CSE 215: Data Structures & Algorithms II

Week 1
Heaps and Priority Queue

Lec Sumaiya Nuha Mustafina
Dept Of CSE
sumaiyamustafina@cse.mist.ac.bd

Google Classroom (CSE 215+ CSE 216)

o4d3khk

Course Outline

Link to the outline

Heap?

Is it the same as heap memory ?

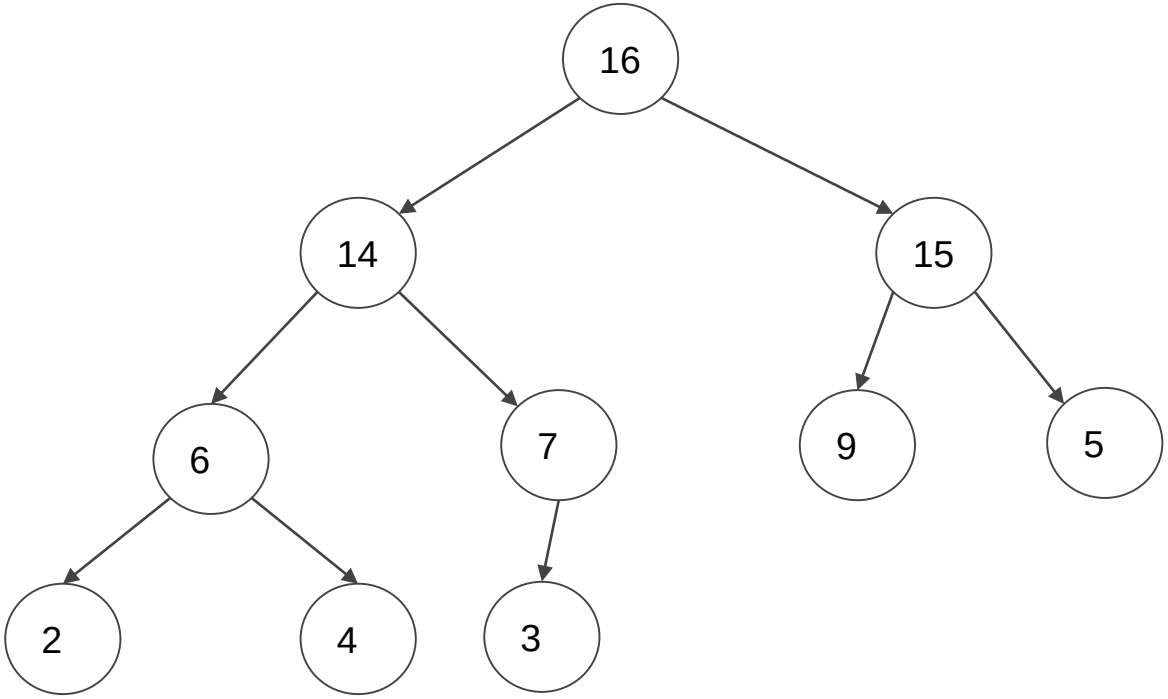
Is it the same as heap memory ?

Heap Data Structure

- The (binary) heap data structure is a **complete binary tree** that satisfies the **heap property.**
- An **array** object that can be viewed as a **complete binary tree.**
- Each node of the tree corresponds to an element of the array.
- The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point.
- Application: Priority Queue, Heapsort etc.

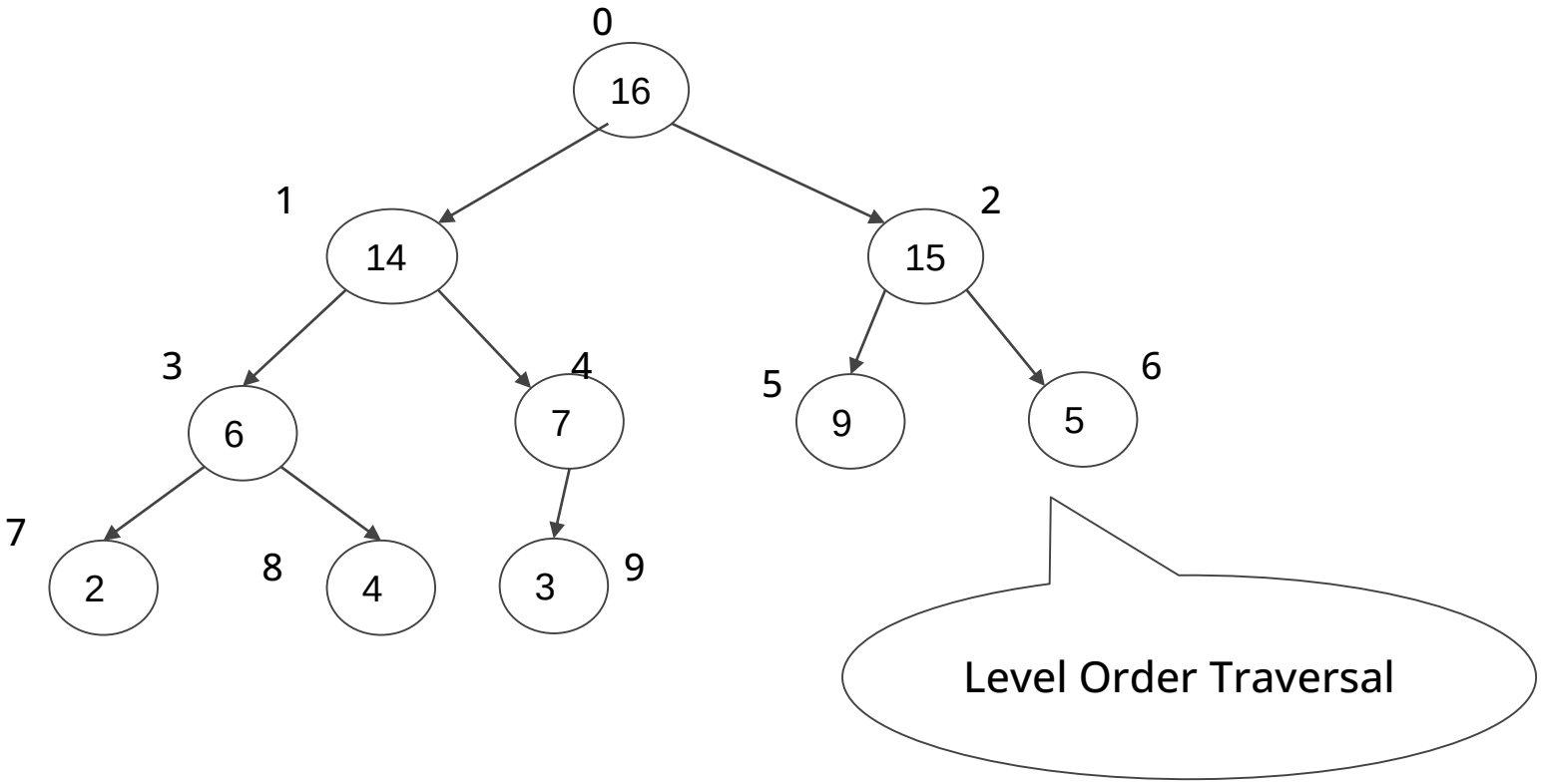
Heap Data Structure

View as a Binary Tree



Heap Data Structure

View as an Array



0	1	2	3	4	5	6	7	8	9
16	14	15	6	7	9	5	2	4	3

Heap Data Structure

- Two kinds of binary heaps:
  - max-heaps
  - min-heaps.

- In both kinds, the values in the nodes satisfy a **heap property**

Heap Properties

In a **max-heap**, the max-heap property is that for every node i other than the root,

A[PARENT(i)] >= A[i] ;

that is, the value of a node is at most the value of its parent.

Thus, the largest element in a max-heap is stored at the root, and the subtree rooted at a node contains values no larger than that contained at the node itself.

Heap Properties

In a **min-heap**; the min-heap property is that for every node i other than the root,

```
A[PARENT( i )] <= A[ i ] ;
```

The smallest element in a min-heap is at the root.

Heap

Q: Is the following array a max-heap?

23	17	14	6	13	10	1	5	7	12

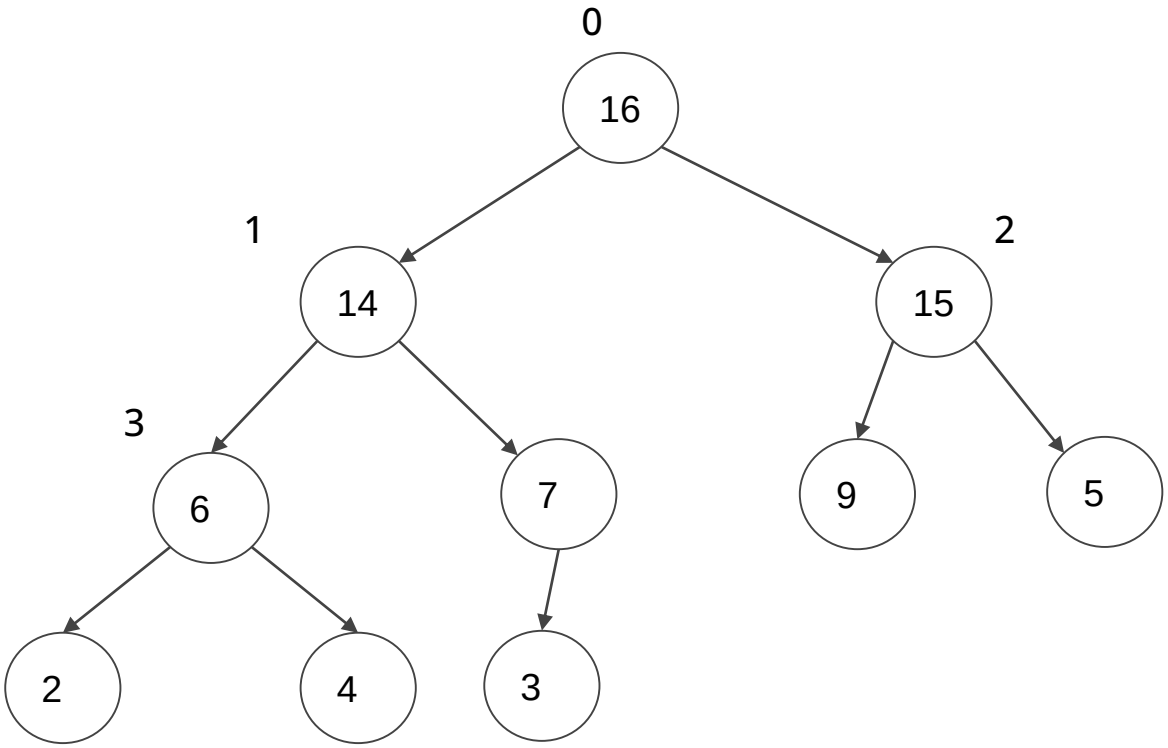
Heap Data Structure

- Since a heap of n elements is based on a complete binary tree, its height is $\Theta(\log n)$.
- The total number of comparisons required in heap is according to the height of the tree.
- Thus the **time complexity** of basic operation would also be O(logn).

Heap: Some Basic Procedures

- The MAX-HEAPIFY procedure, which runs in O(log n) time, is the key to maintaining the **max-heap property**.
- The BUILD-MAX-HEAP procedure, which runs in linear time, **produces a max-heap** from an unordered input array.
- The HEAPSORT procedure, which runs in O(n.log n) time, **sorts** an array **in place**.
- The MAX-HEAP-INSERT ,HEAP-EXTRACT-MAX , HEAP-INCREASE-KEY, and HEAP-MAXIMUM procedures, which run in O(log n) time, allow the heap data structure to implement a **priority queue**.

Heap represented as a Binary tree and an array



0	1	2	3	4	5	6	7	8	9
16	14	15	6	7	9	5	2	4	3

Heap

For 0-based indexing of node numbers.

```
PARENT(i)
  return ⌊(i-1)/2⌋

LEFT(i)
  return 2i+1

RIGHT(i)
  return 2i+2
```

For leave nodes: indexed by $\lfloor (n-1)/2 \rfloor$, $\lfloor (n-1)/2 \rfloor + 1$, ... $n-1$.
n: number of elements in heap.

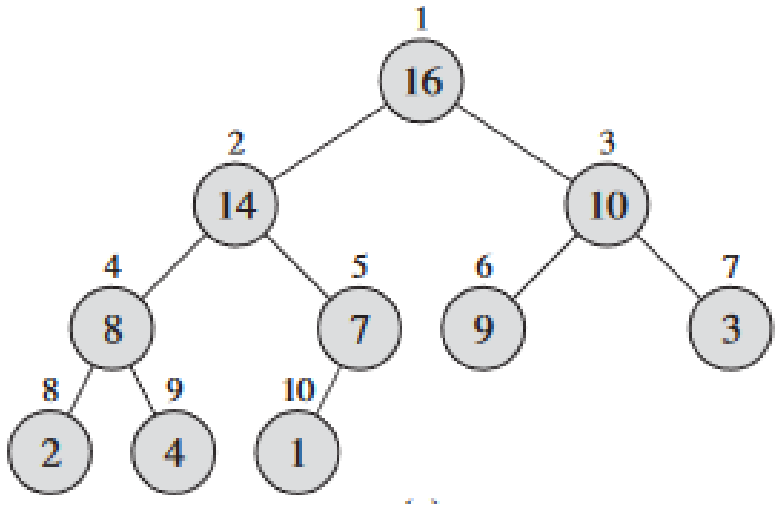
Heap

For 1-based indexing of node numbers.

PARENT(i)
  return ⌊i/2⌋

LEFT(i)
  return 2i

RIGHT(i)
  return 2i+1



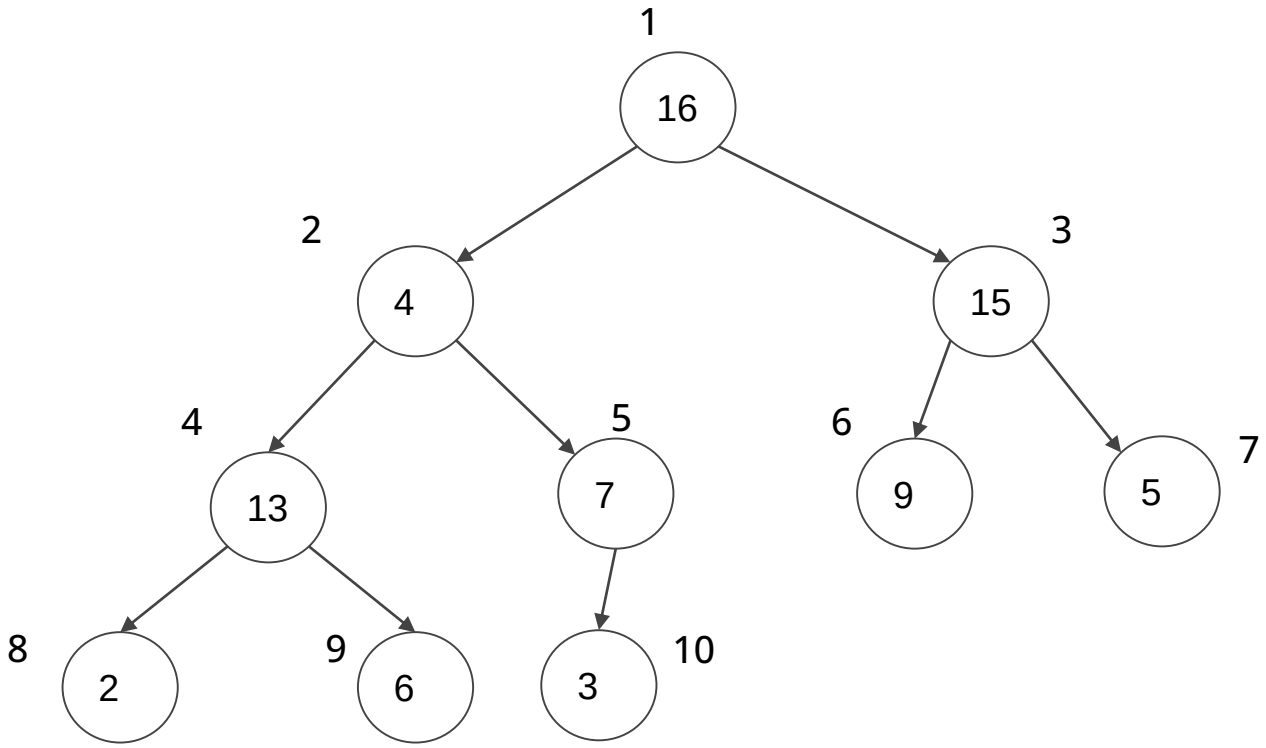
For leave nodes: indexed by ⌊n/2⌋+1, ⌊n/2⌋+2, ….. n.

**All the algorithms here are written considering 1-based indexing.

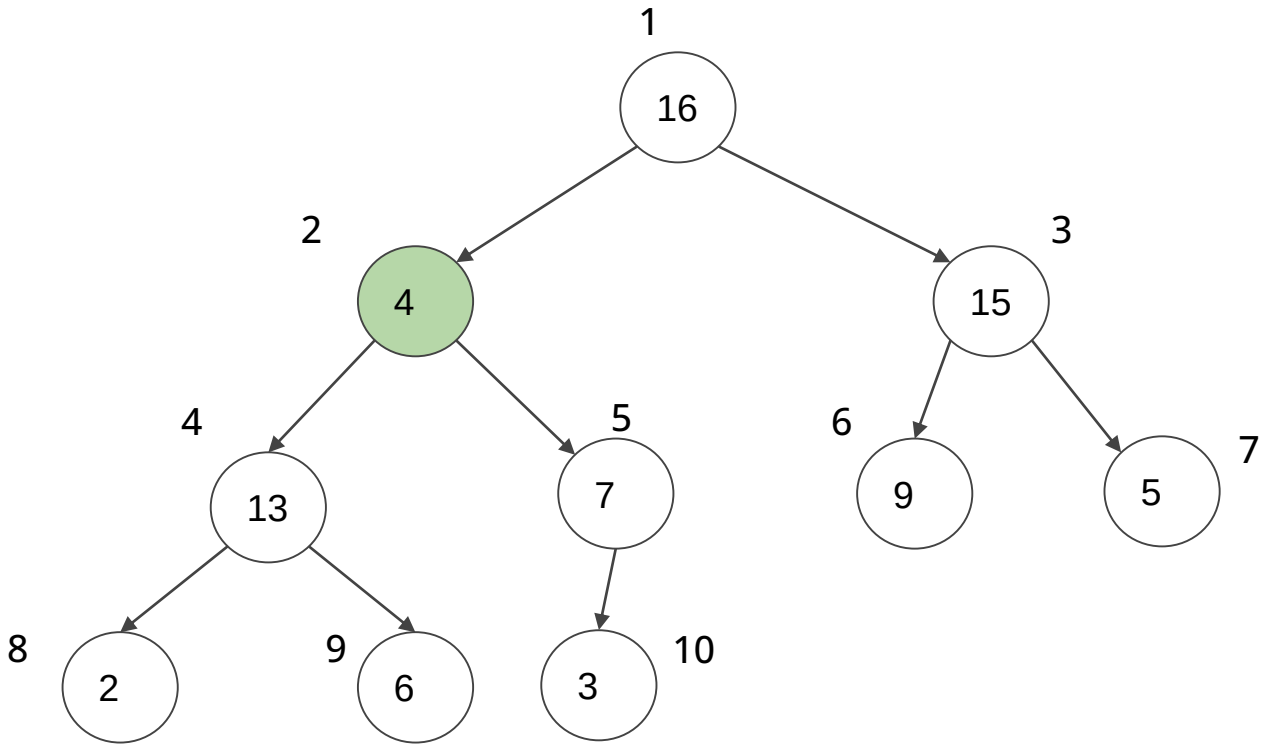
Maintaining the heap property

```
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)
```

MAX-HEAP Property!!



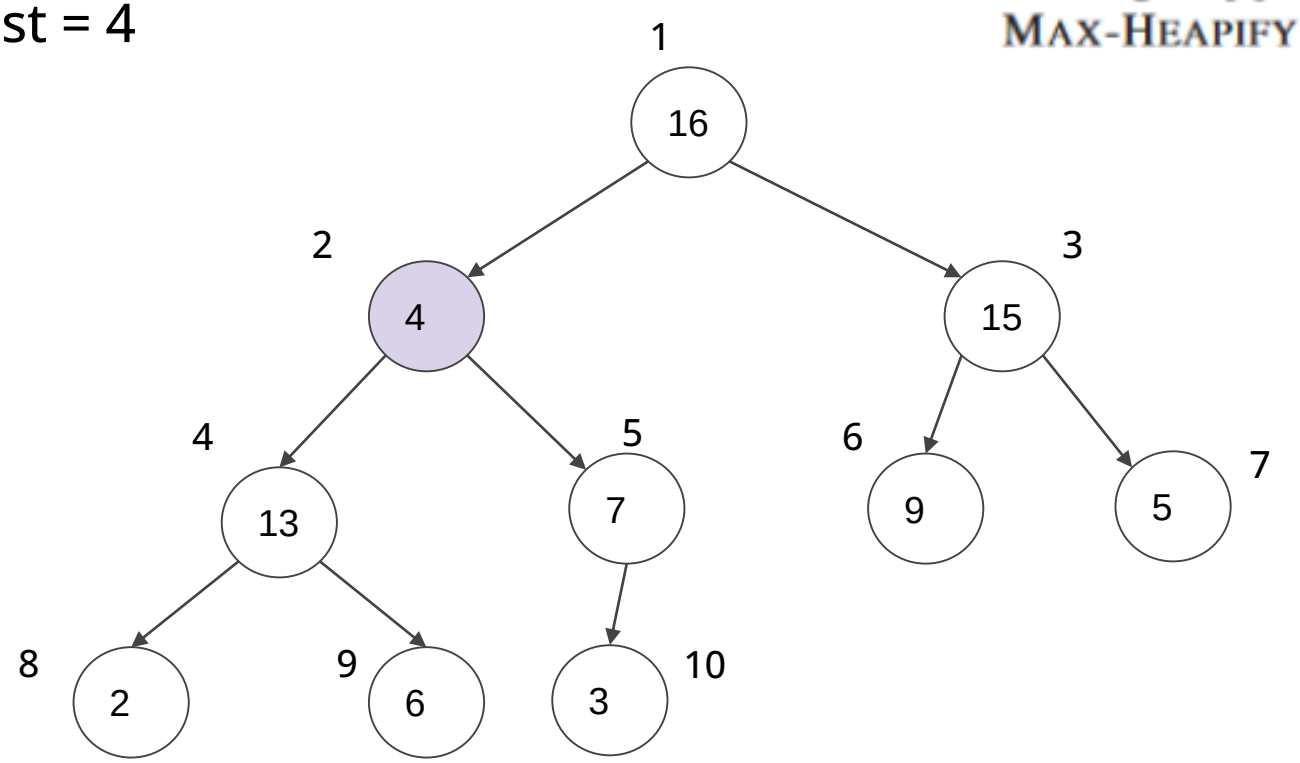
MAX-HEAP Property!!



MAX-HEAPIFY(A,2)

l = 4; r = 5
largest = 4

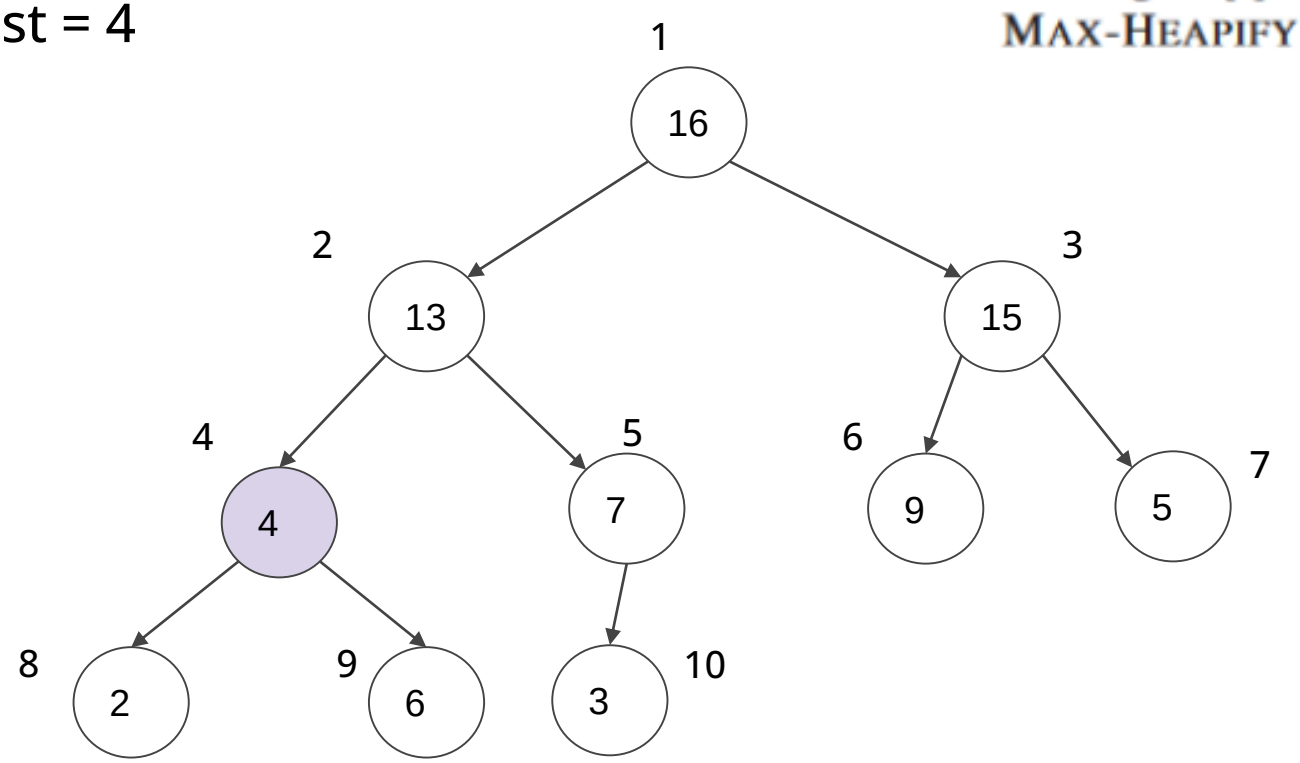
if *largest* $\neq$ *i*
exchange *A*[*i*] with *A*[*largest*]
MAX-HEAPIFY (*A*, *largest*)



MAX-HEAPIFY(A,2)

l = 4; r = 5
largest = 4

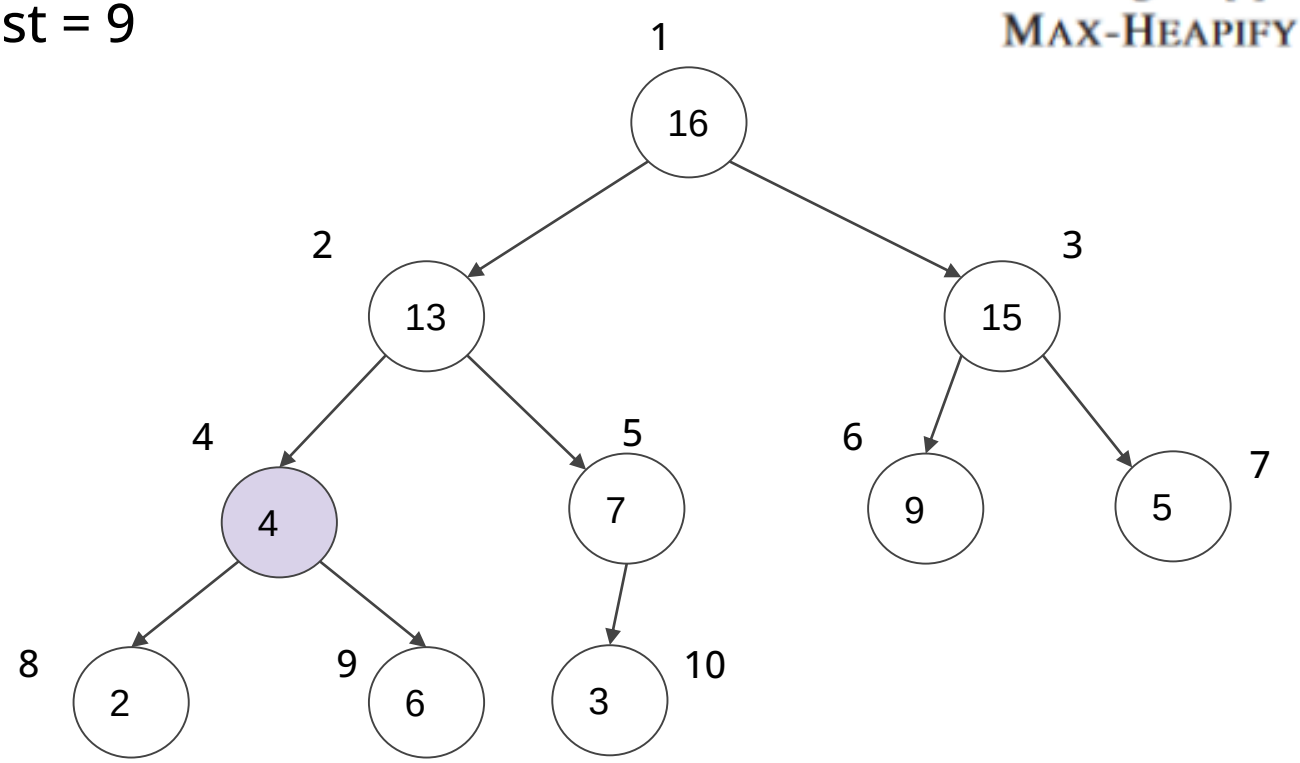
if *largest* $\neq$ *i*
exchange *A*[*i*] with *A*[*largest*]
MAX-HEAPIFY (*A*, *largest*)



MAX-HEAPIFY(A,4)

l = 8; r = 9
largest = 9

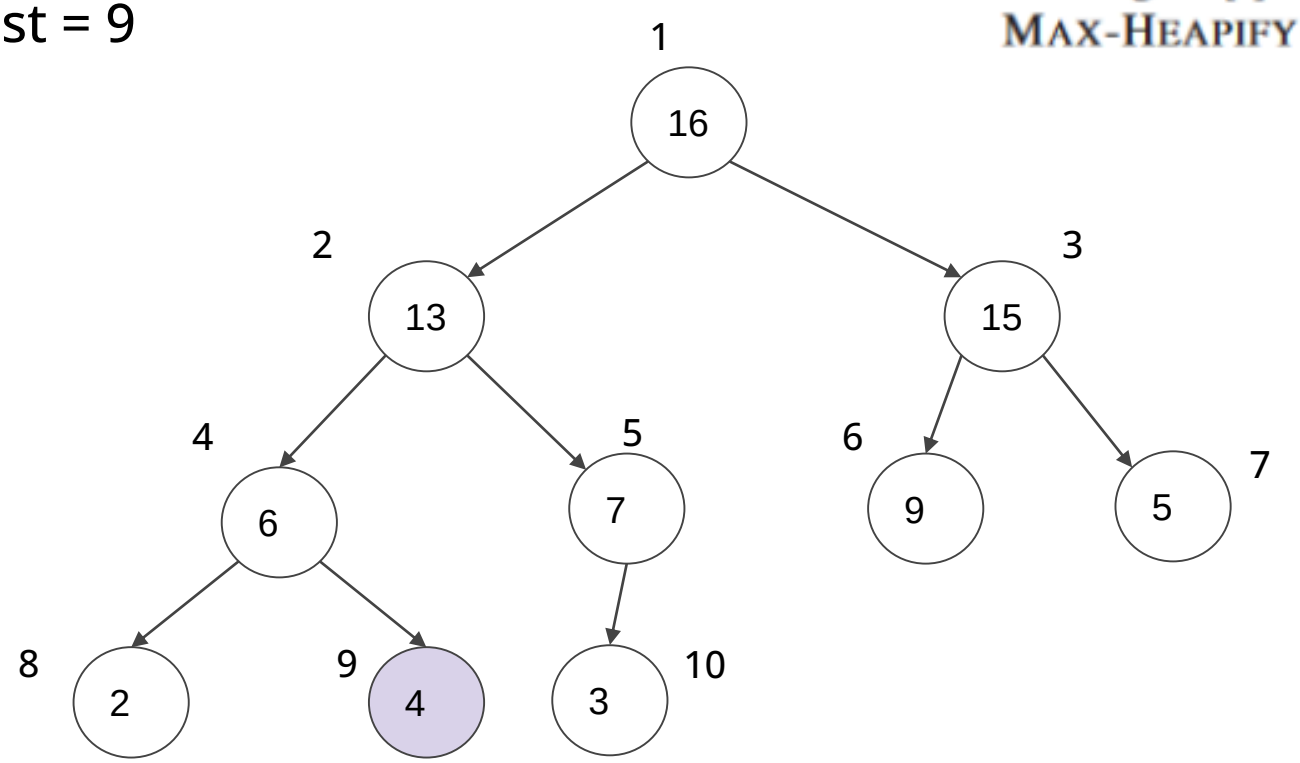
if *largest* $\neq i$
exchange $A[i]$ with $A[largest]$
MAX-HEAPIFY $(A, largest)$



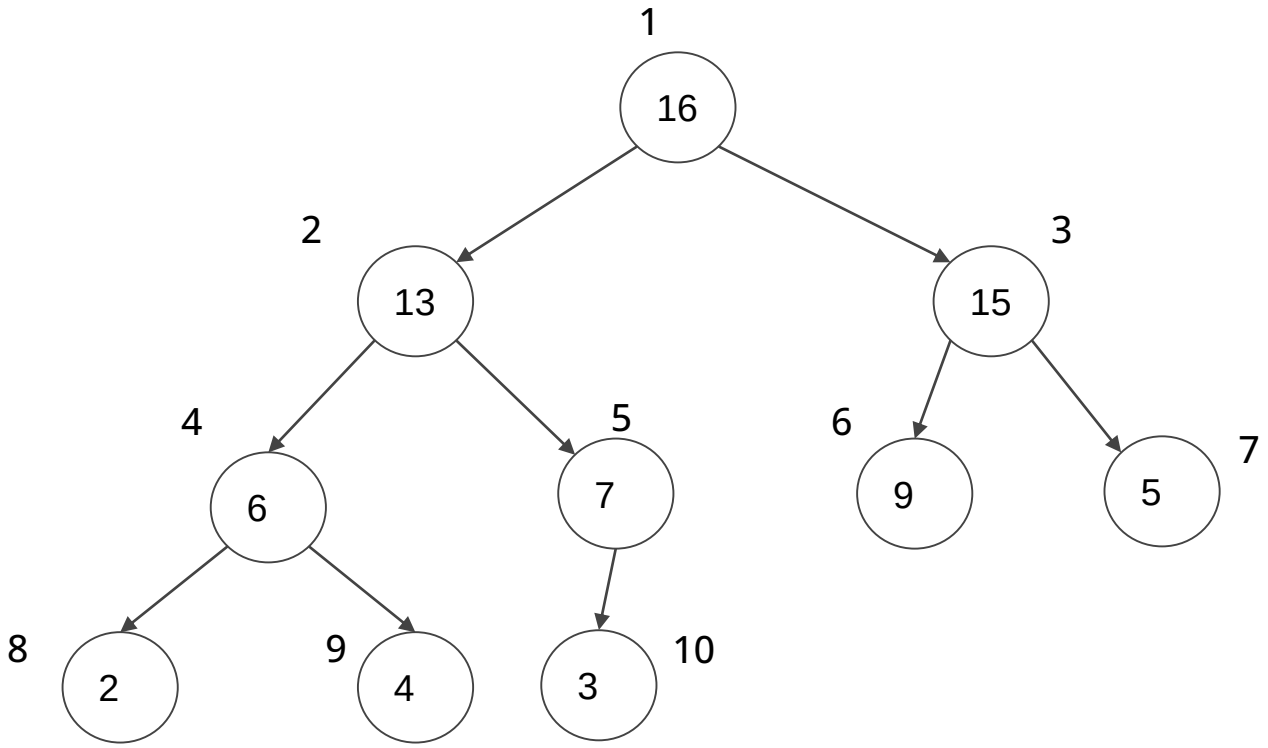
MAX-HEAPIFY(A,4)

l = 8; r = 9
largest = 9

if *largest* $\neq i$
exchange $A[i]$ with $A[largest]$
MAX-HEAPIFY $(A, largest)$



MAX HEAP:



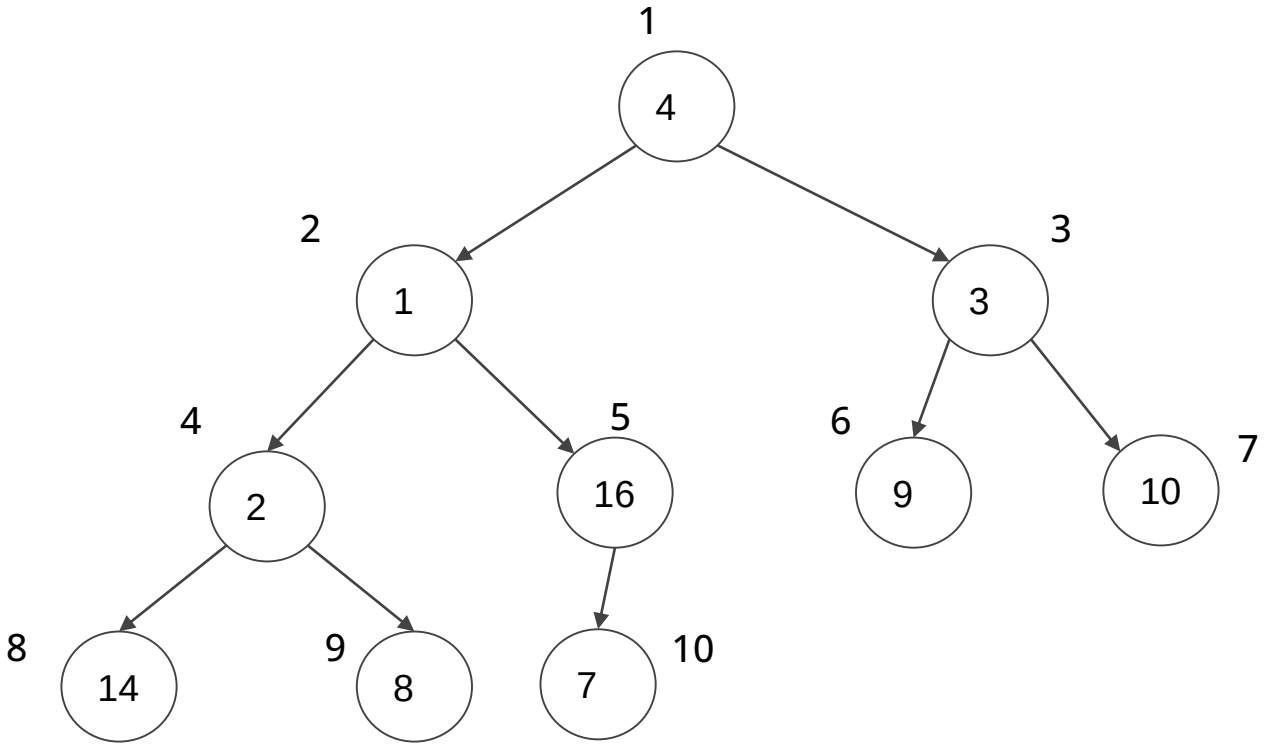
Building a Heap from a given array

- Array A[1..n], where n= A:length
- Use MAX-HEAPIFY in a bottom-up manner to convert into a max-heap.
- The procedure BUILD-MAX-HEAP goes through the non-leaf nodes of the tree and runs MAX-HEAPIFY on each one.

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

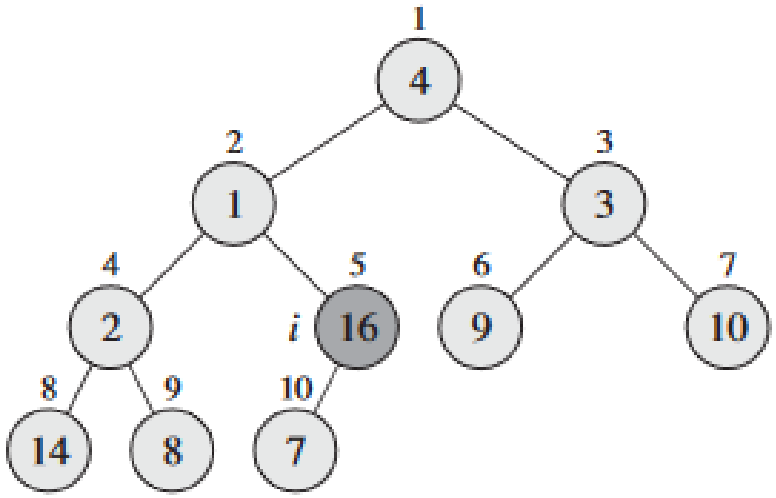
Building a Heap from a given array

0	1	2	3	4	5	6	7	8	9	10
	4	1	3	2	16	9	10	14	8	7



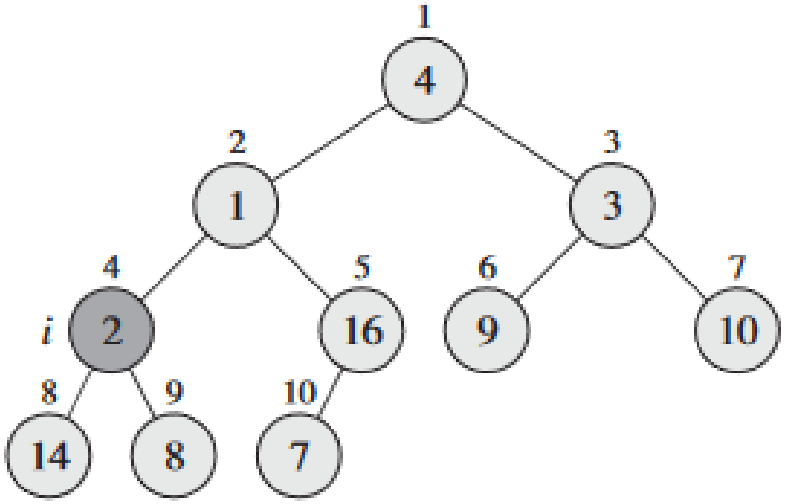
Building a Heap from a given array

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```



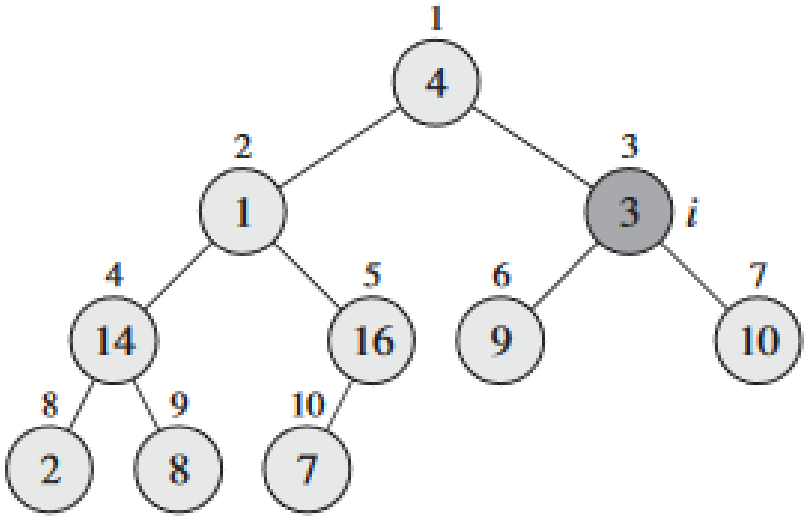
Building a Heap from a given array

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```



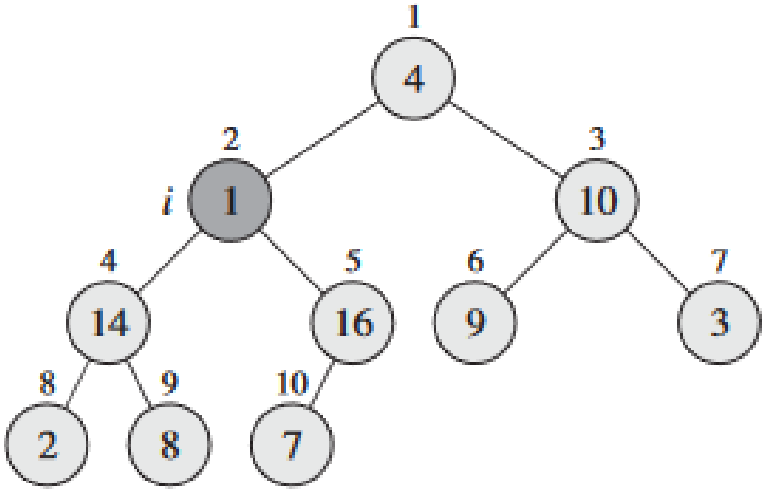
Building a Heap from a given array

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```



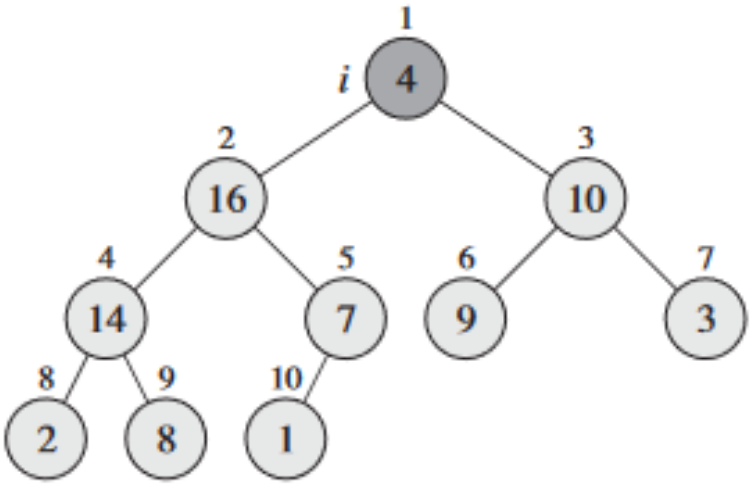
Building a Heap from a given array

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```



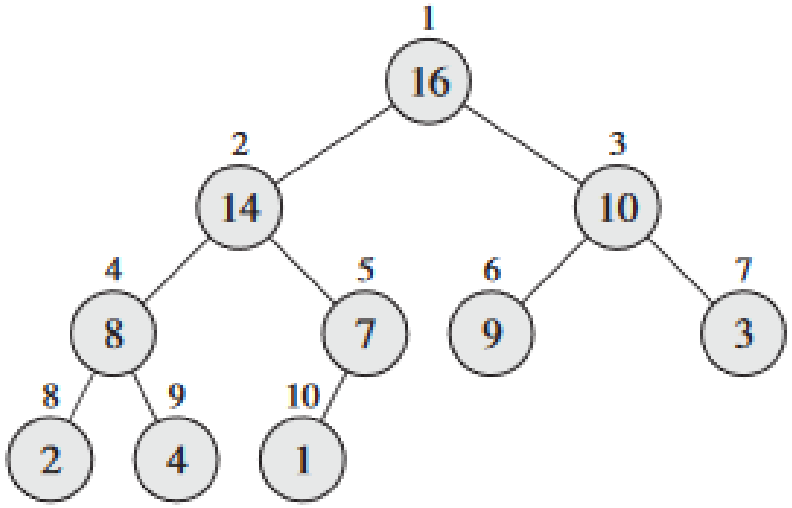
Building a Heap from a given array

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```



Building a Heap from a given array

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```



Building a Heap from a given array

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

10	5	25	40	30	35	20

Time Complexity of BUILD-MAX-HEAP

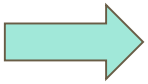
Simple Upper Bound:

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

- The time required by MAX -HEAPIFY when called on a node of height h is O(h).
- Each call to MAX-HEAPIFY costs O(log n) time,
- BUILD-MAX-HEAP makes O(n) such calls. Thus, the running time is O(nlogn).
- This upper bound, though correct, is not asymptotically tight. Why??

Time Complexity of BUILD-MAX-HEAP

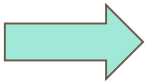
Why is the upper bound not asymptotically tight?



```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

Time Complexity of BUILD-MAX-HEAP

Why is the upper bound not asymptotically tight?



```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

The time for MAX-HEAPIFY to run at a node varies with the height of the node in the tree.

Time Complexity of BUILD-MAX-HEAP

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) .$$

There are at most

$$\left\lceil \frac{n}{2^{h+1}} \right\rceil$$

nodes of **height h** in any **n-element** heap.

Substituting x by 1/2 in

$$\sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2}$$

for $|x| < 1$.

we get,

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2 .$$

Time Complexity of BUILD-MAX-HEAP

So the tighter upper bound
time complexity :

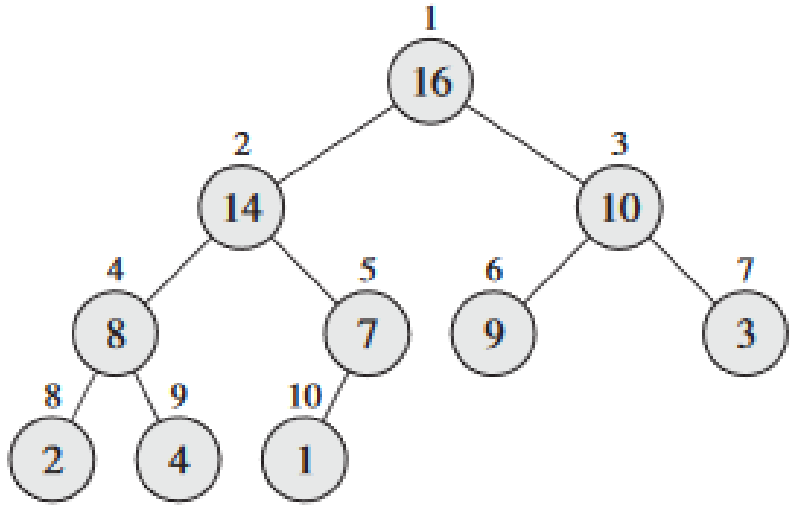
$$O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right)$$
$$= O(n) .$$

To build a max-heap from an unordered array, it takes linear time

HEAP: Insertion Deletion

Insertion: (O(nlogn)or O(n))

1. Insert in the leaf node.
2. Increase heap size
3. Heapify the whole tree (BUILD-MAX-HEAP).



Deletion: (O(nlogn) or O(n))

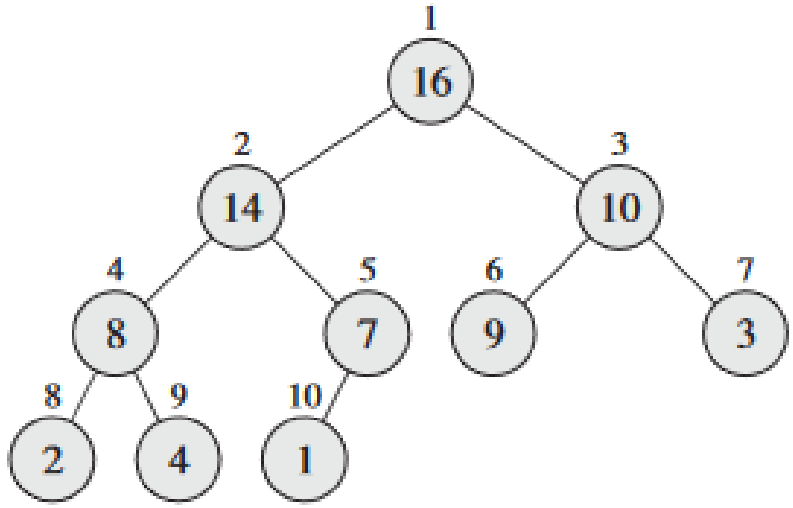
1. Swap the node to be deleted with the last leaf node.
2. Remove the last leaf node.
3. Heapify the whole tree (BUILD-MAX-HEAP).

HEAP: Insertion

Insertion (O(logn)):

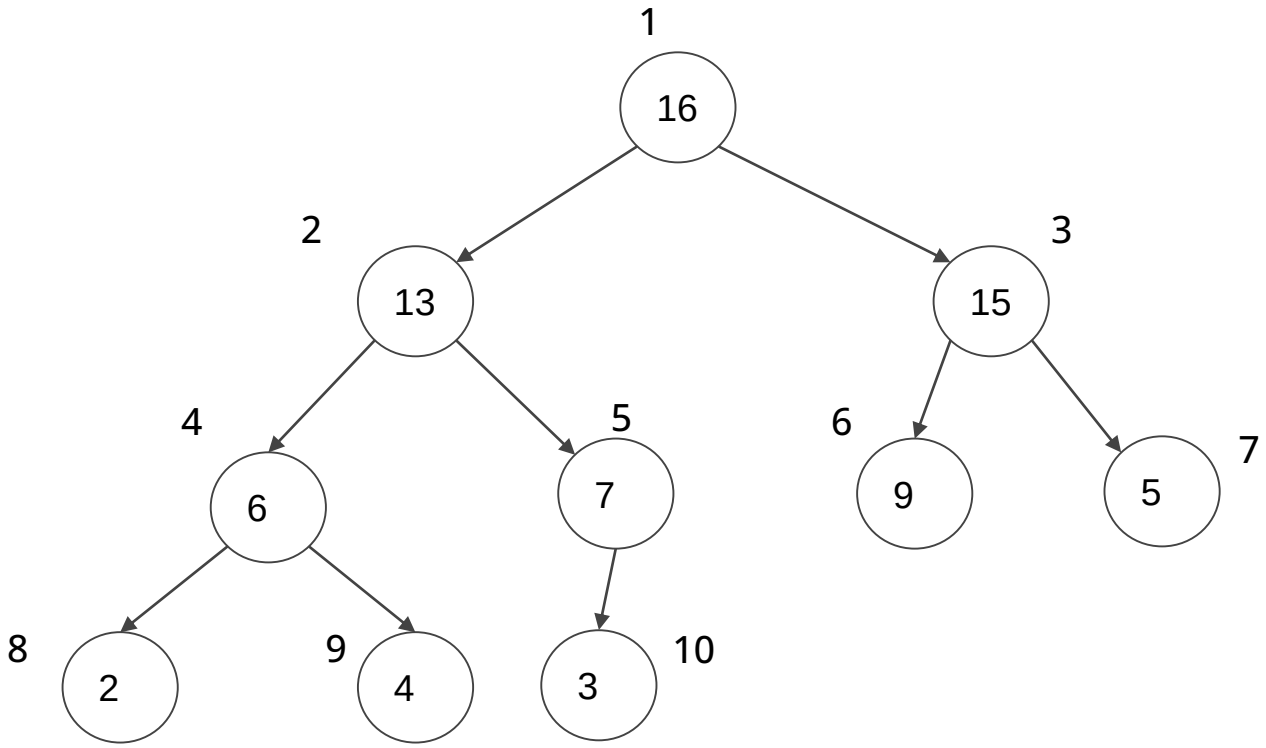
1. Insert in the leaf node and Increase heap size
2. Maintain heap property by exchanging with the parent nodes(ancestors).

```
while i > 1 and A[PARENT(i)] < A[i]
    exchange A[i] with A[PARENT(i)]
    i = PARENT(i)
```



HEAP: Insertion (logn)

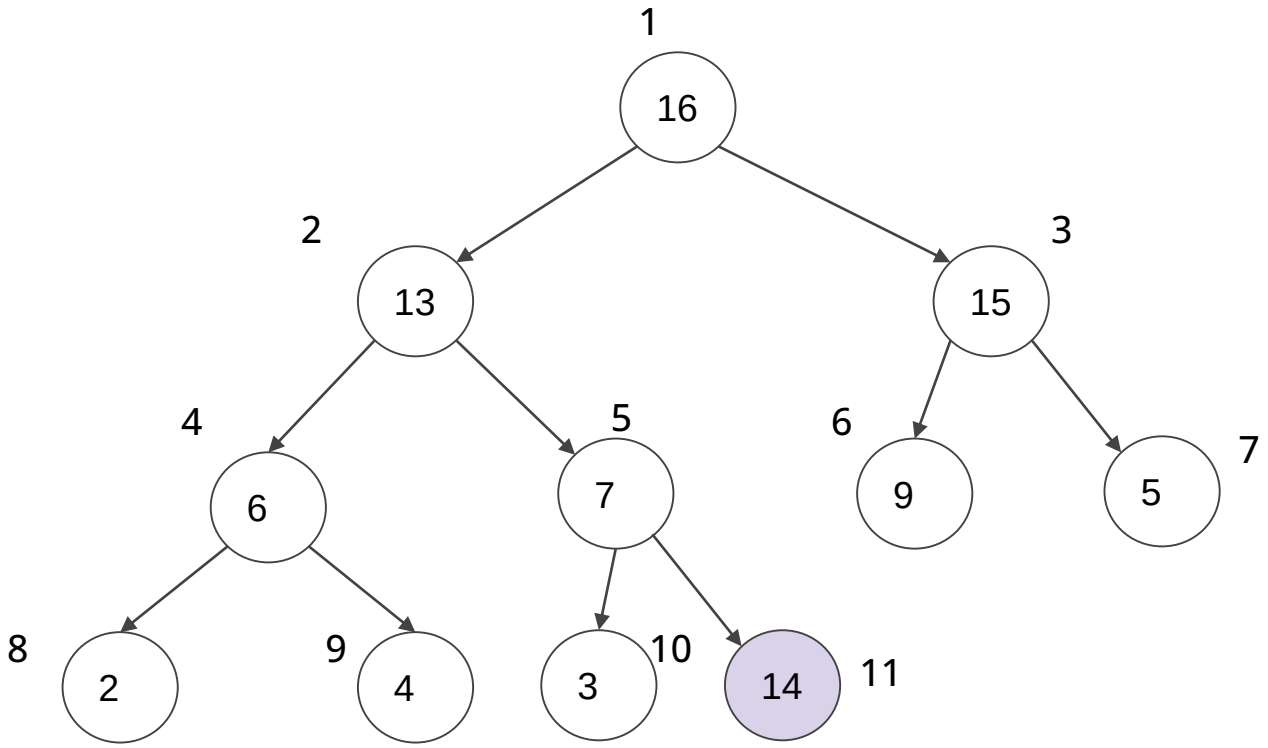
INSERT : 14



HEAP: Insertion (logn)

INSERT : 14

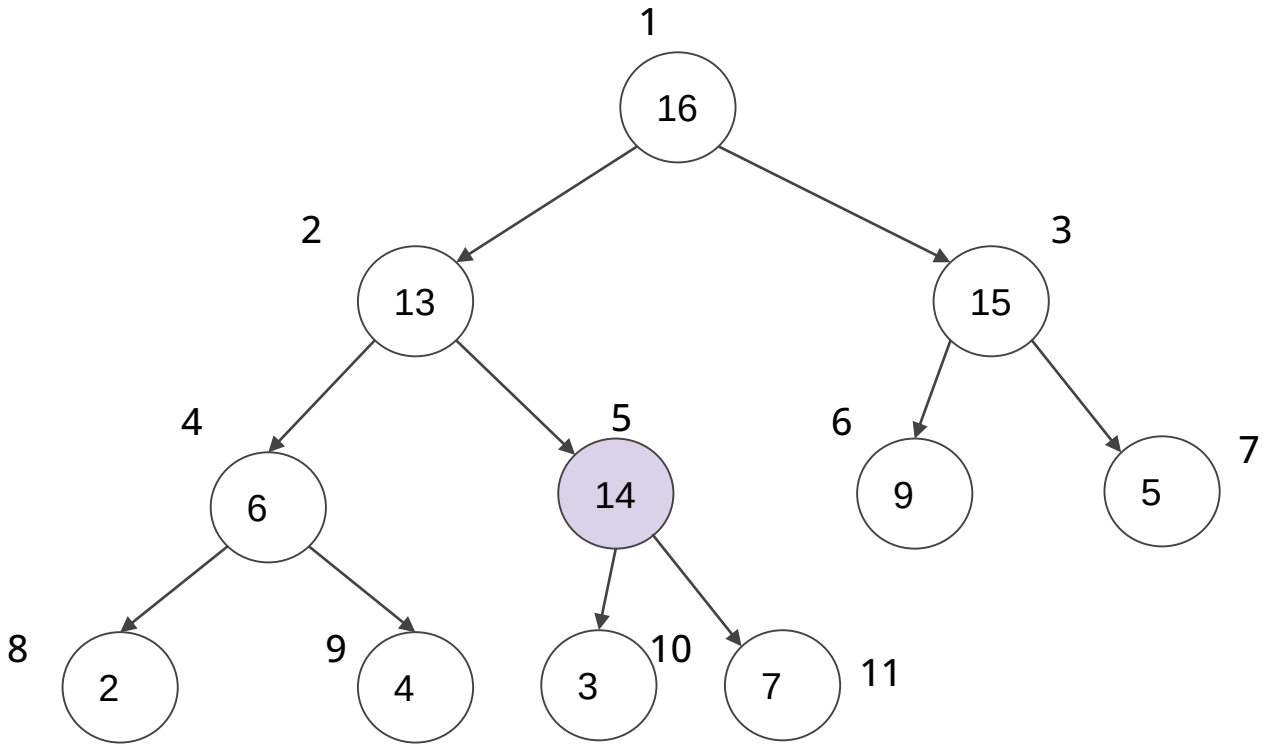
```
while i > 1 and A[PARENT(i)] < A[i]
    exchange A[i] with A[PARENT(i)]
    i = PARENT(i)
```



HEAP: Insertion (logn)

INSERT : 14

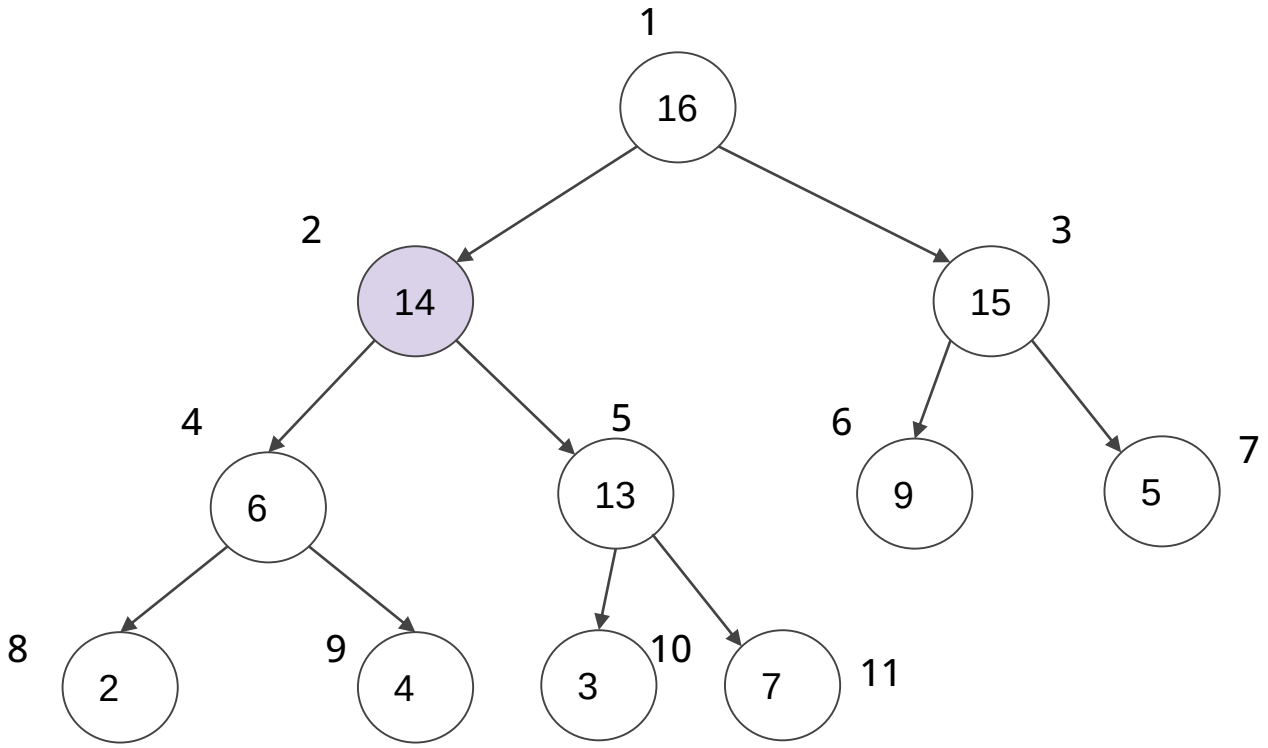
```
while i > 1 and A[PARENT(i)] < A[i]
    exchange A[i] with A[PARENT(i)]
    i = PARENT(i)
```



HEAP: Insertion (logn)

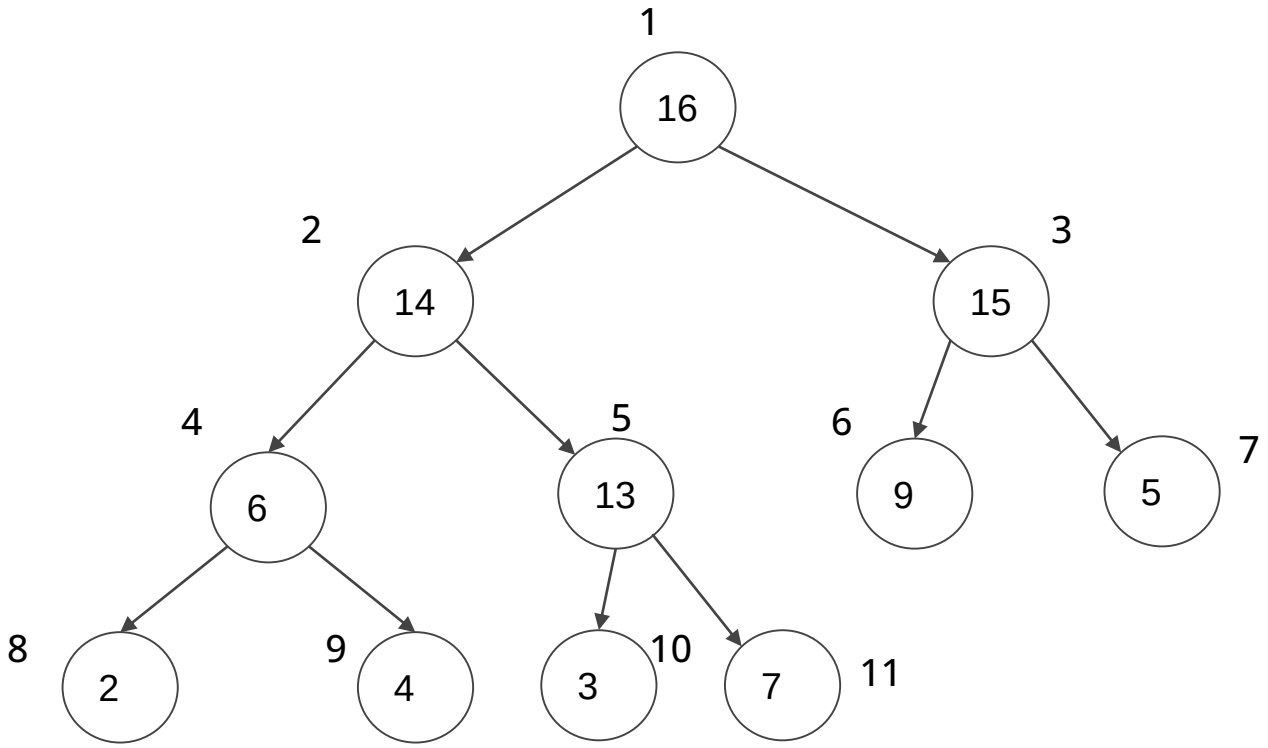
INSERT : 14

```
while i > 1 and A[PARENT(i)] < A[i]
    exchange A[i] with A[PARENT(i)]
    i = PARENT(i)
```



HEAP: Insertion (logn)

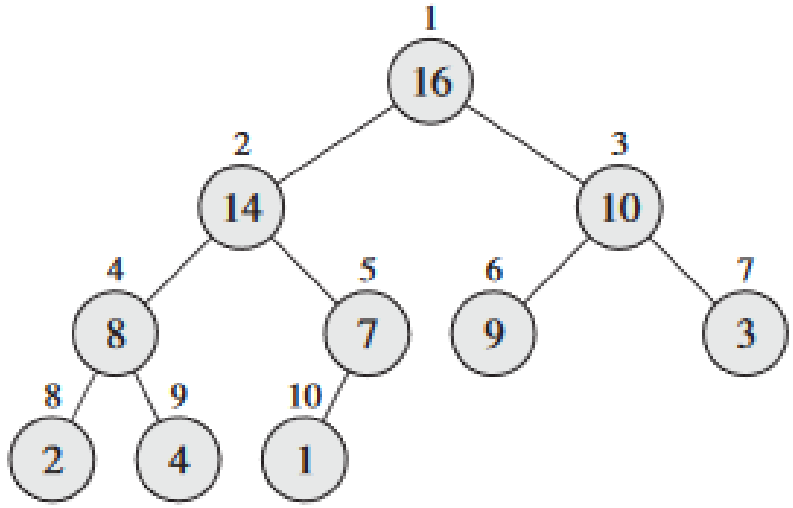
After INSERTION



HEAP: Deletion

Deletion (O(logn)):

1. Swap the node to be deleted with the last leaf node.
2. Remove the last leaf node.
3. Maintain heap property



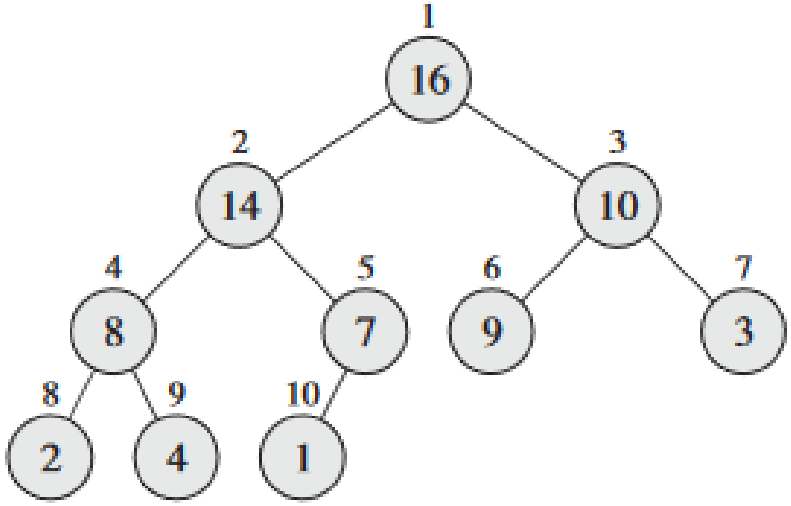
HEAP: Deletion

Deletion (O(logn)):

3. Maintain heap property

```
HEAP-DELETE(A, i)
    if A[i] > A[A.heap-size]
        swap(A[i], A[A.heap-size])
        A.heap-size = A.heap-size - 1
        MAX-HEAPIFY(A, i)
    else
        swap(A[i], A[A.heap-size])
        A.heap-size = A.heap-size - 1
```

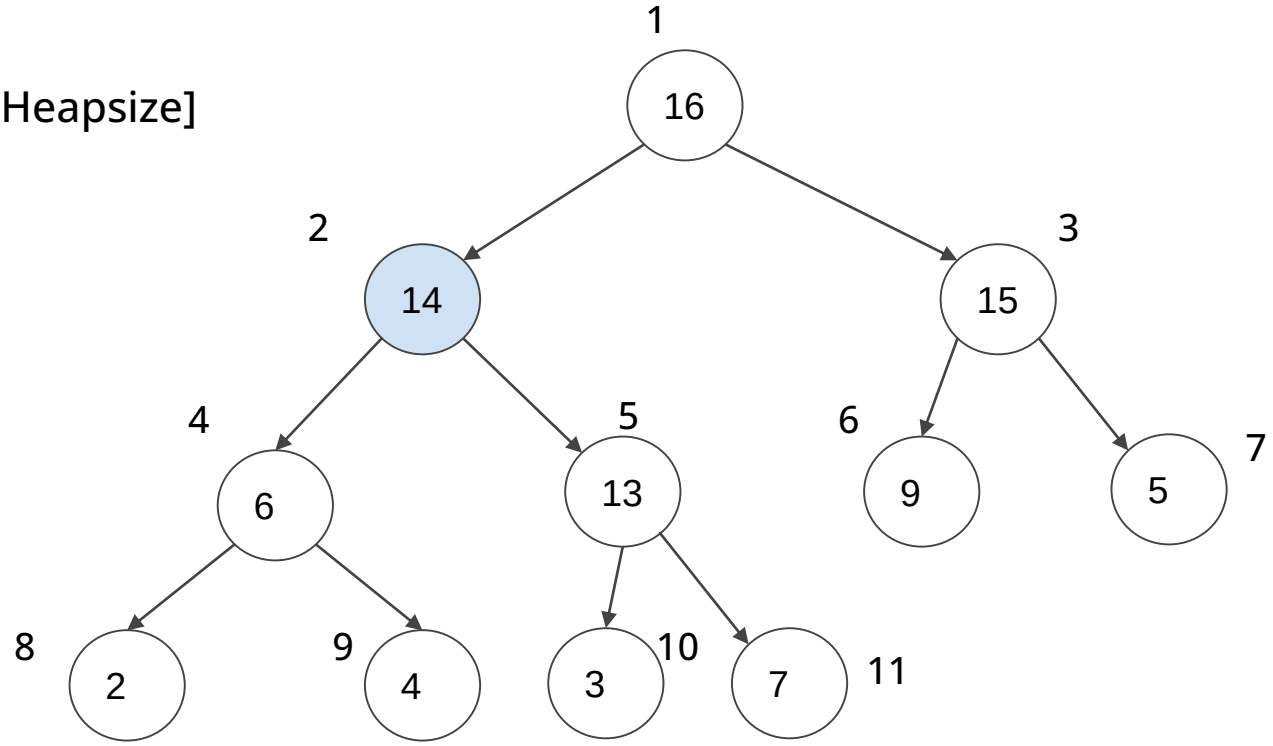
```
while i > 1 and A[PARENT(i)] < A[i]
    exchange A[i] with A[PARENT(i)]
    i = PARENT(i)
```



HEAP: Deletion (logn)

Delete 14 ,i=2
HEAP-DELETE(A, 2)

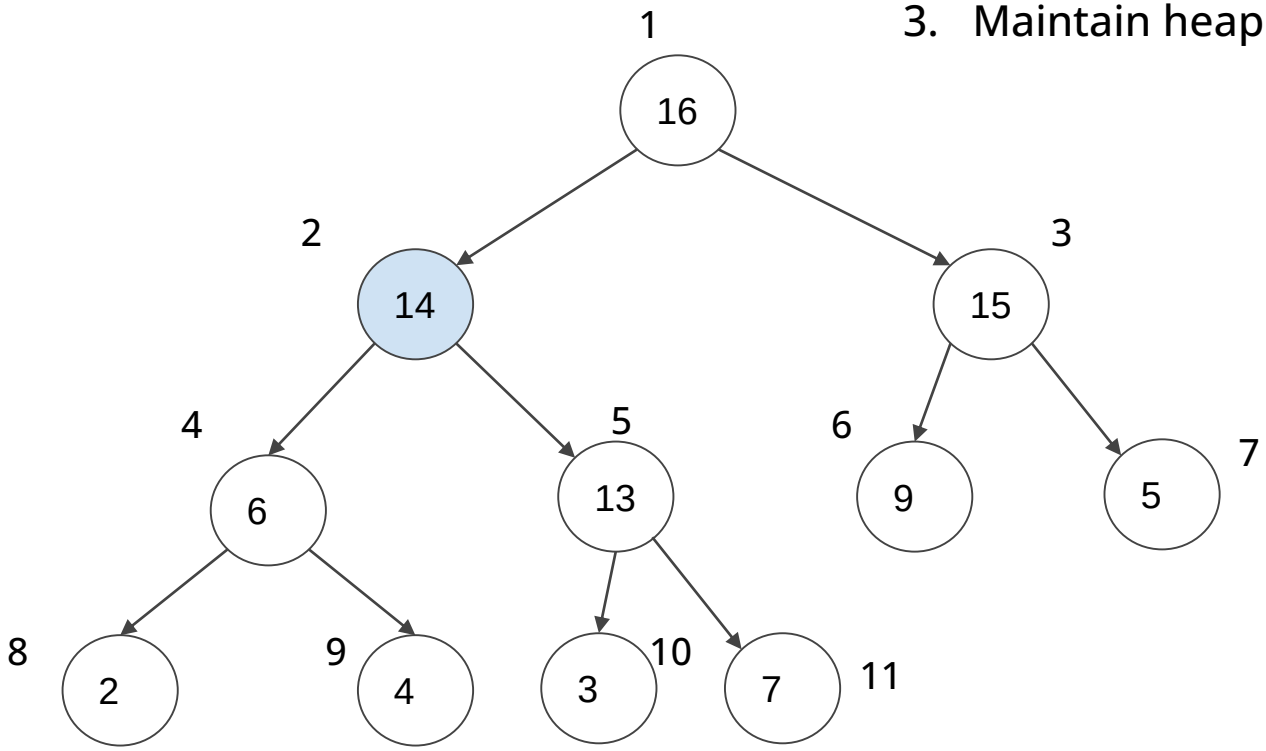
Here,
A[i] > A[Heapsize]



HEAP: Deletion (logn)

Delete 14 ,i=2
HEAP-DELETE(A, 2)

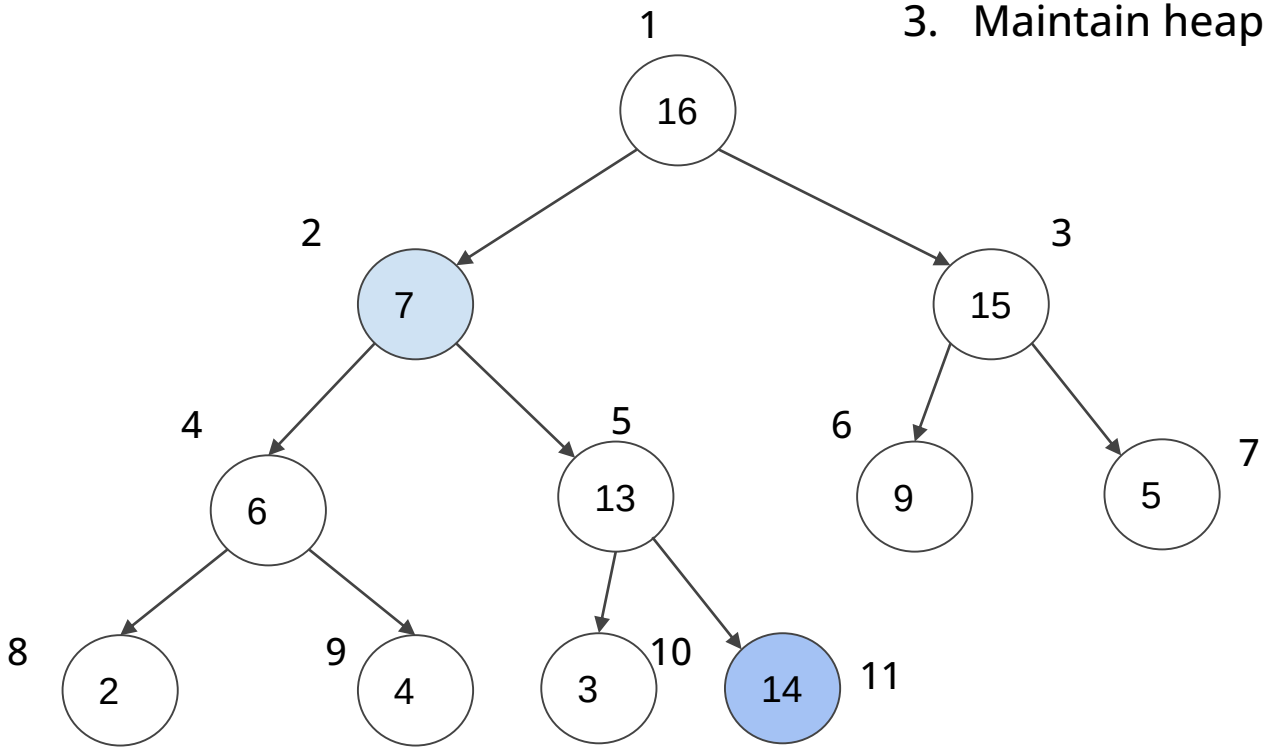
1. **Swap the node to be deleted with the last leaf node.**
2. Remove the last leaf node.
3. Maintain heap property



HEAP: Deletion (logn)

Delete 14 ,i=2
HEAP-DELETE(A, 2)

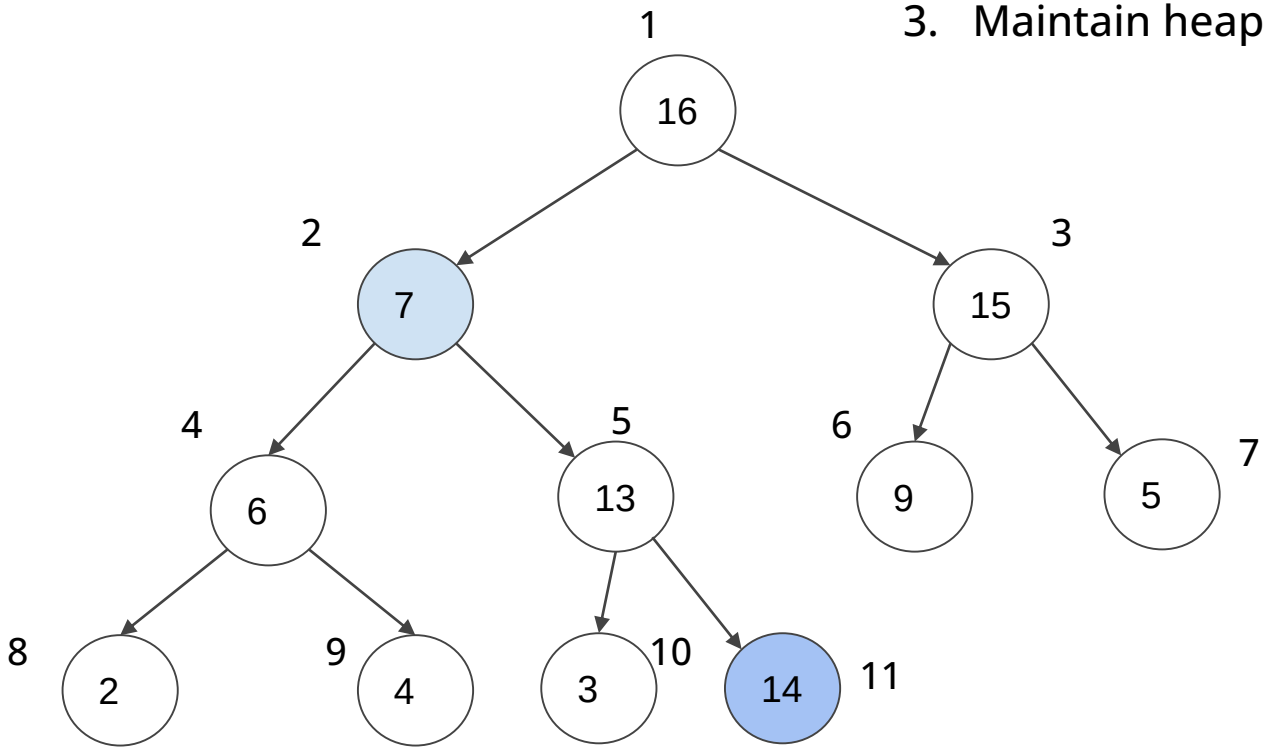
1. **Swap the node to be deleted with the last leaf node.**
2. Remove the last leaf node.
3. Maintain heap property



HEAP: Deletion (logn)

Delete 14 ,i=2
HEAP-DELETE(A, 2)

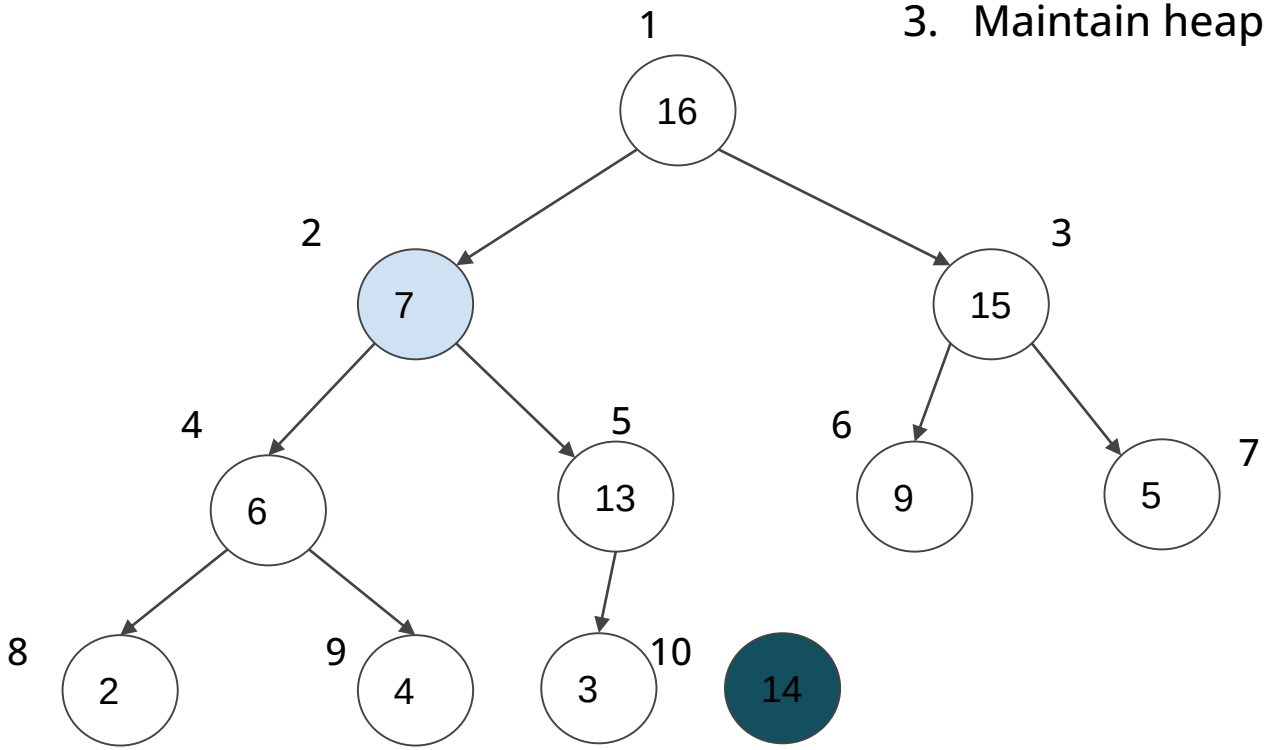
1. Swap the node to be deleted with the last leaf node.
2. **Remove the last leaf node.**
3. Maintain heap property



HEAP: Deletion (logn)

Delete 14 ,i=2
HEAP-DELETE(A, 2)

1. Swap the node to be deleted with the last leaf node.
2. **Remove the last leaf node.**
3. Maintain heap property

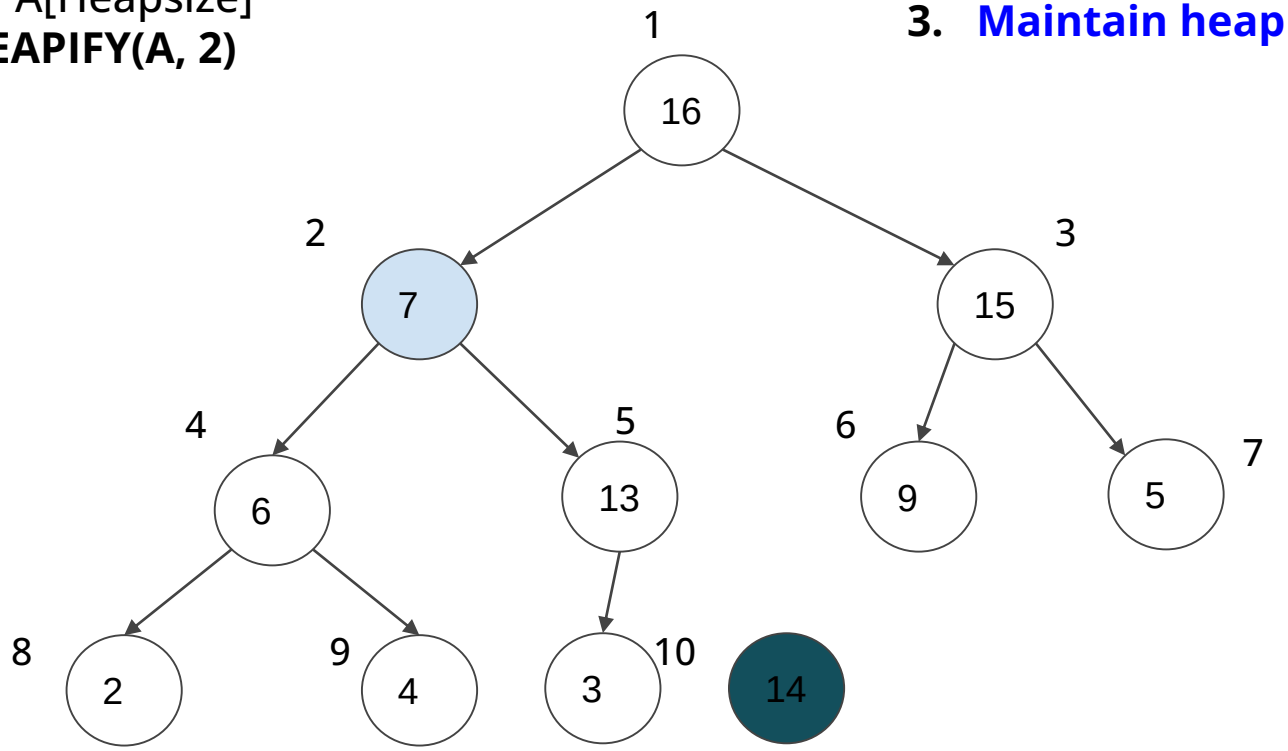


HEAP: Deletion (logn)

HEAP-DELETE(A, 2)

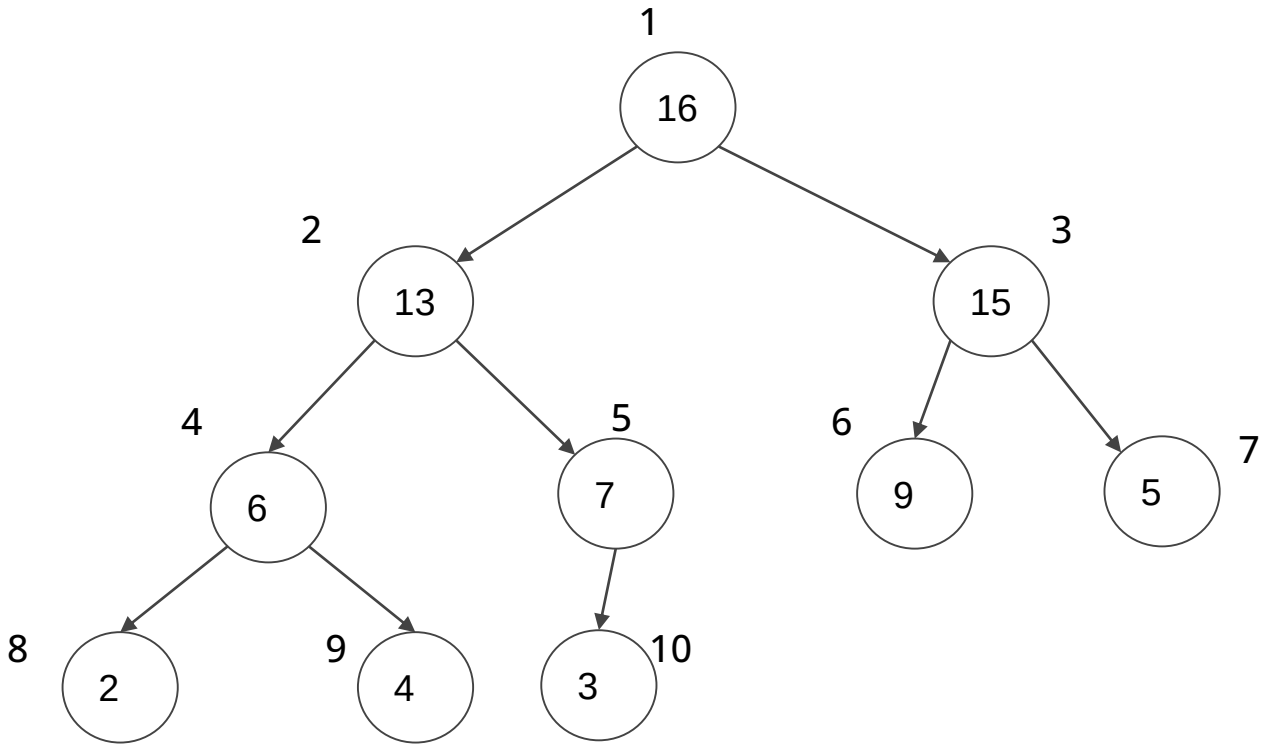
As A[i] > A[Heapsize]
MAX-HEAPIFY(A, 2)

1. Swap the node to be deleted with the last leaf node.
2. Remove the last leaf node.
3. **Maintain heap property**



HEAP: Deletion (logn)

After Deleting 14



HEAP SORT

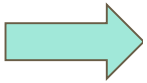
```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

Time Complexity? O(n logn)

HEAP SORT

Let's take an array and apply heap sort to it.

10	5	25	40	30	35	20



```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

Priority Queue

Priority queues come in two forms:
max-priority queues and ***min-priority queues.***

A priority queue is a data structure for maintaining a set S of elements, each with an associated value called a key.

Key = priority value

Elements with higher priority values are retrieved before elements with lower priority values. (max-priority queue)

Priority Queue

Operations of Max Priority Queue:

INSERT(S, x): inserts the element x into the set S.

MAXIMUM(S): returns the element of S with the largest key.

EXTRACT-MAX(S): removes and returns the element of S with the largest key

INCREASE-KEY (S, x, k) increases the value of element x's key to the new value k, which is assumed to be at least as large as x's current key value.

Priority Queue

Operations of Min Priority Queue:

INSERT(S, x): inserts the element x into the set S.

MINIMUM(S): returns the element of S with the smallest key.

EXTRACT-MIN(S): removes and returns the element of S with the smallest key

DECREASE-KEY (S, x, k): decreases the value of element x's key to the new value k.

Priority Queue

Implementation of **Max Priority Queue** Using Max Heap:

HEAP-MAXIMUM(A): Implements MAXIMUM(S) operation in $\theta(1)$ time.

HEAP-EXTRACT-MAX: implements EXTRACT-MAX(S) and running time is O(logn).

HEAP-INCREASE-KEY: implements the INCREASE-KEY operation and running time is O(logn).

MAX-HEAP-INSERT : implements the INSERT operation and running time for a n element heap is O(logn).

Priority Queue Operations

Implementation of Max Priority Queue Using Max Heap:

HEAP-MAXIMUM (A)
    return A[1]

Priority Queue Operations

Implementation of Max Priority Queue Using Max Heap:

```
HEAP-EXTRACT-MAX(A)
1  if A.heap-size < 1
2        error "heap underflow"
3  max = A[1]
4  A[1] = A[A.heap-size]
5  A.heap-size = A.heap-size − 1
6  MAX-HEAPIFY(A, 1)
7  return max
```

Priority Queue Operations

Implementation of Max Priority Queue Using Max Heap:

```
HEAP-INCREASE-KEY(A, i, key)
1  if key < A[i]
2       error "new key is smaller than current key"
3  A[i] = key
4  while i > 1 and A[PARENT(i)] < A[i]
5       exchange A[i] with A[PARENT(i)]
6       i = PARENT(i)
```

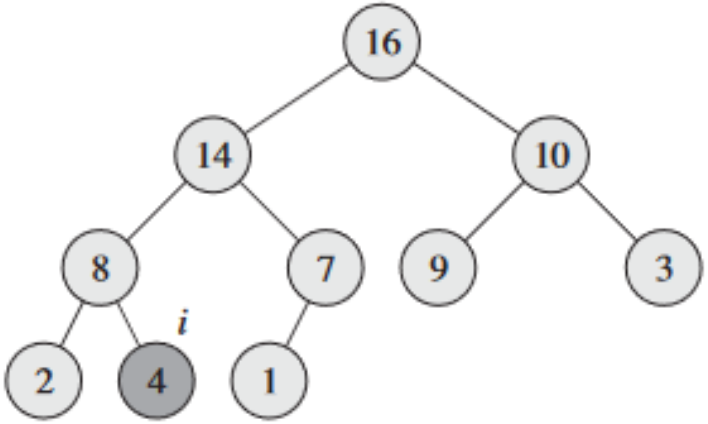
Priority Queue Operations

Implementation of Max Priority Queue Using Max Heap:

```
MAX-HEAP-INSERT(A, key)
1  A.heap-size = A.heap-size + 1
2  A[A.heap-size] = −∞
3  HEAP-INCREASE-KEY(A, A.heap-size, key)
```

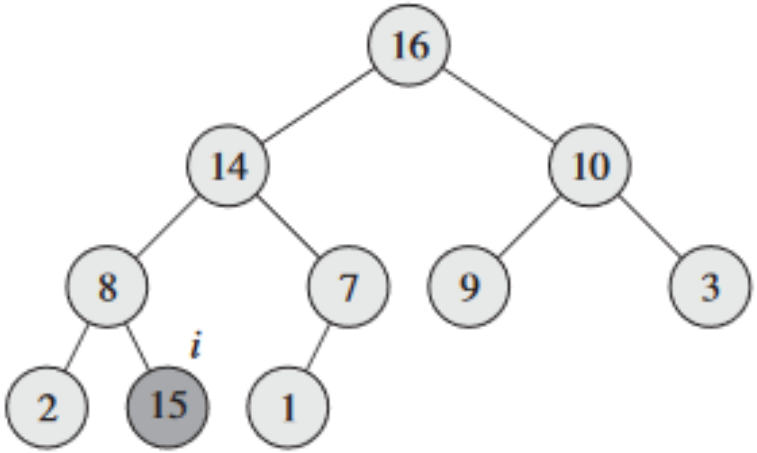
Priority Queue Operations

The operation of HEAP-INCREASE-KEY(A,i,k) :



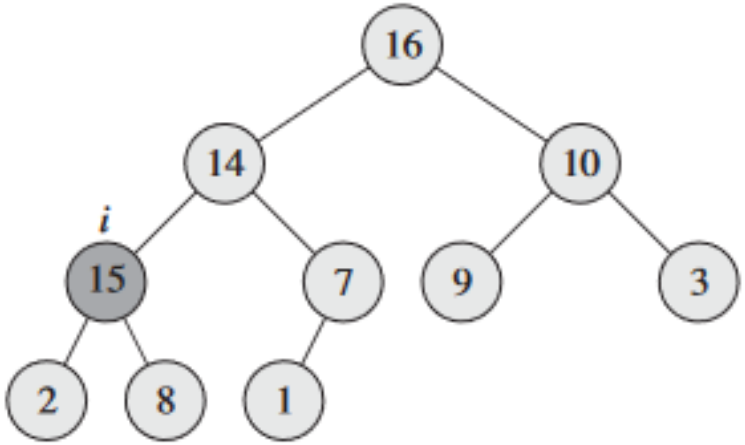
Priority Queue Operations

The operation of HEAP-INCREASE-KEY:



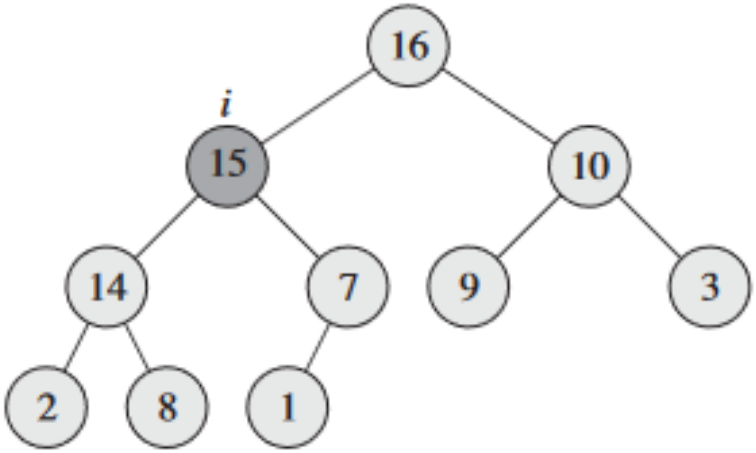
Priority Queue Operations

The operation of HEAP-INCREASE-KEY:



Priority Queue Operations

The operation of HEAP-INCREASE-KEY:



Priority Queue Operations

Illustrate the operation of HEAP-EXTRACT-MAX:

A={15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1}

Heap Exercises

- What are the minimum and maximum numbers of elements in a heap of height h?
- Show that an n-element heap has height ⌊log n⌋.
- Where in a max-heap might the smallest element reside, assuming that all elements are distinct?
- Show that there are at most ⌈n/2^(h+1)⌉ nodes of height h in any n-element heap.
- What is the running time of HEAPSORT on an array A of length n that is already sorted in increasing order? What about decreasing order?


thank
you

TRIE

Prepared By

Lec Swapnil Biswas

TRIE

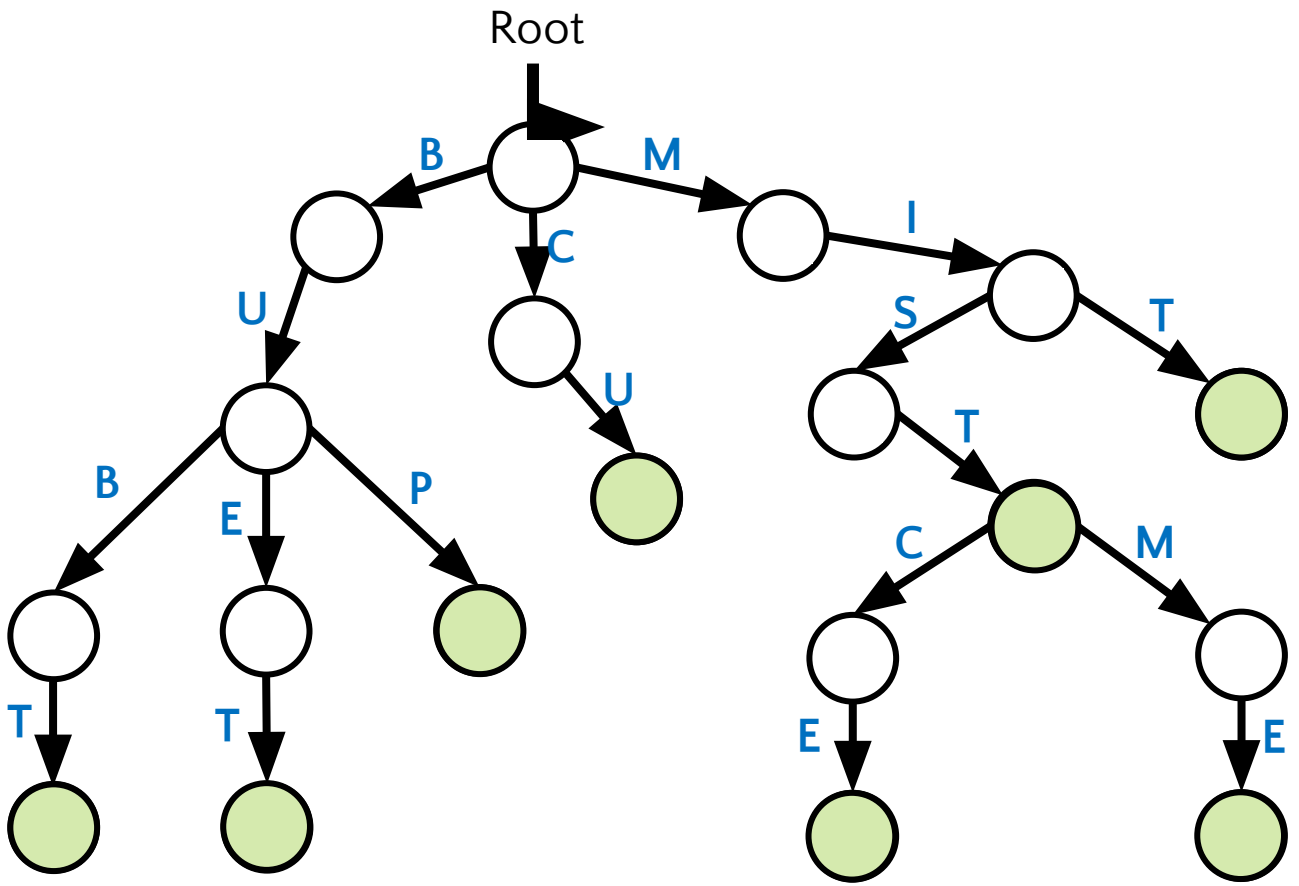
- A tree based data structure (k-ary tree)
- Root is an empty node.
- (k=26) Each node will have 26 children (Each child represents a alphabetic letter)
- Implemented by linked data structure
- It allows for very fast searching and insertion operations
- The word TRIE comes from the word Re**trie**val
- It refers to the quick retrieval of strings
- Used for storing strings, string matching, lexicographical sorting etc.

WHY TRIE?

- Consider a database of strings
- Number of strings in the database is ***n***
- Now what is the complexity to find a given string ***x*** whether ***x*** exists in the database or not
- Ans: O(***n*** x ***m***) where ***m*** is the average length of the strings
- Now if the database is too big, then finding a string from the database will be time consuming
- Goal is to find a string ***x*** without the dependency of ***n***
- TRIE will solve this issue to find a string ***x*** in O(length(***x***)) complexity
- So doesn't matter how long the database is, time complexity of finding a string x will remain length(x)

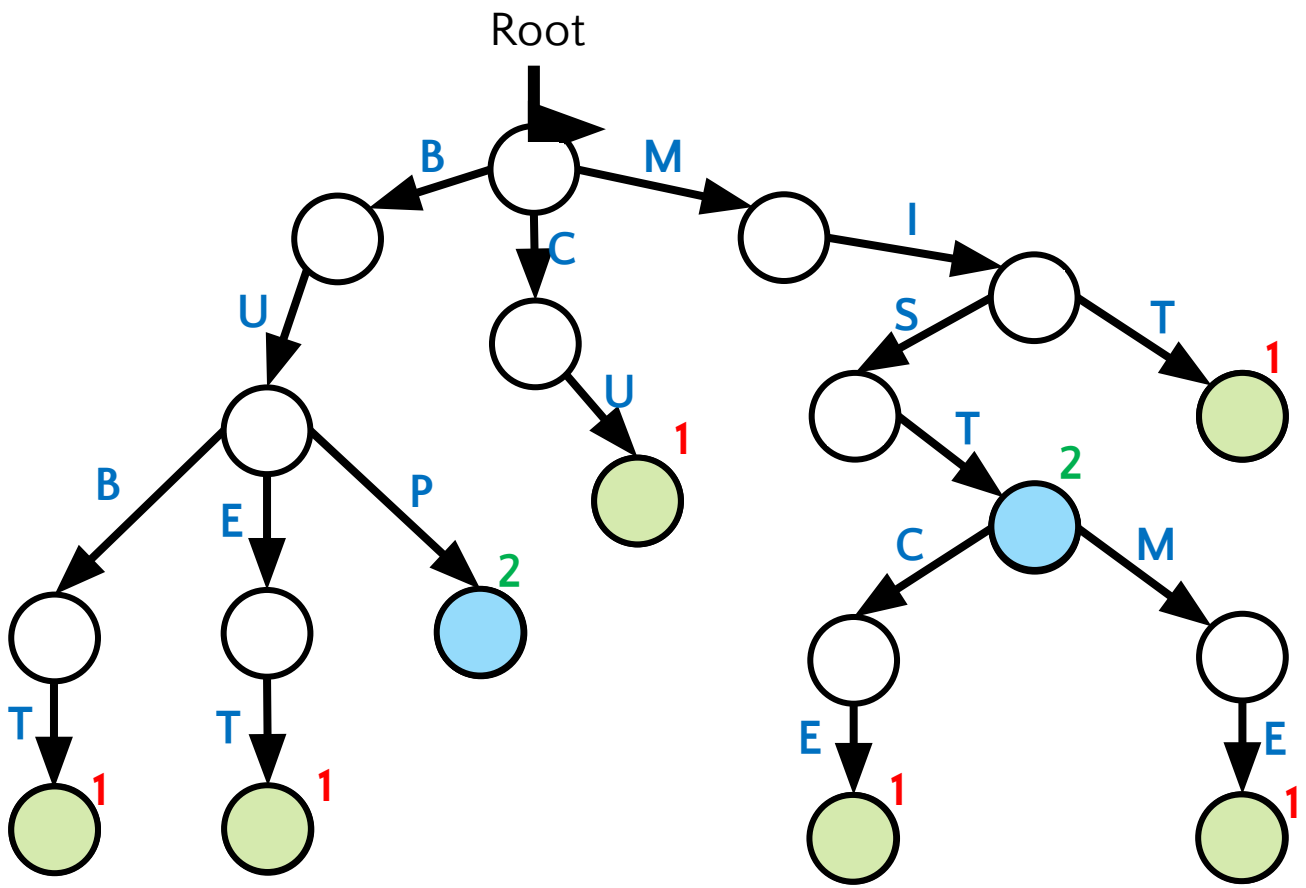
INSERT IN TRIE

- insert("MIT")
- insert("MIST")
- insert("BUET")
- insert("MISTCE")
- insert("BUBT")
- insert("MISTME")
- insert("BUP")
- insert("CU")
- insert("MIST")
- Is it possible to know the frequency of any string in the TRIE?
  - NO
- But keeping a counter variable at each node can address this issue



INSERT IN TRIE (WITH COUNTER)

- insert("MIT")
- insert("MIST")
- insert("BUET")
- insert("MISTCE")
- insert("BUBT")
- insert("MISTME")
- insert("BUP")
- insert("CU")
- insert("MIST")
- insert("BUP")

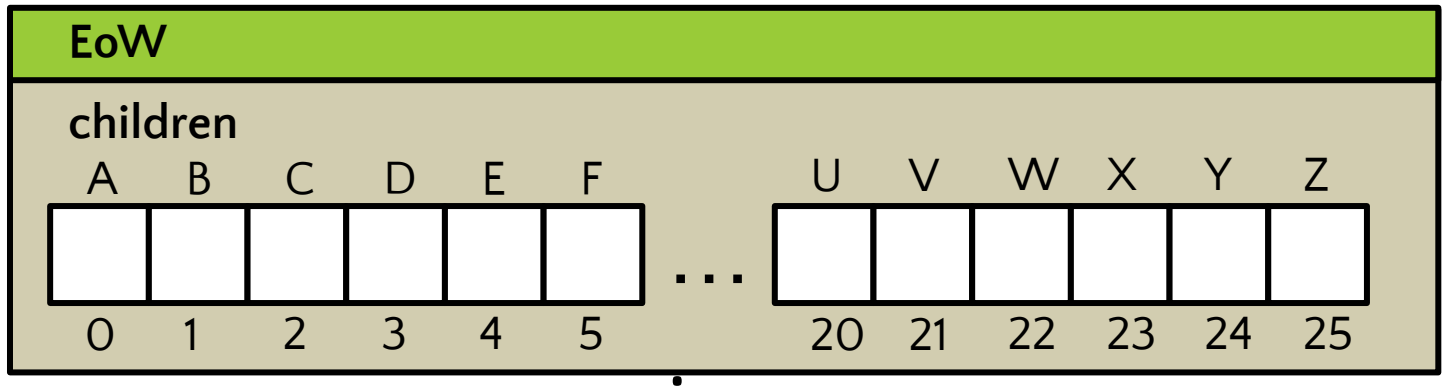


INSERT IN TRIE (WITH COUNTER)

- insert(“MIT”)
- insert(“MIST”)
- insert(“BUET”)
- insert(“MISTCE”)
- insert(“BUBT”)
- insert(“MISTME”)
- insert(“BUP”)
- insert(“CU”)
- insert(“MIST”)
- insert(“BUP”)

NODE REPRESENTATION

```
struct Node{
    int EoW;
    Node  *children[26];
}
```



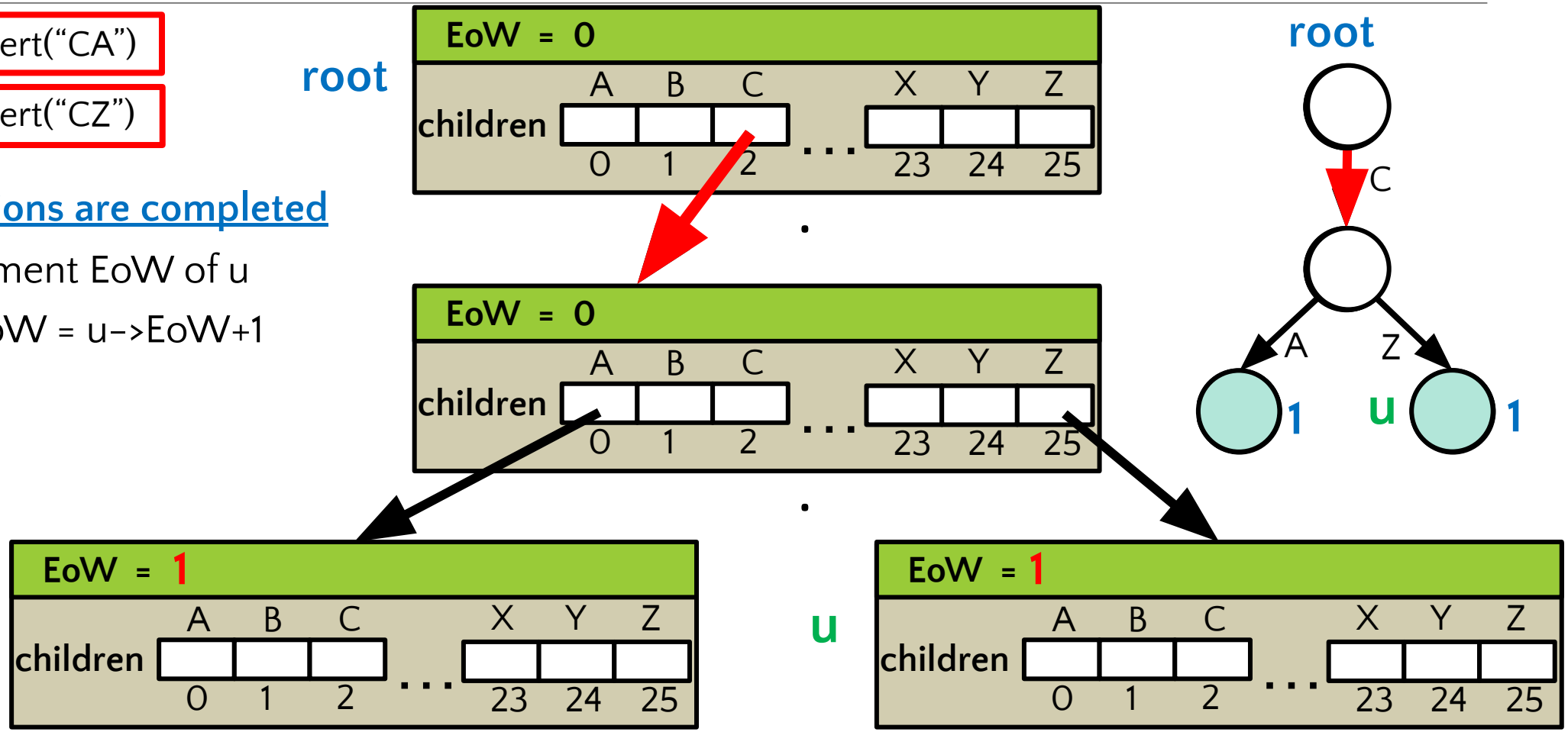
NODE REPRESENTATION

- insert("CA")
- insert("CZ")

Iterations are completed

Increment EoW of u

u->EoW = u->EoW+1



INSERT IN TRIE

```
insert(x)
    Node pointer u ← root
    for k ← 0 to size(x) - 1
        r ← x[k] - 65
        if u->children[r] is NULL
            u->children[r] ← new Node( )
        u ← u->children[r]
    u->EoW ← u->EoW + 1;
```

O(|x|)

Initially pointing *u* at the *root*

Iterates for size(*x*) number of times

r is the relative position of current char

No children condition

Creates new node under children[*r*]

Pushes *u* down for next iteration

Increments u->EoW after completing iteration

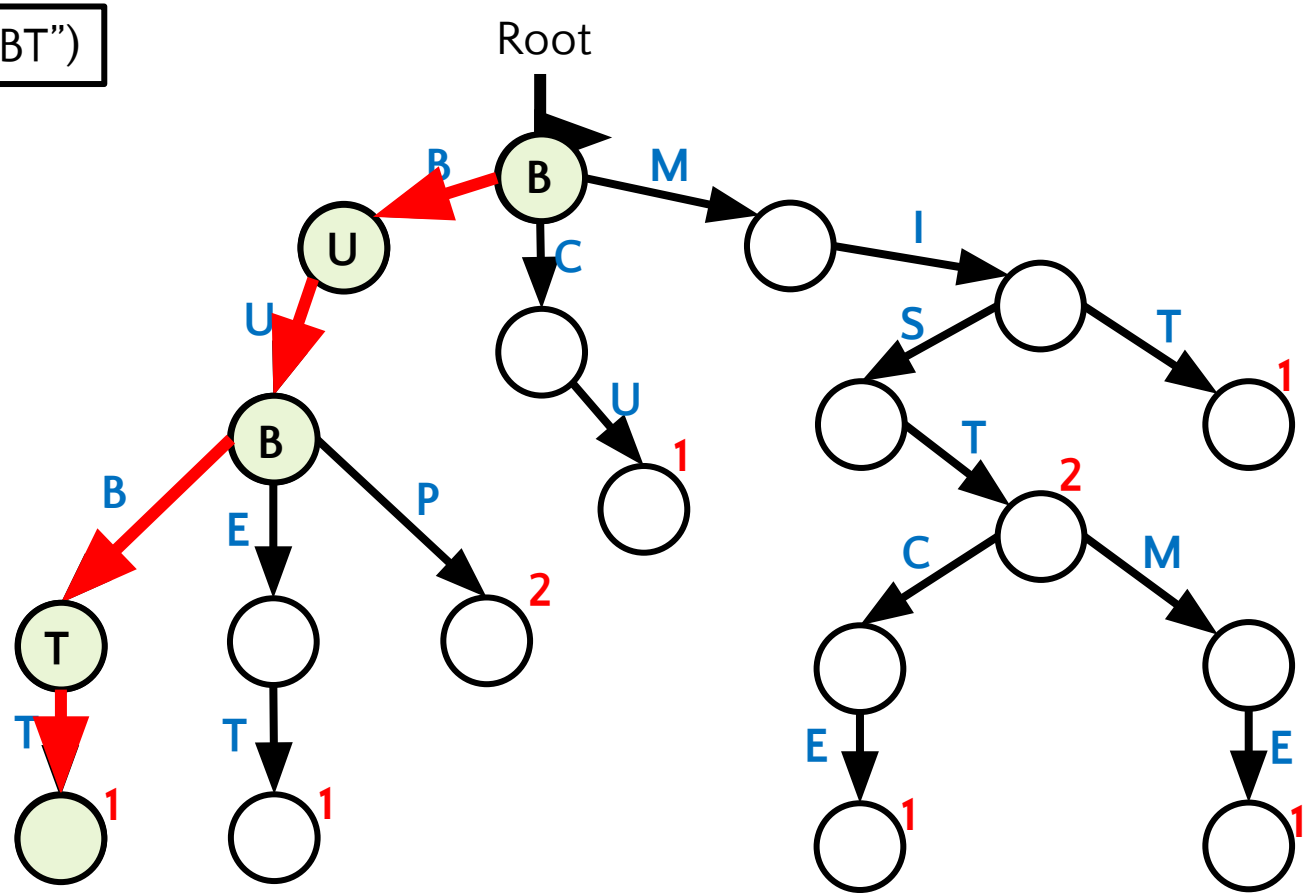
SEARCH IN TRIE

- insert("MIT")
- insert("MIST")
- insert("BUET")
- insert("MISTCE")
- insert("BUBT")
- insert("MISTME")
- insert("BUP")
- insert("CU")
- insert("MIST")
- insert("BUP")

- search("BUBT")

We reach a vertex with counter >0

Means "BUBT" exists



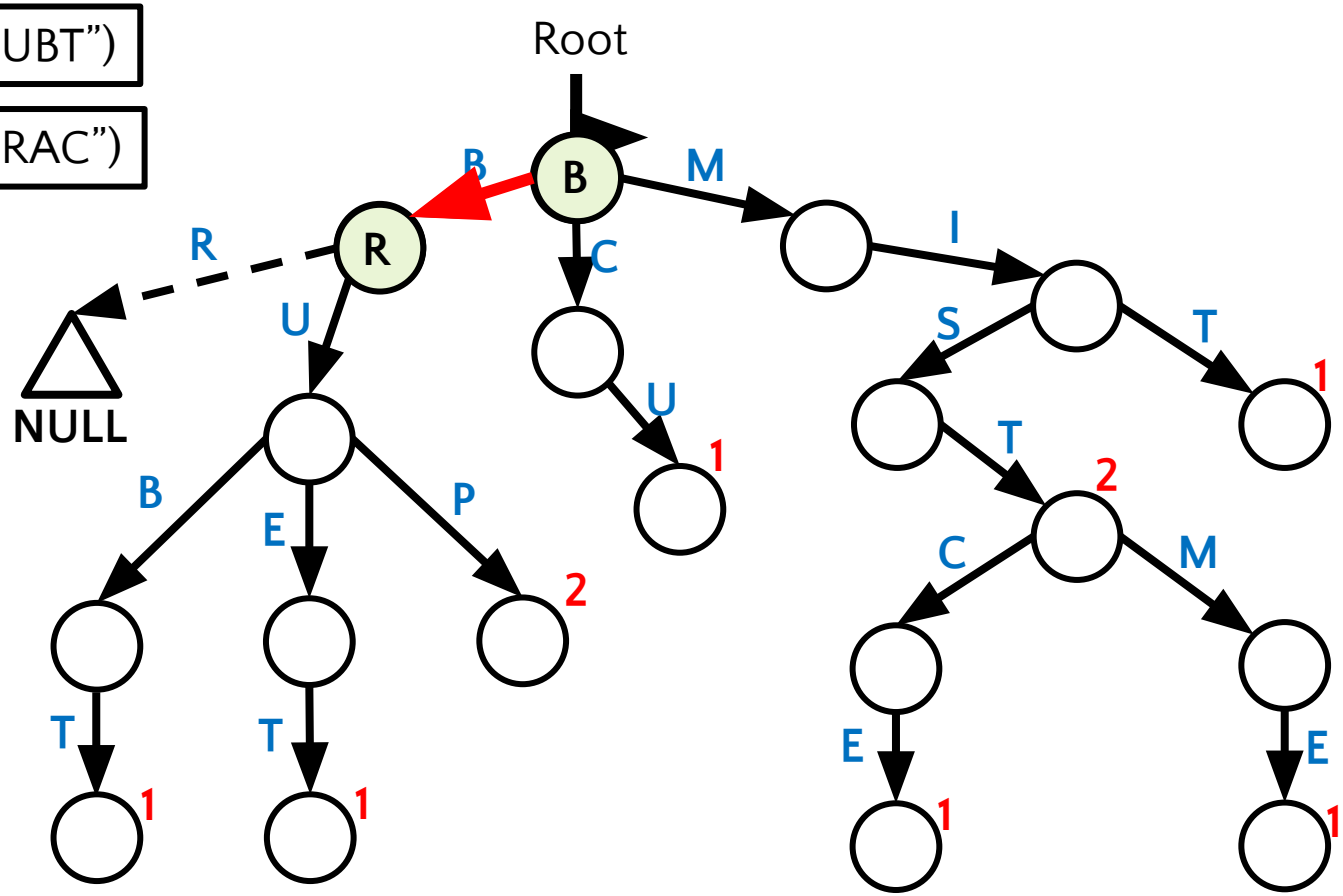
SEARCH IN TRIE

- insert("MIT")
- insert("MIST")
- insert("BUET")
- insert("MISTCE")
- insert("BUBT")
- insert("MISTME")
- insert("BUP")
- insert("CU")
- insert("MIST")
- insert("BUP")

- search("BUBT")
- search("BRAC")

We reach to NULL

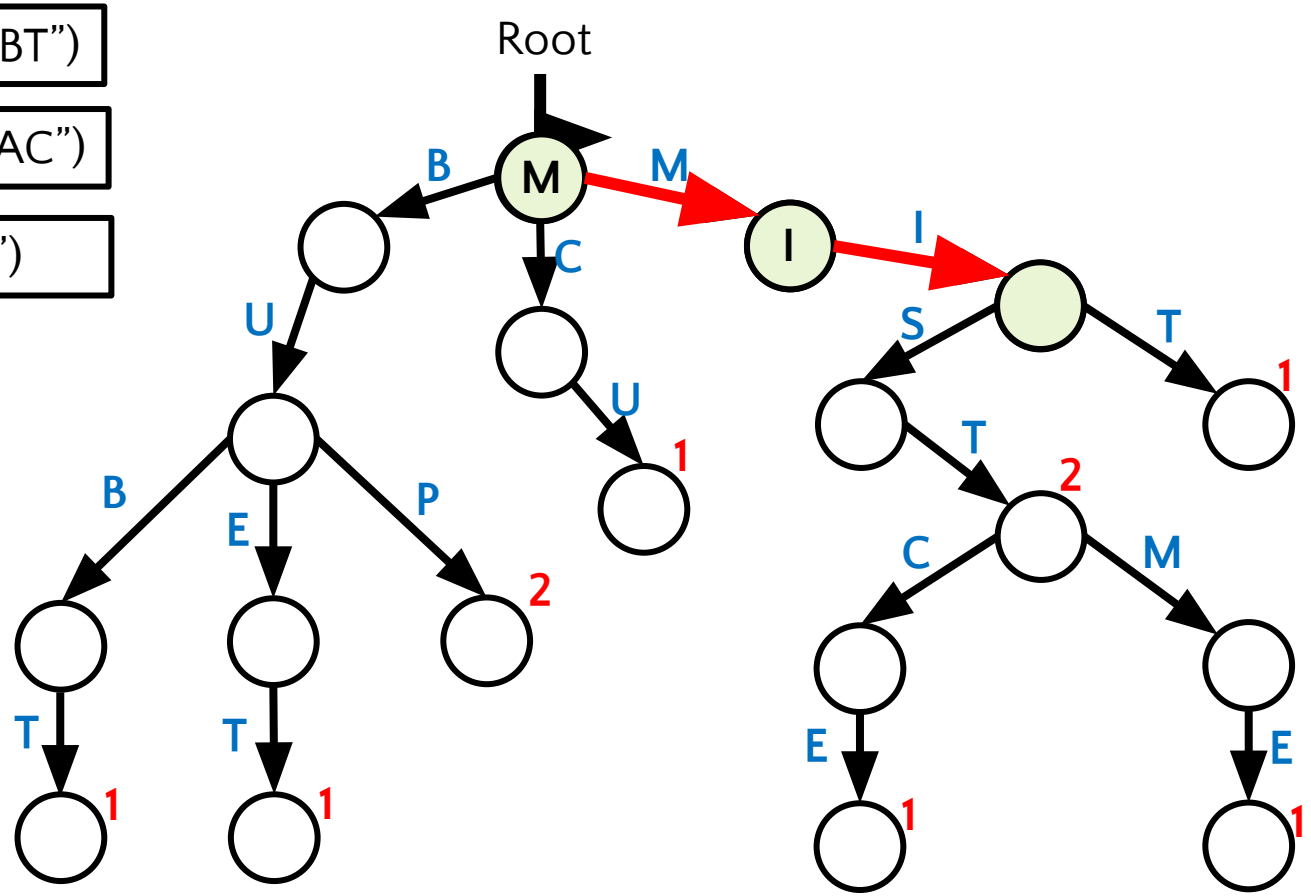
Means "BRAC" doesn't exist



SEARCH IN TRIE

- insert("MIT")
- insert("MIST")
- insert("BUET")
- insert("MISTCE")
- insert("BUBT")
- insert("MISTME")
- insert("BUP")
- insert("CU")
- insert("MIST")
- insert("BUP")

- search("BUBT")
- search("BRAC")
- search("MI")



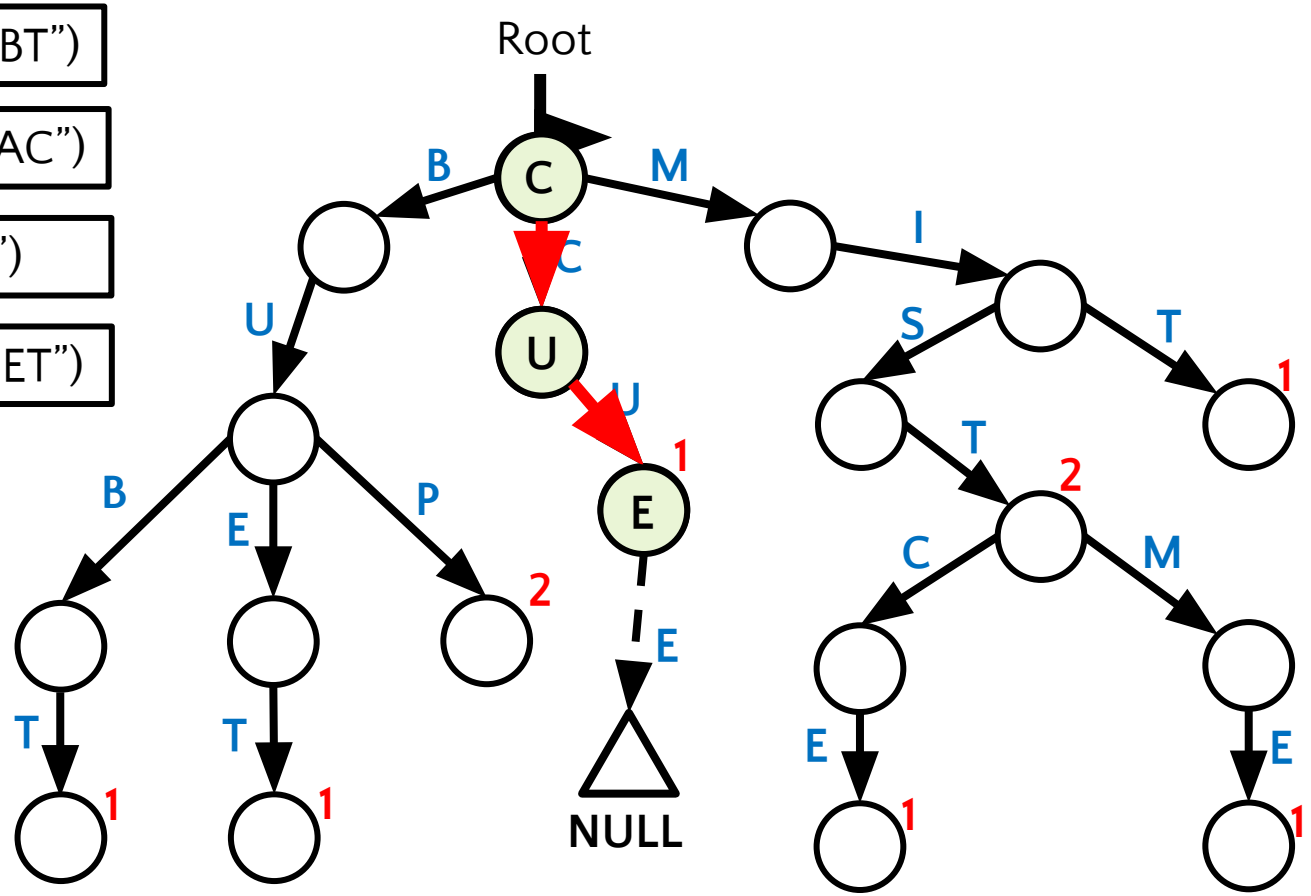
We can't reach a node with counter=0

Means "MI" doesn't exist

SEARCH IN TRIE

- insert("MIT")
- insert("MIST")
- insert("BUET")
- insert("MISTCE")
- insert("BUBT")
- insert("MISTME")
- insert("BUP")
- insert("CU")
- insert("MIST")
- insert("BUP")

- search("BUBT")
- search("BRAC")
- search("MI")
- search("CUET")

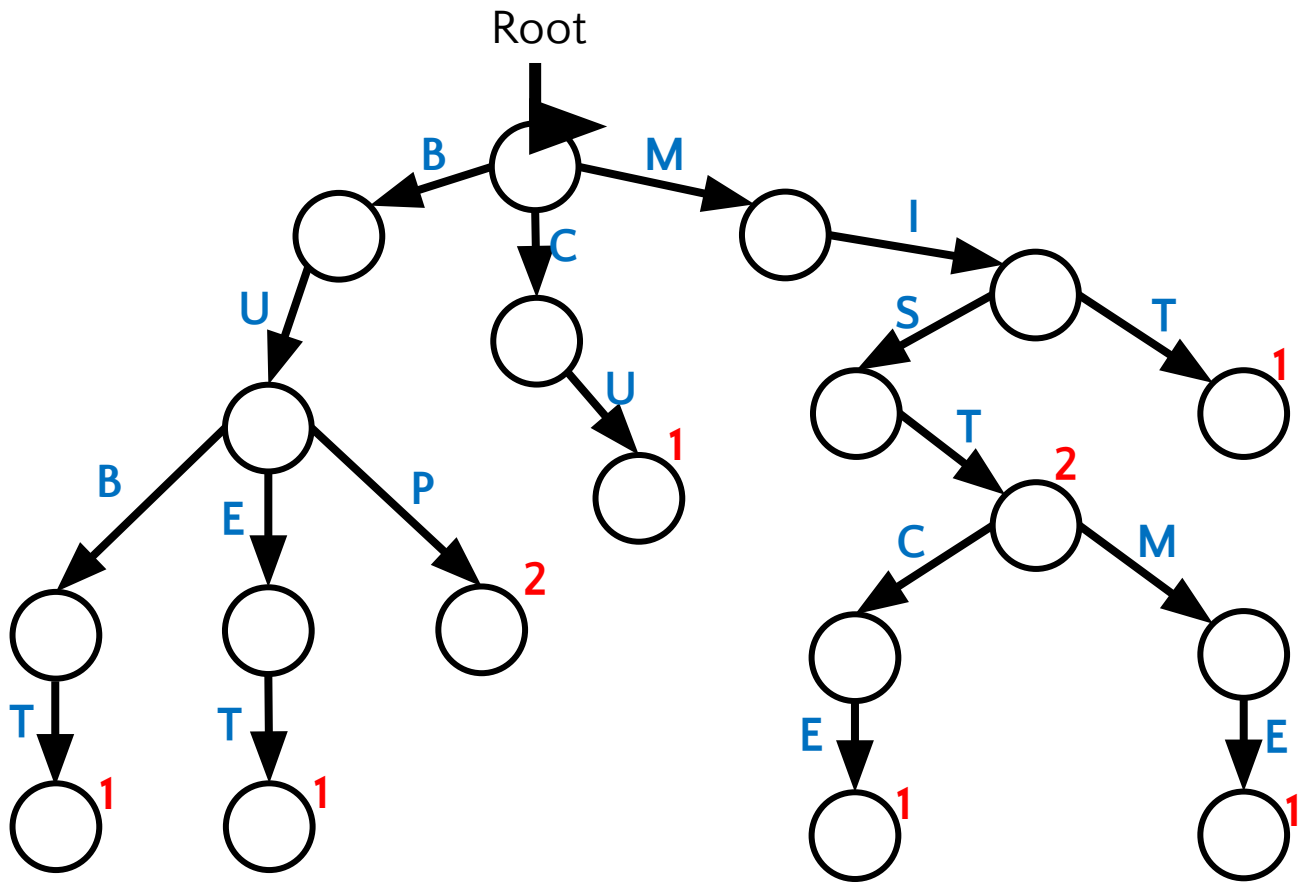


We reach to NULL

Means "CUET" doesn't exist

SEARCH IN TRIE

- We don't find a string in TRIE if
  - The search ends to a NULL
  - The search ends to a node with counter = 0 (Not the end of a word)



METHODS

- void insert(string x)
- int search(string x)
- bool delete(string x)
- void lexSort( )

RELATIVE POSITION OF A CHARACTER

- Consider the strings can only contain uppercase letters
- The relative position of a character is obtained by subtracting 65 from it

Character	Relative Position	Character	Relative Position	Character	Relative Position
A	0	I	9	R	18
B	1	J	10	S	19
C	2	K	11	T	20
D	3	L	12	U	21
E	4	M	13	V	22
F	5	N	14	W	23
G	6	O	15	X	24
H	7	P	16	Y	25
I	8	Q	17		

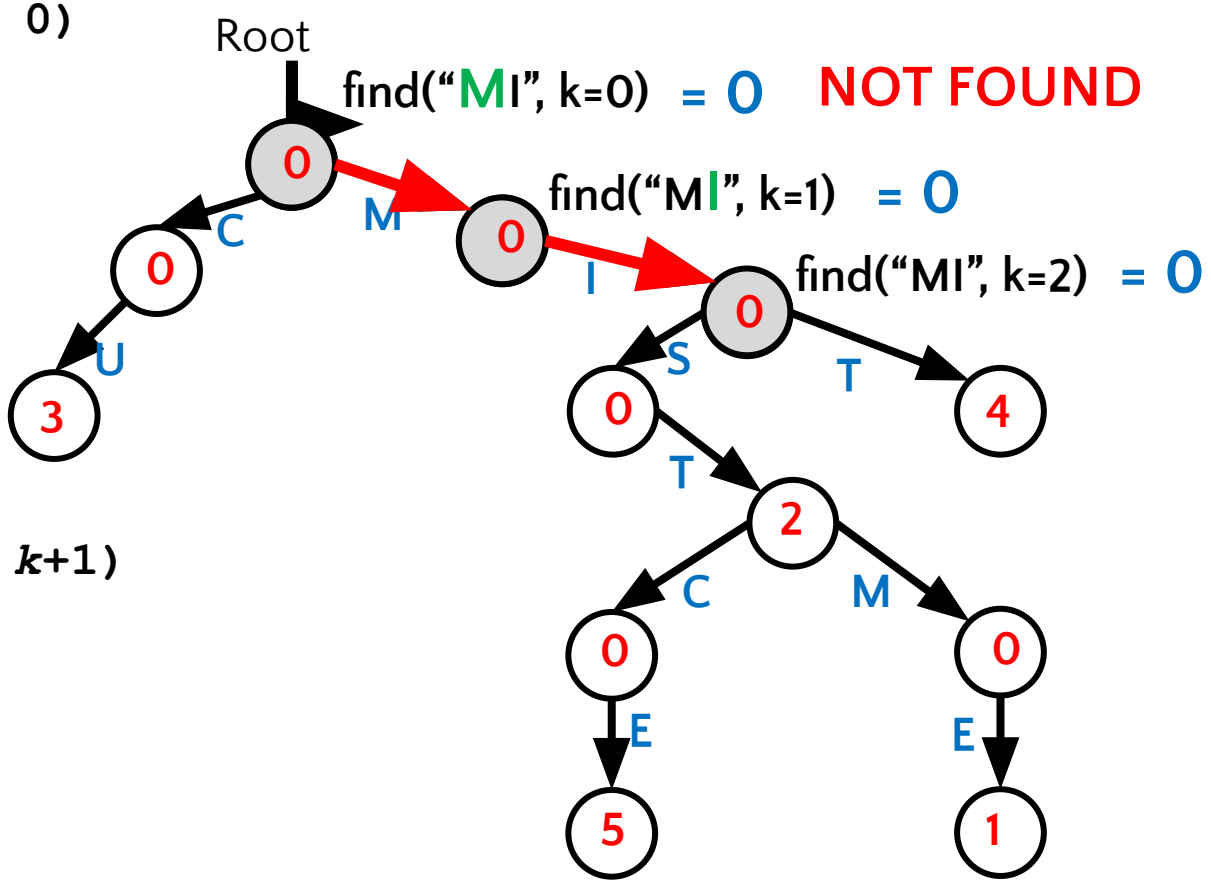
RELATIVE POSITION OF A CHARACTER

```
int relPos(char c){
        int ascii = (int) c;
        return ascii – 65;
}
```

SEARCH IN TRIE

```
find(x, Node pointer cur ← root, k ← 0)
    if cur is NULL
        return 0

    if k equals size(x)
        return cur->EoW

    r ← x[k] - 65
    return find(x, cur->children[r], k+1)
```

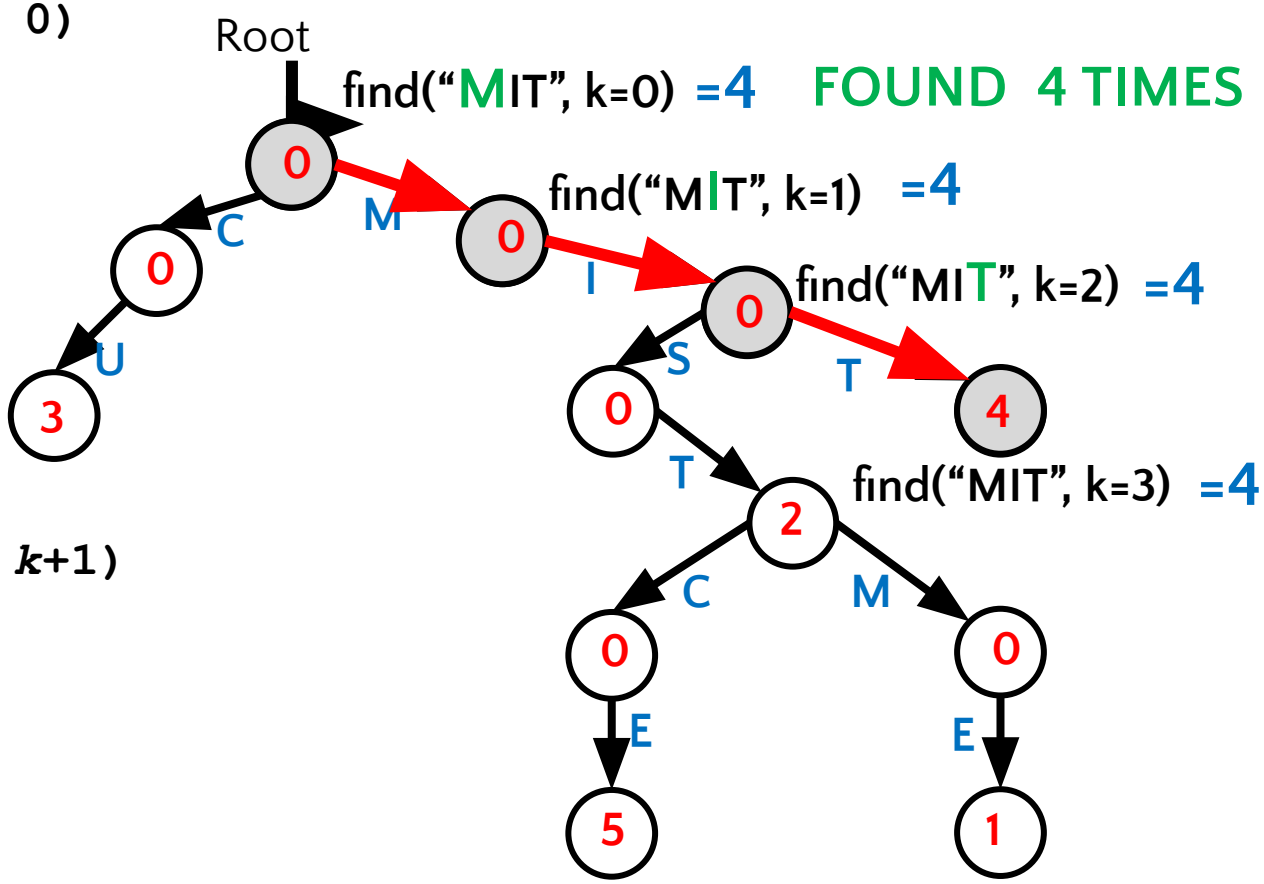
- find("MI")



SEARCH IN TRIE

```
find(x, Node pointer cur ← root, k ← 0)
    if cur is NULL
        return 0

    if k equals size(x)
        return cur->EoW

    r ← x[k] - 65
    return find(x, cur->children[r], k+1)
```

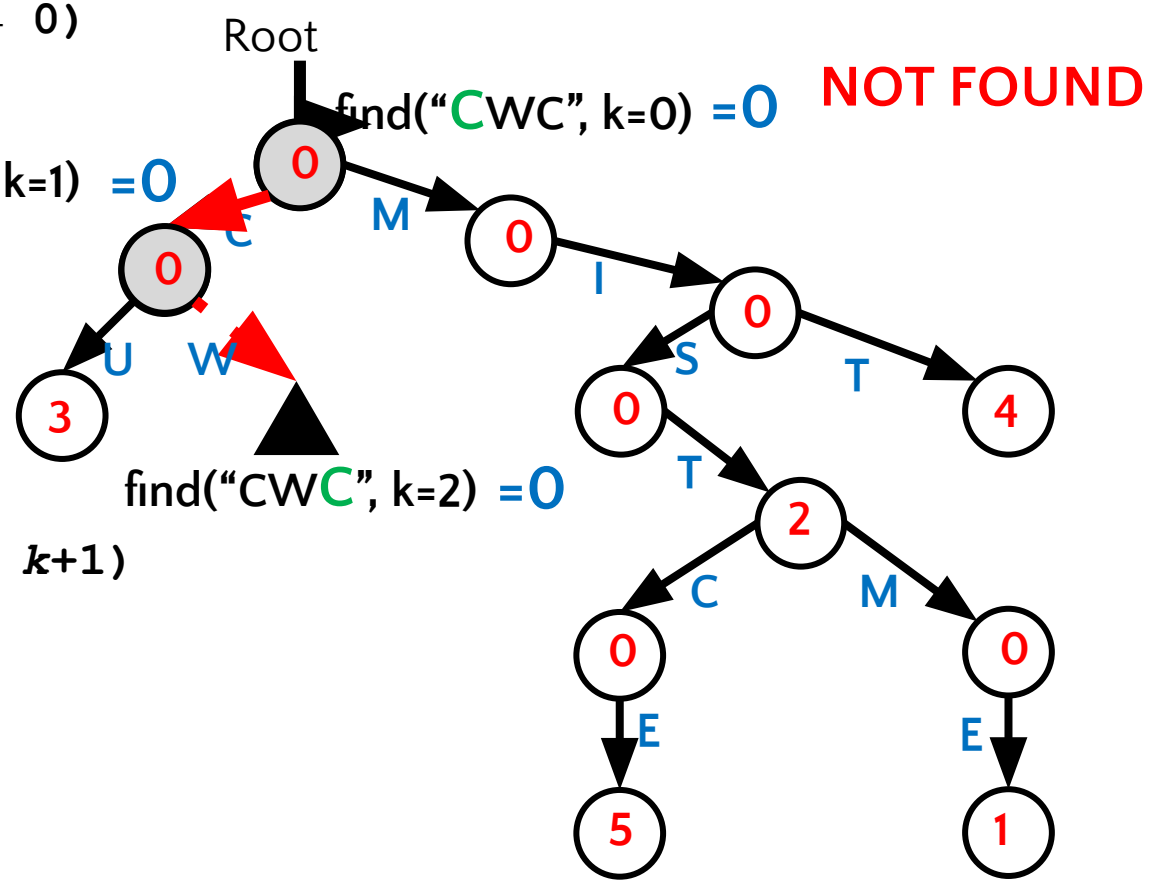
- find("MI")
- find("MIT")



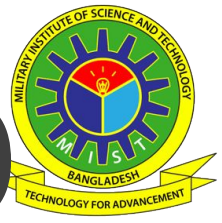
SEARCH IN TRIE

```
find(x, Node pointer cur ← root, k ← 0)
    if cur is NULL
        return 0

    if k equals size(x)
        return cur->EoW

    r ← x[k] - 65
    return find(x, cur->children[r], k+1)
```

- find("MI")
- find("MIT")
- find("CWC")



SEARCH IN TRIE (COMPLEXITY)



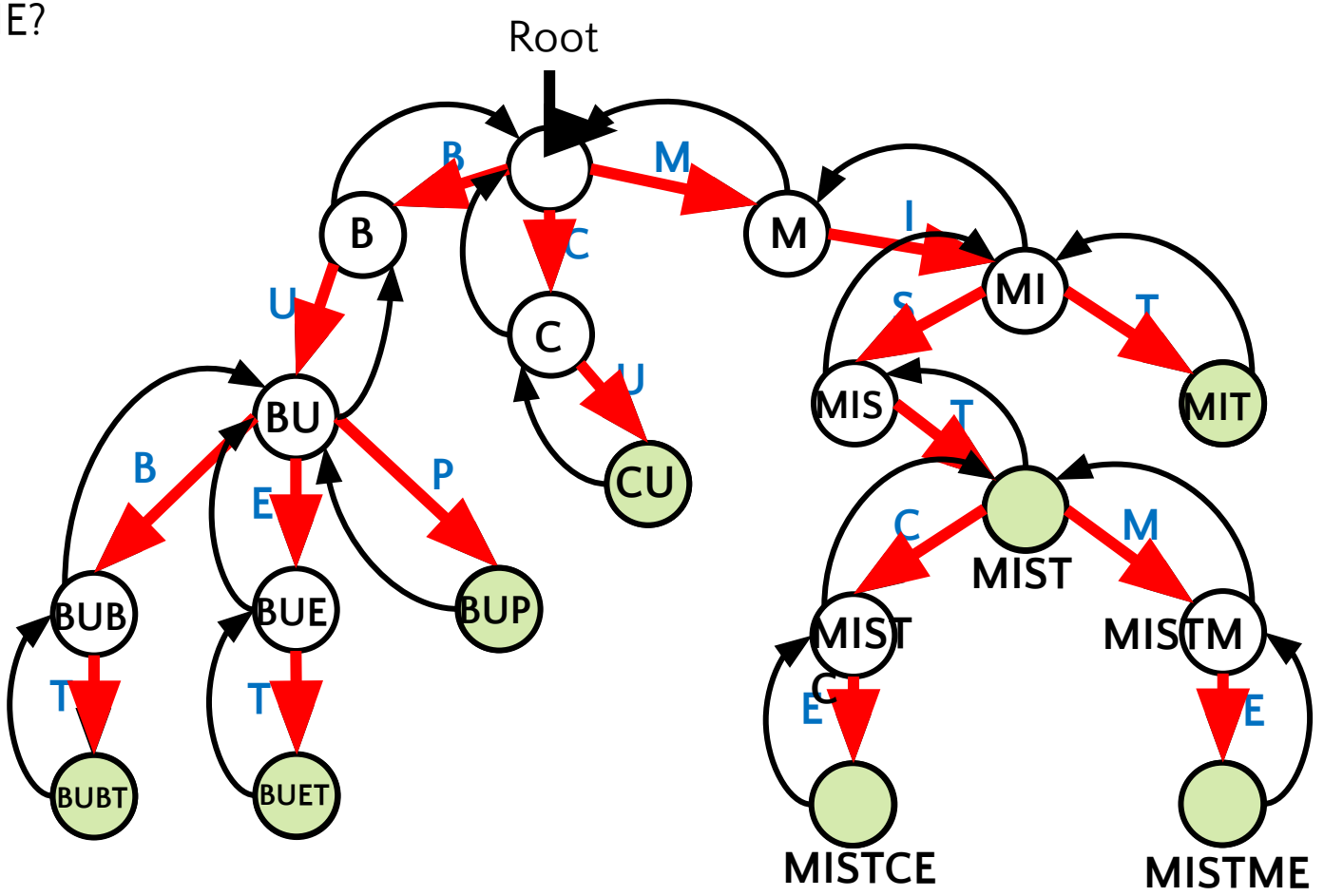
- Number of recursive call can not exceed the length of longest string in the TRIE
  - Let the longest string in the TRIE is ***s***
  - So the time complexity of searching is O(|***s***|)

LEXICOGRAPHICAL ORDER

- What are the strings stored in the TRIE?

BUBT
BUET
BUP
CU
MIST
MISTCE
MISTME
MIT

- Strings are sorted lexicographically
- Left to Right approach
  (Merging with parent)



LEXICOGRAPHICAL ORDER

```
void printTRIE(Node *cur = root, string s="")
{
    if(cur->EoW>0)
    {
        cout<<s<<endl;
    }
    for(int i=0; i<26; i++)
    {
        if(cur->children[i]!=NULL)
        {
            char c = char(i + 65);
            printTRIE(cur->children[i], s+c);
        }
    }
}
```

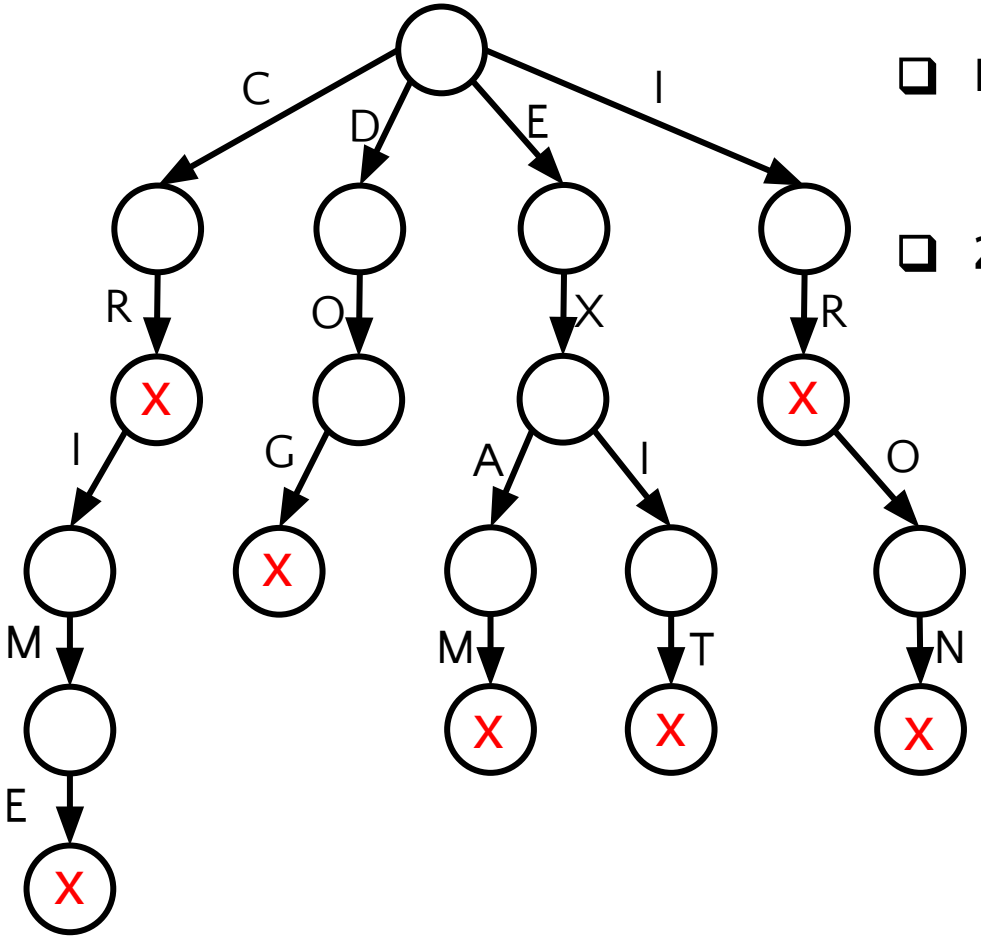
Base case:
If the pointer reaches to the end of a word
Then the word is printed

Traversing all the edges of a node from left to right

Calling the function recursively for those nodes
Having at least one child(edge).

So for leaf node: No recursive call is made

DELETE FROM TRIE



- **List down the words**

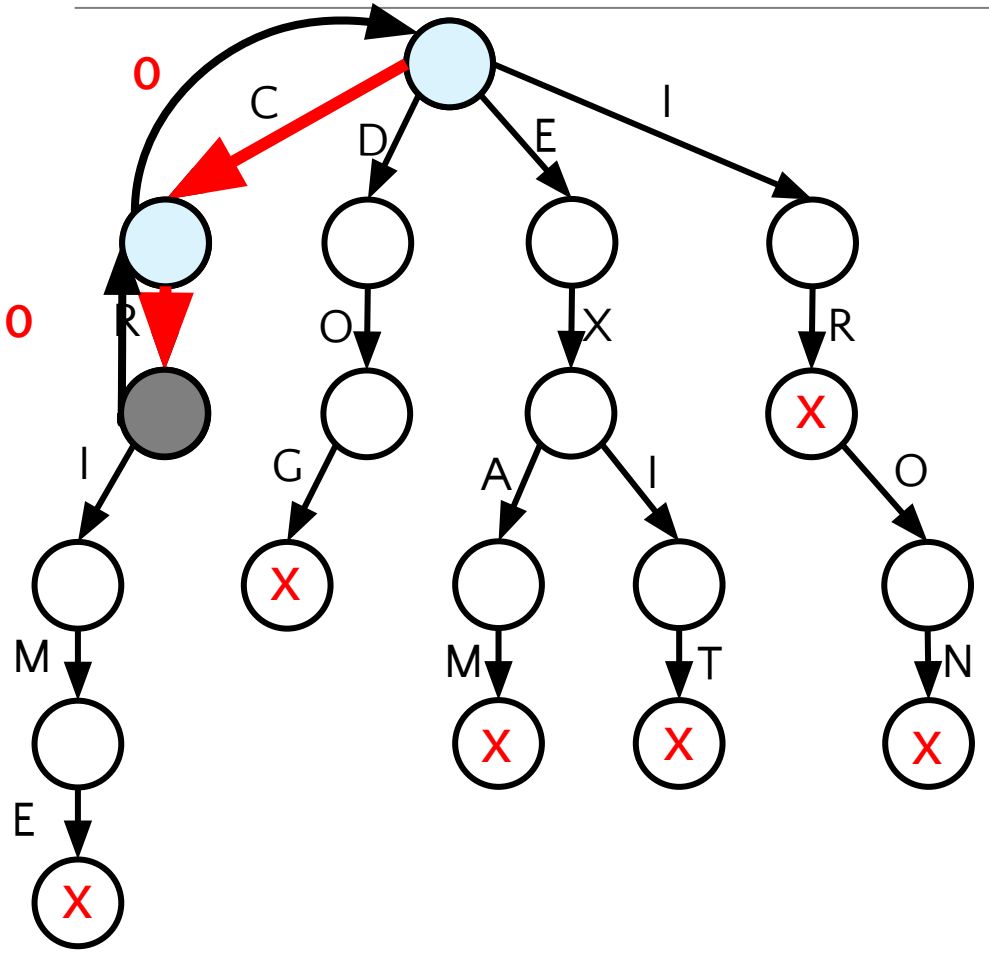
  CR CRIME DOG EXAM EXIT IR IRON

- **2 Cases for deletion**

  - The word is a prefix of other words
    - Ex : CR IR
  - The word is not a prefix of any other words
    - Ex : CRIME DOG EXAM EXIT IRON

But it is to be checked that whether the word exists in the TRIE or not before deletion

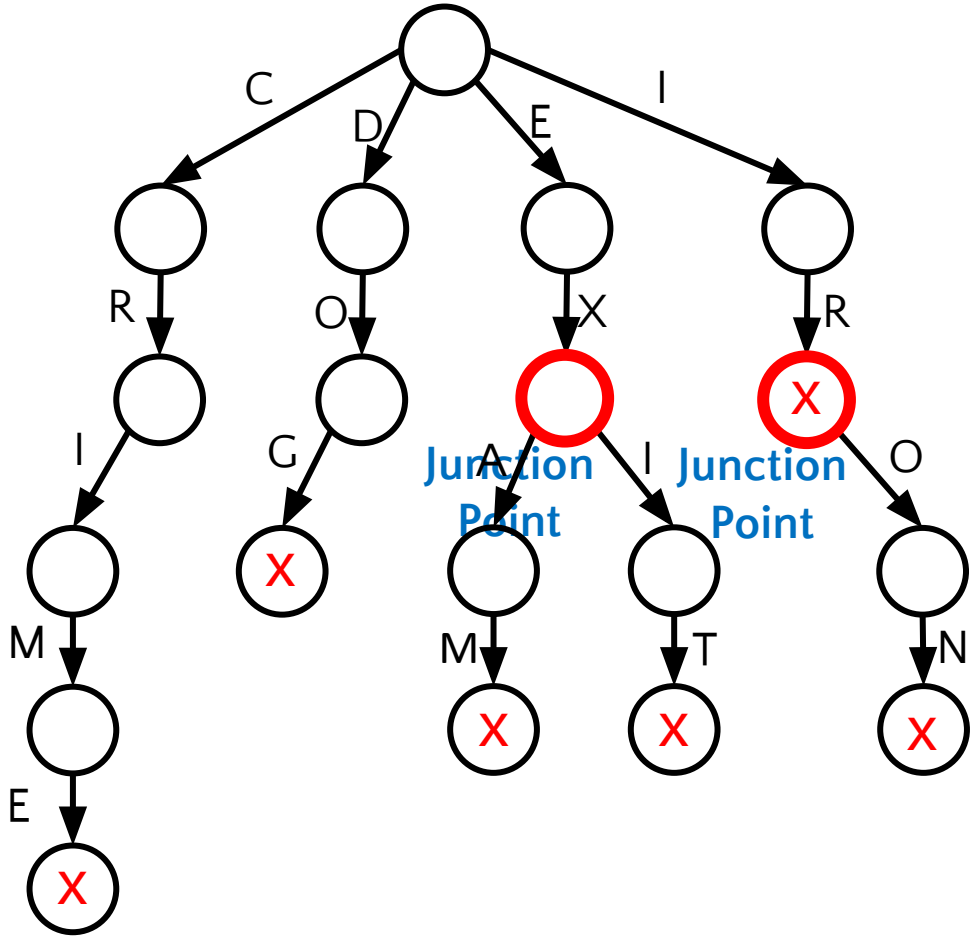
DELETE FROM TRIE (CASE-1)



- **The word is a prefix of other words**
  - Simply remove the EoW mark from the final node of the string in TRIE
  - delete("CR")
- **How did we understand that "CR" is a prefix of other words?**
  - Because the final node of CR in TRIE is not a leaf
- **How to check that whether a node is a leaf or not?**
  - Leaf: If a node having no child or all the child point to NULL

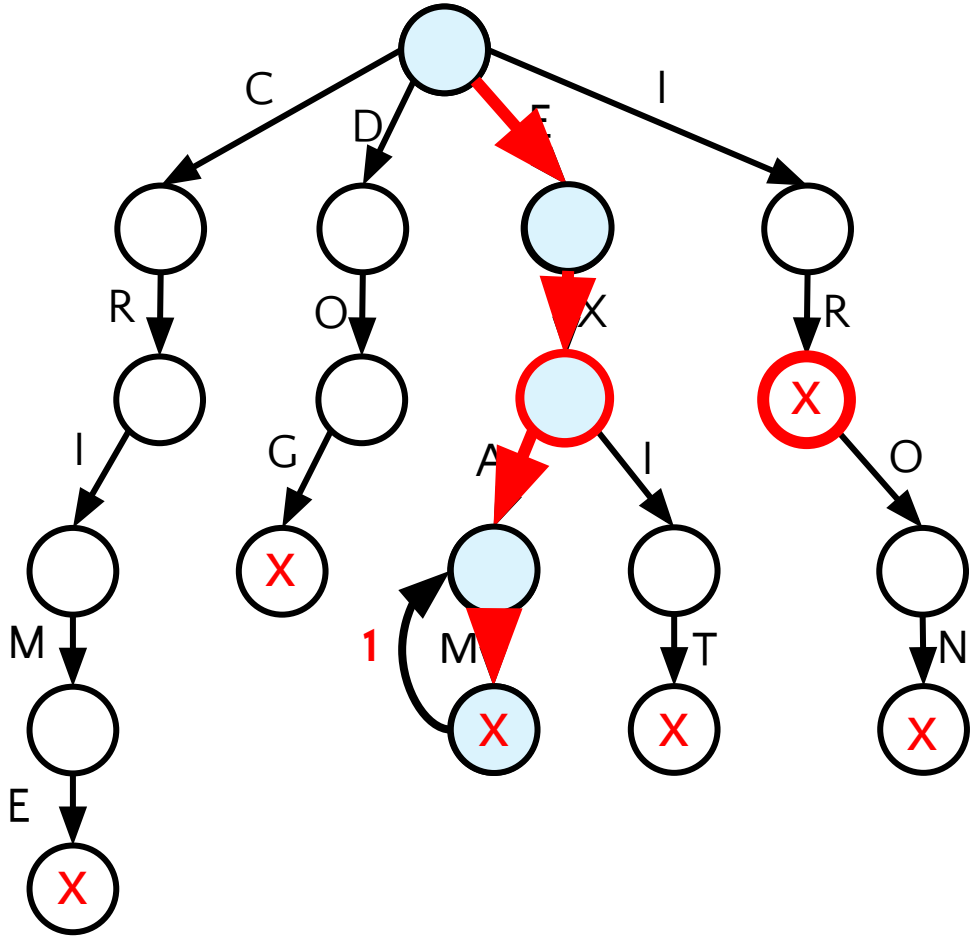
```
bool isLeaf(Node *u){
    for(int i=0; i<26; i++)    if(u->children[i]!=NULL)    return false;
    return true;
}
```

DELETE FROM TRIE (CASE-2)



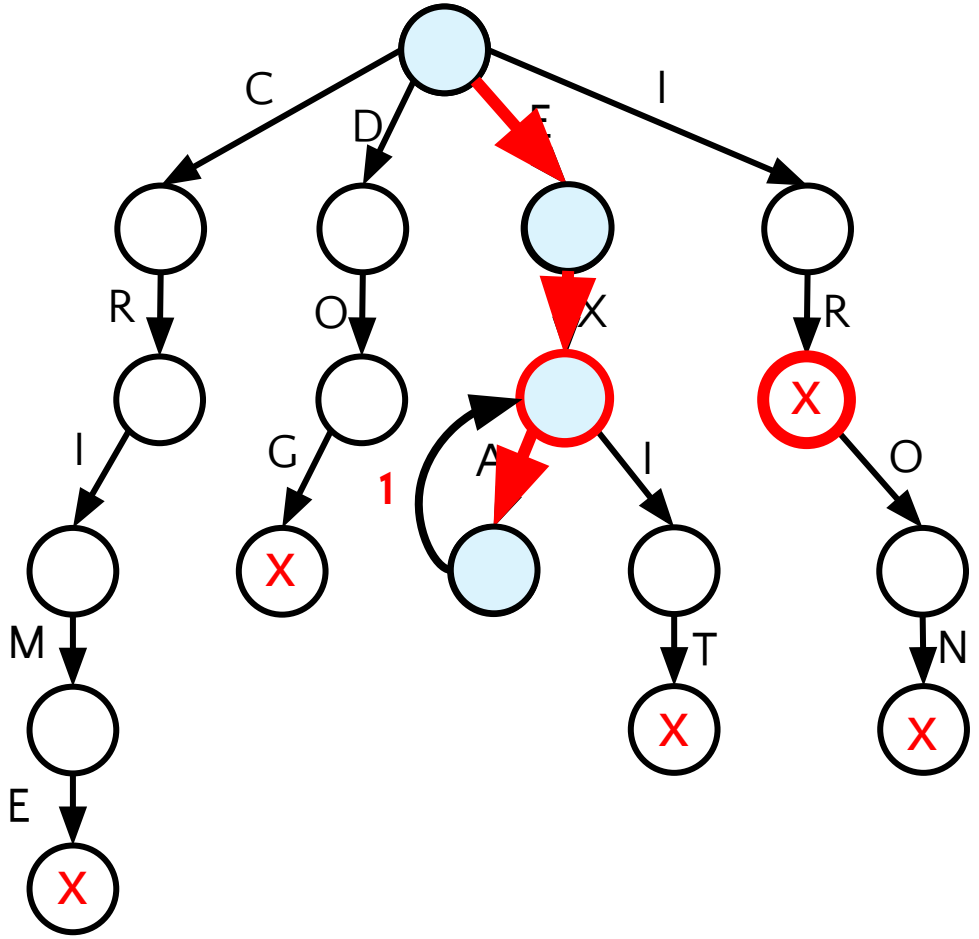
- **The word is not a prefix of other words**
  - Remove all the nodes from leaf node to the first junction node associated with the string along with the edges
  - delete("EXAM")
  - delete("IRON")

DELETE FROM TRIE (CASE-2)



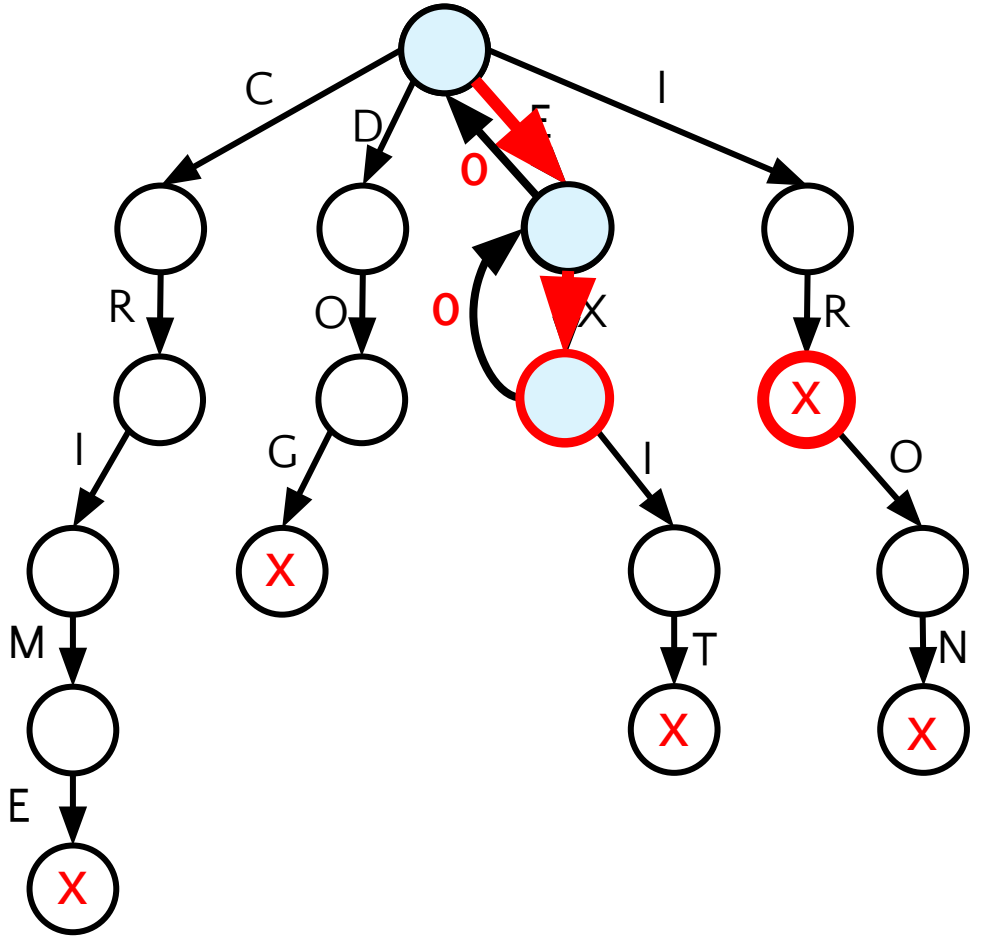
- **The word is not a prefix of other words**
  - Remove all the nodes from leaf node to the first junction node associated with the string along with the edges
  - delete("EXAM")

DELETE FROM TRIE (CASE-2)



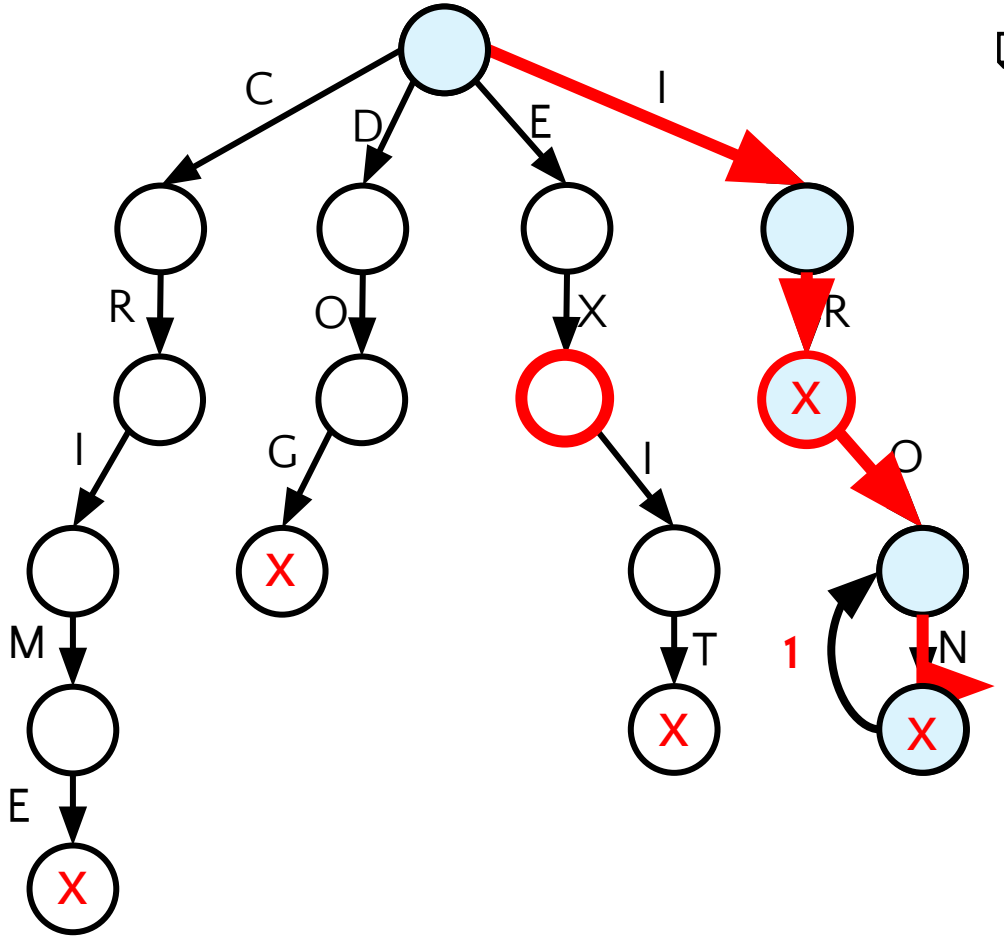
- **The word is not a prefix of other words**
  - Remove all the nodes from leaf node to the first junction node associated with the string along with the edges
  - delete("EXAM")

DELETE FROM TRIE (CASE-2)



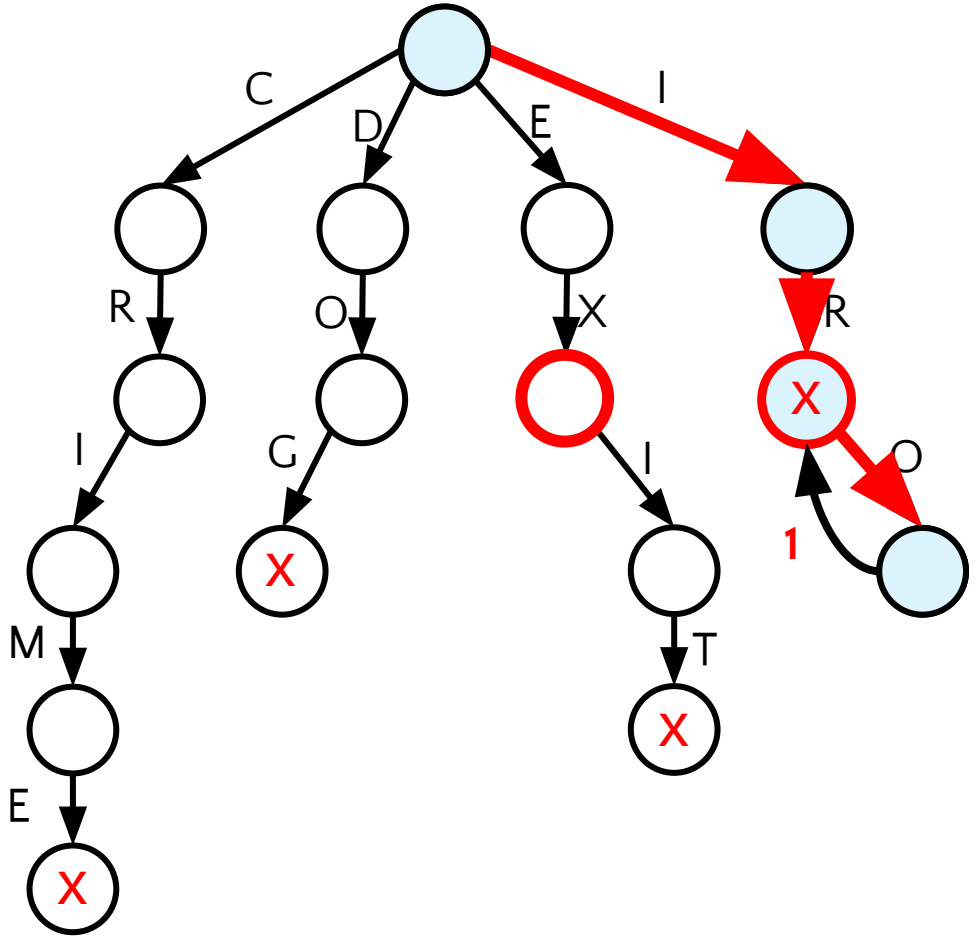
- **The word is not a prefix of other words**
  - Remove all the nodes from leaf node to the first junction node associated with the string along with the edges
  - delete("EXAM")

DELETE FROM TRIE (CASE-2)



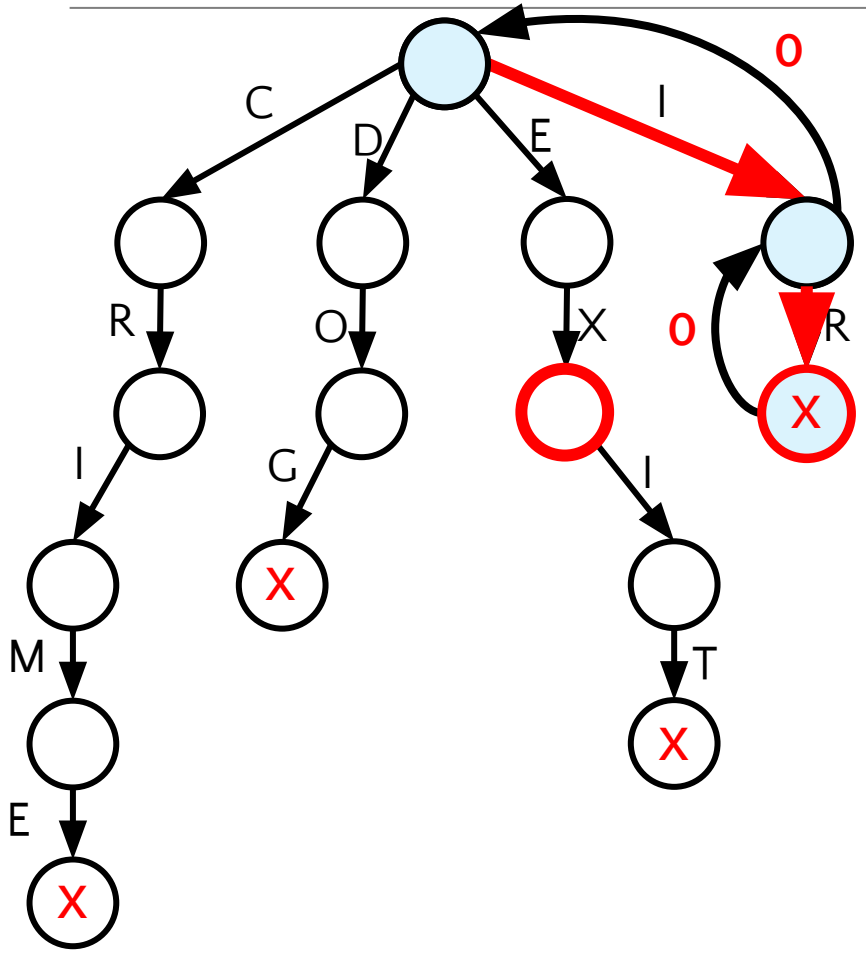
- **The word is not a prefix of other words**
  - Remove all the nodes from leaf node to the first junction node associated with the string along with the edges
  - delete("EXAM")
  - delete("IRON")

DELETE FROM TRIE (CASE-2)



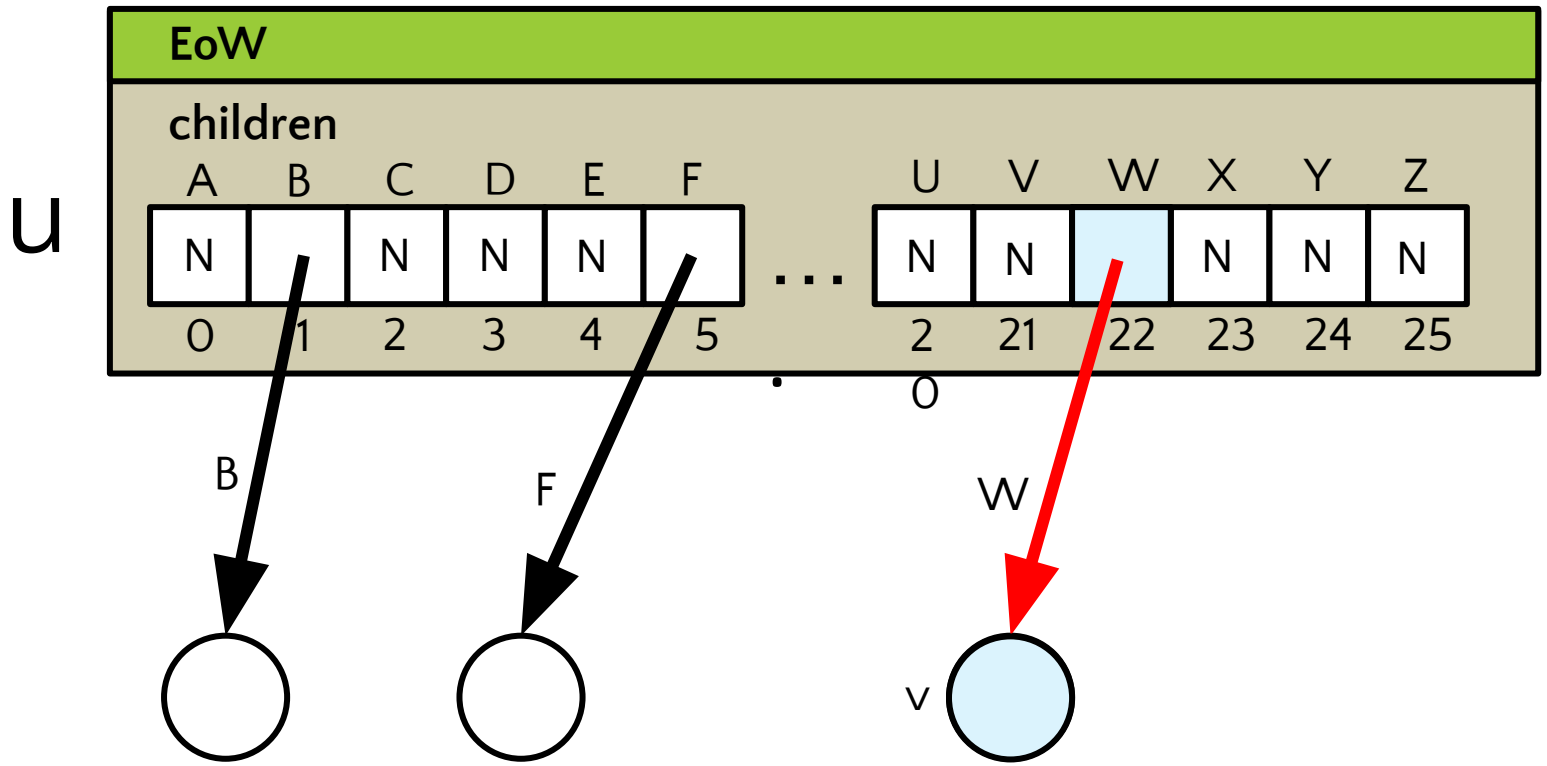
- **The word is not a prefix of other words**
  - Remove all the nodes from leaf node to the first junction node associated with the string along with the edges
  - delete("EXAM")
  - delete("IRON")

DELETE FROM TRIE (CASE-2)



- **The word is not a prefix of other words**
  - Remove all the nodes from leaf node to the first junction node associated with the string along with the edges
  - delete("EXAM")
  - delete("IRON")

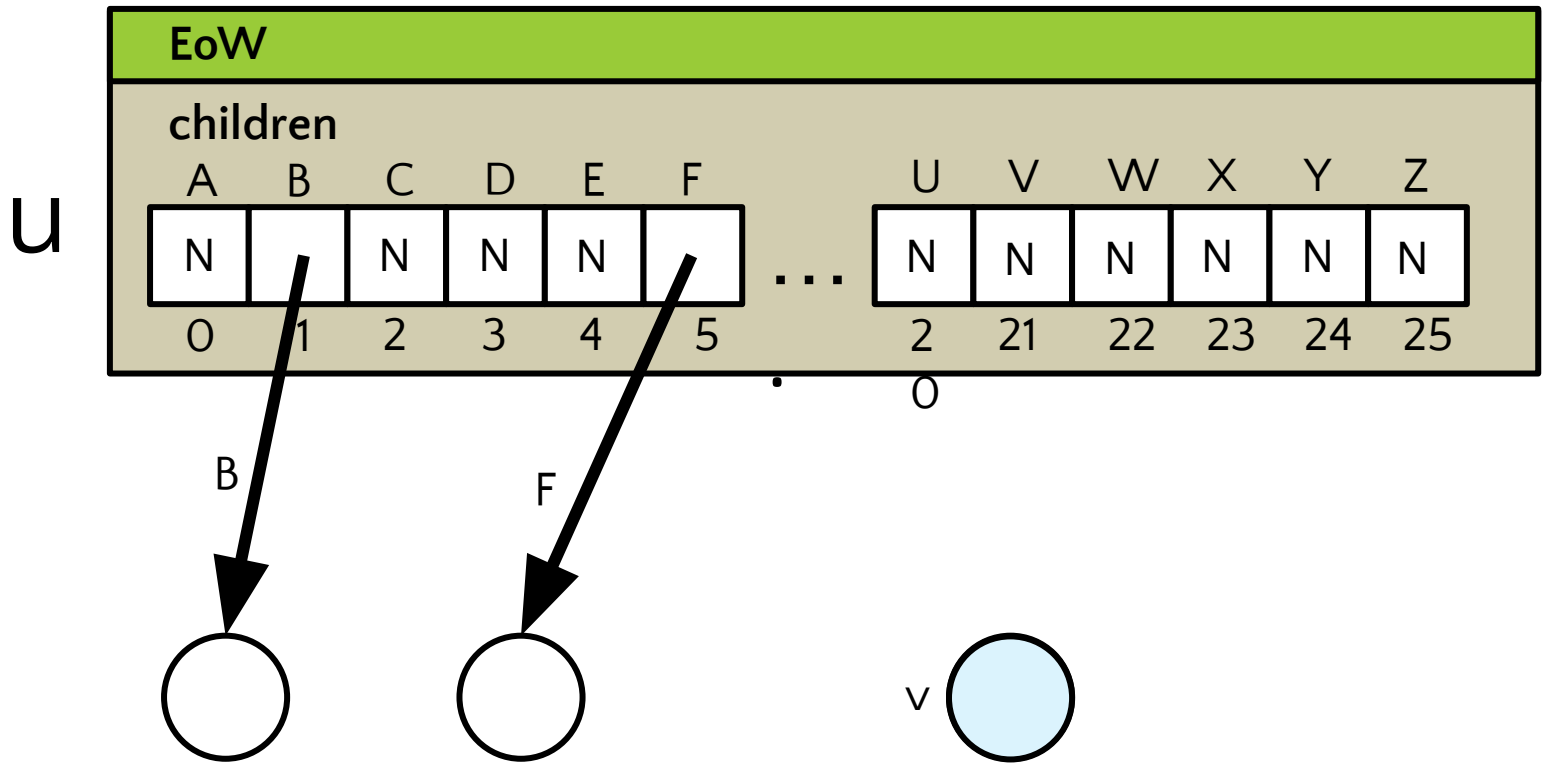
DELETION OF AN EDGE




```
r ← 22
Node *v ← u->children[r]
u->children[r] = NULL
```

DELETE THE RED MARKED EDGE

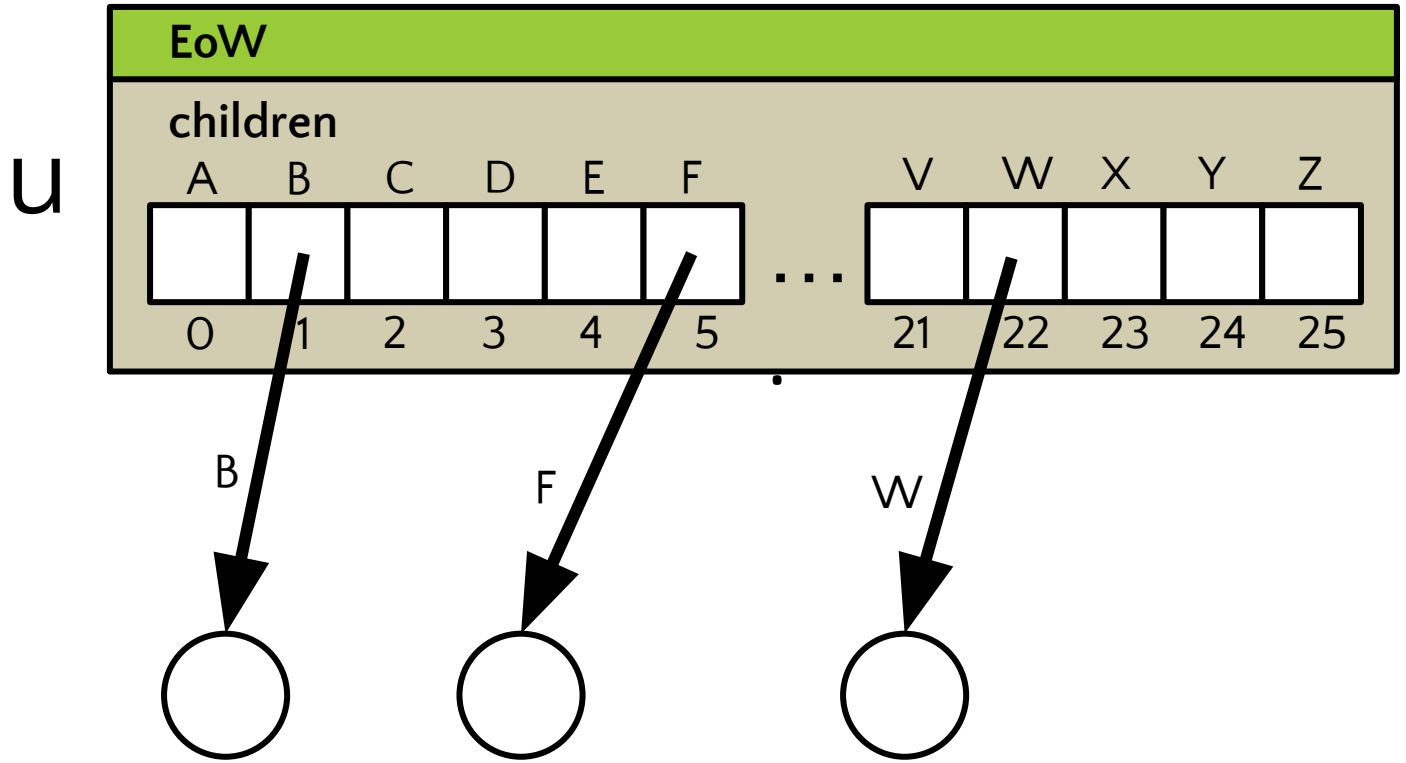
DELETION OF AN EDGE




```
r ← 22
Node *v ← u->children[r]
u->children[r] = NULL
delete v
```

DELETE THE RED MARKED EDGE

DELETION OF AN EDGE




```
deleteEdge(Node *u, char c, int d)
    if d is 0
        return without doing anything

    r ← c-65
    Node *v ← u->children[r]
    u->children[r] ← NULL
    delete v
```

DELETE IN TRIE

```
delete(string x, Node *u ← root, k ← 0)
    if u is NULL
        return 0
    if k equals size(x)
        if u->EoW is 0
            return 0
        if isLeaf(u) is false
            u->EoW = 0
            return 0
        return 1
    r ← x[k]-65
    d ← delete(x, u->children[r],k+1)
    j <- isJunction(u)
    removeEdge(u, x[k], d)
    if j is 1
        d ← 0
    return d
```

Traversing of x is not complete

r becomes the relative position of k-th character in x

d becomes 1 if the next node is removable

Otherwise d becomes 0

If u is a junction then set j variable to 1.

Removes the k-th edge of u if d permits

Then if u was a junction before removing the edge then sets the permission as 0

Then sends the permission to it's parent

JUNCTION POINT

- A node containing an EoW=1 mark
- A node having at least 2 child

```
bool isJunction(Node *u)
    count<-0
    for(int i=0;i<26;i++)
        if(u->children[i]!=NULL)    count++;
    if(u->EoW>0 or count>1)    return true;
    return false;
```


MILITARY INSTITUTE OF SCIENCE AND TECHNOLOGY
M I S T
BANGLADESH
TECHNOLOGY FOR ADVANCEMENT

Thank You!

CSE 215: Data Structures & Algorithms II

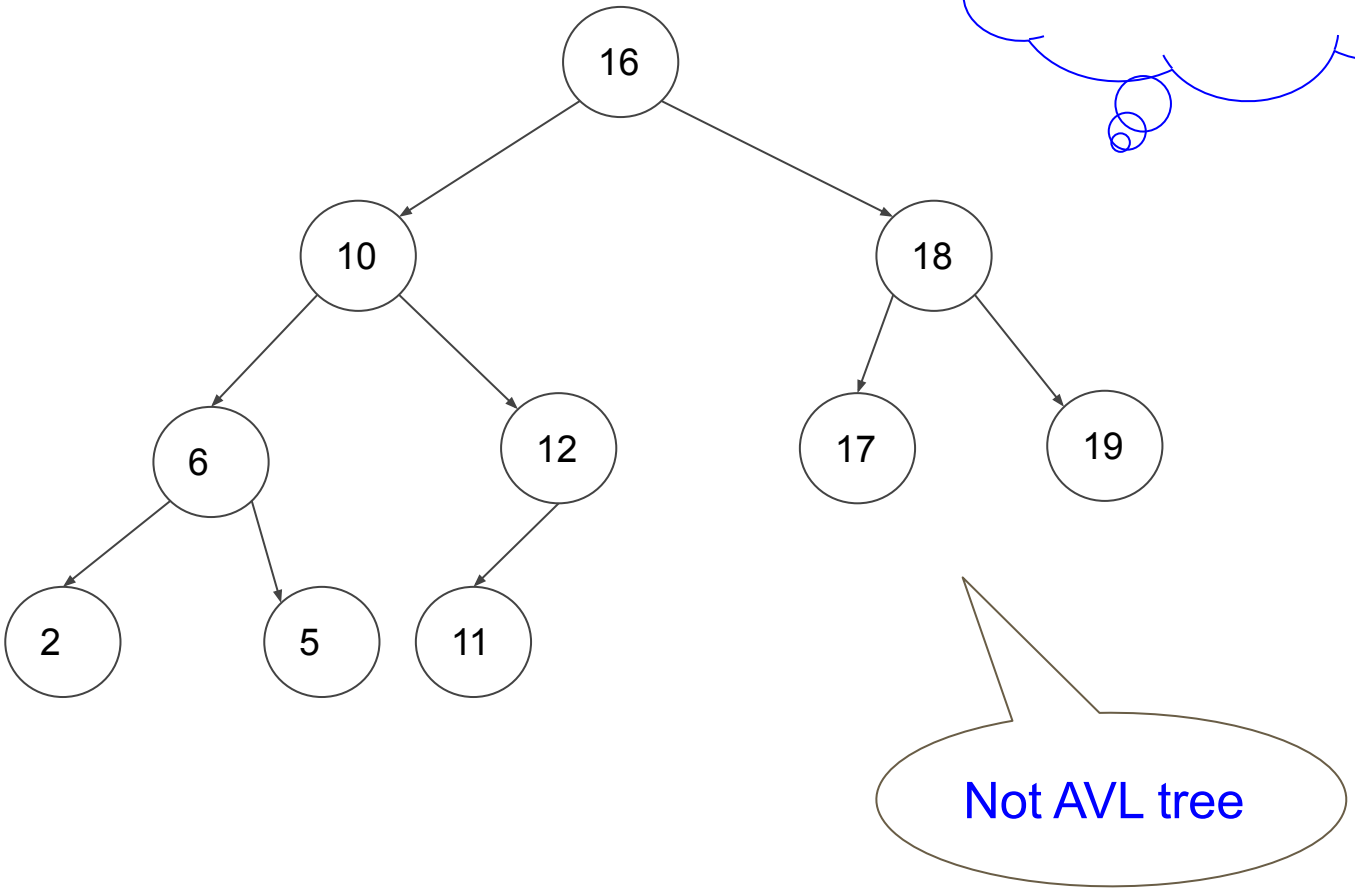
AVL Tree

Lec Sumaiya Nuha Mustafina
Dept Of CSE
sumaiyamustafina@cse.mist.ac.bd

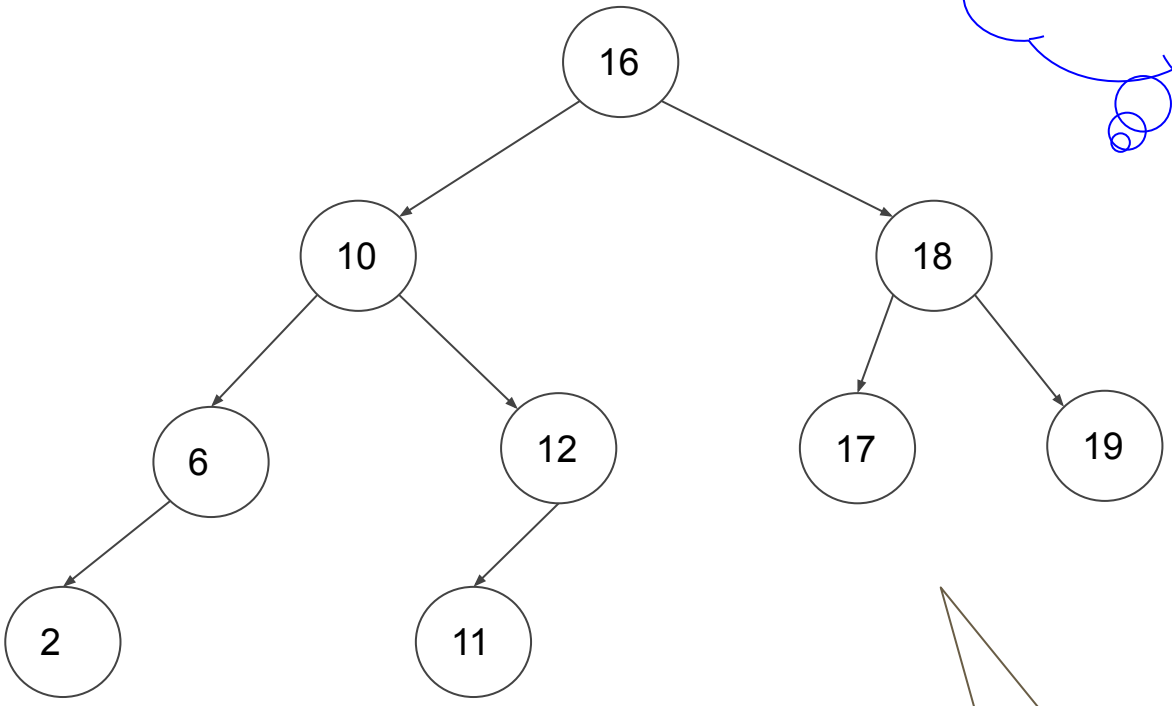
AVL Tree

- Adelson-Velskii and Landis Tree
- Was invented in 1962
- A binary search tree that is *height balanced*
- *Balanced condition:* For every node in the tree, the height of the left and right subtrees can differ by at most 1.
- Self balancing BST.
- Maintains trees minimum possible height
- Used in database indexing, corporate areas etc
- Height : O(log n)
- Insertion Deletion : O(log n)

AVL Tree



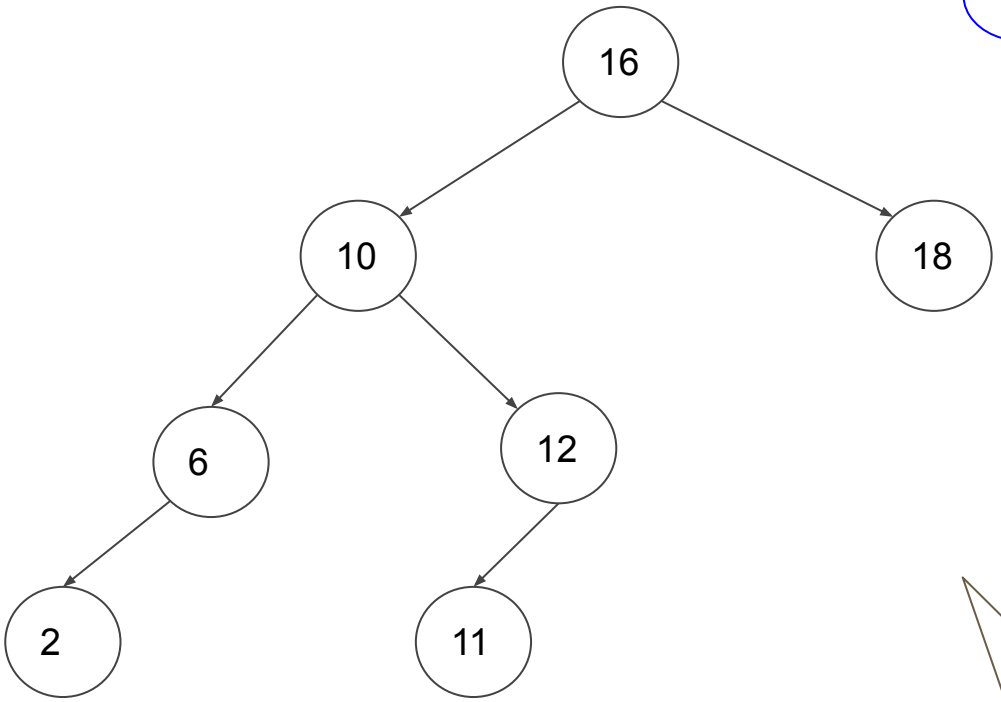
AVL Tree



AVL Tree or Not !!!!

AVL Tree

AVL Tree



AVL Tree or Not !!!!

Not AVL tree

AVL Tree

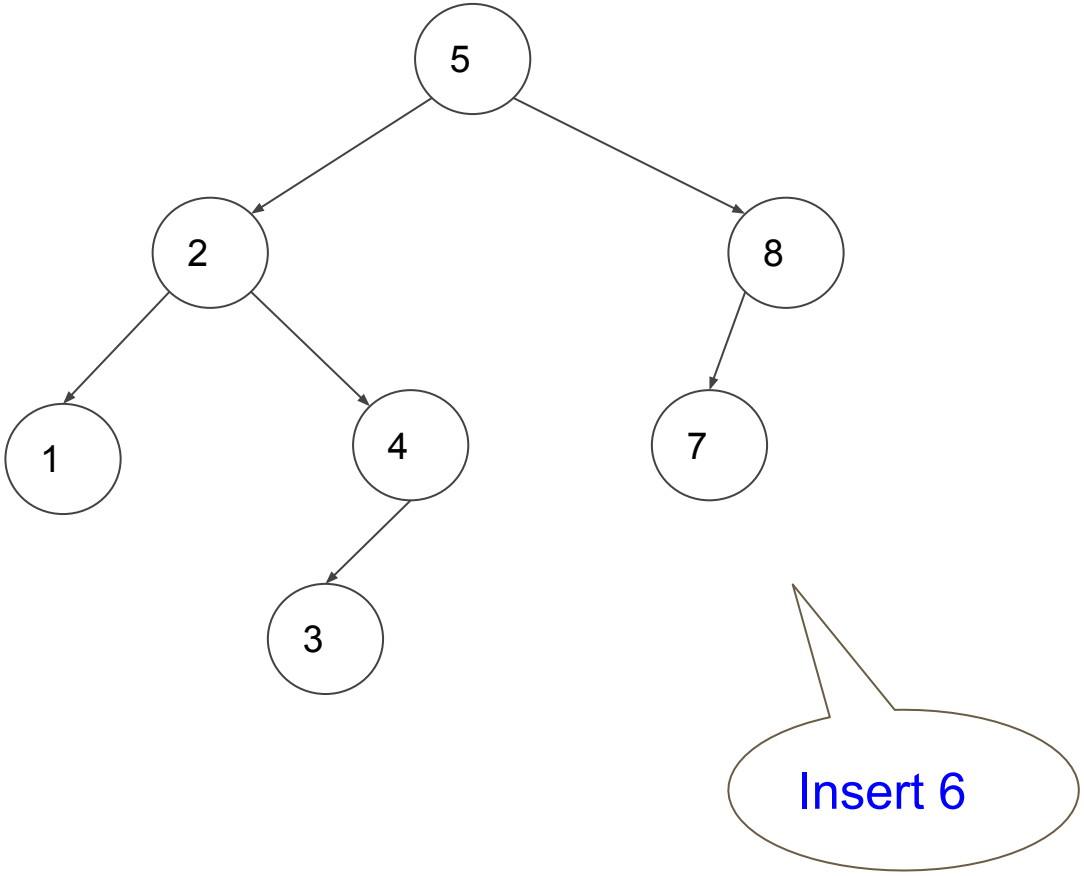
Node Representation:

```
struct Node{
    Node *left;
    Node *right;
    int element;
    int height;
}
```

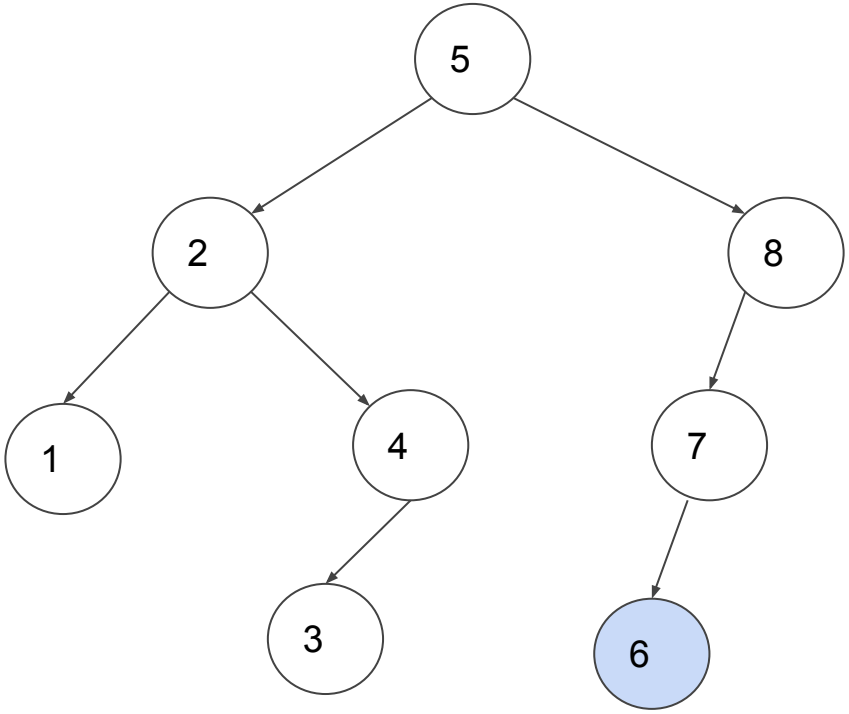
Height of node / tree:

```
int h(Node *u)
{
  return (u == NULL) ? -1 : u->height;
}
```

AVL Tree : Insertion

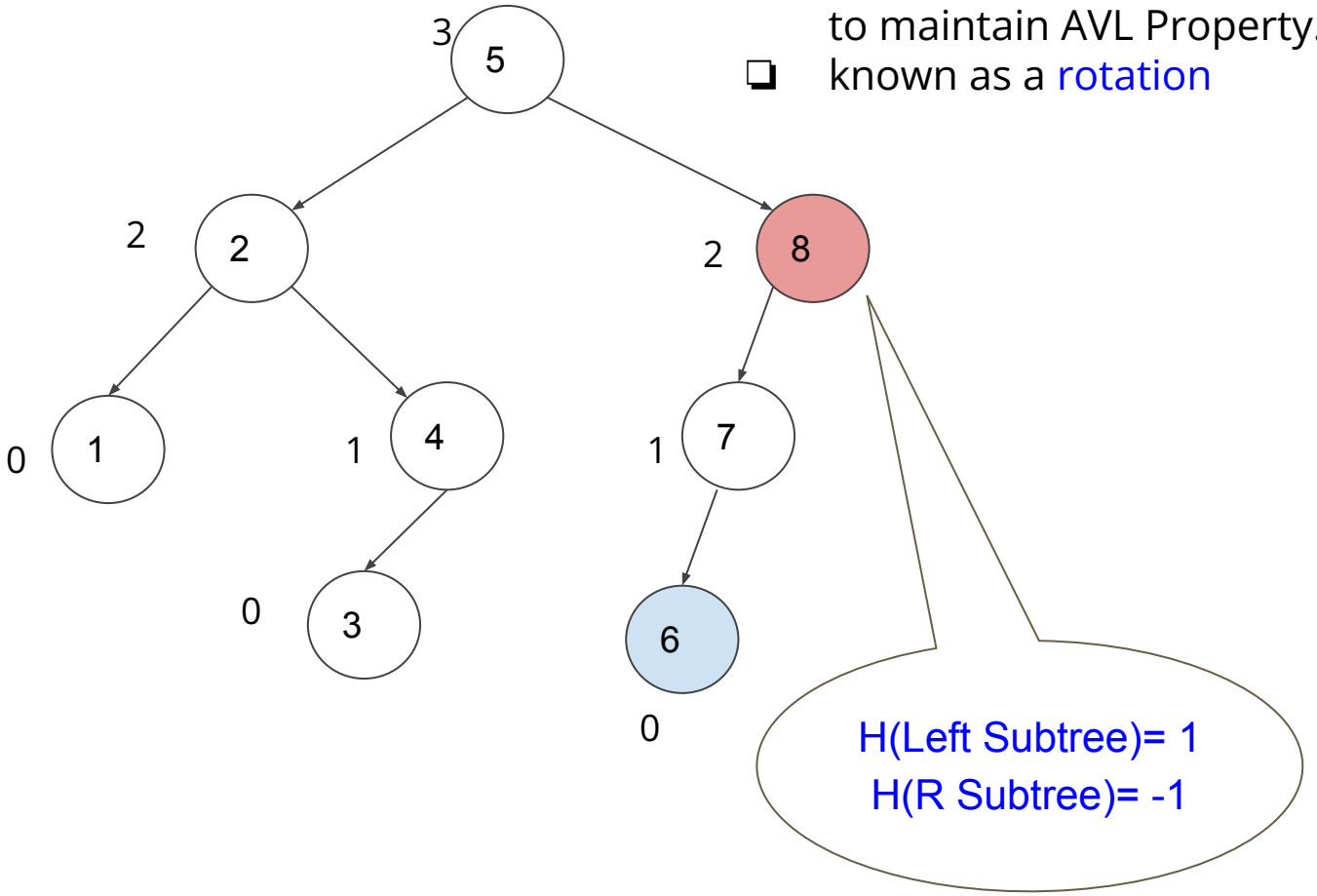


AVL Tree : Insertion



AVL Tree : Insertion

- It violates the AVL condition.
- Need to rebalance the tree to maintain AVL Property.
- known as a rotation



AVL Tree : Insertion

Say, the node that must be rebalanced is **u.**

Since any node has at most two children, and a height imbalance requires that α's two subtrees' heights differ by two,

So, a violation might occur in four cases:

- An insertion into the left subtree of the left child of u
- An insertion into the right subtree of the left child of u
- An insertion into the left subtree of the right child of u
- An insertion into the right subtree of the right child of u

AVL Tree : Insertion

Two types of Rebalancing technique (Rotation) is used.

- ❏ Single Rotation
- ❏ Double Rotation

- Case 1 and Case 4 (left - left insertion, right-right insertion i.e. insertion occur on the outside) is fixed by a *single rotation* of the tree.
- Case 2 and 3, in which the insertion occurs on the “inside” (i.e., left–right or right–left) is handled by the slightly more complex *double rotation*.

AVL Tree : Insertion

Steps of Insertion in an AVL tree:

1. Insert in proper position maintaining BST property.
2. Check if balance property is being maintained in the ancestor nodes of the node inserted.
3. If not, use proper rotation technique.
4. Repeat step 2 and 3 till the first unbalanced node or till root.

AVL Tree : Insertion

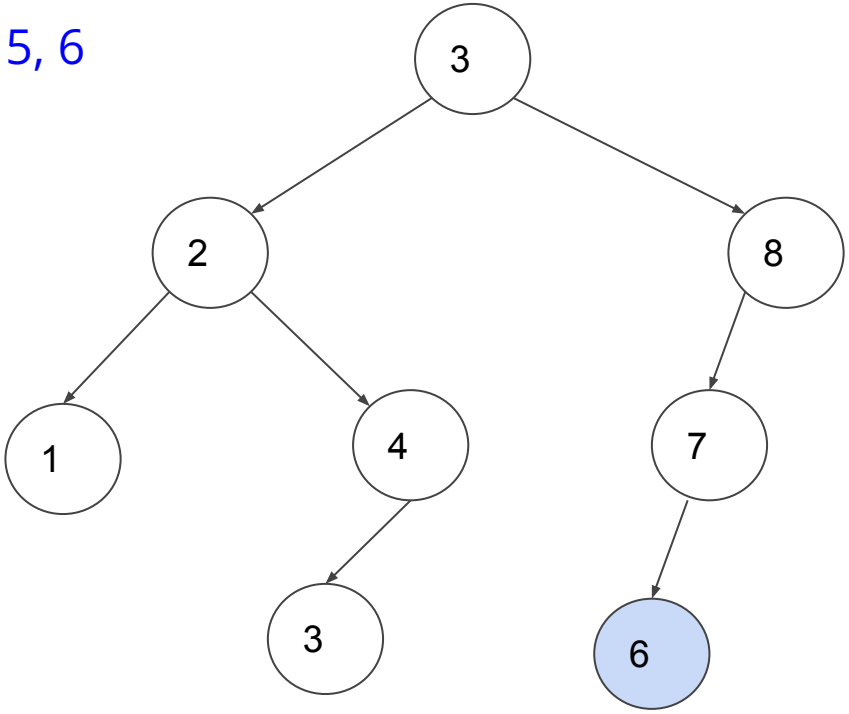
Single Rotation

- ❏ Right Rotation : single rotation between the parent and its left child
- ❏ Left Rotation : single rotation between the parent and its right child

- left - left insertion : Right Rotation
- right-right insertion: Left Rotation

AVL Tree : Insertion

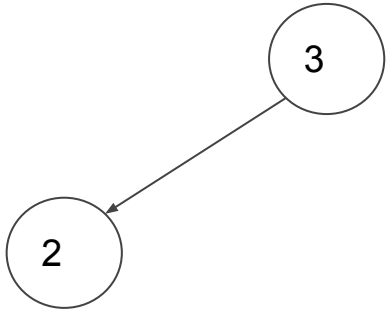
Insert 3 ,2 , 1, 4, 5, 6



- left - left insertion : Right Rotation

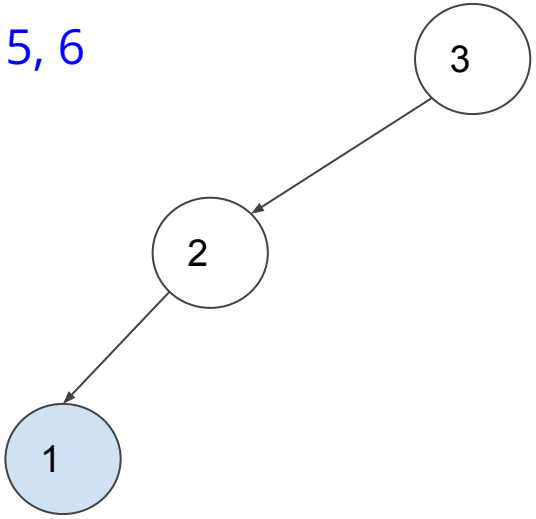
AVL Tree : Insertion

Insert 3 ,2 , 1, 4, 5, 6



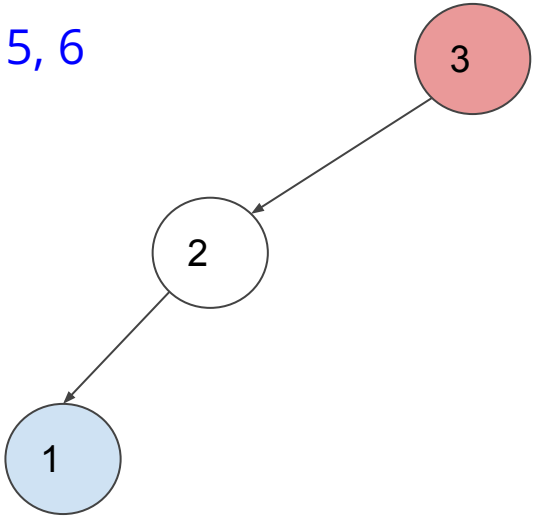
AVL Tree : Insertion

Insert 3 ,2 , 1, 4, 5, 6



AVL Tree : Insertion

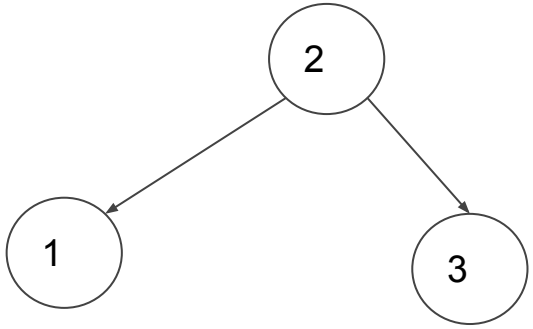
Insert 3 ,2 , 1, 4, 5, 6



- u = node with value 3
- left - left insertion : Right Rotation

AVL Tree : Insertion

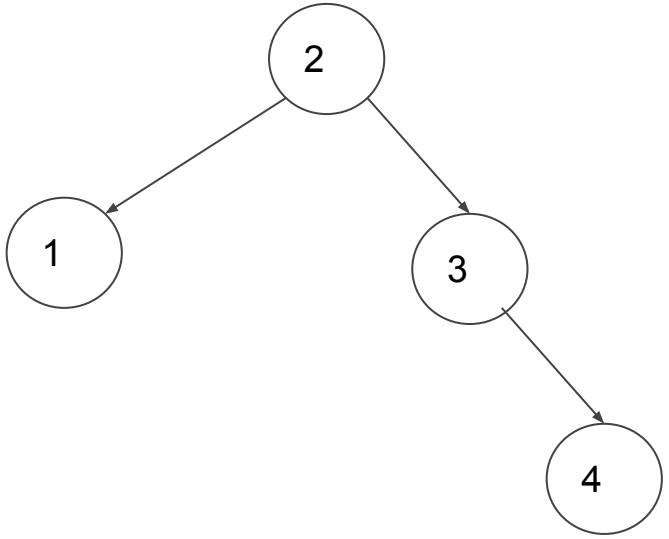
Insert 3 ,2 , 1, 4, 5, 6



After Right Rotation

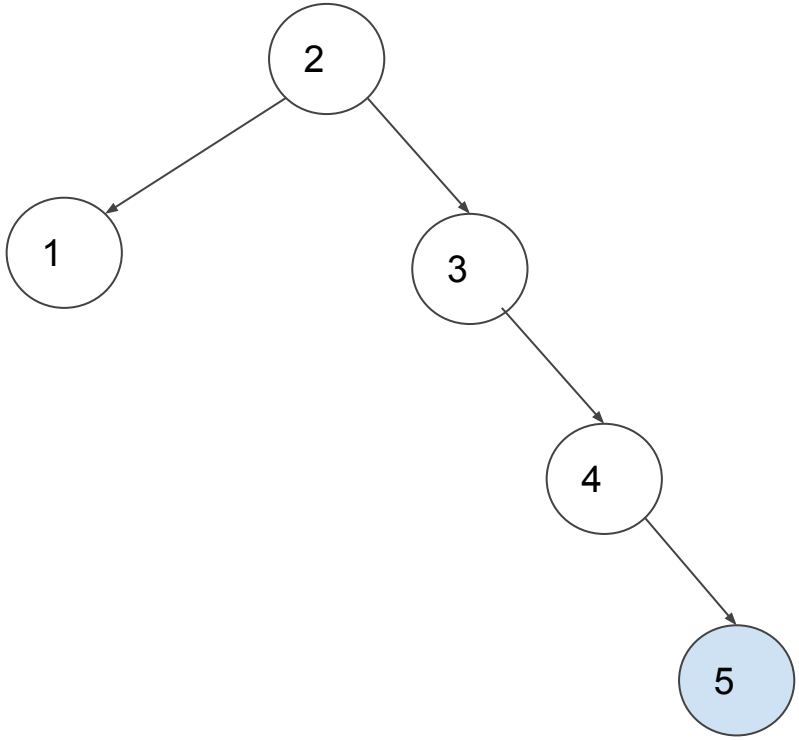
AVL Tree : Insertion

Insert 3 ,2 , 1, 4, 5, 6



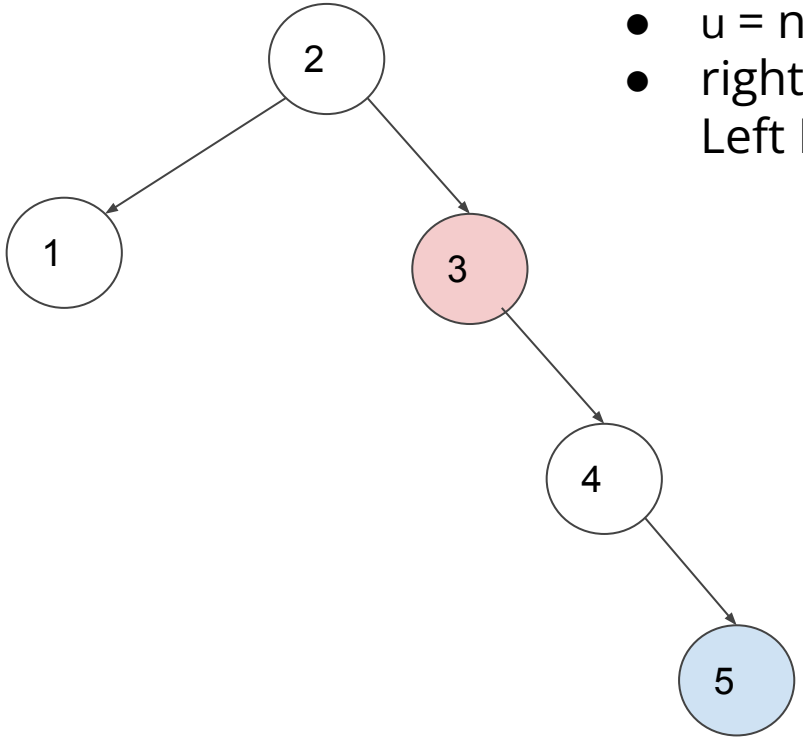
AVL Tree : Insertion

Insert 3 ,2 , 1, 4, 5, 6



AVL Tree : Insertion

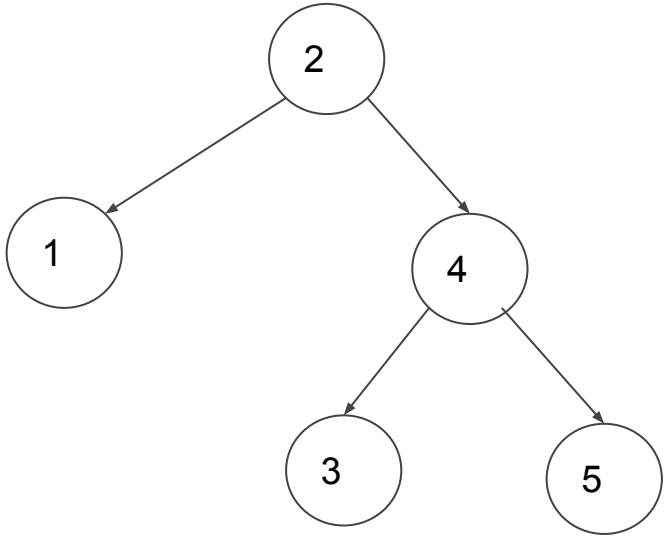
Insert 3 ,2 , 1, 4, 5, 6



- u = node with value 3
- right - right insertion : Left Rotation

AVL Tree : Insertion

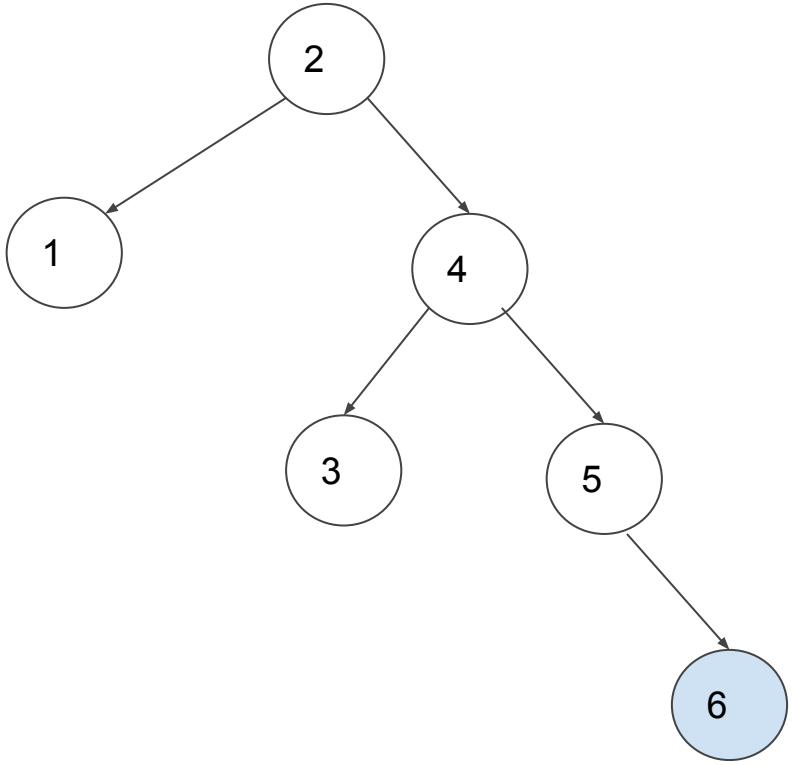
Insert 3 ,2 , 1, 4, 5, 6



After Left Rotation

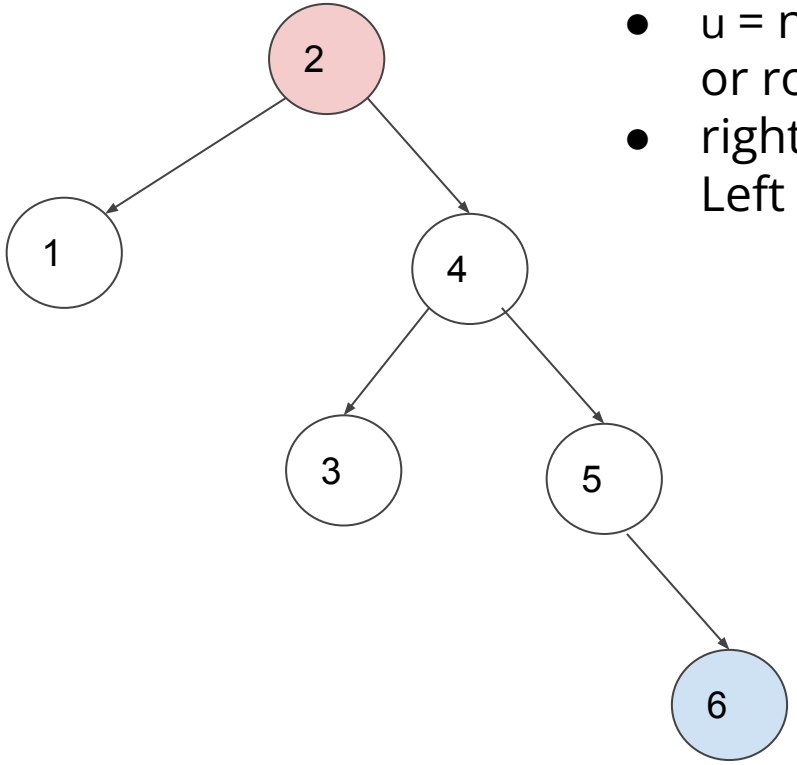
AVL Tree : Insertion

Insert 3 ,2 , 1, 4, 5, 6



AVL Tree : Insertion

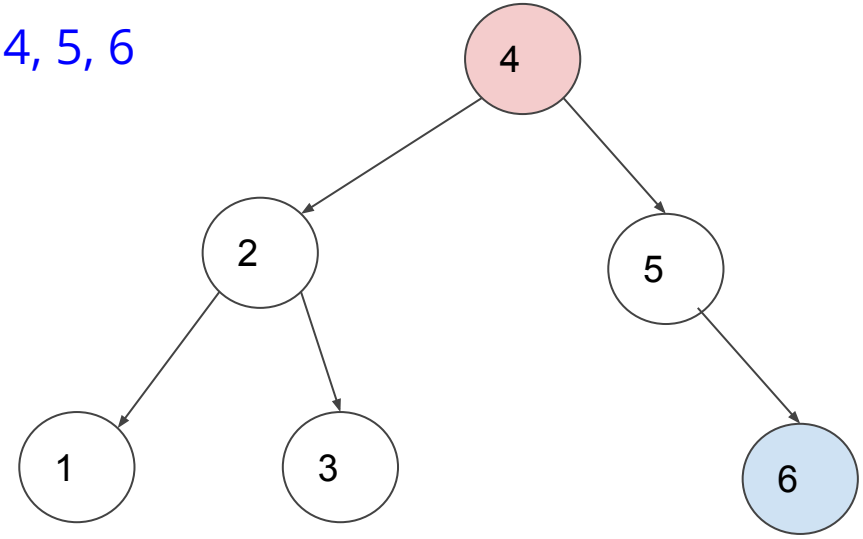
Insert 3 ,2 , 1, 4, 5, 6



- u = node with value 2 or root
- right - right insertion : Left Rotation

AVL Tree : Insertion

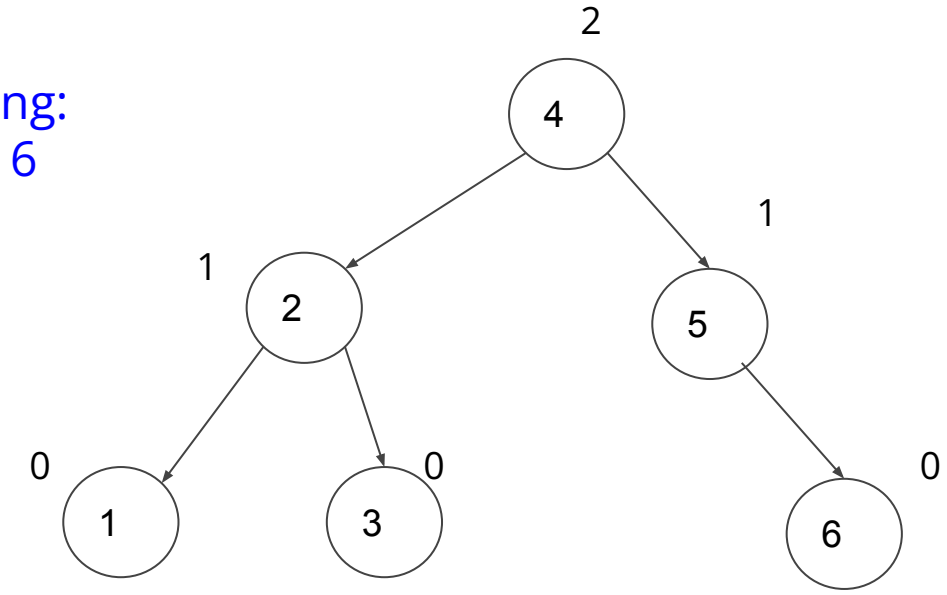
Insert 3 ,2 , 1, 4, 5, 6



After Left Rotation

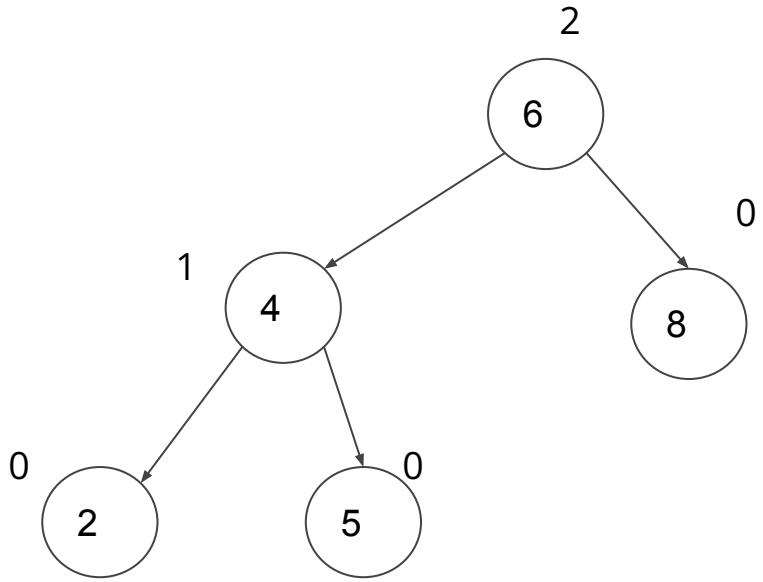
AVL Tree : Insertion

After Inserting:
3 ,2 , 1, 4, 5, 6



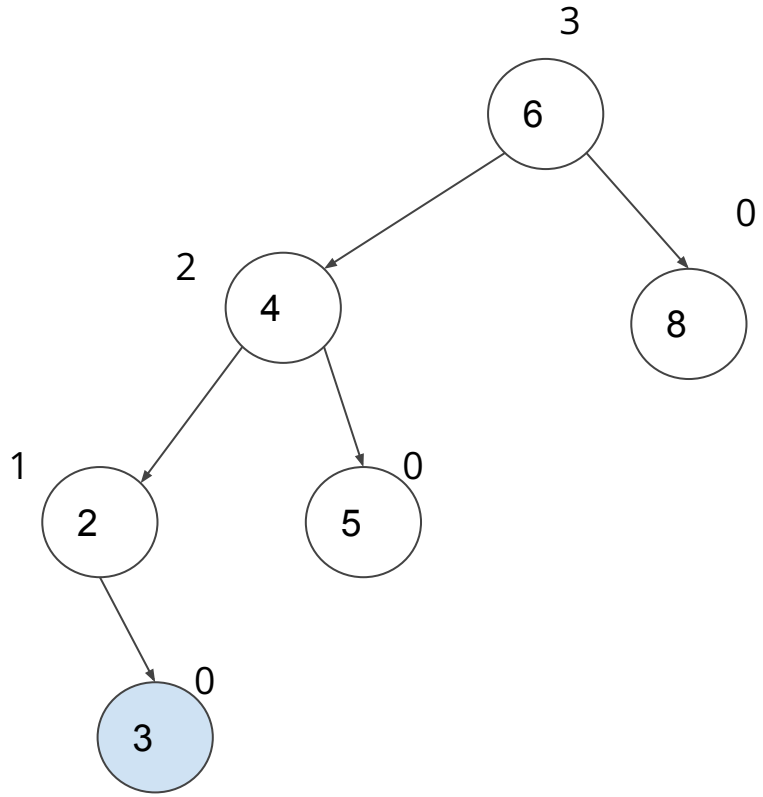
AVL Tree : Insertion

Insert: 3



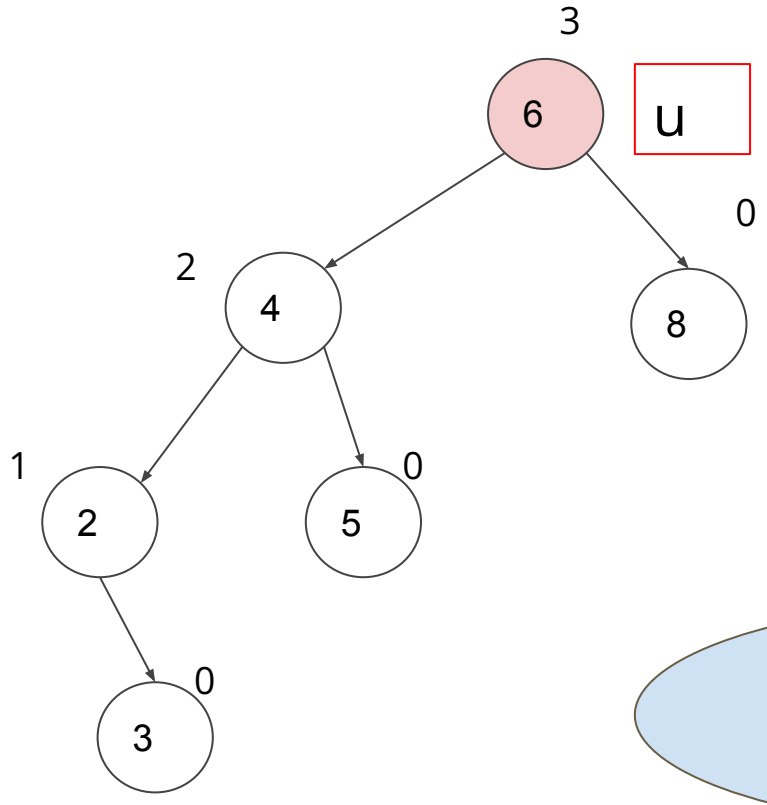
AVL Tree : Insertion

Inserting: 3



AVL Tree : Insertion

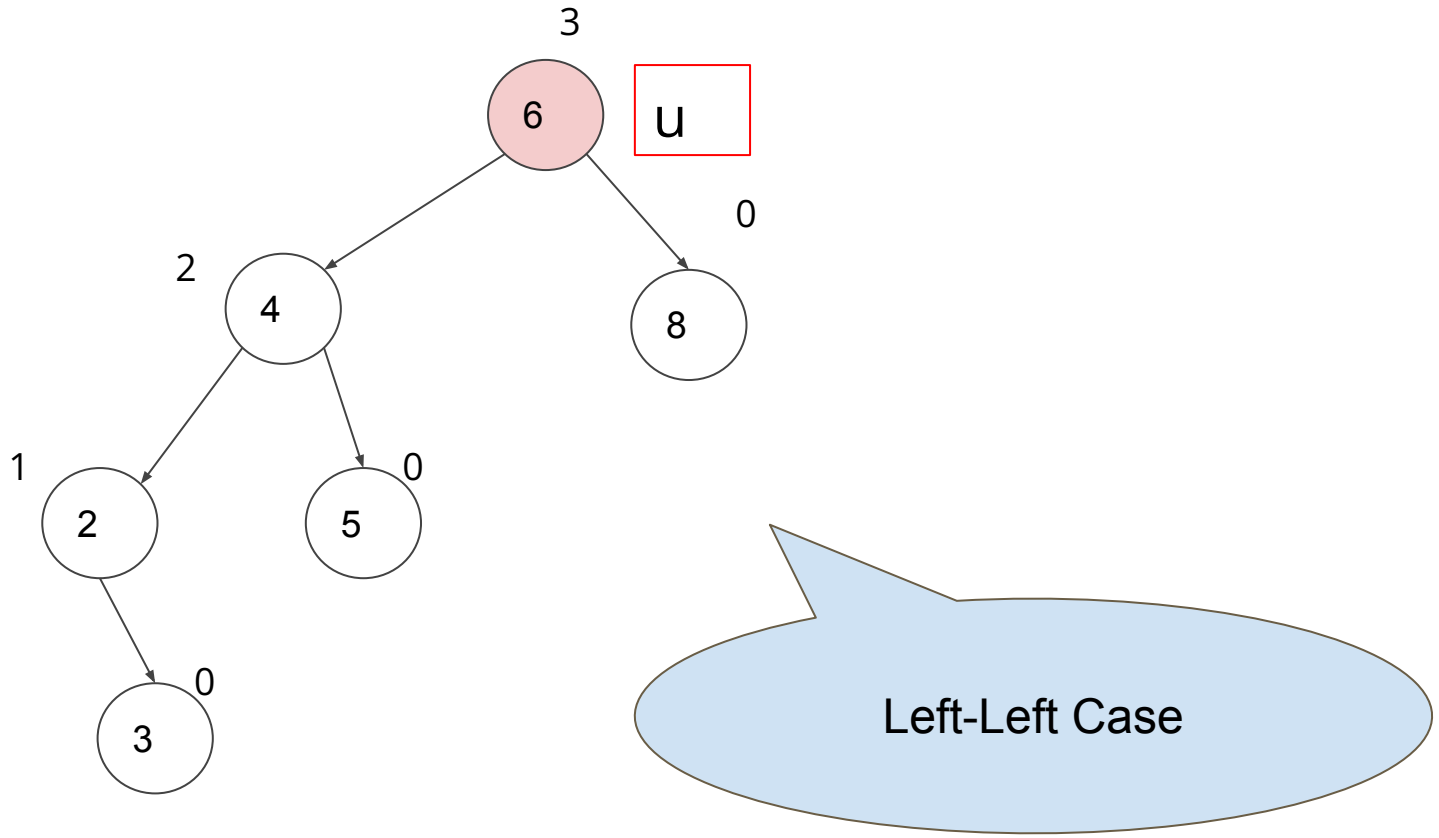
Inserting: 3



Left-Left Case or
Left-Right Case?

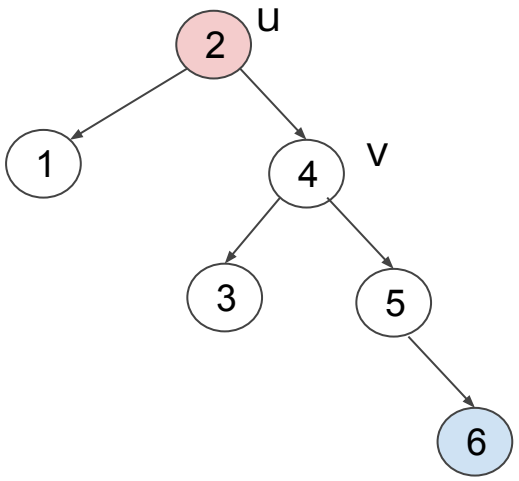
AVL Tree : Insertion

Inserting: 3



AVL Tree : Insertion

Single Rotation



- Left Rotation :

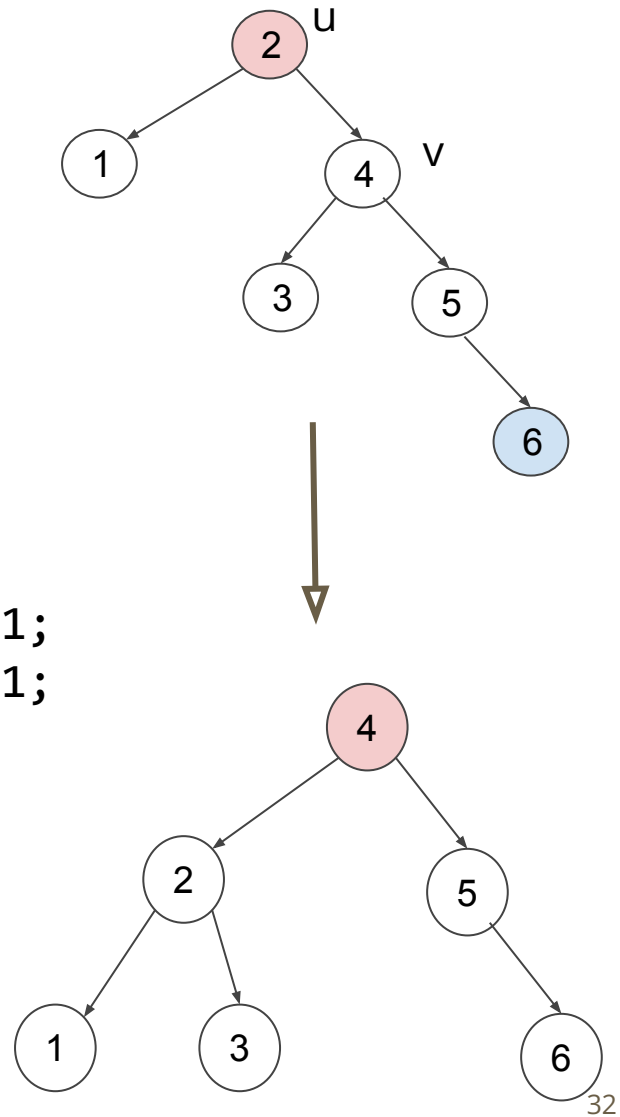
```
void LeftRotation(Node *&u)
{
    Node *v = u->right;
    u->right = v->left;
    v->left = u;
    u->height = max(h(u->right), h(u->left))+1;
    v->height = max(h(v->right), h(v->left))+1;
    u=v;
}
```

AVL Tree : Insertion

Single Rotation

- Left Rotation :

```
void LeftRotation(Node *&u)
{
    Node *v = u->right;
    u->right = v->left;
    v->left = u;
    u->height = max(h(u->right), h(u->left))+1;
    v->height = max(h(v->right), h(v->left))+1;
    u=v;
}
```

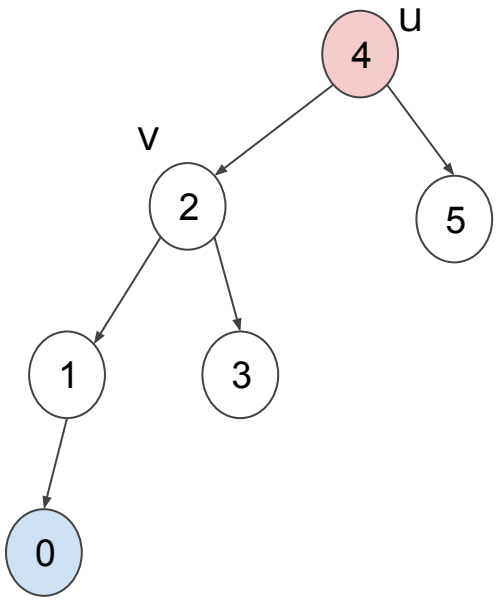


AVL Tree : Insertion

Single Rotation

- Right Rotation :

```
void RightRotation(Node *&u)
{
    Node *v = u->left;
    u->left = v->right;
    v->right = u;
    u->height = max(h(u->right), h(u->left))+1;
    v->height = max(h(v->right), h(v->left))+1;
    u=v;
}
```

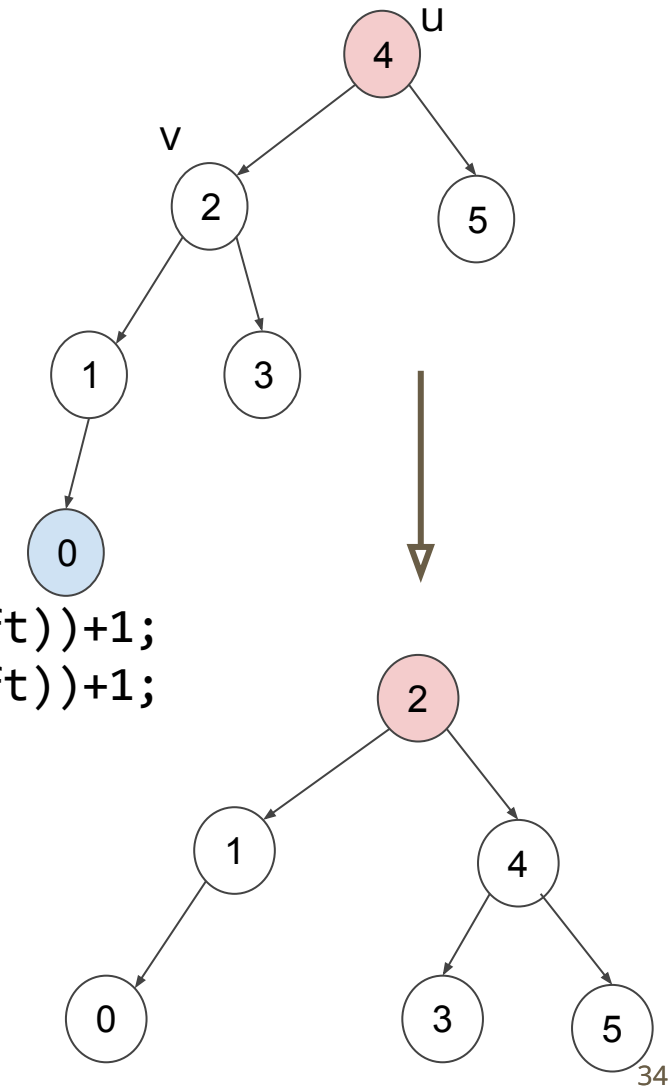


AVL Tree : Insertion

Single Rotation

- Right Rotation :

```
void RightRotation(Node *&u)
{
    Node *v = u->left;
    u->left = v->right;
    v->right = u;
    u->height = max(h(u->right), h(u->left))+1;
    v->height = max(h(v->right), h(v->left))+1;
    u=v;
}
```



AVL Tree : Insertion

Double Rotation

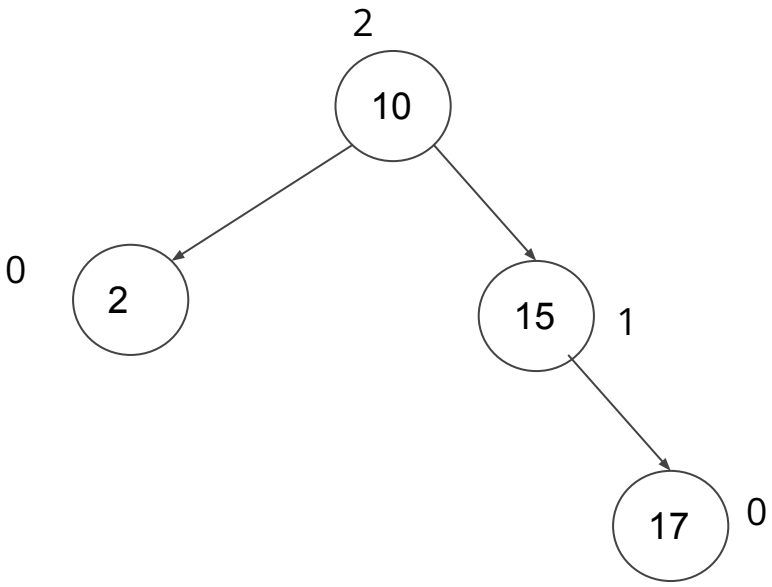
- Right-Left Rotation :
  Right rotate the child node of the unbalanced node and Left rotate the unbalanced node.

- Left-Right Rotation:
  Left rotate the child node of the unbalanced node and Right rotate the unbalanced node.

- left - right insertion : Left-Right Rotation
- right-left insertion: Right-Left Rotation

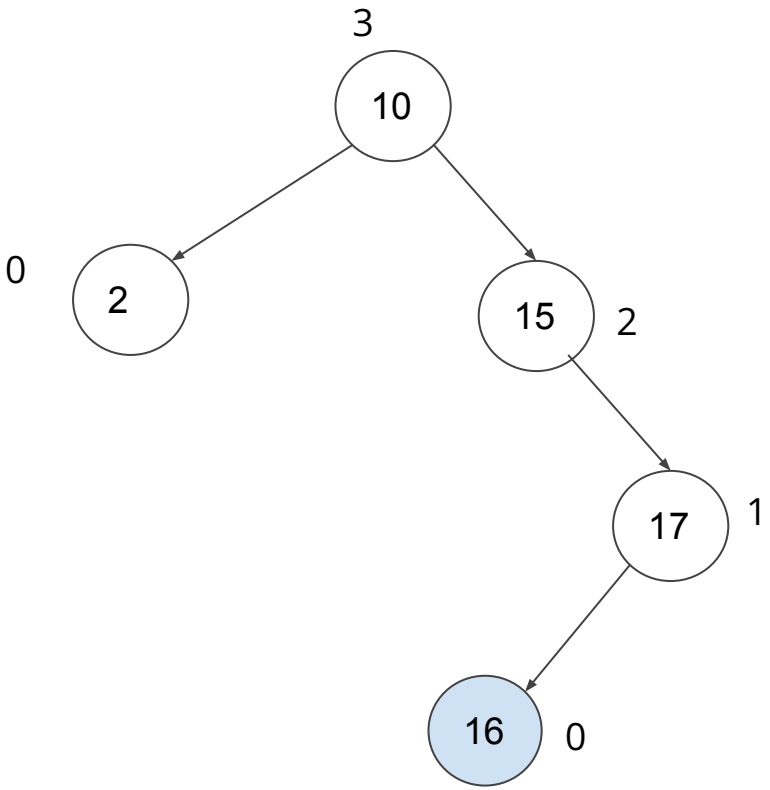
AVL Tree : Insertion

Insert 16



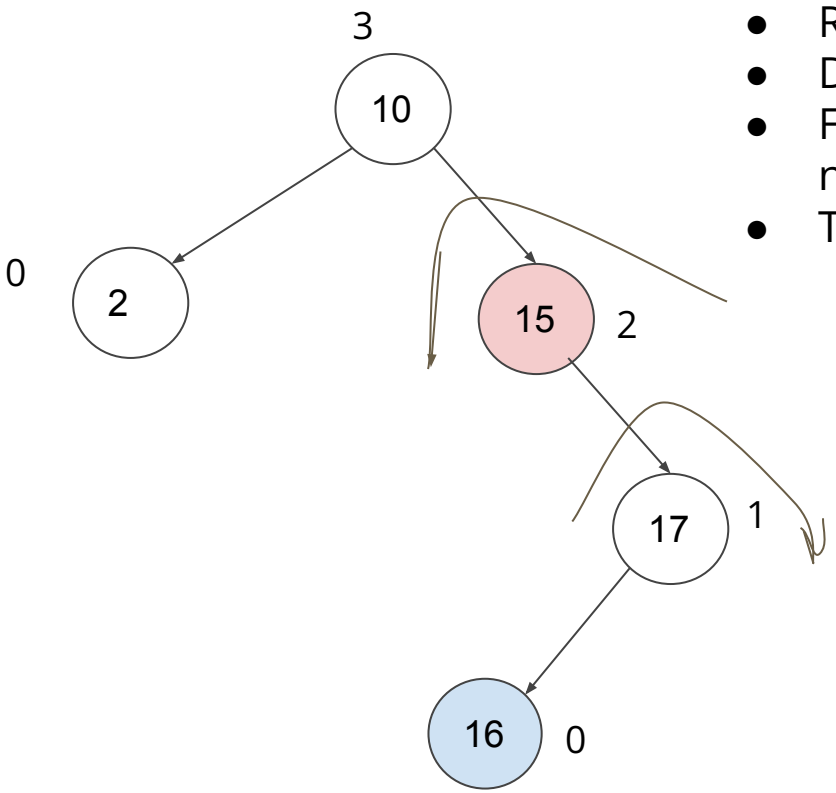
AVL Tree : Insertion

Insert 16



AVL Tree : Insertion

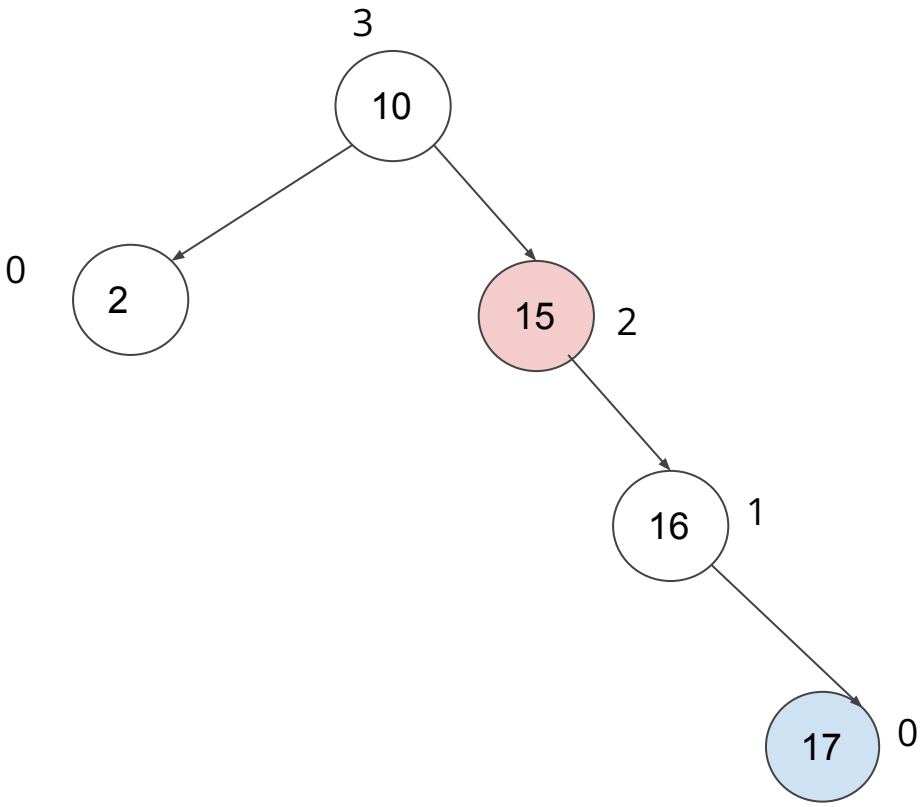
Insert 16



- u = node with value 15
- Right Left case
- Double Rotation
- First Rotate the child node of u .
- Then rotate u.

AVL Tree : Insertion

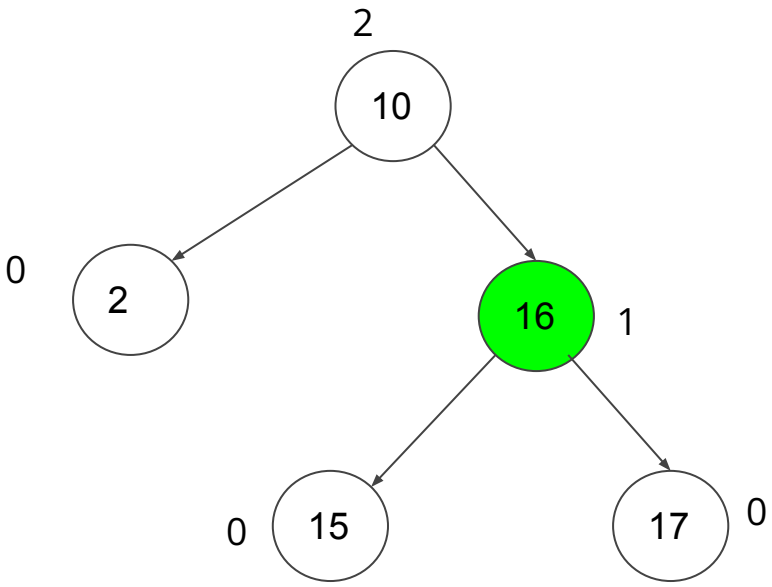
Insert 16



AVL Tree : Insertion

After Insertion of 16

After Double rotation

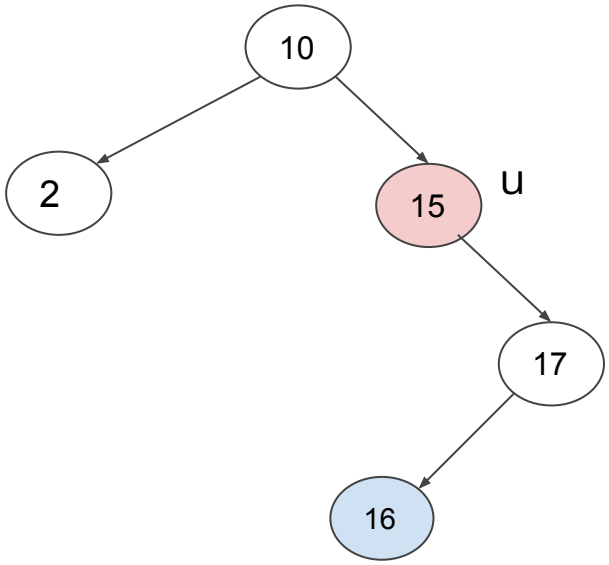


AVL Tree : Insertion

Double Rotation

- Right-Left Rotation :

```
void RightLeftRotation(Node *&u)
{
    RightRotation(u->right);
    LeftRotation(u);
}
```

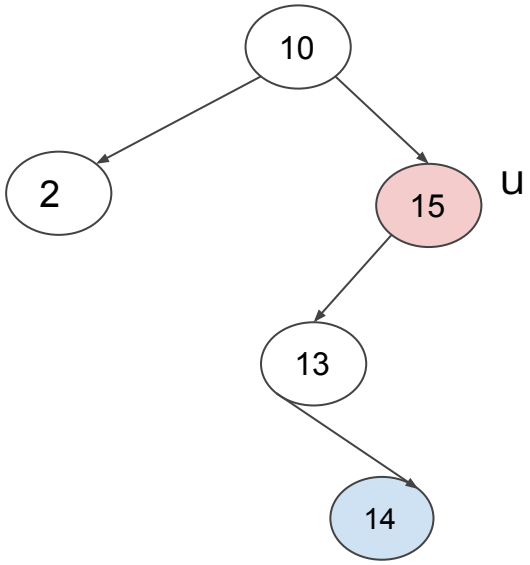


AVL Tree : Insertion

Double Rotation

- Left-Right Rotation :

```
void LeftRightRotation(Node *&u)
{
    LeftRotation(u->left);
    RightRotation(u);
}
```



AVL Tree : Insertion

```
void insert(int x, Node *&u <-root)
{
   if( u == NULL )
       u = new Node( x, NULL, NULL );
   else if( x < u->element )
       insert( x, u->left );
   else if( u->element < x )
       insert( x, u->right );

   balance( u );
}
```

AVL Tree : Insertion

```
void balance( Node *&u )
{
    if( u == NULL )
       return;

    if( h( u->left ) - h( u->right ) > 1 )
        if( h( u->left->left ) >= h( u->left->right ) )
            RightRotation( u );
        else
            LeftRightRotation( u );
    else if( h( u->right ) - h( u->left ) > 1 )
        if( h( u->right->right ) >= h( u->right->left ) )
            LeftRotation( u );
        else
            RightLeftRotation( u );

    u->height = max( h( u->left ), h( u->right ) ) + 1;
}
```

Left-Left case or Left-Right

Right-Right case or Right-Left

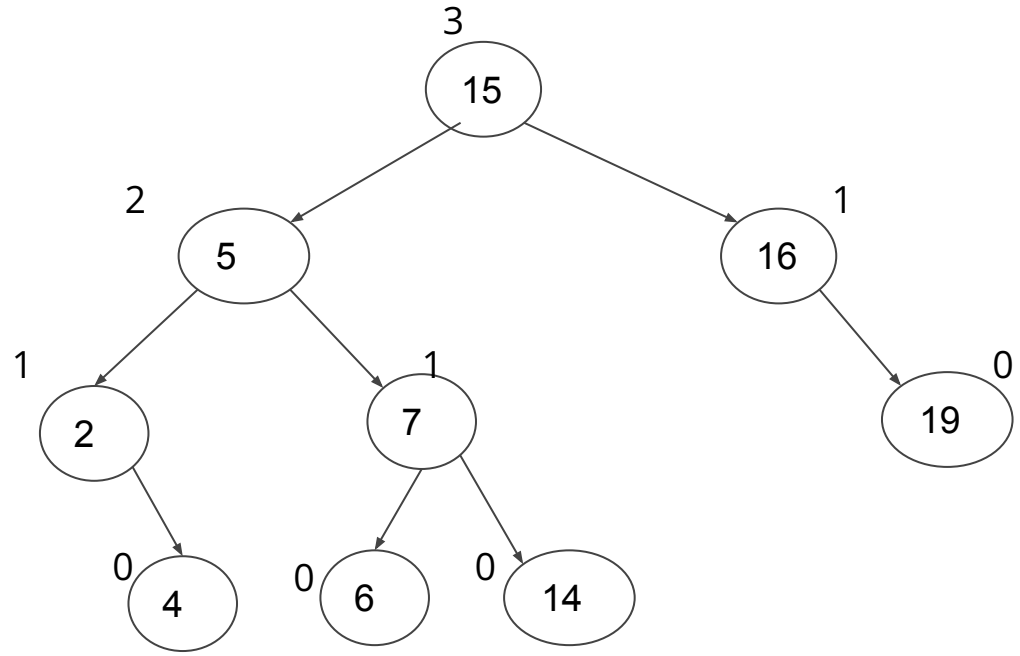
AVL Tree Insertion

Insert the following items in an AVL tree.
Illustrate each step.

16	14	15	6	7	19	5	2	4

AVL Tree Insertion

The Final AVL tree:



16	14	15	6	7	19	5	2	4

AVL Tree : Deletion

Steps:

- Delete using BST property (3 Cases)
- Balance the tree

AVL Tree : Deletion

Deletion in BST (3 Cases)

- Node with no children (Leaf Node):

  Simply remove the node from the tree.

- Node with one child:

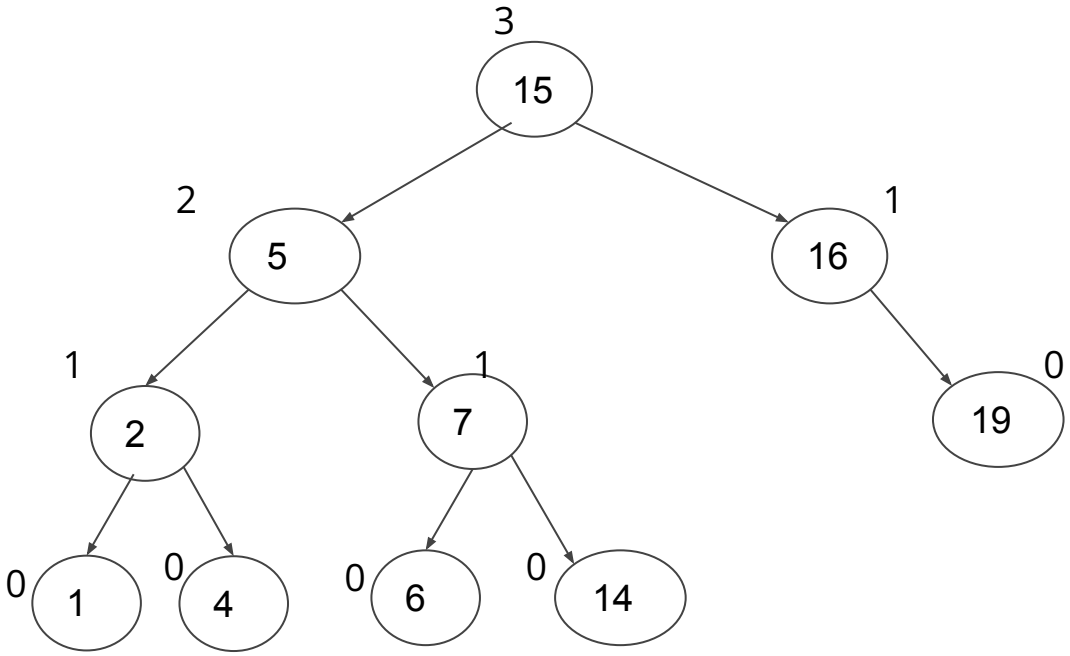
  Connect the parent of the node to the child of the node to be deleted. Delete the node.

- Node with two children:
  - Replace the node to be deleted with either its in-order predecessor or in-order successor. The in-order predecessor is the maximum value node in the left sub-tree, and the in-order successor is the minimum value node in the right subtree.
  - After replacing the node, delete the predecessor or successor, which will have at most one child.

AVL Tree : Deletion

```
void remove( int x, Node *&u )
 {
     if( u == NULL )
        return; // Item not found; do nothing

     if( x < u->element )
          remove( x, u->left );
     else if( u->element < x )
          remove( x, u->right );
     else if( u->left != NULL && u->right != NULL ) // Two children
     {
          u->element = findMin( u->right )->element;
          remove( u->element, u->right );
     }
     else // one child or no child
     {
          Node *oldNode = u;
          u = ( u->left != NULL ) ? u->left : u->right;
          delete oldNode;
     }
     balance( u );
 }
```

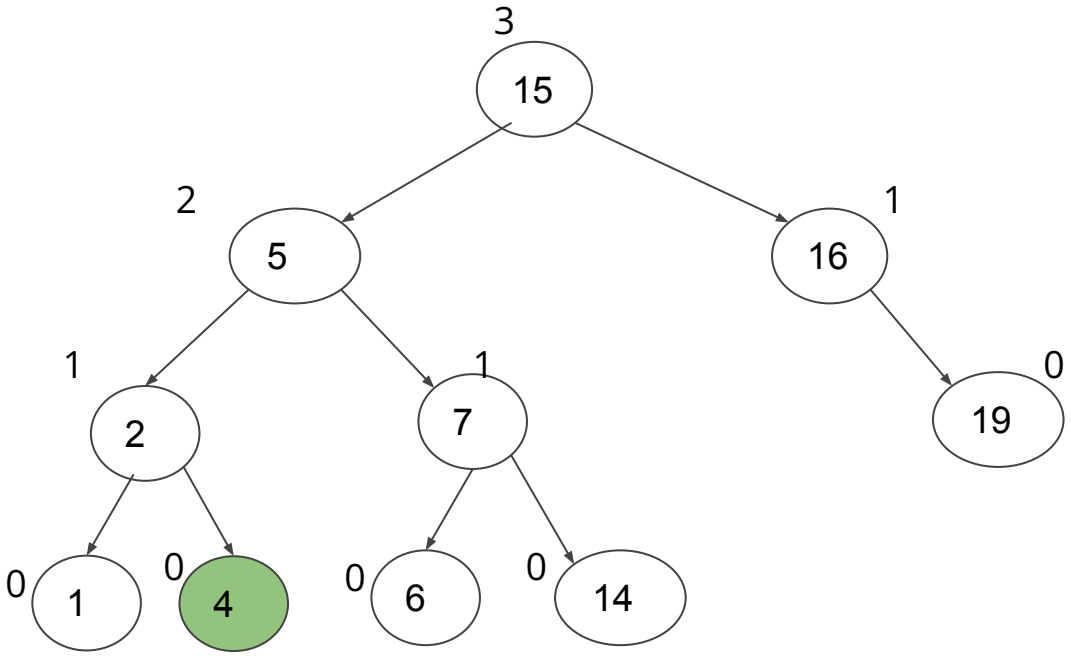
AVL Tree: Deletion



AVL Tree: Deletion

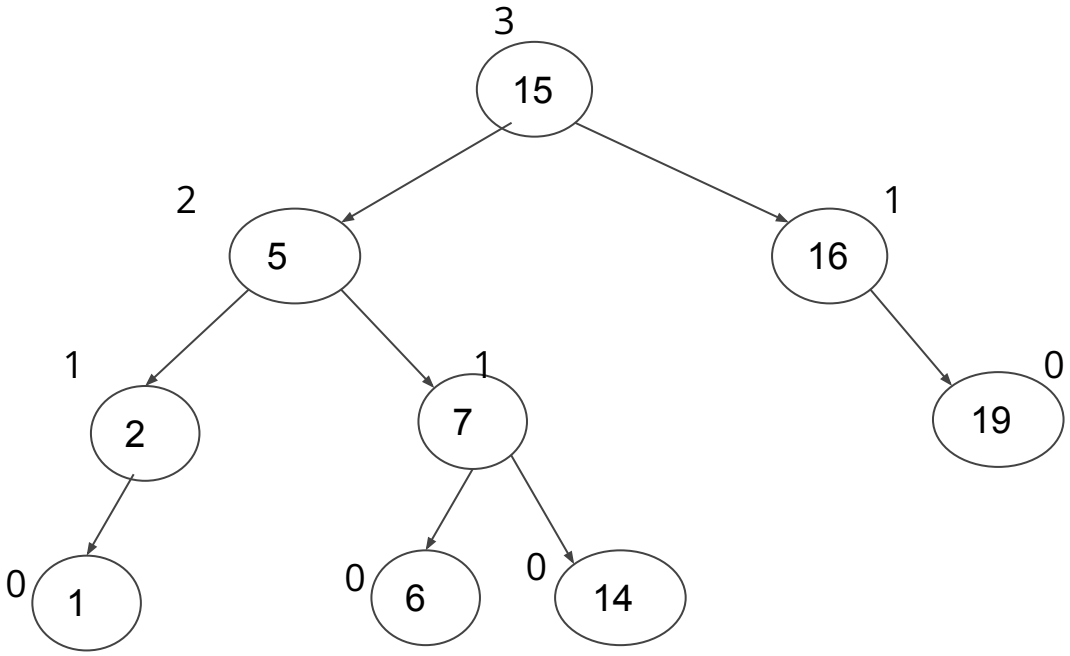
Delete 4

4 is a leaf node



AVL Tree: Deletion

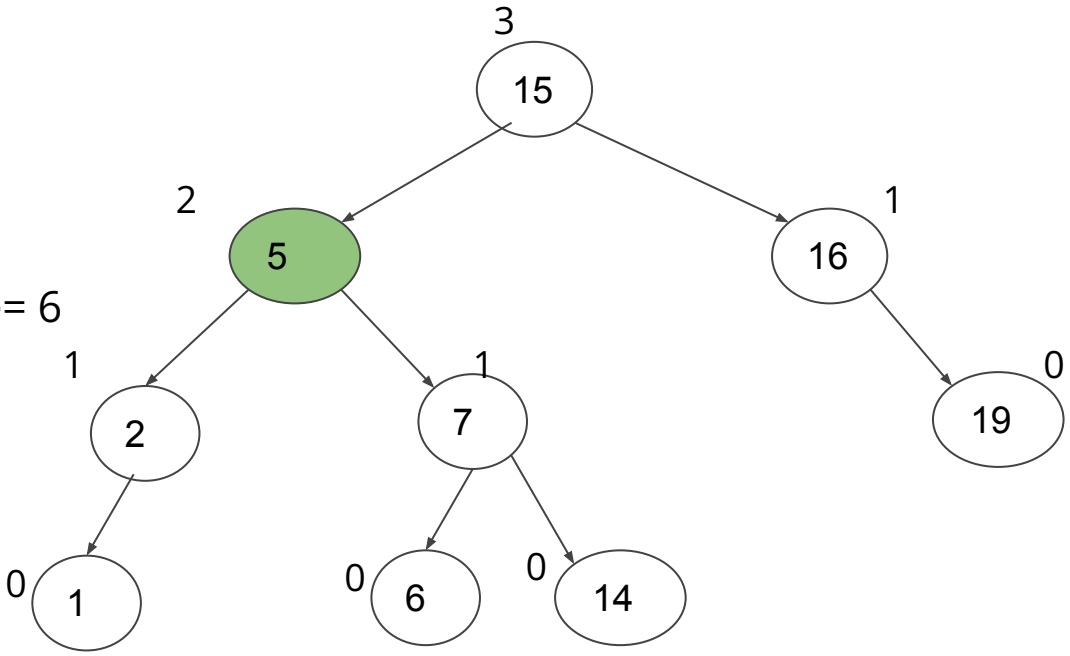
Tree is balanced



AVL Tree: Deletion

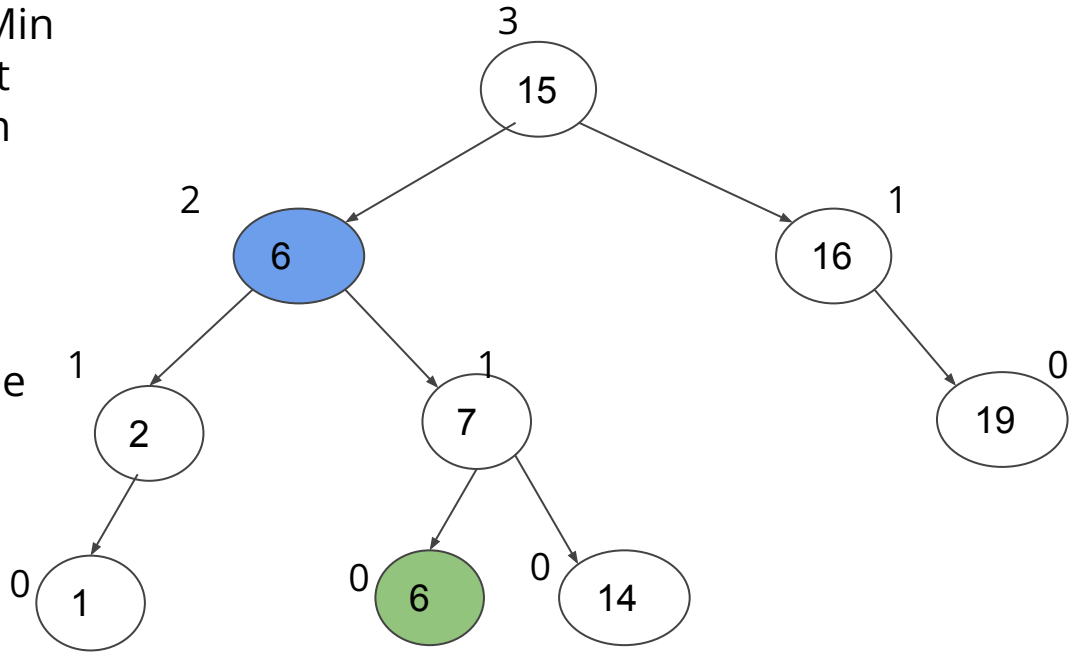
Delete 5

- Two children
- Min value of right subtree= 6



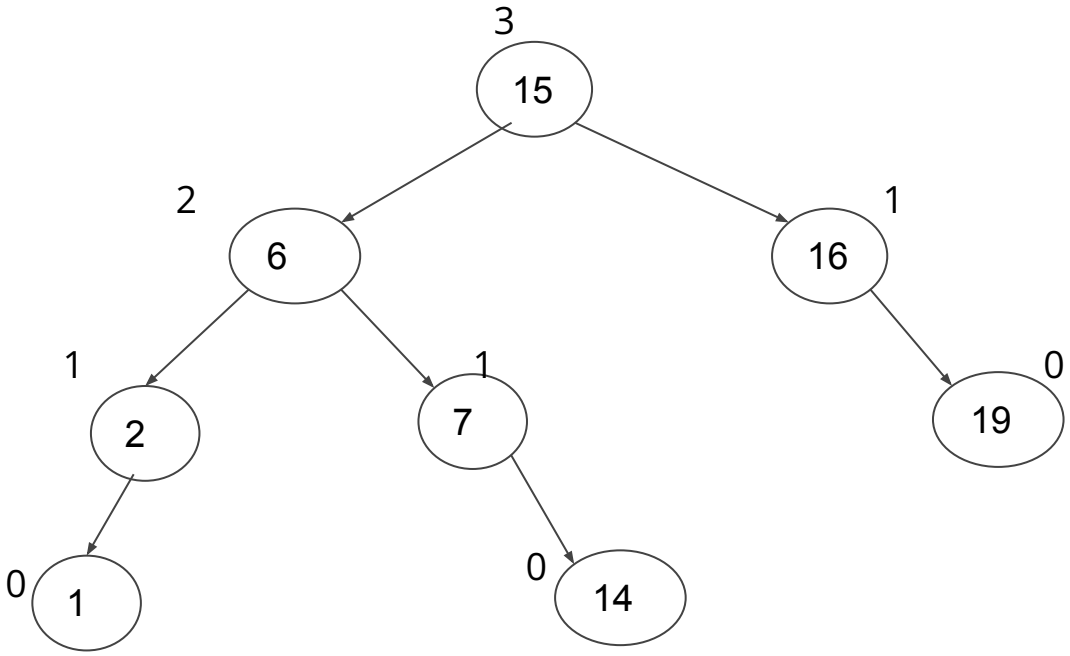
AVL Tree: Deletion

- Putting the Min value of right subtree (6) in place of 5
- Remove 6 from the right subtree



AVL Tree: Deletion

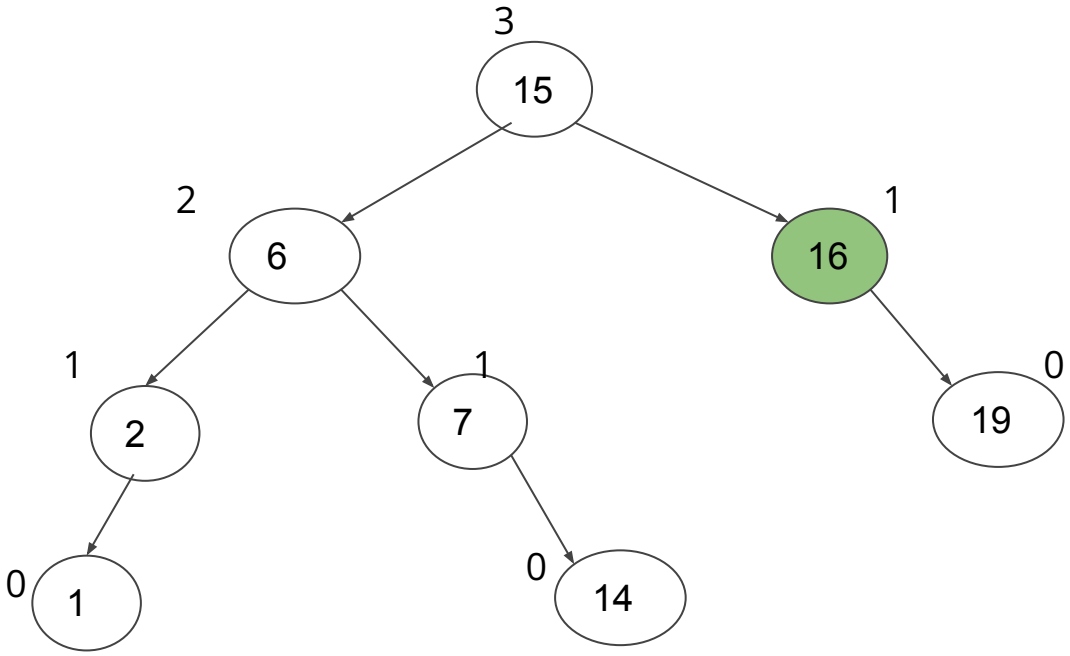
Tree is balanced



AVL Tree: Deletion

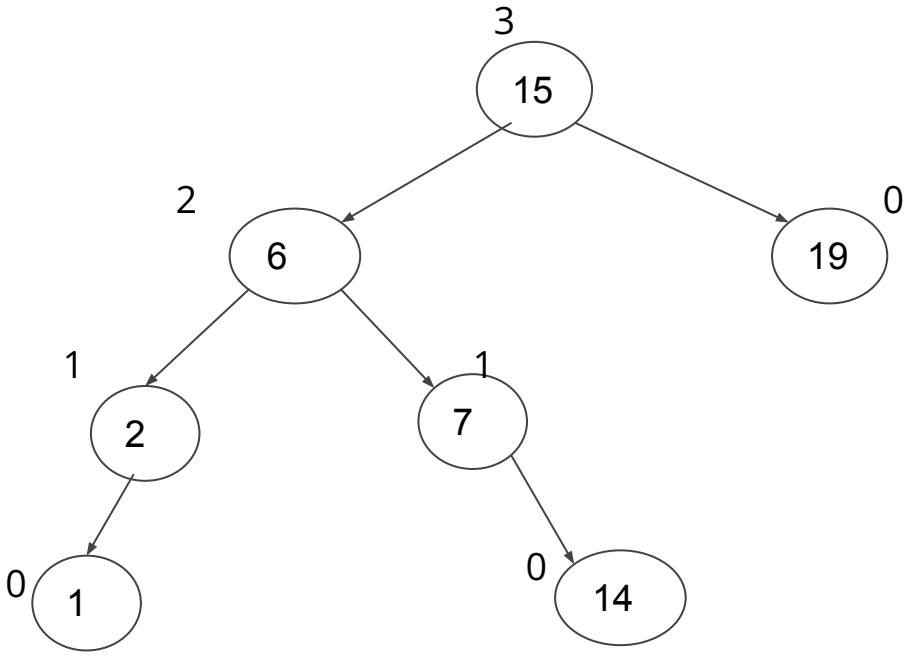
Delete 16

- One Child
- Make 19 the child of 15



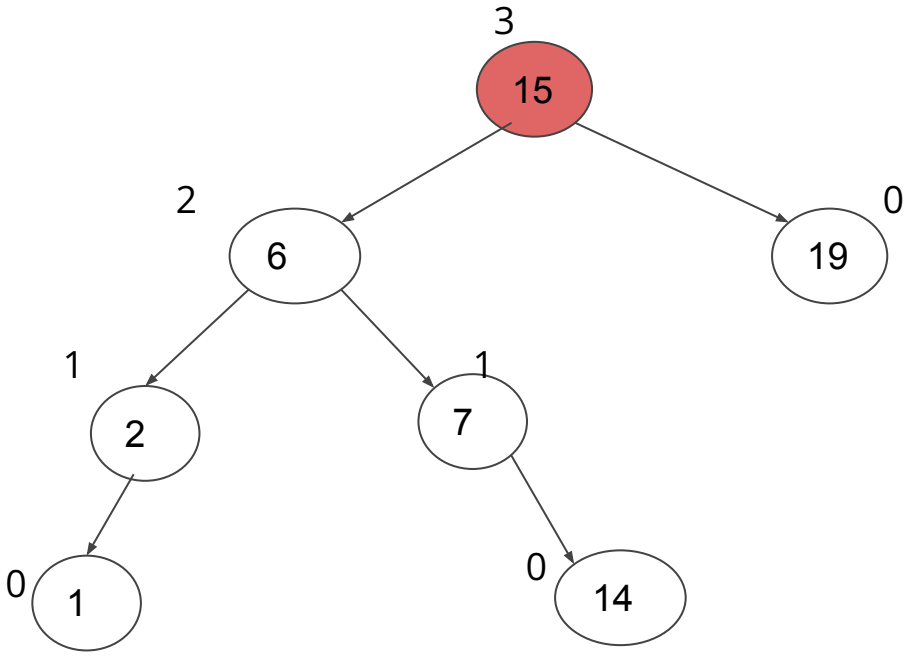
AVL Tree: Deletion

Is the tree balanced?



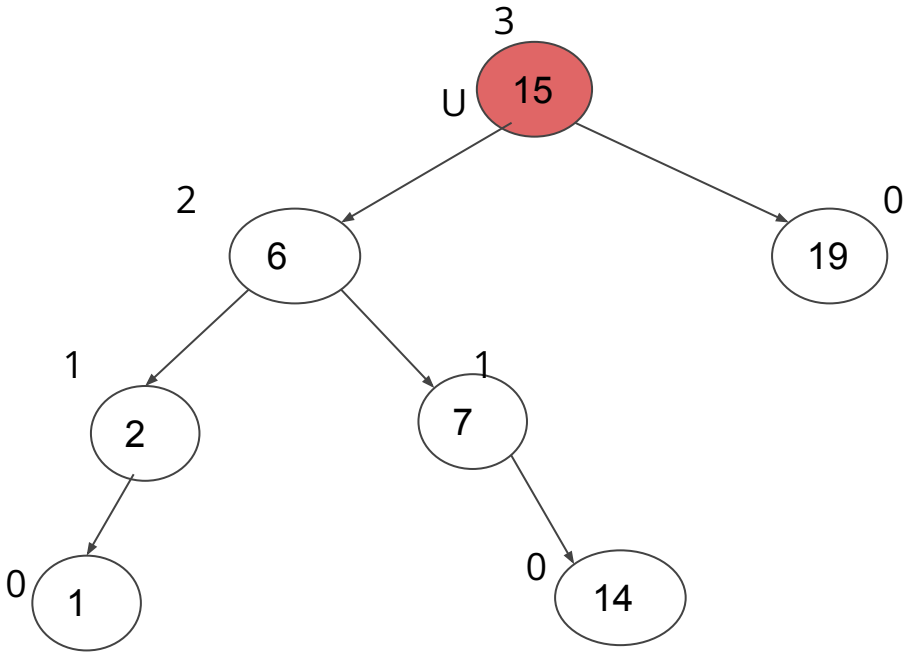
AVL Tree: Deletion

The tree is not balanced.



AVL Tree: Deletion

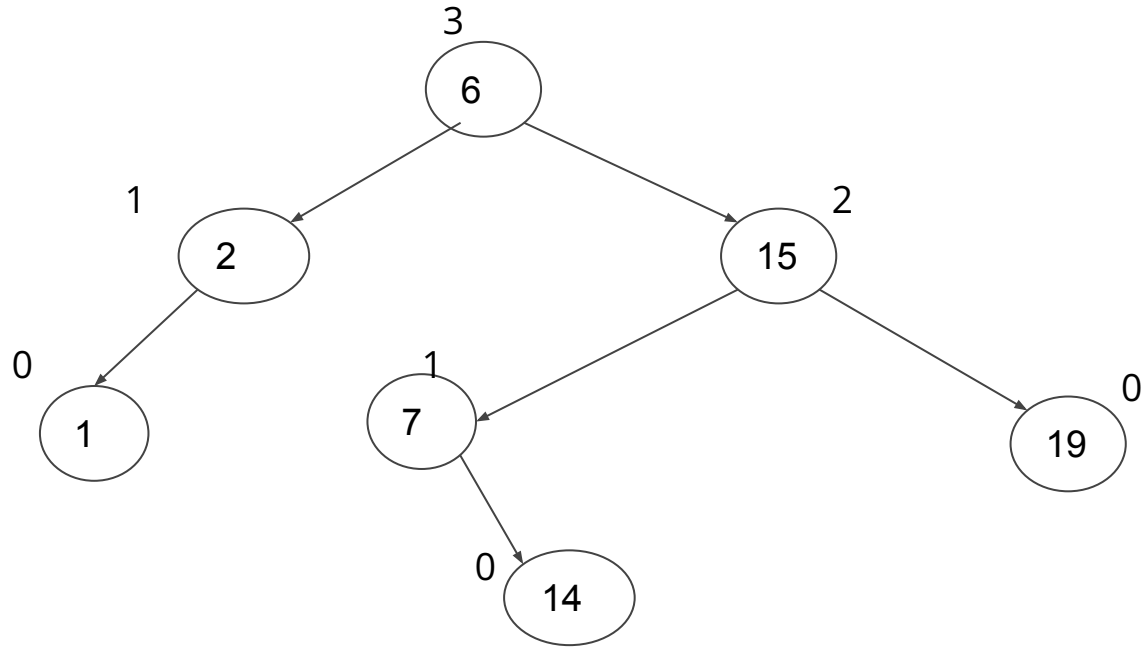
h(2) = h(7)
Left-Left case
Right Rotation




```
if( h( u->left ) - h( u->right ) > 1 )
    if( h( u->left->left ) >= h( u->left->right ) )
        RightRotation( u );
    else
        LeftRightRotation( u );
```

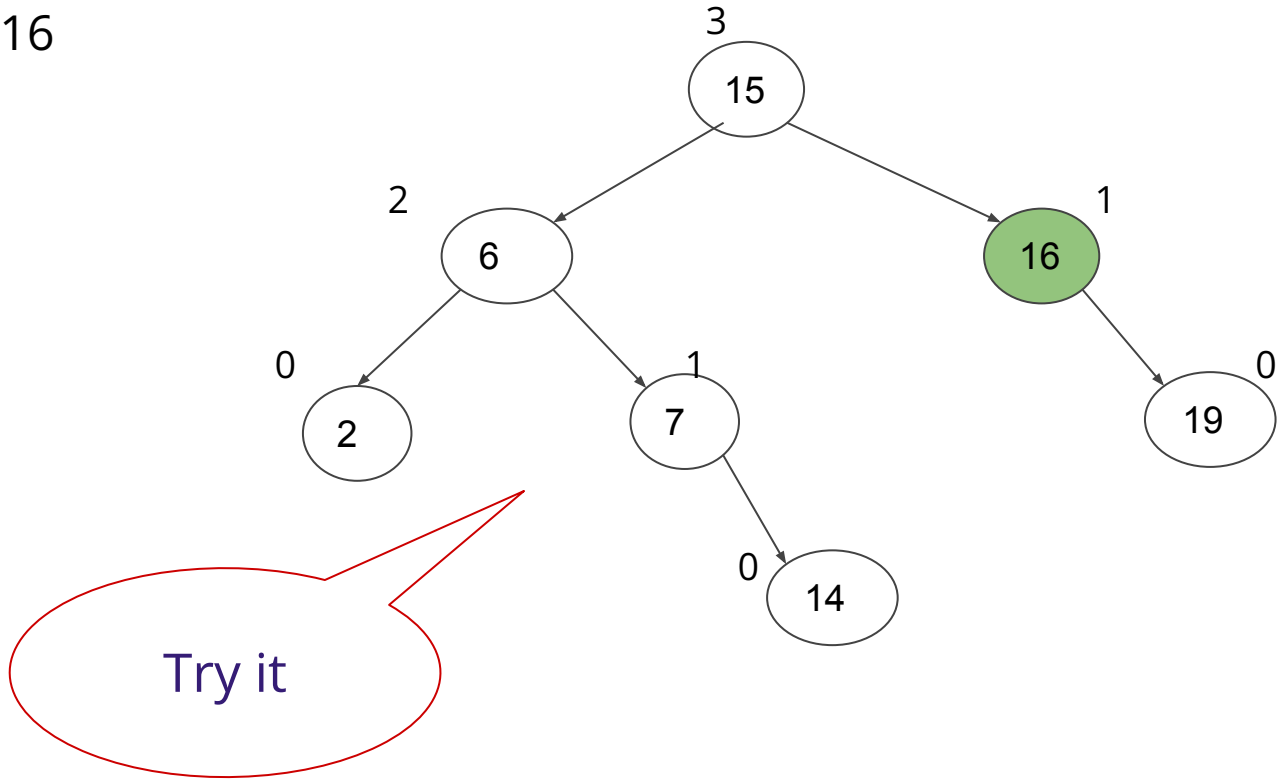
AVL Tree: Deletion

Tree balanced



AVL Tree: Deletion

Delete 16



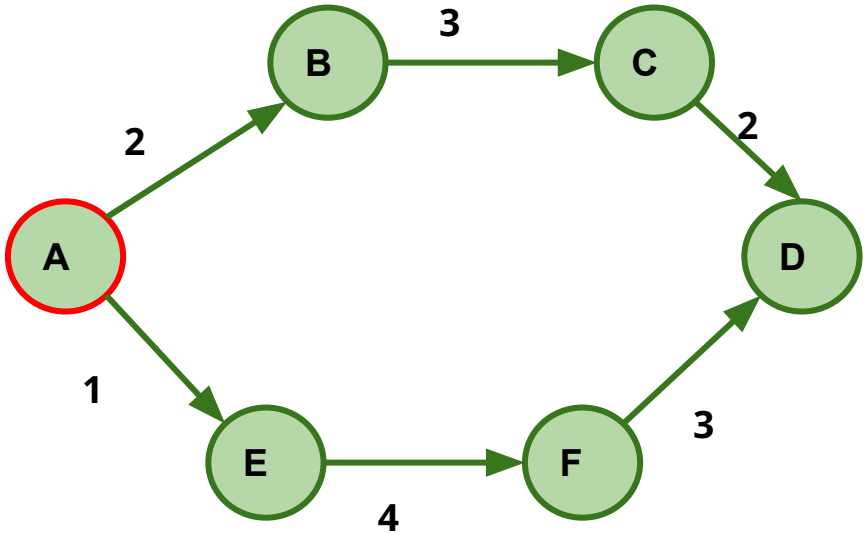

thank
you

CSE 215: Data Structures & Algorithms II

Single Source Shortest Paths

Lec Sumaiya Nuha Mustafina
Dept Of CSE
sumaiyamustafina@cse.mist.ac.bd

Single Source Shortest Paths



Source = **A**

All possible paths:
A->B
A->B->C
A->B->C->D
A->E
A->E->F
A->E->F->D

Minimum distance from A to D is 2+3+2 = 7.
So shortest path from A to D : A->B->C->D

Shortest Path Problem

A Shortest Path Problem includes :

- A directed graph G = (V,E)
- Weight (w) associated with each edge
- Weight associated with each path

If path p = v0 -> v2 -> ... -> vk, then the weight of path p is

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i) .$$

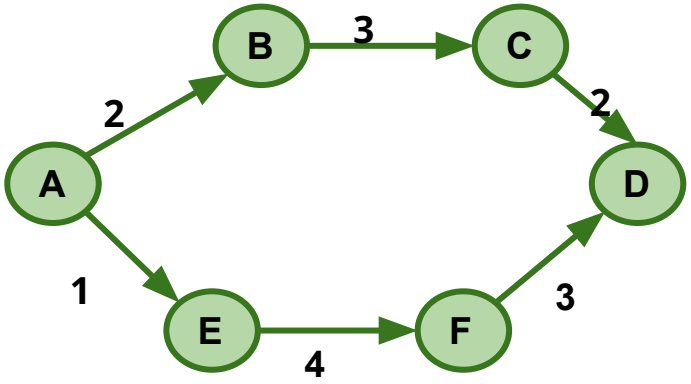
Weight of Path

The weight of path p = < v0 -> v2 -> ... -> vk > is

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i) .$$

If, p = < A -> B -> C-> D>
w(p) = w (A,B) + w(B,C) +w(C,D)
= 2+3+2 =7

If, p = <A->E->F->D>
w(p) = w (A,E) + w(E,F) +w(F,D)
= 1+4+3 =8



Shortest Path Weight

The ***shortest-path weight*** δ (u, v) from u to v is defined by

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v\} & \text{if there is a path from } u \text{ to } v\,, \\ \infty & \text{otherwise}\,. \end{cases}$$

A ***shortest path*** from vertex u to vertex v is then defined as any path p with weight δ (u, v)

Single-source shortest-paths problem : Variants

Single-source shortest-paths problem:
Given a graph G (V, E), we want to find a shortest path from a given source vertex s ∈ V to each vertex v ∈ V .

Variants:

- ***Single-destination shortest-paths problem***: Find a shortest path to a given destination vertex t from each vertex .
- ***Single-pair shortest-path problem***: Find a shortest path from u to v for given vertices u and v .
- ***All-pairs shortest-paths problem***: Find a shortest path from u to v for every pair of vertices u and v.

Optimal substructure property of shortest path

Shortest-paths algorithms typically rely on the property that a shortest path between two vertices contains other shortest paths within it.

*********Optimal substructure is one of the key indicators that dynamic programming and the greedy method might be applied.*

Optimal substructure property of shortest path

Subpaths of shortest paths are shortest paths

Given a weighted, directed graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$, let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be a shortest path from vertex v_0 to vertex v_k and, for any i and j such that $0 \le i \le j \le k$, let $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ be the subpath of p from vertex v_i to vertex v_j. Then, p_{ij} is a shortest path from v_i to v_j.

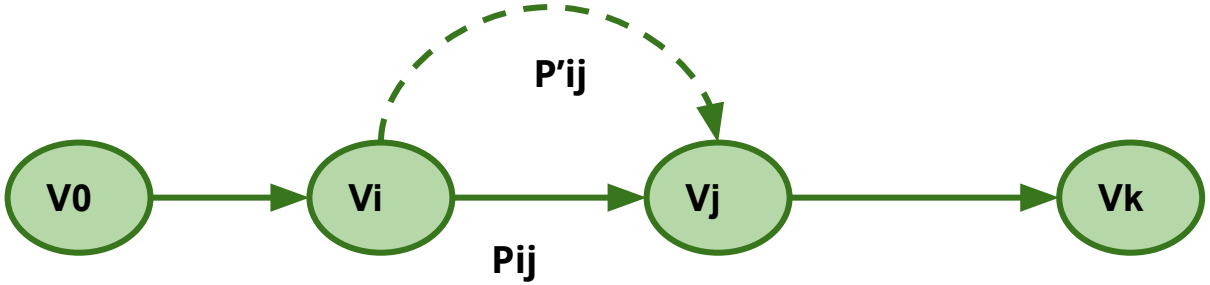
Proof!!!!

Optimal substructure property of shortest path

Subpaths of shortest paths are shortest paths

Given a weighted, directed graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$, let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be a shortest path from vertex v_0 to vertex v_k and, for any i and j such that $0 \le i \le j \le k$, let $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ be the subpath of p from vertex v_i to vertex v_j. Then, p_{ij} is a shortest path from v_i to v_j.

Proof!!!!

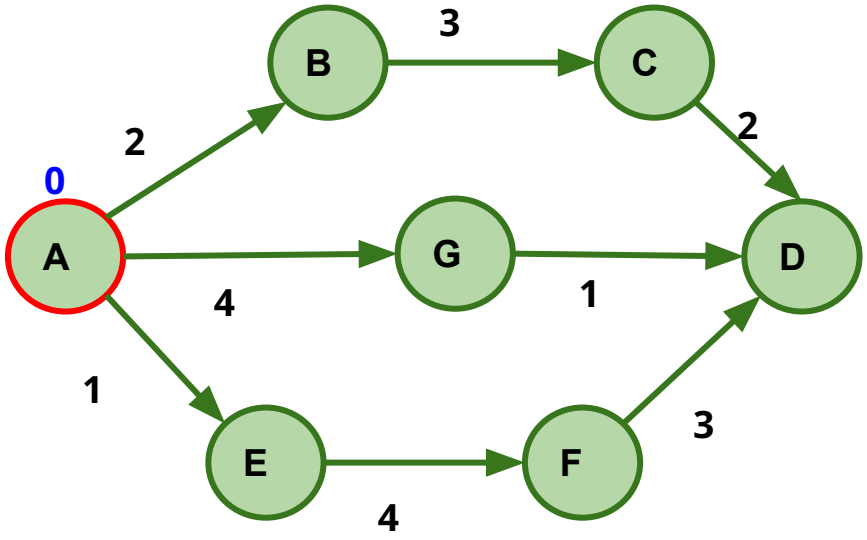


Single-source shortest-paths problem : Algorithms

- ***Dijkstra Algorithm (Greedy Method)***
  O((V+E)log V) using Binary Heap as priority queue

- ***Bellman Ford Algorithm (Dynamic Problem)***
  O(VE)

Shortest Path Problem Concept



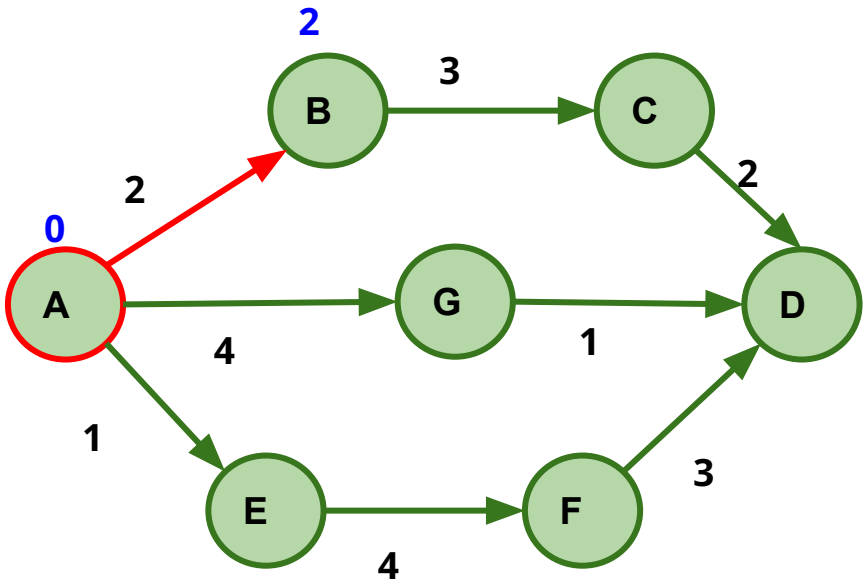
d[v] : Distance from source to vertex v

d[v] : shortest path estimate

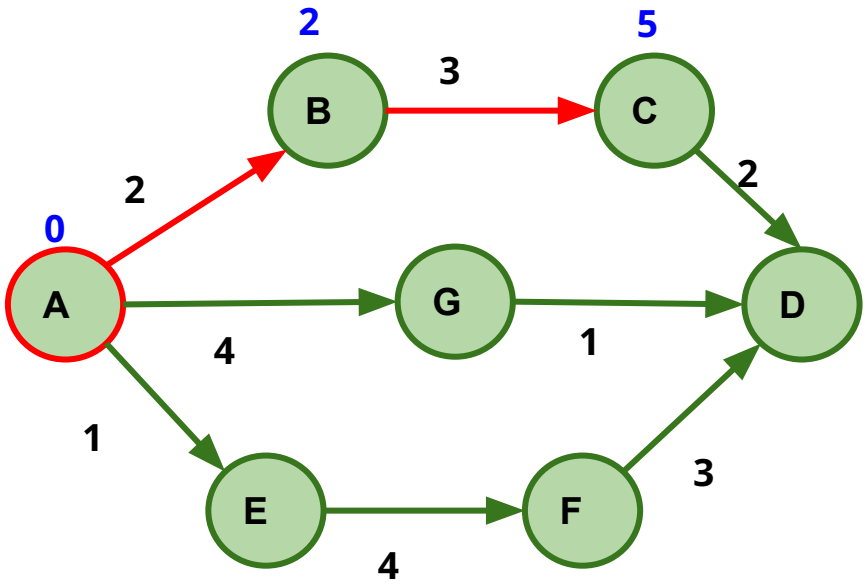
d[v] >= δ (s, v)

If d[v] = δ (s, v), it never changes

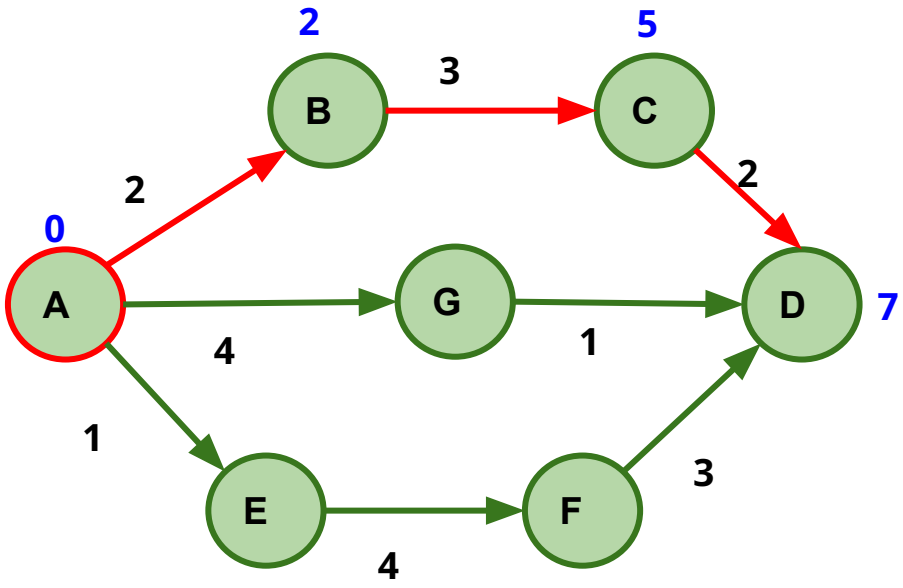
Shortest Path Problem Concept



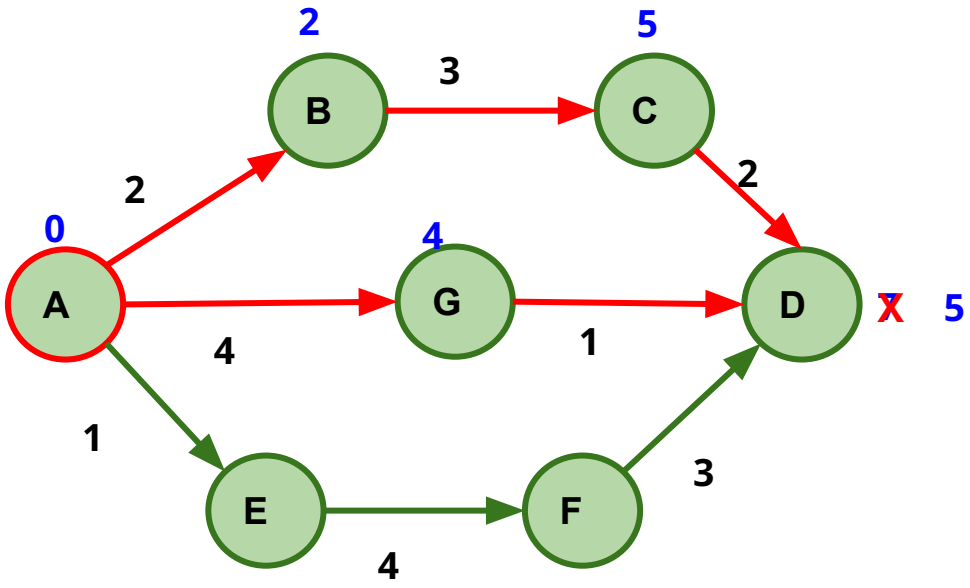
Shortest Path Problem Concept



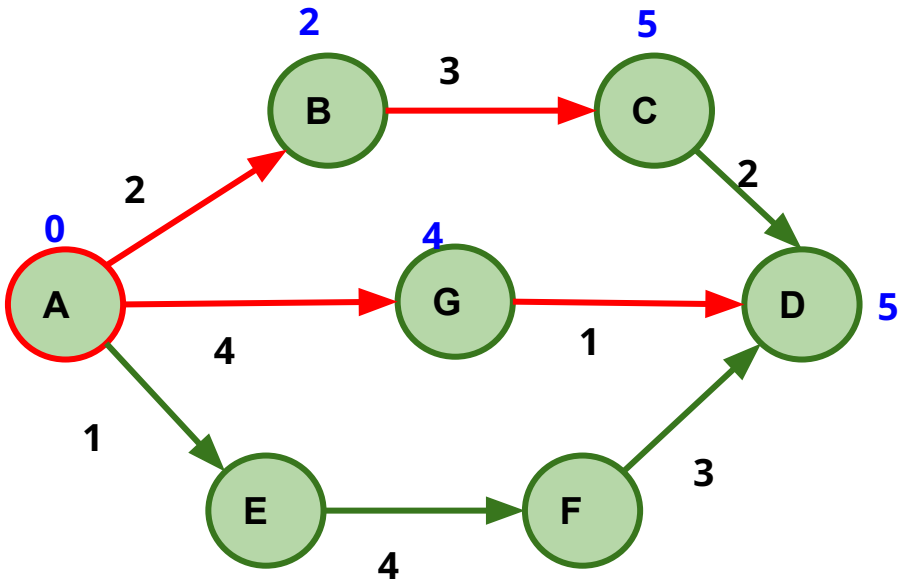
Shortest Path Problem Concept



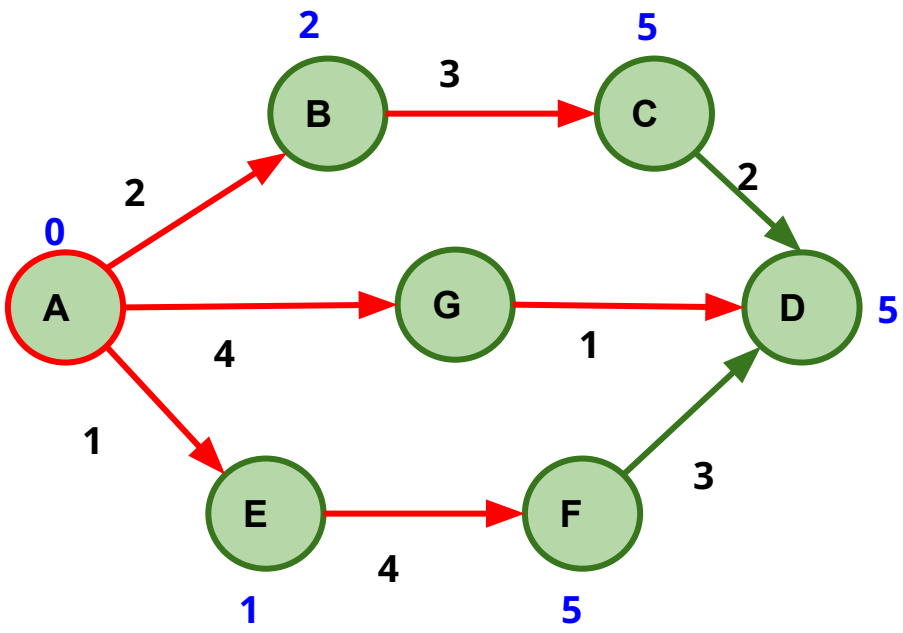
Shortest Path Problem Concept



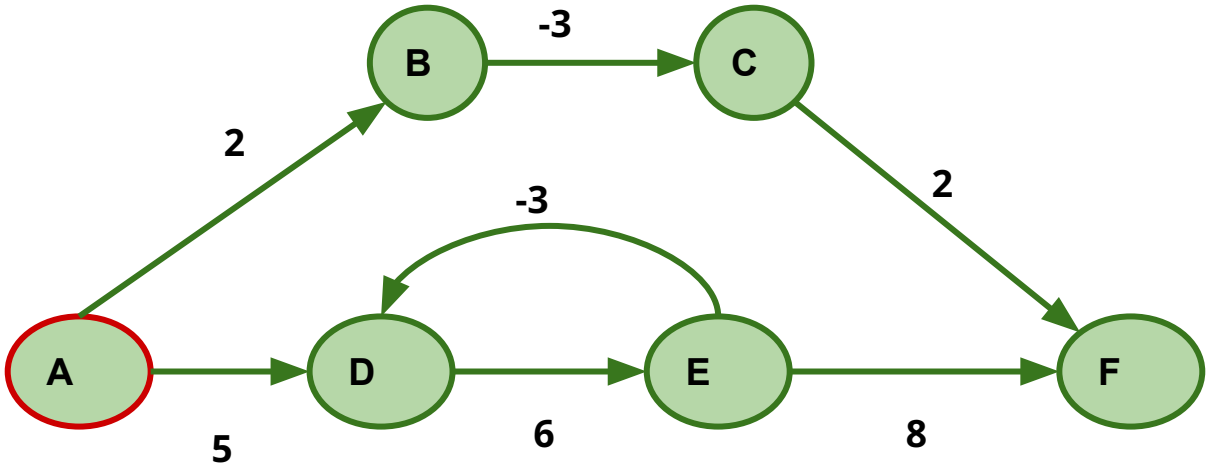
Shortest Path Problem Concept



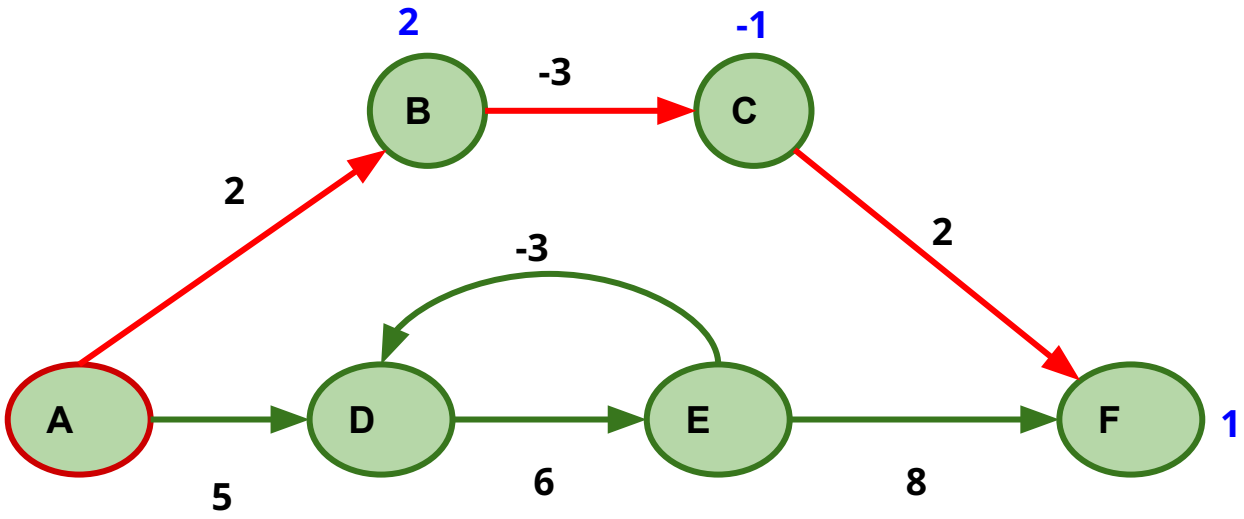
Shortest Path Problem Concept



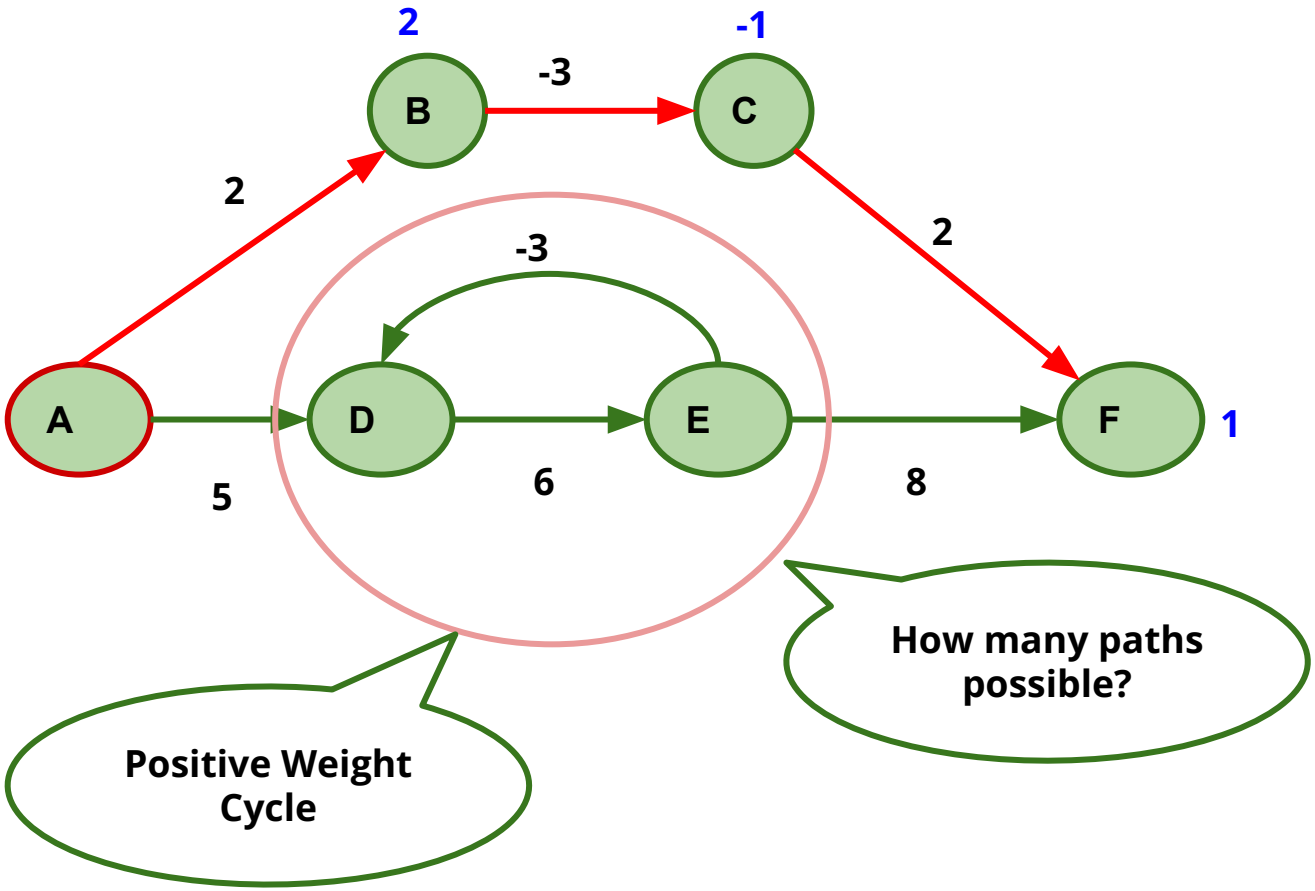
Negative Weight Edge



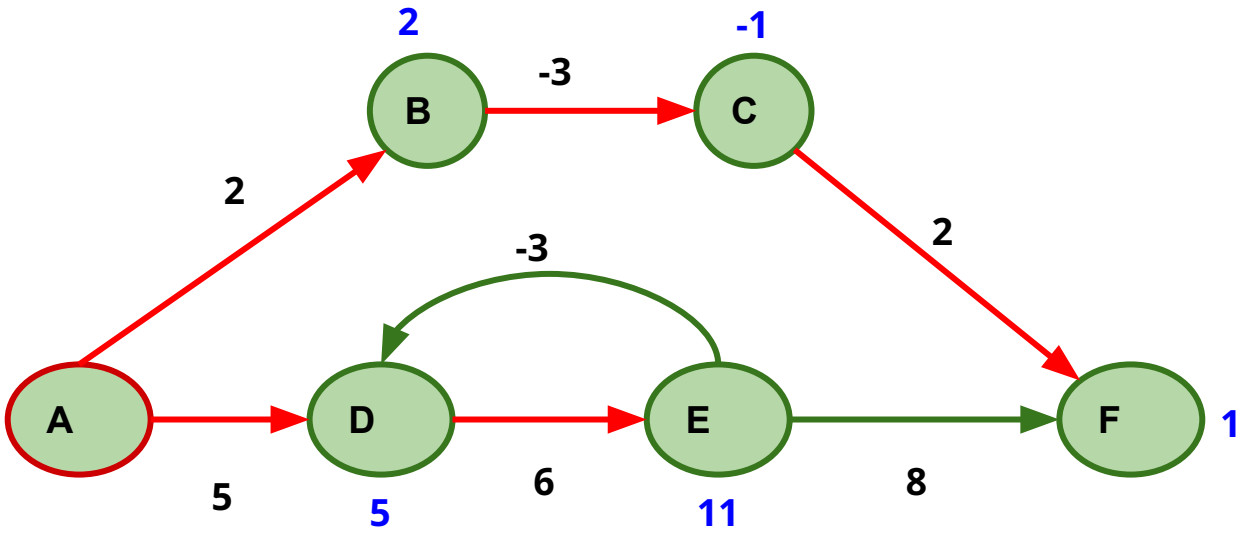
Negative Weight Edge



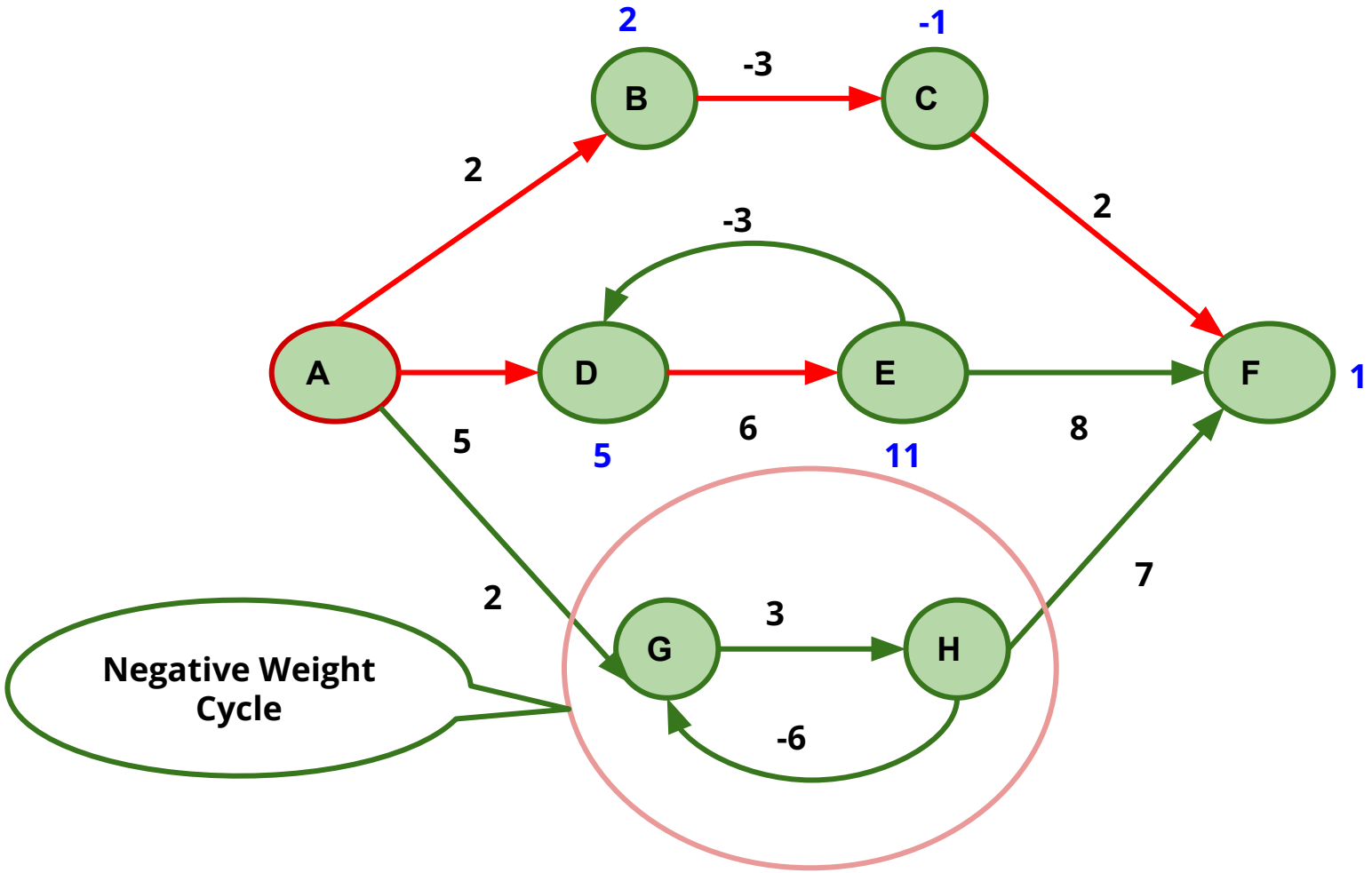
Negative Weight Edge



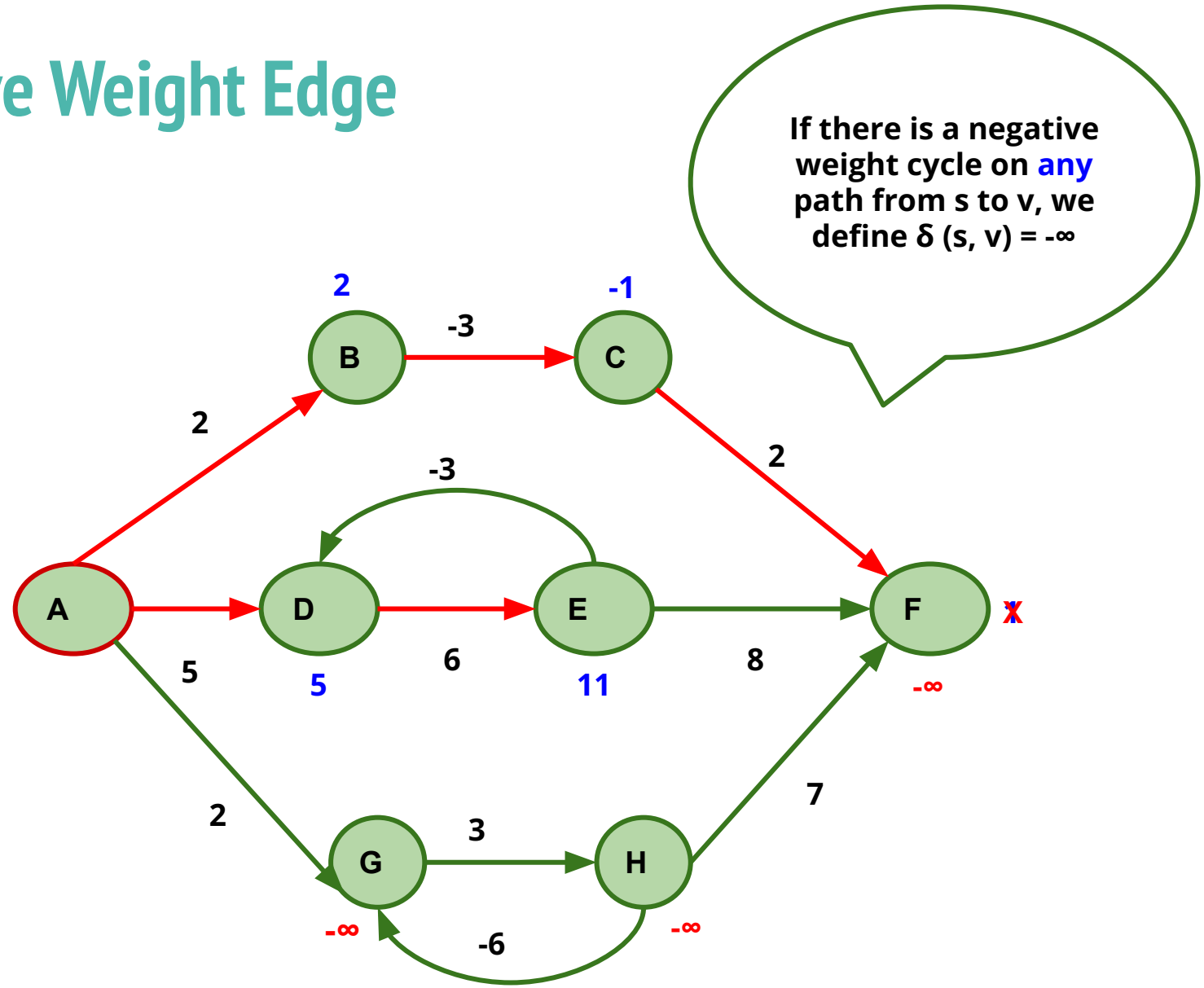
Negative Weight Edge



Negative Weight Edge



Negative Weight Edge



Negative Weight Edge

- If the graph G=(V, E) contains no negative weight cycles reachable from the source s, then for all vertex v , the shortest-path weight δ (s, v) remains well defined, even if it has a negative value.

- If the graph contains a negative-weight cycle reachable from s, then, shortest-path weights are not well defined.

- If there is a negative weight cycle on any path from s to v, then δ (s, v) = $-\infty$

Negative Weight Edge

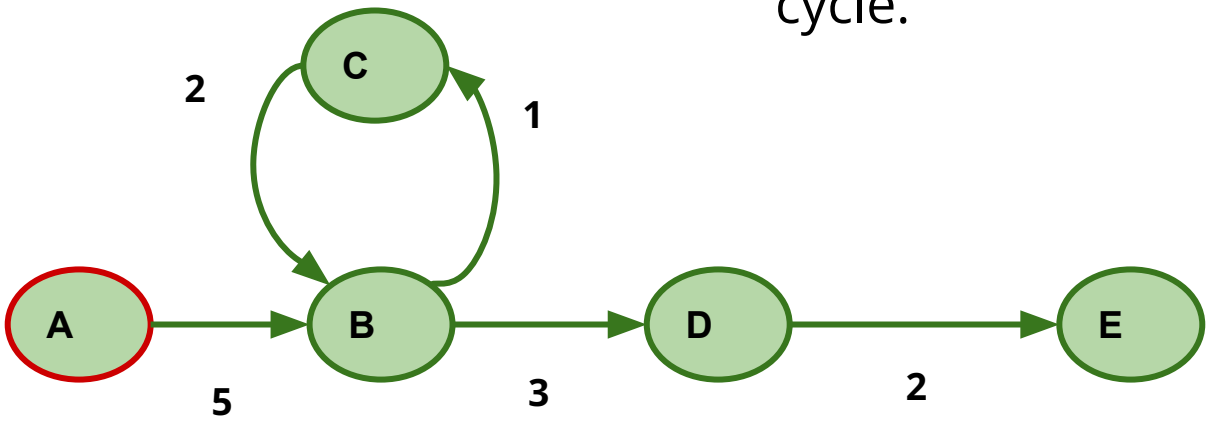
- ***Dijkstra Algorithm***
  Consider all edges' weights are nonnegative

- ***Bellman Ford Algorithm (Dynamic Problem)***
  - Allow negative-weight edges in the input graph and produce a correct answer as long as no negative-weight cycles are reachable from the source.

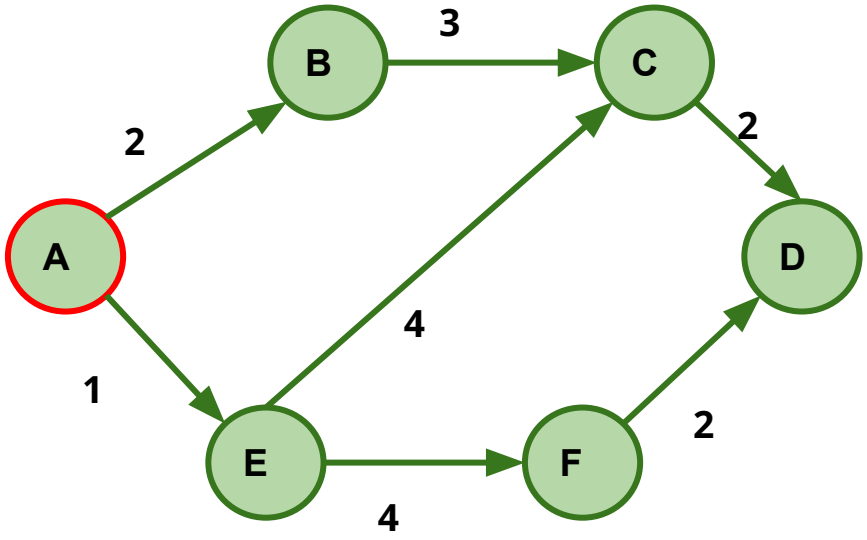
  - Can detect negative-weight cycles

Cycle in shortest path!!

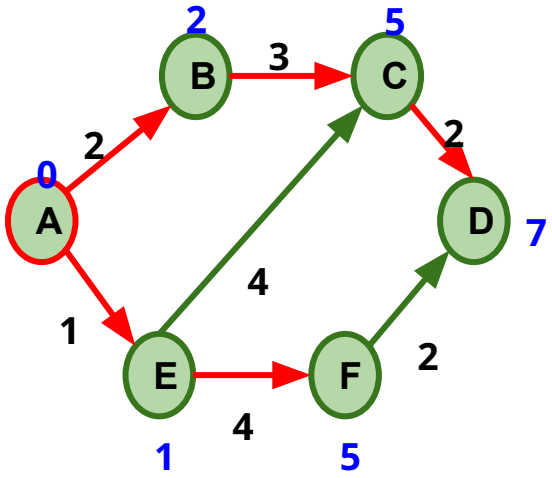
Shortest paths are simple paths with no weighted cycle.

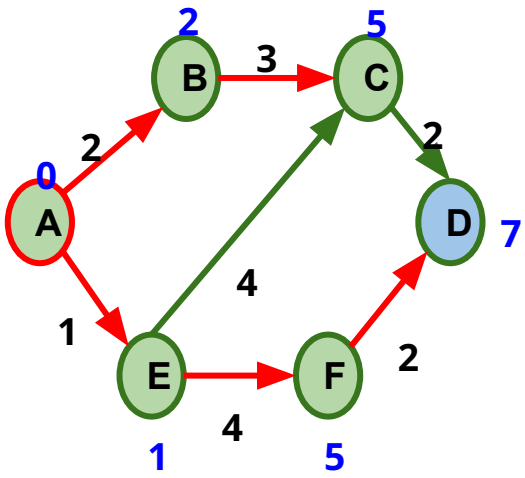


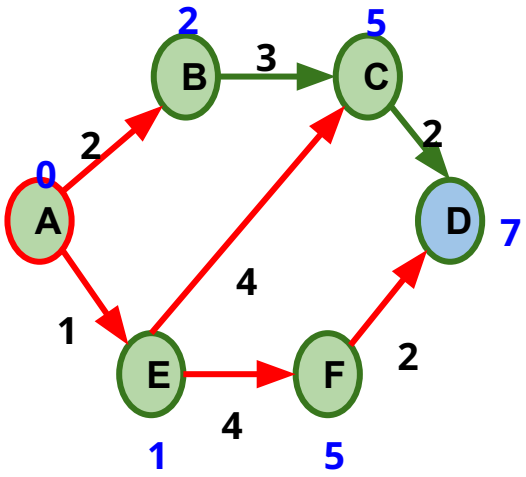
Are Shortest Paths Unique?



Is Shortest Path Unique?



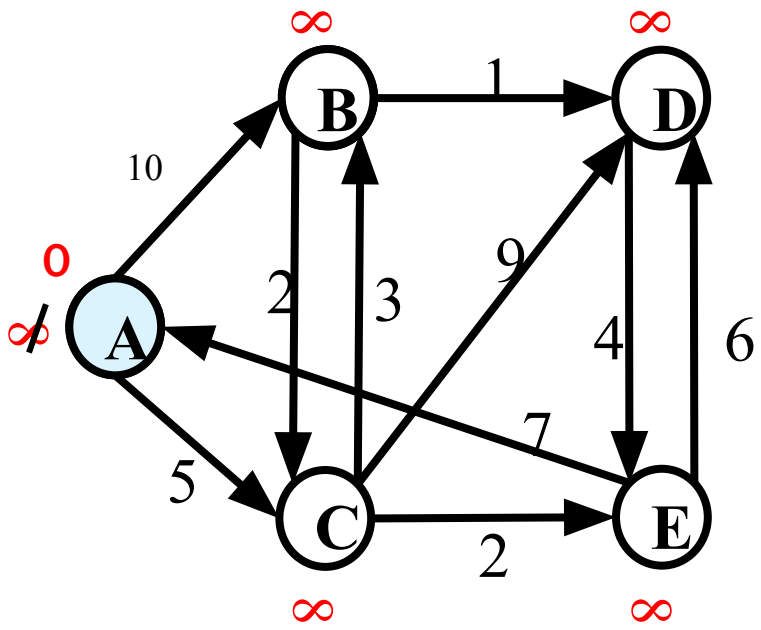




Shortest paths are not unique.

Shortest Path Concepts

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ G.V
2      v.d = ∞
3      v.π = NIL
4  s.d = 0
```



Shortest Path Concepts

Relaxation:

The process of relaxing an edge (u,v) consists of
testing whether we can improve the shortest path to v found so far by going through u
and, if so, updating v.d and v.π

```
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
```

Properties of shortest paths and relaxation

Triangle inequality

Upper-bound property

No-path property

Convergence property

Path-relaxation property

Predecessor-subgraph property

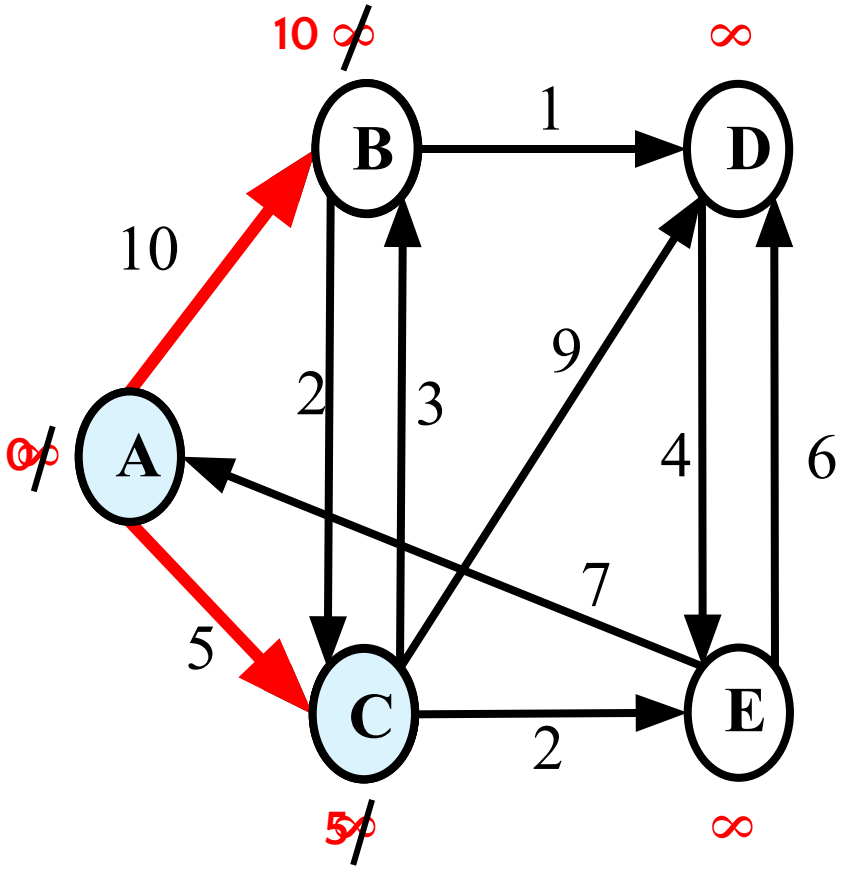
Dijkstra Algorithm

```
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)
```

Q : Min Priority Queue with priority value based on d.

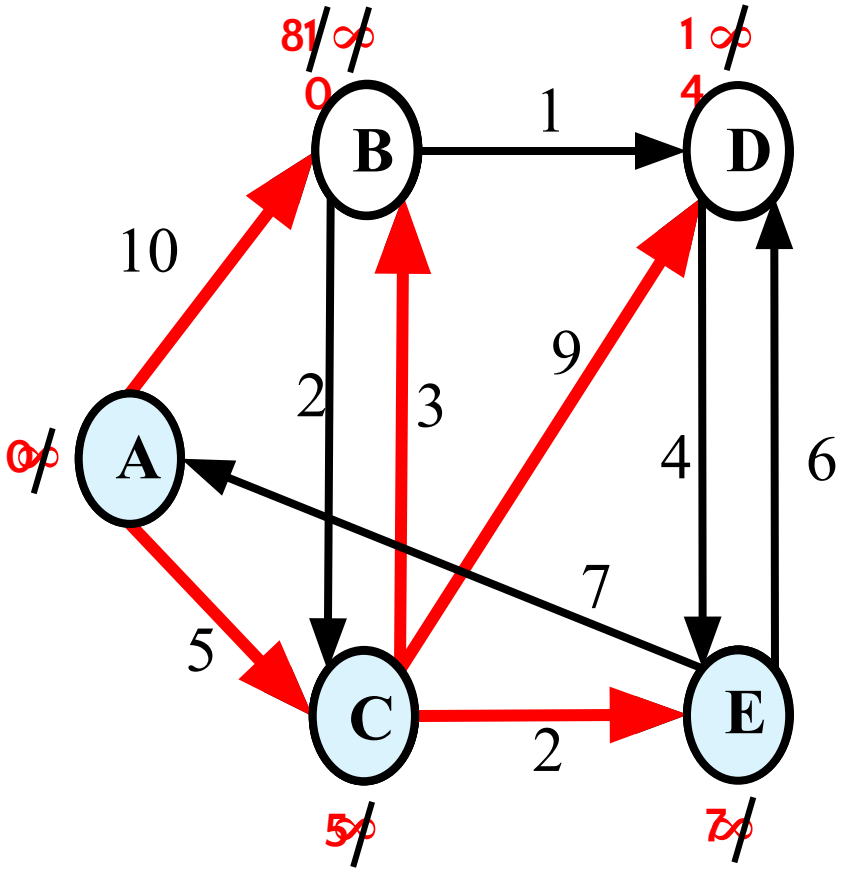
S : set of vertices whose final shortest-path weights from the source s have already been determined

DIJSKTRA's SIMULATION



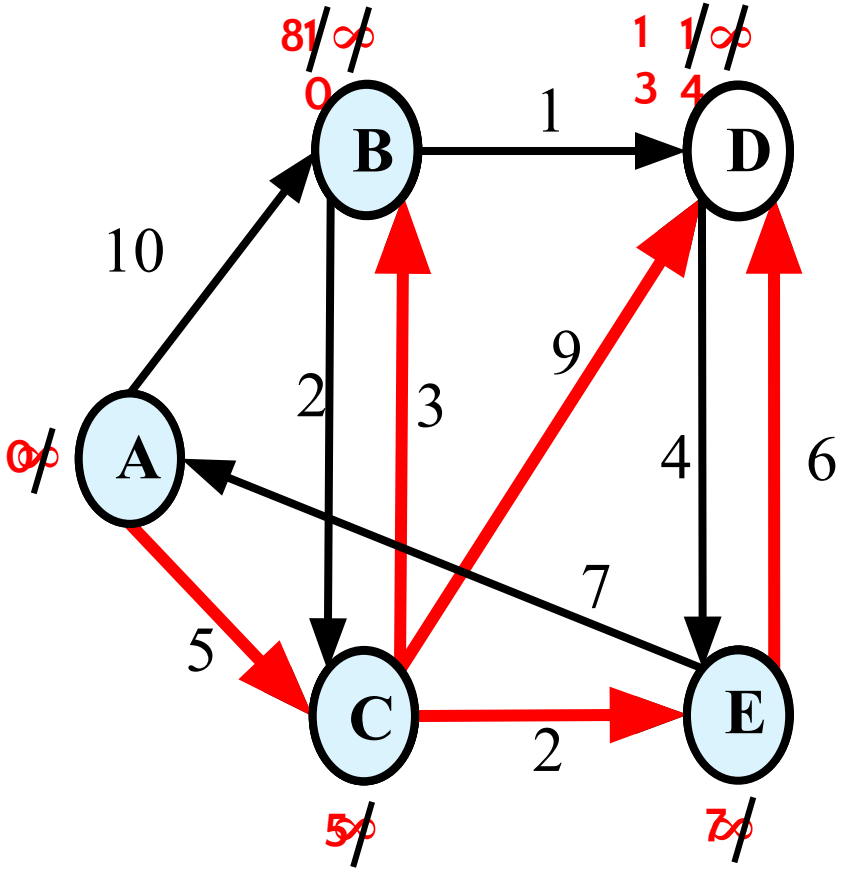
Special thanks to: Lec Swapnil Biswas

DIJSKTRA's SIMULATION



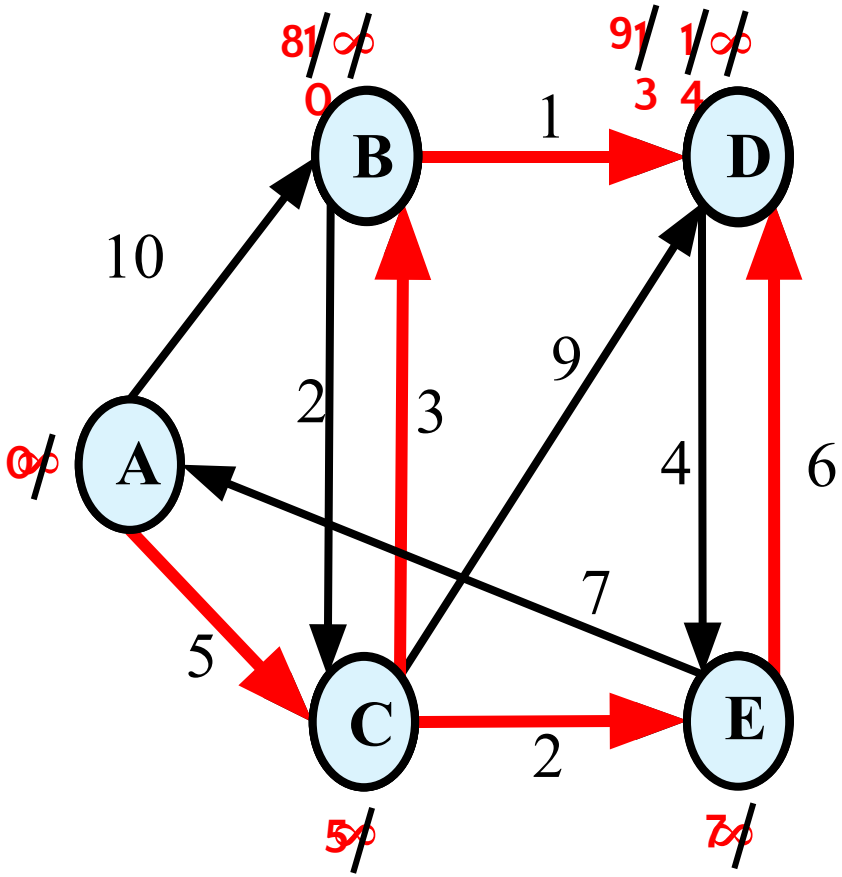
Special thanks to: Lec Swapnil Biswas

DIJSKTRA's SIMULATION



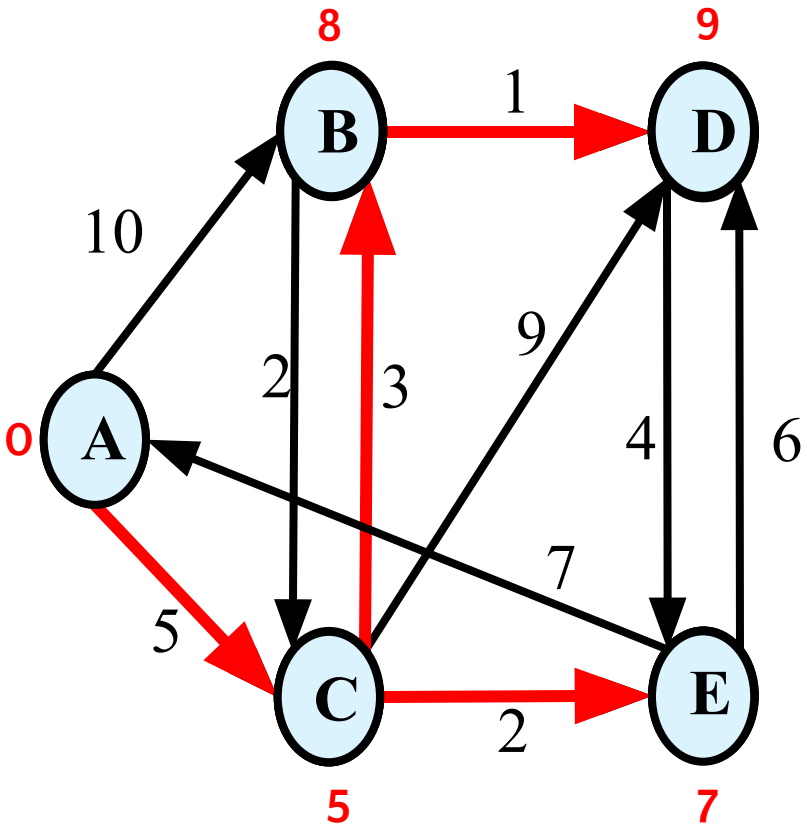
Special thanks to: Lec Swapnil Biswas

DIJSKTRA's SIMULATION



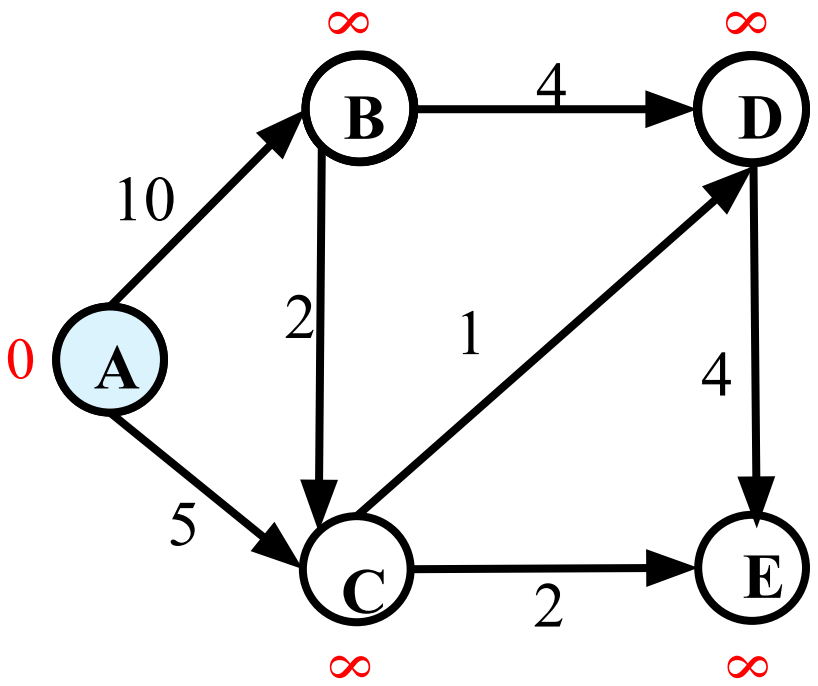
Special thanks to: Lec Swapnil Biswas

DIJSKTRA's SIMULATION



Bellman Ford Algorithm

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```



Bellman Ford Algorithm

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

Finds the shortest path and distance from source

Detects Negative Weighted Cycle

Correctness of Bellman Ford Algorithm

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

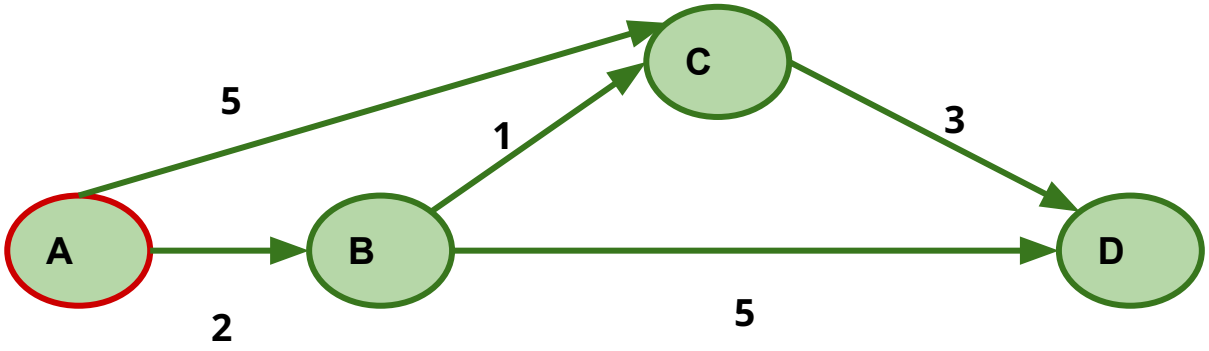
How does this part find the shortest path and distance from source ? / Correctness of Line 2-4 !!!

Correctness of Bellman Ford Algorithm

Correctness of Line 2-4 !!!

A.C.T path-relaxation property:

If $p = \langle v_0, v_1, \ldots, v_k \rangle$ is a shortest path from $s = v_0$ to v_k, and we relax the edges of p in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, then $v_k.d = \delta(s, v_k)$.

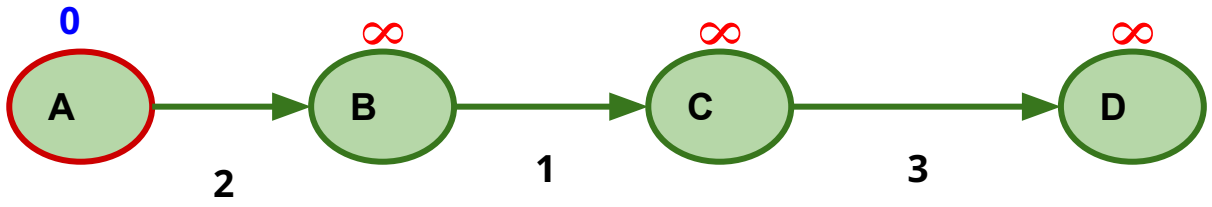


Correctness of Bellman Ford Algorithm

Correctness of Line 2-4 !!!

A.C.T path-relaxation property:

If $p = \langle v_0, v_1, \ldots, v_k \rangle$ is a shortest path from $s = v_0$ to v_k, and we relax the edges of p in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, then $v_k.d = \delta(s, v_k)$.



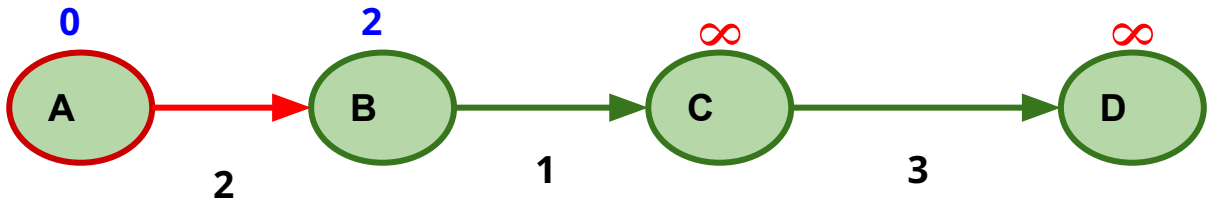
Correctness of Bellman Ford Algorithm

Correctness of Line 2-4 !!!

A.C.T path-relaxation property:

If $p = \langle v_0, v_1, \ldots, v_k \rangle$ is a shortest path from $s = v_0$ to v_k, and we relax the edges of p in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, then $v_k.d = \delta(s, v_k)$.

`relax( A, B, 2)`



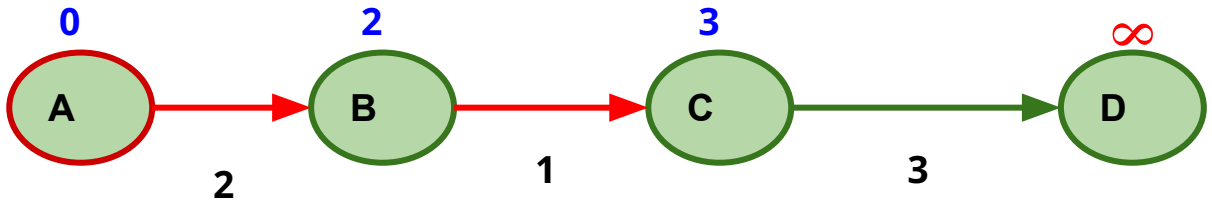
Correctness of Bellman Ford Algorithm

Correctness of Line 2-4 !!!

A.C.T path-relaxation property:

If $p = \langle v_0, v_1, \ldots, v_k \rangle$ is a shortest path from $s = v_0$ to v_k, and we relax the edges of p in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, then $v_k.d = \delta(s, v_k)$.

`relax( B, C, 1)`



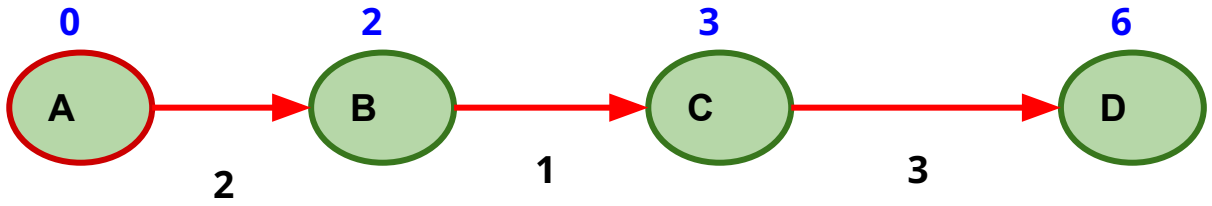
Correctness of Bellman Ford Algorithm

Correctness of Line 2-4 !!!

A.C.T path-relaxation property:

If $p = \langle v_0, v_1, \ldots, v_k \rangle$ is a shortest path from $s = v_0$ to v_k, and we relax the edges of p in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, then $v_k.d = \delta(s, v_k)$.

`relax( C, D, 3)`



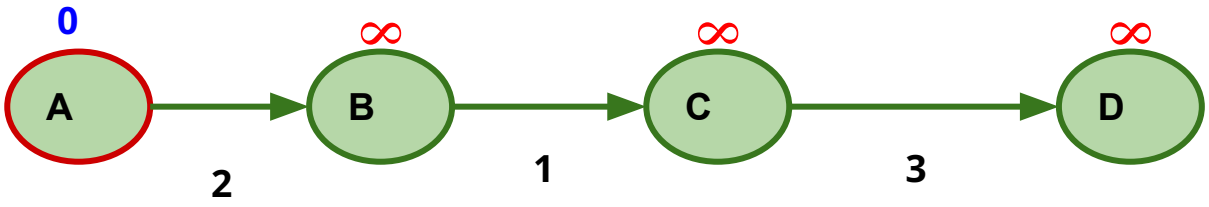
For each vertex, we found the shortest distance from the source A.
Observation :
If the edge relaxation order is maintained then each edge needs to be relaxed once to get the shortest distance of all vertices.

Correctness of Bellman Ford Algorithm

Correctness of Line 2-4 !!!

What if the edge relaxation order is done differently ????

Let's assume : E= { CD, BC, AB }

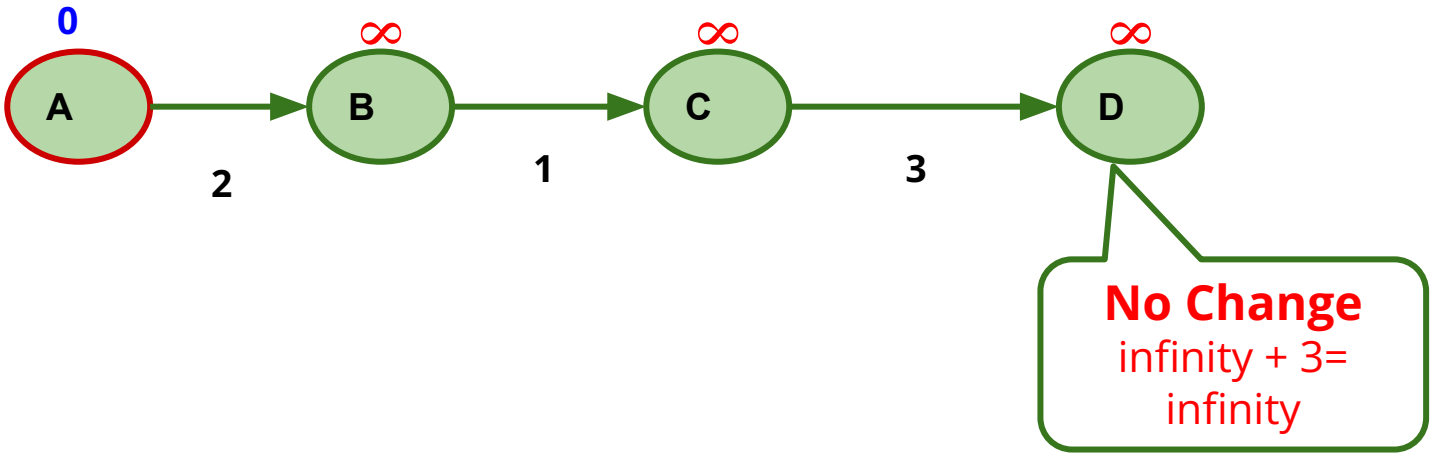


Correctness of Bellman Ford Algorithm

Correctness of Line 2-4 !!!

E= { CD, BC, AB }

`relax( C, D, 3)`

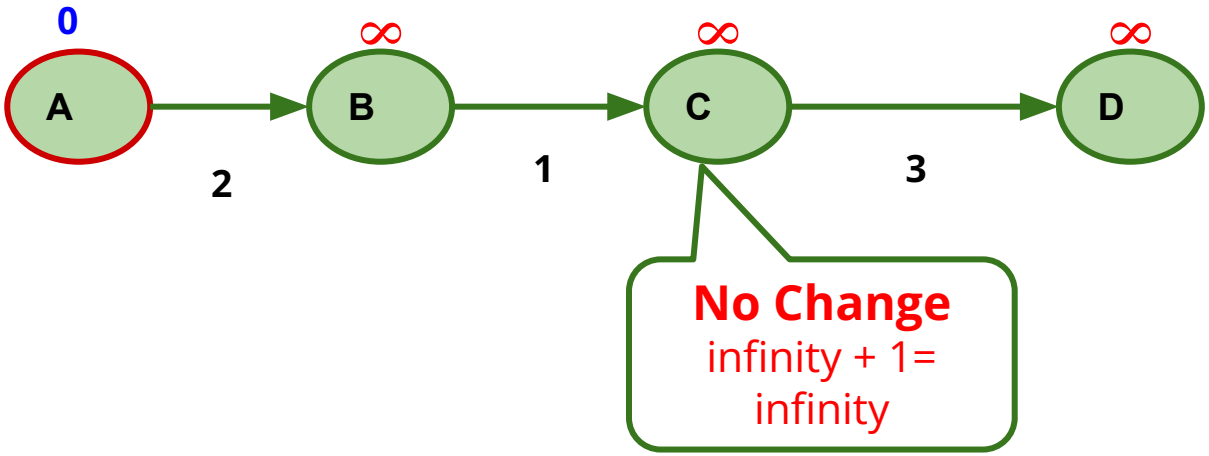


Correctness of Bellman Ford Algorithm

Correctness of Line 2-4 !!!

E= { CD, BC, AB }

```
relax( B, C, 1)
```

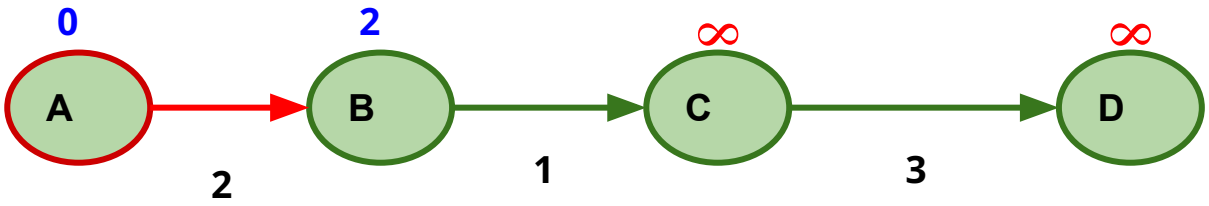


Correctness of Bellman Ford Algorithm

Correctness of Line 2-4 !!!

E= { CD, BC, AB }

```
relax( A, B, 2)
```



After relaxing each edge once , we found the shortest distance of only one vertex.

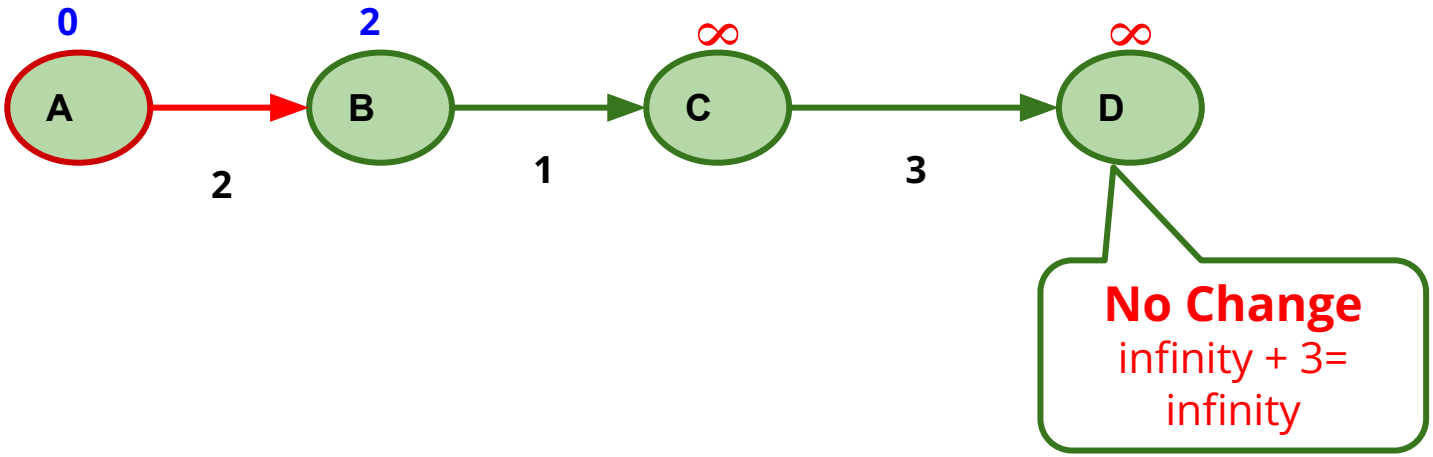
Correctness of Bellman Ford Algorithm

Correctness of Line 2-4 !!!

Let's repeat the process: (Relax the edges for **2nd** time**)**

E= { CD, BC, AB }

```
relax( C, D, 3)
```



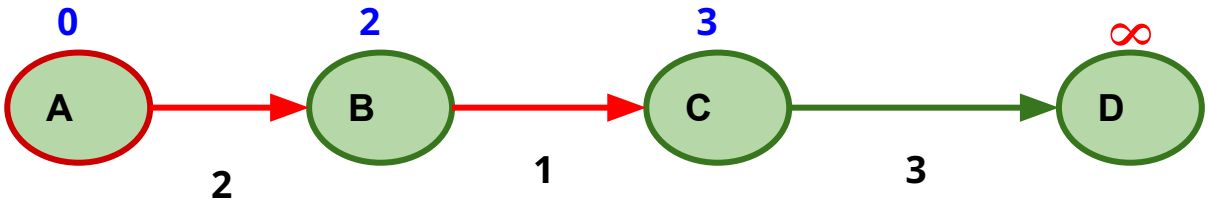
Correctness of Bellman Ford Algorithm

Correctness of Line 2-4 !!!

Let's repeat the process: (Relax the edges for **2nd** time**)**

E= { CD, BC, AB }

```
relax( B, C, 1)
```



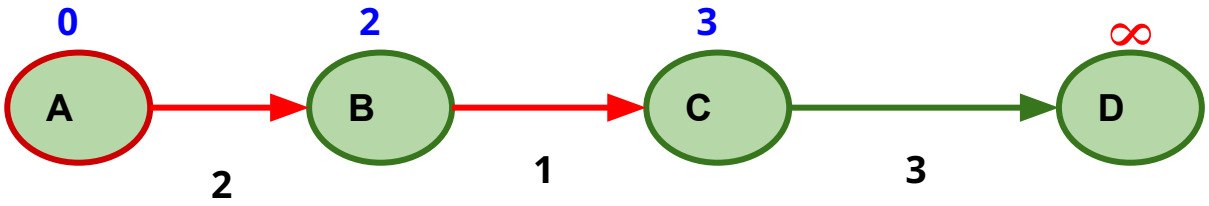
Correctness of Bellman Ford Algorithm

Correctness of Line 2-4 !!!

Let's repeat the process: (Relax the edges for **2nd** time**)**

E= { CD, BC, AB }

```
relax( A, B, 2)
```



*After relaxing each edge **twice** , we found the shortest distance of two vertices.*

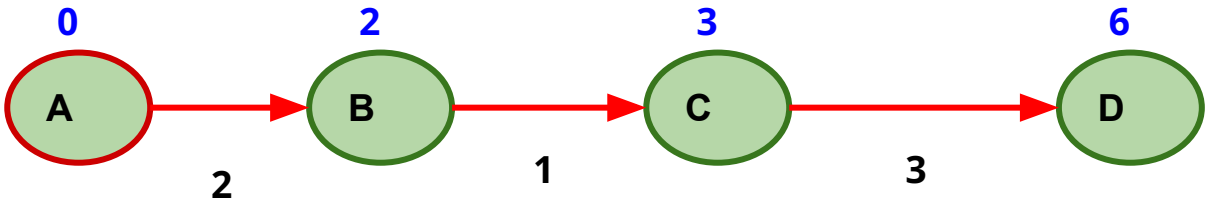
Correctness of Bellman Ford Algorithm

Correctness of Line 2-4 !!!

Let's repeat the process: (Relax the edges for **3rd** time**)**

E= { CD, BC, AB }

```
relax( C, D, 3)
```



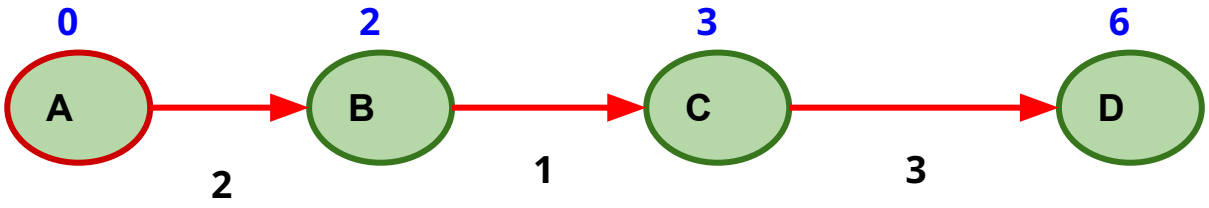
Correctness of Bellman Ford Algorithm

Correctness of Line 2-4 !!!

Let's repeat the process: (Relax the edges for **3rd** time**)**

E= { CD, BC, AB }

```
relax( B, C, 1)
```



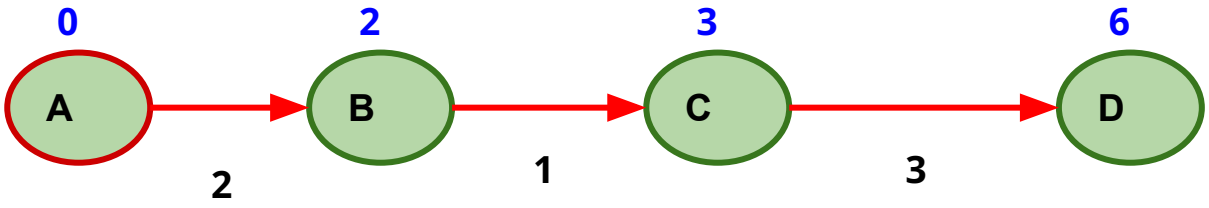
Correctness of Bellman Ford Algorithm

Correctness of Line 2-4 !!!

Let's repeat the process: (Relax the edges for **3rd** time**)**

E= { CD, BC, AB }

```
relax( A, B, 2)
```



*After relaxing each edge **thrice** , we found the shortest distance of three vertices.*

Correctness of Bellman Ford Algorithm

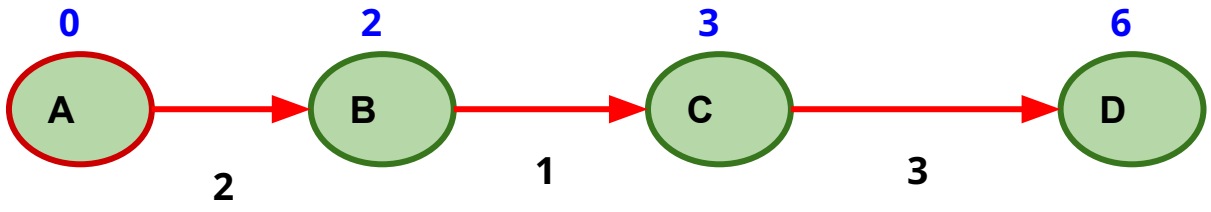
Correctness of Line 2-4 !!!

Maximum Number of edges in a shortest path = V-1

In the example shown, V= {A,B,C,D} and *E= { CD, BC, AB }*

*And after relaxing each edge **thrice (V-1 times)** , we found the shortest distance of all vertices.*

Thus, lines 2-4 find the v.d = δ(s,v) for each vertices v ∈ V after relaxing each edge V-1 times.



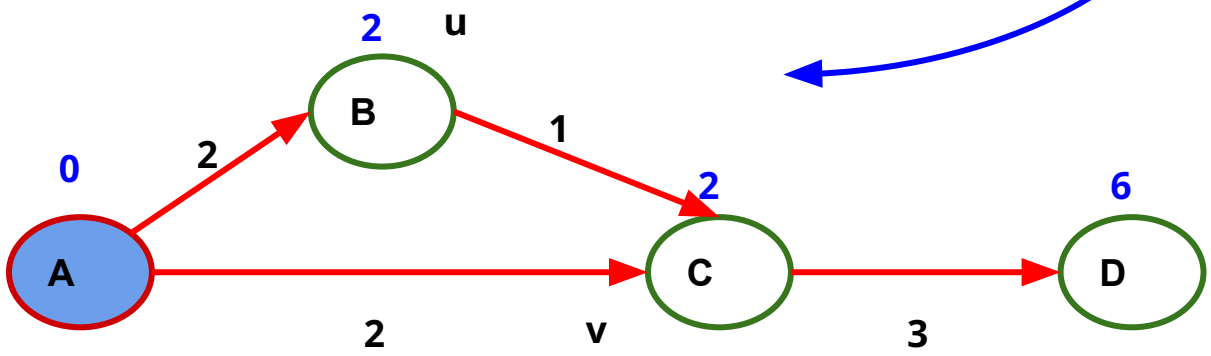
Correctness of Bellman Ford Algorithm

Correctness of the algorithm returning true

Let's assume graph has no negative weight cycle reachable from source.

A.C.T triangle inequality property,

For any edge $(u, v) \in E$, we have $\delta(s, v) \leq \delta(s, u) + w(u, v)$.



Correctness of Bellman Ford Algorithm

Correctness of the algorithm returning true

Let's assume graph has no negative weight cycle reachable from source.

Lines 2-4 find the v.d = δ(s,v) for each vertices $v \in V$ after relaxing each edge V-1 times.

A.C.T triangle inequality property,

For any edge $(u, v) \in E$, we have $\delta(s, v) \leq \delta(s, u) + w(u, v)$.

$$
\begin{aligned}
v.d &= \delta(s, v) \\
&\leq \delta(s, u) + w(u, v) \quad \text{(by the triangle inequality)} \\
&= u.d + w(u, v) ,
\end{aligned}
$$

So none of the tests in line 6 causes BELLMAN-FORD to return FALSE. Therefore, it returns TRUE.

Correctness of Bellman Ford Algorithm

Correctness of the algorithm returning False

Let's assume graph has negative weight cycle C reachable from source.

Let, C = <V0,V1,.....,Vk>, where V0 =Vk.
So,

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0 .$$

Let's assume, for the purpose of contradiction that the Bellman-Ford algorithm returns TRUE.

Correctness of Bellman Ford Algorithm

Correctness of the algorithm returning False

Let's assume, for the purpose of contradiction that the Bellman-Ford algorithm returns TRUE. So,

$$v_i.d \leq v_{i-1}.d + w(v_{i-1}, v_i) \text{ for } i = 1, 2, \ldots, k.$$

Summing the inequalities around cycle c gives us

$$\begin{aligned} \sum_{i=1}^{k} v_i.d &\leq \sum_{i=1}^{k} (v_{i-1}.d + w(v_{i-1}, v_i)) \\ &= \sum_{i=1}^{k} v_{i-1}.d + \sum_{i=1}^{k} w(v_{i-1}, v_i) . \end{aligned}$$

Now, in circle, V0=Vk. So,

$$\sum_{i=1}^{k} v_i.d = \sum_{i=1}^{k} v_{i-1}.d$$

$$0 \leq \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Correctness of Bellman Ford Algorithm

Correctness of the algorithm returning False

But,

$$0 \leq \sum_{i=1}^{k} w(v_{i-1}, v_i) \quad \text{Contradicts} \quad \sum_{i=1}^{k} w(v_{i-1}, v_i) < 0 .$$

We conclude that the Bellman-Ford algorithm returns TRUE if graph G contains no negative-weight cycles reachable from the source, and FALSE otherwise.

Correctness of Dijkstra's algorithm

Self Study:
Correctness of Dijkstra Algorithm (Theorem 24.6)

Reference:
Introduction to ALGORITHMS: THOMAS H. CORMEN (3rd Edition)


thank
you

CSE 215: Data Structures & Algorithms II

Dynamic Programming : LCS

Lec Sumaiya Nuha Mustafina
Dept Of CSE
sumaiyamustafina@cse.mist.ac.bd

Dynamic Programming

Dynamic programming :

- Applied to optimization problems. Such problems can have many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value.

- Solves every subproblems once and stores it in a table.

- Dynamic programming refers to a tabular method, not computer code.

Dynamic Programming

Two key characteristics that a problem must have for dynamic programming to be a viable solution technique.

- Optimal substructure property.
- Overlapping subproblems property.

*Recall: *Optimal substructure is one of the key indicators that dynamic programming and the greedy method might be applied.*

Dynamic Programming

Optimal substructure property.

- The optimal substructure property states that an optimal solution to a problem contains **optimal solutions to its sub problems**.
- In simpler terms, if you can solve a larger problem by breaking it down into smaller subproblems, and the solution to the larger problem relies on the solutions to those sub problems, then the problem exhibits optimal substructure.

Overlapping subproblems property.

- Overlapping subproblems occur when a problem can be broken down into sub problems which are reused several times.
- This means that the same sub problem is solved multiple times in the process of solving the larger problem.
- Dynamic programming takes advantage of this property by solving each sub problem **only once** and storing the results (usually in a table or array) so that when the same subproblem is encountered again, it can be quickly retrieved from the table instead of being recalculated.

Dynamic Programming

There are usually ***two equivalent ways*** to implement a dynamic-programming approach.

The first approach is ***top-down with memoization***.

- Memoization is derived from the Latin word "memorandum" ("to be remembered").
- In this approach, the procedure is written recursively in a natural manner, but modified to save the result of each subproblem (usually in an array or hash table).
- Recursive code + Memoization code
- The procedure now first checks to see whether it has previously solved this subproblem. If so, it returns the saved value, saving further computation at this level; if not, the procedure computes the value in the usual manner.
- The recursive procedure is said to be memoized as it “remembers” what results it has computed previously.

Dynamic Programming

There are usually ***two equivalent ways*** to implement a dynamic-programming approach.

The second approach is ***the bottom-up method (Tabulation)***.

- This approach typically depends on some natural notion of the "size" of a subproblem, such that solving any particular subproblem depends only on solving "smaller" subproblems.
- We sort the subproblems by size and solve them in size order, smallest first.
- When solving a particular subproblem, we have already solved all of the smaller subproblems its solution depends upon, and we have saved their solutions.

Dynamic Programming

The development of dynamic-programming, is broken into a sequence of ***four steps:***

1. Characterize the structure of an optimal solution.

1. Recursively define the value of an optimal solution.

1. Compute the value of an optimal solution, typically in a bottom-up fashion.

1. Construct an optimal solution from computed information.

Longest common subsequence (LCS)

LCS:

- is defined as the longest subsequence that is common to all the given sequences.
- the elements of the subsequence need not be consecutive.
- but the sequence or order will be maintained.

Let's say, X = ABCD Y= JBAGHCED

AB**CD** JB**A**GH**C**E**D**

length of LCS : 3
longest common subsequence is: **ACD**

LCS is a optimization problem as we are try to perform maximization here.

Longest common subsequence (LCS)

X = **ABCBDAB**
Y= **BDCABA**

Find the **LCS** of X and Y.

One of the LCS of X and Y = **BCBA**

Longest common subsequence (LCS)

X = **ABCBDAB**
Y= **BDCABA**

Find the **LCS** of X and Y.
One of the LCS of X and Y = **BCBA**

***Brute Force Approach of finding LCS*:**

- Enumerate all subsequences of X and check each subsequence to see whether it is also a subsequence of Y, keeping track of the longest subsequence we find.
- Each subsequence of X corresponds to a subset of the indices {1,2,....,m} of X.
- Because X has 2^m subsequences, this approach requires exponential time, making it impractical for long sequences.

Longest common subsequence (LCS)

Solving LCS using Dynamic Programming:
Step 1: Characterizing a longest common subsequence

The LCS problem has an optimal-substructure property.

Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

Proof : Theorem 15.1 (3rd Edition)
Self Study

Prefix: defined as **i**th prefix of X, for i=0, 1,.... ,m, as $\mathbf{X_i = \langle x_1, x_2, \ldots, x_i \rangle}$.
For example, if X=<A,B,C,B,D> , then $\mathbf{X_2}$=<A,B> and $\mathbf{X_0}$ is the empty sequence.

Longest common subsequence (LCS)

Solving LCS using Dynamic Programming:
Step 2: A recursive solution

The optimal-substructure property implies that we should examine either one or two subproblems when finding an LCS of
$X=\langle x_1, x_2, \ldots, x_m \rangle$ and $Y=\langle y_1, y_2, \ldots, y_n \rangle$.

- If **$x_m = y_n$**:
  we must find an LCS of **X_{m-1} and Y_{n-1}.** *Appending* $x_m = y_n$ to this LCS yields an LCS of X and Y .

- If **$x_m \mathrel{!}= y_n$:**
  then we must solve two subproblems:
  finding an LCS of X_{m-1} and Y and finding an LCS of X and Y_{n-1}.
  Whichever of these two LCSs is longer is an LCS of X and Y .

Longest common subsequence (LCS)

Solving LCS using Dynamic Programming:
Step 2: A recursive solution

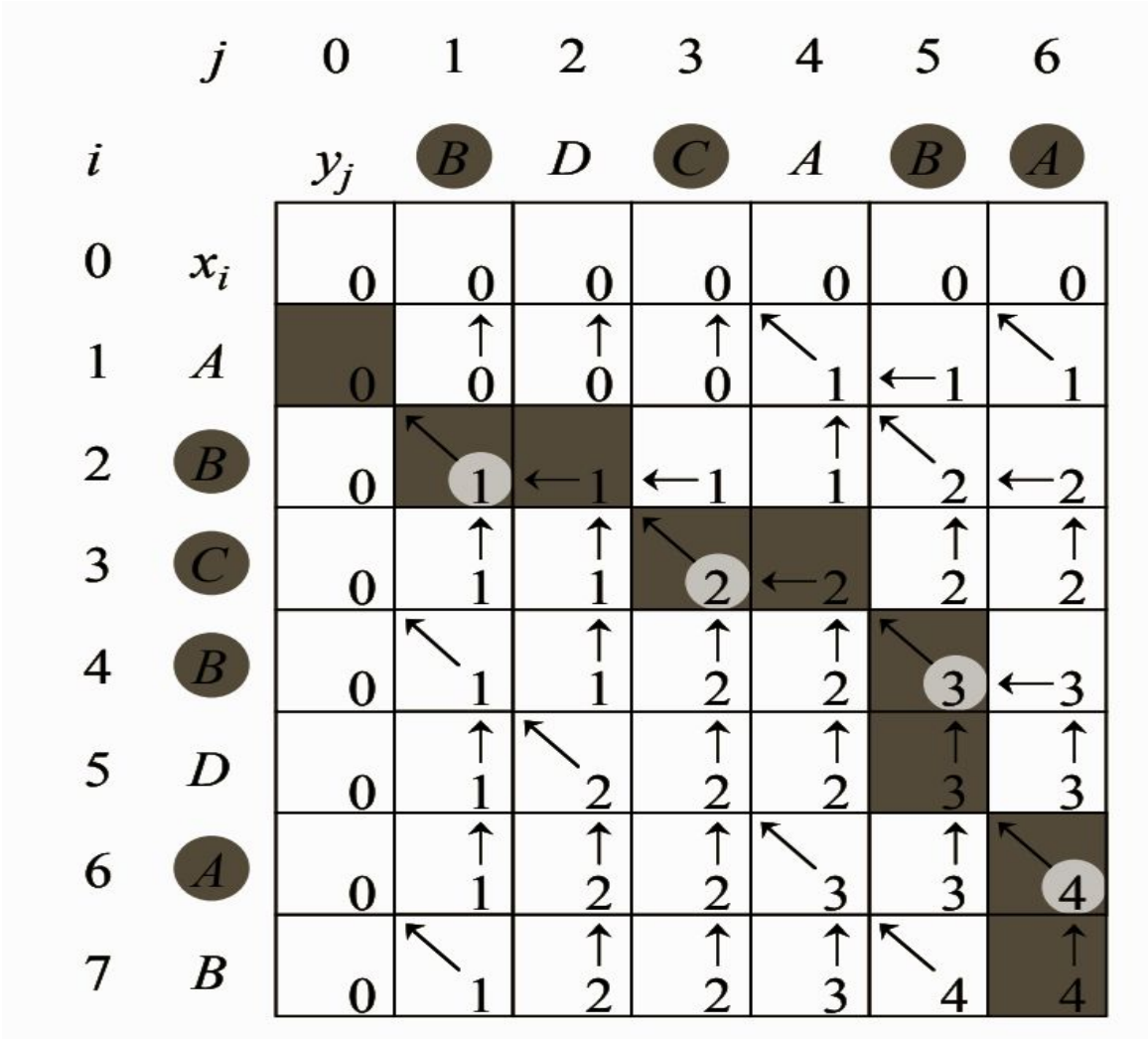
Let, ***c[i , j]*** be the length of an LCS of the sequences $\mathbf{X}_i$ and $\mathbf{Y}_j$
Thus optimal substructure of the LCS problem gives the recursive formula:

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

Longest common subsequence (LCS)

Step 3: Computing the length of an LCS

X = **ABCBDAB**
Y= **BDCABA**



i		j	0	1	2	3	4	5	6
		y_j		B	D	C	A	B	A
0	x_i		0	0	0	0	0	0	0
1	A		0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B		0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C		0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B		0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D		0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A		0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B		0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Longest common subsequence (LCS)

Step 3: Computing the length of an LCS (Bottom-Up)

It computes the entries in ***row-major order****. (That is, the procedure fills in the first row of c from left to right, then the second row, and so on.)*

The procedure also maintains the table b=[1..m; 1..n] to help in constructing an optimal solution.

X = **ABCBDAB**

Y= **BDCABA**

LCS-LENGTH(X,Y)

```
LCS-LENGTH(X, Y)
 1  m = X.length
 2  n = Y.length
 3  let b[1..m, 1..n] and c[0..m, 0..n] be new tables
 4  for i = 1 to m
 5      c[i, 0] = 0
 6  for j = 0 to n
 7      c[0, j] = 0
 8  for i = 1 to m
 9      for j = 1 to n
10          if x_i == y_j
11              c[i, j] = c[i − 1, j − 1] + 1
12              b[i, j] = "↖"
13          elseif c[i − 1, j] ≥ c[i, j − 1]
14              c[i, j] = c[i − 1, j]
15              b[i, j] = "↑"
16          else c[i, j] = c[i, j − 1]
17              b[i, j] = "←"
18  return c and b
```

O(mn)

Longest common subsequence (LCS)

Step 3: Computing the length of an LCS (Bottom-Up)

X = **ABCBDAB**
Y= **BDCABA**

LCS-LENGTH(X,Y)

```
LCS-LENGTH(X, Y)
 1  m = X.length
 2  n = Y.length
 3  let b[1..m, 1..n] and c[0..m, 0..n] be new tables
 4  for i = 1 to m
 5      c[i, 0] = 0
 6  for j = 0 to n
 7      c[0, j] = 0
 8  for i = 1 to m
 9      for j = 1 to n
10          if x_i == y_j
11              c[i, j] = c[i − 1, j − 1] + 1
12              b[i, j] = "↖"
13          elseif c[i − 1, j] ≥ c[i, j − 1]
14              c[i, j] = c[i − 1, j]
15              b[i, j] = "↑"
16          else c[i, j] = c[i, j − 1]
17              b[i, j] = "←"
18  return c and b
```

in code x[i-1] == y[j-1]

Longest common subsequence (LCS)

Step 4: Constructing an Optimal Solution / Constructing an LCS

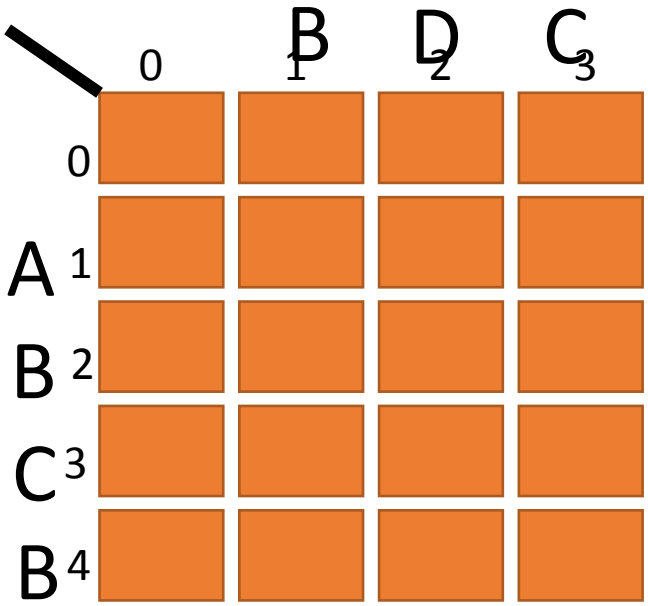
```
PRINT-LCS(b, X, i, j)
1  if i == 0 or j == 0
2       return
3  if b[i, j] == "↖"
4       PRINT-LCS(b, X, i − 1, j − 1)
5       print x_i
6  elseif b[i, j] == "↑"
7       PRINT-LCS(b, X, i − 1, j)
8  else PRINT-LCS(b, X, i, j − 1)
```

O(m+n)

Longest common subsequence (LCS)

Step 3: Computing the length of an LCS (Top-Down : Memoization)

Recursive code + Memoization code

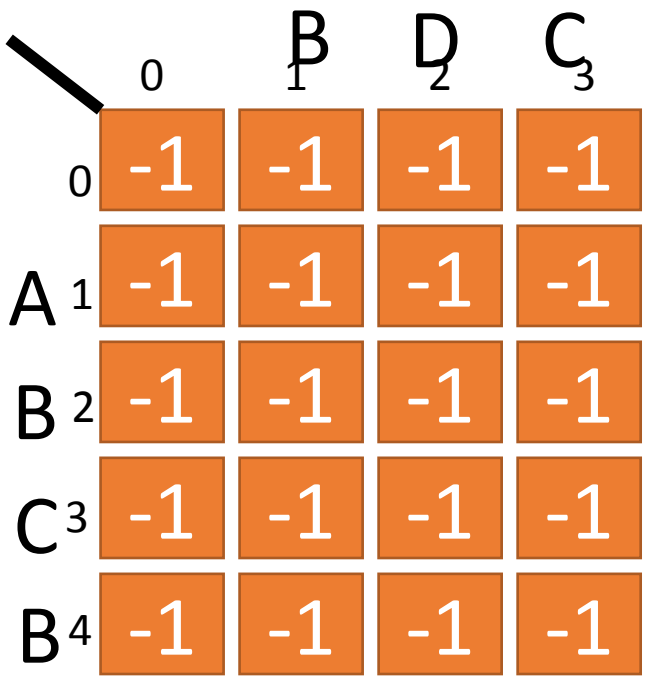


X= ABCB
Y=BDC

Memo

Longest common subsequence (LCS)

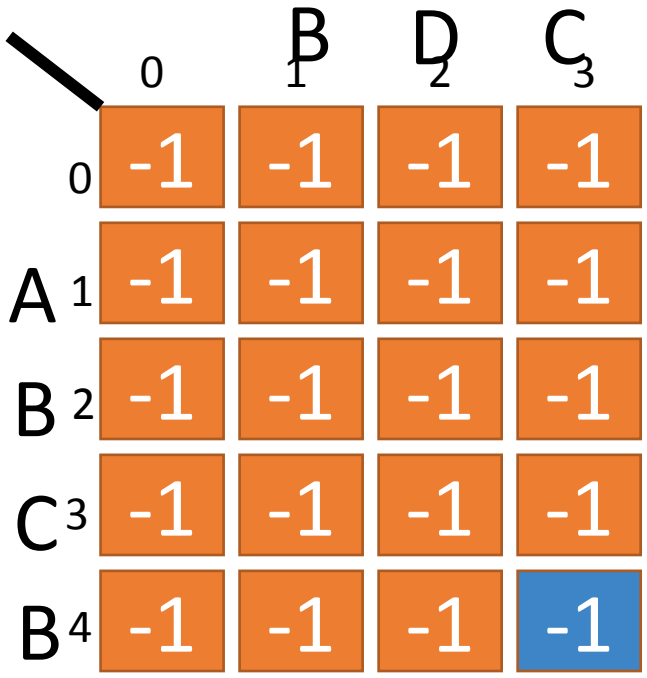
Step 3: Computing the length of an LCS (Top-Down : Memoization)



X= ABCB
Y=BDC

Longest common subsequence (LCS)

Step 3: Computing the length of an LCS (Top-Down : Memoization)



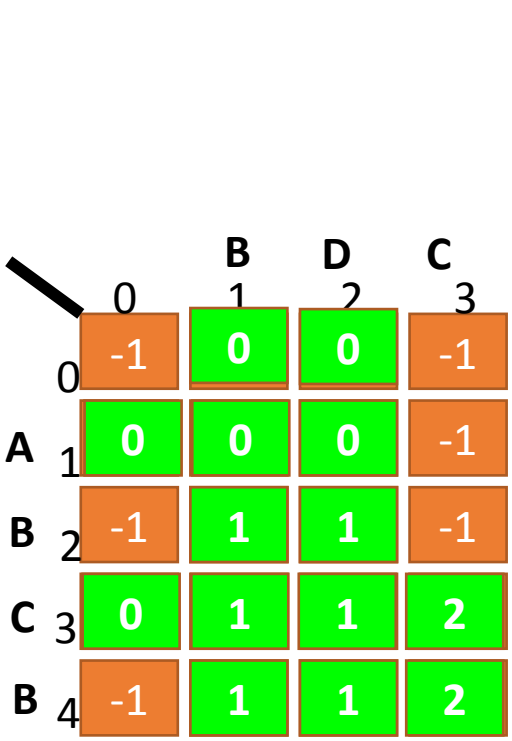
As it is a Top-Down approach, we will start from the top i.e. we will consider the bigger problem first.

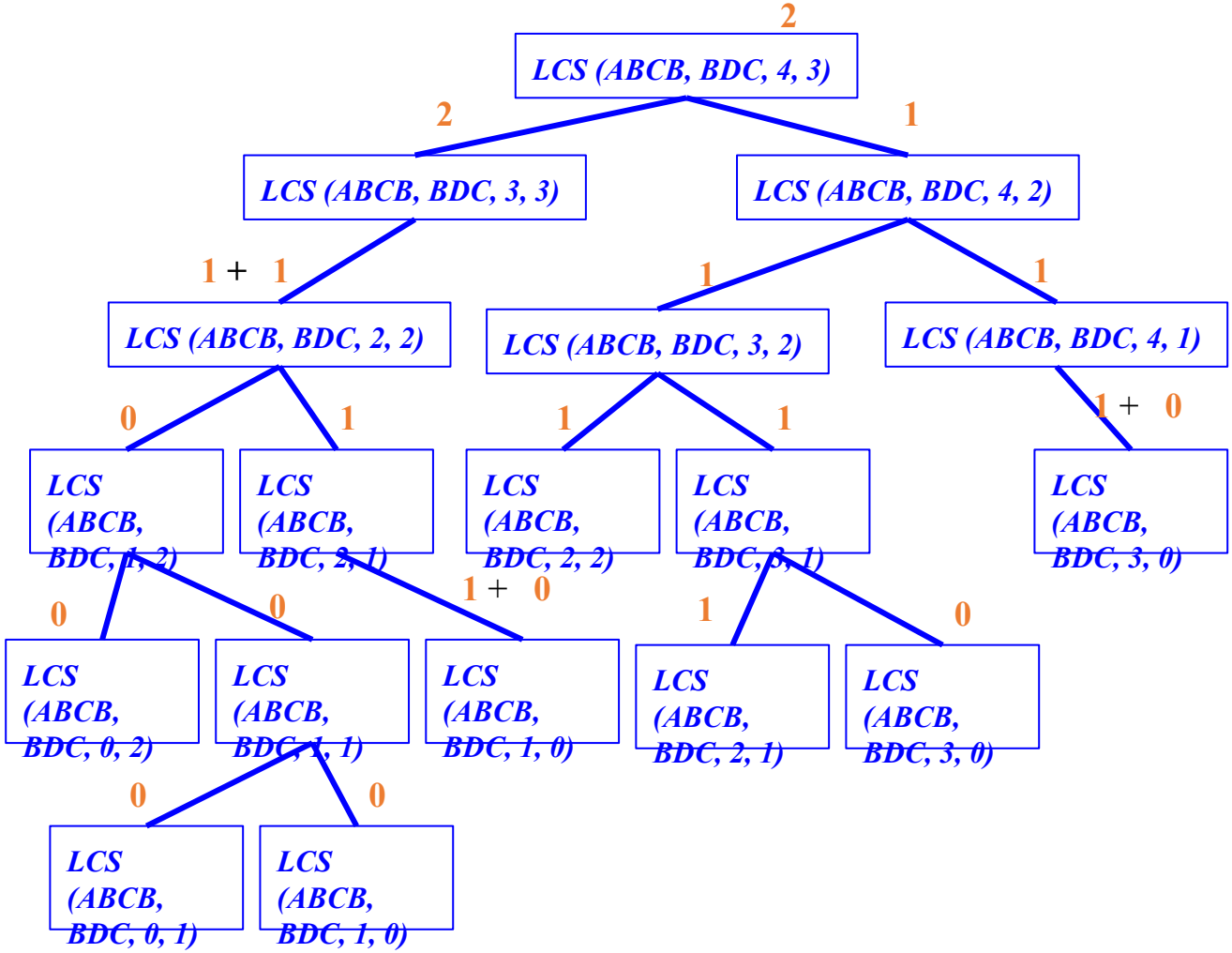
X= ABCB , Y=BDC

LCS (X, Y, 4, 3)

Longest common subsequence (LCS)

Step 3: Computing the length of an LCS (Top-Down : Memoization)





Longest common subsequence (LCS)

Time Complexity of LCS:

m and n *are* ***length*** *of sequence* ***X and Y.***

Bottom-Up Approach : O(mn)

Top-Down Approach : O(mn)

Brute Force Approach : Exponential time

Longest common subsequence (LCS)

Exercises:

15.4-2

Give pseudocode to reconstruct an LCS from the completed c table and the original sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ in $O(m + n)$ time, without using the b table.

15.4-3

Give a memoized version of LCS-LENGTH that runs in $O(mn)$ time.


thank
you

CSE 215: Data Structures & Algorithms II

Dynamic Programming : Knapsack

Lec Sumaiya Nuha Mustafina
Dept Of CSE
sumaiyamustafina@cse.mist.ac.bd

Dynamic Programming

The development of dynamic-programming, is broken into a sequence of ***four steps:***

1. Characterize the structure of an optimal solution.

2. Recursively define the value of an optimal solution.

3. Compute the value of an optimal solution, typically in a bottom-up fashion.

4. Construct an optimal solution from computed information.

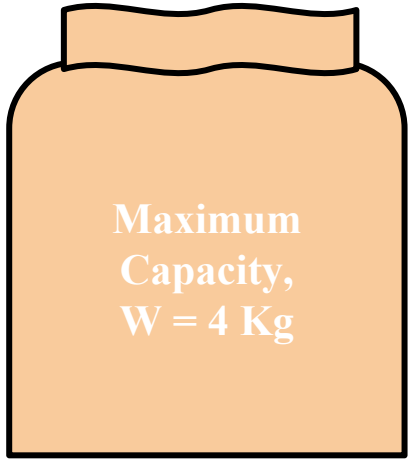
0-1 Knapsack

- A thief robbing a store finds **n** items.
- The **i-th** item is worth $\mathbf{v_i}$ dollars and weighs $\mathbf{w_i}$ pounds, where $\mathbf{v_i}$ and $\mathbf{w_i}$ are integers.
- The thief wants to take as valuable a load as possible, but he can carry **at most W pounds** in his knapsack, for some integer W.
- Which items should he take to get the maximum profit?
- This problem is called the ***0-1 knapsack*** problem because for each item, the thief must ***either take it or leave it*** behind.

0-1 Knapsack

- A thief robbing a store finds **n** items.
- The **i-th** item is worth v_i dollars and weighs w_i pounds, where v_i and w_i are integers.
- The thief wants to take as valuable a load as possible, but he can carry **at most W pounds** in his knapsack, for some integer W.

Item No:	1st	2nd	3rd
v	5 tk	10 tk	16 tk
weight	3 kg	1 kg	2 kg



0-1 Knapsack

Solving 0-1 knapsack using Dynamic Programming:
Step 1: Characterizing a 0-1 Knapsack

The 0-1 Knapsack problem has an optimal-substructure property.

Consider the most valuable load that weighs at most W pounds.

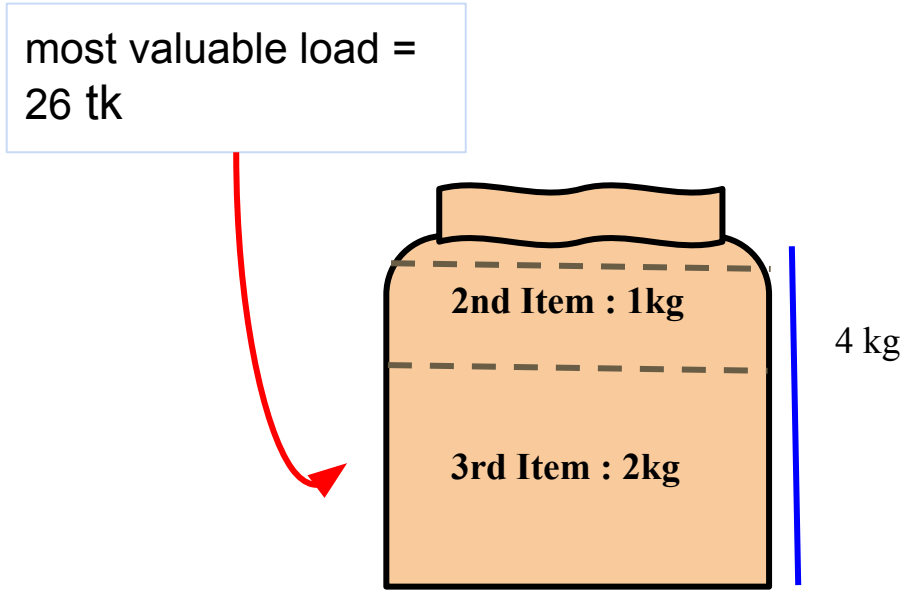
If we remove item j from this load, the remaining load must be the most valuable load weighing at most $W-w_j$ that the thief can take from the n-1 original items excluding j.

0-1 Knapsack

Solving 0-1 knapsack using Dynamic Programming:
Step 1: Characterizing a 0-1 Knapsack

Maximum Capacity/size of the knapsack
W = 4 Kg
And n=3

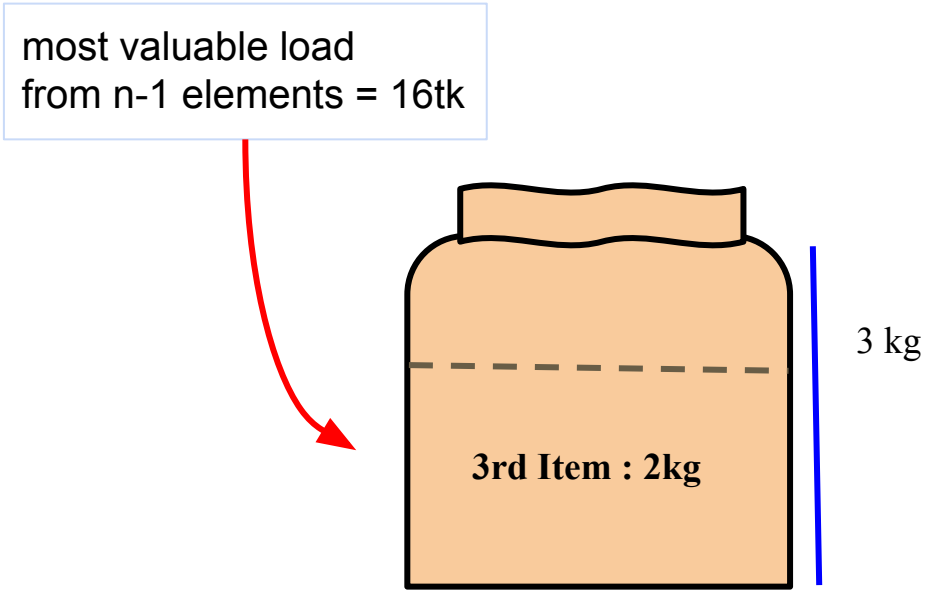
Item No:	1st	2nd	3rd
v	5 tk	10 tk	16 tk
weight	2kg	1 kg	2 kg



0-1 Knapsack

Solving 0-1 knapsack using Dynamic Programming:
Step 1: Characterizing a 0-1 Knapsack

Maximum Capacity/size of the knapsack
W - w_2= 4 Kg - 1 kg = 3 kg
And n-1=3-1=2



0-1 Knapsack

Solving 0-1 Knapsack using Dynamic Programming:
Step 2: A recursive solution

Table: c[n+1][W+1]
Let, ***c[i , w]*** be the value of solution for items **1** to **i** and maximum weight of knapsack w.
Thus optimal substructure of the 0-1 Knapsack problem gives the recursive formula:

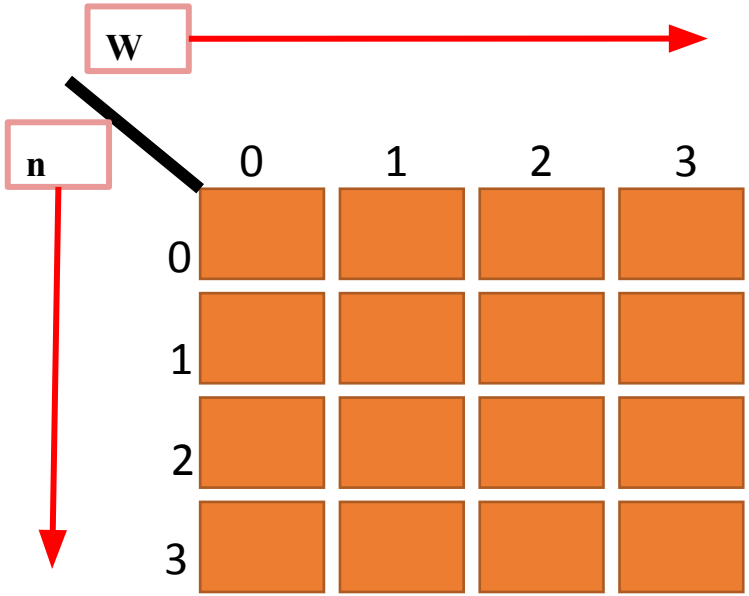
$$c[i,w] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ c[i-1,w] & \text{if } w_i > w \\ \max(v_i + c[i-1, w-w_i], c[i-1,w]) & \text{if } i > 0 \text{ and } w \geq w_i \end{cases}$$

0-1 Knapsack

Step 3: solution of an 0-1 Knapsack (Bottom-Up)

Number of **items, n = 3**
Size of Knapsack, **W=3kg**
Table: c[n+1][W+1]

$$c[i,w] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ c[i-1,w] & \text{if } w_i > w \\ \max(v_i + c[i-1, w-w_i], c[i-1,w]) & \text{if } i > 0 \text{ and } w \geq w_i \end{cases}$$



Item No:	1	2	3
v	5 tk	10 tk	16 tk
weight	2 kg	1 kg	2 kg

0-1 Knapsack

Step 3: solution of an 0-1 Knapsack (Bottom-Up)

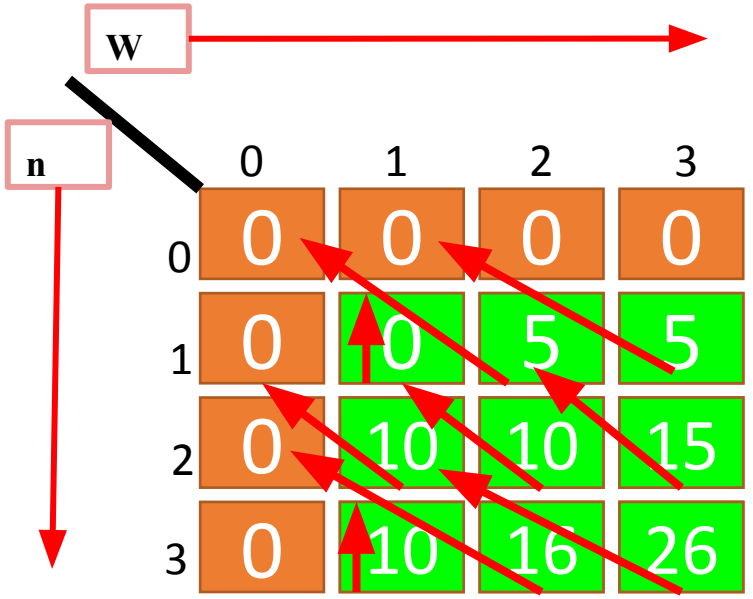
Number of **items, n = 3**
Size of Knapsack, **W=3kg**
Table: c[n+1][W+1]

Item No: 1 2 3

v	5 tk	10 tk	16 tk

weight	2 kg	1 kg	2 kg

$$c[i,w] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ c[i-1,w] & \text{if } w_i > w \\ \max(v_i + c[i-1, w-w_i], c[i-1,w]) & \text{if } i > 0 \text{ and } w \geq w_i \end{cases}$$



So maximum valuable load/
Maximum profit: **26**

0-1 Knapsack

Step 3: Computing the solution of an 0-1 Knapsack (Bottom-Up)

```
DP-KNAPSACK(n, W)
1. for w = 0 to W
2.   c[0][w] = 0
3. for i=0 to n
4.   c[i][0]=0
5. for i = 1 to n
6.   for w = 1 to W
7.       if(weight[i-1]>w)
8.           c[i][w]=c[i-1][w];
9.           b[i][w]='N';
10.      else if((v[i-1]+c[i-1][w-weight[i-1]])>=c[i-1][w])
11.          c[i][w]=v[i-1]+c[i-1][w-weight[i-1]];
12.          b[i][w]='T';
13.      else
14.          c[i][w]=c[i-1][w];
15.          b[i][w]='N';
16. return c[n][W]
```

O(n*W)

*It computes the entries in **row-major order**. (That is, the procedure fills in the first row of c from left to right, then the second row, and so on.) The procedure also maintains the table b=[0..n; 0..W] to help in constructing an optimal solution.*

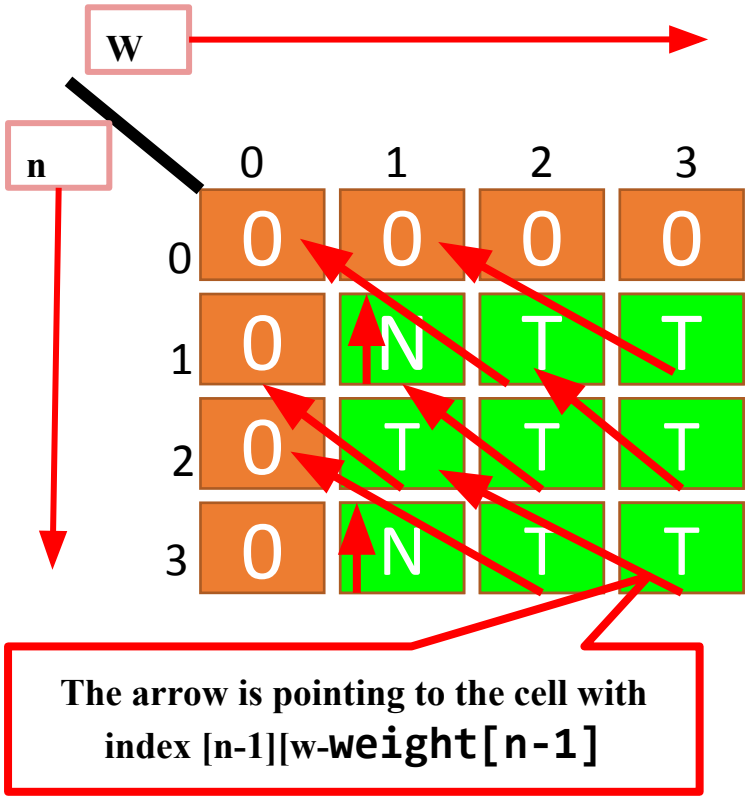
0-1 Knapsack

Step 4: Constructing an Optimal Solution

Using the extra table "b" kept during step 3

```
Print-Items (b, n, w)
1. if n == 0 or w == 0
2.  return
3. if b[n][w]=='T'
4.  Print-Items(b, n-1, w-weight[n-1])
5.  print "item n taken"
6. else
7.  Print-Items(b, n-1, w)
```

b table



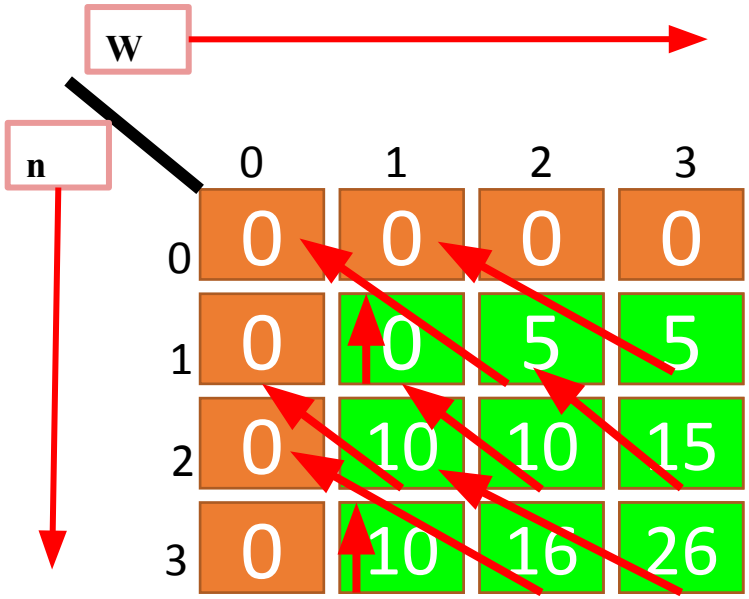
0-1 Knapsack

Step 4: Constructing an Optimal Solution

Another way:
No extra table "b" used.
Using the table "c" used during step 3.

```
Print-Items (c, n, w)
1. while(1)
2.  if n == 0 or w == 0
3.       return
4.  if c[n][w]==c[n-1][w]
5.       n=n-1
6.       else
7.       print "item n taken"
8.       w=w-weight[n-1]
9.       n=n-1
```

c table



0-1 Knapsack

Time Complexity of LCS:

n : Number of items.
W : Knapsack size

Bottom-Up Approach : O(n*W)

Top-Down Approach : O(n*W)

Naive Recursive Solution: O(2^n)

Fractional Knapsack

Fractional Knapsack

The thief can take fractions of items, rather than having to make a binary (0-1) choice for each item.
You can think of an item in the 0-1 knapsack problem as being like a gold bar and an item in the fractional knapsack problem as more like gold dust.

Fractional Knapsack & 0-1 Knapsack

- Both 0-1 and fractional knapsack problems exhibit the optimal substructure property.
- The fractional knapsack problem is solvable by a greedy strategy.
- The 0-1 knapsack problem is not solvable by a greedy strategy.
- The dynamic-programming is needed to find optimal solution for the 0-1 knapsack problem.

Fractional Knapsack

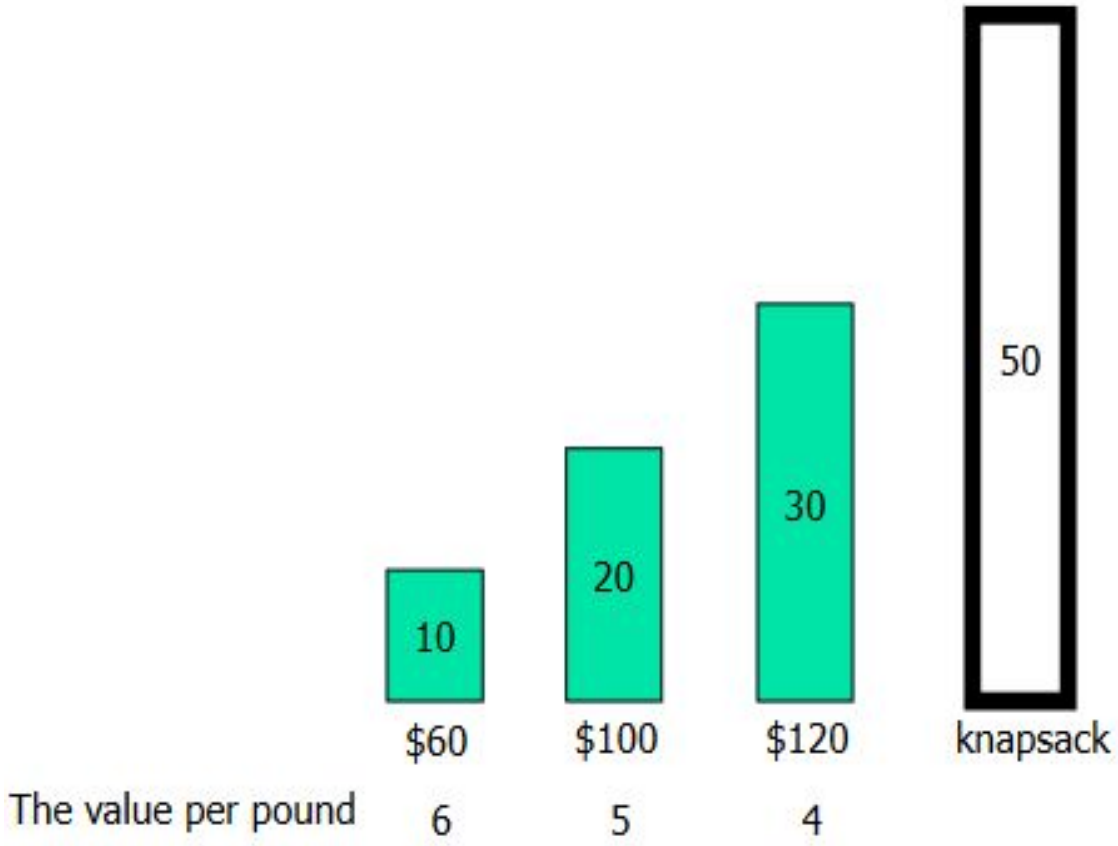
Optimal substructure property of Fractional Knapsack:

If we remove a weight w of one item j from the optimal load, the remaining load must be the most valuable load weighing at most W- w that the thief can take from the n-1 original items plus w_j-w pounds of item j.

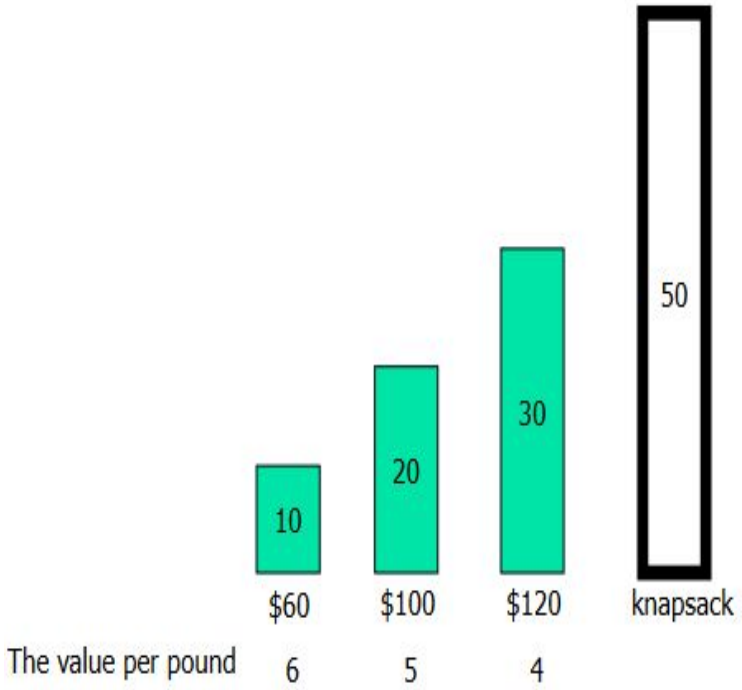
To solve the fractional problem:

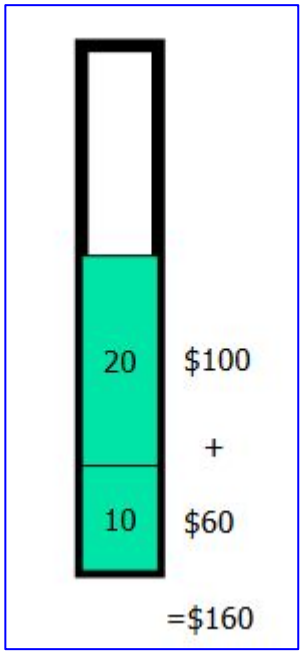
- First compute the value per pound v_i/w_i for each item.
- Obeying a greedy strategy, the thief begins by taking as much as possible of the item with the greatest value per pound.
- If the supply of that item is exhausted and he can still carry more, he takes as much as possible of the item with the next greatest value per pound, and so forth, until he reaches his weight limit W .

Knapsack

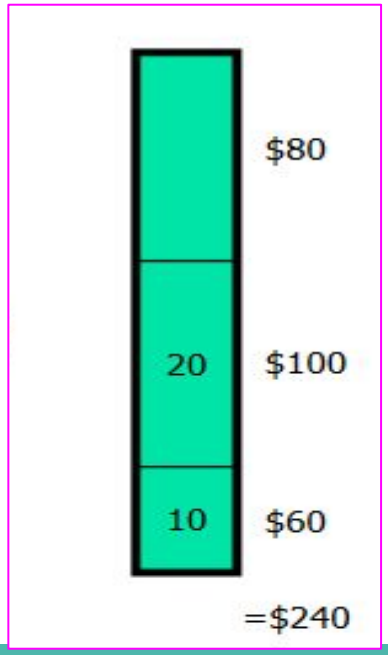


0-1 Knapsack





Fractional Knapsack



Reference

- Introduction to Algorithms (Thomas H. Cormen) 3rd edition
- Slides
- Notes


thank
you