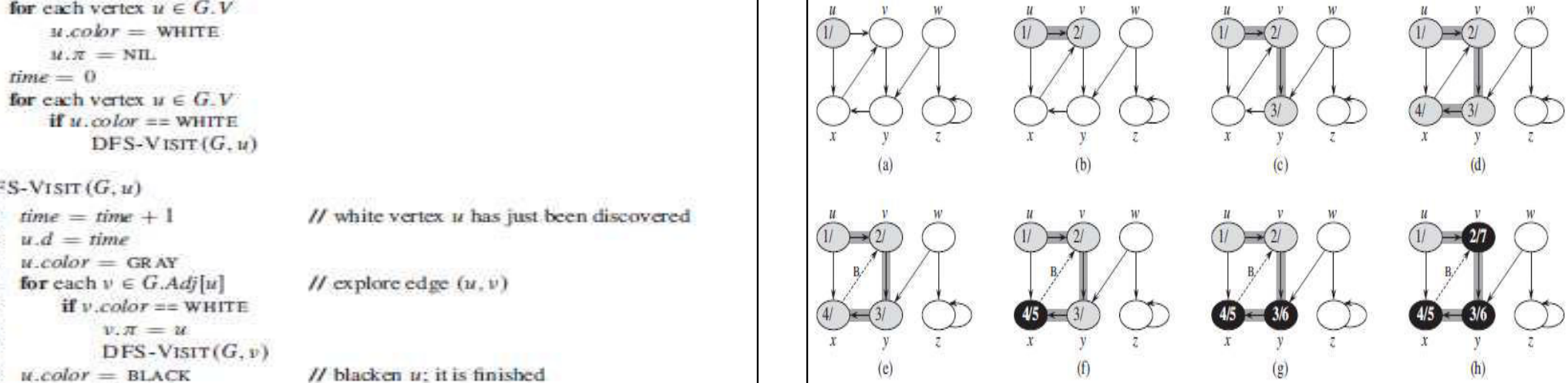
Cycle Finding

```
Algorithm cycleDFS(G, v, z)
    setLabel(v, VISITED)
    S.push(v)
     for all e ∈ G.incidentEdges(v)
         if getLabel(e) = UNEXPLORED
           w ← opposite(v,e)
           S.push(e)
           if getLabel(w) = UNEXPLORED
                        setLabel(e, DISCOVERY)
                        pathDFS(G, w, z)
                        S.pop()
           else
                        C ← new empty stack
                        repeat
                                    o ← S.pop()
                                    C.push(o)
                        until o = w
                        return C.elements()
    S.pop()
```

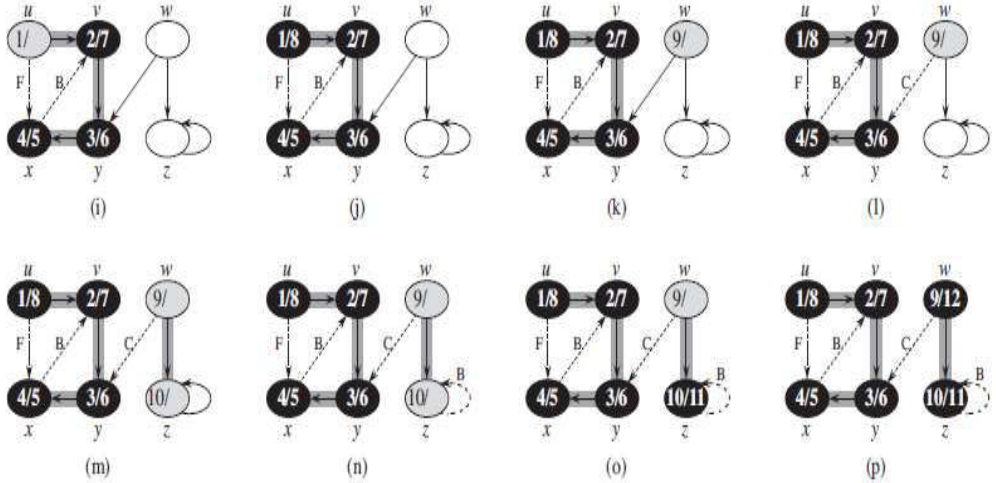
DFS:R2-Chapter 22

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)

DFS-VISIT(G, u)
1  time = time + 1                  // white vertex u has just been discovered
2  u.d = time
3  u.color = GRAY
4  for each v ∈ G.Adj[u]            // explore edge (u, v)
5      if v.color == WHITE
6          v.π = u
7          DFS-VISIT(G, v)
8  u.color = BLACK                  // blacken u; it is finished
9  time = time + 1
10 u.f = time
```


DFS:R2-Chapter 22



DFS:R2-Chapter 22



Connected components

How can DFS be used to find the connected components of a graph!

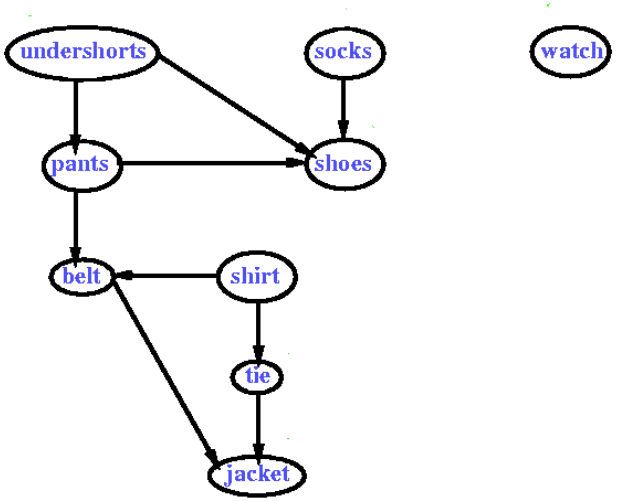
Can you implement it????
What will be the time complexity?

Connected components

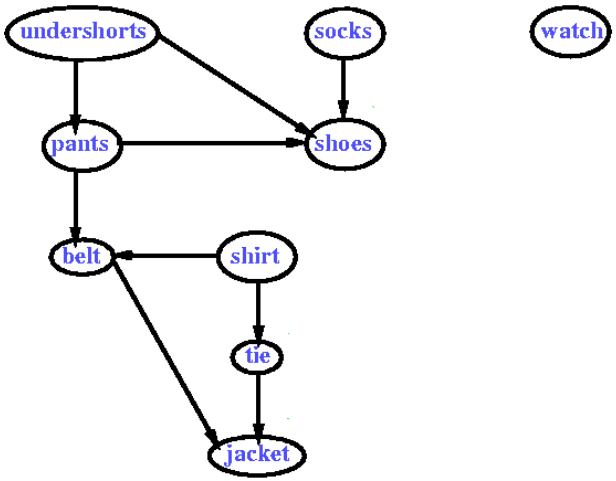
How can DFS be used to check whether a graph is connected or not?

Can you implement it????
What will be the time complexity?

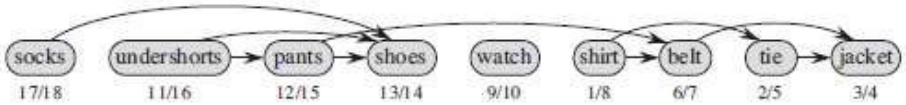
DFS for Toplogical Sort



DFS for Toplogical Sort



DFS for Toplogical Sort-Result



Breadth-first search

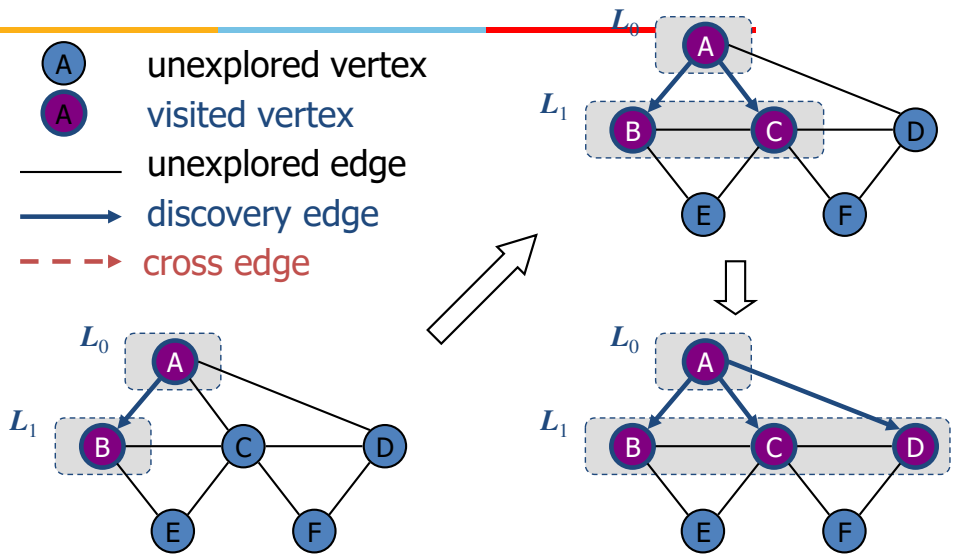
- Algorithm
- Example
- Properties
- Analysis
- Applications

Breadth-first search

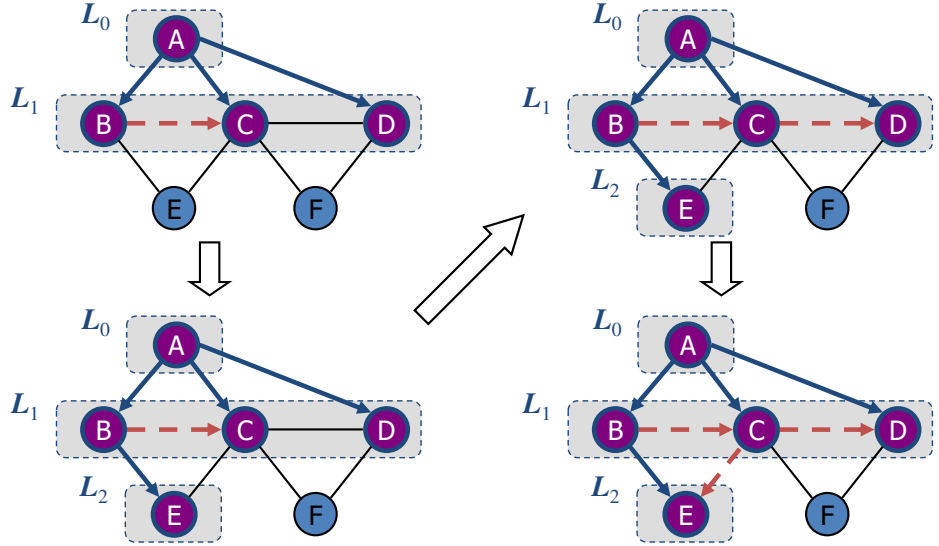
- Breadth-first search (BFS) is a general technique for traversing a graph
- A BFS traversal of a graph G
  - discovers all vertices at distance k from s before discovering any vertices at distance k + 1.

- For any vertex v reachable from vertex s, the simple path in the breadth-first tree from s to v corresponds to a "**shortest path**" from s to v in G, that is, a path containing the smallest number of edges.

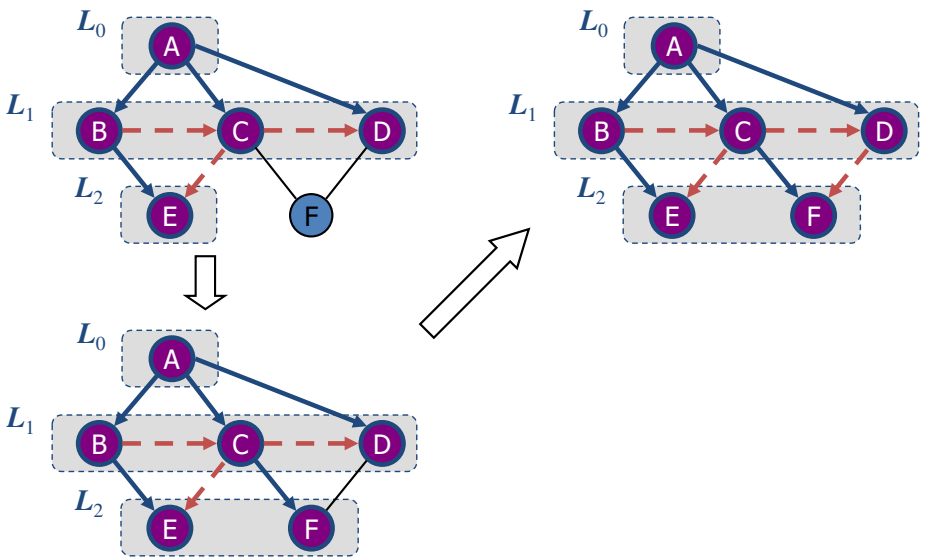
Breadth-first search



Breadth-first search



Breadth-first search



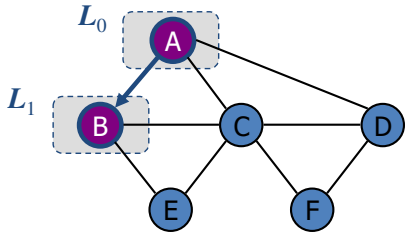
Breadth-first search

- The algorithm uses a mechanism for setting and getting "labels" of vertices and edges

Algorithm *BFS*(*G*)
  Input graph *G*
  Output labeling of the edges and partition of the vertices of *G*
    for all *u* ∈ *G.vertices*()
      setLabel(*u, UNEXPLORED*)
    for all *e* ∈ *G.edges*()
      setLabel(*e, UNEXPLORED*)
    for all *v* ∈ *G.vertices*()
      if *getLabel*(*v*) = *UNEXPLORED*
        BFS(*G, v*)

Breadth-first search – Algorithm *BFS*(*G, s*)

L_0 ← new empty sequence
L_0.*insertLast*(*s*)
setLabel(*s, VISITED*)
i ← 0
while ¬L_i.*isEmpty*()
  L_{i+1} ← new empty sequence
  for all *v* ∈ L_i.*elements*()
    for all *e* ∈ *G.incidentEdges*(*v*)
    if *getLabel*(*e*) = *UNEXPLORED*
      w ← *opposite*(*v,e*)
      if *getLabel*(*w*) = *UNEXPLORED*
        setLabel(*e, DISCOVERY*)
        setLabel(*w, VISITED*)
        L_{i+1}.*insertLast*(*w*)
      else
        setLabel(*e, CROSS*)
i ← *i* +1



Breadth-first search – Algorithm *BFS*(*G, s*)

- We use auxiliary space to label edges, mark visited vertices, and store containers associated with levels.
- That is, the containers L0, L 1 , L2, and so on, store the nodes that are in level 0, level 1 , level 2, and so on.

Properties

Notation

G_s: connected component of s

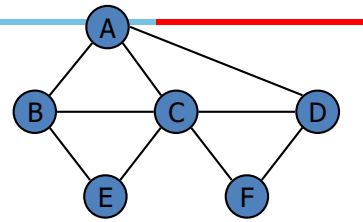
Property 1

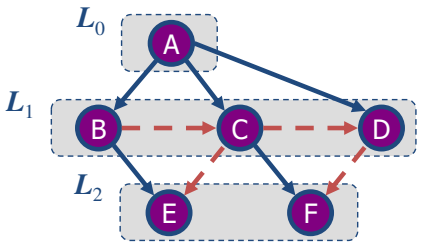
$BFS(G, s)$ visits all the vertices and edges of G_s

Property 2

The discovery edges of a connected component labeled by $BFS(G, s)$ form a spanning tree T_s of G_s

Properties





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
  - Recall that $\sum_v \deg(v) = 2m$

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
  - Compute a spanning forest 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

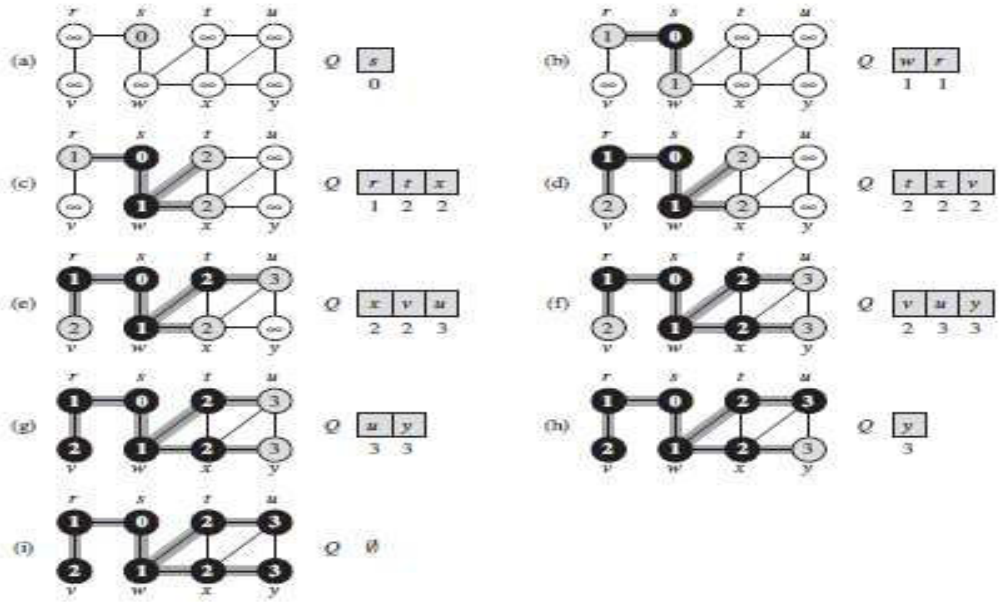
Facebook as Graph

- Traversal: go to 'Friends' to display all your friends (like G.Neighbors)
- BFS: the tabs are a queue - open all friends profiles in new tabs, then close current tab and go to the next one
- DFS: the history is a stack - open the first hot friend profile in the same window; when hitting a dead end, use back button

BFS-CLRS

```
BFS(G, s)
1  for each vertex u ∈ G.V − {s}
2      u.color = WHITE
3      u.d = ∞
4      u.π = NIL
5  s.color = GRAY
6  s.d = 0
7  s.π = NIL
8  Q = ∅
9  ENQUEUE(Q, s)
10 while Q ≠ ∅
11     u = DEQUEUE(Q)
12     for each v ∈ G.Adj[u]
13         if v.color == WHITE
14             v.color = GRAY
15             v.d = u.d + 1
16             v.π = u
17             ENQUEUE(Q, v)
18     u.color = BLACK
```

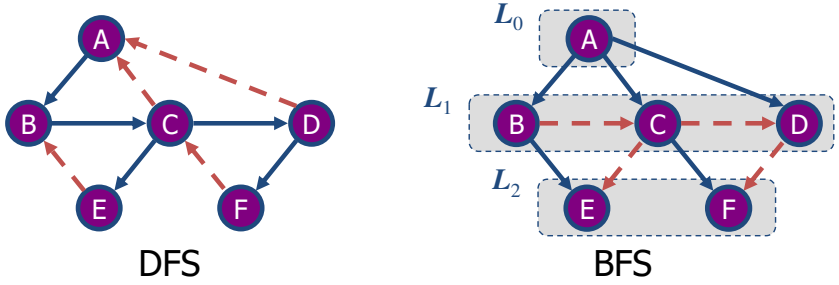

BFS-CLRS



DFS vs. BFS

Application	DFS	BFS
Spanning forest, connected components, paths, cycles	Y	Y
Shortest Paths		Y

DFS vs. BFS



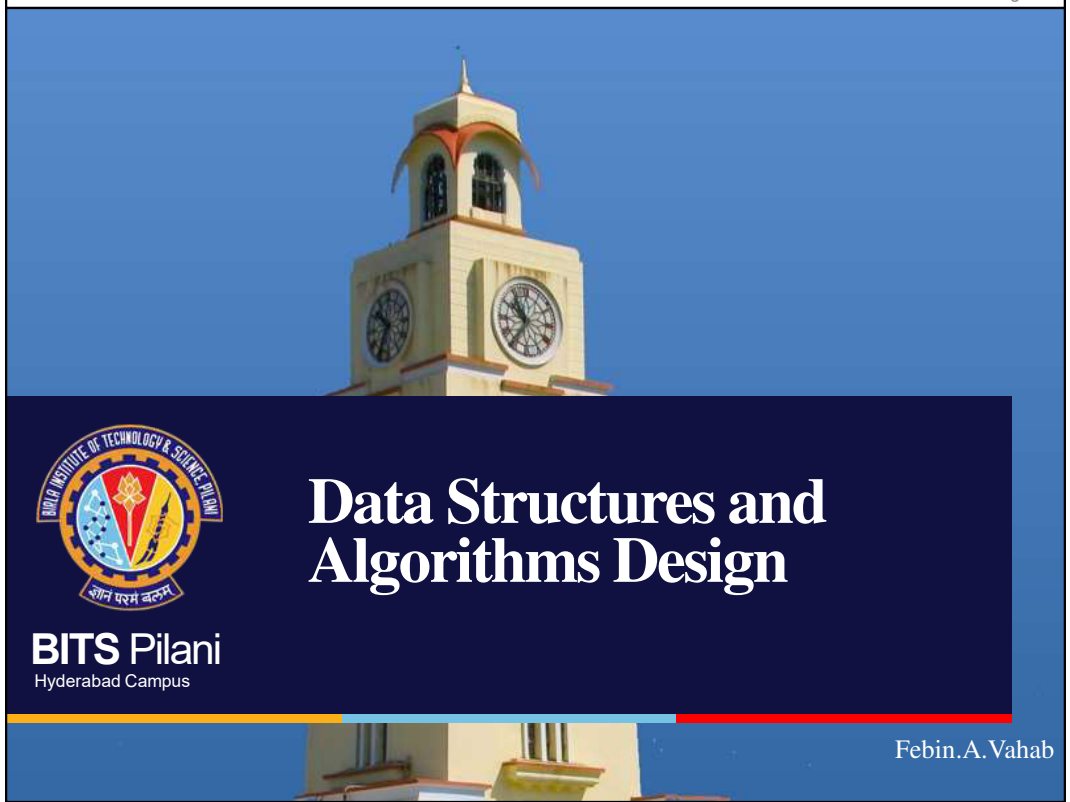
DFS vs. BFS

Back edge (v,w)

- w is an ancestor of v in the tree of discovery edges

Cross edge (v,w)

- w is in the same level as v or in the next level in the tree of discovery edges



SESSION -PLAN

Session(#)	List of Topic Title	Text/Ref Book/external resource
10	**M5:Algorithm Design Techniques** Greedy Method - Design Principles and Strategy, Fractional Knapsack Problem, Task Scheduling Problem Minimum Spanning Tree, Shortest Path Problem - Dijkstra's Algorithm	T1: 5.1, 7.1, 7.3

Greedy method-Change making problem

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

Greedy Method

- It works best when applied to problems with the **greedy-choice** property:
  - ie. a globally optimal solution can be arrived at by making a locally optimal (greedy) choice.
  - When a locally optimal choice can lead to a globally optimal solution, the problem has this property.
- **Optimal Substructure property** : An optimal solution to the problem contains optimal solution to the sub problems.

Making Change-Example 2

- Problem: A dollar amount to reach and a collection of coin amounts to use to get there.
- Configuration: A dollar amount yet to return to a customer plus the coins already returned
- Objective function: Minimize number of coins returned.
- Greedy solution: Always return the largest coin you can

Making Change

- Example 1: Coins are valued $.32, $.08, $.01
  - Has the greedy-choice property, since no amount over $.32 can be made with a minimum number of coins by omitting a $.32 coin.
- Example 2: Coins are valued $.30, $.20, $.05, $.01
  - Does not have greedy-choice property, since $.40 is best made with two $.20's, but the greedy solution will pick three coins (which ones?)

Greedy Method

- An algorithm design strategy that always tries to find the best solution for each sub problem with the hope that it will yield a good solution for the problem as a whole.
- It makes the choice that looks best at the moment.
- Don't worry about the effect these choices have in the future.
- At each step the choice should be
  - Feasible
  - Locally optimal
  - Irrevocable

Greedy Method

- Applied for **optimization problems**
- In order to solve a given optimization problem, we proceed by a sequence of choices.
- The sequence starts from some well-understood starting configuration, and then iteratively makes the decision that seems best from all of those that are currently possible.
- Greedy approach does not always lead to an optimal solution. It works optimally for problems with **greedy-choice** property

The Fractional Knapsack Problem

- Given: A knapsack of capacity M and n objects of weights w1,w2,..,wn with profits p1,p2,p3..,pn.
- **Goal: Choose items with maximum total benefit but with weight at most M.**

The Fractional Knapsack Problem

- If we are allowed to take fractional amounts, then this is the **fractional knapsack problem**.
- Let x1,x2,..,xn be the fractions of objects that are supposed to be added to knapsack.
  - Maximize $\sum_{i=1}^{n} p_i x_i$
  - Constraint : $\sum_{i=1}^{n} w_i x_i \leq M$

The Fractional Knapsack Problem

- Greedy choice:
  - **Keep taking item with highest value (profit to weight ratio)**
- Algorithm: Iteratively picks the item with the greatest value(pi/wi).
- If, at the end, the knapsack cannot fit the entire last item with greatest value among the remaining items, we will take a fraction of it to fill the knapsack.

The Fractional Knapsack Problem

- Let n=3 M=40

Weight(wi)	Profit(Pi)
20	30
25	40
10	35

The Fractional Knapsack Problem

- Let n=3 M=40

Weight(wi)	Profit(Pi)
20	30
25	40
10	35

The Fractional Knapsack Algorithm

Algorithm

fractionalKnapsack(m,n, w,x,p)

Input:

m-Capacity of Knapsack ,

n- no:of objects,

w-array of weights of all n objects,

p-array of profits of all n objects

Output:

x –array containing the solution

Arrange the objects in the decreasing order of pi/wi

```
for i ← 1 to n
      x[i]=0
end for

rc← m

for i  ← 1 to n
      if (w[i]>rc) break;
      x[i] ← 1
      rc←rc-W[i]
      //p←p+p[i]
end for

if(i<=n)
      {x[i]=rc/w[i]}
      //p=(p[i]*x[i])+p
```


The Fractional Knapsack Algorithm

- Time complexity
- **O(nlogn)**
- Why?
- Specifically, we use a heap-based priority queue to store the items of S, where the key of each item is its value . With this data structure, each greedy choice, which removes the item with greatest value, takes O(log n) time. Process is repeated for n items. Hence O(nlogn)

Correctness

Greedy Choice Property: Let i be the index of the item with maximum pi/wi. Then there exists an optimal solution in which you take as much of item i as possible.

Given a set of n items $\{1, 2, ..., n\}$.

- Assume items sorted by per-pound values: $\rho_1 \geq \rho_2 \geq ... \geq \rho_n$.

Let the greedy solution be $G = \langle x_1, x_2, ..., x_k \rangle$

- x_i indicates fraction of item i taken (all $x_i = 1$, except possibly for $i = k$).

Consider any optimal solution $O = \langle y_1, y_2, ..., y_n \rangle$

- y_i indicates fraction of item i taken in O (for all i, $0 \leq y_i \leq 1$).
- Knapsack must be full in both G and O: $\sum_{i=1}^{n} x_i w_i = \sum_{i=1}^{n} y_i w_i = K$.

Consider the first item i where the two selections differ.

- By definition, solution G takes a greater amount of item i than solution O (because the greedy solution always takes as much as it can). Let $x = x_i - y_i$.

Correctness

- There must exist some item j ≠ i with pj/wj < pi/wi that is in the knapsack.
- We can therefore take a piece of j, with x weight, out of the knapsack, and put a piece of i with x weight in.
- This increases the knapsack's value by
- x[pi/wi] –x[pj/wj]= x[(pi/wi) – (pj/wj)] >0
- **Contradiction** to the original solution being optimal.

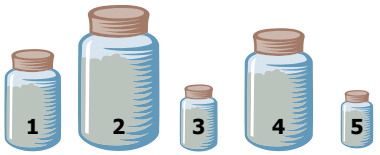
Applications

- Portfolio optimization:
- Cutting stock problems
- Huffman encoding for text compression : Compute high frequency characters with short code words.
- Web auction Optimization
- **K Means procedure in clustering**: It can be viewed as a greedy algorithm for partitioning the n examples into k clusters so as to minimize the sum of the squared distances to the cluster centers

Web Auction Optimization

- Suppose you are designing a new online auction website that is intended to process bids for multi-lot auctions.
- This website should be able to handle a single auction for 100 units of the same digital camera or 500 units of the same smartphone, where bids are of the form, "x units for $y," meaning that the bidder wants a quantity of x of the items being sold and is willing to pay $y for all x of them.
- The challenge for your website is that it must allow for a large number of bidders to place such multi-lot bids and it must decide which bidders to choose as the winners.
- Naturally, one is interested in designing the website so that it always chooses a set of winning bids that maximizes the total amount of money paid for the items being auctioned. **So how do you decide which bidders to choose as the winners?**

Exercise



	1	2	3	4	5
Items:					
Weight:	4 ml	8 ml	2 ml	6 ml	1 ml
Benefit:	$12	$32	$40	$30	$50
Value: ($ per ml)	3	4	20	5	50

10 ml

"knapsack"

Solution:
- 1 ml of 5
- 2 ml of 3
- 6 ml of 4
- 1 ml of 2

Task Scheduling Problem

- Job sequencing With deadlines
- Interval Scheduling

Job sequencing with deadlines

- Given a set of n jobs and associated with each job i is an integer deadline d_i >=0 and profit pi >0,it is required to find the set of jobs such that all the chosen jobs should be completed within their deadlines and the profit earned should be maximum with the following constraints.
- Only one machine is available for processing jobs
- Only one job must be processed at any point of time
- A job is said to be completed if it is processed on a machine for one unit time.
- **Greedy choice property**: Arrange the job in decreasing order of profits and pick the highest profit job first .

Job sequencing with deadlines

- Example

i	1	2	3	4	5
pi	20	10	15	1	5
di	2	1	2	3	3

Job sequencing with deadlines

- Example

i	1	2	3	4	5
pi	20	15	10	5	1
di	2	2	1	3	3

Job sequencing with deadlines

Algorithm JobSeq(job[1..n],p[1..n],d[1..n],list[1..n],n)

```
Arrange the objects in decreasing order of profits.
Initialise list with 0s
Profit=0
for i=1 to n do
        k=d[i]
        while (k>0)
                if(list[k]=0)
                        list[k]=job[i]
                        add p[i] to profit
                        break;
                end if
                decrement k by 1
        end while
  End for
  Print list
```


Job sequencing with deadlines

- Time complexity
- $O(n^2)$
- Why?

Example

- Given a set of 9 jobs where each job has a deadline and profit associated to it .Each job takes 1 unit of time to complete and only one job can be scheduled at a time. We earn the profit if and only if the job is completed by its deadline. The task is to find the maximum profit and the number of jobs done.

Jobs	Profit	Deadline
J1	85	5
J2	25	4
J3	16	3
J4	40	3
J5	55	4
J6	19	5
J7	92	2
J8	80	3
J9	15	7

Task Scheduling Problem [Interval Partitioning]

- 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."

Task Scheduling Algorithm [Interval Partitioning]

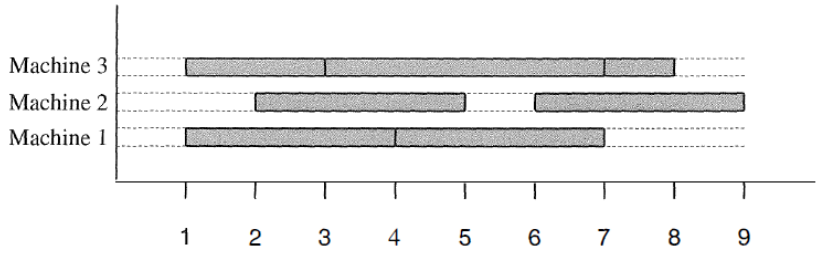
- Suppose we are given a set T of n tasks, such that each task i has a start time, s_i , and a finish time, f_i (where $s_i < f_i$). Task i must start at time s_i and it is guaranteed to be finished by time f_i.
- Each task has to be performed on a machine and each machine can execute only one task at a time.
- Two tasks i and j are nonconflicting if $f_i <= s_j$ or $f_j <= s_i$.
- Two tasks can be scheduled to be executed on the same machine only if they are nonconflicting

Task Scheduling Problem [Interval Partitioning]

- The task scheduling problem we consider here is to schedule all the tasks in T on the fewest machines possible in a nonconflicting way.

- Greedy choice.
  - [Earliest start time] Consider tasks in ascending order of sj.
  - [Earliest finish time] Consider tasks in ascending order of fj
  - [Shortest interval] Consider tasks in ascending order of fj - sj.

Task Scheduling Problem [Interval Partitioning]



An illustration of a solution to the task scheduling problem, for tasks whose collection of pairs of start times and finish times is { (1 , 3) , (1 , 4) , (2, 5) , (3, 7) , (4, 7) , (6, 9) , (7, 8) } .

Task Scheduling Problem [Interval Partitioning]-Algorithm

- Consider tasks in increasing order of start time:

Algorithm TaskSchedule(T):
Input: A set T of tasks, such that each task has a start time s_i and a finish time f_i
Output: A nonconflicting schedule of the tasks in T using a minimum number of machines

```
m ← 0          {optimal number of machines}
while T ≠ ∅ do
    remove from T the task i with smallest start time s_i
    if there is a machine j with no task conflicting with task i then
        schedule task i on machine j
    else
        m ← m + 1          {add a new machine}
        schedule task i on machine m
```


Task Scheduling Problem [Interval Partitioning]-

- Time complexity : O(n log n)
  - Sorting by start times takes O(n log n) time.
  - Store machines in a priority queue (key = finish time of its last job).
  - To allocate a new machine, INSERT machine onto priority queue.
  - To schedule Task j in machine k, INCREASE-KEY of classroom k to fj.
  - To determine whether Task j is compatible with any machine, compare sj to FIND-MIN
  - Total # of priority queue operations is O(n); each takes O(log n) time.
  - **This implementation chooses a machine k whose finish time of its last job is the earliest**

Sample problem1-**Meeting Room Scheduling**

- Schedule the meetings in as few conference rooms as possible.
- *Given a list of meeting times, checks if any of them overlaps. The follow-up question is to return the minimum number of rooms required to accommodate all the meetings.*
- Suppose we use interval (start, end) to denote the start and end time of the meeting, we have the following meeting times: (1, 4), (5, 6), (8, 9), (2, 6).

Sample problem1

- In the above example, apparently we should return true for the first question since (1, 4) and (2, 6) have overlaps.
- For the follow-up question, we need at least 2 meeting rooms so that (1, 4), (5, 6) will be put in one room and (2, 6), (8, 9) will be in the other.

Sample problem2

- A teacher assign homework at the beginning of the first day of class. Each problem carries 1 mark + bonus if submitted within specified number of days. The deadline for submitting and bonus marks are different for each problem. Each problem takes exactly one day to complete. Maximum bonus one can obtain is 15.Your task is to find a schedule to complete all homework problems so as to maximize bonus marks..

Problem number	1	2	3	4	5	6	7
Deadline for bonus	1	1	3	3	2	2	6
Bonus	6	7	2	1	4	5	1

Sample problem3

- A large ship is to be loaded with cargo. The cargo is containerized, and all containers are the same size. Different containers may have different weights. Let w_i be the weight of the ith container, 1<=i<=n. The cargo capacity of the ship is c. We wish to load the ship with the maximum number of containers.

Sample problem3

Formulation of the problem :

- Variables : x_i $(1 \leq i \leq n)$ is set to 0 if the container i is not to be loaded and 1 in the other case.
- Constraints : $\sum_{i=1}^{n} w_i x_i \leq c.$
- Optimization function : $\sum_{i=1}^{n} x_i$

Sample problem3

- Using the greedy algorithm the ship may be loaded in stages; one container per stage.
- At each stage, the greedy criterion used to decide which container to load is the following :
  - *From the remaining containers, select the one with least weight.*
- This order of selection will keep the total weight of the selected containers minimum and hence leave maximum capacity for loading more containers.

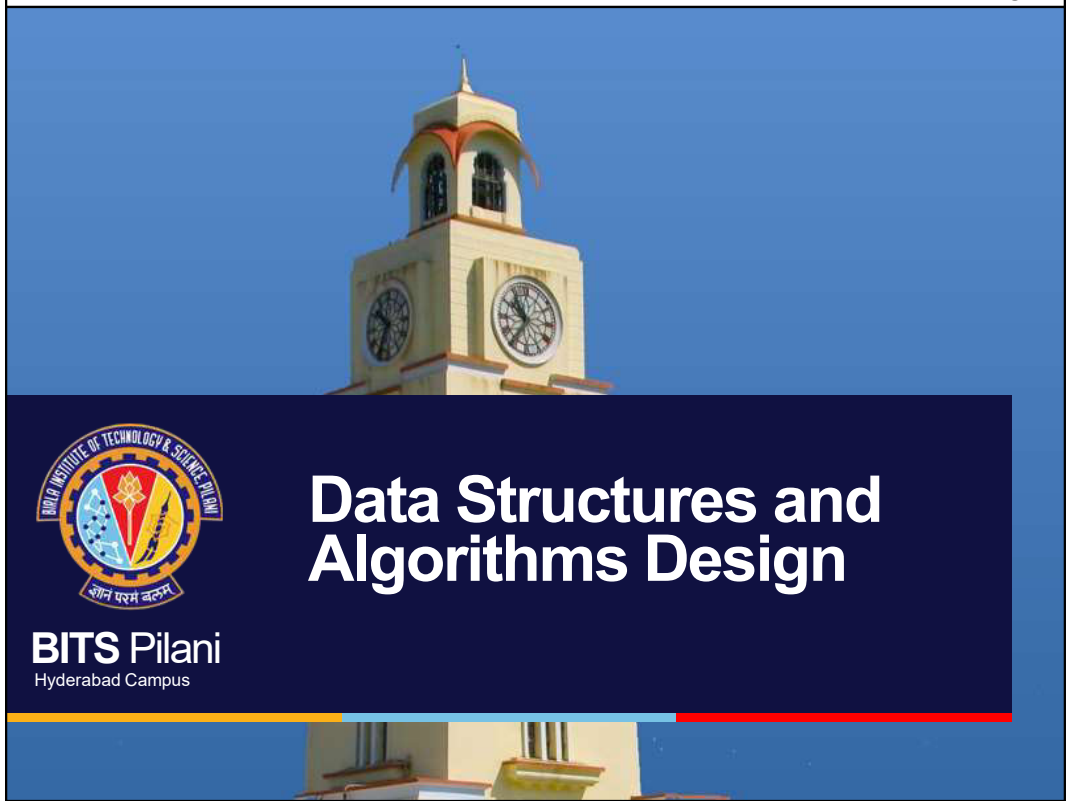
Sample problem3

- Suppose that :
  - $n = 8$,
  - $[w_1, \ldots, w_8] = [100, 200, 50, 90, 150, 50, 20, 80]$,
  - and $c = 400$.
- When the greedy algorithm is used, the containers are considered for loading in the order 7,3,6,8,4,1,5,2.
- Containers 7,3,6,8,4 and 1 together weight 390 units and are loaded.
- The available capacity is now 10 units, which is inadequate for any of the remaining containers.
- In the greedy solution we have :

$$[x_1, \ldots, x_8] = [1, 0, 1, 1, 0, 1, 1, 1] \text{ and } \sum x_i = 6.$$

Sample problem4

- Lecture j starts at sj and finishes at fj.
- Goal: find minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.



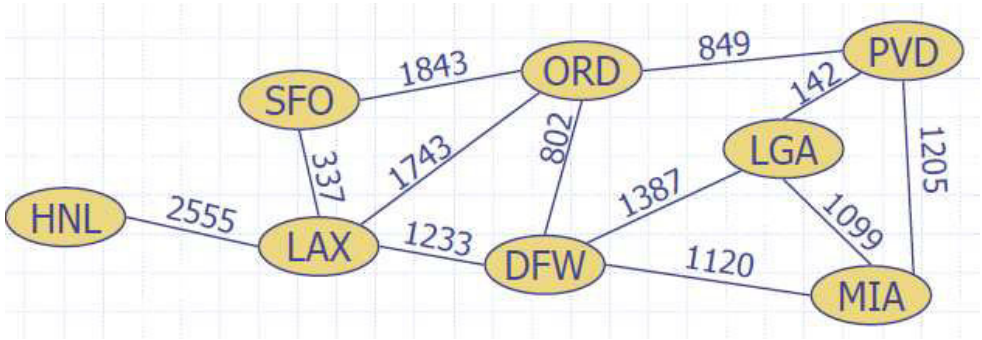
ONLINE SESSION -PLAN

Sessions(#)	List of Topic Title	Text/Ref Book/external resource
10	**M5:Algorithm Design Techniques** Greedy Method - Minimum Spanning Tree, Shortest Path Problem - Dijkstra's Algorithm	T1: 5.1, 7.1, 7.3

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: A flight route graph,the weight of an edge represents the distance in miles between the endpoint airports.

Weighted Graphs



Dijkstra's Algorithm-Single source shortest path algorithm

- The ***distance*** of a vertex ***v*** from a vertex ***s*** is the length of a shortest path between ***s*** and ***v***
- Dijkstra's algorithm computes the ***distances*** of all the vertices from a given start vertex ***s***
- Assumptions:
  - The graph is connected
  - The edge weights are nonnegative

Dijkstra's Algorithm

- We grow 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

Dijkstra's Algorithm

- At each step
  - We add to the cloud the vertex ***u*** outside the cloud with the smallest distance label, ***d*(*u*)**

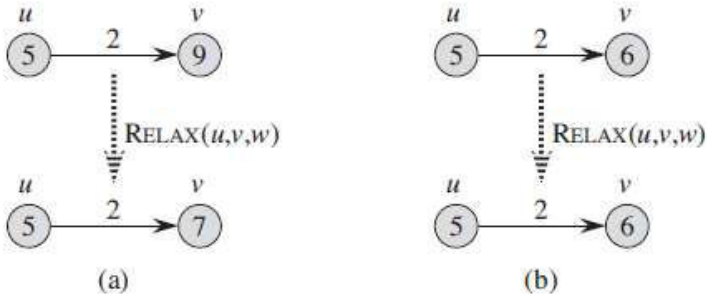
  [Greedy choice]
  - We update the labels of the vertices adjacent to ***u***

Dijkstra's Algorithm

Edge Relaxation

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2        v.d = u.d + w(u, v)
3        v.π = u
```


Dijkstra's Algorithm



- Relaxing an edge (u,v)with weight =2. The shortest-path estimate of each vertex appears within the vertex.
  a) Because v.*d* > u.*d* + w(u,v) prior to relaxation, the value of v,*d* decreases.
  b) Here, v.d < u.*d* + w(u,v) before relaxing the edge, and so the relaxation step leaves v.*d* unchanged.

Dijkstra's Algorithm-{Cormen}

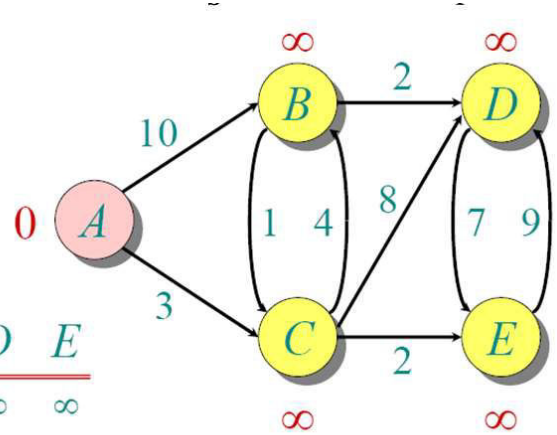
```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ G.V
2      v.d = ∞
3      v.π = NIL
4  s.d = 0
```

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2      v.d = u.d + w(u, v)
3      v.π = u
```


Dijkstra's Algorithm-{Cormen}

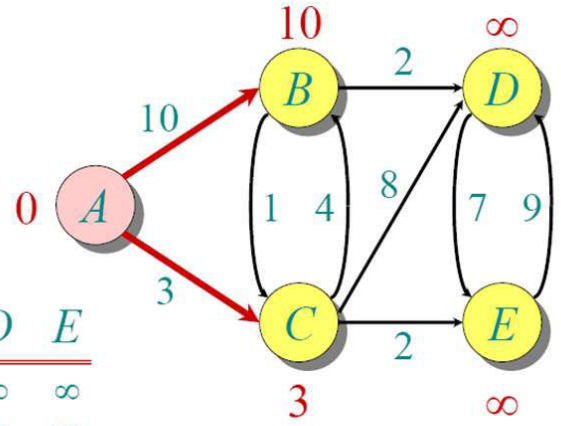
```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```


Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞

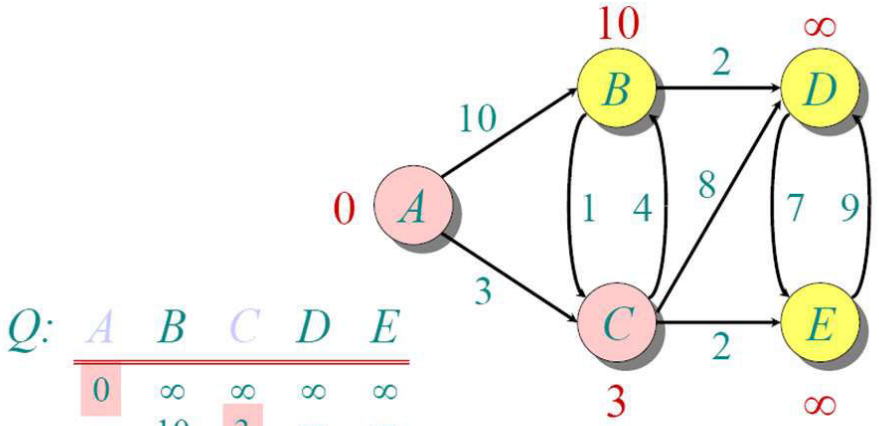
Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A }

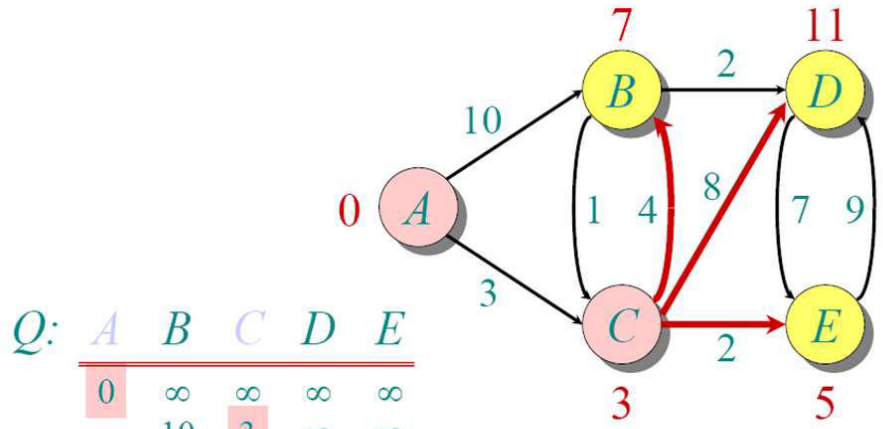
Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A, C }

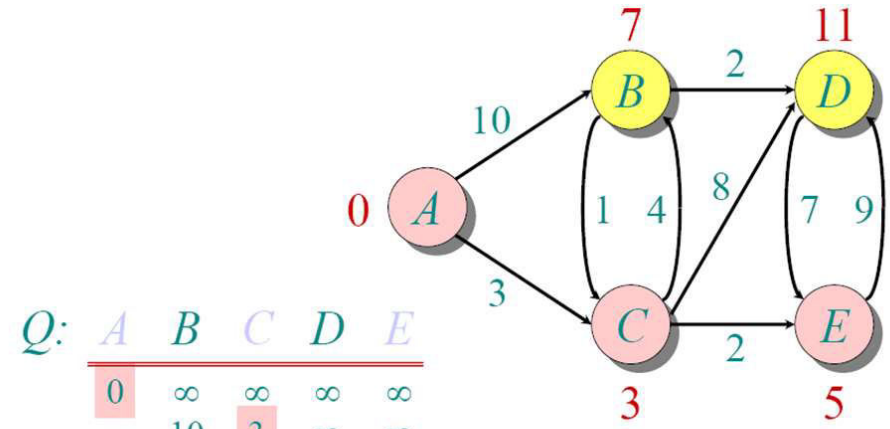
Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { A, C }

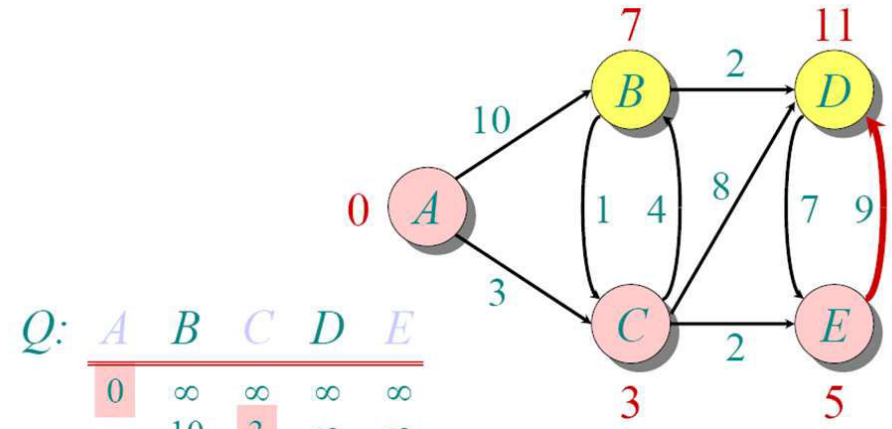
Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { A, C, E }

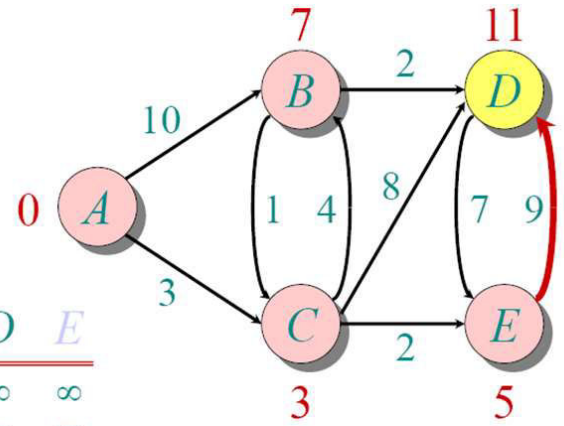
Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { A, C, E }

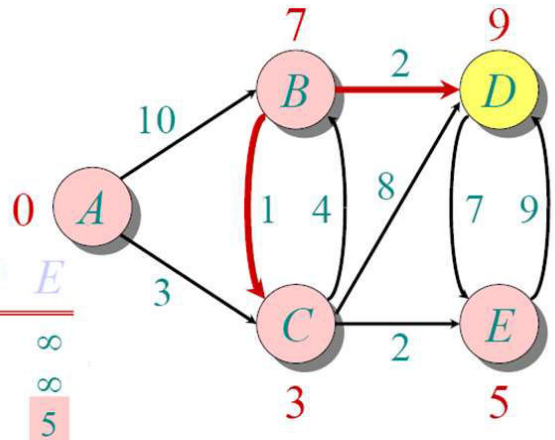
Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { A, C, E, B }

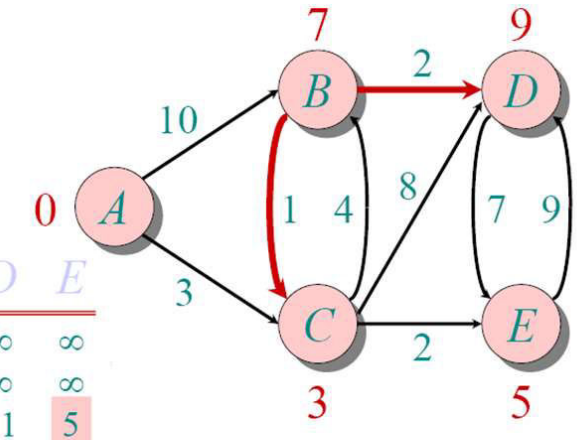
Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

S: { A, C, E, B }

Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

S: { A, C, E, B, D }

Dijkstra's Algorithm-{Cormen}

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

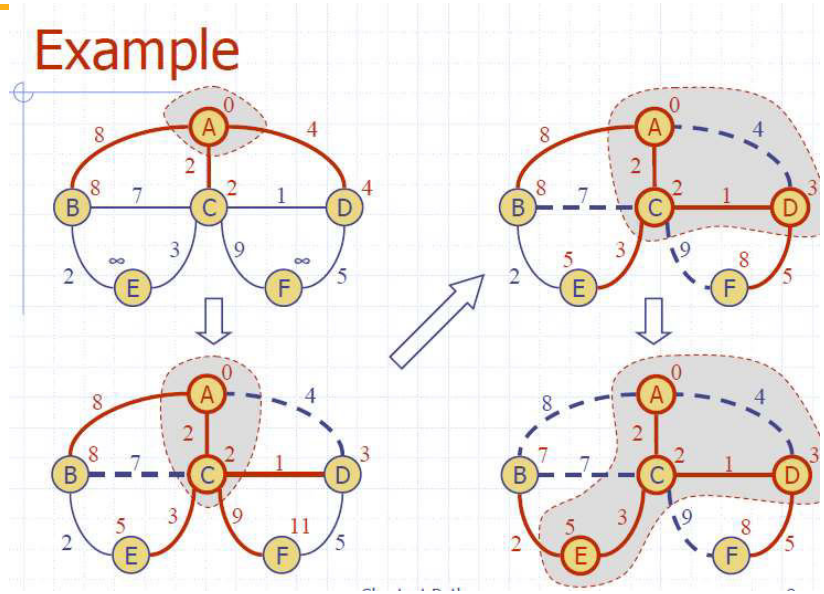
Dijkstra's Algorithm-Analysis

- The simplest implementation of the Dijkstra's algorithm stores vertices in an ordinary linked list or array
- Good for dense graphs (many edges)
  - |V| vertices and |E| edges
  - Initialization O(|V|)
  - While loop O(|V|)
  - Find and remove min distance vertices O(|V|)
  - Potentially |E| updates
  - Update costs O(1)
  - Total time $O(|V^2| + |E|) = O(|V^2|)$

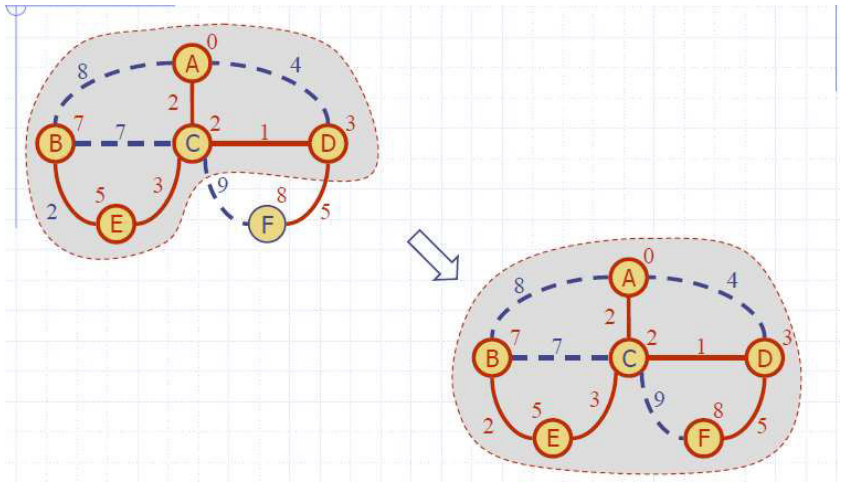
Dijkstra's Algorithm-Analysis

- For sparse graphs, (i.e. graphs with much less than $|V^2|$ edges) Dijkstra's implemented more efficiently by priority queue
  - Initialization O(|V|) using O(|V|) buildHeap
  - While loop O(|V|)
  - Find and remove min distance vertices = O(log |V|) using O(log |V|) deleteMin
  - Taken together that part of the loop and the calls to ExtractMin take O(VlogV) time
  - Potentially |E| updates:The for loop is executed once for each edge in the graph (E times), and within the for loop ,the update costs O(log |V|) using decreaseKey
  - Total time O(|V|log|V| + |E|log|V|) = O ((V+E)log V)
  - Recall $\Sigma_v \deg(v) = 2m$ ***ie.*** |V| = O(|E|) assuming a connected graph
  - Hence **the total time=O(|E| log |V|)**

Dijkstra's Algorithm



Dijkstra's Algorithm

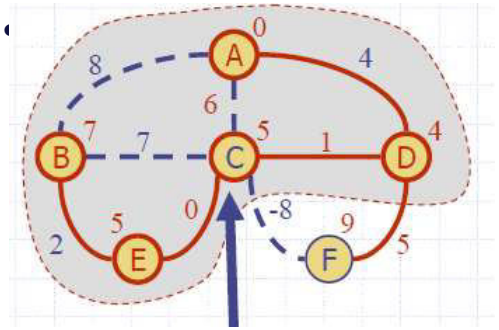


Dijkstra's Algorithm-Why It Doesn't Work for Negative-Weight Edges

- Dijkstra relies on one "simple" fact: if all weights are non-negative, adding an edge can never make a path shorter. That's why picking the shortest candidate edge (local optimality) always ends up being correct (global optimality).

Dijkstra's Algorithm-Why It Doesn't Work for Negative-Weight Edges

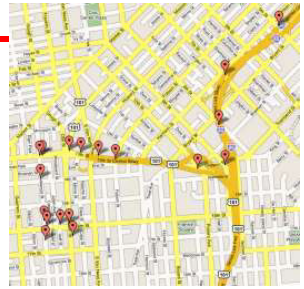
- If a node with a negative incident edge were to be added late to the cloud, it could mess up distances for vertices already in the cloud.



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

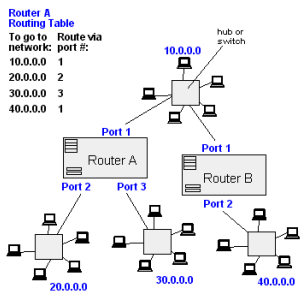
Dijkstra's Applications

- Network routing protocols - Open Shortest Path First
- It is used in geographical Maps.
- A* Algorithm in graph traversals.(AI)



From Computer Desktop Encyclopedia
© 1998 The Computer Language Co. Inc.

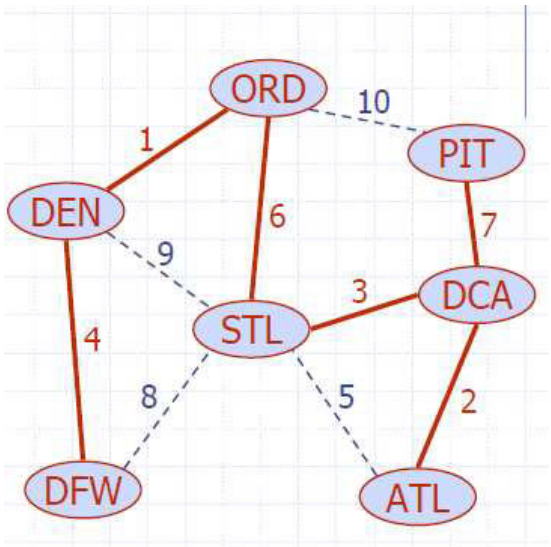
- Applied example
- https://prezi.com/po0cka9lrrqd/applied-example-of-dijkstras-algorithm/



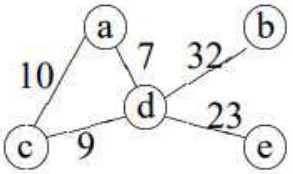
Minimum Spanning Tree

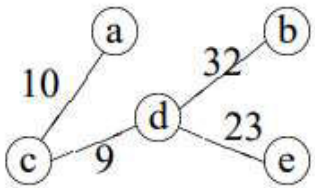
- Spanning subgraph
  - Subgraph of a graph G containing all the vertices of G
- Spanning tree
  - Spanning subgraph that is itself a tree
- Minimum spanning tree (MST)
  - Spanning tree of a weighted graph with minimum total edge weight
- Applications
  - Communications networks
  - Transportation networks

Minimum Spanning Tree

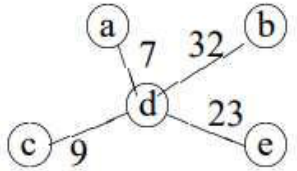


Minimum Spanning Tree
Prim's Algorithm

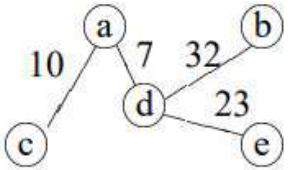




Tree 1. w=74



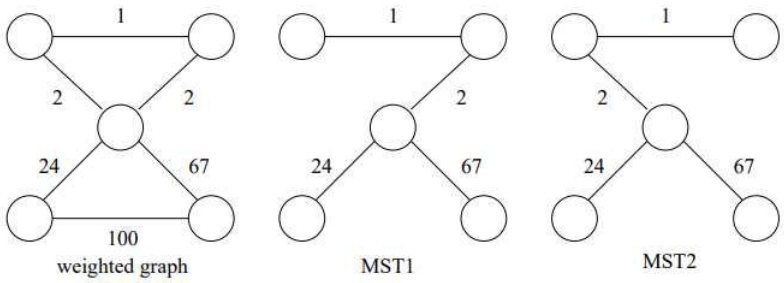
Tree 2, w=71



Tree 3, w=72

Minimum Spanning Tree
Prim's Algorithm

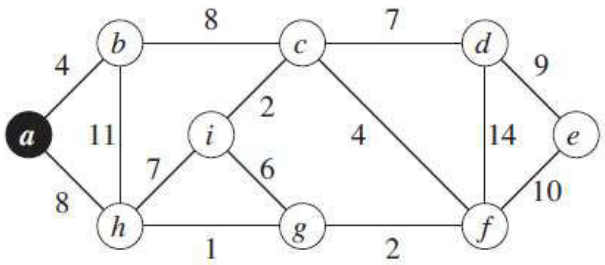
- The minimum spanning tree may not be unique. However, if the weights of all the edges are pairwise distinct, it is indeed unique



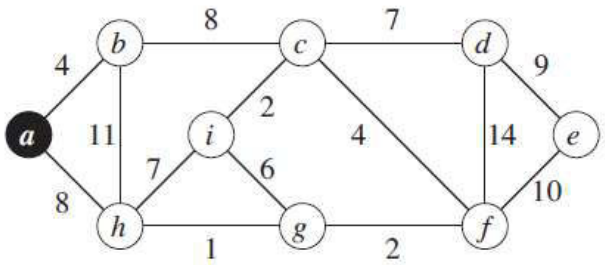
Minimum Spanning Tree
Prim's Algorithm

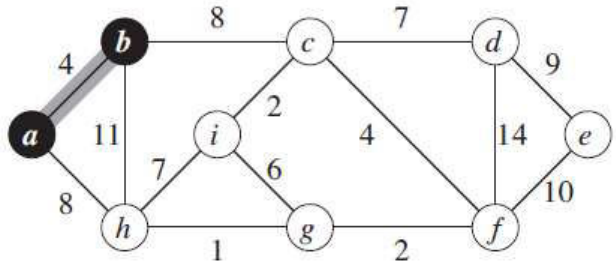
- Prim's algorithm is a greedy algorithm that finds a minimum spanning tree for a weighted undirected graph.
- This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized.
- At each step:
  - We add to the cloud the vertex ***u*** outside the cloud with the smallest distance label
  - We update the labels of the vertices adjacent to ***u***

Example

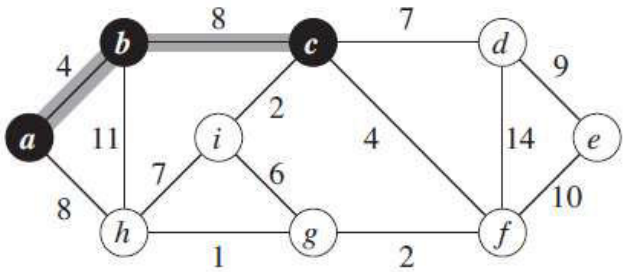


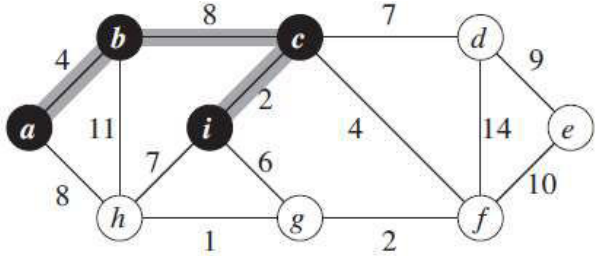
Example



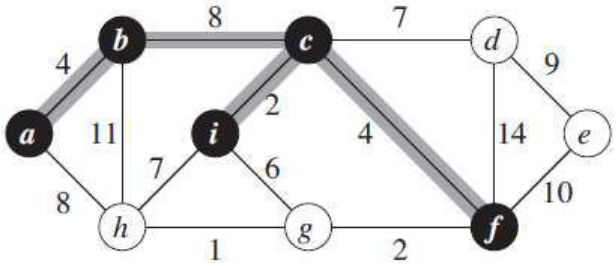


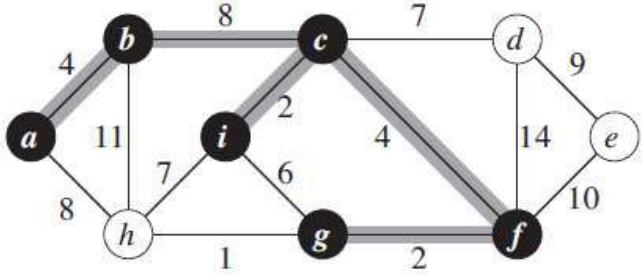
Example



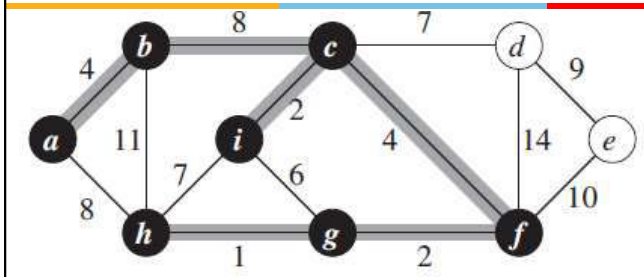


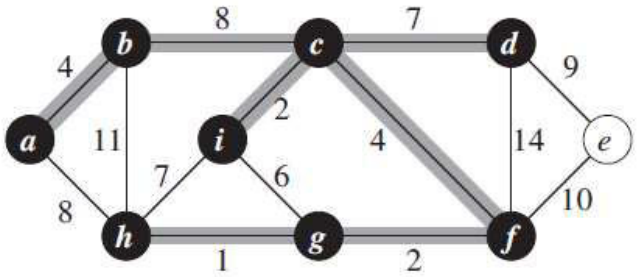
Example



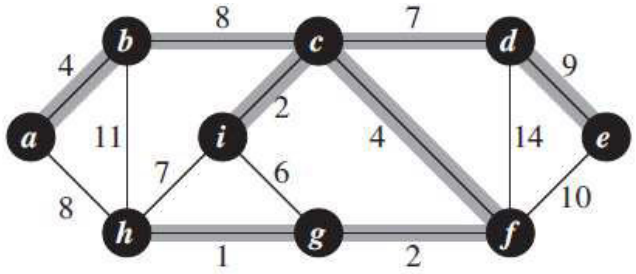


Example





Example



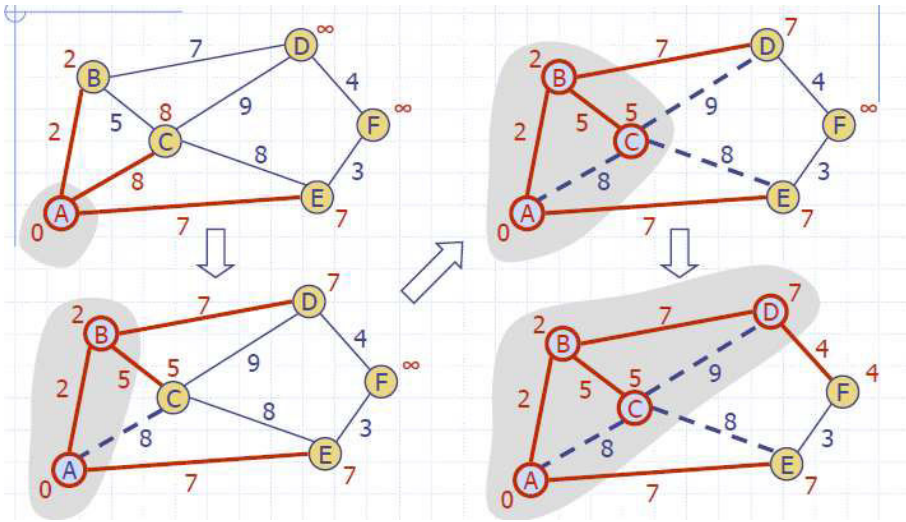
Minimum Spanning Tree
Prim's Algorithm

```
MST-PRIM(G, w, r)
 1  for each u ∈ G.V
 2      u.key = ∞
 3      u.π = NIL
 4  r.key = 0
 5  Q = G.V
 6  while Q ≠ ∅
 7      u = EXTRACT-MIN(Q)
 8      for each v ∈ G.Adj[u]
 9          if v ∈ Q and w(u, v) < v.key
10              v.π = u
11              v.key = w(u, v)
```

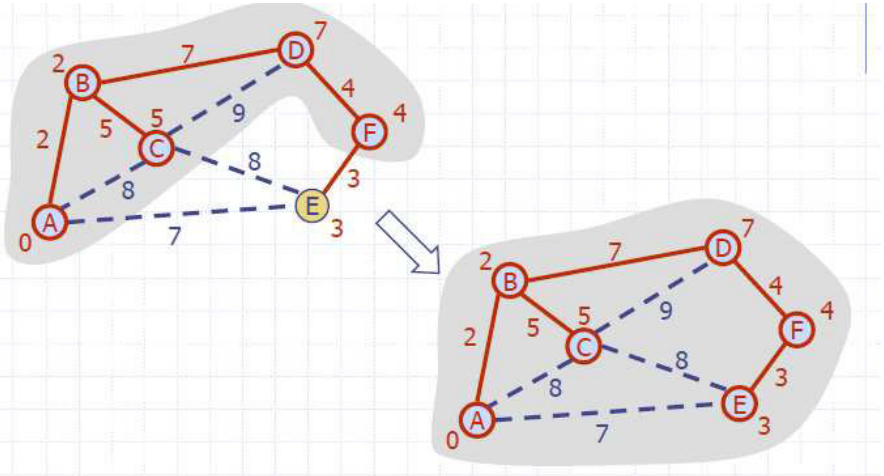
Prims's Algorithm-Analysis

- Similar to Dijkstra's,Prim's alogorithm can be implemented more efficiently by priority queue
  - Initialization O(|V|) using O(|V|) buildHeap
  - While loop O(|V|)
  - Find and remove min distance vertices = O(log |V|) using O(log |V|) deleteMin
  - Taken together that part of the loop and the calls to ExtractMin take O(VlogV) time
  - Potentially |E| updates:The for loop is executed once for each edge in the graph (E times), and within the for loop ,the update costs O(log |V|) using decreaseKey
  - Total time O(|V|log|V| + |E|log|V|) = O ((V+E)log V)
  - Since graph is connected, number of edges should be atleast n-1 is m>=n-1 ***ie.*** |V| = O(|E|) assuming a connected graph
  - Hence **the total time=O(|E| log |V|)**

Minimum Spanning Tree Prim's Algorithm-Example



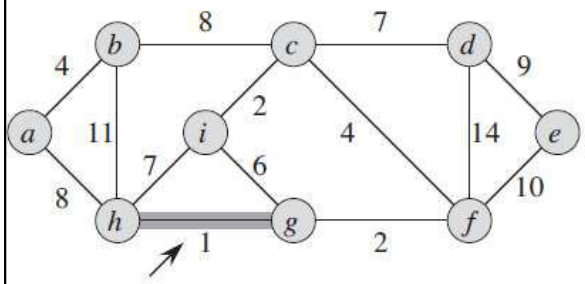
Minimum Spanning Tree Prim's Algorithm



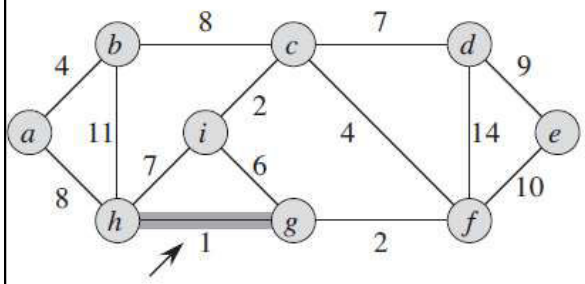
Minimum Spanning Tree **Kruskal's Algorithm**

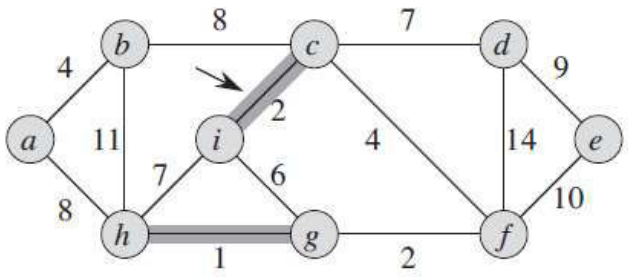
- It builds the MST in forest.
- Initially, each vertex is in its own tree in forest.
- Then, algorithm consider each edge in turn, order by increasing weight.
- If an edge (u, v) connects two different trees, then (u, v) is added to the set of edges of the MST, and two trees connected by an edge (u, v) are merged into a single tree
- If an edge (u, v) connects two vertices in the same tree, then edge (u, v) is discarded.

Minimum Spanning Tree
Kruskal's Algorithm-Example

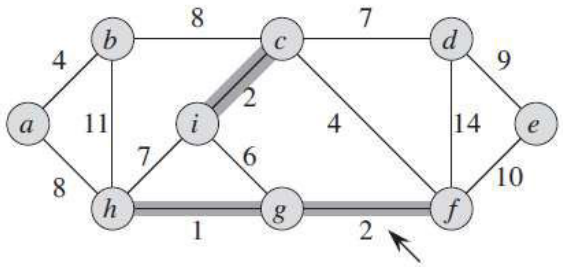


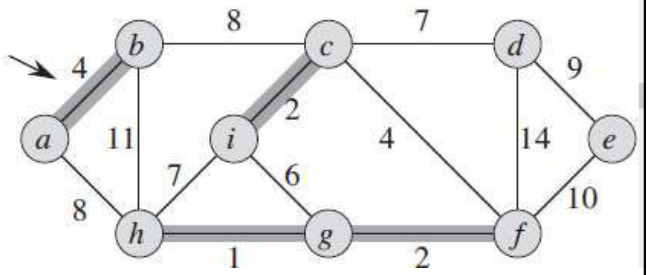
Minimum Spanning Tree
Kruskal's Algorithm-Example



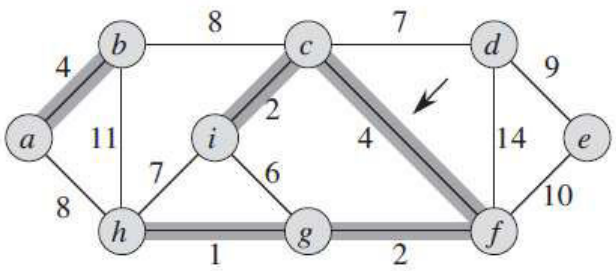


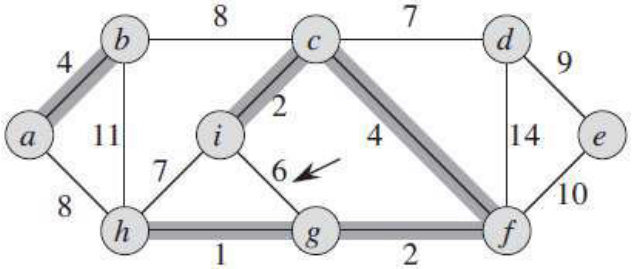
Minimum Spanning Tree
Kruskal's Algorithm-Example



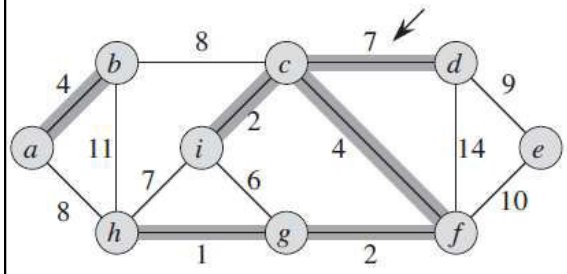


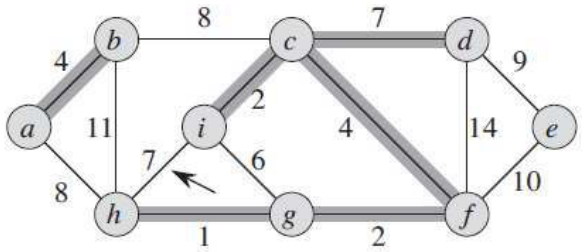
Minimum Spanning Tree
Kruskal's Algorithm-Example





Minimum Spanning Tree
Kruskal's Algorithm-Example





Minimum Spanning Tree
Kruskal's Algorithm

- A priority queue stores the edges-
  - **Key: weight**
  - **Element: edge**
- We can implement the priority queue Q using a heap.
- Thus, we can initialize Q in **O(E) time using bottom-up heap construction**
- In addition, at each iteration of the **while** loop, we can remove a minimum-weight edge in O(log E) time.
- Taken together that part of the loop and the calls to removeMin take O(ElogE) time
- Contd…

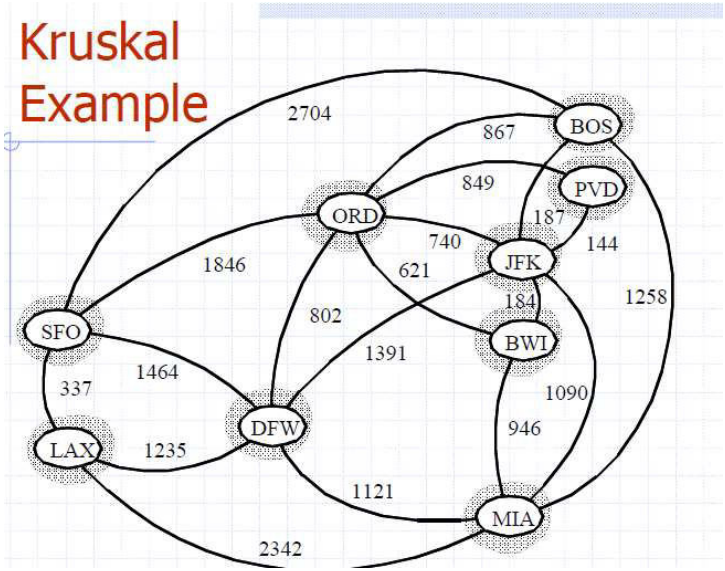
Minimum Spanning Tree
Kruskal's Algorithm

Algorithm *KruskalMST(G)*
  for each vertex *V* in *G* **do**
    define a *Cloud(v)* of ← {*v*}
  let *Q* be a priority queue.
  Insert all edges into *Q* using their weights as the key
  T ← ∅
  while *T* has fewer than *n*-1 edges **do**
    edge *e* = *T.removeMin()*
    Let *u*, *v* be the endpoints of *e*
    if *Cloud(v)* ≠ *Cloud(u)* **then**
      Add edge *e* to *T*
      Merge *Cloud(v)* and *Cloud(u)*
  return *T*

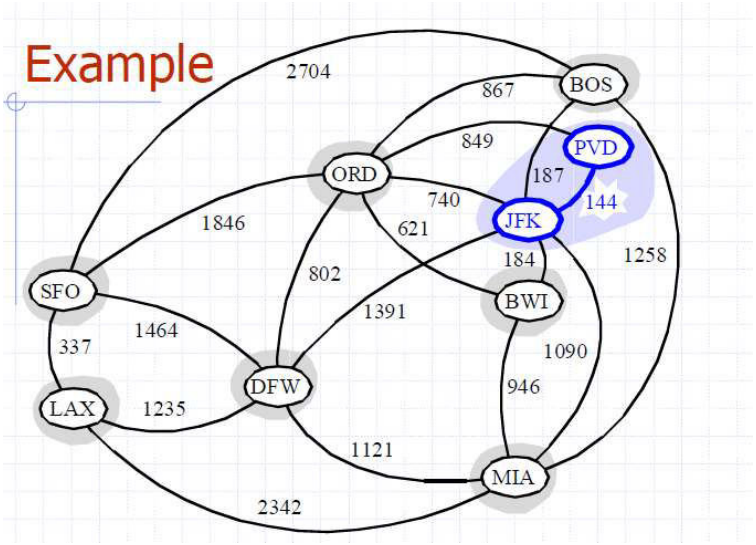
Minimum Spanning Tree
Kruskal's Algorithm-Analysis

- Here final tree will have n-1 edges
  - So n union operations need to be performed-O(n log n) [Cost of find included]
  - Since graph is connected, number of edges should be atleast n-1 is m>=n-1
  - So cost of union operations=O(m log m)=O(E log E)
- Thus, the total time spent performing priority queue operations is no more than O(E log E).
  - The number of edges in a simple graph m =n(n – 1)/2.
  - ie **m is atmost n^2** ie **log m =2log n ie lg |E| = O(lg |V|)**
- Thus, the total time spent performing priority queue operations is O(E log V). [Same as Prim's]

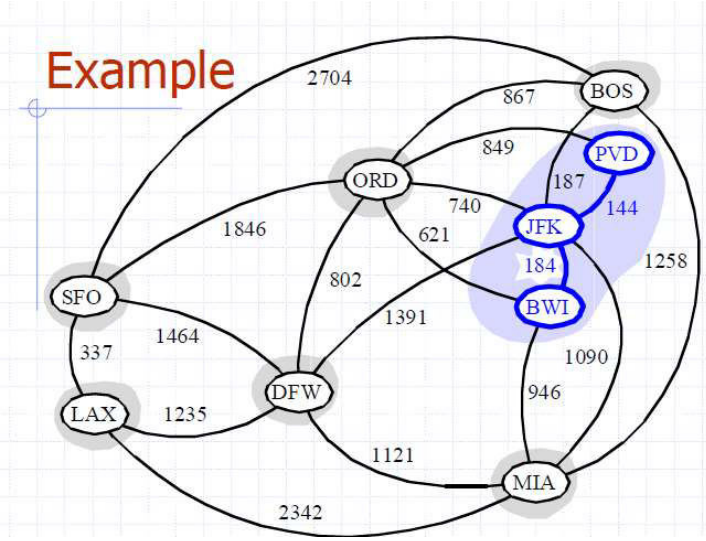
Minimum Spanning Tree



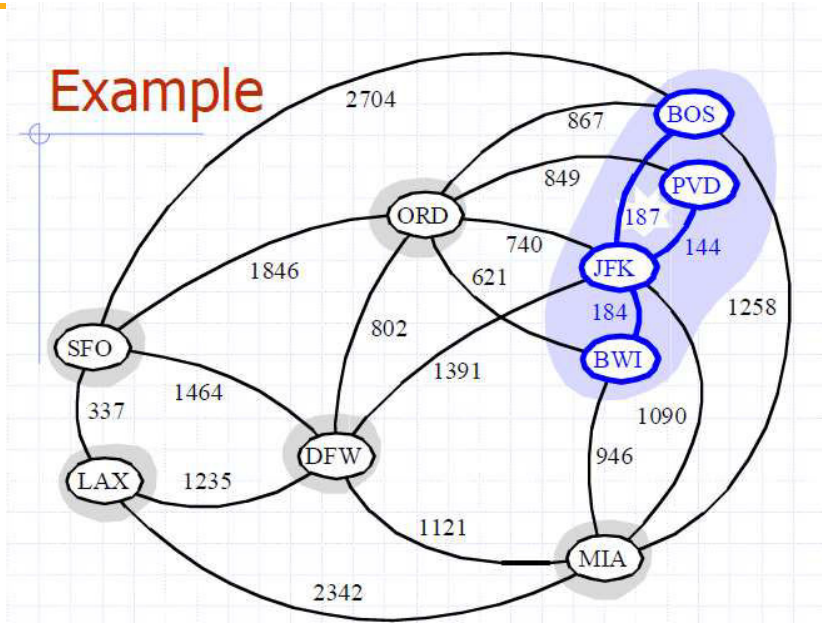
Minimum Spanning Tree



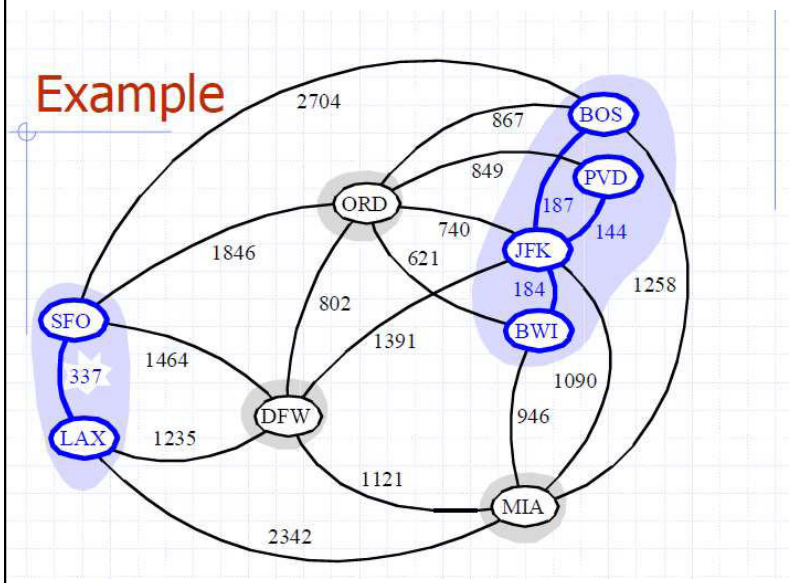
Minimum Spanning Tree



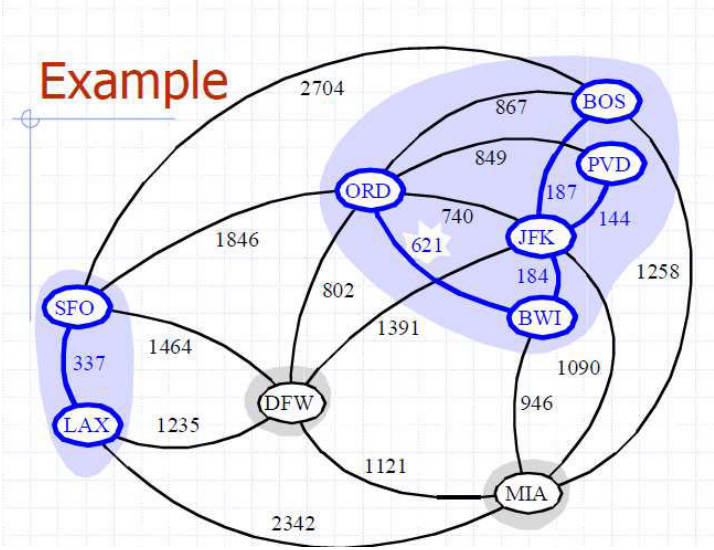
Minimum Spanning Tree



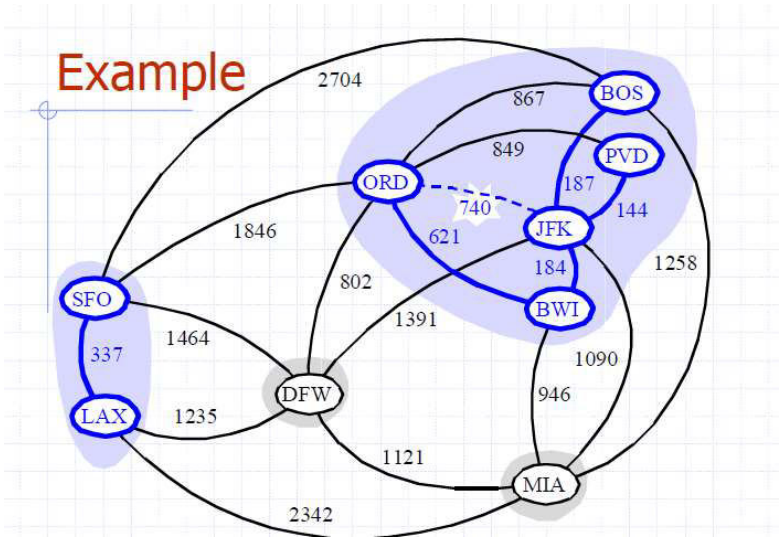
Minimum Spanning Tree



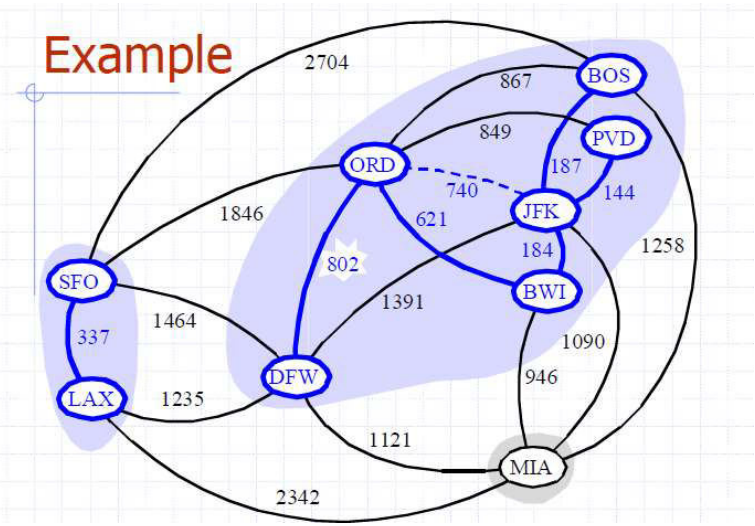
Minimum Spanning Tree



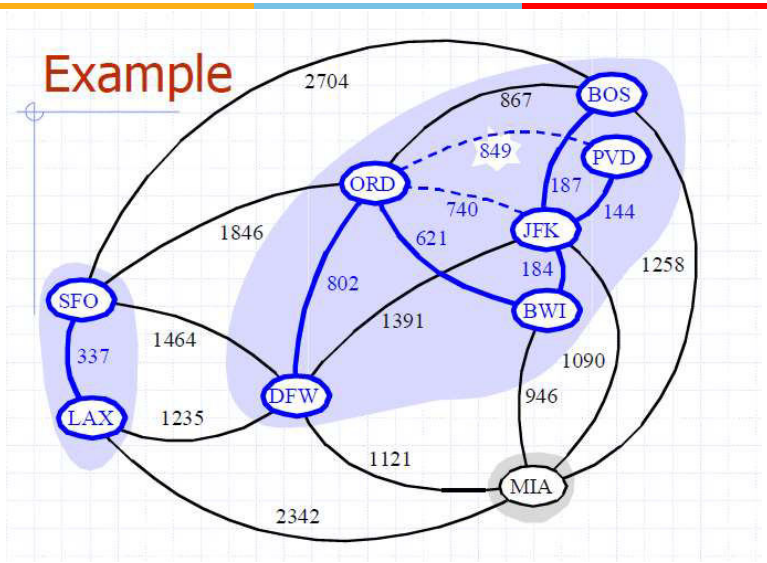
Minimum Spanning Tree



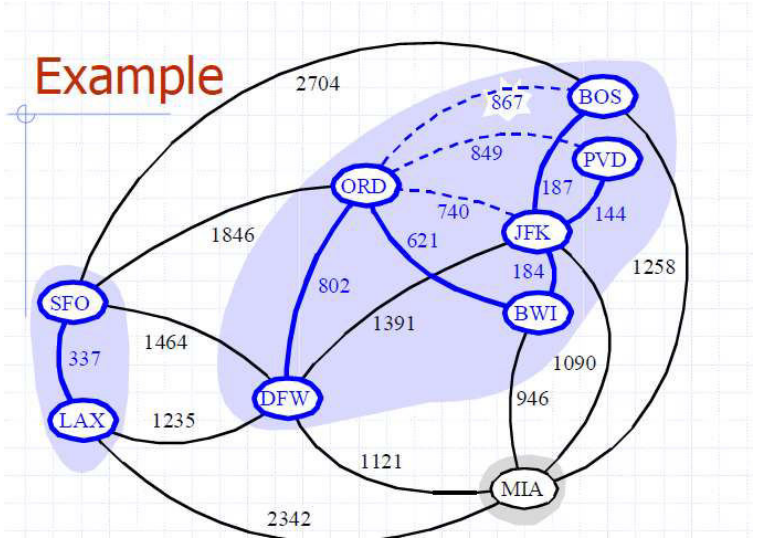
Minimum Spanning Tree



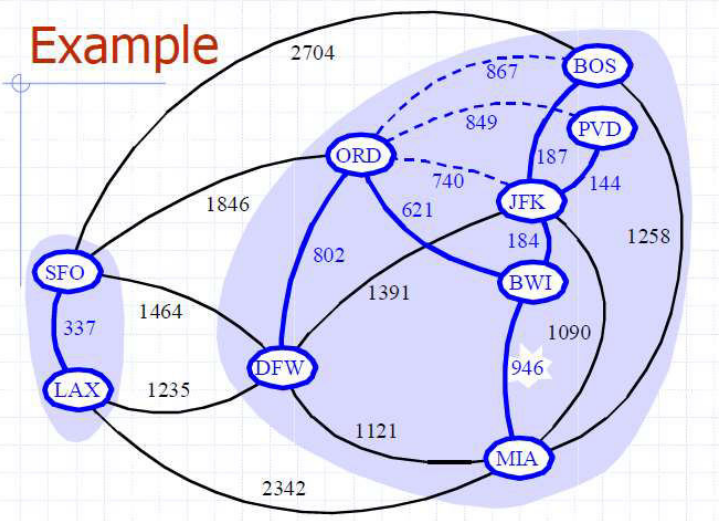
Minimum Spanning Tree



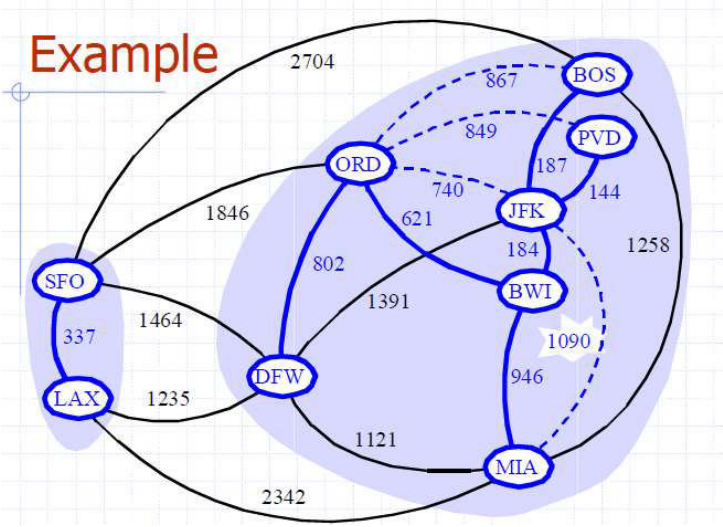
Minimum Spanning Tree



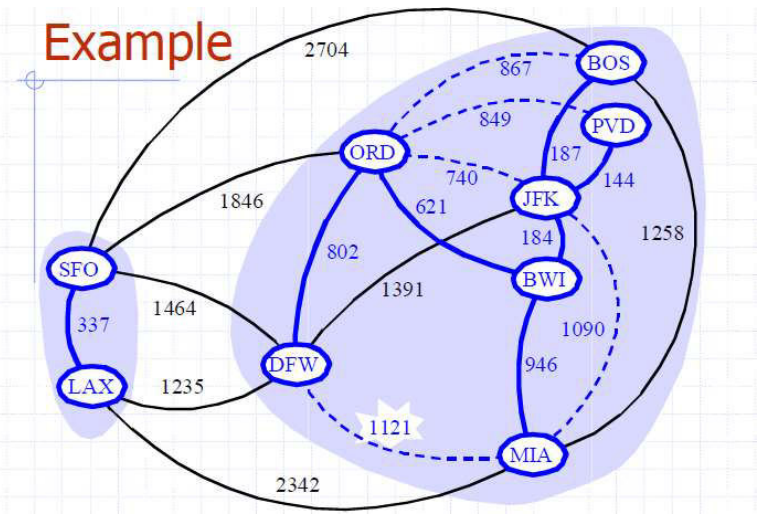
Minimum Spanning Tree



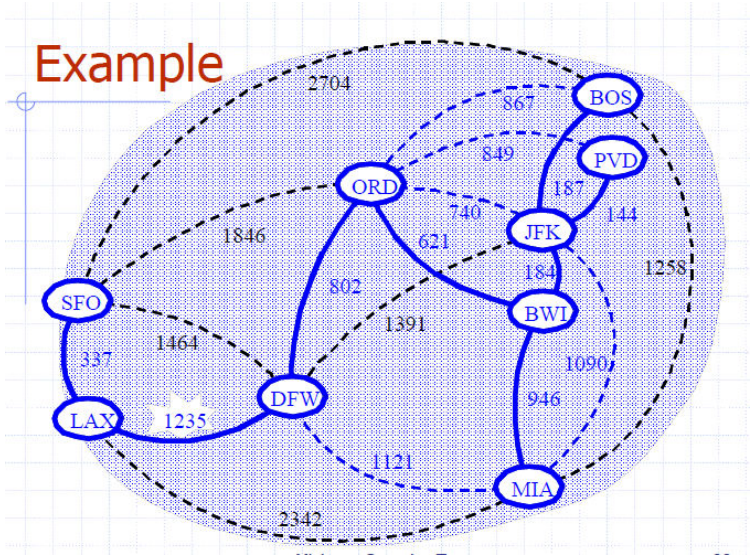
Minimum Spanning Tree



Minimum Spanning Tree



Minimum Spanning Tree



PRIMS'S Vs KRUSKAL'S ALGORITHM

- **KRUSKAL's**
- Select the shortest edge in a network
- Select the next shortest edge which does not create a cycle
- Repeat step 2 until all vertices have been connected
- Kruskal's begins with forest and merge into tree.
- **PRIM's**
- Select any vertex
- Select the shortest edge connected to that vertex
- Select the shortest edge connected to any vertex already connected
- Prim's always stays as a tree.

PRIMS'S Vs KRUSKAL'S ALGORITHM

- **KRUSKAL's**
- Can be used when graph is sparse(less edges)
- **PRIM's**
- Can be used if the graph has more edges.

MST-Applications

- **Network design.**

– *telephone, electrical, hydraulic, TV cable, computer, road*
The standard application is to a problem like phone network design. You have a business with several offices; you want to lease phone lines to connect them up with each other; and the phone company charges different amounts of money to connect different pairs of cities. You want a set of lines that connects all your offices with a minimum total cost. It should be a spanning tree, since if a network isn't a tree you can always remove some edges and save money.

MST-Applications

- **Cluster analysis**
  k clustering problem can be viewed as finding an MST and deleting the k-1 most expensive edges.
- **Image registration and segmentation**
  - **Image segmentation** is the process of portioning image to components and its purpose is to decompose an image to significant and convenient regions and also extract a specific object from image: Image segmentation strives to partition a digital image into regions of pixels with similar properties, e.g. homogeneity.
  - **Image registration** is the process of transforming different sets of data into one coordinate system. Data may be multiple photographs, data from different sensors, times, depths, or viewpoints. Registration is necessary in order to be able to compare or integrate the data obtained from these different measurements.

MST-Applications

- **Taxonomy**:
  - Taxonomy is the practice and science of classification
- **Feature extraction**
  - In machine learning, pattern recognition and in image processing, feature extraction starts from an initial set of measured data and builds derived values (features) intended to be informative and non-redundant, facilitating the subsequent learning and generalization steps, and in some cases leading to better human interpretations
  - When the input data to an algorithm is too large to be processed and it is suspected to be redundant (e.g. the same measurement in both feet and meters, or the repetitiveness of images presented as pixels), then it can be transformed into a reduced set of features (also named a feature vector). Determining a subset of the initial features is called feature selection.

MST-Applications

- Regionalization of socio-geographic areas, the grouping of areas into homogeneous, contiguous regions.
- Comparing ecotoxicology data.:
  - Ecotoxicology is the study of the effects of toxic chemicals on biological organisms, especially at the population, community, ecosystem, and biosphere levels.
- Topological observability in power systems.
- Measuring homogeneity of two-dimensional materials.

MST-Problem -HW

MST QN

PLAN

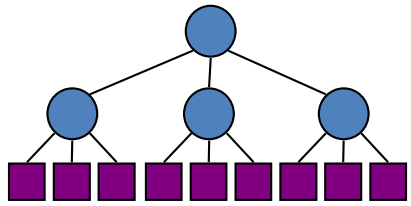
Sessions(#)	List of Topic Title	Text/Ref Book/external resource
11	**Divide and Conquer -** Design Principles and Strategy, Analysing Divide and Conquer Algorithms, Integer Multiplication Problem **Sorting Problem** - Merge Sort Algorithm	**T1: 5.2, 4.1, 4.3**

Data Structures and Algorithms Design

BITS Pilani
Hyderabad Campus

Divide-and-Conquer

- **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 sub problems recursively
  - Conquer: combine the solutions for S_1, S_2, …, into a solution for S
- The base case for the recursion are sub problems of constant size
- Analysis can be done using **recurrence equations**



Merge Sort

- Merge-sort on an input sequence S with n elements consists of three steps:
  - Divide: partition S into two sequences S_1 and S_2 of about $n/2$ elements each
  - Recur: recursively sort S_1 and S_2
  - Conquer: merge S_1 and S_2 into a unique sorted sequence

Merge Sort

38	27	43	3	9	82	10

Merge Sort

MergeSort(arr[], l, r)
If r > l

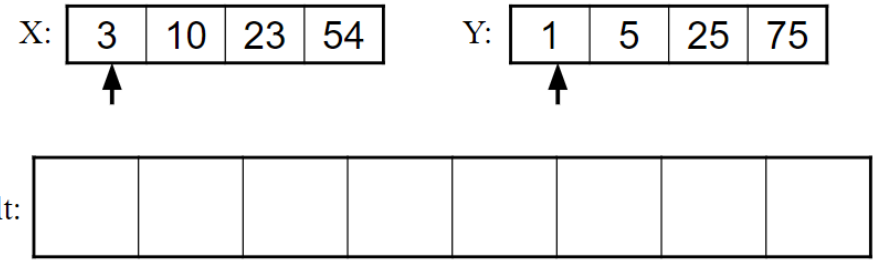
1. ***Find the middle point to divide the array into two halves:***
   middle m = (l+r)/2
2. ***Call MergeSort for first half:***
   Call mergeSort(arr, l, m)
3. ***Call MergeSort for second half:***
   Call mergeSort(arr, m+1, r)
4. ***Merge the two halves sorted in step 2 and 3***
   Call merge(arr, l, m, r)

Merge Sort

- The conquer step of merge-sort consists of merging two sorted sequences A and B into a sorted sequence S containing the union of the elements of A and B
- Merging two sorted sequences, each with $n/2$ elements and implemented by means of a doubly linked list, takes $O(n)$ time

Merge Sort

Merging two sorted lists, merge(A,B)



Merge Sort

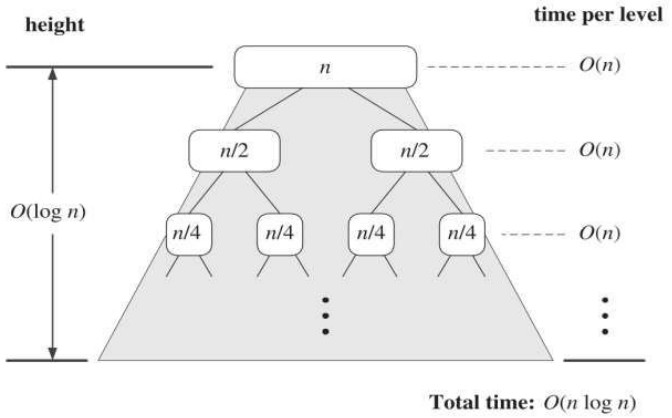
Algorithm *merge*(*A*, *B*)

Input sequences *A* and *B* with $n/2$ elements each

Output sorted sequence of $A \cup B$

```
S ← empty sequence
while ¬A.isEmpty() ∧ ¬B.isEmpty()
    if A.first().element() < B.first().element()
        S.insertLast(A.remove(A.first()))
    else
        S.insertLast(B.remove(B.first()))
while ¬A.isEmpty()          S.insertLast(A.remove(A.first()))
while ¬B.isEmpty()          S.insertLast(B.remove(B.first()))
return S
```

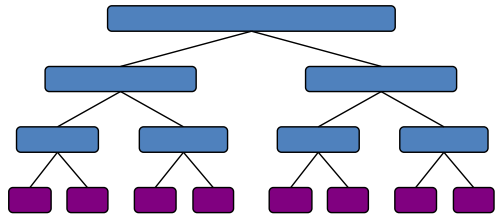

Analysis of Merge-Sort



Analysis of Merge-Sort

- An execution of merge-sort is depicted by a binary tree
- The height h of the merge-sort tree is $O(\log n)$
  - at each recursive call we divide in half the sequence,
- The overall amount or work done at the nodes of depth i is $O(n)$
  - we partition and merge 2^i sequences of size $n/2^i$
  - we make 2^{i+1} recursive calls
- Thus, the total running time of merge-sort is $O(n \log n)$

depth	#seqs	size
0	1	n
1	2	$n/2$
i	2^i	$n/2^i$
…	…	…



Master Method

- 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 $\delta < 1$.

Master Method

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn & \text{if } n \geq 2 \end{cases}$$

Using Master method to solve

- Case 2 applies
- **$T(n)=\theta(n \log n)$**

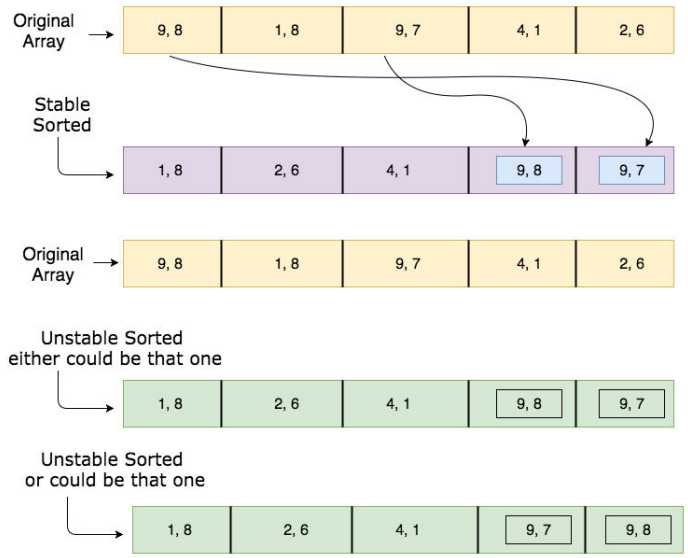
Merge-Sort Properties

- **Not Adaptive** : Running time doesn't change with data pattern
  - algorithm will re-order every single item in the list even if it is already sorted.
- **Stable/ Unstable** : Both implementations are possible .
- **Not Incremental** : Does not sort one by one element in each pass.
- **Not online** : Need all data to be in memory at the time of sorting.
- **Not in place** : It need O(n) extra space to sort two sub list of size(n/2).

Merge-Sort Properties

- **Merge sort is a stable sort**
- A sorting algorithm is said to be **stable** if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted

Merge-Sort Properties



Merge-Sort Applications

- **Merge sort is often the best choice for sorting a linked list.**
  - Linked list nodes may not be adjacent in memory. In linked list, we can insert items in the end in O(1) extra space and O(1) time.
  - In linked list to access i'th index, we have to travel each and every node from the head to i'th node as we don't have continuous block of memory. Merge sort accesses data sequentially and the need of random access is low.

Merge-Sort Applications

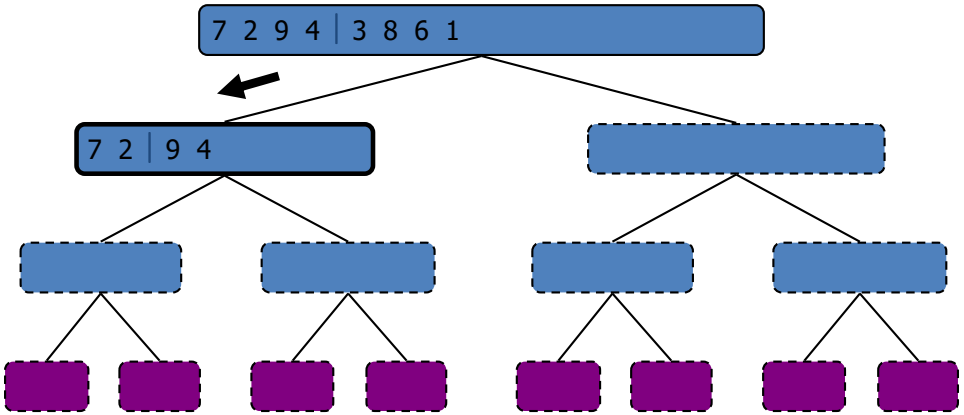
- **External Sorting:**
  - External sorting is a class of sorting algorithms that can handle massive amounts of data. External sorting is required when the data being sorted do not fit into the main memory of a computing device (usually RAM) and instead they must reside in the slower external memory, usually a hard disk drive.
  - External merge sort uses a hybrid sort-merge strategy. In the sorting phase, chunks of data small enough to fit in main memory are read, sorted, and written out to a temporary file. In the merge phase, the sorted subfiles are combined into a single larger file.

Merge-Sort Applications

- In Java, the Arrays.sort() methods use merge sort
- The Linux kernel uses merge sort for its linked lists
- Python uses Timsort, another tuned hybrid of merge sort and insertion sort, that has become the standard sort algorithm in Java SE 7

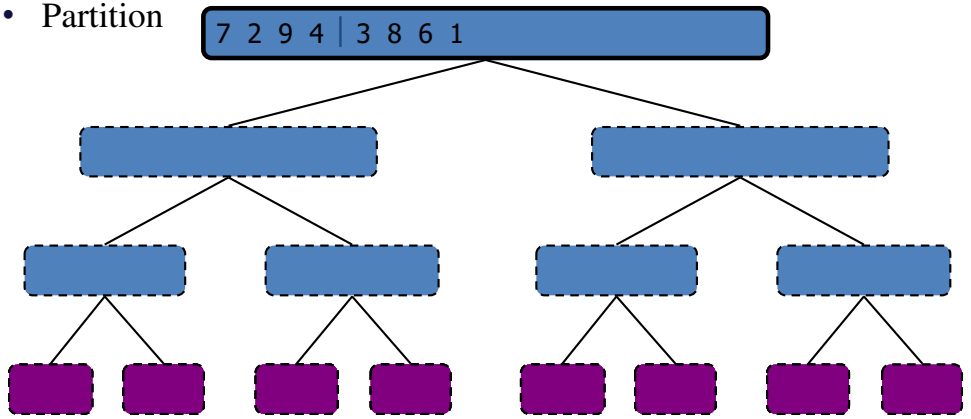
Execution Example (cont.)

- Recursive call, partition



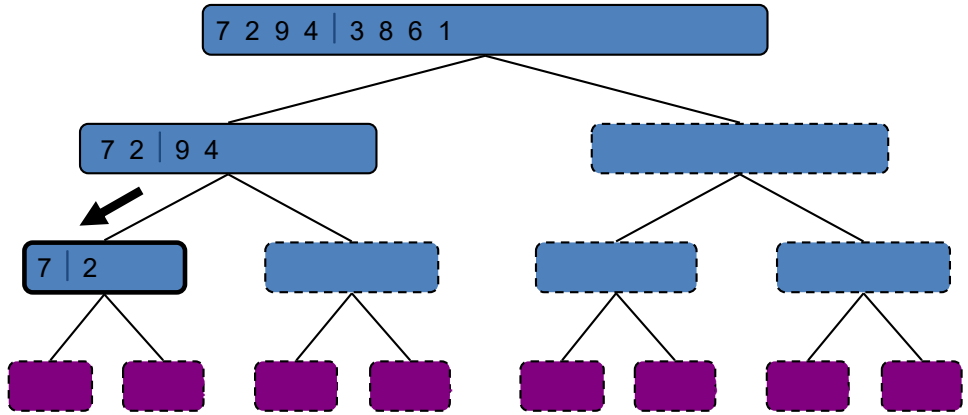
Execution Example-1

- Partition



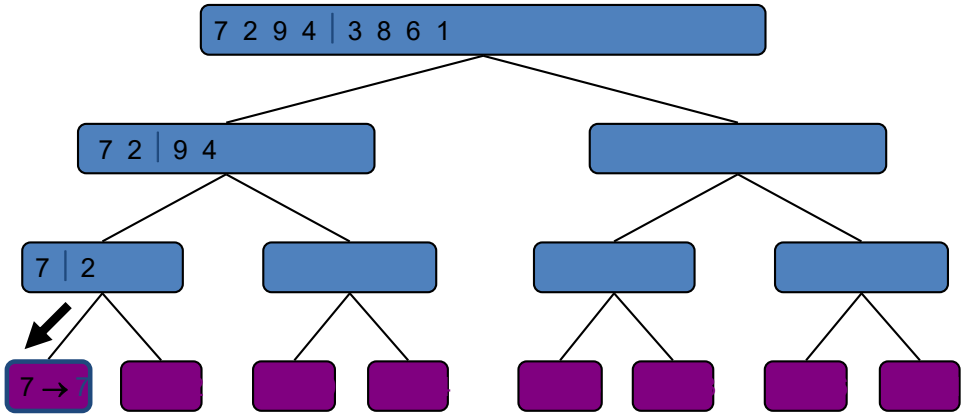
Execution Example (cont.)

- Recursive call, partition



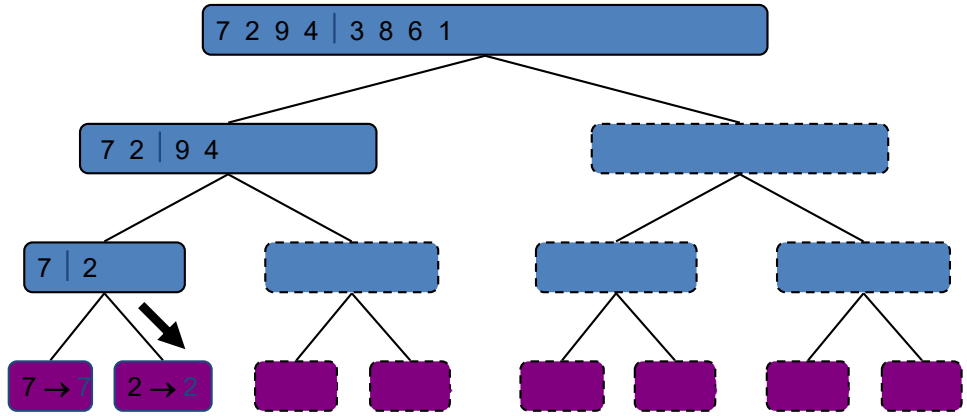
Execution Example (cont.)

- Recursive call, base case



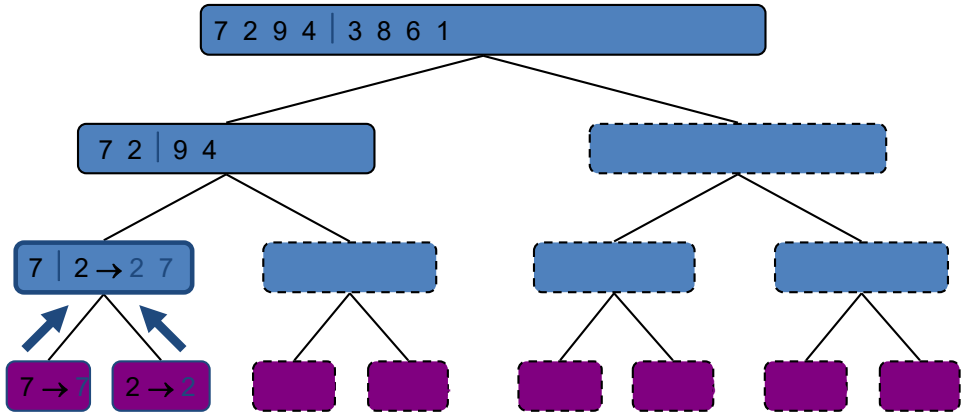
Execution Example (cont.)

- Recursive call, base case



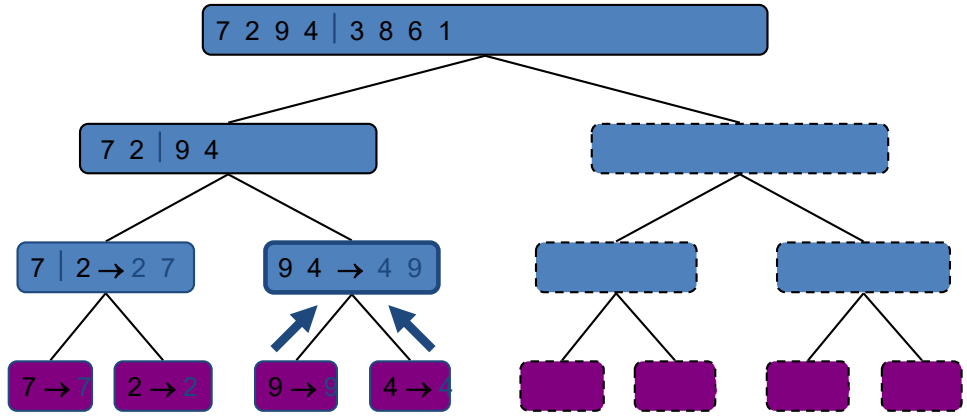
Execution Example (cont.)

- Merge



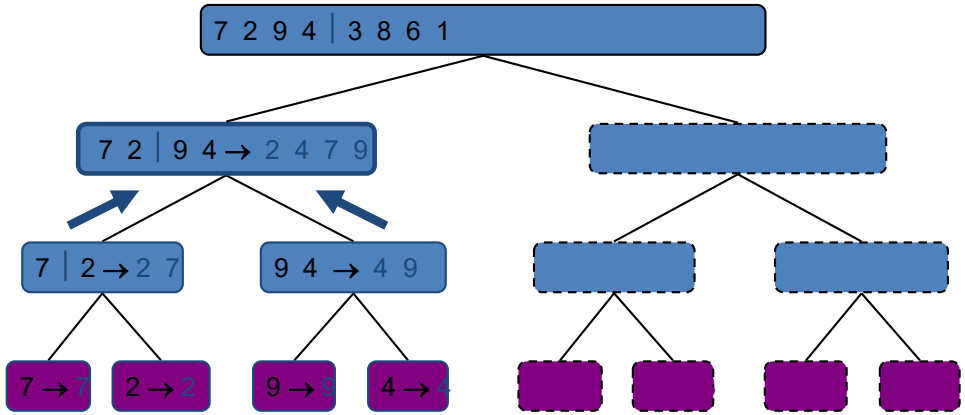
Execution Example (cont.)

- Recursive call, …, base case, merge



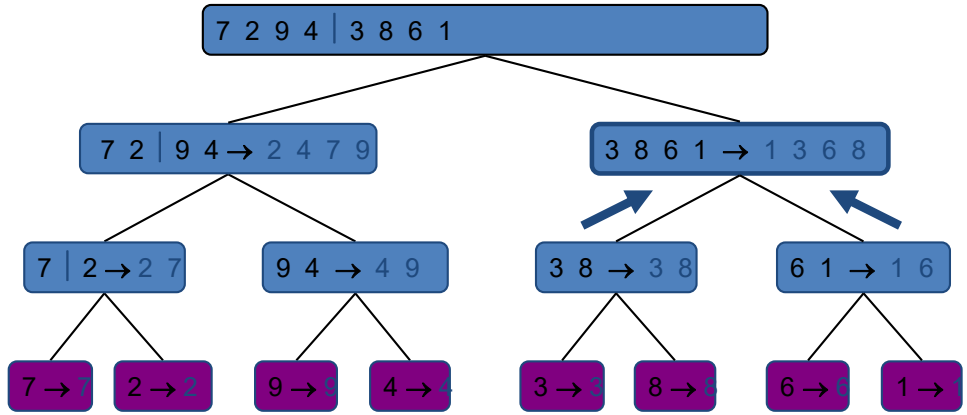
Execution Example (cont.)

- Merge



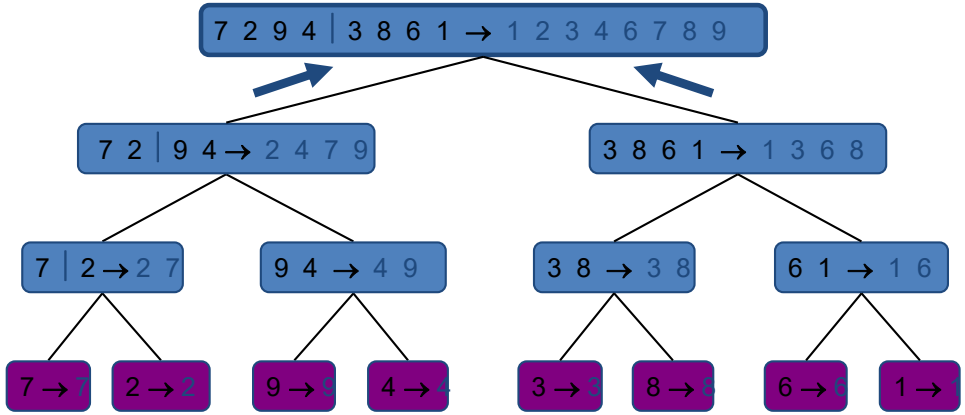
Execution Example (cont.)

- Recursive call, …, merge, merge

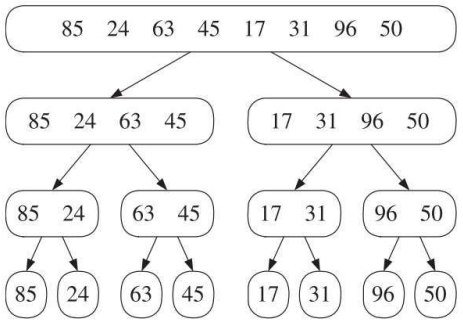


Execution Example (cont.)

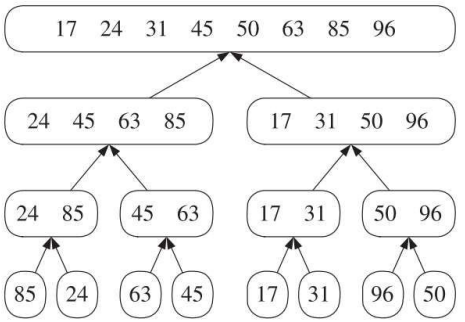
- Merge



Merge sort - Example -2



Input sequence processed at each node - Divide



Output sequence generated at each node - Conquer

Integer Multiplication

- The problem of multiplying big integers, that is, integers represented by a large number of bits that cannot be handled directly by the arithmetic unit of a single processor.

Integer Multiplication

- Given two big integers I and J represented with n bits each, we can easily compute I + J and I - J in O(n) time.

Integer Multiplication

- Efficiently computing the product I · J using the common grade-school algorithm requires, however, O (n^2) time.

Integer Multiplication-Divide and Conquer

- Divide and Conquer to Multiply - Attempt- #1
  - Let us represent I and J as below.
    - Attempt to rewrite the multiplication of I and J in terms of their components.
      - That is, this gives raise to recursion

Integer Multiplication-Divide and Conquer

- Divide and Conquer to Multiply - Attempt- #1

$$I = I_h 2^{n/2} + I_l$$
$$J = J_h 2^{n/2} + J_l$$

Integer Multiplication-Divide and Conquer

– We can then define I*J by multiplying the parts and adding:

$$I = I_h 2^{n/2} + I_l$$
$$J = J_h 2^{n/2} + J_l$$

$$I * J = (I_h 2^{n/2} + I_l) * (J_h 2^{n/2} + J_l)$$
$$= I_h J_h 2^n + I_h J_l 2^{n/2} + I_l J_h 2^{n/2} + I_l J_l$$

Integer Multiplication-Divide and Conquer

$$I * J = (I_h 2^{n/2} + I_l) * (J_h 2^{n/2} + J_l)$$
$$= I_h J_h 2^n + I_h J_l 2^{n/2} + I_l J_h 2^{n/2} + I_l J_l$$

Multiplication of 2 n bit numbers is now broken down into

- 4 Multiplications of n/2 bit numbers
- Plus 3 Additions

Multiplying any binary numbers by any arbitrary power of 2 is just a shift operation of bits

T(n) = 4T(n/2) + n,
implies T(n) is $O(n^2)$

Integer Multiplication-Divide and Conquer

- $O(n^2)$ is not an improvement indeed.
  - But we are able to multiply large integers in terms of smaller ones, now !
- We need to compute the following with lesser # of multiplication

$$I \cdot J = I_h J_h 2^n + I_l J_h 2^{n/2} + I_h J_l 2^{n/2} + I_l J_l$$

Integer Multiplication-Divide and Conquer

- Let us try to rewrite the following with 3 multiplications

$$I \cdot J = I_h J_h 2^n + I_l J_h 2^{n/2} + I_h J_l 2^{n/2} + I_l J_l$$

- Let $P1=(I_h+I_l) * (J_h+J_l)$
  $=I_hJ_h + I_hJ_l + I_lJ_h + I_lJ_l$
  $P2= I_hJ_h$
  $P3= I_lJ_l$
  $P1-P2-P3= I_hJ_l + I_lJ_h$

Now

$I*J = P2 * 2^n + [P1-P2-P3]* 2^{n/2} + P3$

Integer Multiplication-Divide and Conquer

- Let us try to rewrite the following with 3 multiplications

$$I \cdot J = I_h J_h 2^n + I_l J_h 2^{n/2} + I_h J_l 2^{n/2} + I_l J_l$$

- Let $P1=(I_h+I_l) * (J_h+J_l)$
  $=I_hJ_h + I_hJ_l + I_lJ_h + I_lJ_l$
  $P2= I_hJ_h$
  $P3= I_lJ_l$
  $P1-P2-P3= I_hJ_l + I_lJ_h$

Now

$I*J = P2 * 2^n + [P1-P2-P3]* 2^{n/2} + P3$

T(n) = 3T(n/2) + n,
By Master Theorem.
T(n) is $O(n^{\log_2 3})$,
Thus, T(n) is $O(n^{1.585})$.

Algorithm

Input: Positive integers x and y, in binary
Output: Their product

$n =$ max(size of x, size of y)
if $n = 1$: return xy

$x_L, x_R =$ leftmost $\lceil n/2 \rceil$, rightmost $\lfloor n/2 \rfloor$ bits of x
$y_L, y_R =$ leftmost $\lceil n/2 \rceil$, rightmost $\lfloor n/2 \rfloor$ bits of y

$P_1 = \text{multiply}(x_L, y_L)$
$P_2 = \text{multiply}(x_R, y_R)$
$P_3 = \text{multiply}(x_L + x_R, y_L + y_R)$
return $P_1 \times 2^n + (P_3 - P_1 - P_2) \times 2^{n/2} + P_2$

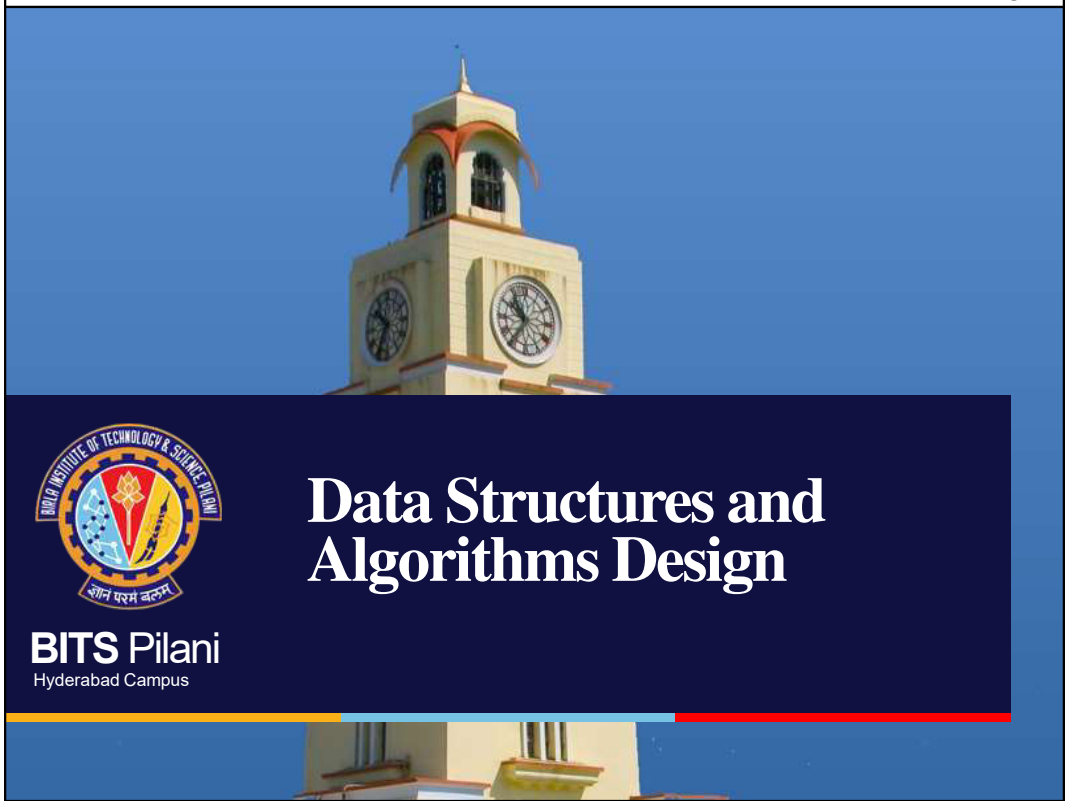
An Improved Integer Multiplication Algorithm

- Algorithm: Multiply two n-bit integers I and J.
- **O(n log n)** , by using a more complex divide-and-conquer algorithm called the Fast Fourier transform

Example-Decimal

Application

- Multiplying big integers has applications to data security, where big integers are used in encryption schemes.
  - More specifically, some important cryptographic algorithms such as RSA critically depend on the fact that prime factorization of large numbers takes a long time. Basically you have a "public key" consisting of a product of two large primes used to encrypt a message, and a "secret key" consisting of those two primes used to decrypt the message. You can make the public key public, and everyone can use it to encrypt messages to you, but only you know the prime factors and can decrypt the messages. Everyone else would have to factor the number, which takes too long to be practical, given the current state of the art of number theory.

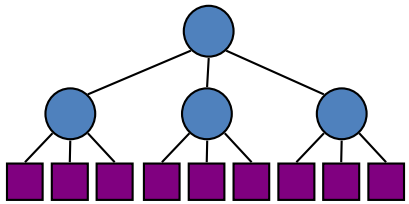


PLAN

Sessions(#)	List of Topic Title	Text/Ref Book/external resource
11	**Divide and Conquer -** Design Principles and Strategy, Analysing Divide and Conquer Algorithms, Integer Multiplication Problem **Sorting Problem** - Merge Sort Algorithm	**T1: 5.2, 4.1, 4.3**

Divide-and-Conquer

- **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 sub problems recursively
  - Conquer: combine the solutions for S_1, S_2, …, into a solution for S
- The base case for the recursion are sub problems of constant size
- Analysis can be done using **recurrence equations**



Merge Sort

- Merge-sort on an input sequence S with n elements consists of three steps:
  - Divide: partition S into two sequences S_1 and S_2 of about $n/2$ elements each
  - Recur: recursively sort S_1 and S_2
  - Conquer: merge S_1 and S_2 into a unique sorted sequence

Merge Sort

38	27	43	3	9	82	10

Merge Sort

MergeSort(arr[], l, r)
If r > l

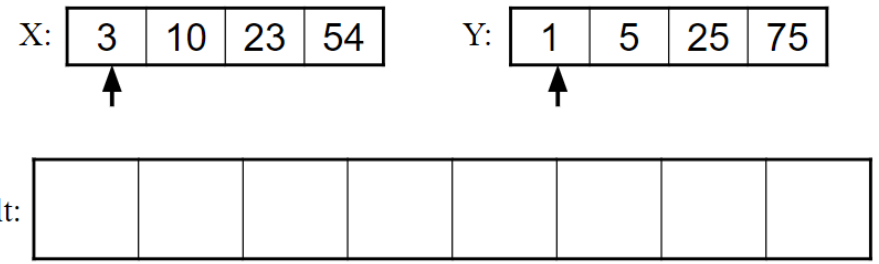
1. ***Find the middle point to divide the array into two halves:***
   middle m = (l+r)/2
2. ***Call MergeSort for first half:***
   Call mergeSort(arr, l, m)
3. ***Call MergeSort for second half:***
   Call mergeSort(arr, m+1, r)
4. ***Merge the two halves sorted in step 2 and 3***
   Call merge(arr, l, m, r)

Merge Sort

- The conquer step of merge-sort consists of merging two sorted sequences ***A*** and ***B*** into a sorted sequence ***S*** containing the union of the elements of ***A*** and ***B***
- Merging two sorted sequences, each with $n/2$ elements and implemented by means of a doubly linked list, takes $O(n)$ time

Merge Sort

Merging two sorted lists, merge(A,B)

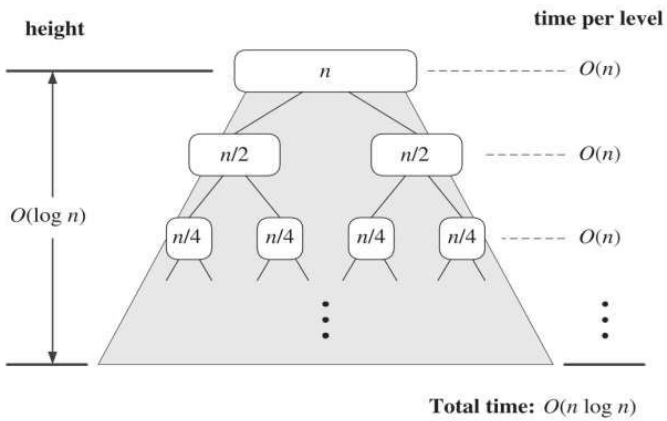


Merge Sort

Algorithm *merge*(*A*, *B*)
Input sequences *A* and *B* with $n/2$ elements each
Output sorted sequence of $A \cup B$

```
S ← empty sequence
while ¬A.isEmpty() ∧ ¬B.isEmpty()
    if A.first().element() < B.first().element()
        S.insertLast(A.remove(A.first()))
    else
        S.insertLast(B.remove(B.first()))
while ¬A.isEmpty()          S.insertLast(A.remove(A.first()))
while ¬B.isEmpty()          S.insertLast(B.remove(B.first()))
return S
```


Analysis of Merge-Sort

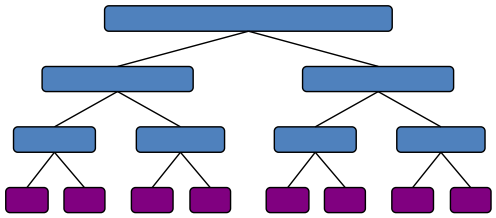


Total time: $O(n \log n)$

Analysis of Merge-Sort

- An execution of merge-sort is depicted by a binary tree
- The height ***h*** of the merge-sort tree is ***O***(log ***n***)
  - at each recursive call we divide in half the sequence,
- The overall amount or work done at the nodes of depth ***i*** is ***O***(***n***)
  - we partition and merge 2^i sequences of size $n/2^i$
  - we make 2^{i+1} recursive calls
- Thus, the total running time of merge-sort is ***O***(***n*** log ***n***)

depth	#seqs	size
0	1	n
1	2	$n/2$
i	2^i	$n/2^i$
…	…	…



Master Method

- 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 $\delta < 1$.

Master Method

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn & \text{if } n \geq 2 \end{cases}$$

Using Master method to solve

- Case 2 applies
- **T(n)=θ(n log n)**

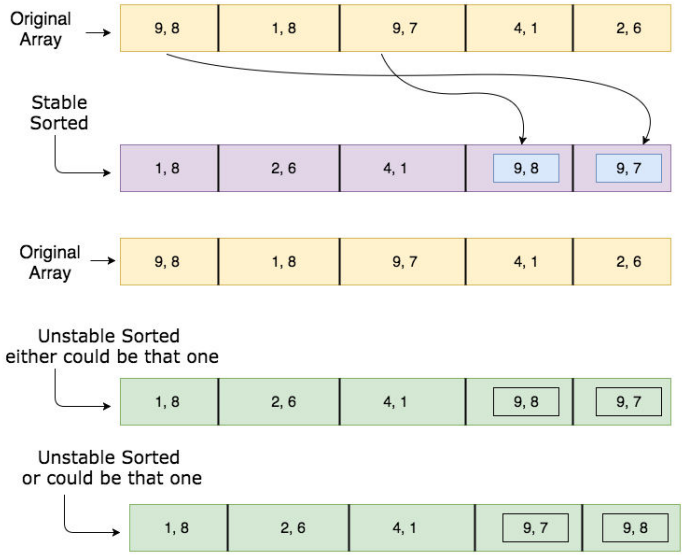
Merge-Sort Properties

- **Not Adaptive** : Running time doesn't change with data pattern
  - algorithm will re-order every single item in the list even if it is already sorted.
- **Stable/ Unstable** : Both implementations are possible .
- **Not Incremental** : Does not sort one by one element in each pass.
- **Not online** : Need all data to be in memory at the time of sorting.
- **Not in place** : It need O(n) extra space to sort two sub list of size(n/2).

Merge-Sort Properties

- **Merge sort is a stable sort**
- A sorting algorithm is said to be **stable** if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted

Merge-Sort Properties



Merge-Sort Applications

- **Merge sort is often the best choice for sorting a linked list.**
  - Linked list nodes may not be adjacent in memory. In linked list, we can insert items in the end in O(1) extra space and O(1) time.
  - In linked list to access i'th index, we have to travel each and every node from the head to i'th node as we don't have continuous block of memory. Merge sort accesses data sequentially and the need of random access is low.

Merge-Sort Applications

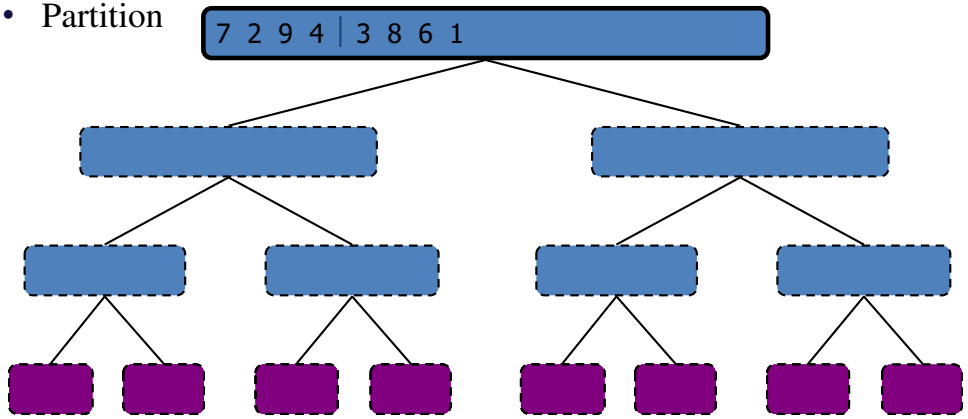
- **External Sorting:**
  - External sorting is a class of sorting algorithms that can handle massive amounts of data. External sorting is required when the data being sorted do not fit into the main memory of a computing device (usually RAM) and instead they must reside in the slower external memory, usually a hard disk drive.
  - External merge sort uses a hybrid sort-merge strategy. In the sorting phase, chunks of data small enough to fit in main memory are read, sorted, and written out to a temporary file. In the merge phase, the sorted subfiles are combined into a single larger file.

Merge-Sort Applications

- In Java, the Arrays.sort() methods use merge sort
- The Linux kernel uses merge sort for its linked lists
- Python uses Timsort, another tuned hybrid of merge sort and insertion sort, that has become the standard sort algorithm in Java SE 7

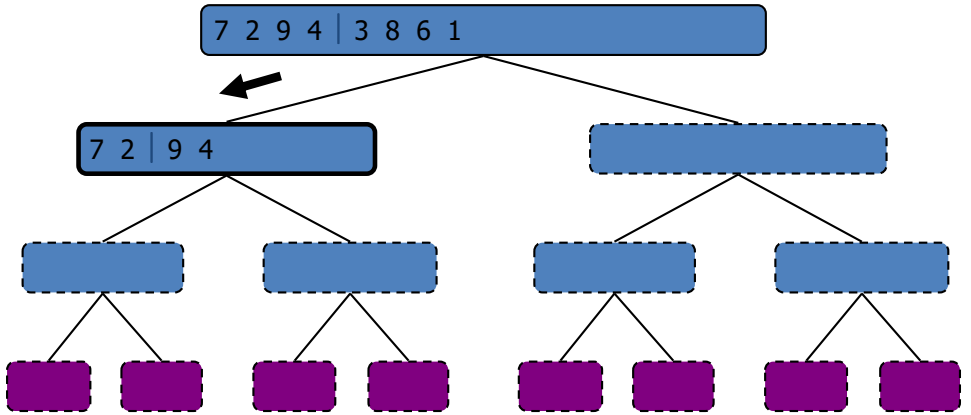
Execution Example-1

- Partition



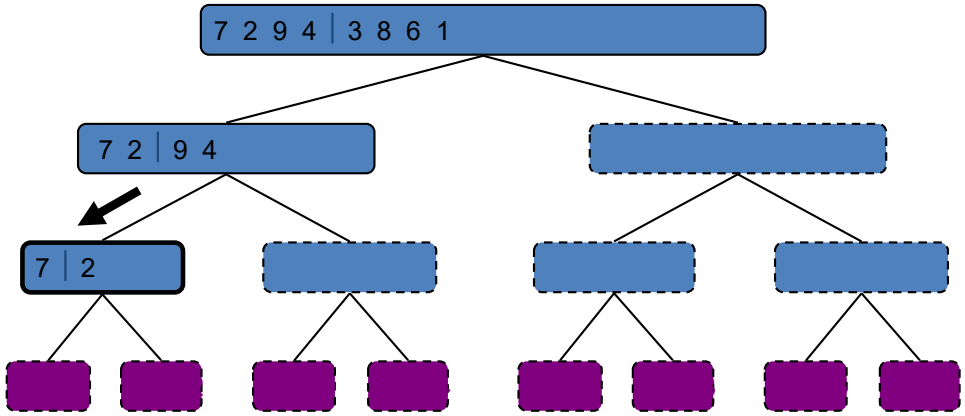
Execution Example (cont.)

- Recursive call, partition



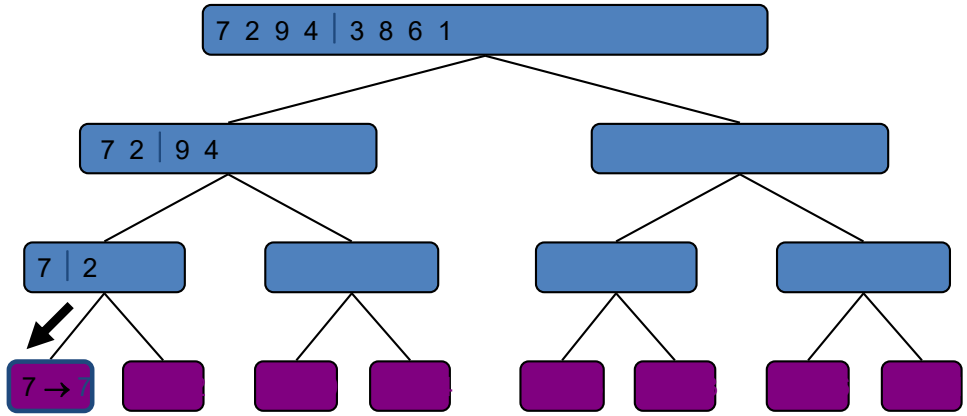
Execution Example (cont.)

- Recursive call, partition



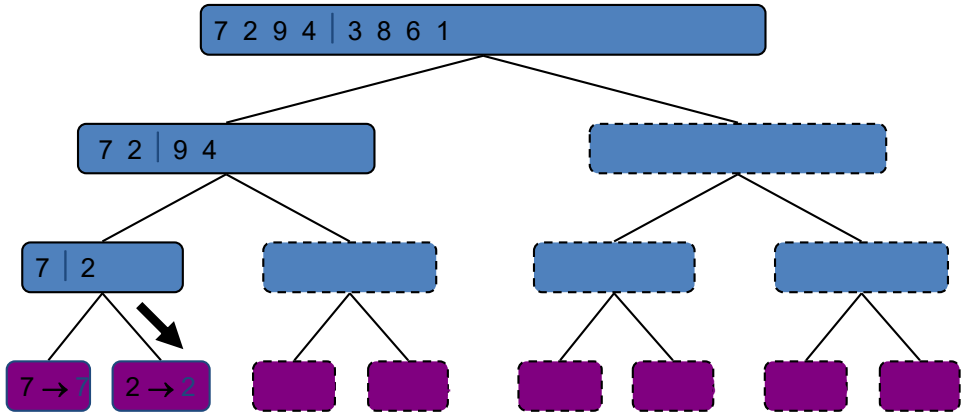
Execution Example (cont.)

- Recursive call, base case



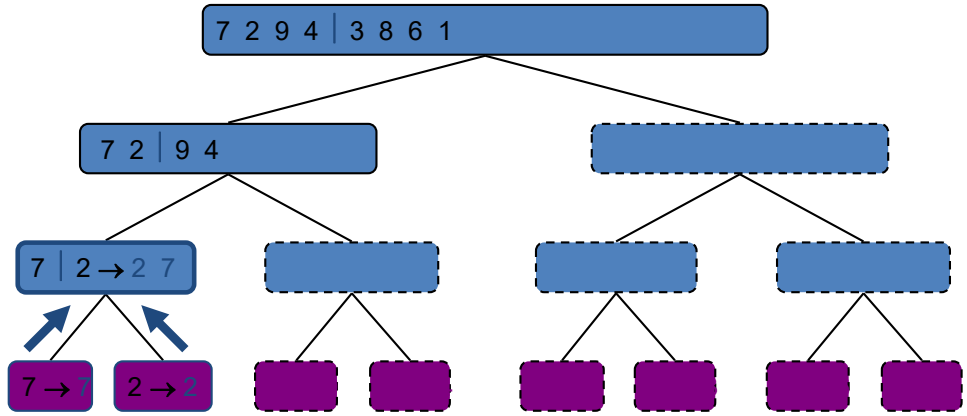
Execution Example (cont.)

- Recursive call, base case



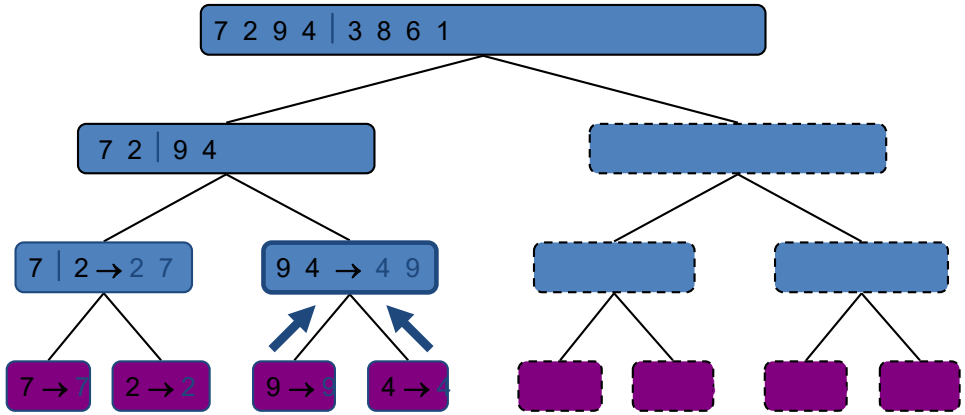
Execution Example (cont.)

- Merge



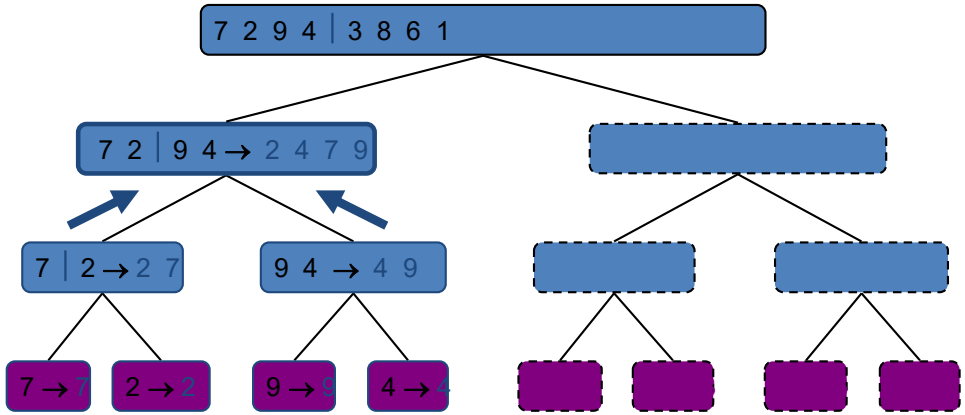
Execution Example (cont.)

- Recursive call, …, base case, merge



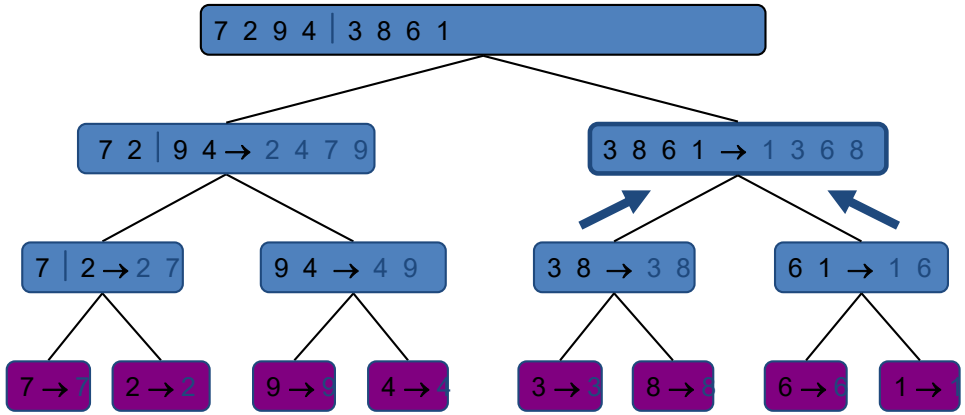
Execution Example (cont.)

- Merge



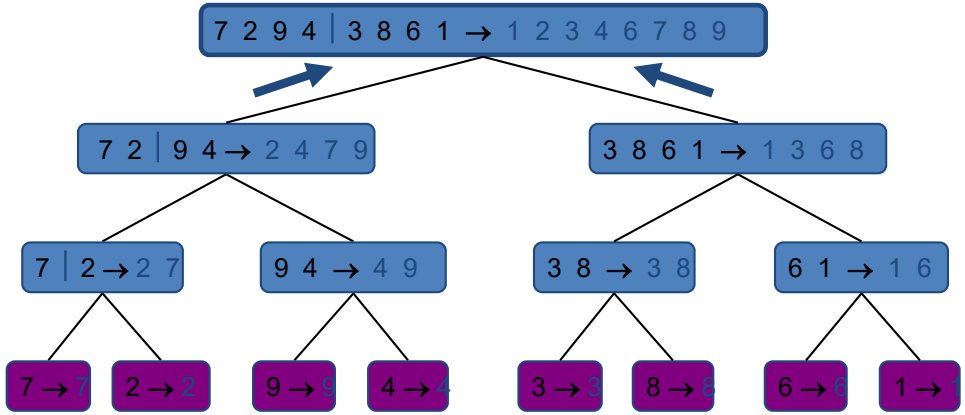
Execution Example (cont.)

- Recursive call, …, merge, merge

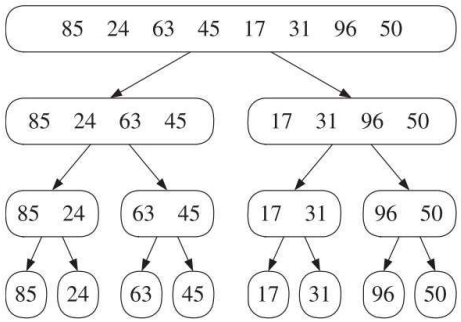


Execution Example (cont.)

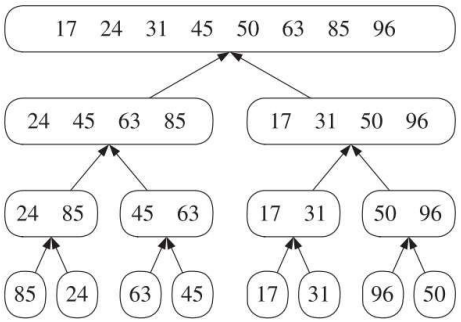
- Merge



Merge sort - Example -2



Input sequence processed at each node - Divide



Output sequence generated at each node - Conquer

Integer Multiplication

- The problem of multiplying big integers, that is, integers represented by a large number of bits that cannot be handled directly by the arithmetic unit of a single processor.

Integer Multiplication

- Given two big integers I and J represented with n bits each, we can easily compute I + J and I - J in O(n) time.

Integer Multiplication

- Efficiently computing the product I · J using the common grade-school algorithm requires, however, O (n^2) time.

Integer Multiplication-Divide and Conquer

- Divide and Conquer to Multiply - Attempt- #1
  - Let us represent I and J as below.
    - Attempt to rewrite the multiplication of I and J in terms of their components.
      - That is, this gives raise to recursion

Integer Multiplication-Divide and Conquer

- Divide and Conquer to Multiply - Attempt- #1

$$I = I_h 2^{n/2} + I_l$$
$$J = J_h 2^{n/2} + J_l$$

Integer Multiplication-Divide and Conquer

– We can then define I*J by multiplying the parts and adding:

$$I = I_h 2^{n/2} + I_l$$
$$J = J_h 2^{n/2} + J_l$$

$$I * J = (I_h 2^{n/2} + I_l) * (J_h 2^{n/2} + J_l)$$
$$= I_h J_h 2^n + I_h J_l 2^{n/2} + I_l J_h 2^{n/2} + I_l J_l$$

Integer Multiplication-Divide and Conquer

$$I * J = (I_h 2^{n/2} + I_l) * (J_h 2^{n/2} + J_l)$$
$$= I_h J_h 2^n + I_h J_l 2^{n/2} + I_l J_h 2^{n/2} + I_l J_l$$

Multiplication of 2 n bit numbers is now broken down into

- 4 Multiplications of n/2 bit numbers
- Plus 3 Additions

Multiplying any binary numbers by any arbitrary power of 2 is just a shift operation of bits

T(n) = 4T(n/2) + n,
implies T(n) is $O(n^2)$

Integer Multiplication-Divide and Conquer

- $O(n^2)$ is not an improvement indeed.
  - But we are able to multiply large integers in terms of smaller ones, now !
- We need to compute the following with lesser # of multiplication

$$I \cdot J = I_h J_h 2^n + I_l J_h 2^{n/2} + I_h J_l 2^{n/2} + I_l J_l$$

Integer Multiplication-Divide and Conquer

- Let us try to rewrite the following with 3 multiplications

$$I \cdot J = I_h J_h 2^n + I_l J_h 2^{n/2} + I_h J_l 2^{n/2} + I_l J_l$$

- Let P1=(I_h+I_l) * (J_h+J_l)
  $=I_hJ_h + I_hJ_l + I_lJ_h + I_lJ_l$
  P2= I_hJ_h
  P3= I_lJ_l
  P1-P2-P3= $I_hJ_l + I_lJ_h$

Now

I*J = P2 * 2^n + [P1-P2-P3]* $2^{n/2}$ +P3

Integer Multiplication-Divide and Conquer

- Let us try to rewrite the following with 3 multiplications

$$I \cdot J = I_h J_h 2^n + I_l J_h 2^{n/2} + I_h J_l 2^{n/2} + I_l J_l$$

- Let P1=(I_h+I_l) * (J_h+J_l)
  $=I_hJ_h + I_hJ_l + I_lJ_h + I_lJ_l$
  P2= I_hJ_h
  P3= I_lJ_l
  P1-P2-P3= $I_hJ_l + I_lJ_h$

Now

I*J = P2 * 2^n + [P1-P2-P3]* $2^{n/2}$ +P3

T(n) = 3T(n/2) + n,
By Master Theorem.
T(n) is $O(n^{\log_2 3})$,
Thus, T(n) is $O(n^{1.585})$.

Algorithm

```
Input:  Positive integers x and y, in binary
Output:  Their product

n = max(size of x, size of y)
if n = 1:  return xy

x_L, x_R = leftmost ⌈n/2⌉, rightmost ⌊n/2⌋ bits of x
y_L, y_R = leftmost ⌈n/2⌉, rightmost ⌊n/2⌋ bits of y

P_1 = multiply(x_L, y_L)
P_2 = multiply(x_R, y_R)
P_3 = multiply(x_L + x_R, y_L + y_R)
return P_1 × 2^n + (P_3 − P_1 − P_2) × 2^{n/2} + P_2
```


An Improved Integer Multiplication Algorithm

- Algorithm: Multiply two n-bit integers I and J.
- **O(n log n)** , by using a more complex divide-and-conquer algorithm called the Fast Fourier transform

Example-Decimal

Application

- Multiplying big integers has applications to data security, where big integers are used in encryption schemes.
  - More specifically, some important cryptographic algorithms such as RSA critically depend on the fact that prime factorization of large numbers takes a long time. Basically you have a "public key" consisting of a product of two large primes used to encrypt a message, and a "secret key" consisting of those two primes used to decrypt the message. You can make the public key public, and everyone can use it to encrypt messages to you, but only you know the prime factors and can decrypt the messages. Everyone else would have to factor the number, which takes too long to be practical, given the current state of the art of number theory.

Data Structures and Algorithms Design

BITS Pilani
Hyderabad Campus

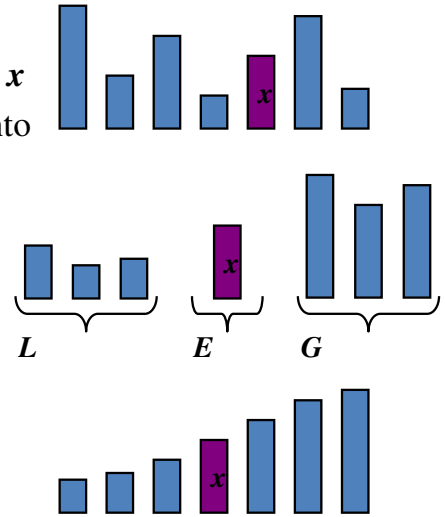
Febin.A.Vahab
2019-20

ONLINE SESSION 11 -PLAN

Sessions(#)	List of Topic Title	Text/Ref Book/external resource
12	Quick Sort Algorithm.	**T1: 5.2, 4.1, 4.3**

Quick-Sort

- Quick-sort is a randomized sorting algorithm based on the divide-and-conquer paradigm:
  - Divide: pick a random element ***x***
  - (called pivot) and partition ***S*** into
    - ***L*** elements less than ***x***
    - ***E*** elements equal ***x***
    - ***G*** elements greater than ***x***
  - Recur: sort ***L*** and ***G***
  - Conquer: join ***L***, ***E*** and ***G***



Quicksort

44	75	23	43	55	12	64	77	33

Partition

- We partition an input sequence as follows:
  - We remove, in turn, each element y from S and
  - We insert y into L, E or G, depending on the result of the comparison with the pivot x
- Each insertion and removal is at the beginning or at the end of a sequence, and hence takes $O(1)$ time
- Thus, the partition step of quick-sort takes $O(n)$ time

Partition

Algorithm *partition*(*S*, *p*)
  Input sequence *S*, position *p* of pivot
  Output subsequences *L*, *E*, *G* of the elements of *S* less than, equal to, or greater than the pivot, resp.
  L, *E*, *G* ← empty sequences
  x ← *S.remove*(*p*)
  while ¬*S.isEmpty*()
    y ← *S.remove*(*S.first*())
    if *y* < *x*
      L.insertLast(*y*)
    else if *y* = *x*
      E.insertLast(*y*)
    else { *y* > *x* }
      G.insertLast(*y*)
  return *L*, *E*, *G*

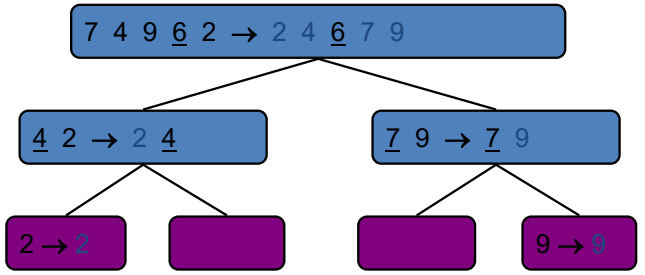
Quick-Sort

Algorithm inPlaceQuickSort(S, a, b):
Input: Sequence S of distinct elements; integers a and b
Output: Sequence S with elements originally from ranks from a to b, inclusive, sorted in nondecreasing order from ranks a to b

```
if a ≥ b then return          {empty subrange}
p ← S.elemAtRank(b)           {pivot}
l ← a                         {will scan rightward}
r ← b − 1                     {will scan leftward}
while l ≤ r do
  {find an element larger than the pivot}
  while l ≤ r and S.elemAtRank(l) ≤ p do
    l ← l + 1
  {find an element smaller than the pivot}
  while r ≥ l and S.elemAtRank(r) ≥ p do
    r ← r − 1
  if l < r then
    S.swapElements(S.atRank(l), S.atRank(r))
{put the pivot into its final place}
S.swapElements(S.atRank(l), S.atRank(b))
{recursive calls}
inPlaceQuickSort(S, a, l − 1)
inPlaceQuickSort(S, l + 1, b)
```

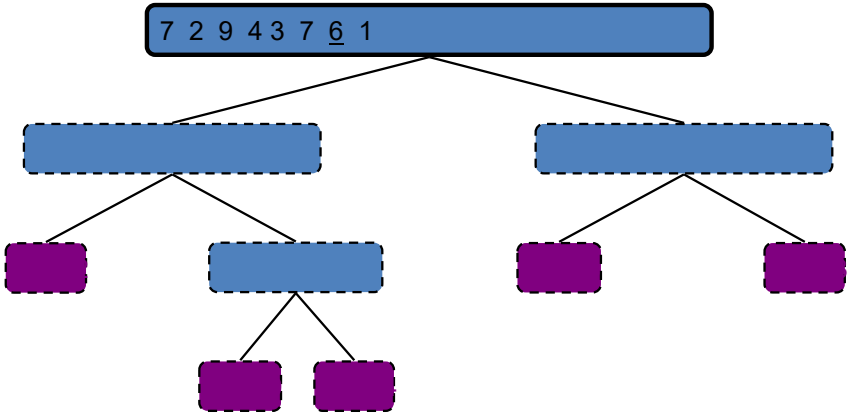
Quick-Sort Tree

- An execution of quick-sort is depicted by a binary tree
  - Each node represents a recursive call of quick-sort and stores
    - Unsorted sequence before the execution and its pivot
    - Sorted sequence at the end of the execution
  - The root is the initial call
  - The leaves are calls on subsequences of size 0 or 1



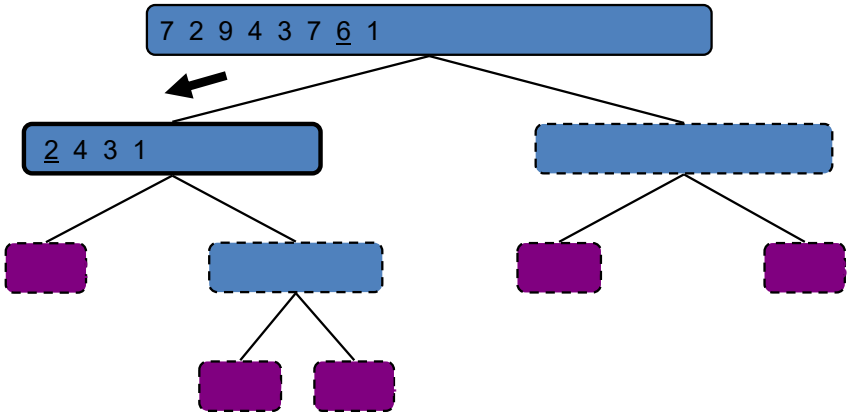
Execution Example

- Pivot selection



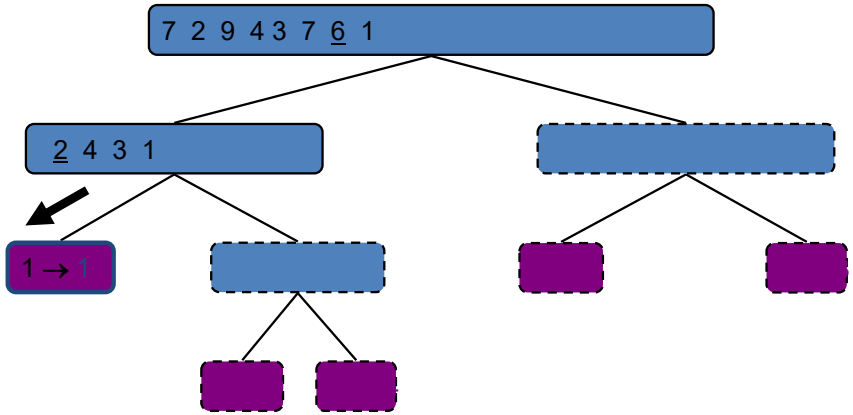
Execution Example (cont.)

- Partition, recursive call, pivot selection



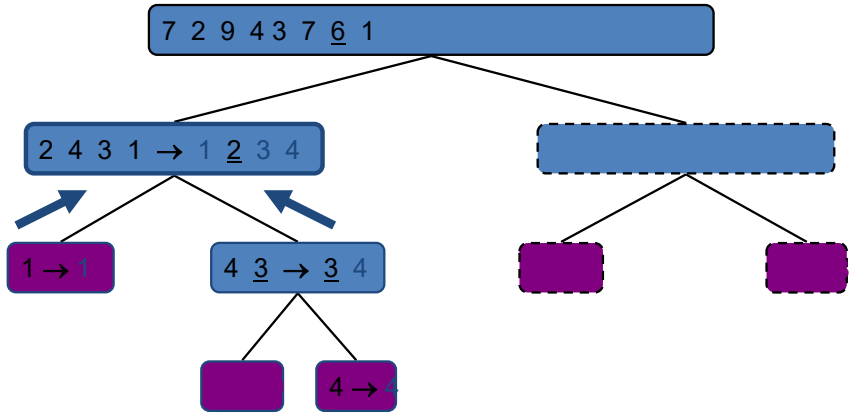
Execution Example (cont.)

- Partition, recursive call, base case



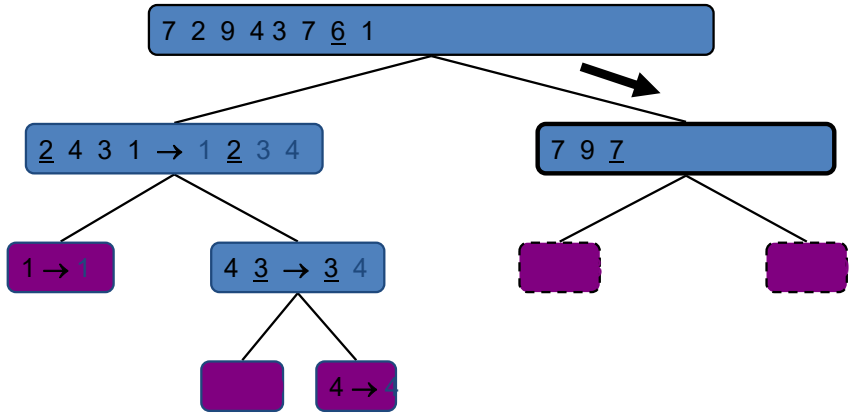
Execution Example (cont.)

- Recursive call, …, base case, join



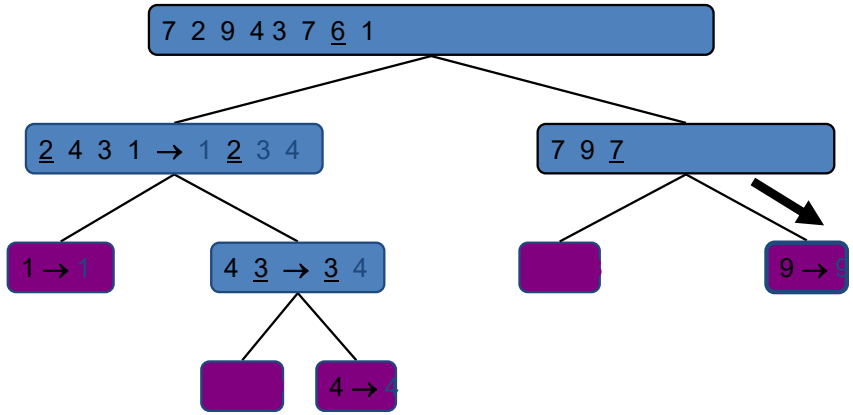
Execution Example (cont.)

- Recursive call, pivot selection



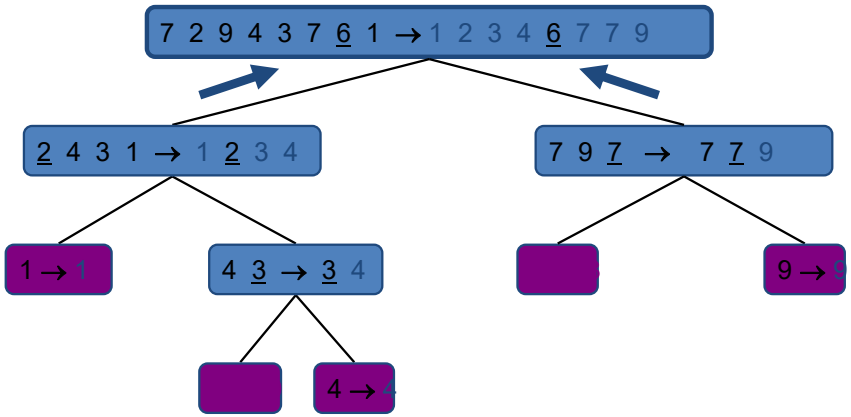
Execution Example (cont.)

- Partition, …, recursive call, base case



Execution Example (cont.)

- Join, join

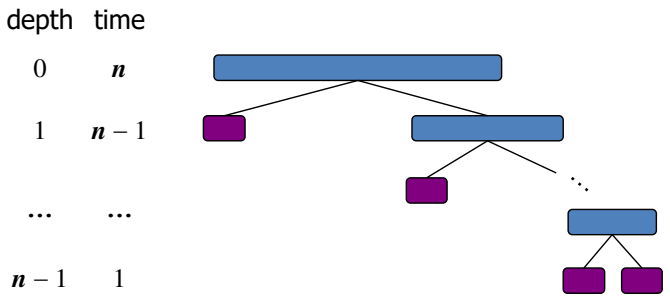


Worst-case Running Time

- The worst case for quick-sort occurs when the pivot is the unique minimum or maximum element
- One of L and G has size $n - 1$ and the other has size 0
- The running time is proportional to the sum

$$n + (n - 1) + \ldots + 2 + 1$$

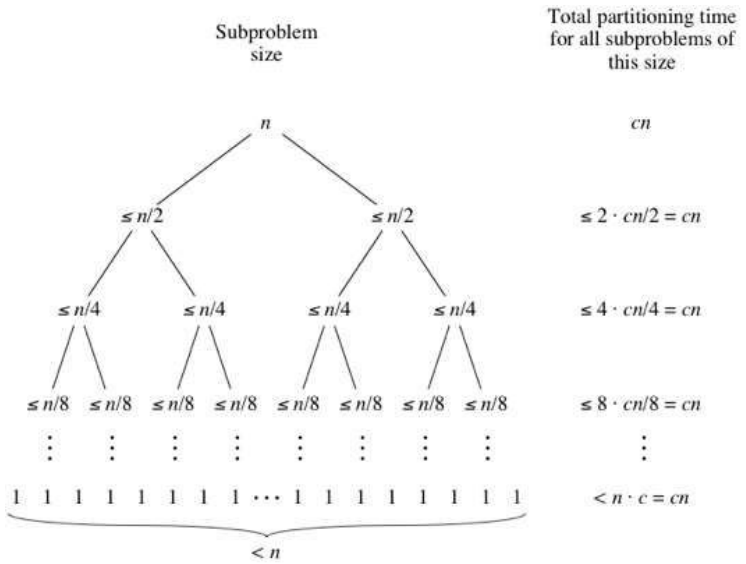
- Thus, the worst-case running time of quick-sort is $O(n^2)$



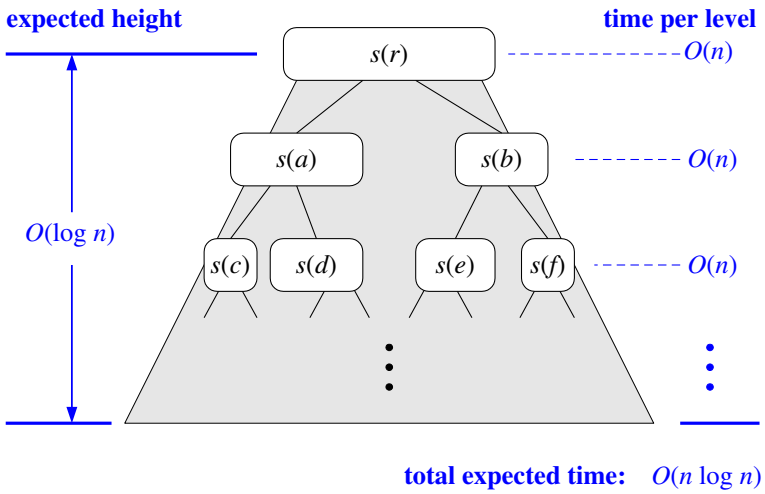
Best Case Analysis

- Best case occurs when the partitions are as evenly balanced as possible
- If the subarray has an odd number of elements and the pivot is right in the middle after partitioning, then each partition has (n-1)/2 elements.
- If the subarray has an even number n of elements and one partition has n/2 and other having n/2-1.
- In either of these cases, each partition has at most n/2 elements

Best Case Analysis



A visual time analysis of the quick-sort tree T

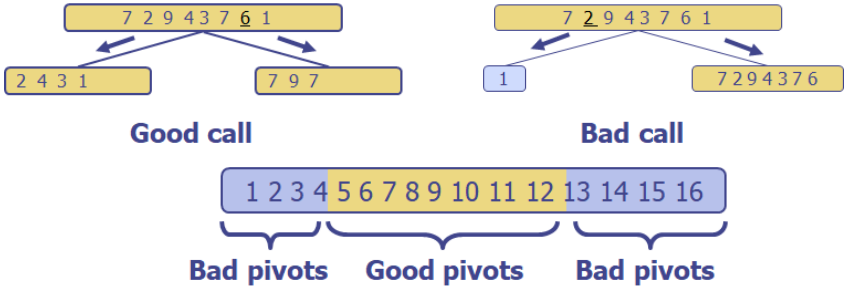


Best Case Running Time

- The amount or work done at the nodes of the same depth is O(n)
- The expected height of the quick-sort tree is O(log n)
- Thus, the expected running time of quick-sort is O(n log n)

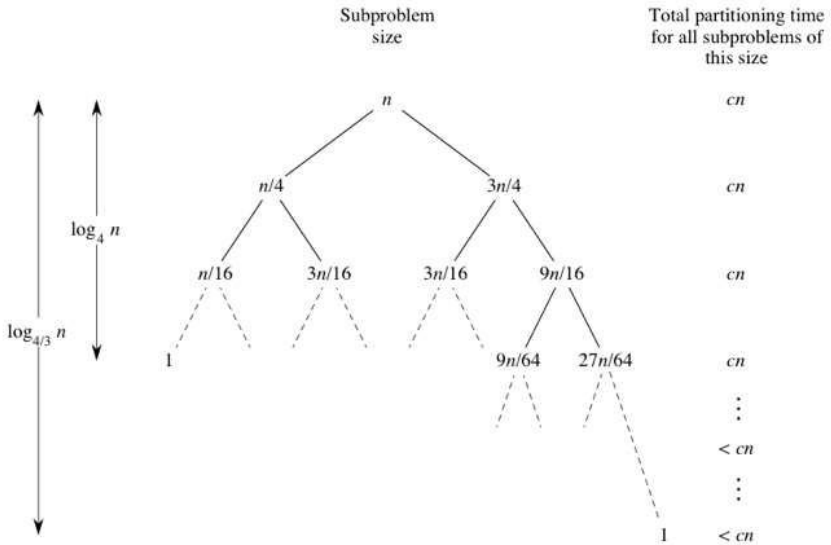
Expected Running Time

- Consider a recursive call of quick-sort on a sequence of size ***s***
  Good call: the sizes of ***L*** and ***G*** are each less than 3***s***/4
  Bad call: one of ***L*** and ***G*** has size greater than 3***s***/4

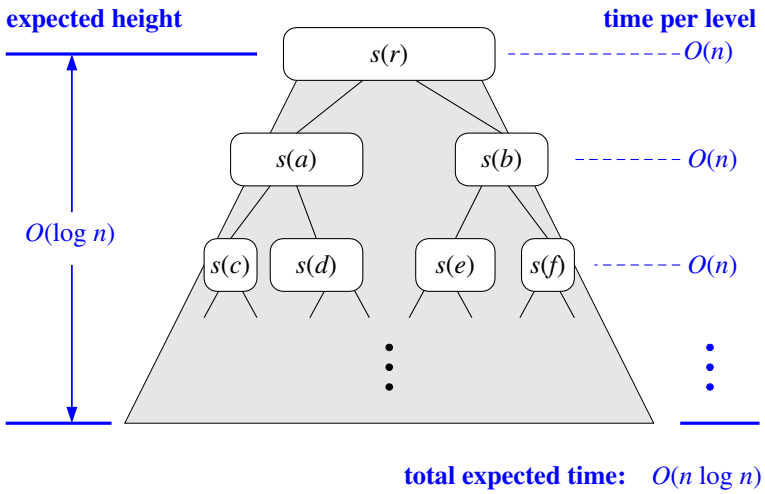


- A call is good with probability 1/2
  - 1/2 of the possible pivots cause good calls:

Expected Running Time



A visual time analysis of the quick-sort tree T



Expected Running Time

Intuitively,

- (In best case) For the tree to have height $\log_2 n$, with each step, the size of the sequence must shrink to at most (n/2).
  - That is, $\log_2 n$ invocations
- For each call to be good,
  - the input size should shrink to atmost (¾)n
  - If all calls are good calls, the expected height of tree is $\log_{(4/3)} n$
  - In other words, $\log_{(4/3)} n$ good calls are needed to get O(log n) height
  - If pivots are randomly chosen, we expect that, out of 2*log (4/3) n calls, log (4/3) n calls are good !
  - $\log_{(4/3)} n$ and log n differs by only a factor of $\log_2(4/3)$ which is a constant.
  - This implies that the height of tree is O(log n)
- Expected complexity of quick sort is O (n log n)

Expected Running Time

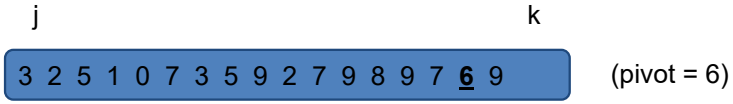
- The amount or work done at the nodes of the same depth is O(n)
- The expected height of the quick-sort tree is O(log n)
- Thus, the expected running time of quick-sort is O(n log n)

In-Place Quick-Sort

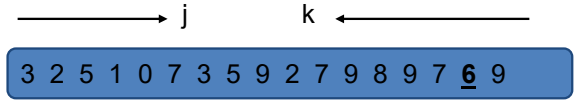
- A sorting algorithm is in-place if it uses only a constant amount of memory in addition to that needed for the objects being sorted themselves

In-Place Partitioning

- Perform the partition using two indices to split S into L and E U G (a similar method can split E U G into E and G).



- Repeat until j and k cross:
  - Scan j to the right until finding an element ≥ x.
  - Scan k to the left until finding an element < x.
  - Swap elements at indices j and k



In-Place Quick-Sort

- Quick-sort can be implemented to run in-place
- In the partition step, we use replace operations to rearrange the elements of the input sequence such that
  - the elements less than the pivot have rank less than ***h***
  - the elements equal to the pivot have rank between ***h*** and ***k***
  - the elements greater than the pivot have rank greater than ***k***
- The recursive calls consider
  - elements with rank less than ***h***
  - elements with rank greater than ***k***

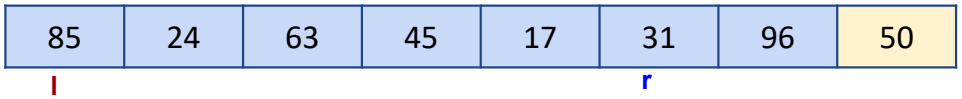
In-Place Quick-Sort

Algorithm inPlaceQuickSort(S, a, b):
Input: Sequence S of distinct elements; integers a and b
Output: Sequence S with elements originally from ranks from a to b, inclusive, sorted in nondecreasing order from ranks a to b

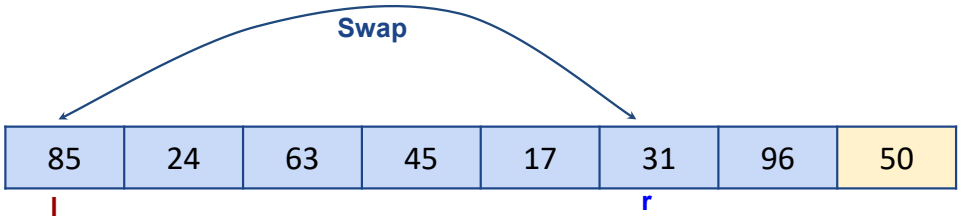
```
if a ≥ b then return          {empty subrange}
p ← S.elemAtRank(b)           {pivot}
l ← a                         {will scan rightward}
r ← b − 1                     {will scan leftward}
while l ≤ r do
  {find an element larger than the pivot}
  while l ≤ r and S.elemAtRank(l) ≤ p do
    l ← l + 1
  {find an element smaller than the pivot}
  while r ≥ l and S.elemAtRank(r) ≥ p do
    r ← r − 1
  if l < r then
    S.swapElements(S.atRank(l), S.atRank(r))
{put the pivot into its final place}
S.swapElements(S.atRank(l), S.atRank(b))
{recursive calls}
inPlaceQuickSort(S, a, l − 1)
inPlaceQuickSort(S, l + 1, b)
```

In-Place Quick-Sort

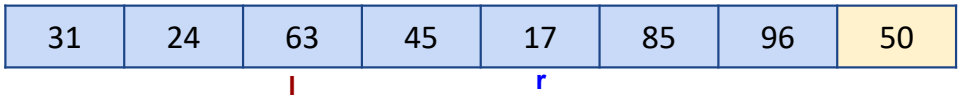
In-Place Quick-Sort



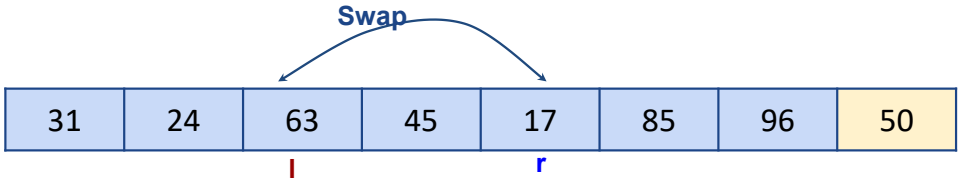
In-Place Quick-Sort



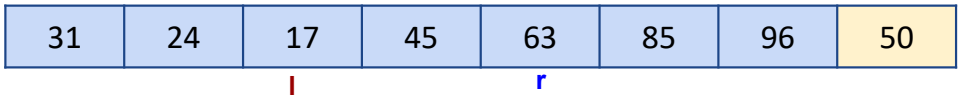
In-Place Quick-Sort



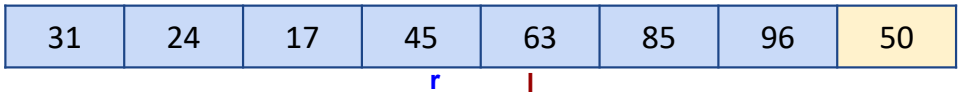
In-Place Quick-Sort



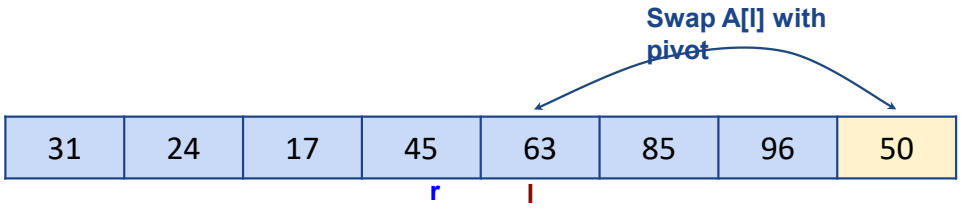
In-Place Quick-Sort



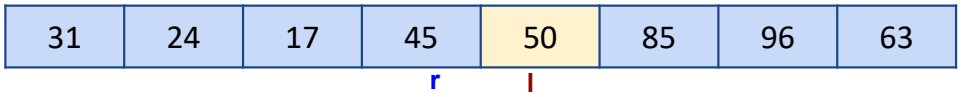
In-Place Quick-Sort



In-Place Quick-Sort



In-Place Quick-Sort



Next Recursive Calls:

inPlaceQuickSort(A, a, l − 1)
inPlaceQuickSort(A, l + 1, b)

Quick-Sort Properties

- **Not Adaptive**
- **Not Stable-Can be made stable**.
- **Not Incremental** : Incremental versions possible
- **Not online**
- **In Place**

QuickSort

- **Advantages of Quicksort**
- Its average-case time complexity to sort an array of n elements is O(nlogn).
- It requires no additional memory.
- **Disadvantages of Quicksort**
- Its worst-case running time, $O(n^2)$ to sort an array of n elements, happens when pivot is an extreme
- It is not stable.

QuickSort and Mergesort

- Quick Sort: traditionally built-in for many runtimes, hence used by programs that call the default. *Can't be used where worst-case behavior could be exploited or cause significant ramifications*, such as services that might receive denial-of-service attacks, or real-time systems.
- Java's systems programmers have chosen to use quicksort (with 3-way partitioning) to implement the primitive-type methods, and merge sort for reference-type methods. The primary practical implications of these choices are to trade speed and memory usage (for primitive types) for stability and guaranteed performance (for reference types).
- Merge Sort: used in *database scenarios*, because stable and external (results don't all fit in memory).

QuickSort and MergeSort

- In most practical situations, quicksort is the method of choice.
- If stability is important and space is available, merge sort might be best.
- In some performance-critical applications, the focus may be on just sorting numbers, so it is reasonable to avoid the costs of using references and sort primitive types instead.

BITS Pilani
Hyderabad Campus

THANK YOU!

Range query

- A **range query** q(A,i,j) on an array A=[a1,a2,a3,..,an] of *n* elements of some set *S*, denoted A[1,n], takes two indices 1<=i<=j<=n ,a function *f* defined over arrays of elements of *S* and outputs f(A[i,j])=f(ai,……,aj)

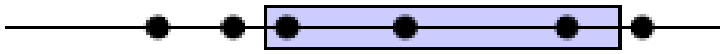
BITS Pilani
Hyderabad Campus

Data Structures and Algorithms Design

Febin.A.Vahab

Range query

- Data:Points P={p_1, p_2, ... p_n} in 1-D space (set of real numbers)
- Query: Which points are in 1-D query rectangle (in interval [x, x'])



Range query

- Range: [x, x']
- **Data Structure 1:Sorted Array**

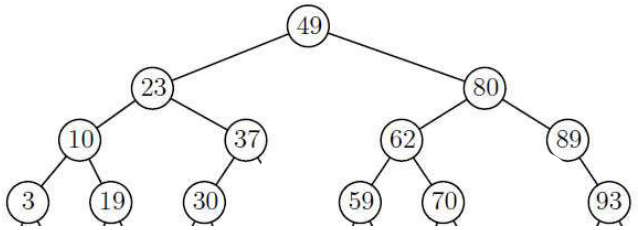
A=

3	9	27	28	29	98	141	187	200	201	202	999

- Search for x and x' in A by Binary search takes
- O(log n) time
- Output all points between them ,takes
- O(k) time
- Total : O(k+logn)

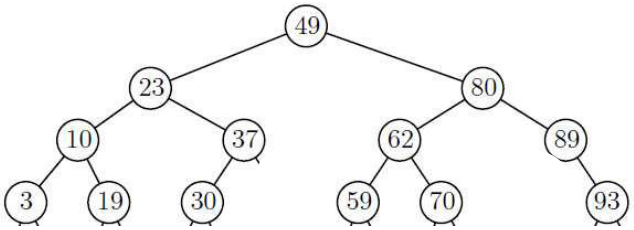
Range query

- Data Structure 2:BST
  - Search using binary search property.
  - Some subtrees are eliminated during search.



Range query

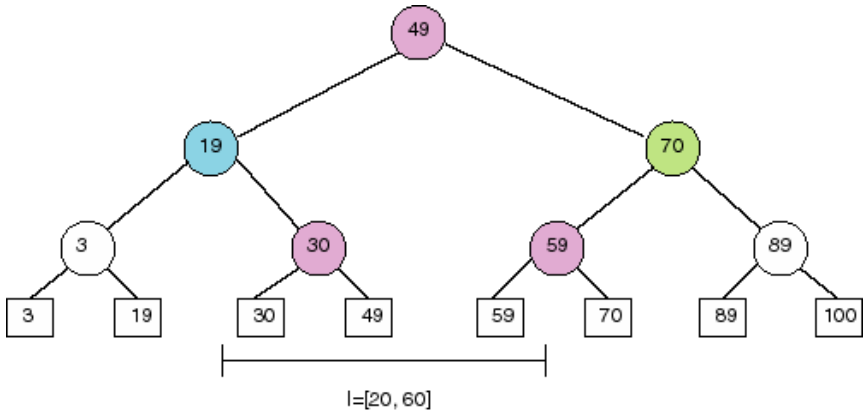
- Data Structure 2:BST
  - Search using binary search property.
  - Some subtrees are eliminated during search.



Range query

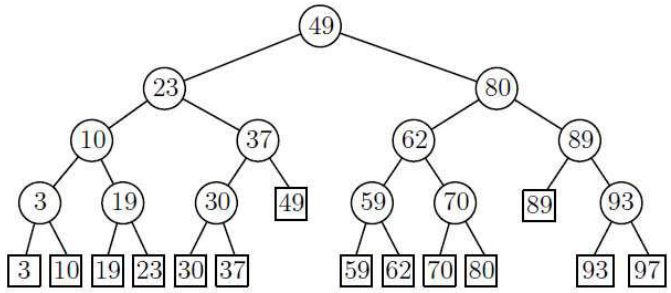
```
FindPoints([x, x'], T)
  if T is a leaf node, then
    if x <= val(T) <= x' then
      return { val(T) }
    else
      return {}
    end if
  end if
  <else T is an interior node of tree>
  if x' <= val(T) then
    return FindPoints([x, x'], left(T))
  else if x > val(T) then
    return FindPoints([x, x'], right(T))
  else <interval spans splitting value>
    return FindPoints([x, x'], left(T)) union FindPoints([x, x'], right(T))
  end if
```

Range query

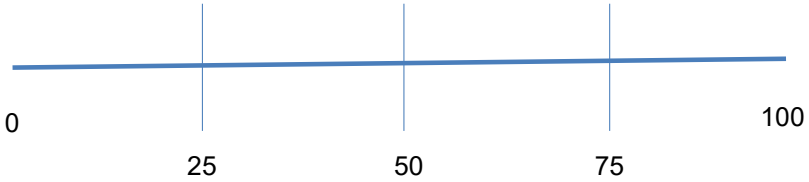


Range query

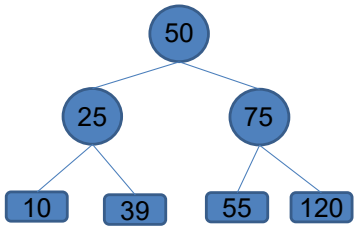
- Data Structure 3: BST with data stored in leaves
  - Internal nodes store splitting values (i.e., not necessarily same as data).
  - Data points are stored in the leaf nodes.



BST with data stored in leaves

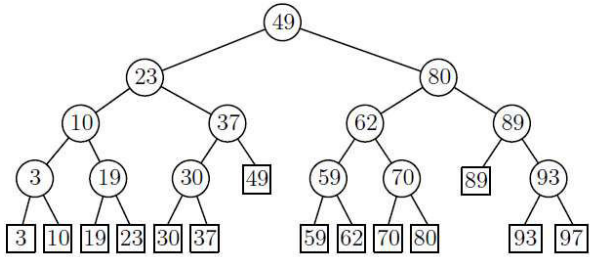


Data: 10, 39, 55, 120



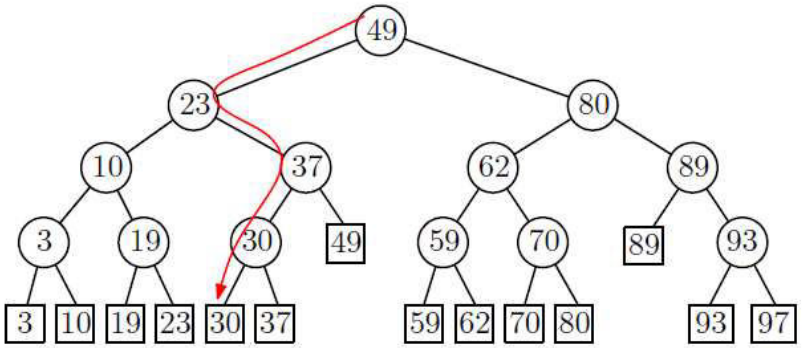
BST with data stored in leaves

- Retrieving data in [x, x']
  - Perform binary search twice, once using x and the other using x'
  - Suppose binary search ends at leaves l and l'
  - The points in [x, x'] are the ones stored between l and l' plus, possibly, the points stored in l and l'



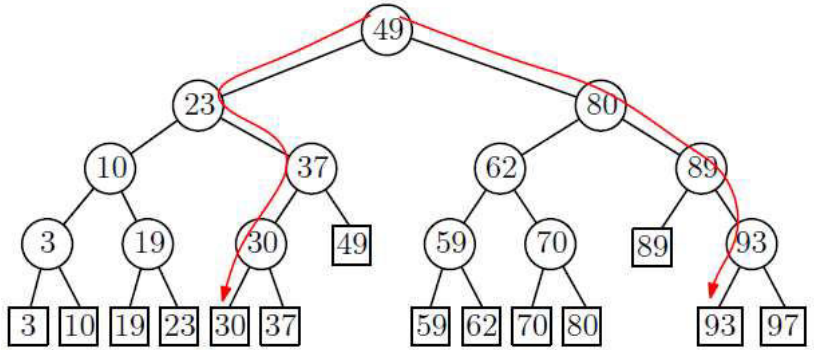
BST with data stored in leaves

- **Example:** retrieve all points in [25, 90]
  - The search path for 25 is:



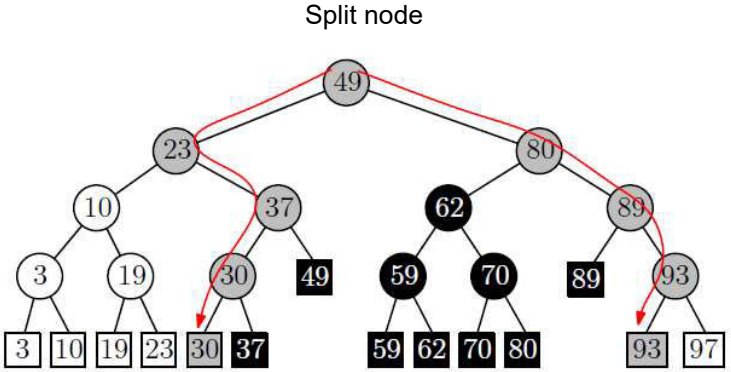
BST with data stored in leaves

- **Example:** The search for 90 is:



BST with data stored in leaves

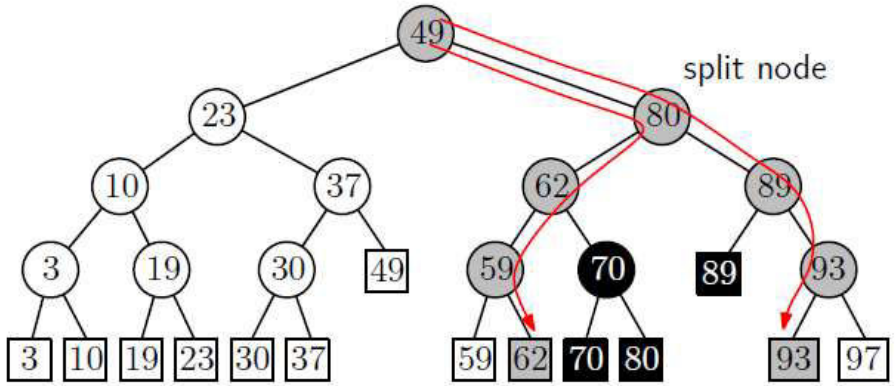
- Examine the leaves in the sub-trees between the two traversing paths from the root.



Retrieve all points in [25, 90]

Range Search – Another Example

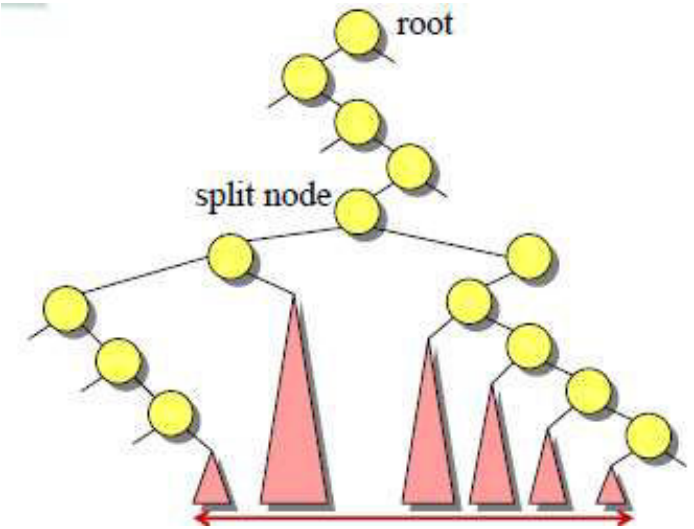
A 1-dimensional range query with $[61, 90]$



Range Search

- How do we find the leaves of interest?
  - Find **split node** (i.e., node where the paths to x and x' split).
  - **Left turn**: report leaves in right subtrees
  - **Right turn**: report leaves in left substrees

Range Search

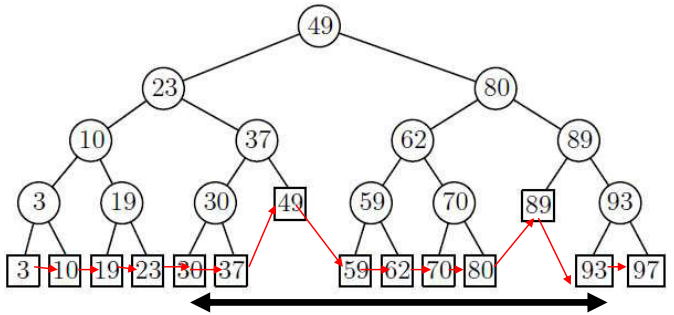


Range Search

- O(logn + k) time where k is the number of items reported.

Range Search

- Speed-up search by keeping the leaves in sorted order using a linked-list



- Applications

One-dimensional range searching [Goodrich]

- Given an ordered dictionary D, we want to perform the following query operation:
- findAIIInRange(k1 , k2) : Return all the elements in dictionary D with key k such that k1 <=k<=k2

One-dimensional range searching

- How we can use a binary search tree T representing dictionary D to perform query
- findAIIInRange(k1 , k2)
- Use a recursive method lDTreeRangeSearch that takes as arguments the range parameters k1 and k2 and a node v in T

One-dimensional range searching

- If node v is external, we are done.
- If node v is internal, we have three cases, depending on the value of key (v) , the key of the item stored at node v:
- **key(v)< k1** : Rrecurse on the right child of v.
- **k1 <= key(v)<=k2** : Report element(v) and recurse on both children of v.
- **key(v) > k2** : Recurse on the left child of v.

Algorithm lDTreeRangeSearch (k1 , k2 , v) :

Algorithm 1DTreeRangeSearch(k_1, k_2, v):
Input: Search keys k_1 and k_2, and a node v of a binary search tree T
Output: The elements stored in the subtree of T rooted at v, whose keys are greater than or equal to k_1 and less than or equal to k_2

```
if T.isExternal(v) then
    return ∅
if k1 ≤ key(v) ≤ k2 then
    L ← 1DTreeRangeSearch(k1, k2, T.leftChild(v))
    R ← 1DTreeRangeSearch(k1, k2, T.rightChild(v))
    return L ∪ {element(v)} ∪ R
else if key(v) < k1 then
    return 1DTreeRangeSearch(k1, k2, T.rightChild(v))
else if k2 < key(v) then
    return 1DTreeRangeSearch(k1, k2, T.leftChild(v))
```

lDTreeRangeSearch (k1 , k2 , v)

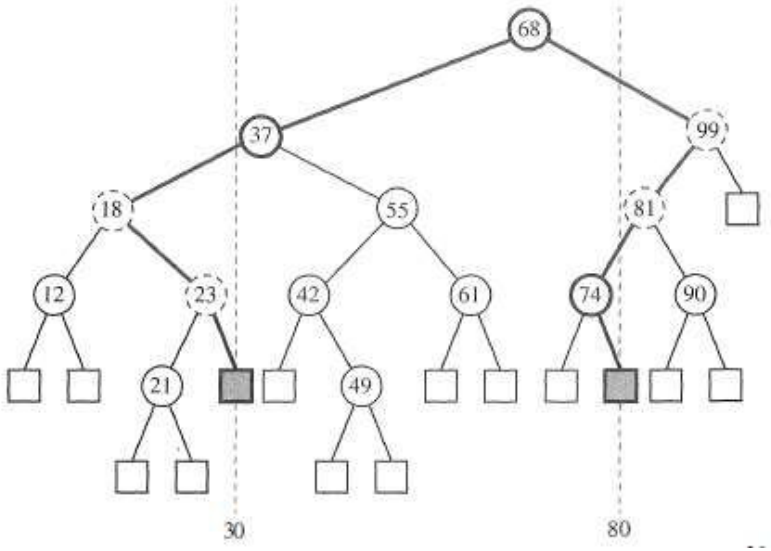
- We perform operation findAIIInRange(k1 , k2) by calling

  lDTreeRangeSearch (k1 , k2 , T. root( ))

  Example(Figure next slide)
- One-dimensional range search using a binary search tree for

  k1 = 30 and k2 = 80.
- Paths PI and P2 of boundary nodes are drawn with thick lines.
- The boundary nodes storing items with key outside the interval [k1 , k2] are drawn with dashed lines.

Algorithm lDTreeRangeSearch (k1 , k2 , v) :



lDTreeRangeSearch -Performance

- Let P1 be the search path traversed when performing a search in tree T for key k1 .
- Path P1 starts at the root of T and ends at an external node of T .
- Define a path P2 similarly with respect to k2 . We identify each node v of T as belonging to one of following three groups

lDTreeRangeSearch -Performance

- Case 1:Node v is a **boundary node** if v belongs to PI or P2 ; a boundary node stores an item whose key may be inside or outside the interval [k1 , k2] .
- Case 2:Node v is an **inside node** if v is not a boundary node and v belongs to a subtree rooted at a right child of a node of PI or at a left child of a node of P2 ;an internal inside node stores an item whose key is inside the interval [k1 , k2] .

lDTreeRangeSearch -Performance

- Case 3:Node v is an **outside node** if v is not a boundary node and v belongs to a subtree rooted at a left child of a node of PI or at a right child of a node of P2 ; an internal outside node stores an item whose key is outside the interval (k1 , k2] .

lDTreeRangeSearch -Performance

- A balanced binary search tree supports one-dimensional range searching in an ordered dictionary with n items:
- The space used is O(n).
- Operation findAIIInRange takes O(log n + s) time, where s is the number of elements reported.
- Operations insertltem and removeElement each take O(log n) time.

BST-Applications

- Used in many search applications where data is constantly entering/leaving, such as the map and set objects in many languages' libraries.
- Binary Space Partition - Used in almost every 3D video game to determine what objects need to be rendered. Binary space partitioning (BSP) is a method for recursively subdividing a space into convex sets by hyper planes. This subdivision gives rise to a representation of objects within the space by means of a tree data structure known as a BSP tree

BST-Applications

- Huffman Coding Tree: The branches of the tree represent the binary values 0 and 1 according to the rules for common prefix-free code trees. The path from the root tree to the corresponding leaf node defines the particular code word.
- It is used to implement multilevel indexing in DATABASE.

Sample Question 1

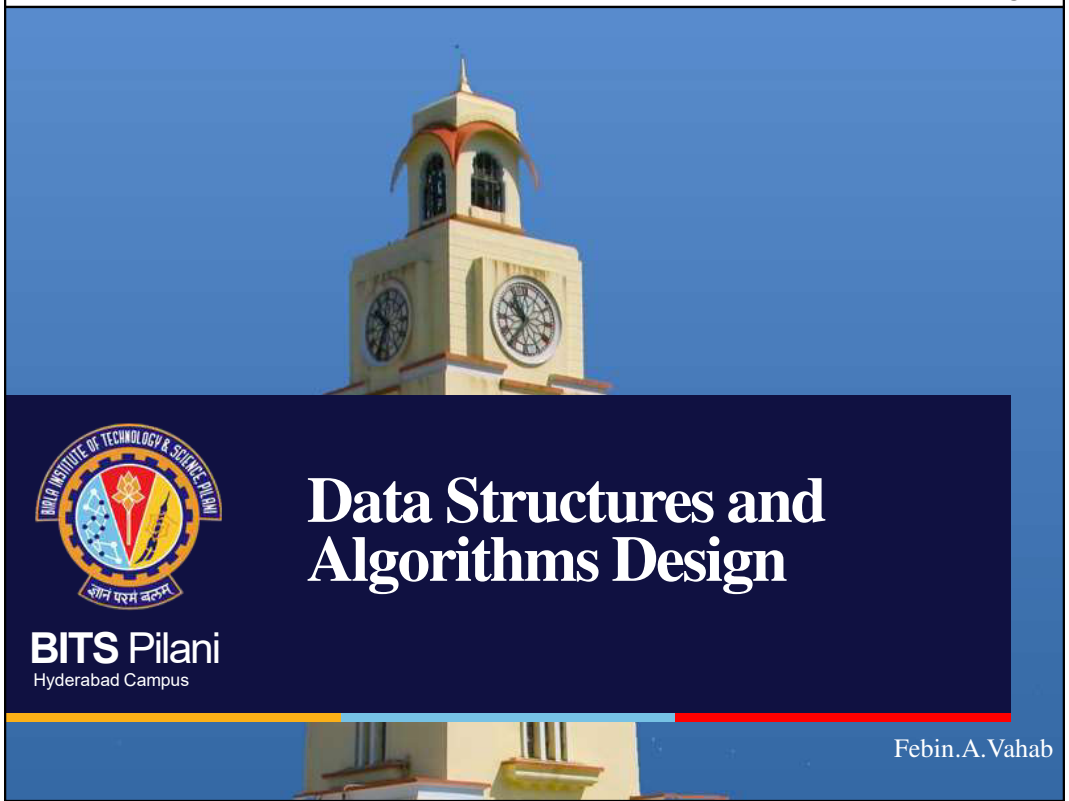
- *Prof X is standing at the door of his classroom. There are currently **N** students in the class, **i** th student got A_i candies. There are still **M** more students to come. At every instant, a student enters the class and wishes to be seated with a student who has **exactly** the same number of candies. For each student, Professor shouts YES if such a student is found, NO otherwise. Even if the student entering the class can't find a partner with equal no. of candies, he will still enter the class and be seated.*
- *Identify a data structure to implement the above problem statement.*

Sample Question 2

You have decided to run off to Los Angeles for the summer and start a new life as a rockstar. However, things aren't going great, so you're consulting for a hotel on the side. This hotel has N one-bed rooms, and guests check in and out throughout the day. When a guest checks in, they ask for a room whose number is in the range $[l, h]$.[1]

You want to implement a data structure that supports the following data operations as efficiently as possible.

1. INIT(N): Initialize the data structure for N empty rooms numbered $1, 2, \ldots, N$, in polynomial time.
2. COUNT(l, h): Return the number of **available** rooms in $[l, h]$, in $O(\log N)$ time.
3. CHECKIN(l, h): In $O(\log N)$ time, return the first empty room in $[l, h]$ and mark it occupied, or return NIL if all the rooms in $[l, h]$ are occupied.
4. CHECKOUT(x): Mark room x as not occupied, in $O(\log N)$ time.



ONLINE SESSION 13-PLAN

Sessions(#)	List of Topic Title	Text/Ref Book/external resource
13	**Dynamic Programming - Design Principles and Strategy, Matrix Chain Product Problem, 0/1 Knapsack Problem, All-pairs Shortest Path Problem**	**T1: 5.3, 7.2**

Dynamic Programming

- Invented by a prominent U.S. mathematician, Richard Bellman
- The word “programming” in the name of this technique stands for “planning” and does not refer to computer programming.

Dynamic Programming

- Dynamic programming is a technique for solving problems with overlapping subproblems.
- Typically, given problem’s solution can be related to solutions of its smaller subproblems.
- Rather than solving overlapping subproblems again and again, dynamic programming suggests solving each of the smaller subproblems only once and recording the results in a table from which a solution to the original problem can then be obtained.

Dynamic Programming

- A straightforward application of dynamic programming can be interpreted as a special variety of space-for-time trade-off.
- Optimisation of plain recursion.

Dynamic Programming-Example

- The Fibonacci numbers are the numbers in the following integer sequence.
- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144…
- F(n) = F(n-1) + F(n-2), F0 = 0 and F1 = 1

```
int fib(int n)
{
  if ( n <= 1 )
    return n;
  return fib(n-1) + fib(n-2);
}
```


Dynamic Programming-Example

- Time complexity:
- T(n) = T(n-1) + T(n-2)
- which is exponential.

Dynamic Programming-Properties

Overlapping Sub-problems:

- Sub-problems needs to be solved again and again.
- In recursion we solve those problems every time and in dynamic programming we solve these sub problems only once and store it for future use

Optimal Substructure:

- A problem can be solved by using the solutions of the sub problems

Dynamic Programming-Example

```
Algorithm DynamicFibonacci(n)
{
 f[0]=0,f[1]=1
 for(i =2;i<=n;i++)
      f[i]=f[i-1]+f[i-2]
return f[n];
}
```

Time complexity??

Dynamic Programming-Example

- A man put a pair of rabbits in a place surrounded by a wall. How many pairs of rabbits will be there in a year if the initial pair of rabbits (male and female) are newborn and all rabbit pairs are not fertile during their first month of life but thereafter give birth to one new male-female pair at the end of every month?

Dynamic Programming

- The circumstances and restrictions are not realistic.
- Still, this isn't THAT unrealistic a situation in the short term.

Dynamic Programming

- Similar to Fibonacci problem.
- To solve Fibonacci's problem, we'll let f(n) be the number of pairs during month n.
- By convention, f(0) = 0. f(1) = 1 for our new first pair.
- f(2) = 1 as well, as conception just occurred.
- The new pair is born at the end of month 2, so during month 3, f(3) = 2.
- Only the initial pair produces offspring in month 3, so f(4) = 3.

Dynamic Programming

- In month 4, the initial pair and the month 2 pair breed, so f(5) = 5. We can proceed this way, presenting the results in a table. At the end of a year, Fibonacci has 144 pairs of rabbits.

Month	0	1	2	3	4	5	6	7	8	9	10	11	12
Pairs	0	1	1	2	3	5	8	13	21	34	55	89	144

The General Dynamic Programming Technique

- Applies to a problem that at first seems to require a lot of time (possibly exponential), provided we have:
  - **Simple subproblems:** the subproblems can be defined in terms of a few variables
  - **Subproblem optimality:** the global optimum value can be defined in terms of optimal subproblems
  - **Subproblem overlap:** the subproblems are not independent, but instead they overlap (hence, should be constructed bottom-up).

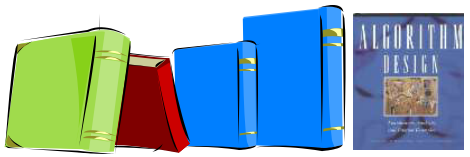
The 0/1 Knapsack Problem

- Given: A set S of n items, with each item i having
  - w_i - a positive weight
  - p_i - a positive benefit
- Goal: Choose items with maximum total benefit but with weight at most W.
- This problem is called **a "0- 1 " problem**, because each item must be entirely accepted or rejected.
  - In this case, we let T denote the set of items we take
  - Objective: maximize $\sum p_i x_i$
  - Constraint: $\sum_{i \varepsilon T} w_i x_i \leq W$

Example

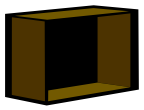
- Given: A set S of n items, with each item i having
  - b_i - a positive "benefit"
  - w_i - a positive "weight"
- Goal: Choose items with maximum total benefit but with weight at most W.

Items:



	1	2	3	4	5
Weight:	4 kg	2 kg	2 kg	6 kg	2 kg
Benefit:	$20	$3	$6	$25	$80

"knapsack"

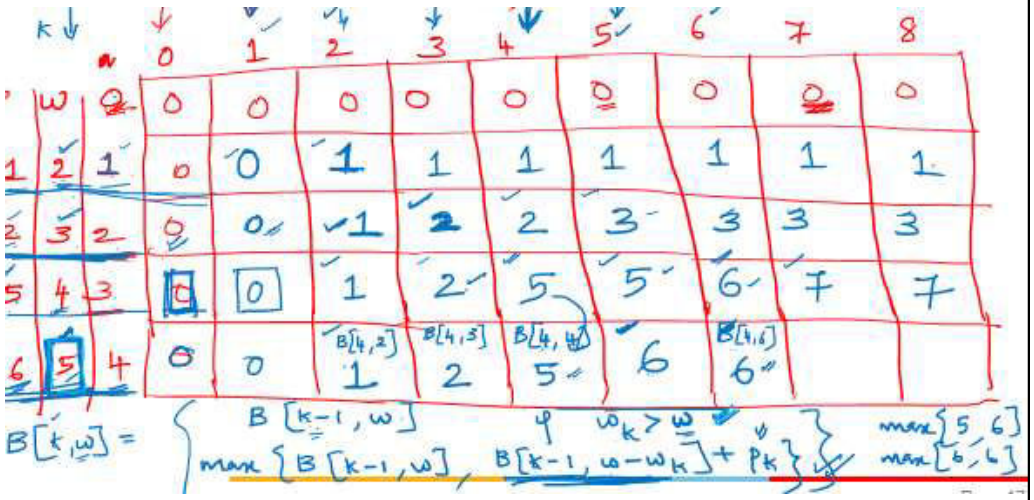


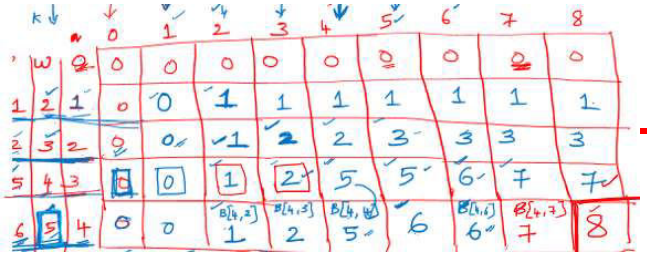
box of width 9 in

Solution:
- item 5 ($80, 2 kg)
- item 3 ($6, 2 kg)
- item 1 ($20, 4 kg)

Example

Example



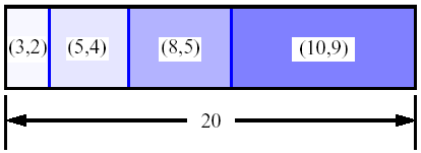


A 0/1 Knapsack Algorithm, First Attempt

- S_k: Set of items numbered 1 to k.
- Define B[k] = best selection from S_k.
- Problem: does not have subproblem optimality:
  - Consider set S={(3,2),(5,4),(8,5),(4,3),(10,9)} of (benefit, weight) pairs and total weight W = 20

Best for S_4:

Best for S_5:



A 0/1 Knapsack Algorithm, First Attempt

- Let S_k be the optimal subset of elements from $\{I_0, I_1, \ldots I_k\}$.
- The solution to the optimization problem for S_{k+1} might NOT contain the optimal solution from problem S_k.

A 0/1 Knapsack Algorithm, Second Attempt

- S_k: Set of items numbered 1 to k.
- Define B[k,w] to be the best selection from S_k with weight at most w
- Good news: this does have subproblem optimality.

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w],\ B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

- I.e., the best subset of S_k with weight at most w is either
  - the best subset of S_{k-1} with weight at most w or
  - the best subset of S_{k-1} with weight at most $w-w_k$ plus item k
  -

A 0/1 Knapsack Algorithm, Second Attempt

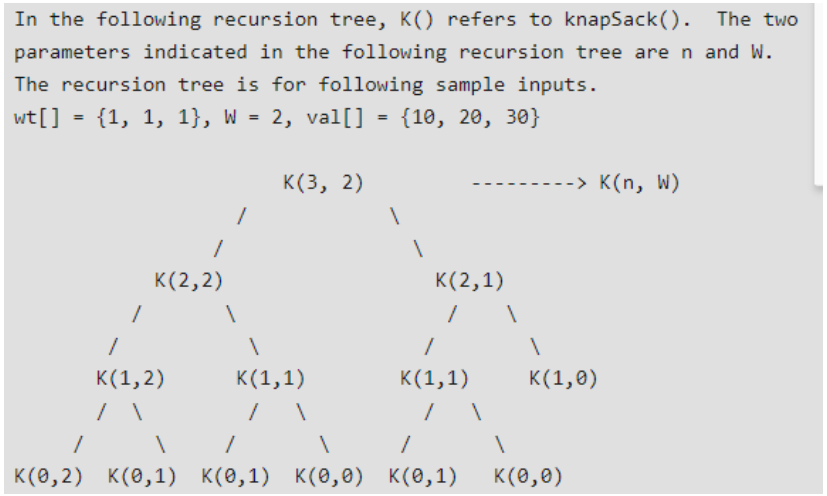
- **Case 1**
- The maximum value of a knapsack with a subset of items from $\{I_0, I_1, \ldots I_k\}$ with weight w is the same as the maximum value of a knapsack with a subset of items from $\{I_0, I_1, \ldots I_{k-1}\}$ with weight w, if item k weighs greater than w.
- Basically, you can NOT increase the value of your knapsack with weight w if the new item you are considering weighs more than w – because it WON'T fit!!!

A 0/1 Knapsack Algorithm, Second Attempt

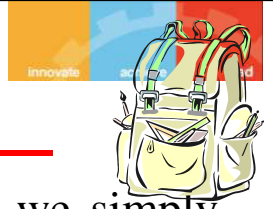
- **Case 2**
- The maximum value of a knapsack with a subset of items from $\{I_0, I_1, \ldots I_k\}$ with weight w could be the same as the maximum value of a knapsack with a subset of items from $\{I_0, I_1, \ldots I_{k-1}\}$ with weight $w-w_k$, plus item k.
- You need to compare the values of knapsacks in both case 1 and 2 and take the maximal one.

A 0/1 Knapsack Algorithm, Second Attempt

- Recursively, we will STILL have an $O(2^n)$ algorithm.

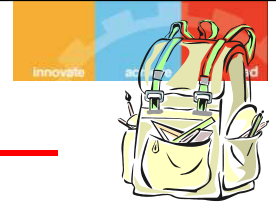


A 0/1 Knapsack Algorithm, Second Attempt



- But, using dynamic programming, we simply have to do a double loop - one loop running n times and the other loop running W times.

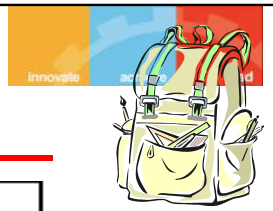
0/1 Knapsack Algorithm



0-1 Knapsack Algorithm

```
for w = 0 to W
    B[0,w] = 0
for i = 1 to n
    B[i,0] = 0
for i = 1 to n
    for w = 0 to W
        if w_i <= w // item i can be part of the solution
            if b_i + B[i-1,w-w_i] > B[i-1,w]
                B[i,w] = b_i + B[i-1,w- w_i]
            else
                B[i,w] = B[i-1,w]
        else B[i,w] = B[i-1,w]  // w_i > w
```


0/1 Knapsack Algorithm



Running time

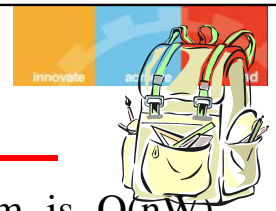
```
for w = 0 to W                    O(W)
    B[0,w] = 0
for i = 1 to n
    B[i,0] = 0
for i = 1 to n                    Repeat n times
    for w = 0 to W                O(W)
        < the rest of the code >
```

What is the running time of this algorithm?

O(n*W)

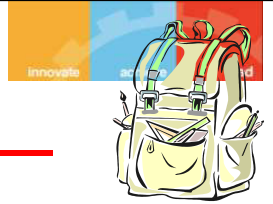
Remember that the brute-force algorithm takes $O(2^n)$

0/1 Knapsack Algorithm



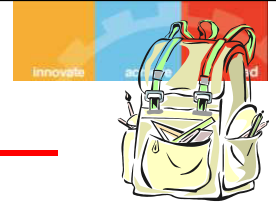
- Clearly the run time of this algorithm is O(nW), based on the nested loop structure and the simple operation inside of both loops.
- Depending on W, either the dynamic programming algorithm is more efficient or the brute force **$O(2^n)$,** algorithm could be more efficient.
- (For example, for n=5, W=100000, brute force is preferable, but for n=30 and W=1000, the dynamic programming solution is preferable.)

0/1 Knapsack Algorithm



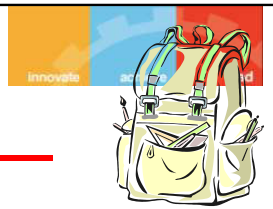
- **Pseudo-polynomial time algorithm**
- Running time of the algorithm depends on a parameter W that, strictly speaking, is not proportional to the size of the input (the n items, together with their weights and benefits, plus the number W).
- If W is very large (say $W = 2^n$), then this dynamic programming algorithm would actually be asymptotically slower than the brute force method.
- Thus, technically speaking, this algorithm is not a polynomial-time algorithm, for its running time is not actually a function of the size of the input

0/1 Knapsack Algorithm



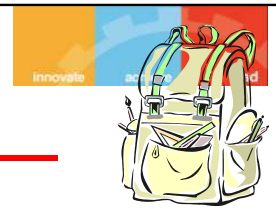
- **Pseudo-polynomial time algorithm**
- Running time depends on the magnitude of a number given in the input, not its encoding size

0/1 Knapsack Algorithm



Knapsack
Example Traced

0/1 Knapsack Algorithm-Example2



- **W = 10**

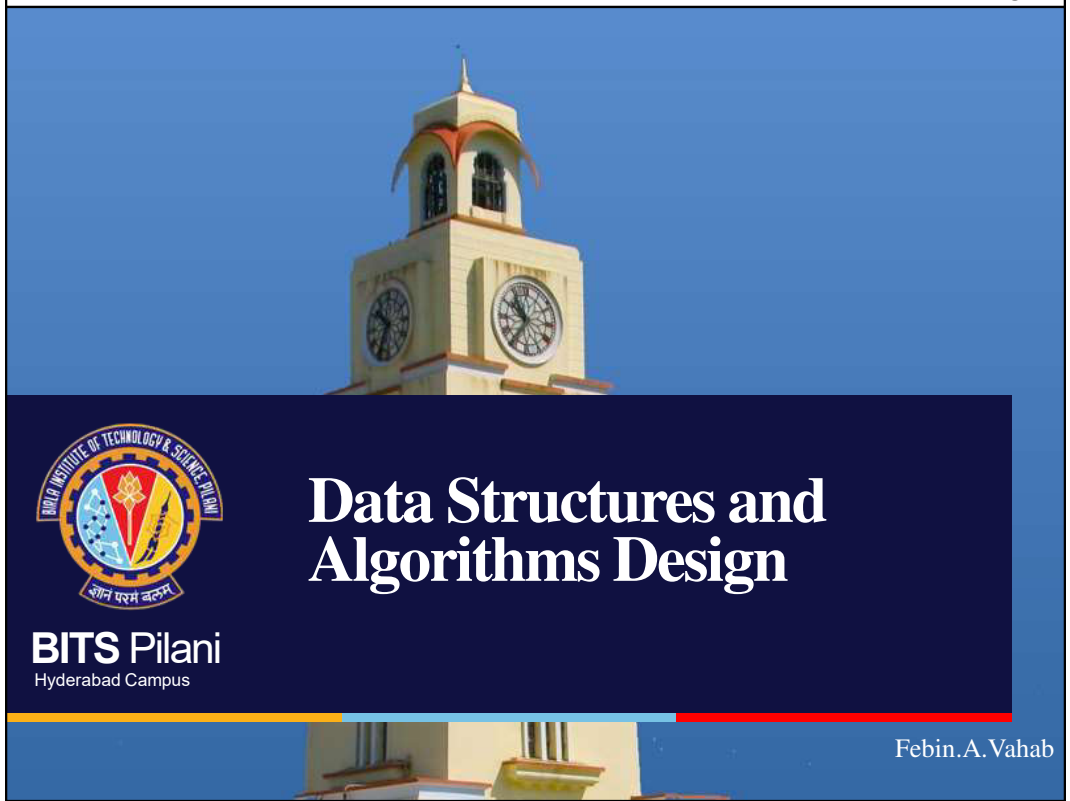
i	Item	w_i	v_i
0	I_0	4	6
1	I_1	2	4
2	I_2	3	5
3	I_3	1	3
4	I_4	6	9
5	I_5	4	7

0/1 Knapsack Algorithm-Example:Ans

Item	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	6	6	6	6	6	6	6
1	0	0	4	4	6	6	10	10	10	10	10
2	0	0	4	5	6	9	10	11	11	15	15
3	0	3	4	7	8	9	12	13	14	15	18
4	0	3	4	7	8	9	12	13	14	16	18
5	0	3	4	7	8	10	12	14	15	16	19

BITS Pilani
Hyderabad Campus

THANK YOU!



BITS Pilani
Hyderabad Campus

Data Structures and Algorithms Design

Febin.A.Vahab

ONLINE SESSION 9-PLAN

Sessions(#)	List of Topic Title	Text/Ref Book/external resource
9	Dynamic Programming - Design Principles and Strategy, Matrix Chain Product Problem, 0/1 Knapsack Problem, All-pairs Shortest Path Problem	T1: 5.3, 7.2

Dynamic Programming

- Invented by a prominent U.S. mathematician, Richard Bellman
- The word “programming” in the name of this technique stands for “planning” and does not refer to computer programming.

Dynamic Programming

- Dynamic programming is a technique for solving problems with overlapping subproblems.
- Typically, given problem’s solution can be related to solutions of its smaller subproblems.
- Rather than solving overlapping subproblems again and again, dynamic programming suggests solving each of the smaller subproblems only once and recording the results in a table from which a solution to the original problem can then be obtained.

Dynamic Programming

- A straightforward application of dynamic programming can be interpreted as a special variety of space-for-time trade-off.
- Optimisation of plain recursion.

Dynamic Programming-Example

- The Fibonacci numbers are the numbers in the following integer sequence.
- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144…
- F(n) = F(n-1) + F(n-2), F0 = 0 and F1 = 1

```
int fib(int n)
{
  if ( n <= 1 )
    return n;
  return fib(n-1) + fib(n-2);
}
```

Dynamic Programming-Example

- Time complexity:
- T(n) = T(n-1) + T(n-2)
- which is exponential.

Dynamic Programming-Properties

Overlapping Sub-problems:

- Sub-problems needs to be solved again and again.
- In recursion we solve those problems every time and in dynamic programming we solve these sub problems only once and store it for future use

Optimal Substructure:

- A problem can be solved by using the solutions of the sub problems

Dynamic Programming-Example

```
Algorithm DynamicFibonacci(n)
{
 f[0]=0,f[1]=1
 for(i =2;i<=n;i++)
      f[i]=f[i-1]+f[i-2]
return f[n];
}
```

Time complexity??

Dynamic Programming-Example

- A man put a pair of rabbits in a place surrounded by a wall. How many pairs of rabbits will be there in a year if the initial pair of rabbits (male and female) are newborn and all rabbit pairs are not fertile during their first month of life but thereafter give birth to one new male/female pair at the end of every month?

Dynamic Programming

- The circumstances and restrictions are not realistic.
- Still, this isn't THAT unrealistic a situation in the short term.

Dynamic Programming

- Similar to Fibonacci problem.
- To solve Fibonacci's problem, we'll let f(n) be the number of pairs during month n.
- By convention, f(0) = 0. f(1) = 1 for our new first pair.
- f(2) = 1 as well, as conception just occurred.
- The new pair is born at the end of month 2, so during month 3, f(3) = 2.
- Only the initial pair produces offspring in month 3, so f(4) = 3.

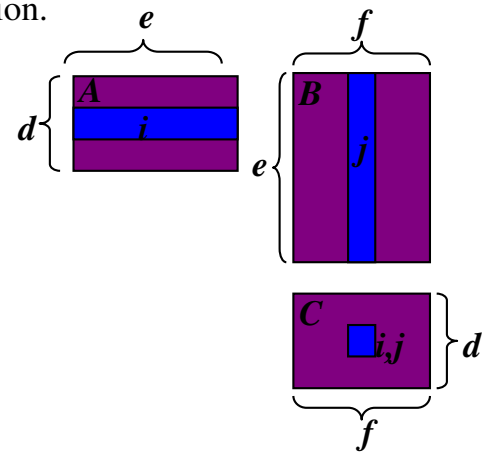
Dynamic Programming

- In month 4, the initial pair and the month 2 pair breed, so f(5) = 5. We can proceed this way, presenting the results in a table. At the end of a year, Fibonacci has 144 pairs of rabbits.

Month	0	1	2	3	4	5	6	7	8	9	10	11	12
Pairs	0	1	1	2	3	5	8	13	21	34	55	89	144

Matrix Chain-Products

- Review: Matrix Multiplication.
  - $C = A*B$
  - A is $d \times e$ and B is $e \times f$
  - $O(d \cdot e \cdot f)$ time



$$C[i,j] = \sum_{k=0}^{e-1} A[i,k] * B[k,j]$$

Matrix Multiplication

Matrix Chain-Products

- Matrix multiplication is associative
- $B \cdot (C \cdot D) = (B \cdot C) \cdot D$.
- Thus, we can parenthesize the expression for multiplication any way we wish and we will end up with the same answer.
- **Number of primitive (that is, scalar) multiplications in each parenthesization, however, might not be the same.**

Matrix Chain-Products

- Example
  - B is 3 × 100
  - C is 100 × 5
  - D is 5 × 5
  - (B*C)*D takes 1500 + 75 = 1575 ops
  - B*(C*D) takes 1500 + 2500 = 4000 ops

Matrix Chain-Products

- **Matrix Chain-Product:**
- Suppose we are given a collection of n two-dimensional matrices for which we wish to compute the product
  - Compute $\mathbf{A=A_1*\ldots*A_n}$
  - A_i is $d_{i-1} \times d_i$ matrix, for i = 1 , 2, . . . , n .
  - ie. Input A[]={10,20,30,40,50}
  - A1 is10x20 matrix,A2=20X30 matrix,A3 is 30X40 matrix and A4 is 40X50 matrix.
  - Problem: How to parenthesize?

Matrix Chain Product

- Input A[]={10,20,30,40,50}
- A1=10 X 20
- A2= 20 X 30
- A3=30 X 40
- A4= 40 X 50
- A_i is $d_{i-1} \times d_i$ Matrix
- $A_{i..j} = A_{i..k} \text{ X } A_{k+1..j}$
- $A_{1..4} = (A_{1..2}) \text{ X } (A_{3..4})$

Matrix Chain-Products

- The Matrix Chain-Products problem is to determine the parenthesization of the expression defining the product A that minimizes the total number of scalar multiplications performed.
- The problem is not actually to perform the multiplications, but merely to decide in which order to perform the multiplications.

Matrix Chain-Products-Enumeration Approach

- **Matrix Chain-Product Algorithm:**
  - Try all possible ways to parenthesize $A=A_1*\ldots*A_n$
  - Calculate number of ops for each one
  - Pick the one that is best

$$\begin{aligned} A_1A_2A_3A_4 &= (A_1A_2)(A_3A_4) \\ &= A_1(A_2(A_3A_4)) = A_1((A_2A_3)A_4) \\ &= ((A_1A_2)A_3)(A_4) = (A_1(A_2A_3))(A_4) \end{aligned}$$

Matrix Chain-Products-Enumeration Approach

- Running time:
  - The number of parenthesizations is equal to the number of binary trees with n nodes.
  - This is **exponential**!
  - It is called the Catalan number, and it is almost 4^n.
  - This is a terrible algorithm!

Number of Binary trees with n nodes

- Total number of possible Binary Search Trees with n different keys (countBST(n)) = Catalan number Cn = (2n)!/(n+1)!*n!
- For n = 0,1, 2, 3, … values of Catalan numbers are 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, …. So are numbers of Binary Search Trees.
- Total number of possible Binary Trees with n different keys (countBT(n)) = countBST(n) * n!

Matrix Chain-Products-Greedy Approach

- Idea #1: repeatedly select the product that uses (up) the most operations.
- Counter-example:
  - A is 10 × 5
  - B is 5 × 10
  - C is 10 × 5
  - D is 5 × 10
  - Greedy idea #1 gives (A*B)*(C*D), which takes 500+1000+500 = 2000 ops
  - A*((B*C)*D) takes 500+250+250 = 1000 ops

Matrix Chain-Products-Greedy Approach

- Idea #2: repeatedly select the product that uses the fewest operations.
- Counter-example:
  - A is 101 × 11
  - B is 11 × 9
  - C is 9 × 100
  - D is 100 × 99
  - Greedy idea #2 gives A*((B*C)*D)), which takes 109989+9900+108900=228789 ops
  - (A*B)*(C*D) takes 9999+89991+89100=189090 ops
- The greedy approach is not giving us the optimal value.

Example-Dynamic Programming

Example-Dynamic Programming

Example-Dynamic Programming

Example-Dynamic Programming

Example-Dynamic Programming

Example-Dynamic Programming

Example-Dynamic Programming

Example-Dynamic Programming

Matrix Chain-Products-Dynamic Programming

- Define **subproblems**:
  - Find the best parenthesization of $A_i*A_{i+1}*\ldots*A_j$.
  - Let $N_{i,j}$ denote the number of operations done by this subproblem.
  - The optimal solution for the whole problem is $N_{1,n}$.

Matrix Chain-Products-Dynamic Programming

- **Subproblem optimality**: The optimal solution can be defined in terms of optimal subproblems
  - There has to be a final multiplication (root of the expression tree) for the optimal solution.
  - Say, the final multiply is at index i: $(A_1*\ldots*A_i)*(A_{i+1}*\ldots*A_n)$.
  - Then the optimal solution $N_{1,n}$ is the sum of two optimal subproblems, $N_{1,i}$ and $N_{i+1,n}$ plus the time for the last multiply.
  - If the global optimum did not have these optimal subproblems, we could define an even better "optimal" solution.

Matrix Chain-Products-Characterizing Equation

- The global optimal has to be defined in terms of optimal subproblems, depending on where the final multiply is at.
- Let us consider all possible places for that final multiply:
  - Recall that A_i is a $d_{i-1} \times d_i$ dimensional matrix.
  - So, a characterizing equation for $N_{i,j}$ is the following:

$$N_{i,j} = \min_{i \le k < j} \{N_{i,k} + N_{k+1,j} + d_{i-1} d_k d_j\}$$

$$N_{i,i} = 0$$

Matrix Chain Products-Dynamic Programming Algorithm

```
Matrix-Chain(p, n)
{  for (i = 1 to n) m[i, i] = 0;
   for (l = 2 to n)
   {
      for (i = 1 to n − l + 1)
      {
         j = i + l − 1;
         m[i, j] = ∞;
         for (k = i to j − 1)
         {
            q = m[i, k] + m[k + 1, j] + p[i − 1] * p[k] * p[j];
            if (q < m[i, j])
            {
               m[i, j] = q;
               s[i, j] = k;
            }
         }
      }
   }
   return m and s; (Optimum in m[1, n])
}
```

Matrix Chain-Products-Dynamic Programming Algorithm

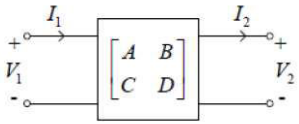
- The bottom-up construction fills in the N array by diagonals
- $N_{i,j}$ gets values from previous entries in i-th row and j-th column
- Filling in each entry in the N table takes O(n) time.
- Total run time: $O(n^3)$
- Getting actual parenthesization can be done by remembering "k" for each N entry.

Matrix Chain Products-Dynamic Programming Algorithm

- Since subproblems overlap, we don't use recursion.
- Instead, we construct optimal subproblems "bottom-up."
- $N_{i,i}$'s are easy, so start with them
- Then do problems of "length" 2,3,… subproblems, and so on.
- Running time: $O(n^3)$

Matrix Chain Products-Applications

- **Perspective projections**, which is the foundation for 3D animation-Orthographic projection (sometimes orthogonal projection) is a means of representing three-dimensional objects in two dimensions.
- Minimum and Maximum values of an expression with * and +
- **Network analysis (electrical circuits)- Network analysis** is the process of finding the voltages across, and the currents through, every component in the network. For example, when two or more N-port networks are connected in cascade, the combined network is the product on the individual ABCD matrices
- Used extensively in NLP and Machine learning: Principal Component Analysis and Singular Value Decomposition



Read about "Word to Vectors—Natural Language Processing"

Matrix Chain Products-Applications

The ABCD Matrix is defined as:

$$\begin{bmatrix} V_1 \\ I_1 \end{bmatrix} = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \cdot \begin{bmatrix} V_2 \\ I_2 \end{bmatrix}$$

For which:

$$A = \left.\frac{V_1}{V_2}\right|_{I_2=0}, \quad B = \left.\frac{V_1}{I_2}\right|_{V_2=0}$$

$$C = \left.\frac{I_1}{V_2}\right|_{I_2=0}, \quad D = \left.\frac{I_1}{I_2}\right|_{V_2=0}$$

"The usefulness of the ABCD matrix is that cascaded two port networks can be characterized by simply multiplying their ABCD matrices"

Matrix Chain Products-Applications

- Used extensively in NLP and Machine learning

Read about “Word to Vectors—Natural Language Processing”

- **sentence**=” Word Embeddings are Word converted into numbers ”
- A *word* in this **sentence** may be “Embeddings” or “numbers ” etc.
- A *dictionary* may be the list of all unique words in the **sentence.**
- So,a dictionary may look like [‘Word’, ’Embeddings’, ’are’, ’Converted’, ’into’, ’numbers’]
- A ve*ctor* representation of a word may be a one-hot encoded vector where 1 stands for the position where the word exists and 0 everywhere else.
- The vector representation of “numbers” in this format according to the above dictionary is [0,0,0,0,0,1] and of converted is[0,0,0,1,0,0].

Matrix Chain Products-Applications

D1: He is a lazy boy. She is also lazy.

D2: Neeraj is a lazy person.

The dictionary created may be a list of unique tokens(words) in the corpus =['He','She','lazy','boy','Neeraj','person']

Here, D=2, N=6

The count matrix M of size 2 X 6 will be represented as –

	He	She	lazy	boy	Neeraj	person
D1	1	1	2	1	0	0
D2	0	0	1	0	1	1

Matrix Chain Products-Applications

Co-occurrence matrix.

Corpus = He is not lazy. He is intelligent. He is smart.

	He	is	not	lazy	intelligent	smart
He	0	4	2	1	2	1
is	4	0	1	2	2	1
not	2	1	0	1	0	0
lazy	1	2	1	0	0	0
intelligent	2	2	0	0	0	0
smart	1	1	0	0	0	0

Principal Component Analysis and Singular Value Decomposition

Data Structures and Algorithms Design

BITS Pilani
Hyderabad Campus

ONLINE SESSION -PLAN

Sessions(#)	List of Topic Title	Text/Ref Book/external resource
10	**Dynamic Programming - Design Principles and Strategy, Matrix Chain Product Problem, 0/1 Knapsack Problem, All-pairs Shortest Path Problem**	**T1: 5.3, 7.2**

The General Dynamic Programming Technique

- Applies to a problem that at first seems to require a lot of time (possibly exponential), provided we have:
  - **Simple subproblems:** the subproblems can be defined in terms of a few variables, such as j, k, l, m, and so on.
  - **Subproblem optimality:** the global optimum value can be defined in terms of optimal subproblems
  - **Subproblem overlap:** the subproblems are not independent, but instead they overlap (hence, should be constructed bottom-up).

All-Pairs Shortest Paths

- Problem: Given a weighted connected graph (undirected or directed), the ***all-pairs shortest paths problem*** asks to find the distances—i.e., the lengths of the shortest paths—from each vertex to all other vertices.

- Floyd's algorithm computes the distance matrix of a weighted graph with n vertices through a series of $n \times n$ matrices:$D(0), \ldots, D(k-1), D(k), \ldots, D(n)$.

All-Pairs Shortest Paths

- Each of these matrices contains the lengths of shortest paths with certain constraints on the paths considered for the matrix in question.
- The element *d(k)ij* in the *i*th row and the *j*th column of matrix *D(k) (i, j* = 1, 2, . . . , *n,k* = 0, 1, . . . , *n)* is equal to the length of the shortest path among all paths from the *i*th vertex to the *j*th vertex with each intermediate vertex, if any, numbered not higher than *k.*
- In particular, the series starts with *D(*0*),* which does not allow any intermediate vertices in its paths, hence,*D(*0*)* is simply the weight matrix of the graph.

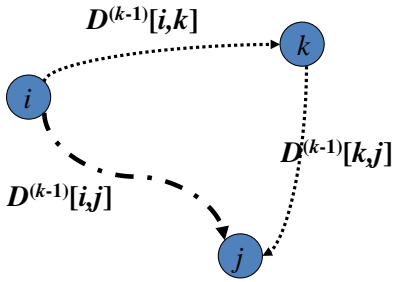
All-Pairs Shortest Paths

- The last matrix in the series, *D(n),* contains the lengths of the shortest paths among all paths that can use all *n* vertices as intermediate and hence is nothing other than the distance matrix being sought.

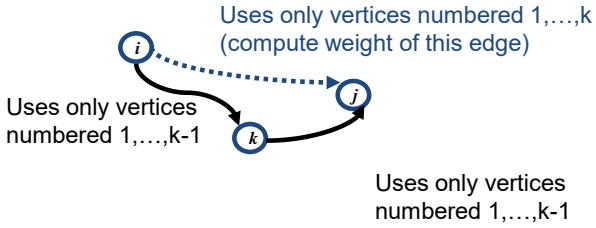
All-Pairs Shortest Paths-Floyd's Algorithm

- On the k-th iteration, the algorithm determines shortest paths between every pair of vertices i, j that use only vertices among 1,…,k as intermediate

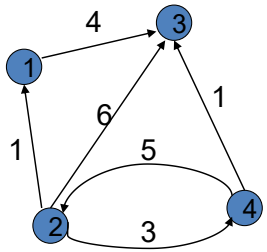
- $D^{(k)}[i,j] = \min \{D^{(k-1)}[i,j],\ D^{(k-1)}[i,k] + D^{(k-1)}[k,j]\}$



All-Pairs Shortest Paths-Floyd's Algorithm

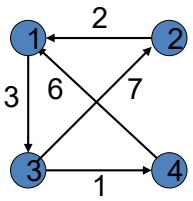


All-Pairs Shortest Paths



$$\begin{matrix} 0 & \infty & 4 & \infty \\ 1 & 0 & 4 & 3 \\ \infty & \infty & 0 & \infty \\ 6 & 5 & 1 & 0 \end{matrix}$$

All-Pairs Shortest Paths



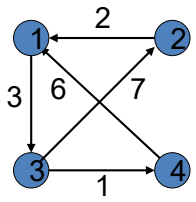
$D^{(0)} =$

$$\begin{matrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{matrix}$$

$D^{(1)} =$

$$\begin{matrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \mathbf{5} & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \mathbf{9} & 0 \end{matrix}$$

All-Pairs Shortest Paths



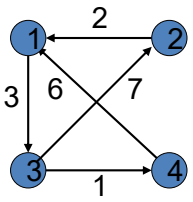
$D^{(1)} =$

$$\begin{matrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \mathbf{5} & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \mathbf{9} & 0 \end{matrix}$$

$D^{(2)} =$

$$\begin{matrix} 0 & \infty & 3 & \infty \\ 2 & 0 & 5 & \infty \\ \mathbf{9} & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{matrix}$$

All-Pairs Shortest Paths



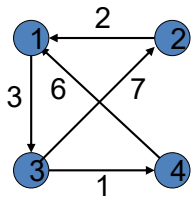
$D^{(2)} =$

$$\begin{matrix} 0 & \infty & 3 & \infty \\ 2 & 0 & 5 & \infty \\ \mathbf{9} & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{matrix}$$

$D^{(3)} =$

$$\begin{matrix} 0 & \mathbf{10} & 3 & \mathbf{4} \\ 2 & 0 & 5 & \mathbf{6} \\ 9 & 7 & 0 & 1 \\ 6 & \mathbf{16} & 9 & 0 \end{matrix}$$

All-Pairs Shortest Paths



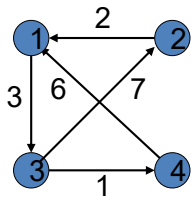
$D^{(3)}$ =

0	**10**	3	**4**
2	0	5	**6**
9	7	0	1
6	**16**	9	0

$D^{(4)}$ =

0	10	3	4
2	0	5	6
7	7	0	1
6	16	9	0

All-Pairs Shortest Paths



$D^{(0)}$ =

0	∞	3	∞
2	0	∞	∞
∞	7	0	1
6	∞	∞	0

$D^{(1)}$ =

0	∞	3	∞
2	0	**5**	∞
∞	7	0	1
6	∞	**9**	0

$D^{(2)}$ =

0	∞	3	∞
2	0	5	∞
9	7	0	1
6	∞	9	0

$D^{(3)}$ =

0	**10**	3	**4**
2	0	5	**6**
9	7	0	1
6	**16**	9	0

$D^{(4)}$ =

0	10	3	4
2	0	5	6
7	7	0	1
6	16	9	0

All-Pairs Shortest Paths-Floyd's Algorithm

ALGORITHM *Floyd(W[1..n, 1..n])*
//Implements Floyd's algorithm for the all-pairs shortest-paths problem
//Input: The weight matrix *W* of a graph with no negative-length cycle
//Output: The distance matrix of the shortest paths' lengths
D ←*W* //is not necessary if *W* can be overwritten
for *k*←*1* ***to*** *n* ***do***
for *i* ←*1* ***to*** *n* ***do***
for *j* ←*1* ***to*** *n* ***do***
D[i, j]←*min{D[i, j], D[i, k]+ D[k, j]}*
return *D*

All-Pairs Shortest Paths

- Find the distance between every pair of vertices in a weighted directed graph G.
- We can make n calls to Dijkstra's algorithm (if no negative edges), which takes O(n mlog n) time.
- Likewise, n calls to Bellman-Ford would take $O(n^2m)$ time.
- We can achieve $O(n^3)$ time using dynamic programming (similar to the Floyd-Warshall algorithm).

All-Pairs Shortest Paths-Negative Cycles

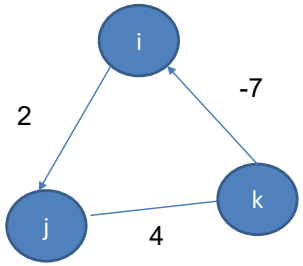
- Negative-weight edges may be present,
- But no negative-weight cycles.Why?

All-Pairs Shortest Paths-Negative Cycles

- A negative cycle is a cycle whose edges sum to a negative value.
- There is no shortest path between any pair of vertices i, j which form part of a negative cycle, because path-lengths from i to j can be arbitrarily small (negative)

All-Pairs Shortest Paths-Negative Cycles

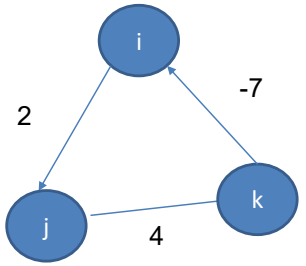
- Given a graph, suppose to have a cycle given by Nodes i , j, k of negative cost.
- Example:



- Suppose you want to find the shortest path between i and j

All-Pairs Shortest Paths-Negative Cycles

- You have P={(i,j)} with cost(P)=2.
- But you can loop through the negative cycle and have:



- P′={(i,j),(j,k),(k,i),(i,j)} with cost(P′)=1
- P″={(i,j),(j,k),(k,i),(i,j),(j,k),(k,i),(i,j)} with cost(P″)=0 and so on.

All-Pairs Shortest Paths-Negative Cycles

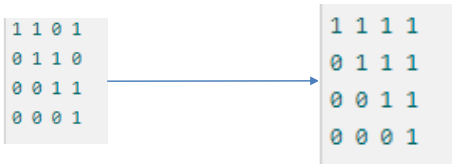
- Nevertheless, if there are negative cycles, the algorithm can be used to detect them.
- The intuition is as follows:
  - The Floyd's algorithm iteratively revises path lengths between all pairs of vertices (i,j), including where i=j
  - Initially, the length of the path (i,i) is zero;
  - A path {i,k,…i} can only improve upon this if it has length less than zero, i.e. denotes a negative cycle;
  - Thus, after the algorithm, (i,i) will be negative if there exists a negative-length path from i back to i.

All-Pairs Shortest Paths-Negative Cycles

ALGORITHM *Floyd(W[1..n, 1..n])*
//Implements Floyd's algorithm for the all-pairs shortest-paths problem
//Input: The weight matrix *W* of a graph with no negative-length cycle
//Output: The distance matrix of the shortest paths' lengths
D ←*W* //is not necessary if *W* can be overwritten
for *k*←*1* ***to*** *n* ***do***
for *i* ←*1* ***to*** *n* ***do***
for *j* ←*1* ***to*** *n* ***do***
D[i, j]←*min{D[i, j], D[i, k]+ D[k, j]}*
for i = 1 to n do
if D[i, i] < 0 then return('graph contains a negative cycle')
return *D*

Transitive Closure

ALGORITHM *Warshall*$(A[1..n, 1..n])$
//Implements Warshall's algorithm for computing the transitive closure
//Input: The adjacency matrix A of a digraph with n vertices
//Output: The transitive closure of the digraph
$R^{(0)} \leftarrow A$
for $k \leftarrow 1$ **to** n **do**
  for $i \leftarrow 1$ **to** n **do**
    for $j \leftarrow 1$ **to** n **do**
      $R^{(k)}[i, j] \leftarrow R^{(k-1)}[i, j]$ **or** $(R^{(k-1)}[i, k]$ **and** $R^{(k-1)}[k, j])$
return $R^{(n)}$



Example

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	0	1	0

Example

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	1	1	0

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	1	1	0

Example

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

Example

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

Data Structures and Algorithms Design

BITS Pilani
Hyderabad Campus

Febin.A.Vahab

CONTACT SESSION 14 -PLAN

Contact Sessions(#)	List of Topic Title	Text/Ref Book/external resource
14	**Definition of P and NP classes and examples, Understanding NP-Completeness: CNF-SAT Cook-Levin theorem** **Polynomial time reducibility: CNF-SAT and 3-SAT, Vertex Cover**	**T1: 13.1, 13.2, 13.3**

P and NP classes

Polynomial Time	Exponential Time
Linear Search O(n)	0/1 Knapsack O(2^n)
Binary Search O(log n)	Tower of Hanoi O(2^n)
Merge Sort O(nlogn)	TSP O(2^n)
Quicksort O(nlogn),O(n^2)	Sum of Subsets O(2^n)
Fractional KS O(nlogn)	
Task Scheduling O(n^2)	
Kruskal's MST O(mlogn)	

Decision and Optimization Problems

- **Decision Problem**: computational problem with intended output of "yes" or "no", 1 or 0
- **Optimization Problem**: computational problem where we try to maximize or minimize some value.
- Introduce parameter k and ask if the optimal value for the problem is at most or at least k. Turn optimization into decision

Decision and Optimization Problems

- Example: Hamiltonian circles:
- A Hamiltonian circle in an undirected graph is a simple circle that passes through every vertex exactly once.
- The **decision** problem is: Does a given undirected graph has a Hamiltonian circle?

Decision and Optimization Problems

- Many problems will have decision and optimization versions
- **Traveling salesman problem**
- Optimization problem: Given a weighted graph, find Hamiltonian cycle of minimum weight.
- Decision problem: Given a weighted graph and an integer k, is there a Hamiltonian cycle with total weight at most k?

Decision and Optimization Problems

- **Knapsack** : Suppose we have a knapsack of capacity W and n objects with weights w1, …, wn, and values v1, …, vn
- **Optimization problem**: Find the largest total value of any subset of the objects that fits in the knapsack (and find a subset that achieves the maximum value)
- **Decision problem**: Given k, is there a subset of the objects that fits in the knapsack and has total value at least k?

Decision and Optimization Problems

- **Graph coloring**: assign a color to each vertex so that adjacent vertices are not assigned the same color. Chromatic number: the smallest number of colors needed to color G.
- We are given an undirected graph G = (V, E) to be colored.
- **Optimization problem**: Given G, determine the chromatic number .
- **Decision problem**: Given G and a positive integer k, is there a coloring of G using at most k colors? If so, G is said to be k-colorable

Decision and Optimization Problems

- **Decision Problem**: computational problem with intended output of "yes" or "no", 1 or 0

  Examples:
  - Does a text T contain a pattern P?
  - Does an instance of 0/1 Knapsack have a solution with benefit at least K?
  - Does a graph G have an MST with weight at most K?

Polynomial-Time

- The class P consists of those problems that are solvable in polynomial time.
- More specifically, they are problems that can be solved in time $O(n^k)$ for some constant k, where n is the size of the input to the problem
- The key is that n is the **size of input**

Problems and Languages

- A **language** L is a set of strings defined over some alphabet Σ
- Every decision algorithm A defines a language L
  - L is the set consisting of every string x such that A outputs "yes" on input x.
  - We say "A **accepts** x" in this case
    - Example:
    - If A determines whether or not a given graph G has an Euler tour, then the language L for A is all graphs with Euler tours.

The Complexity Class P

- A **complexity class** is a collection of languages
- P is the complexity class consisting of all languages that are accepted by **polynomial-time** algorithms
- For each language L in P there is a polynomial-time decision algorithm A for L.
  - If n=|x|, for x in L, then A runs in p(n) time on input x.
  - The function p(n) is some polynomial

The Complexity Class NP

- **NP is not the same as non-polynomial complexity/running time.**
- **NP does not stand for not polynomial.**
- **NP = Non-Deterministic polynomial time**

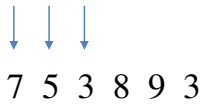
Deterministic

- In a deterministic computer we can determine in advance for every computational cycle, the output state by looking at the input state
- In **deterministic algorithm**, for a given particular input, the computer will always produce the same output going through the same states

Deterministic

```
//input: an array A[1..N] and an element el that is in the array
//output: the position of el in A
search(el, A, i)
{
   if (A[i] = el) then return i
   else
        return search(el, A, i+1)
}
```

Complexity: O(N)



Non Deterministic

- In a nondeterministic computer we may have more than one output state.
- The computer "chooses" the correct one ("nondeterministic choice")
- For the same input, the compiler may produce different output in different runs.
- Non-deterministic algorithms can't solve the problem in polynomial time and can't determine what is the next step

Non Deterministic

- Non-deterministic algorithms can show different behaviors for the same input on different execution and there is a degree of randomness to it.
- A **nondeterministic algorithm** for a problem **A** is a two-stage procedure. **In the first phase**, a procedure makes a guess about the possible solution for A. **In the second phase**, a procedure checks if the guessed solution is indeed a solution for A

Non Deterministic

- **Search an element x on A[1:n] where n>=1, on successful search return j if a[j] is equals to x otherwise return 0.**

```
j= choice(a, n)
if(A[j]==x) then
  {
    write(j);
    success();
  }
write(0); failure();
```

Complexity: O(1)

Non Deterministic

- **choice(X) :** chooses any value randomly from the set X.
- **failure() :** denotes the unsuccessful solution.
- **success() :** Solution is successful and current thread terminates.

ND Sorting

Non Deterministic Sorting

```
1.  Algorithm Nsort(A,N)
2.  // sort n positive numbers
3.  {
4.  For I = 1 to n do B[i] = 0 ; //
    initialize B
5.  For I = 1  to n do
6.  {
7.  J = choice(1,n);
8.  If B[j] != 0 then failure();
9.  B[j] = A[i];
10. }
11. For I = 1 to n-1 do // verify the order
12. If B[i] > B[i+1] then failure ();
13. Write (B);
14. Success();
15. }
```

- In the for loop of 5 to 10 each A[i] is assigned to position in B
- Line 7 non deterministically identifies this position
- Line 8 checks whether this position is already used or not !
- The order of numbers in B is some permutation of the initial order in A
- For loop of line no 11 and 12 verifies the B is in ascending order,
- Since there is always a set of choices at line no 7 for ascending order, this is a sorting algorithm of Complexity O(N)
- All deterministic sorting algorithms must have A complexity O(nlogn)

The Complexity Class NP

- The class **NP** consists of all problems that can be solved in polynomial time by **nondeterministic algorithms**
- That is, both phase 1 and phase 2 run in polynomial time
- If A is a problem in P then A is a problem in NP because
  - Phase 1: use the polynomial algorithm that solves A
  - Phase 2: write a constant time procedure that always returns true

The Complexity Class NP

- Every decision problem solvable by a polynomial time deterministic algorithm is also solvable by a polynomial time nondeterministic algorithm.
- To see this, observe that any deterministic algorithm can be used as the checking stage of a nondeterministic algorithm.
- If $I \in P$, and A is any polynomial deterministic algorithm for I, we can obtain a polynomial nondeterministic algorithm for I merely by using A as the checking stage and ignoring the guess.
- Thus $I \in P$ implies $I \in NP$

The Complexity Class NP

- The key question is are there problems in NP that are not in P or is P = NP?
- We don't know the answer to the previous question
- We say that a non-deterministic algorithm A **accepts** a string x if there exists some sequence of choose operations that causes A to output "yes" on input x.
- NP is the complexity class consisting of all languages accepted by **polynomial-time non-deterministic** algorithms.

The Complexity Class NP

- How to prove that a problem *A* is in NP:

  Show that *A* is in P

  OR

  Write a nondeterministic algorithm solving A that runs in polynomial time

NP example

Problem: Decide if a graph has an MST of weight K

Algorithm:

1. Non-deterministically choose a set T of n-1 edges
2. Test that T forms a spanning tree
3. Test that T has weight at most K

Analysis: Testing takes O(n+m) time, so this algorithm runs in polynomial time.

The Complexity Class NP Alternate Definition

- We say that an algorithm B **verfies** the acceptance of a language L if and only if, for any x in L, there exists a certificate y such that B outputs "yes" on input (x,y).
- NP is the complexity class consisting of all languages verified by **polynomial-time** algorithms.

- We know: P is a subset of NP.
- Major open question: P=NP?
- Most researchers believe that P and NP are different

Satisfiability Problem

- **A Boolean formula is *satisfiable* if there exists some assignment of the values 0 and 1 to its variables that causes it to evaluate to 1.**
- CNF – Conjunctive Normal Form. ANDing of clauses of ORs

$$(x_0 \vee x_2) \wedge (\neg x_0 \vee x_1) \wedge (x_0 \vee x_1 \vee \neg x_2)$$

CNF satisfiability

- This problem is in *NP*. Nondeterministic algorithm:
  - Guess truth assignment
  - Check assignment to see if it satisfies CNF formula

  - It is easy to show that CNF-SAT is in NP
  - For, given a Boolean formula S, we can construct a simple nondeterministic algorithm that first "guesses" an assignment of Boolean values for the variables in S and then evaluates each clause of S in turn.
  - If all the clauses of S evaluate to 1, then S is satisfied; otherwise, it is not.

CNF satisfiability

- Example:
  $(A \vee \neg B \vee \neg C) \wedge (\neg A \vee B) \wedge (\neg B \vee D \vee F) \wedge (F \vee \neg D)$
- Truth assignments:

	A	B	C	D	E	F
1.	0	1	1	0	1	0
2.	1	0	0	0	0	1
3.	1	1	0	0	0	1
4.	... (how many more?)					

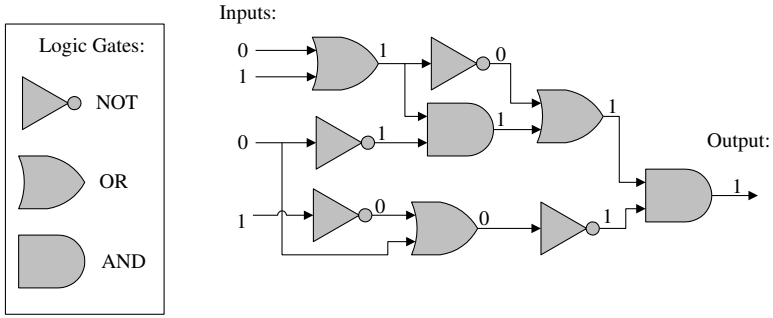
Checking phase: $\Theta(n)$

3 SAT

- Problem: Given a CNF where each clause has 3 variables, decide whether it is satisfiable or not.
- $(x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$
- 3SAT is NP Complete

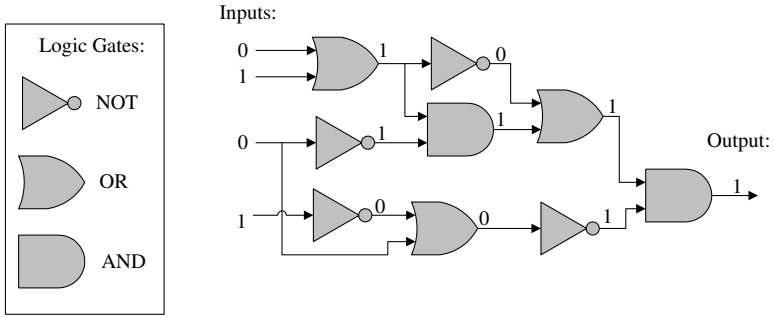
An Interesting Problem

- A Boolean circuit is a circuit of AND, OR, and NOT gates; the CIRCUIT-SAT problem is to determine if there is an assignment of 0's and 1's to a circuit's inputs so that the circuit outputs 1.



CIRCUIT-SAT is in NP

- Non-deterministically choose a set of inputs and the outcome of every gate, then test each gate's I/O.



P And NP Summary

- **P** = set of problems that can be solved in polynomial time
  - Examples: Fractional Knapsack, …
- **NP** = set of problems for which a solution can be verified in polynomial time
  - Examples: 0/1 Knapsack,…, Hamiltonian Cycle, CNF SAT, 3-CNF SAT
- Clearly **P** $\subseteq$ **NP**
- Open question: Does **P = NP**?
  - Most suspect not
  - An August 2010 claim of proof that P ≠ NP, by Vinay Deolalikar, researcher at HP Labs, Palo Alto, has flaws

Polynomial-Time Reducibility

- Language L is polynomial-time reducible to language M if there is a function computable in polynomial time that takes an input x of L and transforms it to an input f(x) of M, such that x is a member of L if and only if f(x) is a member of M.
- Shorthand, $L^{poly}M$ means L is polynomial-time reducible to M

Polynomial-Time Reducibility

- A problem R can be *reduced* to another problem Q if any instance of R can be rephrased to an instance of Q, the solution to which provides a solution to the instance of R
  - This rephrasing is called a *transformation*
- Intuitively: If R reduces in polynomial time to Q, R is "no harder to solve" than Q
- Example: lcm(m, n) = m * n / gcd(m, n),
  lcm(m,n) problem is reduced to gcd(m, n) problem

Polynomial-Time Reducibility

- For example a Hamiltonian circuit problem is polynomially reducible to the decision version of the traveling salesman problem.
- Hamiltonian circuit problem(HCP): Does a given undirected graph have a Hamiltonian cycle?
- Traveling salesman problem(TSP): Given a weighted graph and an integer k, is there a Hamiltonian cycle with total weight at most k?

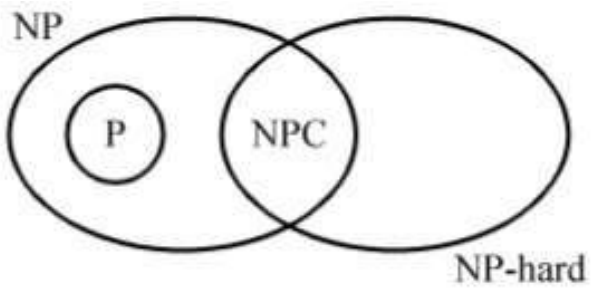
NP Hard and NP complete

- A decision problem *D* is *NP*-complete iff
  1. ***D* ∈ *NP***
  2. **every problem in *NP* is polynomial-time reducible to *D***

NP Hard and NP complete

- If R is *polynomial-time reducible* to Q, we denote this $R \leq_p Q$
- Definition of NP-Hard and NP-Complete:
  - If all problems R ∈ **NP** are *polynomial-time* reducible to Q, then Q is *NP-Hard*
  - We say Q is *NP-Complete* if Q is NP-Hard and Q ∈ **NP**
- If $R \leq_p Q$ and R is NP-Hard, Q is also NP-Hard

NP Hard and NP complete



NP Hard and NP complete

- Belief: P is a proper subset of NP.
- Implication: the NP-complete problems are the hardest in NP.
- Why: Because if we could solve an NP-complete problem in polynomial time, we could solve every problem in NP in polynomial time.
- That is, if an NP-complete problem is solvable in polynomial time, then P=NP.
- Since so many people have attempted without success to find polynomial-time solutions to NP-complete problems, showing your problem is NP-complete is equivalent to showing that a lot of smart people have worked on your problem and found no polynomial-time algorithm

Cook's Theorem

Cook's theorem

NP = P iff the satisfiability problem is a P problem.

- SAT is NP-complete.
- It is the first NP-complete problem.
- Every NP problem reduces to SAT.

Stephen Arthur Cook

Amusing analogy
(thanks to lecture notes at University of Utah)

- Students believe that every problem assigned to them is NP-complete in difficulty level, as they have to find the solutions. ☺
- Teaching Assistants, on the other hand, find that their job is only as hard as NP, as they only have to verify the student's answers.
- When some students confound the TAs, even verification becomes hard

Data Structures and Algorithms Design

BITS Pilani
Hyderabad Campus

Febin.A.Vahab

AVL trees

- From previous lectures:
  - Binary search trees store linearly ordered data
  - Best case height: O(log(n))
  - Worst case height: O(n)
- Requirement:
  - Define and maintain a balance to ensure O(log(n)) operations

AVL trees

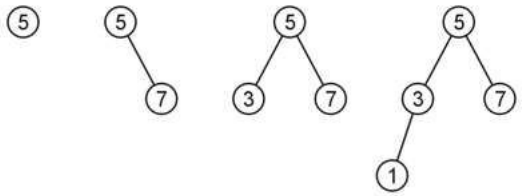
- The AVL tree is the first balanced binary search tree ever invented.
- It is named after its two inventors, G.M. Adelson-Velskii and E.M. Landis, who published it in their 1962 paper "An algorithm for the organization of information."

AVL trees

- AVL trees are balanced
- An AVL Tree is a binary search tree such that for every internal node v of T, the heights of the children of v can differ by at most 1
- This difference is called the **Balance Factor**.
- For an AVL tree |balance factor|<=1 for all the nodes.

AVL trees

- **BalanceFactor=height(left-subtree)− height(right-subtree)**



AVL trees with 1 ,2,3,and 4 nodes

AVL trees

- **BalanceFactor=height(left-subtree)− height(right-subtree)**

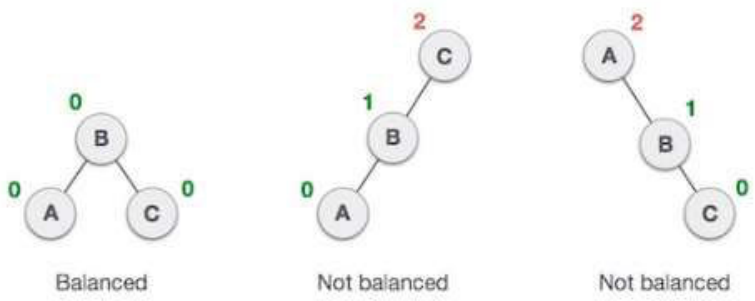


Image credit: Tutorials point

AVL Trees-Example

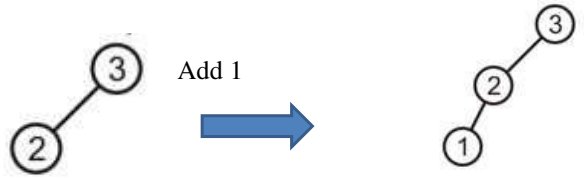
AVL trees-Rotations

- To balance itself, an AVL tree may perform the following four kinds of rotations −
  - Left rotation
  - Right rotation
  - Left-Right rotation
  - Right-Left rotation

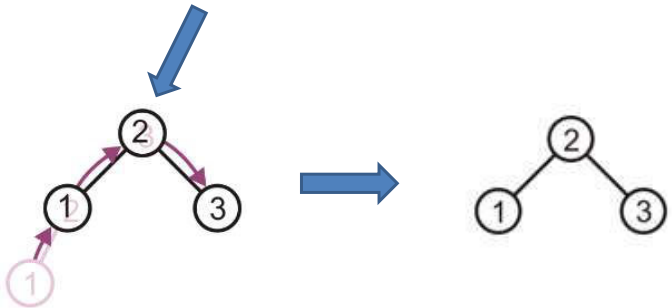
  - To have an unbalanced tree, we at least need a tree of height 2.

AVL trees-Rotations

- **Right Rotation** Node is inserted in the left of left-subtree



The node 3 has become unbalanced



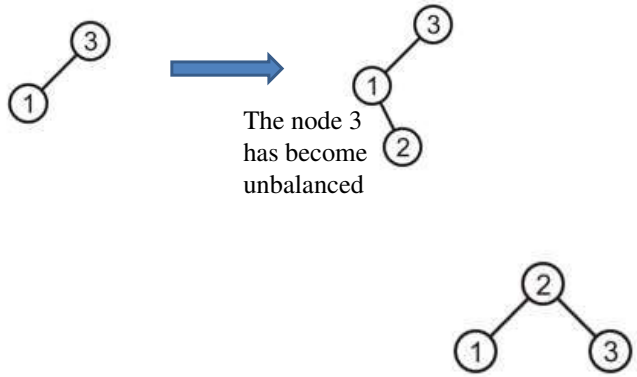
Promote 2 to the root, demote 3 to be 2's right child, and 1 remains the left child of 2

AVL trees-Rotations

- **Left Rotation:** Node is inserted into right of right subtree. After inserting new node, tree becomes unbalanced

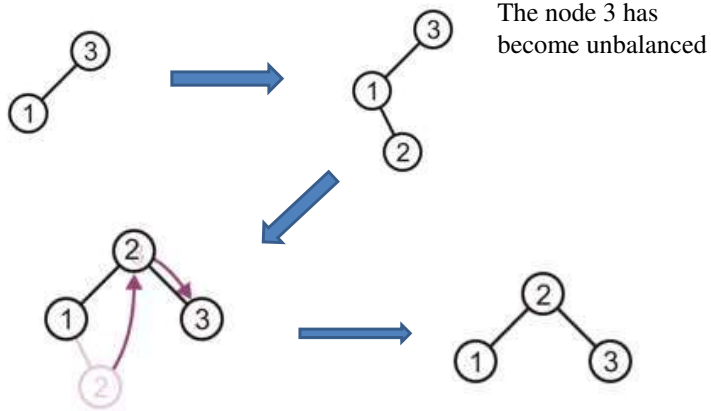
AVL trees-Rotations

- **Left Right Rotation <u>(Double Rotation):</u>** Node is inserted in the right of left-subtree and makes the tree unbalanced.



AVL trees-Rotations

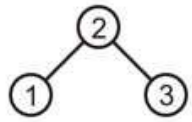
- **Left Right Rotation**



Promote 2 to the root, and assign
1 and 3 to be its children

AVL trees-Rotations

- **Right Left Rotation:** Node is inserted in the left of right subtree and make the tree unbalanced



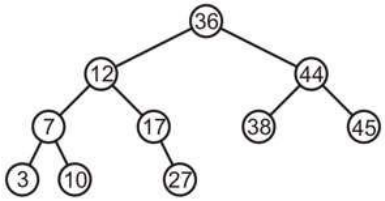
AVL Trees-General Case
LL Imbalance

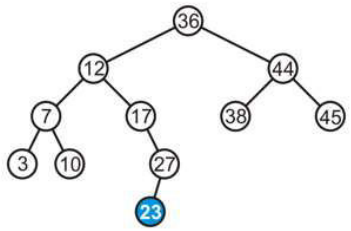
AVL Trees-General Case
LL Imbalance

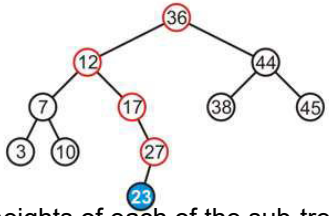
AVL Tree-General Case
LR imbalance

AVL Tree-General Case
LR imbalance

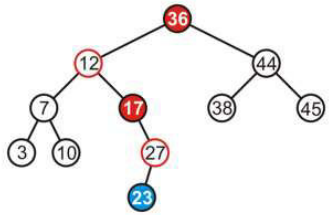
AVL Insertion-Case 1





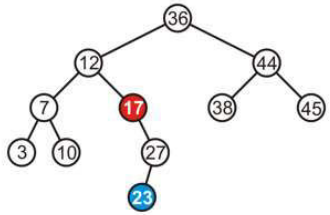


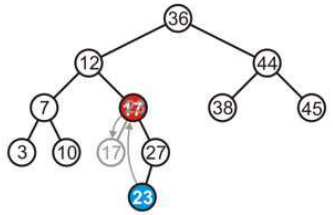
The heights of each of the sub-trees from here to the root are increased by one



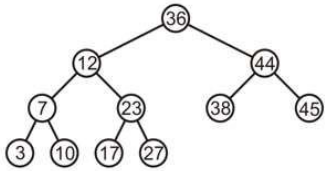
only two of the nodes are unbalanced: 17 and 36

AVL Insertion





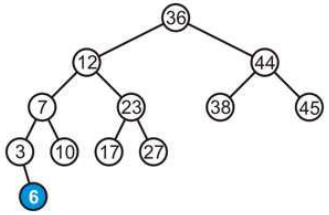
We only have to fix the imbalance at the lowest node



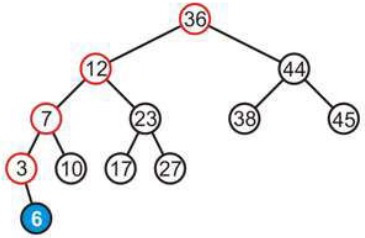
That node is no longer unbalanced. Incidentally, neither is the root .Now balanced again.

AVL Insertion-Case 2

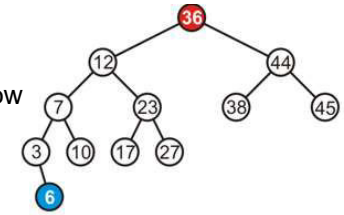
Consider adding 6



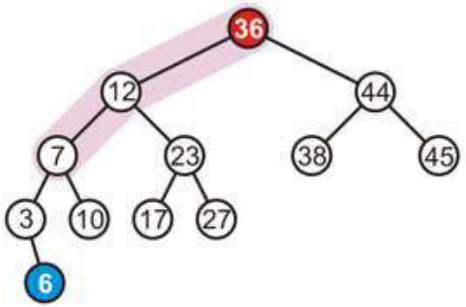
The height of each of the trees in the path back to the root are increased by one

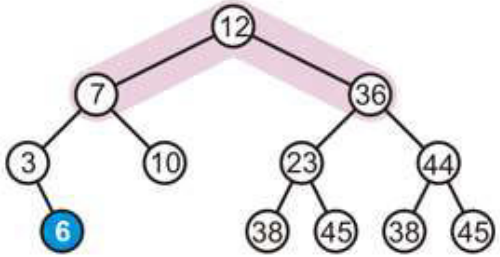


However, only the root node is now unbalanced



AVL Insertion-Case 2





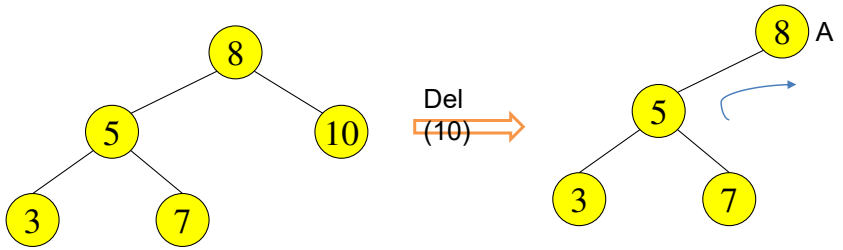
AVL Tree-Creation

Delete an element from AVL Trees

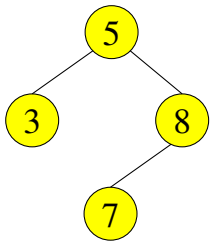
- We first do the normal BST deletion:
  - 0 children: just delete it
  - 1 child: delete it, connect child to parent
  - 2 children: put the inorder successor in node's place
- Calculate Balance Factor again
- A is the critical node whose balance factor is disturbed upon deleting node x.
- If deleted node are from left subtree of A then It is called **Type L** delete otherwise it is called **Type R** delete

Delete an element from AVL Trees R0 Rotation

R0 Rotation

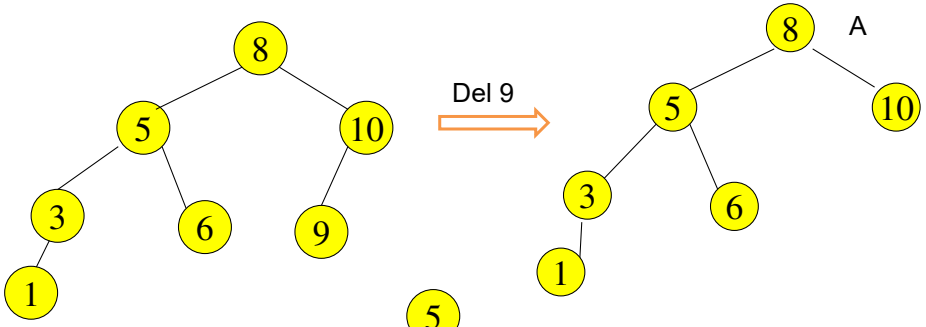


Apply right rotation on A

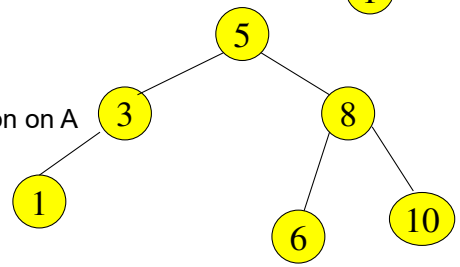


Delete an element from AVL Trees
R1 Rotation

R1 Rotation

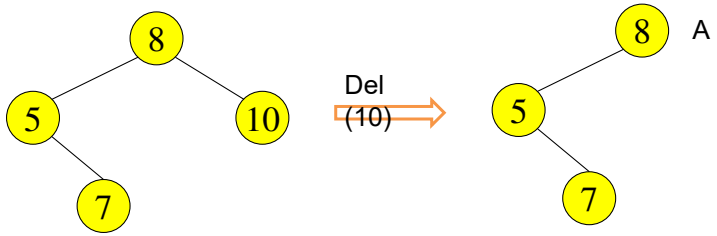


Apply right rotation on A

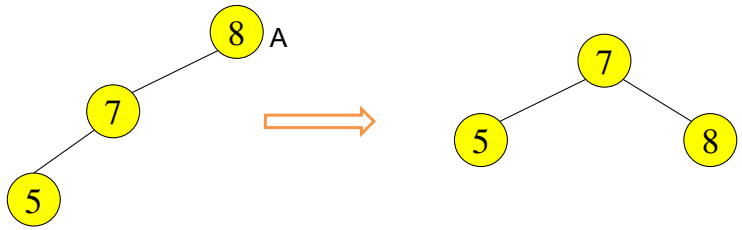


Delete an element from AVL Trees
R -1 Rotation

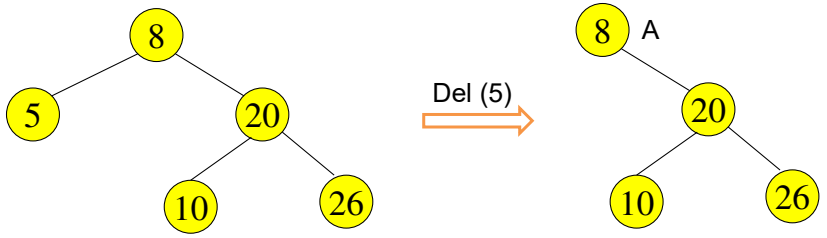
Type R-1 Rotation



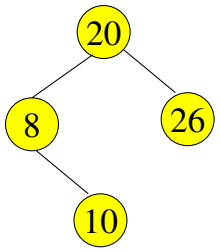
Apply left Rotation on left child of node A and Then right rotation on node A



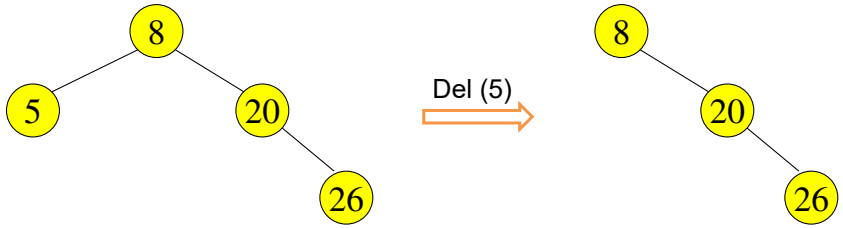
Type L



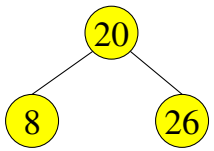
Since it L(0) type apply Left rotation on A



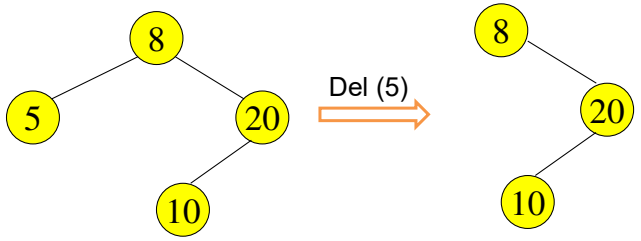
Type L



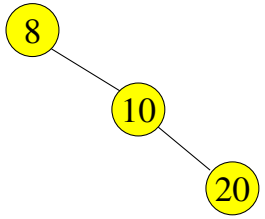
Since it L(-1) = case apply Left rotation on A



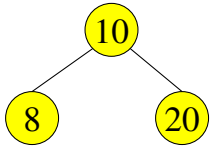
Type L



Since it L(1), In L_1 case we have to solve in two steps,
Step1: Right Rotation at right child of 'A'



Step2: Left rotation at node A



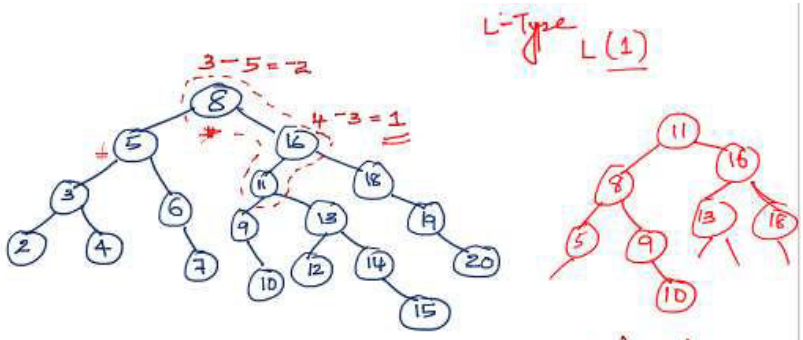
AVL trees-Deletions

- Removing a node from an AVL tree may cause more than one AVL imbalance
- Like insert, remove must check after it has been successfully called on a child to see if it caused an imbalance
- Unfortunately, it may cause O(h) imbalances that must be corrected
- Insertions will only cause one imbalance that must be fixed
- But in removal , a single trinode restructuring may not restore the height-balance property globally
- So, after rebalancing , we continue walking up T looking for unbalanced nodes.
- If we find another, we perform a restructure operation to restore its balance, and continue marching up T looking for more, all the way to the root

AVL trees-Deletions

AVL trees-Deletions

AVL trees-Deletions



Now Node 5 is balanced. But node 8 is now unbalanced.
So perform a left rotation. (L1 Rotation)

AVL trees-Deletions

AVL trees-Applications

- AVL trees are applied in the following situations:
  - There are few insertion and deletion operations
  - Short search time is needed

AVL Trees-Summary

- AVL balance is defined by ensuring the difference in heights is 0 or 1
- Insertions and Removals are like binary search trees
- Each insertion requires at least one correction to maintain AVL balance
- Removals may require O(h) corrections
- These corrections require Q(1) time
- Height of the AVL tree is O(log(n))
- ∴ all O(h) operations are O(log(n))

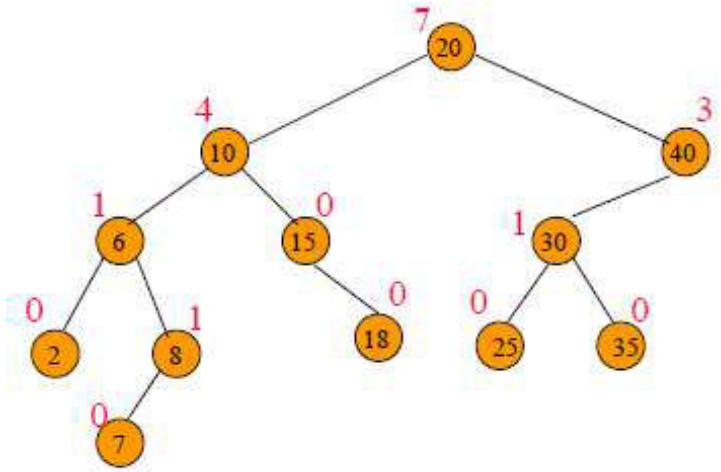
Find k-th smallest element in BST

- The idea is to maintain rank of each node.
- We can keep track of elements in a subtree of any node while building the tree.
- Since we need K-th smallest element, we can maintain number of elements of left subtree in every node.

Rank

- Rank of an element is its position in inorder traversal (inorder = ascending key order).
- [2,6,7,8,10,15,18,20,25,30,35,40]
- rank(2) = 0
- rank(15) = 5
- rank(20) = 7
- **leftSize(x) = rank(x) with respect to elements in subtree rooted at x**

Rank



sorted list = [2,6,7,8,10,15,18,20,25,30,35,40]

Find k-th smallest element in BST

- Assume that the root is having N nodes in its left subtree.
  - If K = N + 1, root is K-th node.
  - If K > N, we continue our search in the right subtree for the (K – (N + 1))-th smallest element.
  - Else we will continue our search (recursion) for the Kth smallest element in the left subtree of root.
  - Note that we need the count of elements in left subtree only.
- Time complexity: O(h) where h is height of tree.

Find k-th smallest element in BST

1. *start*
2. ***if K = root.leftElements + 1***
   1. *root node is the K th node.*
   2. *goto stop*
3. ***else if K > root.leftElements***
   1. *K = K - (root.leftElements + 1)*
   2. *root = root.right*
   3. *goto start*
4. ***else***
   1. *root = root.left*
   2. *goto start*
5. *stop*


THANK YOU!!!
BITS Pilani
Hyderabad Campus