MADF Unit IV

Prof. Amrita Naik
Assistant Professor
DBCE, Goa

String Matching Algorithm

- 3 Types of Pattern matching algorithm:
- 1.Brute Force Algorithm
- 2.KMP Algorithm
- 3.Boyer-Moore Algorithm

Brute Force Algorithm

```
for i= 0 to n-m
    {
     for j=0 to m
        {
                    if (txt[i + j] != pat[j])
                              break;
        }
    }
     if (j == m)
         print("Pattern found at index I")
```

Time complexity=T(nm)

Pattern Matching-Brute Force

Brute Force Algorithm: → Time comp

ex: Pattern matching using Brute Force Algorithm

T=	a	b	a	c	a	a	b	a	c	c	a	b	a	c	a	b	a	a	b	b
P=	a	b	a	c	a	b														
		a	b	a	c	a	b													
			a	b	a	c	a	b												
				a	b	a	c	a	b											
					a	b	a	c	a	b										
						a	b	a	c	a	b									
							a	b	a	c	a	b								
								a	b	a	c	a	b							
									a	b	a	c	a	b						
										a	b	a	c	a	b					
substring exist											$\boxed{a}$	$\boxed{b}$	$\boxed{a}$	$\boxed{c}$	$\boxed{a}$	$\boxed{b}$				
												a	b	a	c	a	b			
													a	b	a	c	a	b		
														a	b	a	c	a	b	
															a	b	a	c	a	b

KMP Matching

Ex 1) Consider a text string T=abacaabaccabacabaabb and pattern string P=abacab. Perform KMP pattern search on it.

Sol: failure function

j	0	1	2	3	4	5
P	a	b	a	c	a	b
f(j)	0	0	1	0	1	2

0 1 2 3 4 5 6 7 8 9

T= a b a c a a b a c c a b a c a b a a b b

P= a b a c a b

a b a c a b

a b a c a b

a b a c a b

$\boxed{a\ b\ a\ c\ a\ b}$

match found

KMP Algorithm

```
Begin
   n := size of text
   m := size of pattern
   call findPrefix(pattern, m, prefArray)

   while i < n, do
      if text[i] = pattern[j], then
         increase i and j by 1
      if j = m, then
         print the location (i-j) as there is the pattern

      else if i < n AND pattern[j] ≠ text[i] then
         if j ≠ 0 then
            j := prefArray[j ]
         else
            increase i by 1
   done
End
```

Time Complexity of KMP

- The Knuth-Morris-Pratt algorithm performs pattern matching on a text string of length n and a pattern string of length m in O(n+m) time.
- Where n – is for searching and m – is for creating the table

Boyer Moore

Algorithm Boyer-Moore Match (T, P)

Input: Strings T (text) with n characters and P (pattern) with m characters

Output: Starting index of the first substring of T matching P, or an indication that P is not a substring of T

```
Compute Last() for P
i ← m – 1
j ← m – 1
Repeat
        if P [j] = T [i] then
                if j = 0 then
                        return  i          //match found.
                else
                        i ← i- 1
                        j ← j - 1
        else
                i ← i –  last(T [i])
                j ← m – 1
until i > n - l
return "There is no substring of T matching P"
```

Boyer Moore

Ex 1) Consider a text string T= abacaabadcabacabaabb and the pattern string P= abacab. Perform Boyer-Moore algo

Sol:

i	0	1	2	3	4	5
c	a	b	a	c	a	b

$m = 6$

Last(a) = $m-(i+1) = 6-0-1 = 5$

Last(b) = $6-1-1 = 6-2 = 4$

Last(a) = $6-2-1 = 6-3 = 3$

Last(c) = $6-3-1 = 6-4 = 2$

Last(a) = $6-4-1 = 6-5 = 1$

Last(b) = $6-5-1 = 6-6 = 0$ → since its last character use Last(b) = 6

c	a	b	c	*
Last(c)	1	6	2	6

```
a b a c a a b a d c a b a c a b a a b b
a b a c a b
  a b a c a b
    a b a c a b
      a b a c a b
          a b a c a b
              a b a c a b
```

$\boxed{a\ b\ a\ c\ a\ b}$

MATCH FOUND

Time Complexity of Boyer Moore

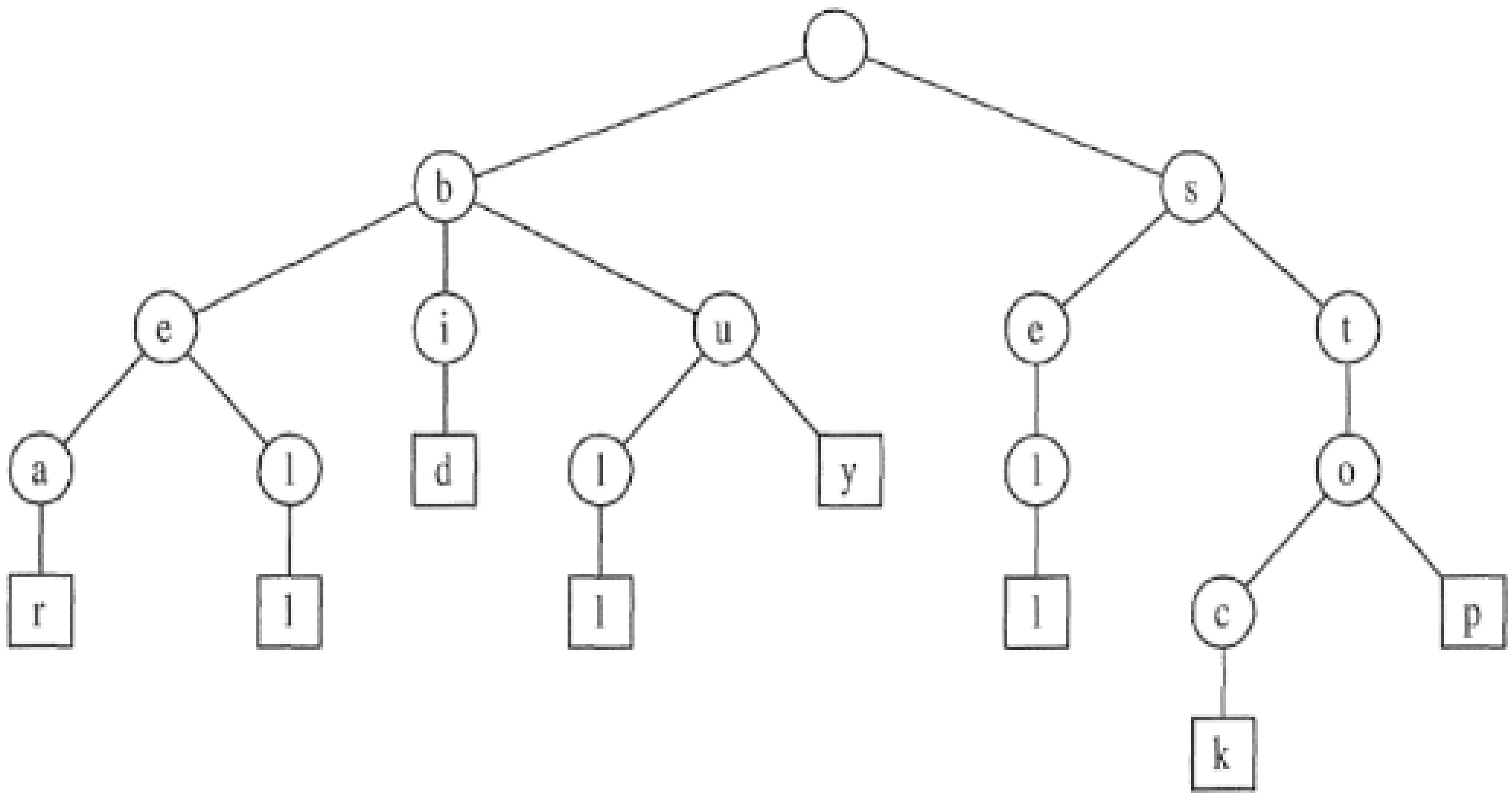
- The worst-case running time of the BM algorithm is O(nm)
- Average case time complexity is O(n)

Tries

- A trie (pronounced "try") is a tree-based data structure for storing strings in order to support fast pattern matching.
- The main application for tries is in information retrieval. Indeed, the name "trie" comes from the word "retrieval.“
- Stores a set of strings. We assign nodes in the tree to letter.

Standard Trie Example

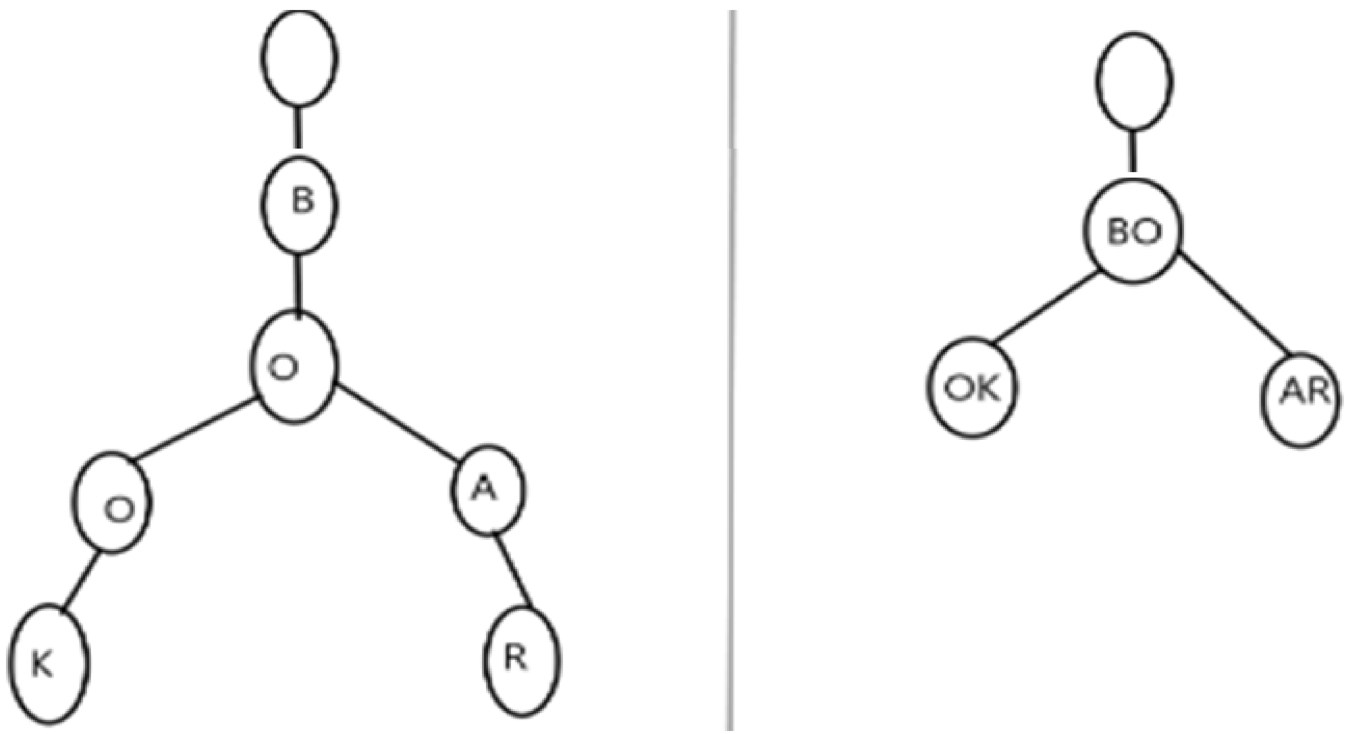
Example: Standard Trie for the strings S={bear, bell, bid, bull, buy, sell, stock, stop}.



NOTE: No words in S should be prefix of the other

Compressed Tries

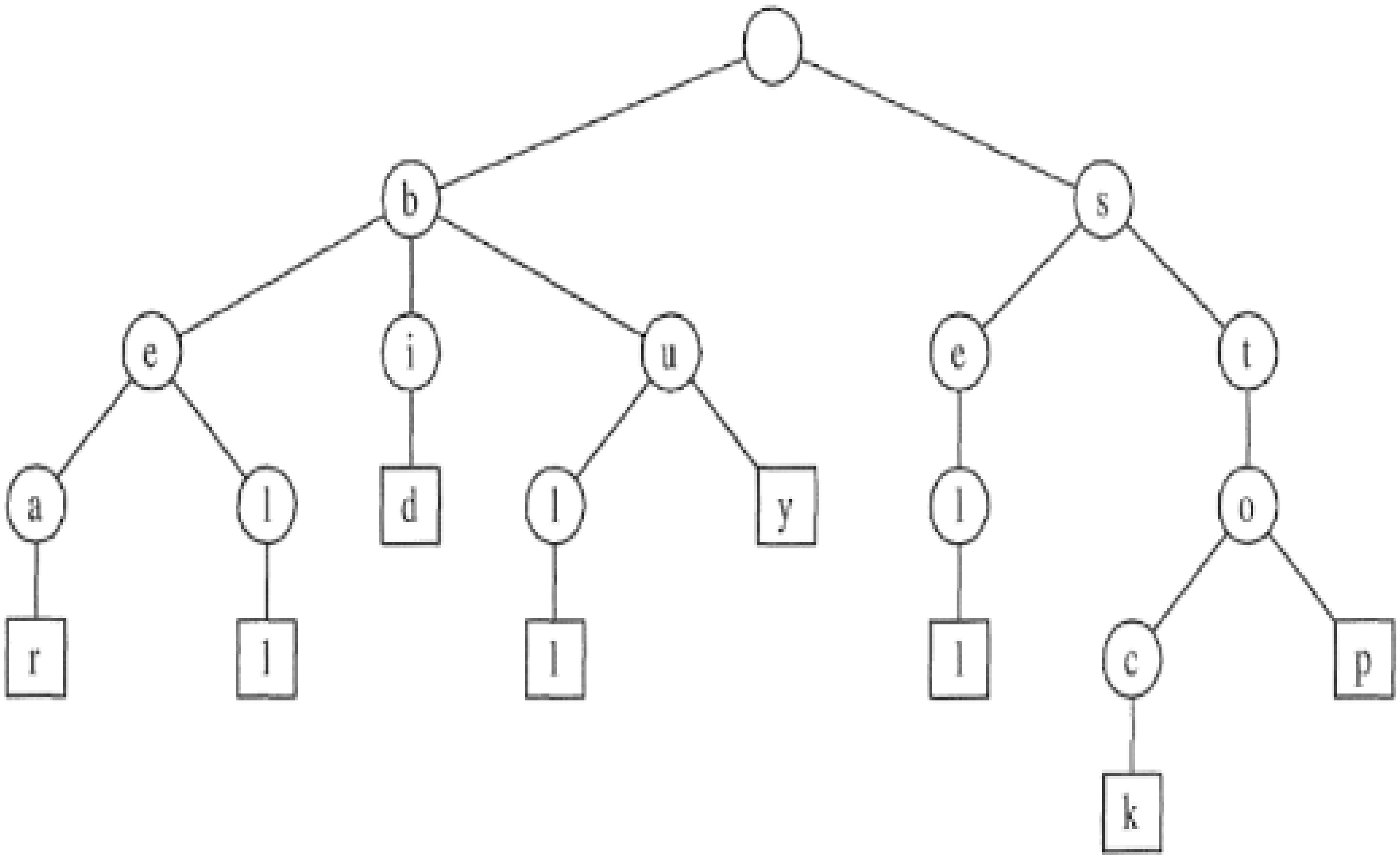
- A compressed trie is similar to a standard trie and is a compact representation of standard trie.



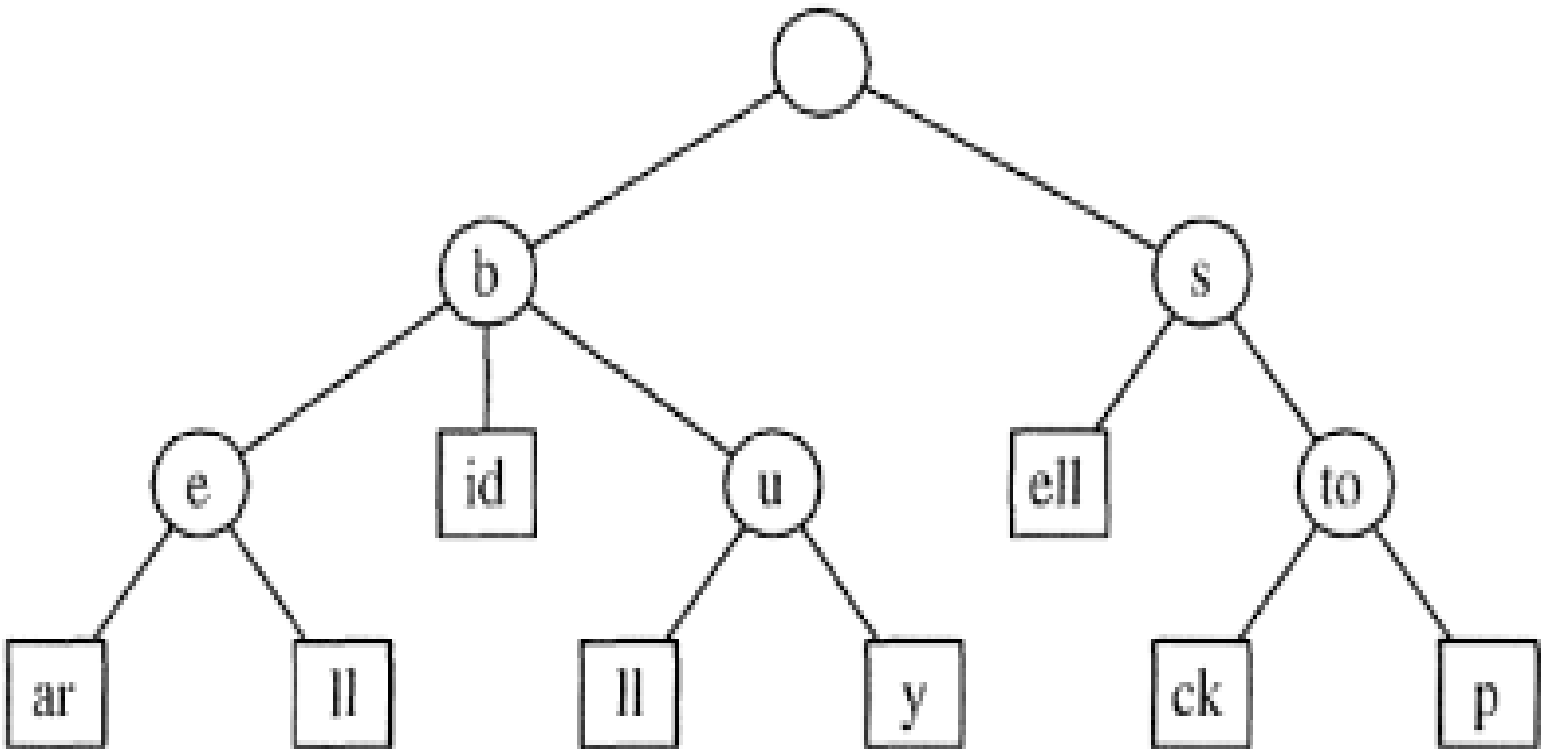
It will contain at least two child nodes.

Compressed Tries Example

Example: Standard Trie for the strings S={bear, bell, bid, bull, buy, sell, stock, stop}.



Example of Compressed Tries



Compact Representation of Compressed Tries

- Let the collection S of strings is an array of strings S[0] , S[1], . . . , S[s - 1] . Instead of storing the label X of a node explicitly, it can be represented implicitly by a triplet of integers (i, j, k) , such that X = S[i] [j..k]; that is, X is the substring of S[i] consisting of the characters from the jth to the kth included.

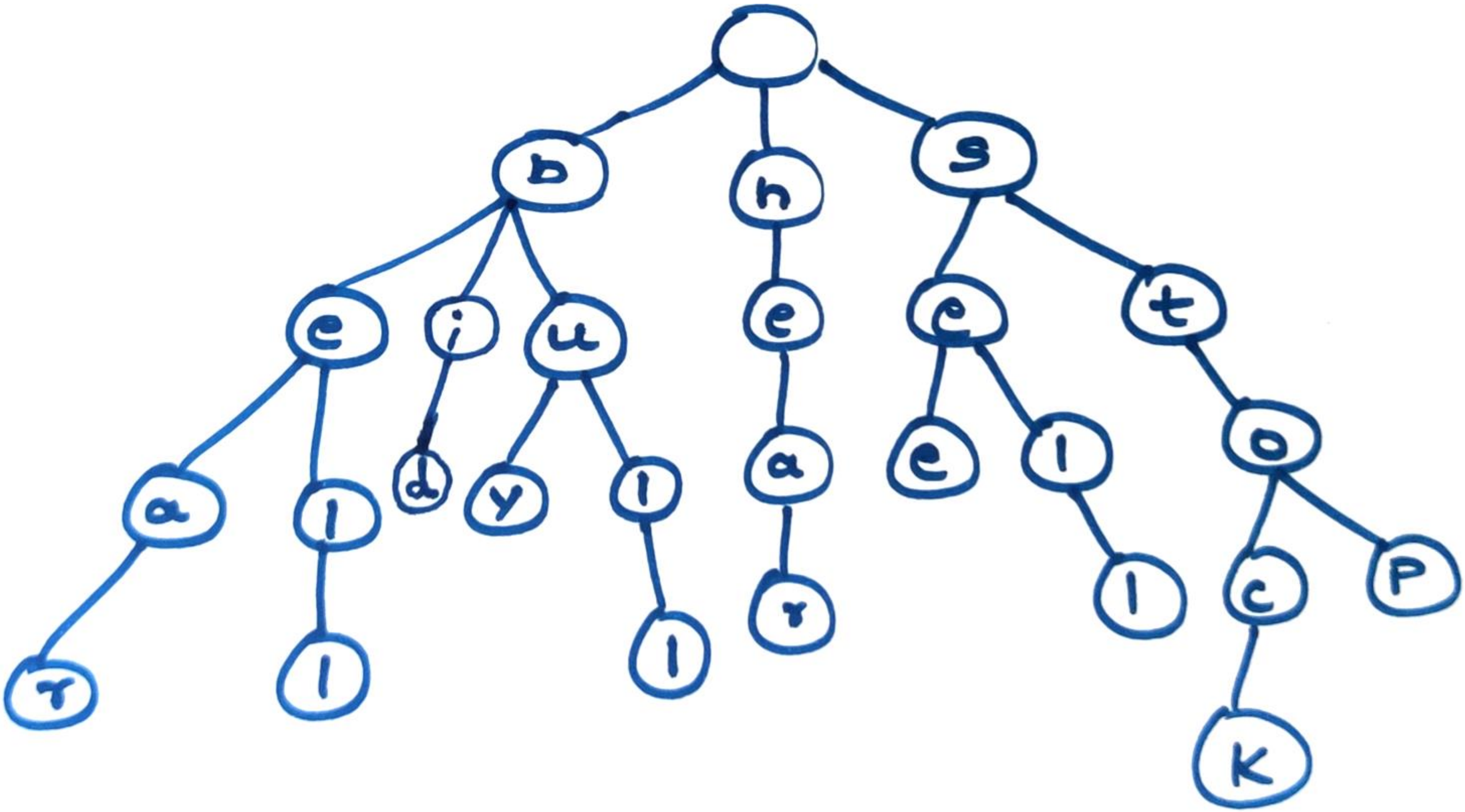
Compact Representation of Compressed Tries

	0	1	2	3	4
$S[0]$ =	s	e	e		
$S[1]$ =	b	e	a	r	
$S[2]$ =	s	e	l	l	
$S[3]$ =	s	t	o	c	k

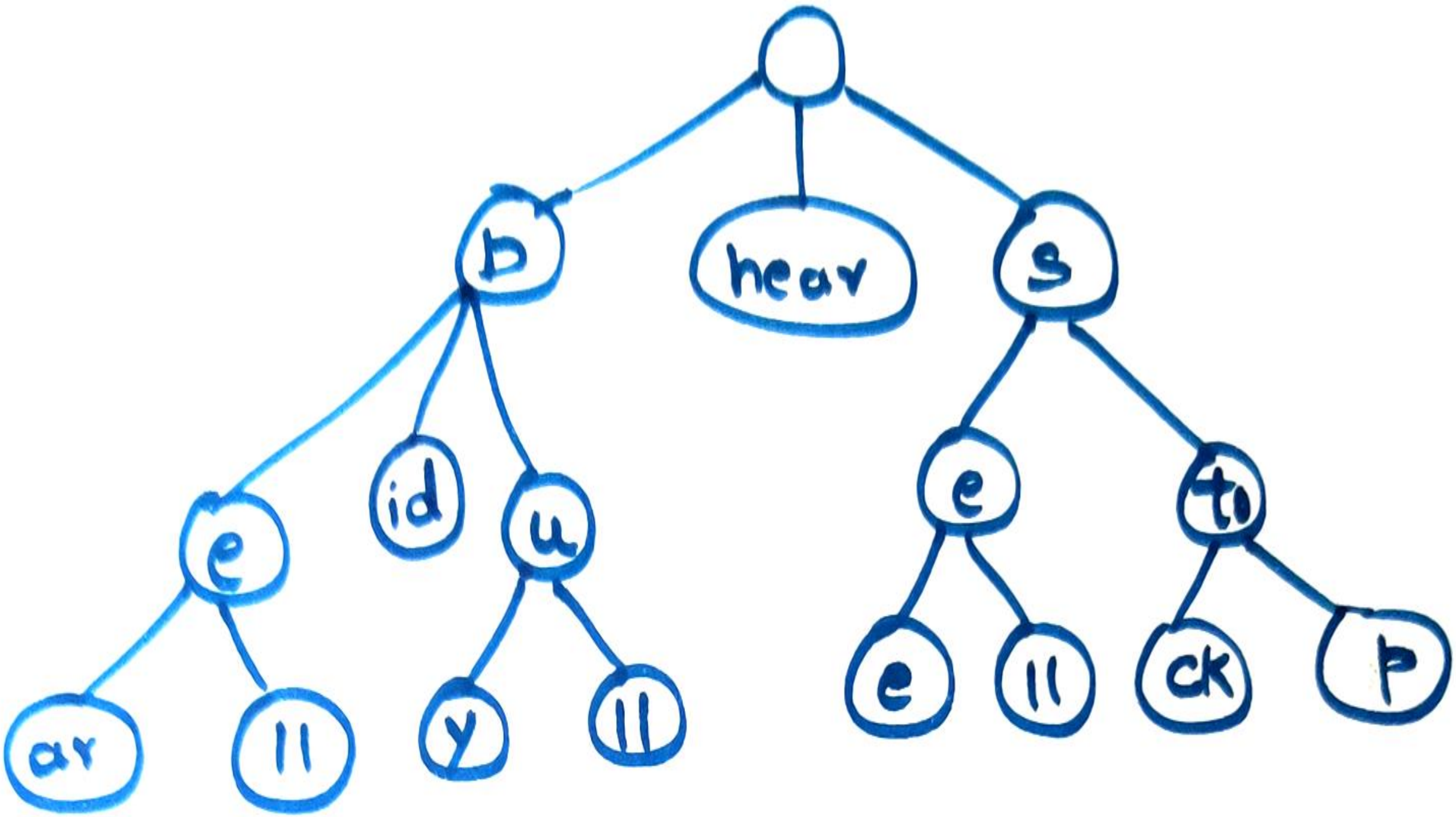
	0	1	2	3
$S[4]$ =	b	u	l	l
$S[5]$ =	b	u	y	
$S[6]$ =	b	i	d	

	0	1	2	3
$S[7]$ =	h	e	a	r
$S[8]$ =	b	e	l	l
$S[9]$ =	s	t	o	p

Compact Representation of Compressed Tries

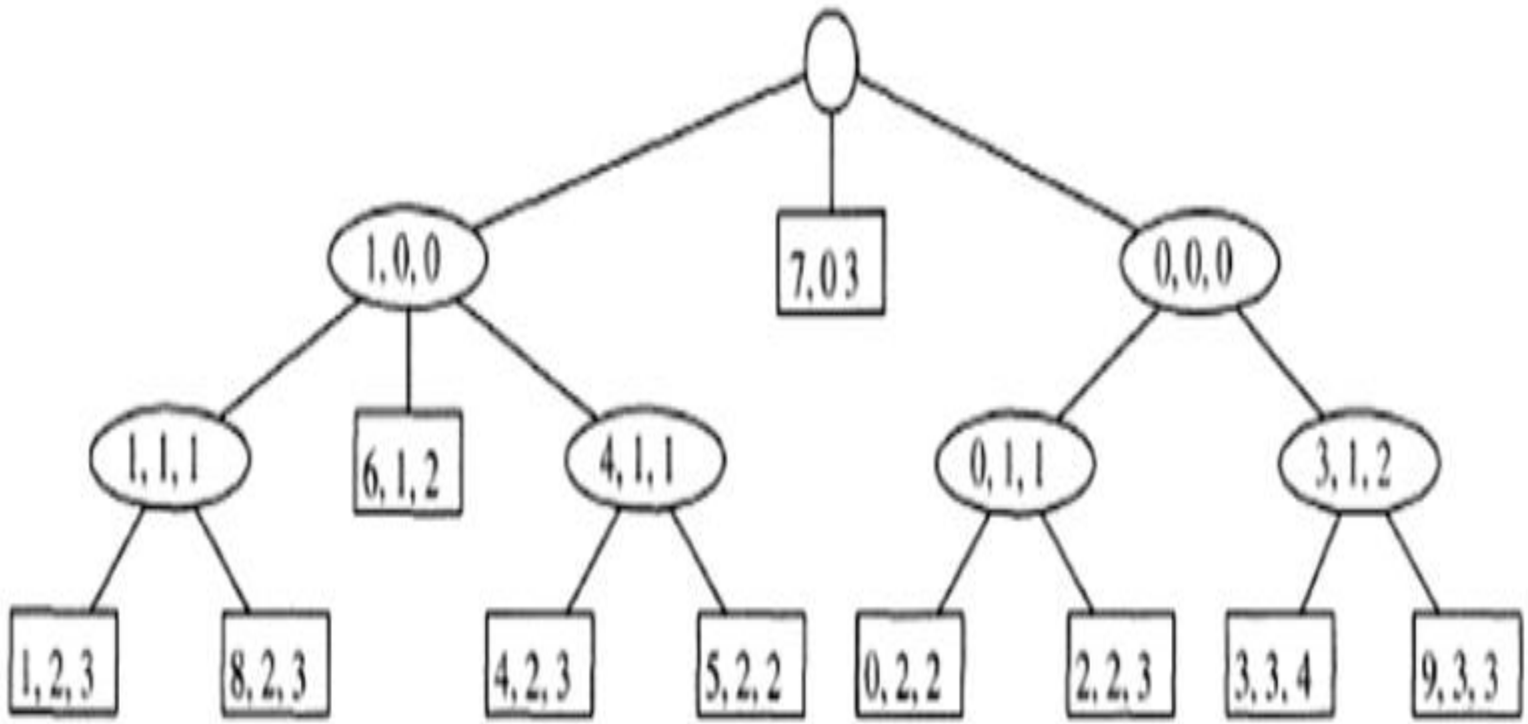


Compact Representation of Compressed Tries



Compact Representation of Compressed Trie

(a)

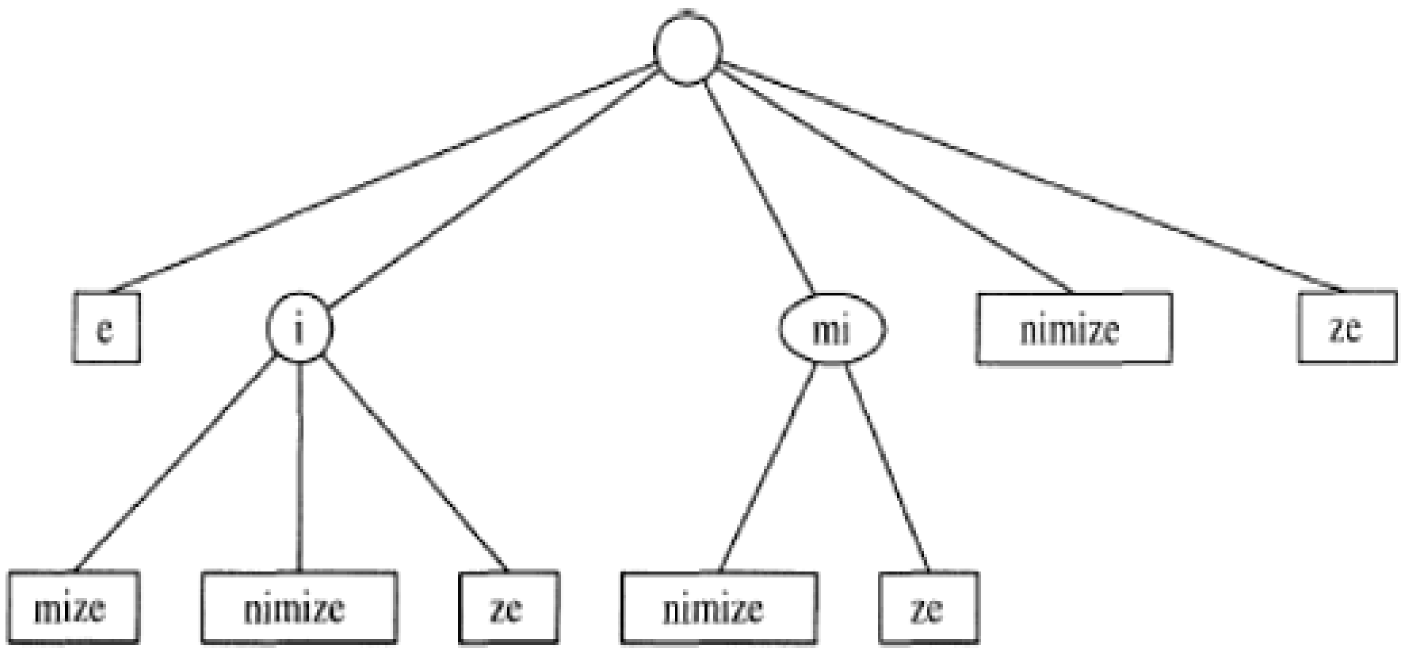


Suffix Tries

- Suffix Tree is compressed trie of all suffixes of a given text/string.
- Application: Pattern Matching, Solving Longest Common Subsequence problem

- Example: Consider a word “minimize”
- Suffix of minimize are: e, ze, ize, mize, imize, nimize, inimize, minimize

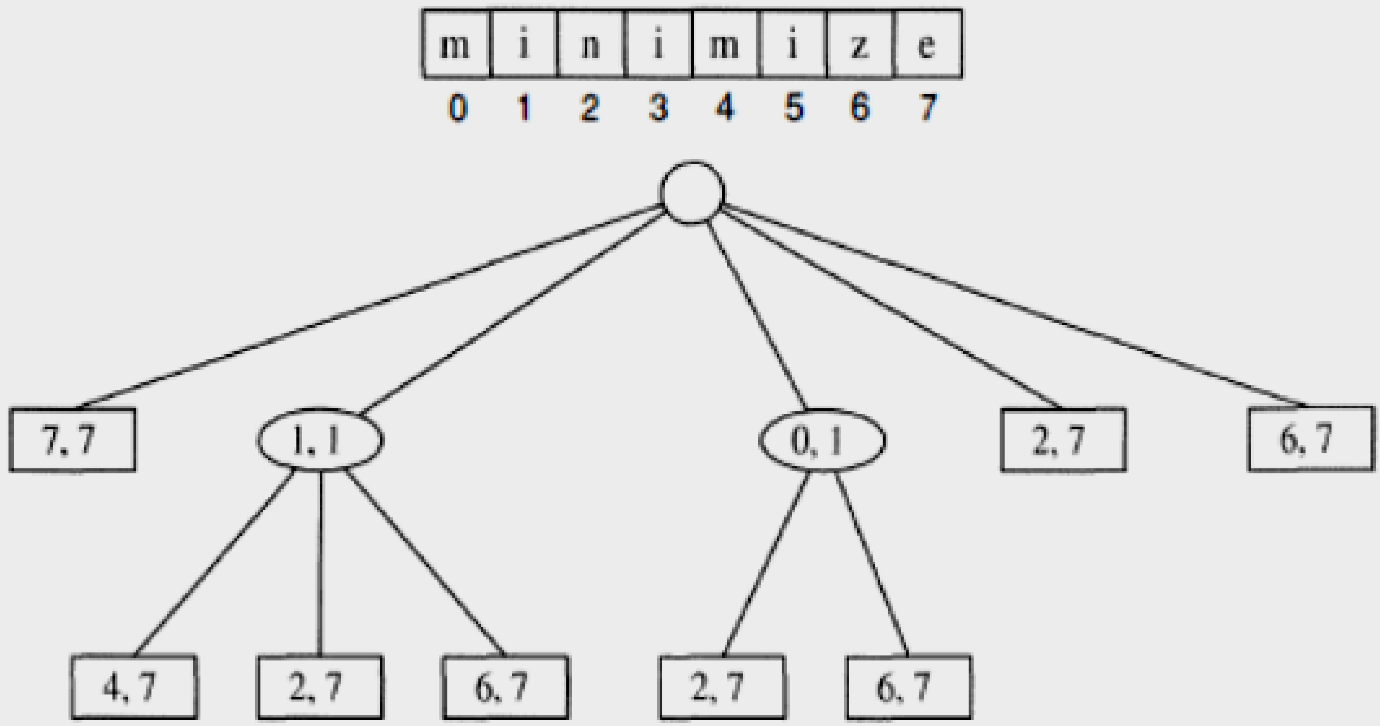
Suffix Trie for string-"minimize"



(a)

m	i	n	i	m	i	z	e
0	1	2	3	4	5	6	7

Suffix Tries for string "minimize"



(b)

Text Compression

- Text compression is also useful for storing collections of large documents more efficiently, so as to allow for a fixed-capacity storage device to contain as many documents as possible.

Building Huffman Tree

- 1.Find the frequency of character
- 2.Arrange text increasing order frequency
- 3.Join lowest cost nodes
- 4.Form the tree

Huffman Algorithm

Algorithm ***Huffman (X)***

Input: String X of length n with d distinct characters

Output: Coding tree for X

Compute the frequency f(c) of each character c of X.

Initialize a priority queue Q.

for each character c in X do

    Create a single-node binary tree T storing c.

    Insert T into Q with key f(c) .

while Q.size() > 1 do

    f1 ← Q. min Key()

    T1 ← Q. removeM in()

    f2 ← Q.min Key()

    T2 ← Q. removeM in()

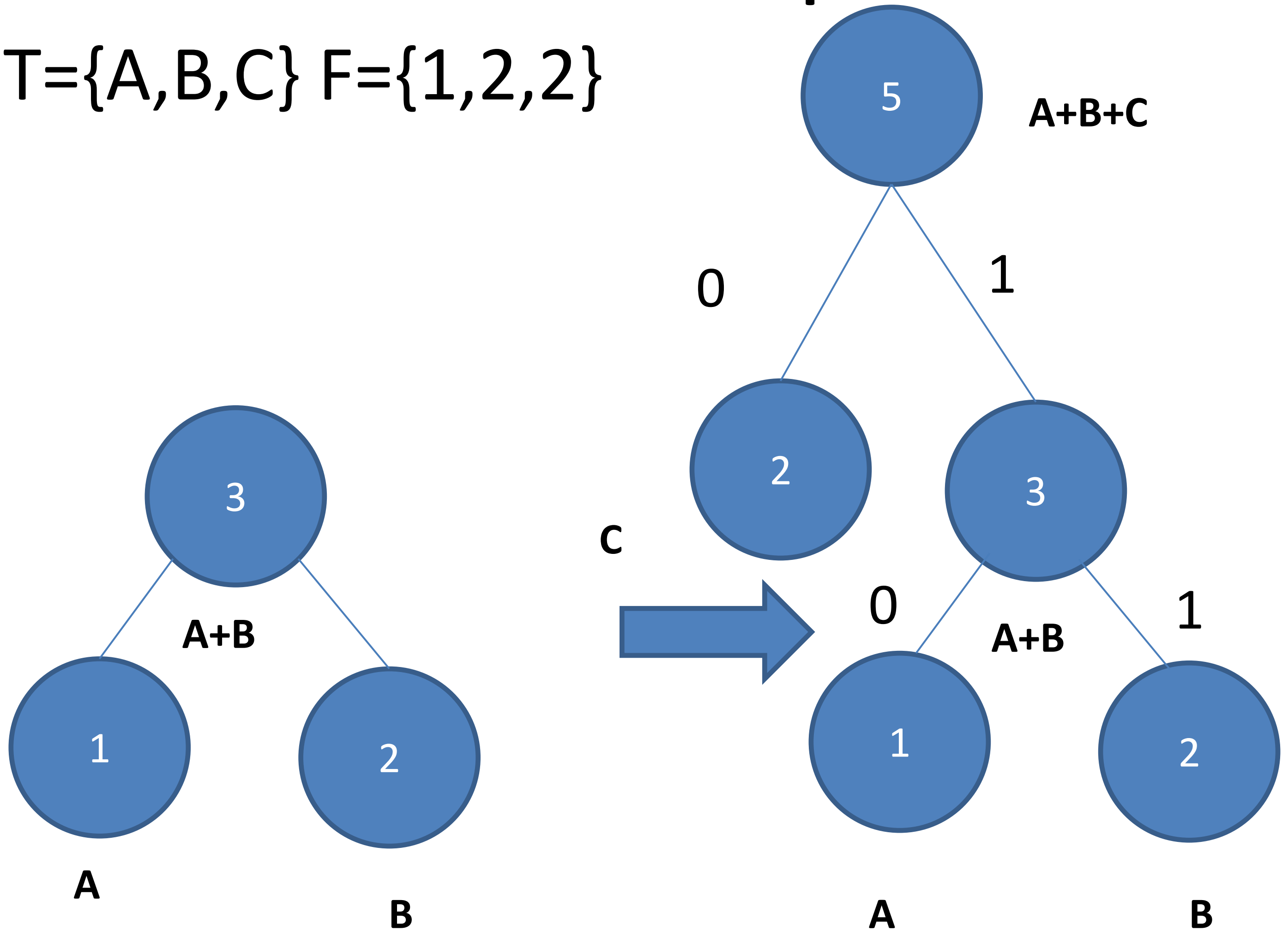
    Create a new binary tree T with left subtree T1 and right subtree T2.

    Insert T into Q with key fi + f2 .

return tree Q.

Example

- T={A,B,C} F={1,2,2}



Therefore the Huffman code of A=10, B=11, C=0

Huffman Coding

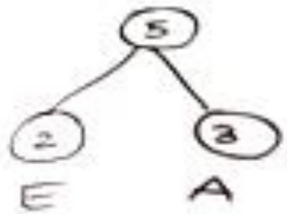
Create a Huffman Table or frequency table

Character	freq
A	3
B	5
C	6
D	4
E	2

place the characters in increasing order of frequency.

E	A	D	B	C
↓	↓	↓	↓	↓
2	3	4	5	6

Huffman Coding



, D → 4 , B → 5 , C → 6

Arrange the list in increasing order

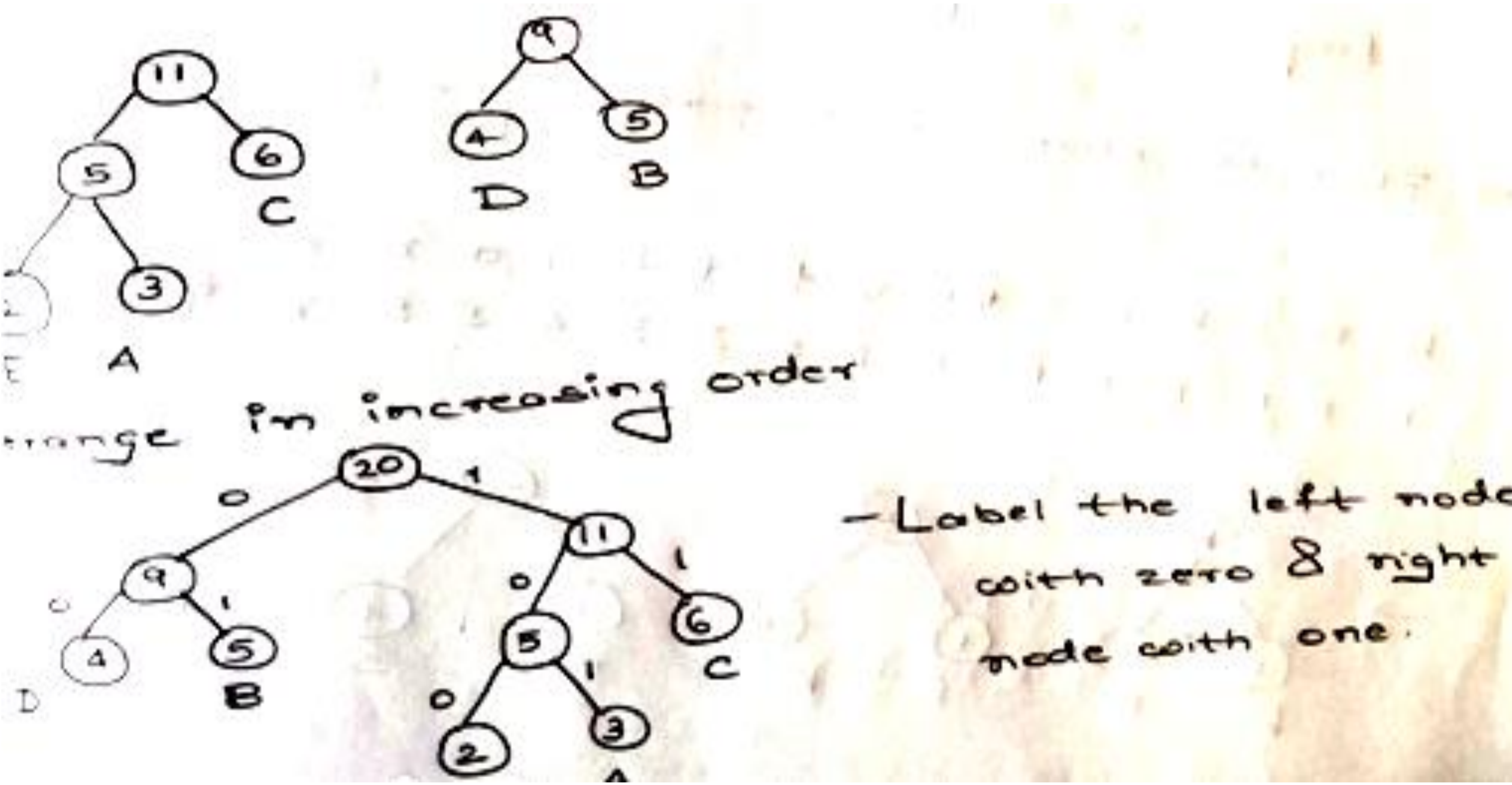
→ D → 4 , B → 5 , E+A → 5 , C → 6

, E+A → 5 , C → 6 ⟹

Arrange the list in ascending order

E+A → 5 , C → 6 , D+B → 9

Huffman Coding



Time Complexity of Huffman

- The time complexity of the Huffman algorithm is O(nlogn)

Text Similarity Testing

- **String Subsequence:**
- Given a string X of size n, a subsequence of X is any string that is of the form, X [i1] X [i2] ... X [ik], ij < i j+ l for j = 1, ..., k that is, it is a sequence of characters that are not necessarily contiguous but are nevertheless taken in order from X.
- For example: Suppose W=abcd
- Subsequence={ab,bd,ac}.

LCS - Longest Common Subsequence Problem

- Given two character strings, X of size n and Y of size m, over some alphabet and to find a longest string S that is a subsequence of both X and Y.
- Ex: W1={abcd} W2={bcd}
- Subsequence of W1={a,ab,ac,ad,abc,bcd,bc,bd,b,c,d,cd,acd,abd,abcd}
- Subsequence of

  W2={b,c,d,bc,cd,bd,bcd}

Then longest common subsequence={bcd}

- The brute-force approach yields an exponential algorithm that runs in $0(2^n\ 2^m)$ time, which is very inefficient.

LCS Algorithm

Algorithm ***LCS (X,Y)***

Input: Strings *X* and Y with n and m elements, respectively

Output: For i = 0, . . . , n - 1, j = 0, . . . ,m - 1, the length L[i, j] of a longest common subsequence of X [0… i] and Y [0 … j]

```
for i ← -1 to n - 1 do
          L [i, -1] ← 0
for j ← 0 to m - 1 do
          L[-1, j] ← 0
for i ← 0 to n - 1  do
     for j ← 0 to m - 1 do
          if X [i] = Y [j] then
               L[i, j] ← L[i - 1, j - 1 ] + 1
          else
               L[i, j] ← max{L[i - 1, j] , L[i, j - 1] }
return array L
```

Longest Common Subsequence

- X=abaaba,   Y=babbab

Y

	*	b	a	b	b	a	b
*	0	0	0	0	0	0	0
a	0	0 ←	1 ↖	1 ←	1 ←	1 ↖	1 ←
b	0	1 ↖	1 ←	2 ↖	2 ↖	2 ←	2 ↖
a	0	1 ↑	2 ↖	2 ←	2 ←	3 ↖	3 ←
a	0	1 ↑	2 ↖	2 ←	2 ←	3 ↖	3 ←
b	0	1 ↖	2 ↑	3 ↖	3 ↖	3 ←	4 ↖
a	0	1 ↑	2 ↖	3 ↑	3 ←	4 ↖	4 ←

X

The longest common subsequence=baba

Time Complexity of LCS

- $T(n)=O(nm)$

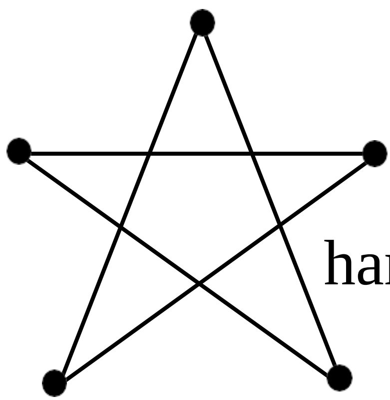
Optimization & Decision Problems

- **Decision problems**
  - Given an input and a question regarding a problem, determine if the answer is **yes** or **no**
- **Optimization problems**
  - Find a solution with the “best” or “optimum” value
- **Optimization problems can be cast as decision problems that are easier to study**
  - e.g.: Shortest path in Graph G
    - Find a path between u and v that uses the fewest edges (optimization)
    - Does a path exist from u to v consisting of at most k edges? (decision)

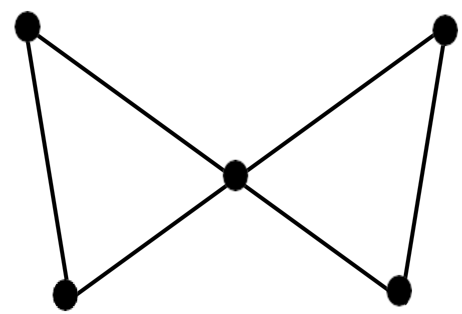
Hamiltonian Cycle

- **Optimization problem:**

  Given a directed graph G = (V, E), determine a cycle that contains each and every vertex in V only once



hamiltonian



Not hamiltonian

- **Decision problem:**

  Given a directed graph G = (V, E), is there a cycle that contains each and every vertex in V only once

Polynomial Algorithm and Exponential Algorithm

Polynomial

- Linear Search-O(n)
- Binary Search-O(logn)
- Bubblesort-O(n^2)
- Mergesort-O(nlogn)
- GreedyKnapsack-O(n)
- HuffmanCod-O(nlogn)

Exponential

- Nqueens-O(n^n)
- SumOfSubset-O(2^n)
- Mcoloring-O(nm^n)
- HamiltonianCycle-O(n^n)
- 0-1 Knapsack-O(2^n)
- SAT

Deterministic and Non Deterministic Algorithm

- Deterministic Algorithm are traceable.
- Non Deterministic Algorithm are non traceable.
- Although Exponential Algorithm cannot be converted into Polynomial Algorithm, they can be converted to non-deterministic Polynomial Algorithm.

Example of Non –Deterministic Algorithm

- Non-Deterministic Search Algorithm with a time complexity of O(1)

```
Algorithm NPSearch(arr,n,k)
{
    i=Choice()  // This step has to be figured out with O(1)
    If(k=arr[i])
    {
        Print("Found")
    }
    Else
    {
        Print("Not Found")
    }
}
```

Class P and NP

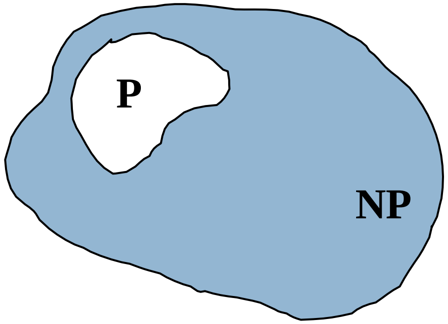
- **<u>Class P</u>** consist of deterministic algorithm that are solvable in polynomial time
- Problems in P are called **tractable.**
- Examples: O(n2), O(n3), O(1), O(n lg n)
- **<u>Class NP</u>**
- Problems **not** in P are **intractable or unsolvable**
- **Example:** $O(2^n)$, $O(n^n)$, $O(n!)$

- Hamiltonian Path: Given a graph G = (V, E), determine a path that contains each and every vertex in V only once
- Traveling Salesman: Find a minimum weight Hamiltonian Path.

Review: **P** and **NP**

- *What do we mean when we say a problem is in* ***P****?*
  - A: A deterministic solution can be found in polynomial time
- *What do we mean when we say a problem is in* ***NP****?*
  - A: A non deterministic solution can be verified in polynomial time
- *What is the relation between* ***P*** *and* ***NP****?*
  - A: **P** $\subseteq$ **NP**, but no one knows whether **P = NP**

Is P = NP?

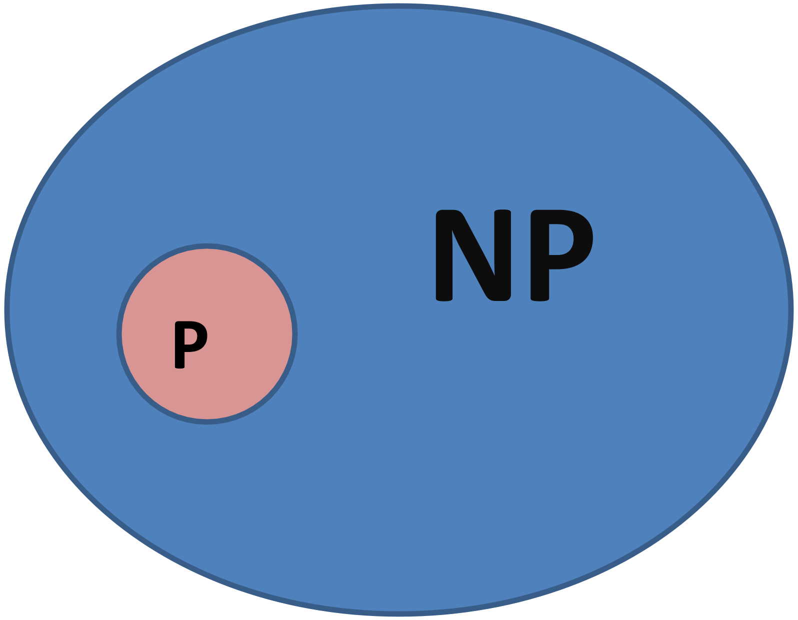


- Any problem in P is also in NP:

$$P \subseteq NP$$

- The big (and open question) is whether P = NP
  - i.e., if it is always easy to check a solution, should it also be easy to find a solution?
- Most computer scientists believe that this is false but we do not have a proof .

Commonly Believed Relationship between P and NP



Satisfiability(SAT)

- I/P : Boolean formula
- O/P : Is formula satisfiable?

SAT $\in$ NP

Proof : Assignment to variables

Verifier: Uses these assignments and checks that the formula evaluates to true (T).

Example: $(x_1 \wedge y_1) \vee (x_2 \wedge y_2) \vee \ldots \vee (x_n \wedge y_n)$

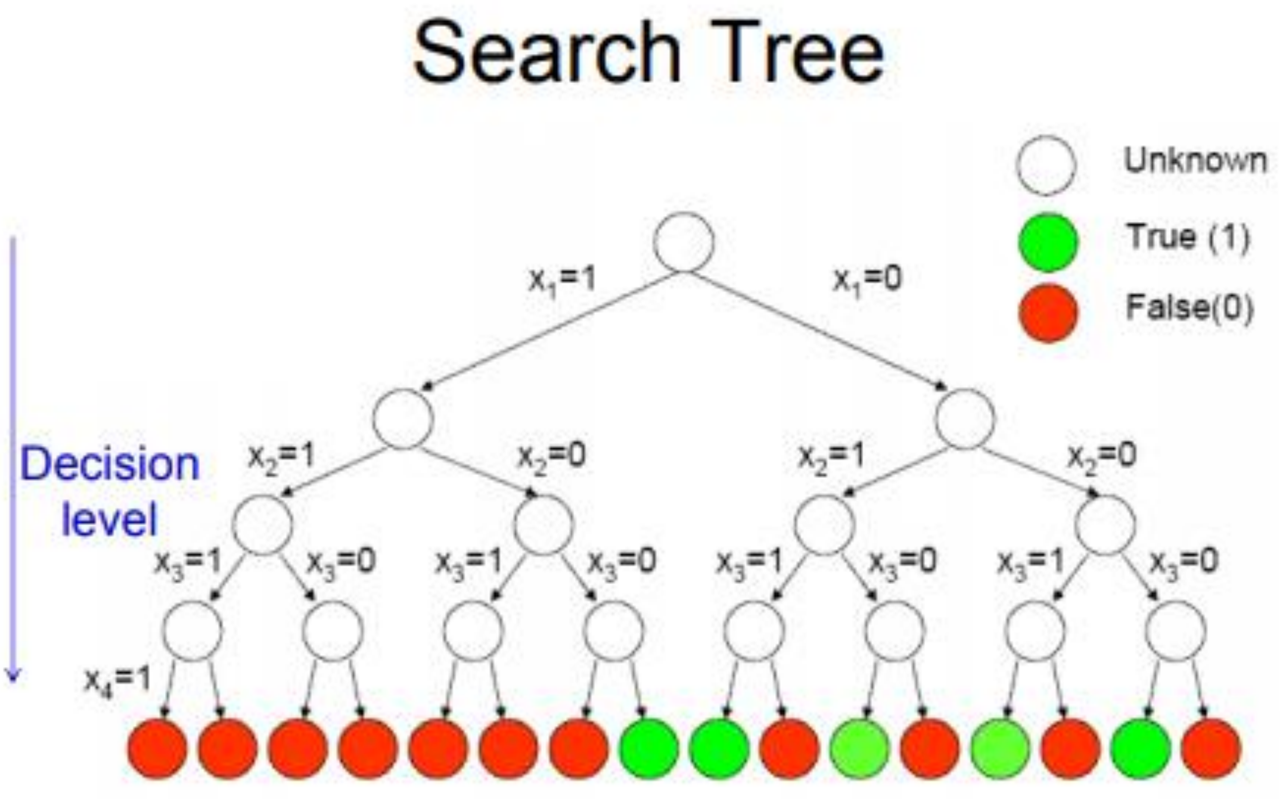
3-SAT Example

a	b	c	$(a \wedge b) \vee c$
1	1	1	1
0	1	1	1
0	0	1	1
0	1	0	0
1	0	1	1
1	0	0	0
1	0	1	1
0	0	0	0

Satisfiability

- SAT is considered as the base problem which has the same exponential time complexity as others.
- If SAT can be solved in polynomial time then all other problems related to it can also be solved in polynomial time.

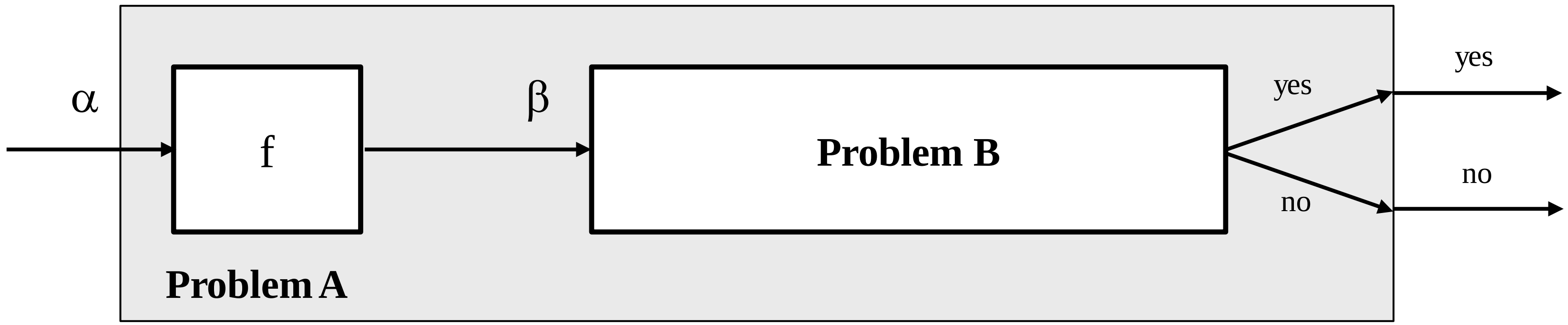
SAT problem can be related to other exponential problem



0-1 Knapsack problem can also be solved using the same decision tree

Reductions

- Reduction is a way in which formula of problem A can be converted to any other problem B
- If we can solve A using the algorithm that can also solve problem B.
- **Idea:** transform the inputs of A to inputs of B



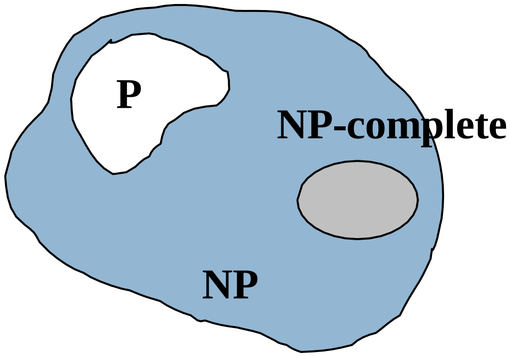
Reduction of Satisfiability

- Assume Satisfiability is NP-Hard.
- If Satisfiability(SAT) can be reduced to some other problem(L), then the problem(L) also becomes NP-Hard.

  SATα L

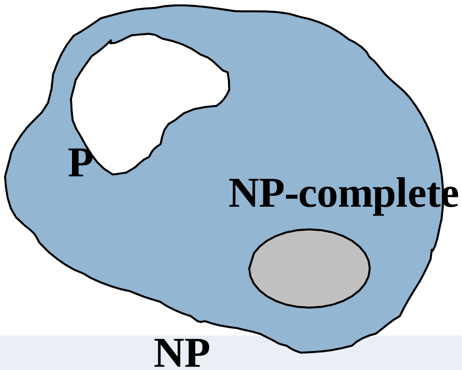
- Conversion has to be done in polynomial time.
- If SAT can be solved in polynomial time then the problem(L) can also be solved in polynomial time and vice-versa.
- Transitive Property=> SATα L1 , L1α L2 , Then SATα L2

NP-Completeness



- SAT has a non deterministic polynomial time algorithm.
- Hence it is NP-Complete.
- Any of the problem has non-deterministic polynomial algorithm, then it is NP-Complete.

NP-Completeness



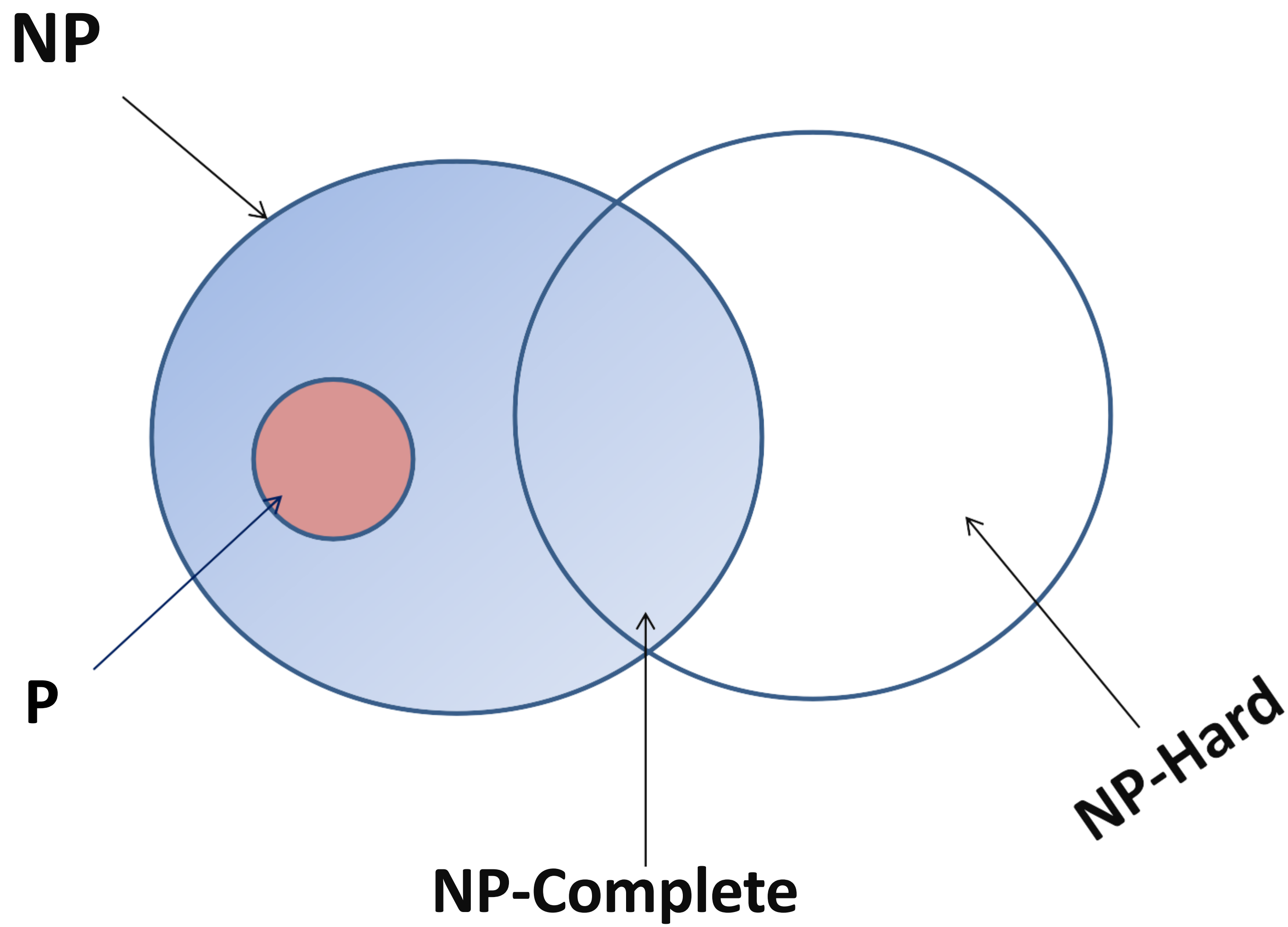
- A problem B is **NP-complete** if:

  (1)B = **NP**

  (2)SAT**α** B

- If B satisfies only property (2) we say that B is **NP-hard**
- No polynomial time algorithm has been found for an **NP-Complete** problem
- No one has ever proven that “no polynomial time algorithm can exist for any **NP-Complete** problem”

Relationship among P,NP,NP-Complete and NP-Hard



Joshi Jesal

COOK's Theorem

- If SAT has an efficient algorithm then so does other problems in NP.

$$NP = P \text{ iff } SAT \in P$$

There is an efficient algo. for SAT

$\downarrow$

There is an efficient algo for all problems in NP.

- If SAT can be proved to be solved in Polynomial time, then all other problems can also be solved in polynomial time

NP Hard and NP-Complete

- A problem A is **NP-hard** if and only if satisfiability reduces to A (satisfiability α A).
- A problem A is **NP-complete** if and only if A is NP-hard and A Є NP.

Randomized Algorithm

- Randomised Algorithm: is an **algorithm** that employs a degree of randomness as part of its logic.

Las Viegas and Monte Carlo

Difference b/w Las-Vegas and Monte Carlo Algo

Las-Vegas	Monte Carlo
1) May not return an answer at all but if they do return a answer, it is gaurant-eed to be correct.	1) May return an answer that is not correct.
2) May take longer time.	2) Has better performance.
3) Running time fluc-tuates	3) Running time is fixed.
Ex: Quicksort	Ex: Karger's algo

Probabilistic Algorithm

- A probabilistic algorithm is an algorithm where the result and/or the way the result is obtained depend on chance.
- These algorithms are also sometimes called randomized.
- The techniques of applying probabilistic algorithms to numerical problems were originally called Monte Carlo methods.

Example of Monte Carlo Algorithm

Algorithm Rand_Primality_Test(n)
{
> **Randomly choose a∈[2,n-1]**
> **Calculate a^{n-1} mode n**
> **If a^{n-1} mode n =1 then**
>> **Print (n is PROBABLY prime)**
>
> **Else**
>> **Print (n is definitely not prime)**

}

NOTE: This is Fermats Primality Test

Approximation Algorithm

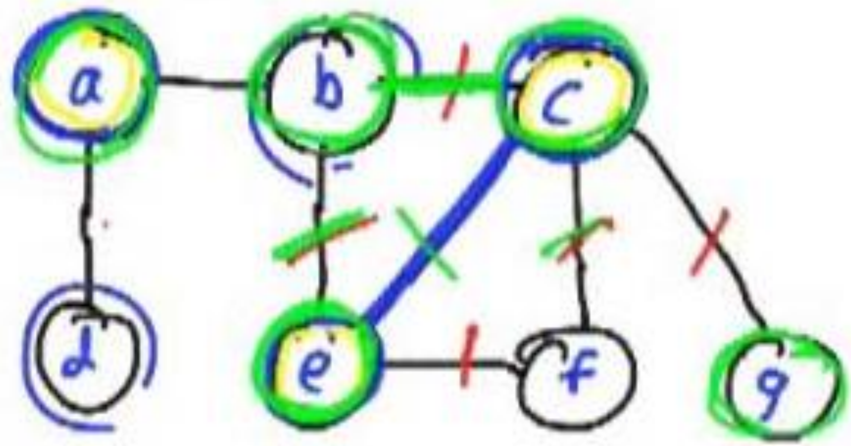
- For a lot of practical applications **near-optimal solutions** are perfectly acceptable.

- Algorithms that return near-optimal solutions for a problem are called **approximation algorithms**.

- **Guaranteed to run in polynomial time**

Approximation Algorithm

Find Vertex Cover $V' \subseteq V$ of $G = (V, E)$ that is of minimum size.



$C = \{a, b, e, f, c, g\}$

```
APPROX-VERTEX-COVER(G)
C = {}
E' = G.E
while E' != {}
     let (u,v) be an arbitrary edge of E'
     C = Union(C, {u,v})
    remove from E' every edge incident on either u or v
return C
```

Text Books

- 1 Fundamentals of Computer Algorithms – E. Horowitz et al, 2nd Edition UP.- - NP Hard,NP Complete, Randomised Algorithm
- 2. Introduction to Algorithms, 3th Edition, Thomas H Cormen, Charles E Lieserson, Ronald L Rivest and Clifford Stein, MIT Press/McGraw-Hill.
- 3. Algorithm Design, 1ST Edition, Jon Kleinberg and ÉvaTardos, Pearson.

Questions asked so far

Compute a table representing facture function for Knuth Morris Pratt's (KMP) algorithm for the pattern string. a b a a b a.

Explain the difference between standard tree and compressed tree with the help of an example.

Using Boyer Moore algorithm check whether the pattern P = abacab lies in the text given by T = abacaabadcabacabaabb. or not.

Implement Boyer Moore algorithm on the given text and pattern.
Text: a pattern matching algorithm
Pattern: rithm

Explain Trie., standard Trie and its compact Representation. With the help of an example.

Write Huffman coding algorithm and draw Huffman Tree for the string X.
X= " a fast runner need never be afraid of the dark"
Also get code's for each alphabet.

Write an algorithm for finding longest common subexpression in a string (LCS).

Questions asked so far

Write Boyer Moore pattern matching algorithm.
Illustrate standard Trie for the following set of strings.
{bear, bell, bid, bull , buy, sell , stock, stop}

Compute a table representing the KMP failure function for the pattern string "cgtacgttcgtac"

Implement Brute Force algorithm to check whether the pattern P = engineer lies in the text T = "Computer Engineering" or not.

Draw the suffix trie and the compact representation of the suffix trie for the string "minimize".

Draw the frequency table and Huffman tree for the following string X.

X = "a fast runner need never be afraid of the dark". Also obtain the code for each character in X.

Questions asked so far

Draw suffix tree and its compact representation for the string "minimize".

Explain and demonstrate how Brute Force algorithm will be used to search the pattern P in text T where
T = "Twinkle Twinkle Little Star"
P = "Little".

Draw the Huffman tree and write the Huffman code for all the symbols based on the data provided in the following table :

Symbol	Frequency
A	24
B	12
C	10
D	8
E	8

Draw standard tree for the following set of strings.

S = {bear, bell, bid, bull, buy, sell, stock, stop}.