CLOSURE PROPERTIES OF CFL

Context Free Languages (CFLs) are accepted by pushdown automata. Context free languages can be generated by context free grammars, which have productions (substitution rules) of the form : A -> ρ (where A ∈ N and ρ ∈ (T ∪ N)* and N is a non-terminal and T is a terminal)

Properties of Context Free Languages

Union : If L1 and L2 are two context free languages, their union L1 ∪ L2 will also be context free.

For example,

L1 = { $a^n b^n c^m$ | m >= 0 and n >= 0 } and L2 = { $a^n b^m c^m$ | n >= 0 and m >= 0 }

L3 = L1 ∪ L2 = { $a^n b^n c^m$ ∪ $a^n b^m c^m$ | n >= 0, m >= 0 } is also context free.

L1 says number of a's should be equal to number of b's and L2 says number of b's should be equal to number of c's. Their union says either of two conditions to be true. So it is also context free language.

So CFL are closed under Union.

Concatenation : If L1 and If L2 are two context free languages, their concatenation L1.L2 will also be context free. For example,

L1 = { $a^n b^n$ | n >= 0 } and L2 = { $c^m d^m$ | m >= 0 }

L3 = L1.L2 = { $a^n b^n c^m d^m$ | m >= 0 and n >= 0} is also context free.

L1 says number of a's should be equal to number of b's and L2 says number of c's should be equal to number of d's. Their concatenation says first number of a's should be equal to number of b's, then number of c's should be equal to number of d's. So, we can create a PDA which will first push for a's, pop for b's, push for c's then pop for d's. So it can be accepted by pushdown automata, hence context free.

So CFL are closed under Concatenation.

Kleene Closure : If L1 is context free, its Kleene closure L1* will also be context free. For example,

L1 = { $a^n b^n$ | n >= 0 }

L1* = { $a^n b^n$ | n >= 0 }* is also context free.

So CFL are closed under Kleen Closure.

Intersection and complementation : If L1 and If L2 are two context free languages, their intersection L1 ∩ L2 need not be context free. For example,

L1 = { $a^n b^n c^m$ | n >= 0 and m >= 0 } and L2 = ($a^m b^n c^n$ | n >= 0 and m >= 0 }

L3 = L1 ∩ L2 = { $a^n b^n c^n$ | n >= 0 } need not be context free.

L1 says number of a's should be equal to number of b's and L2 says number of b's should be equal to number of c's. Their intersection says both conditions need to be true, but push down automata can compare only two. So it cannot be accepted by pushdown automata, hence not context free.

Similarly, complementation of context free language L1 which is ∑* – L1, need not be context

EXAMPLES OF TURING MACHINE

Example 1:

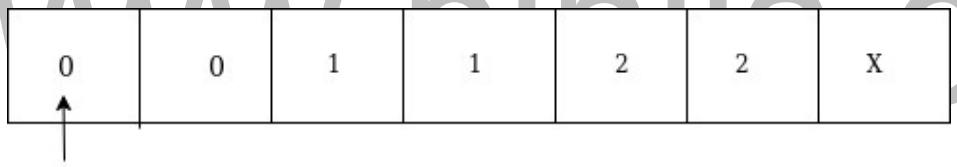
Construct a TM for the language L = $\{0^n1^n2^n\}$ where n≥1

Solution:

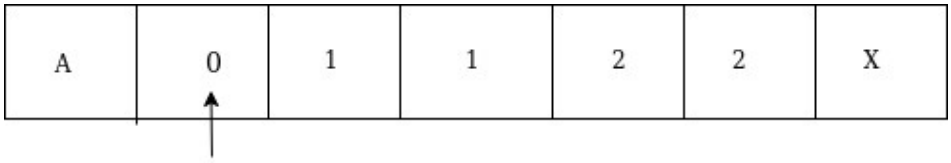
L = $\{0^n1^n2^n \mid n\geq 1\}$ represents langua ge where we use only 3 character, i.e., 0, 1 an d 2. In this, some number of 0's followed by an equal number of 1's and then followed by an equal number of 2's. Any type of string which falls in this categc

The simulation for 001122 can be shov

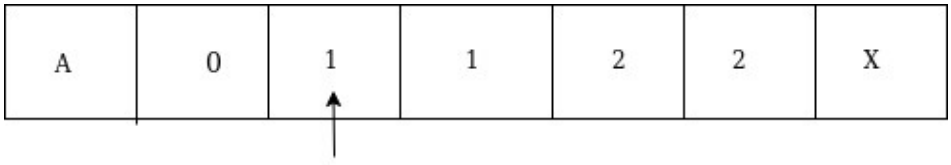
Now, we will see how this Turing machine will work for 001122. Initially, state is 0 and head points to 0 as:



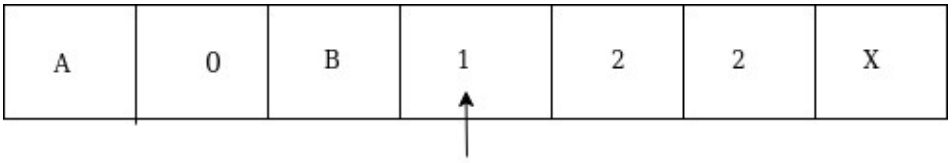
The move will be δ(q0, 0) = δ(q1, A, R) which means it will go to state q1, replaced 0 by A and head will move to the right as:



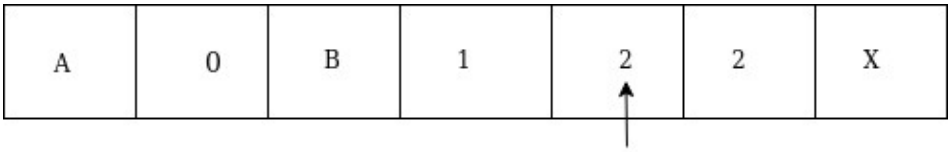
The move will be δ(q1, 0) = δ(q1, 0, R) which means it will not change any symbol, remain in the same state and move to the right as:



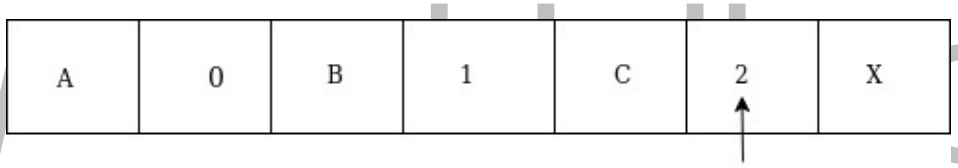
The move will be δ(q1, 1) = δ(q2, B, R) which means it will go to state q2, replaced 1 by B and head will move to right as:



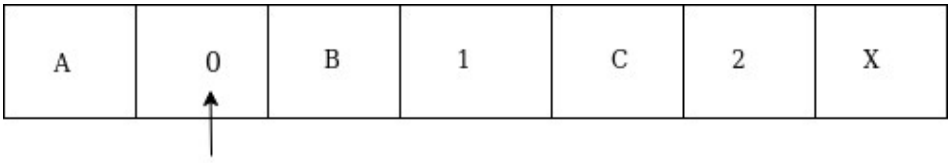
The move will be δ(q2, 1) = δ(q2, 1, R) which means it will not change any symbol, remain in the same state and move to right as:



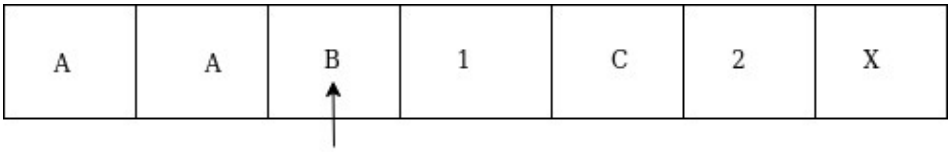
The move will be δ(q2, 2) = δ(q3, C, R) which means it will go to state q3, replaced 2 by C and head will move to right as:



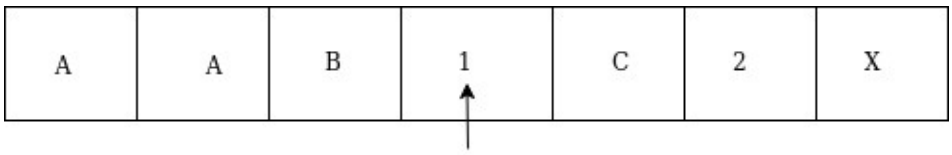
Now move δ(q3, 2) = δ(q3, 2, L) and δ(q3, C) = δ(q3, C, L) and δ(q3, 1) = δ(q3, 1, L) and δ(q3, B) = δ(q3, B, L) and δ(q3, 0) = δ(q3, 0, L), and then move δ(q3, A) = δ(q0, A, R), it means will go to state q0, replaced A by A and head will m



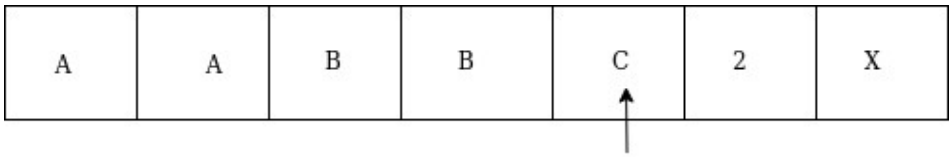
The move will be δ(q0, 0) = δ(q1, A, R) which means it will go to state q1, replaced 0 by A, and head will move to right as:



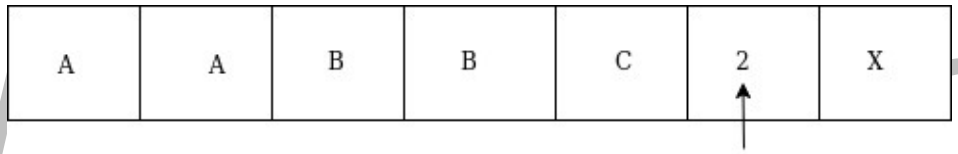
The move will be δ(q1, B) = δ(q1, B, R) which means it will not change any symbol, remain in the same state and move to right as:



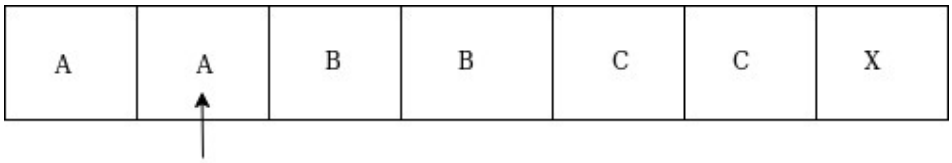
The move will be δ(q1, 1) = δ(q2, B, R) which means it will go to state q2, replaced 1 by B and head will move to right as:



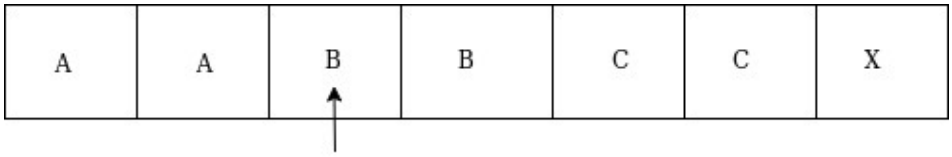
The move will be δ(q2, C) = δ(q2, C, R) which means it will not change any symbol, remain in the same state and move to right as:



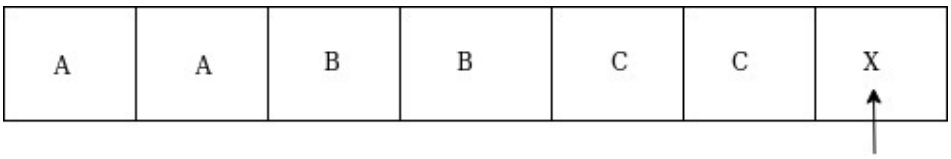
The move will be δ(q2, 2) = δ(q3, C, L) which means it will go to state q3, replaced 2 by C and head will move to left until we reached A as:



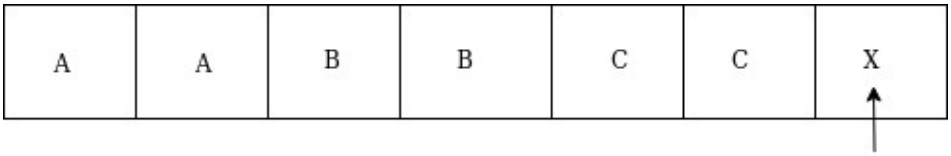
immediately before B is A that means all the 0's are market by A. So we will move right to ensure that no 1 or 2 is present. The move will be δ(q2, B) = (q4, B, R) which means it will go to state q4, will not change any symbol, and move to right as:



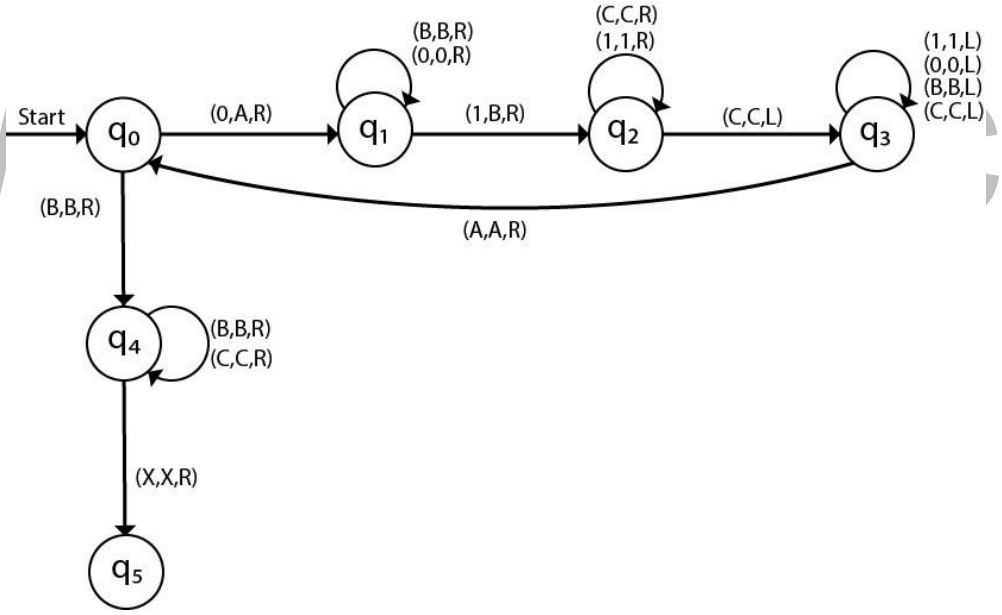
The move will be (q4, B) = δ(q4, B, R) and (q4, C) = δ(q4, C, R) which means it will not change any symbol, remain in the same state and move to right as:



The move δ(q4, X) = (q5, X, R) which means it will go to state q5 which is the HALT state and HALT state is always an accept state for any TM.



The same TM can be represented by Transition Diagram:



Example 2:

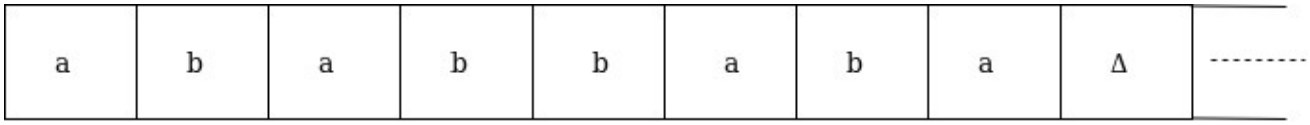
Construct a TM machine for checking the palindrome of the string of even length.

Solution:

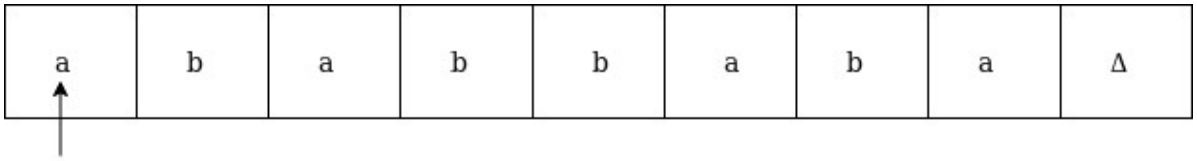
Firstly we read the first symbol from the left and then we compare it with the first symbol from right to check whether it is the same.

Again we compare the second sym ol from left with the second symbol from right. We repeat this process for all the symbols. If we found any symbol not matching, we cannot lead the machine to HALT state.

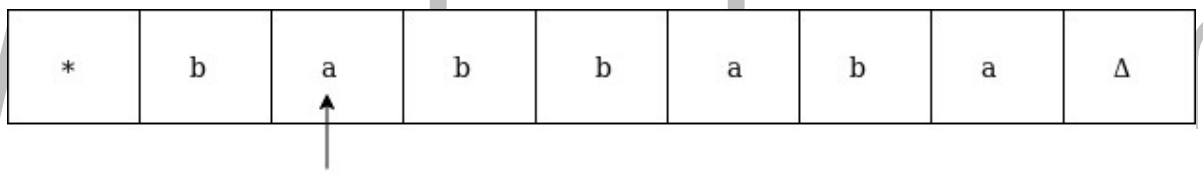
Suppose the string is ababbabaΔ. The simulation for ababbabaΔ can be shown as follows:



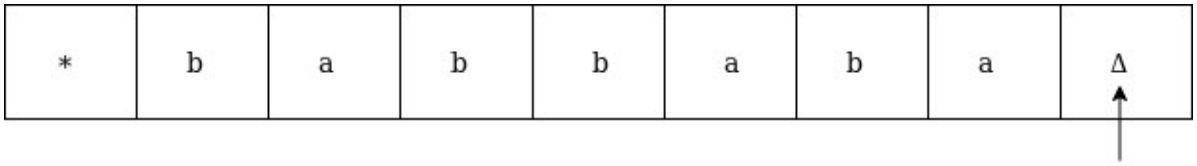
Now, we will see how this Turing machine will work for ababbabaΔ. Initially, state is q0 and head points to a as:



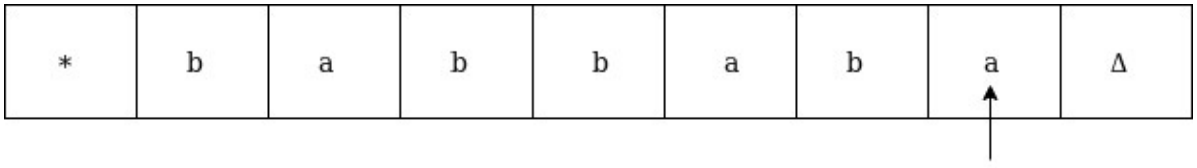
We will mark it by * and move to right end in search of a as:



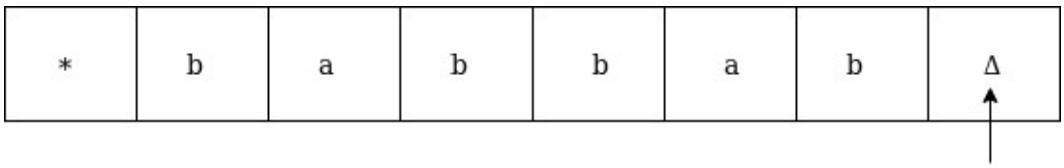
We will move right up to Δ as:



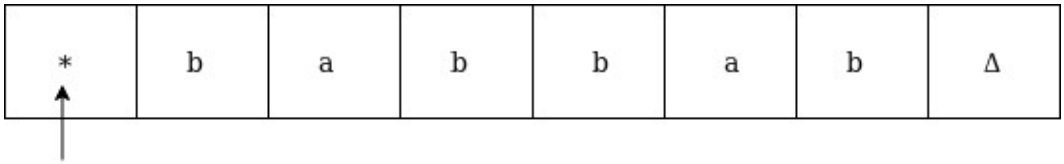
We will move left and check if it is a:



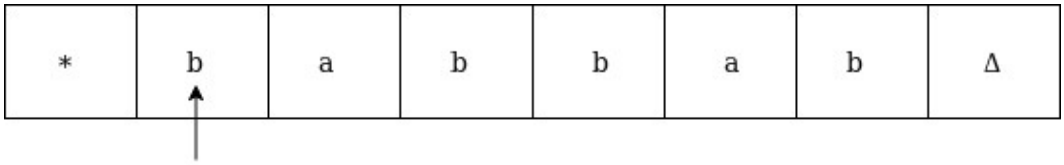
It is 'a' so replace it by Δ and move left as:



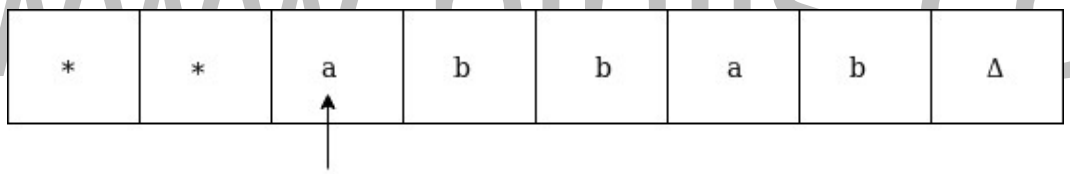
Now move to left up to * as:



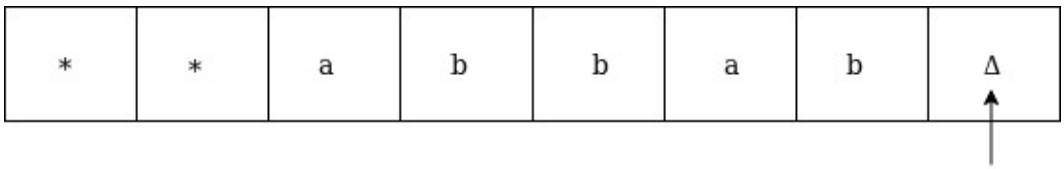
Move right and read it



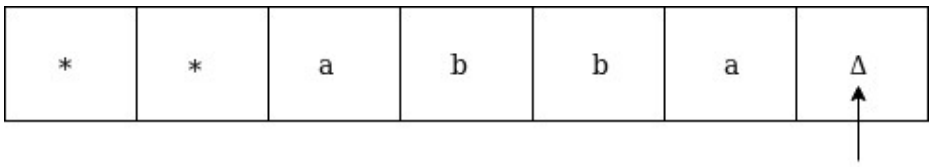
Now convert b by * and move right as:



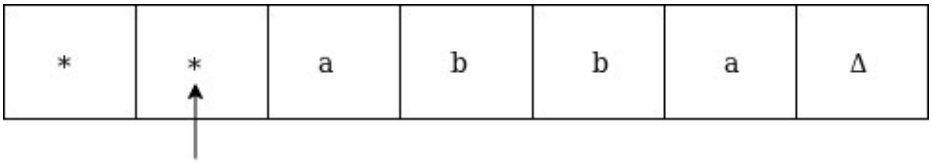
Move right up to Δ in search of b as:

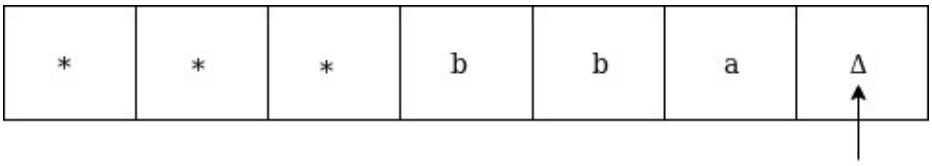


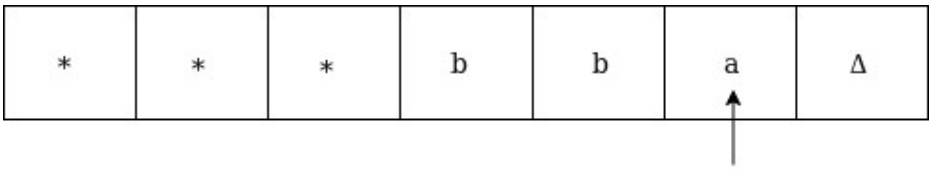
Move left, if the symbol is b then convert it into Δ as:

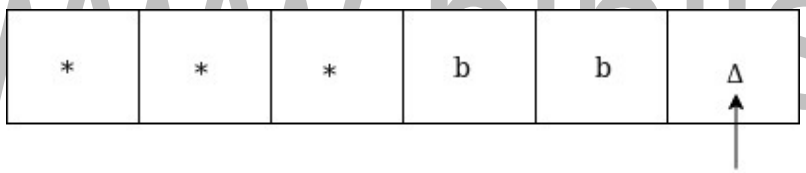


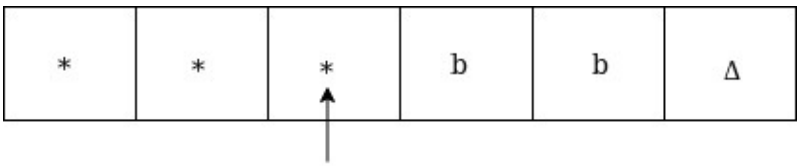
Now move left until * as:

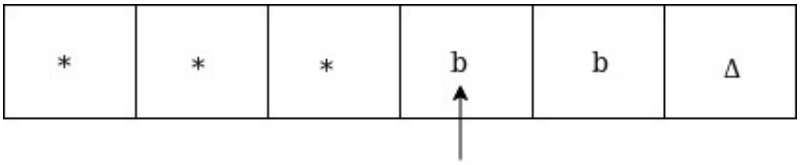


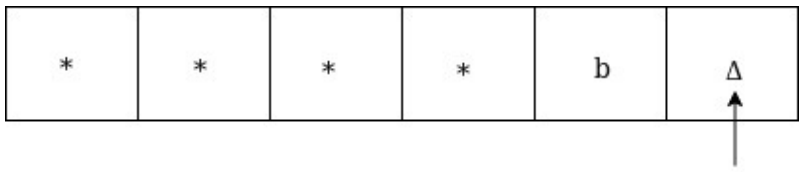




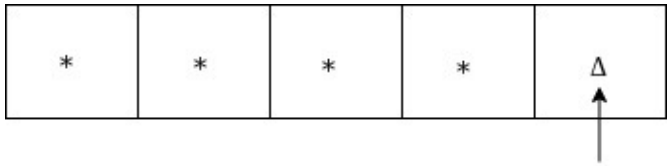




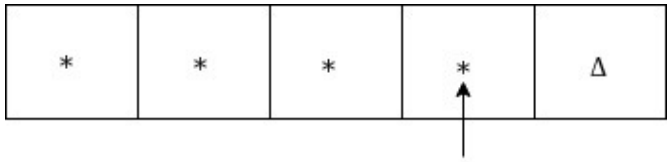




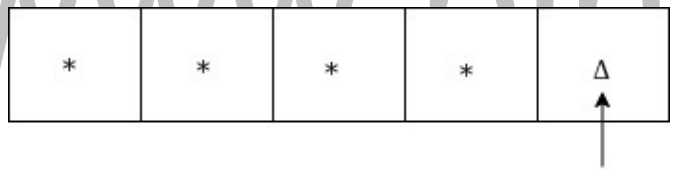
Move left, if the left symbol is b, replace it by Δ as:



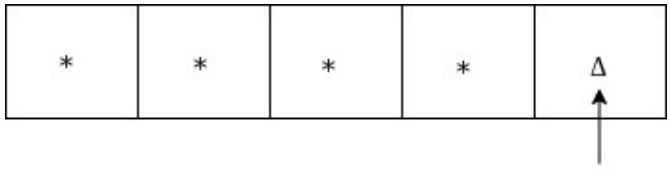
Move left till *



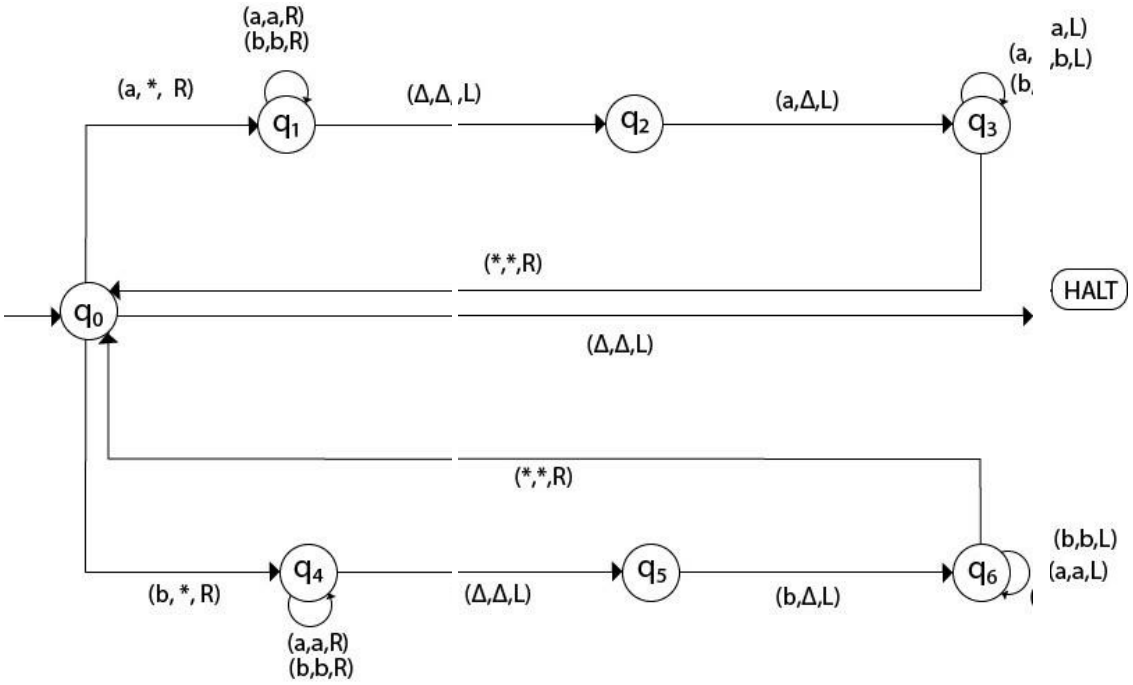
Move right and check whether it is Δ



Go to HALT state



The same TM can be represented by Transition Diagram:



Example 3:

Construct a TM machine for checking the palindrome of the string of odd length.

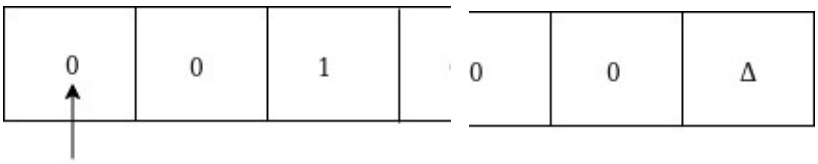
Solution:

Firstly we read the first symbol from left and then we compare it with the first symbol from right to check whether it is the same.

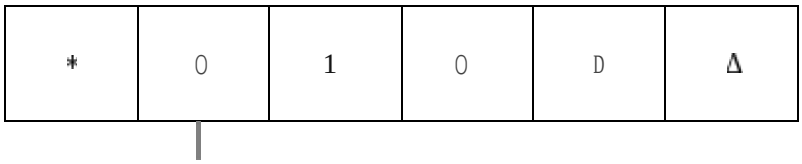
Again we compare the second symbol from left with the second symbol from right. We repeat this process for all the symbols. If we found any symbol not matching, we lead the machine to HALT state.

Suppose the string is 00100Δ. The simulation for 00100Δ can be shown as follows:

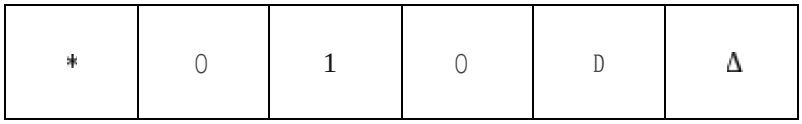
Now, we will see how this Turing machine will work for 00100Δ. Initially, state is q0 and head points to 0 as:



Now replace 0 by * and move right as:

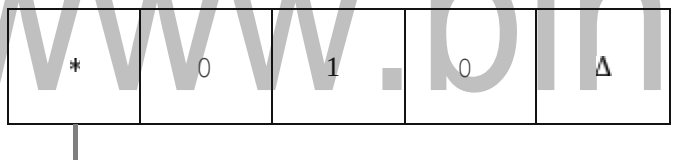


Move right up to Δ as:

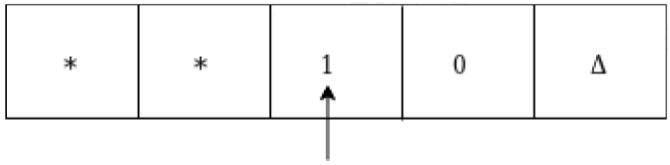


Move left and replace 0 by Δ a**nd** move left:

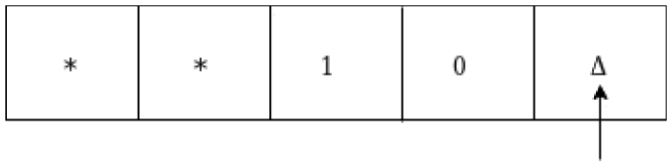
Now move left up to * as:



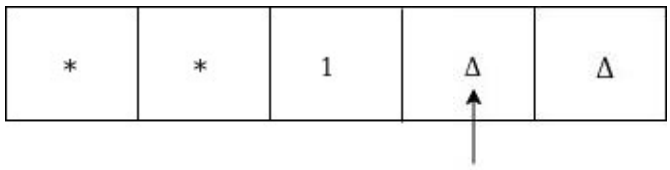
Move right, conve**rt** 0 by * and then move right as:



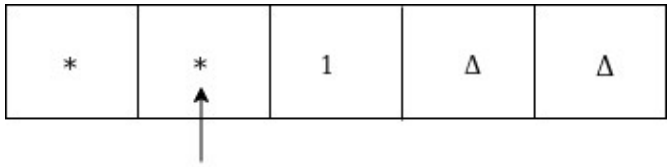
Moved right up to Δ



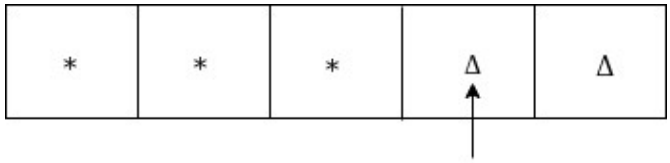
Move left and replace 0 by Δ as:



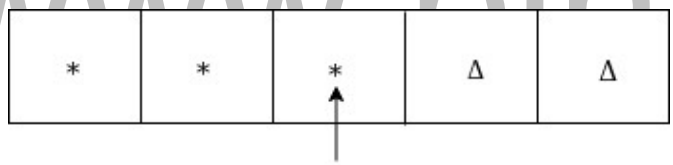
Move left till * as:



Move right and convert 1 to * as:

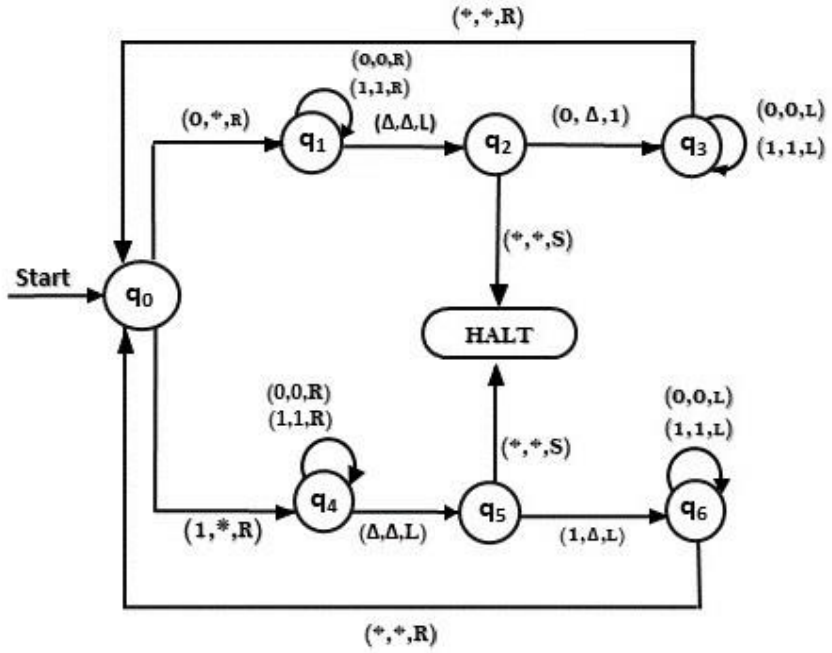


Move left



Since it is *, goto HALT state.

The same TM can be represented by Transition Diagram:



Example 4:

Construct TM for the addition function for the unary number system.

Solution:

The unary number is made up of only one character, i.e. The number 5 can be written in unary number system as 11111. In this TM, we are going to perform the addition of two unary numbers.

For example

2 + 3

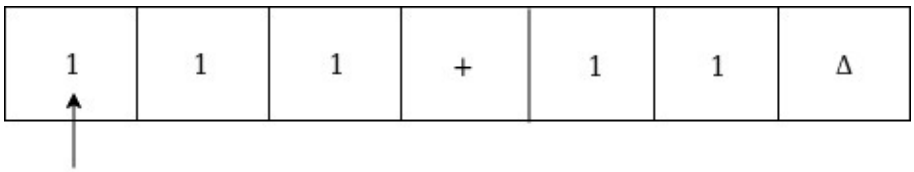
i.e. 11 + 111 = 11111

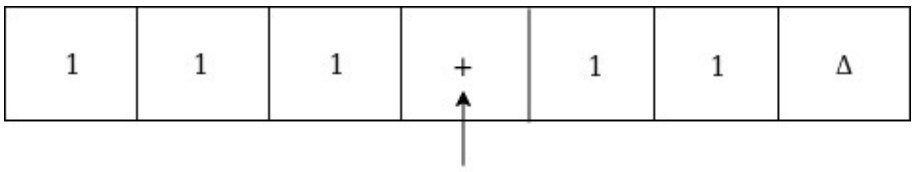
If you observe this process of addition, you will find the resemblance with string concatenation function.

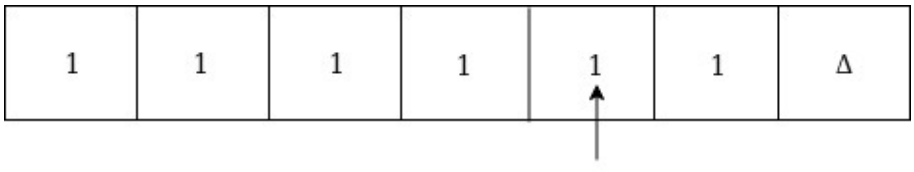
In this case, we simply replace + by 1 and move ahead right for searching end of the string we will convert last 1 to Δ.

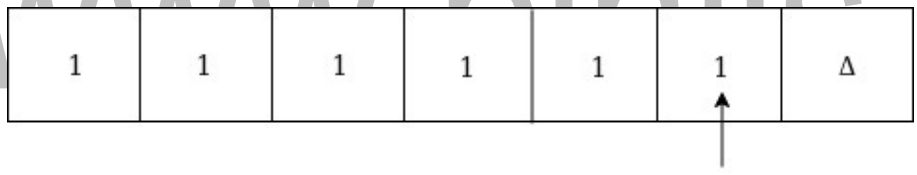
Input: 3+2

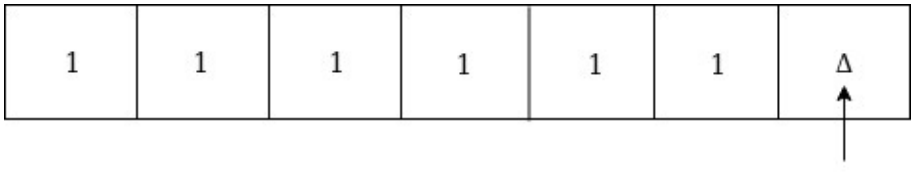
The simulation for 111+11Δ can be shown as below:

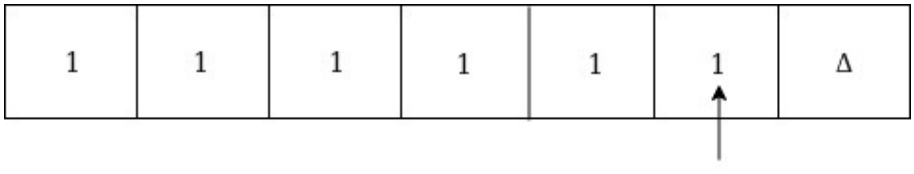










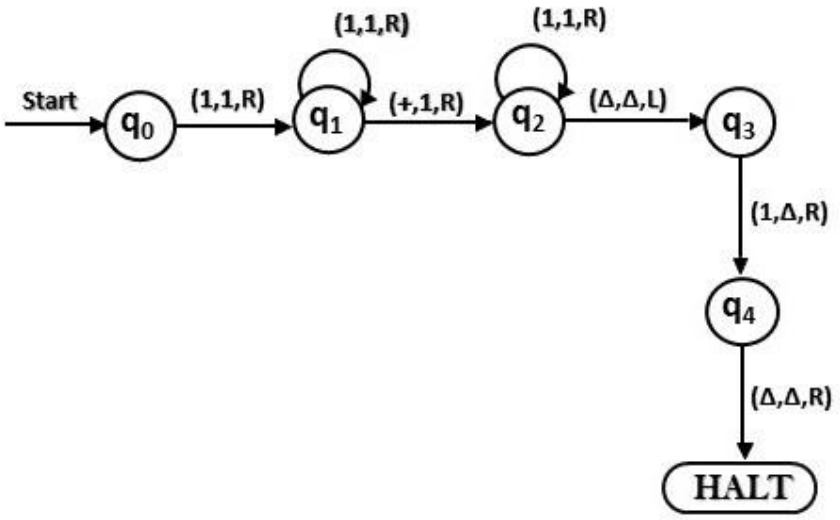


1	1	1	1	1	Δ	Δ

Thus the tape now consists of the addition of two unary numbers.

The TM will look like as follows:

Here, we are implementing the function of f(a + b) = c. We assume a and b both are non zero elements.



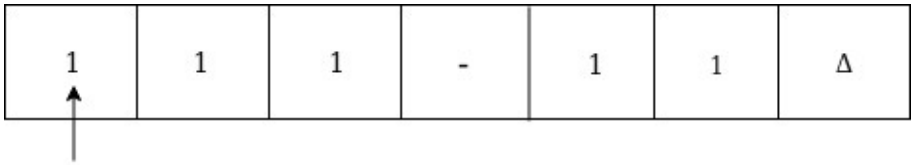
Example 5:

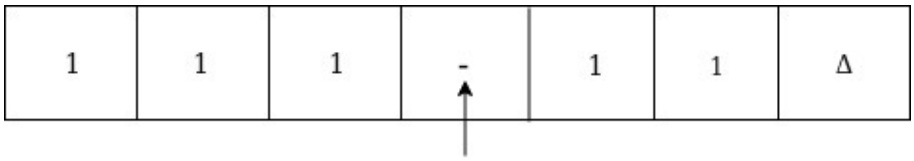
Construct a TM for subtraction of two unary numbers f(a-b) = c where a is always greater than b.

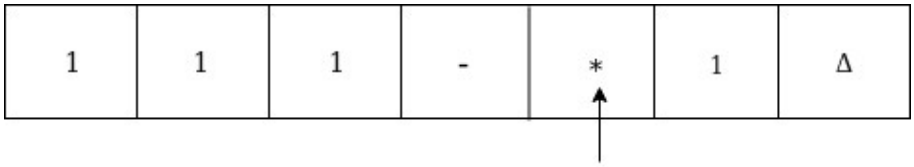
Solution: Here we have certain assumptions as the first number is greater than the second one. Let us assume that a = 3, b = 2, so the input tape will be:

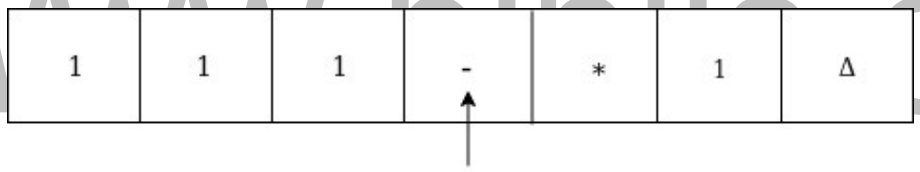
1	1	1	-	1	1	Δ

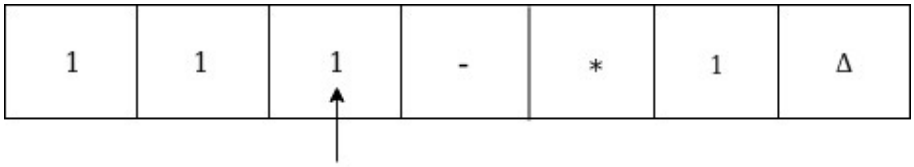
We will move right to - symbol as perform reduction of a number of 1's from the first number. Let us look at the simulation for understanding the logic:

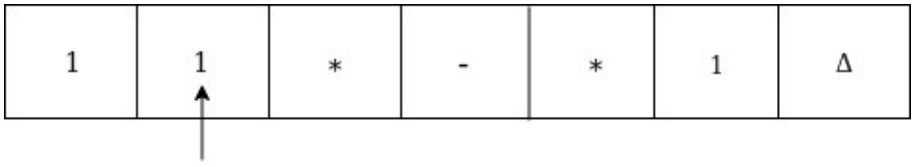


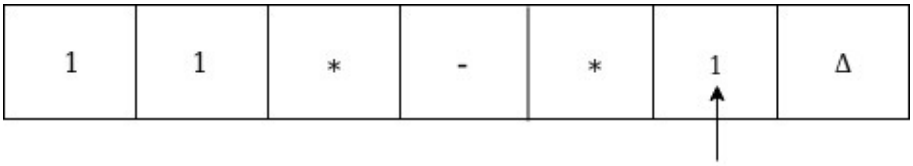


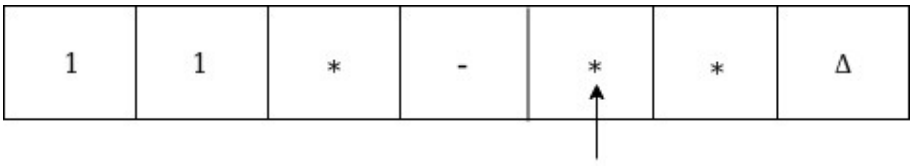


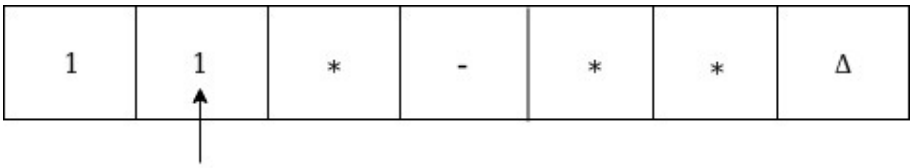


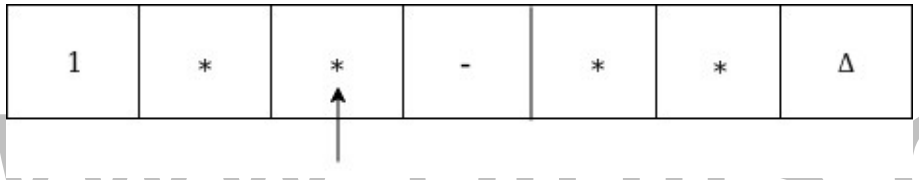


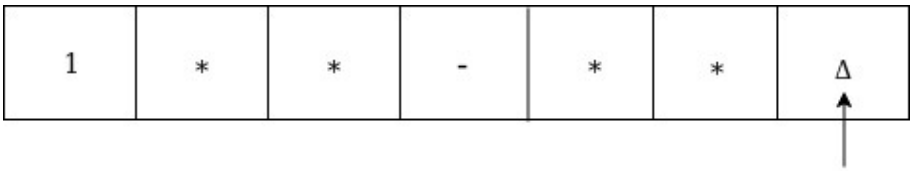


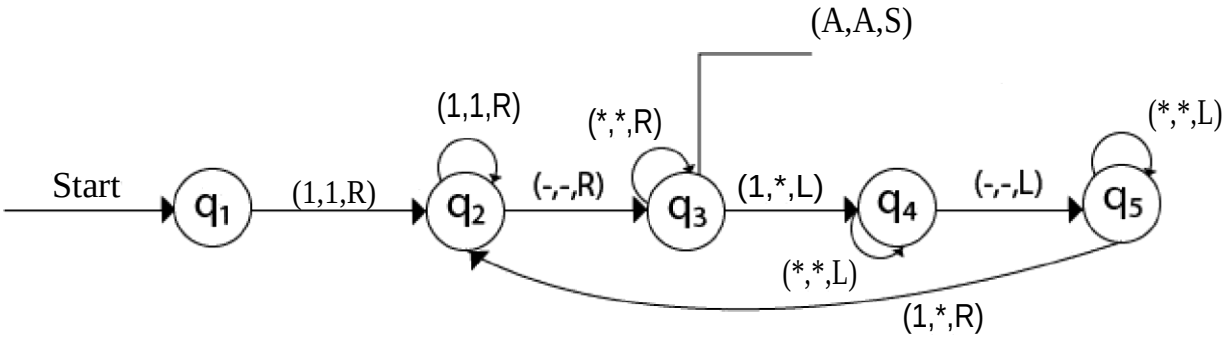












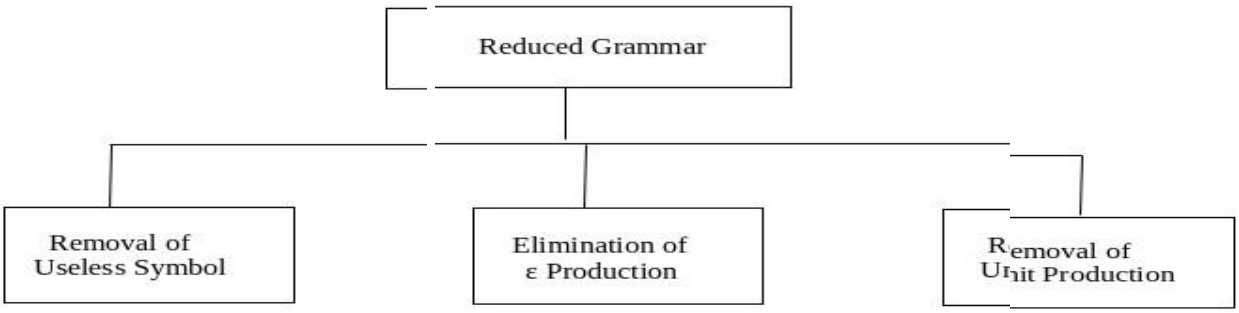
NORMAL FORMS FOR CFG

Simplification of CFG

As we have seen, various languages can efficiently be represented by a context-free grammar. All the grammar are not always optimized that means the grammar may consist of some extra symbols(non-terminal). Having extra symbols, u nnecessary increase the length of grammar. Simplification of grammar means reduction of grammar by removing useless symbols. The pro perties of reduced grammar are given below:

1. Each variable (i.e. non-terminal) and each terminal of G appears in the derivation of some word in L.
2. There should not be any production as X → Y where X and Y are non-terminal.
3. If ε is not in the language L then there need not to be the production X → ε.

Let us study the reduction process in detail.



Removal of Useless Symbols

A symbol can be useless if it does n ot appear on the right-hand side of the produc tion rule and does not take part in the derivation of any string. That symbol is known as a useless symbol. Similarly, a variable can be useless if it does not take part in the derivation of any string. That variable is known as a useless variable.

For Example:

1. T → aaB | abA | aaT
2. A → aA

3. B → ab | b
4. C → ad

In the above example, the variable 'C' will never occur in the derivation of any string, so the production C → ad is useless. So we will eliminate it, and the other productions are written in such a way that variable C can never reach from the starting variable 'T'.

Production A → aA is also useless because there is no way to terminate it. If it never terminates, then it can never produce a string. Hence this production can never take part in any derivation.

To remove this useless production A → aA, we will first find all the variables which will never lead to a terminal string such as variable 'A'. Then we will remove all the productions in which the variable 'B' occurs.

Elimination of ε Production

The productions of type S → ε are called ε productions. These type of productions can only be removed from those grammars that do not generate ε.

Step 1: First find out all nullable non-terminal variable which derives ε.

Step 2: For each production A → a, construct all production A → x, where x is obtained from a by removing one or more non-terminal from step 1.

Step 3: Now combine the result of step 2 with the original production and remove ε productions.

Example:

Remove the production from the following CFG by preserving the meaning of it.

1. S → XYX
2. X → 0X | ε
3. Y → 1Y | ε

Solution:

Now, while removing ε production, we are deleting the rule X → ε and Y → ε. To preserve the meaning of CFG we are actually placing ε at the right-hand side whenever X and Y have appeared.

Let us take

1. S → XYX

   If the first X at right-hand side is ε. Then

1. S → YX

   Similarly if the last X in R.H.S. = ε. Then

1. S → XY

1.

1. If Y = ε then

   S → XX

1. If Y and X are ε then,

   S → X

1. If both X are replaced by ε

   S → Y

1. Now,

   S → XY | YX | XX | X | Y

1. Now let us consider

2. X → 0X | 0

   X → 0X

   Similarly Y → 1Y | 1

   If we place ε at right-hand side for X then,

   Collectively we can rewrite the CFG with removed ε production as

   X → 0

1. S → XY | YX | XX | X | Y
2. X → 0X | 0
3. Y → 1Y | 1

Removing Unit Productions

The unit productions are the productions in which one non-terminal gives another non-terminal. Use the following steps to remove unit production:

Step 1: To remove X → Y, add production X → a to the grammar rule whenever Y → a occurs in the grammar.

Step 2: Now delete X → Y from the grammar.

1.
2.
3.
4.

1.

Step 3: Repeat step 1 and step 2 until all unit productions are

1. For example:
   S → 0A | 1B |
   C A → 0S | 00
   B → 1 | A
   C → 01

1. S → 0A | 1B | 01
2. A → 0S | 00
3. B → 1 | 0S | 00
4. C → 01

Solution:

Chomsky's Normal Form (CNF)

S → C is a unit production but while removing S → C we have to consider what C gives. So, we can add a rule to S.

CNF stands for Chomsky normal form. A CFG(context free grammar) is in CNF(Chomsky normal form) if all production rules satisfy one of the following conditions:

S → 0A | 1B |

Similarly, B → A is also a unit production so we can modify

- Start symbol generating ε. For example, A → ε.
- A non-terminal generating two non-terminals. For example, S → AB.
- A non-terminal generating a terminal. For example, S → a.

For example:

1. G1 = {S → AB, S → c, A → a, B → b}
2. G2 = {S → aA, A → a, B → c}

The production rules of Grammar G1 satisfy the rules specified for CNF, so the grammar G1 is in CNF. However, the production rule of Grammar G2 does not satisfy the rules specified for CNF as S → aZ contains terminal followed by non-terminal. So the grammar G2 is not in CNF.

Steps for converting CFG into CNF

Step 1: Eliminate start symbol from the RHS. If the start symbol T is at the right-hand side of any production, create a new production as:

1. S1 → S

Where S1 is the new start symbol.

Step 2: In the grammar, remove the null, unit and useless productions. You can refer to the Simplification of CFG.

Step 3: Eliminate terminals from the RHS of the production if they exist with other non-terminals or terminals. For example, production S → aA can be decomposed as:

1. S → RA
2. R → a

Step 4: Eliminate RHS with more than two non-terminals. For example, S → ASB can be decomposed as:

1. S → RS
2. R → AS

Example:

Convert the given CFG to CNF. Consider the given grammar G1:

1. S → a | aA | B
2. A → aBB | ε
3. B → Aa | b

Solution:

Step 1: We will create a new production S1 → S, as the start symbol S appears on the RHS. The grammar will be:

1. S1 → S

S → a | aA | B

A → aBB | ε B → Aa | b

Step 2: As grammar G1 contains A → ε null production, its removal from the grammar yields:
S1 → S

S → a | aA | B A → aBB

B → Aa | b | a

Now, as grammar G1 contains Unit production S → B, its removal yield:

S1 → S

S → a | aA | Aa | b A → aBB

B → Aa | b | a

Also remove the unit production S1 → S, its removal from the grammar yields:

1. S0 → a | aA | Aa | b
2. S → a | aA | Aa | b
3. A → aBB
4. B → Aa | b | a

Step 3: In the production rule S0 → aA | Aa, S → aA | Aa, A → aBB and B → Aa, terminal a exists on RHS with non-terminals. So we will replace terminal a with X:

1. S0 → a | XA | AX | b

2. S → a | XA | AX | b
3. A → XBB
4. B → AX | b | a
5. X → a

Step 4: In the production rule A → XBB, RHS has more than two symbols, removing it from grammar yield:

1. S0 → a | XA | AX | b
2. S → a | XA | AX | b
3. A → RB
4. B → AX | b | a
5. X → a
6. R → XB

Hence, for the given grammar, this is the required CNF.

Greibach Normal Form (GNF)

GNF stands for Greibach normal form. A CFG(context free grammar) is in GNF(Greibach normal form) if all the production rules satisfy one of the following conditions:

- A start symbol generating ε. For example, S → ε.
- A non-terminal generating a terminal. For example, A → a.
- A non-terminal generating a terminal which is followed by any number of non-terminals. For example, S → aASB.

For example:

1. G1 = {S → aAB | aB, A → aA| a, B → bB | b}
2. G2 = {S → aAB | aB, A → aA | ε, B → bB |ε}

The production rules of Grammar G1 satisfy the rules specified for GNF, so the grammar G1 is in GNF. However, the production rule of Grammar G2 does not satisfy the rules specified for GNF as A → ε and B → ε contains ε(only start symbol can generate ε). So the grammar G2 is not in GNF.

Steps for converting CFG into GNF

Step 1: Convert the grammar into CNF.

If the given grammar is not in CNF, convert it into CNF. You can refer the following topic to convert the CFG into CNF: Chomsky normal form

Step 2: If the grammar exists left recursion, eliminate it.

If the context free grammar contains left recursion, eliminate it. You can refer the following topic to eliminate left recursion: Left Recursion

Step 3: In the grammar, convert the given production rule into GNF form.

1. If any production rule in the grammar is not in GNF form, convert it.
Example:

S → XB | AA

A → a | SA B → b

X → a

Solution:

As the given grammar G is already in CNF and there is no left recursion, so we can skip step 1 and step 2 and directly go to step 3.
The production rule A → SA is not in GNF, so we substitute S → XB | AA in the production rule A → SA as:
S → XB | AA

A → a | XBA | AAA B → b

X → a

The production rule S → XB and B → XBA is not in GNF, so we substitute X → a in the production rule S → XB and B → XBA as:

1. S → aB | AA
2. A → a | aBA | AAA
3. B → b
4. X → a

Now we will remove left recursion (A → AAA), we get:

1. S → aB | AA
2. A → aC | aBAC
3. C → AAC | ε
4. B → b
5. X → a

Now we will remove null production C → ε, we get:

1. S → aB | AA 2. A → aC | aBAC | a | aBA C → AAC | AA

B → b

X → a

rule S → AA as:

S → aB | aCA | aBACA | aA | aBAA A → aC | aBAC | a | aBA

C → AAC

C → aCA | aBACA | aA | aBAA B → b

X → a

The production rule S → AA is not in GNF, so we substitute A → aC | aBAC | a | aBA in production
The production rule C → AAC is not in GNF, so we substitute A → aC | aBAC | a | aBA in production

rule C → AAC as:

S → aB | aCA | aBACA | aA | aBAA

A → aC | aBAC | a | aBA

C → aCAC | aBACAC | aAC | aBAAC

3. C → aCA | aBACA | aA | aBAA
4. B → b
5. X → a

Hence, this is the GNF form for the grammar G.

PUMPING LEMMA FOR CFL

If L is a context-free language, there is a pumping length p such that any string w ∈ L of length ≥ p can be written as w = uvxyz, where vy ≠ ε, |vxy| ≤ p, and for all i ≥ 0, uvixyiz ∈ L.

Applications of Pumping Lemma

Pumping lemma is used to check whether a grammar is context free or not. Let us take an example and show how it is checked.

Problem

Find out whether the language L = {xnynzn | n ≥ 1} is context free or not.

Solution

Let L is context free. Then, L must satisfy pumping lemma.

At first, choose a number n of the pumping lemma. Then, take z as 0n1n2n.

Break z into uvwxy, where

|vwx| ≤ n and vx ≠ ε.

Hence vwx cannot involve both 0s and 2s, since the last 0 and the first 2 are at least (n+1) positions apart. There are two cases −

Case 1 − vwx has no 2s. Then vx has only 0s and 1s. Then uwy, which would have to be in L, has n 2s, but fewer than n 0s or 1s.

Case 2 − vwx has no 0s.

Here contradiction occurs.

Hence, L is not a context-free language.

CFL Closure Property

Context-free languages are closed under −

- Union
- Concatenation
- Kleene Star operation

Union

Let L1 and L2 be two context free languages. Then L1 ∪ L2 is also context free.

Example

Let L1 = { anbn , n > 0}. Corresponding grammar G1 will have P: S1 → aAb|ab

Let L2 = { cmdm , m ≥ 0}. Corresponding grammar G2 will have P: S2 → cBb| ε

Union of L1 and L2, L = L1 ∪ L2 = { anbn } ∪ { cmdm }

The corresponding grammar G will have the additional production S → S1 | S2

Concatenation

If L1 and L2 are context free languages, then L1L2 is also context free.

Example

Union of the languages L1 and L2, L = L1L2 = { anbncmdm }

The corresponding grammar G will have the additional production S → S1 S2

Kleene Star

If L is a context free language, then L* is also context free.

Example

Let L = { anbn , n ≥ 0}. Corresponding grammar G will have P: S → aAb| ε

Kleene Star L1 = { anbn }*

The corresponding grammar G1 will have additional productions S1 → SS1 | ε

Context-free languages are not closed under −

- Intersection − If L1 and L2 are context free languages, then L1 ∩ L2 is not necessarily context free.
- Intersection with Regular Language − If L1 is a regular language and L2 is a context free language, then L1 ∩ L2 is a context free language.
- Complement − If L1 is a context free language, then L1' may not be context free.

TURING MACHINE

Turing machine was invented in 1936 by **Alan Turing**. It is an accepting device which accepts Recursive Enumerable Language generated by type 0 grammar.

There are various features of the Turing machine:

1. It has an external memory which remembers arbitrary long sequence of input.
2. It has unlimited memory capability.
3. The model has a facility by which the input at left or right on the tape can be read easily.
4. The machine can produce a certain output based on its input. Sometimes it may be required that the same input has to be used to generate the output. So in this machine, the distinction between input and output has been removed. Thus a common set of alphabets can be used for the Turing machine.

Formal definition of Turing machine

A Turing machine can be defined as a collection of 7 components:

Q: the finite set of states
∑: the finite set of input symbols
T: the tape symbol
q0: the initial state
F: a set of final states
B: a blank symbol used as a end marker for input
δ: a transition or mapping function.

The mapping function shows the mapping from states of finite automata and input symbol on the tape to the next states, external symbols and the direction for moving the tape head. This is known as a triple or a program for turing machine.

1. (q0, a) → (q1, A, R)

That means in q0 state, if we read symbol 'a' then it will go to state q1, replaced a by X and move ahead right(R stands for right).

Example:

Construct TM for the language L ={0^n1^n} where n>=1.

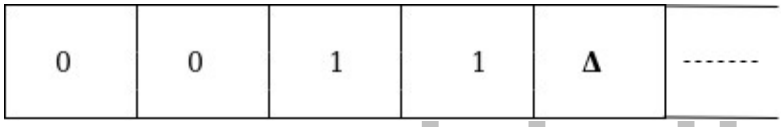
Solution:

We have already solved this problem by PDA. In PDA, we have a stack to remember the previous symbol. The main advantage of the Turing machine is we have a tape head which can be moved forward or backward, and the input tape can be scanned.

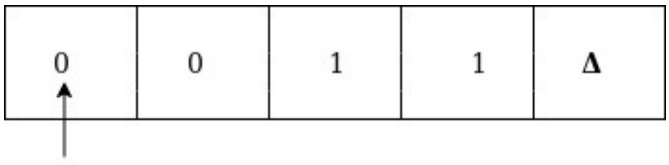
The simple logic which we will apply is read out each '0' mark it by A and then move ahead along with the input tape and find out 1 convert it to B. Now, repeat this process for all a's and b's.

Now we will see how this turing mac

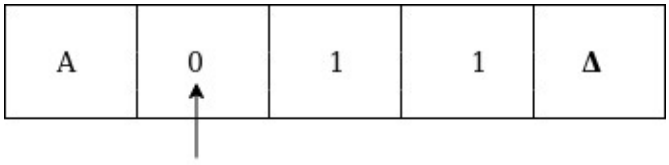
The simulation for 0011 can be shown as below:



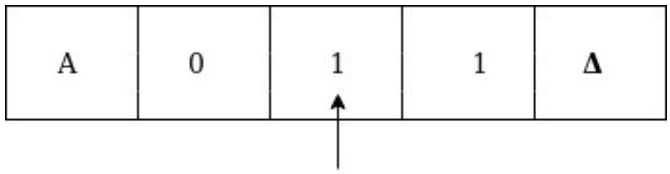
Now, we will see how this turing machine will works for 0011. Initially, state is q0 and head points to 0 as:



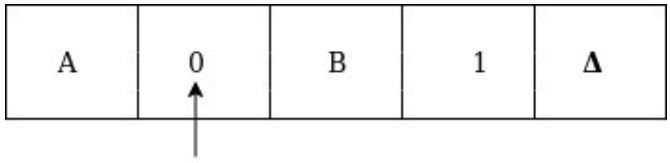
The move will be δ(q0, 0) = δ(q1, A, R) which means it will go to state q1, replaced 0 by A and head will move to the right as:



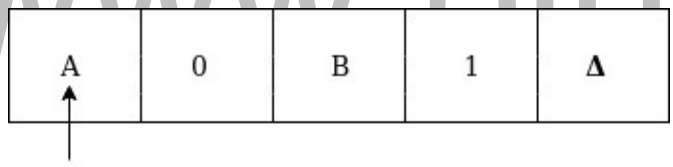
The move will be δ(q1, 0) = δ(q1, 0, R) which means it will not change any symbol, remain in the same state and move to the right as:



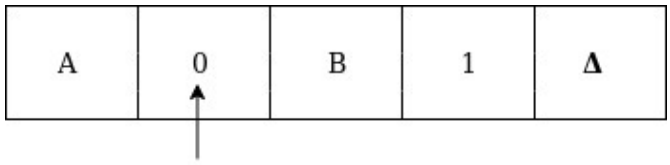
The move will be δ(q1, 1) = δ(q2, B, L) which means it will go to state q2, replaced 1 by B and head will move to left as:



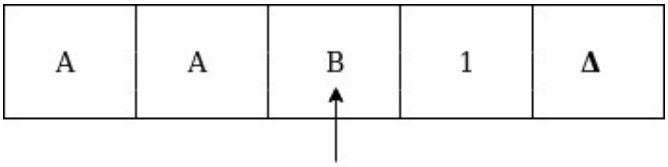
Now move will be δ(q2, 0) = δ(q2, 0, L) which means it will not change any symbol, remain in the same state and move to left as:



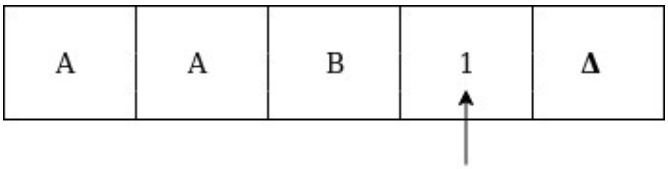
The move will be δ(q2, A) = δ(q0, A, R), it means will go to state q0, replaced A by A and head will move to the right as:



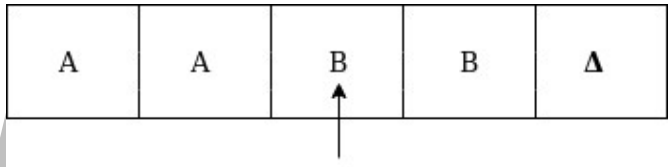
The move will be δ(q0, 0) = δ(q1, A, R) which means it will go to state q1, replaced 0 by A, and head will move to right as:



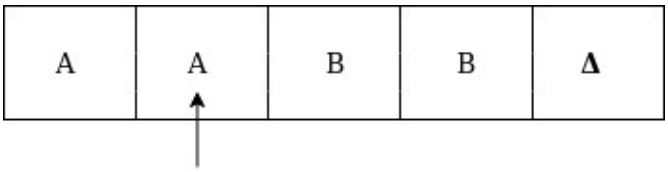
The move will be δ(q1, B) = δ(q1, B, R) which means it will not change any symbol, remain in the same state and move to right as:



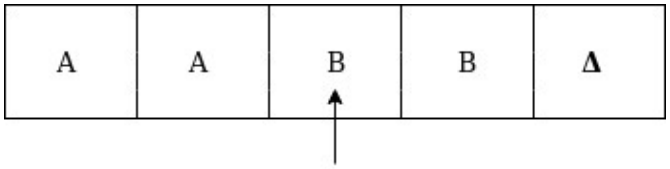
The move will be δ(q1, 1) = δ(q2, B, L) which means it will go to state q2, replaced 1 by B and head will move to left as:



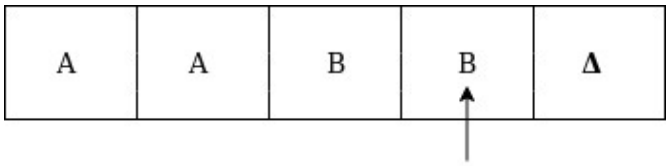
The move δ(q2, B) = (q2, B, L) which means it will not change any symbol, remain in the same state and move to left as:



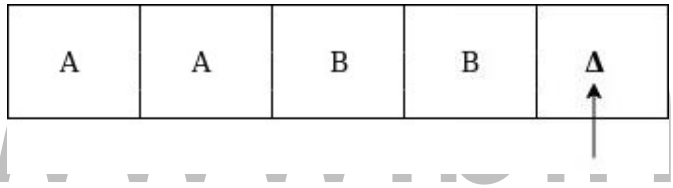
Now immediately before B is A that means all the 0?s are market by A. So we will move right to ensure that no 1 is present. The move will be δ(q2, A) = (q0, A, R) which means it will go to state q0, will not change any symbol, and move to right as:



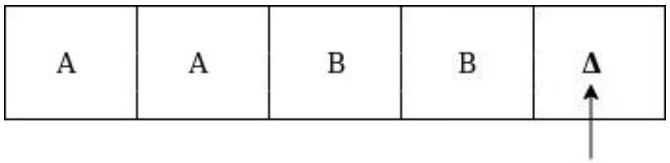
The move δ(q0, B) = (q3, B, R) which means it will go to state q3, will not change any symbol, and move to right as:



The move δ(q3, B) = (q3, B, R) which means it will not change any symbol, remain in the same state and move to right as:



The move δ(q3, Δ) = (q4, Δ, R) which means it will go to state q4 which is the HALT state and HALT state is always an accept state for any TM.

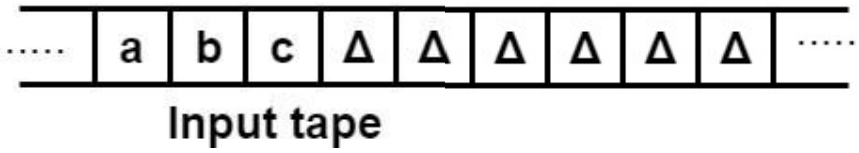


The same TM can be represented by Transition Diagram:

Basic Model of Turing machine

The turning machine can be modelled with the help of the following representation.

1. The input tape is having an infinite number of cells, each cell containing one input symbol and thus the input string can be placed on tape. The empty tape is filled by blank characters.



Input tape

2. The finite control and the tape head which is responsible for reading the current input symbol. The tape head can move to left to right.

3. A finite set of states through which machine has to undergo.

4. Finite set of symbols called external symbols which are used in building the logic of turing machine.

Language accepted by Turing machine

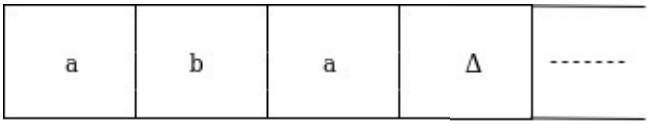
The turing machine accepts all the language even though they are recursively enumerable. Recursive means repeating the same set of rules for any number of times and enumerable means a list of elements. The TM also accepts the computable functions, such as addition, multiplication, subtraction, division, power function, and many more.

Example:

Construct a turing machine which accepts the language of aba over $\sum = \{a, b\}$.

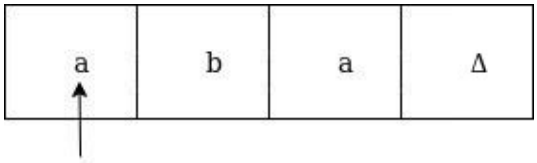
Solution:

We will assume that on input tape the string 'aba' is placed like this:

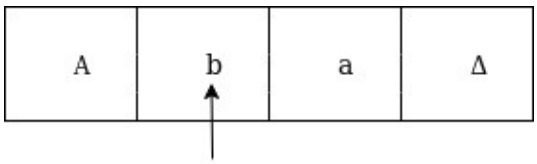


The tape head will read out the sequence up to the Δ characters. If the tape head is readout 'aba' string then TM will halt after reading Δ.

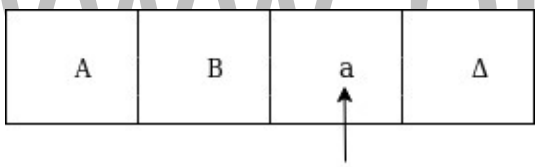
Now, we will see how this turing machine will work for aba. Initially, state is q0 and head points to a as:



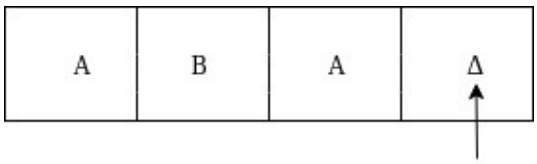
The move will be δ(q0, a) = δ(q1, A, R) which means it will go to state q1, replaced a by A and head will move to right as:



The move will be δ(q1, b) = δ(q2, B, R) which means it will go to state q2, replaced b by B and head will move to right as:



The move will be δ(q2, a) = δ(q3, A, R) which means it will go to state q3, replaced a by A and head will move to right as:



The move δ(q3, Δ) = (q4, Δ, S) which means it will go to state q4 which is the HALT state and HALT state is always an accept state for any TM.

The same TM can be represented by Transition Table:

States	A	b	Δ
q0	(q1, A, R)	–	–
q1	–	(q2, B, R)	–
q2	(q3, A, R)	–	–
q3	–	–	(q4, Δ, S)
q4	–	–	–

The same TM can be represented by Transition Diagram: